

# CLOUD-NATIVE COMPUTUNG

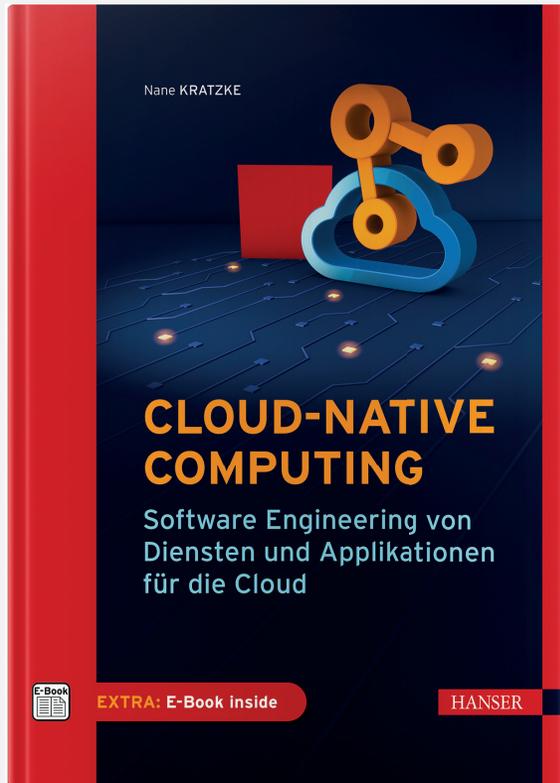
*Unit 04:*

*Standardisierung von  
Deployment Units*

Stand: 27.03.23

# KAPITEL 8

## Standardisierung von Deployment Units (Container)



---

Nane Kratzke

### Cloud-native Computing

Software Engineering von Diensten und Applikationen  
für die Cloud

284 Seiten. E-Book inside

€ 59,99. ISBN 978-3-446-46228-1

Weitere Informationen unter: [www.hanser-fachbuch.de](http://www.hanser-fachbuch.de)

HANSER

## 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.

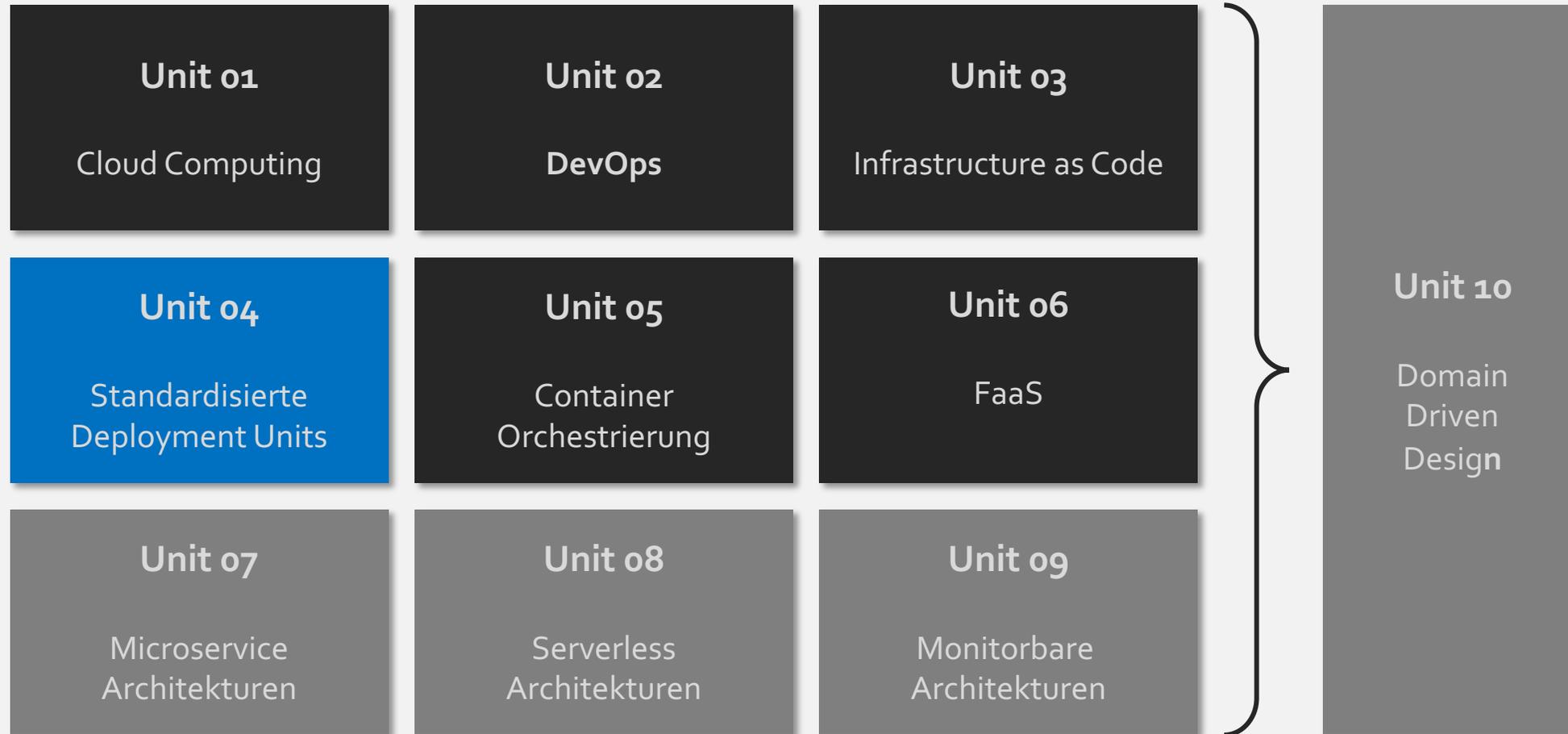
Die CCo Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.



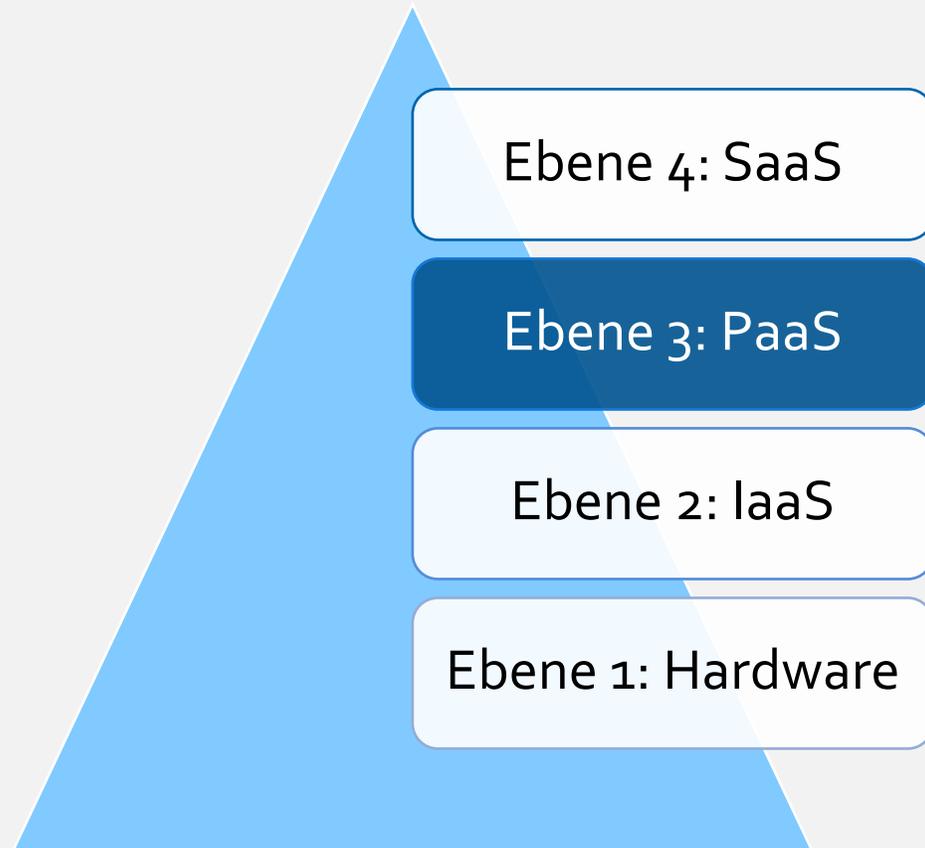
# INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



# DAS SCHICHTENMODELL DES CLOUD COMPUTINGS

Wo sind wir?



## Kunden, Endnutzer

- Anpassbare Software-Dienste
- XaaS (Everything as a Service)
- Transparente Updates

## Entwickler

- **Programmierschnittstellen (APIs)**
- **Plattformdienste**
- **Abstraktion der technischen Infrastruktur**

## Administratoren

- Elastizität
- Virtuelle Ressourcenpools
- Technische Infrastruktur (VM, Storage, Network)

## Rechenzentrum

- Rechner
- Netzwerk
- Storage

# INHALTE

## Hintergrund

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

## Betriebssystem-Virtualisierung

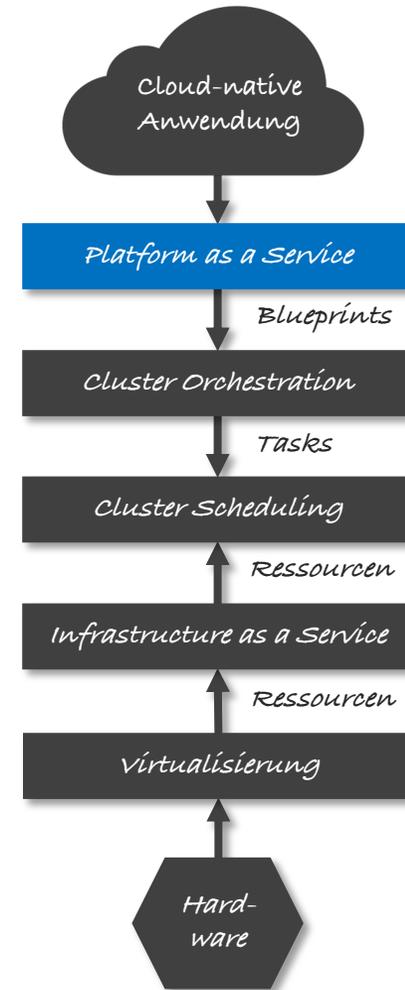
- OS-Virtualisierung
- Linux-basierte Techniken zur OS-Virtualisierung
- Standardisierung von Deployment-Einheiten => Container

## Automationsumgebungen für Container

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

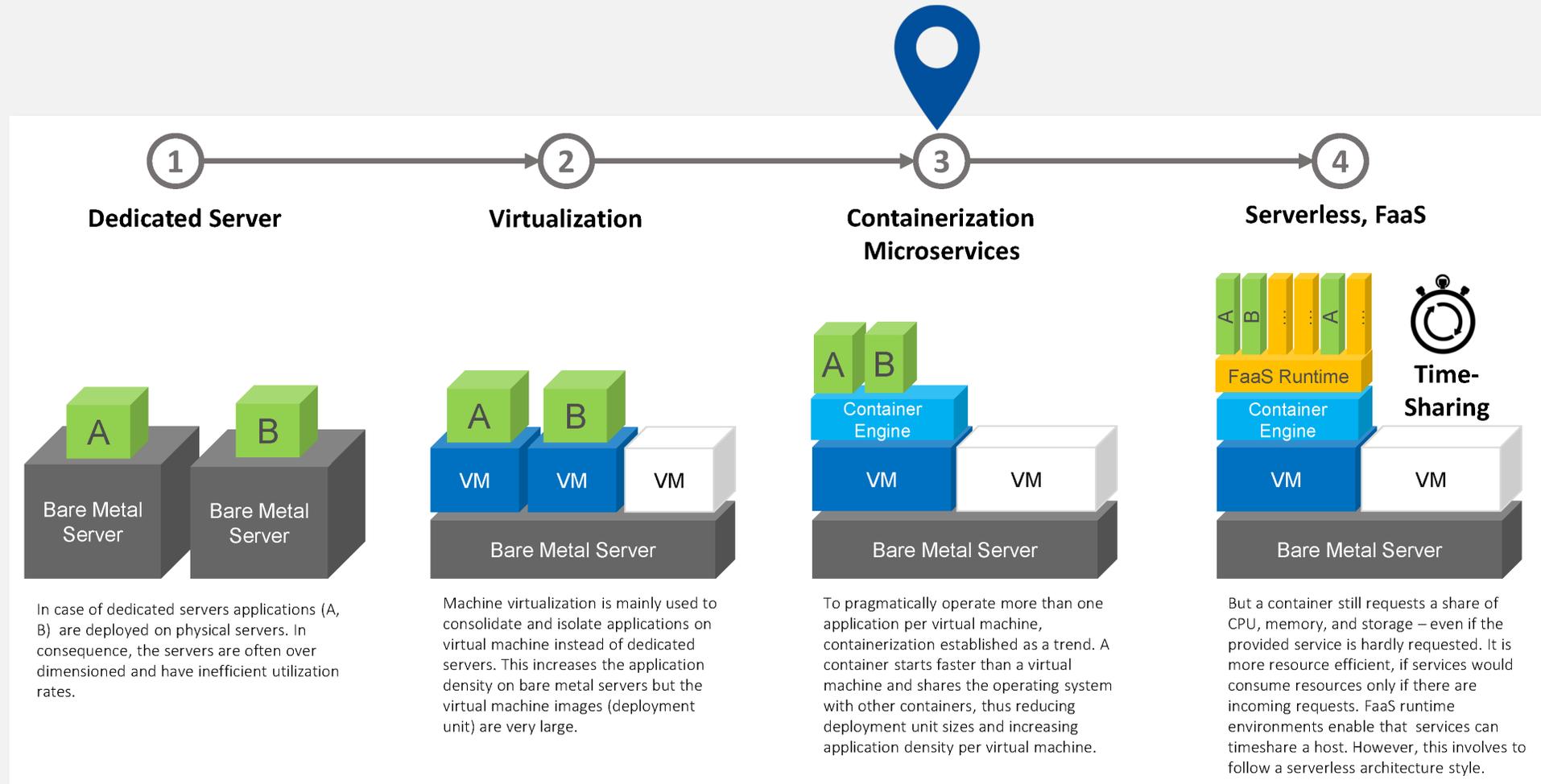
## Container-Pattern

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



# EINE KURZE GESCHICHTE DER CLOUD

Wo sind wir?



# ENTWICKLUNG

## *Von Platform as a Service zu Container as a Service*

- Mittels IaaS und Provisionierungsansätzen lassen sich vordefinierte Deployment-Einheiten (VM-Images) erzeugen und auf Hypervisoren als Virtuelle Maschinen ausbringen.
- Das Problem ist, diese Deployment-Einheiten (VM-Images) sind im allgemeinen sehr groß (>> 100MB bis zu mehreren GB) und die Start-Up-Zeiten dieser Maschinen sind recht lang (bis zu mehreren Minuten).
- Ferner wollen sich Entwickler häufig nicht mit den Details der Infrastruktur „herumschlagen“ sondern einfach nur ihren Code ausbringen und laufen lassen.
- Aus dieser Motivationslage sind sogenannte Platform-as-a-Service PaaS-Angebote entstanden.

# PAAS

*Was ist das?*

Als **Platform as a Service (PaaS)** bezeichnet man eine Dienstleistung, die als Cloud Service eine Plattform für Entwickler von Anwendungen zur Verfügung stellt. Dabei kann es sich sowohl um Laufzeitumgebungen (häufig für Webanwendungen), aber auch um Entwicklungsumgebungen handeln, die mit geringem administrativem Aufwand und ohne Anschaffung der darunterliegenden Hardware und Software genutzt werden können. Sie unterstützen wesentliche Teile des Software-Lebenszyklus von der Entwicklung, den Test, die Auslieferung bis hin zum Betrieb der Anwendungen über das Internet.

**Cloud Anwendungen**  
Software as a Service (SaaS)

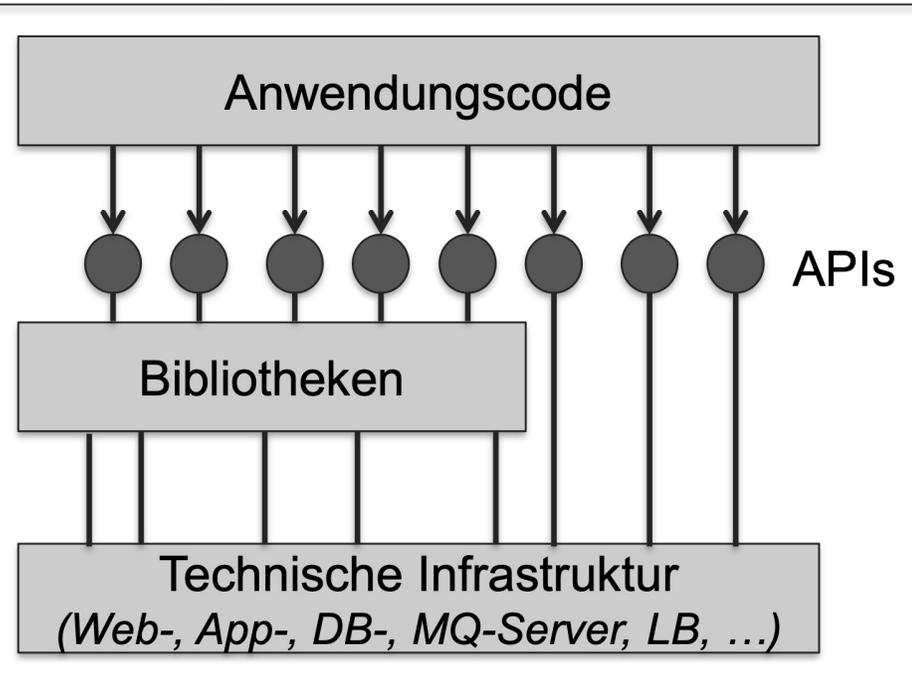
**Cloud Plattformen**  
Platform as a Service (PaaS)

**Cloud Infrastruktur**  
Infrastructure as a Service (IaaS)

# DAS PROBLEM: STOVEPIPE ARCHITEKTUREN

Anwendungen aufwändig „von Hand“ verdrahtet

Das System: Mühevoll verdrahtet.

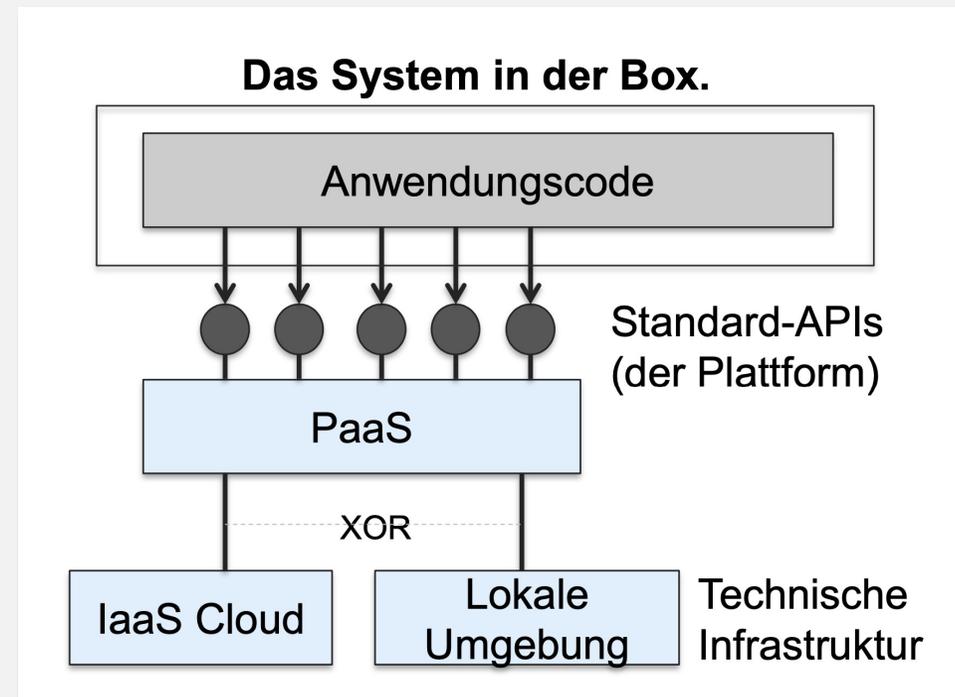


Mund-  
geblasenes mag  
ja schön sein,  
aber es ist teuer  
und vor allem  
kaum 1:1  
wiederholbar!

# DIE LÖSUNG: PLATFORM AS A SERVICE

*PaaS bietet eine ad-hoc Entwicklungs- und Betriebsplattform*

- Die Anwendung wird per Applikationspaket oder als Quellcode deployed. Es ist kein Image mit technischer Infrastruktur notwendig.
- Die Anwendung sieht nur Programmier- oder Zugriffsschnittstellen seiner Laufzeitumgebung.
- Es erfolgt eine automatische Skalierung der Anwendung.
- Entwicklungswerkzeuge (IDEs, Build-Systeme, Testumgebung) stehen zur Verfügung („Deploy to cloud“).
- Die Plattform bietet eine Schnittstelle zur Administration und zum Monitoring.



# PAAS

## Definitionen

### NIST Definition of Cloud Computing

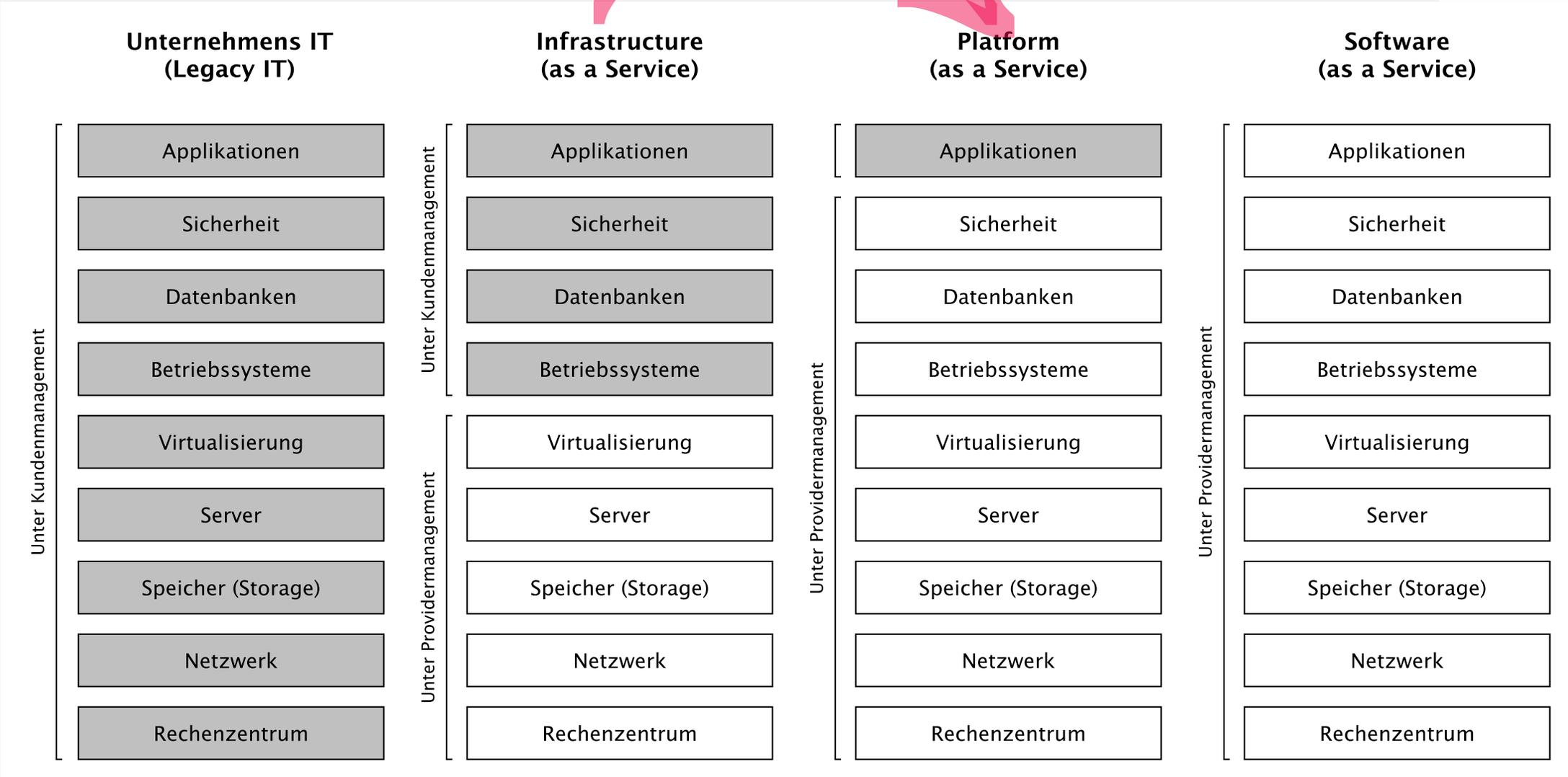
The capability provided to the consumer is to **deploy onto the cloud infrastructure** consumer-created or acquired applications created using programming **languages, libraries, services, and tools supported by the provider**. The **consumer does not manage or control the underlying cloud infrastructure** including network, servers, operating systems, or storage, but has **control over the deployed applications** and possibly configuration settings for the application-hosting environment.

### Forrester

A **complete application platform** for **multitenant** cloud environments that includes **development tools, runtime, and administration** and management tools.

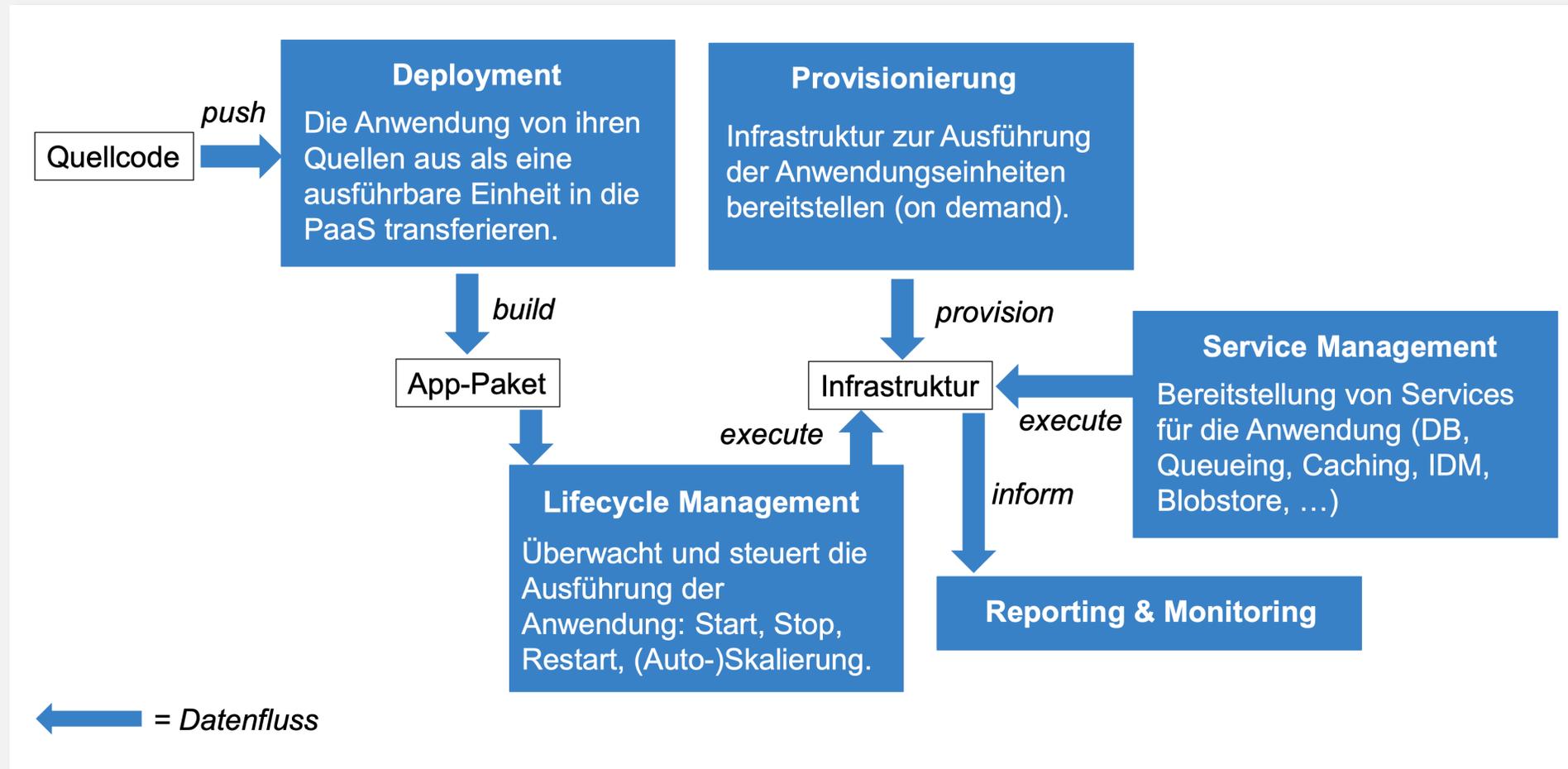
PaaS combines an **application platform with managed cloud infrastructure** services.

# IAAS VS. PAAS



# PAAS CLOUD

Die funktionalen Bausteine einer PaaS Cloud

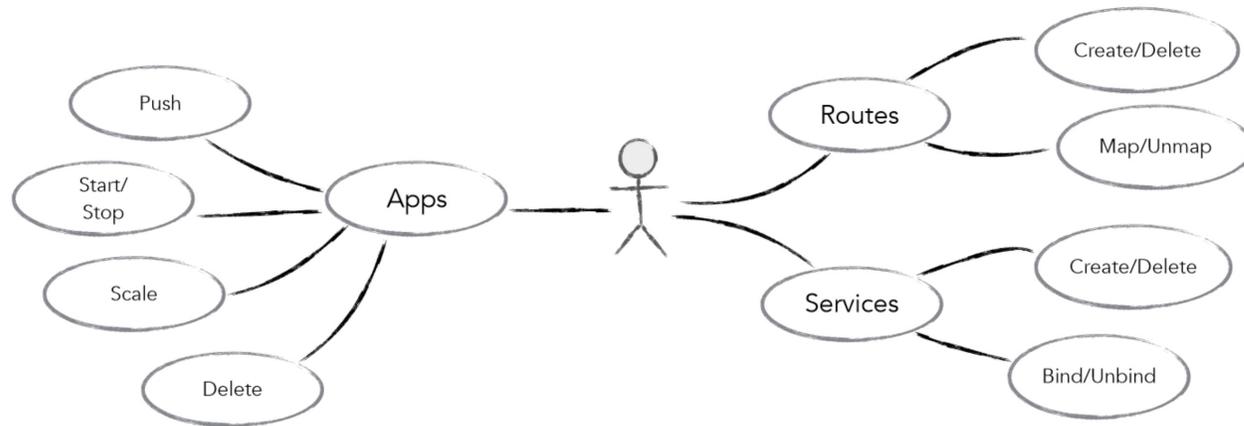


# DER PAAS EFFEKT

*PaaS verbergen Komplexität vor dem Entwickler*

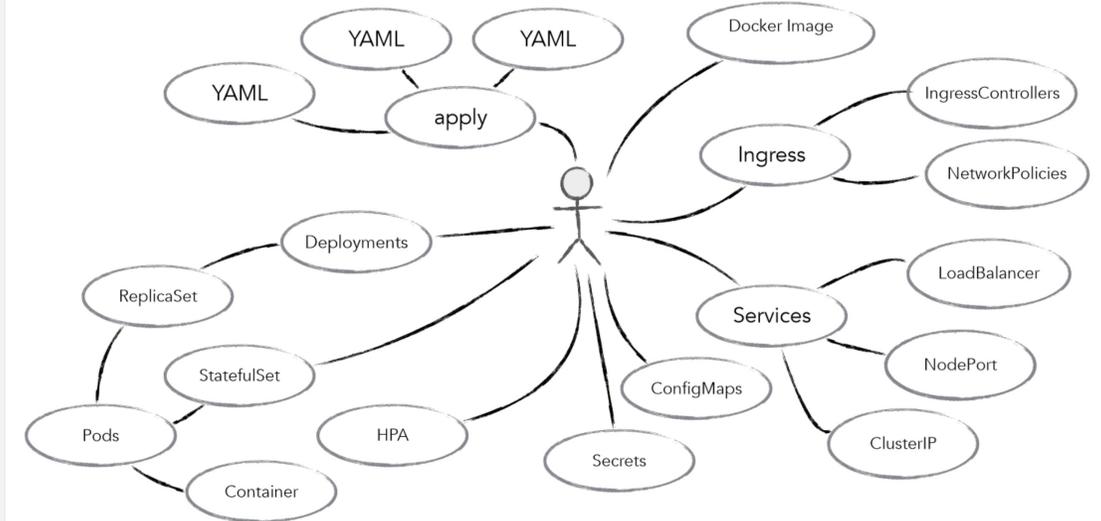
## Minimal Concepts

*Cloud Foundry*



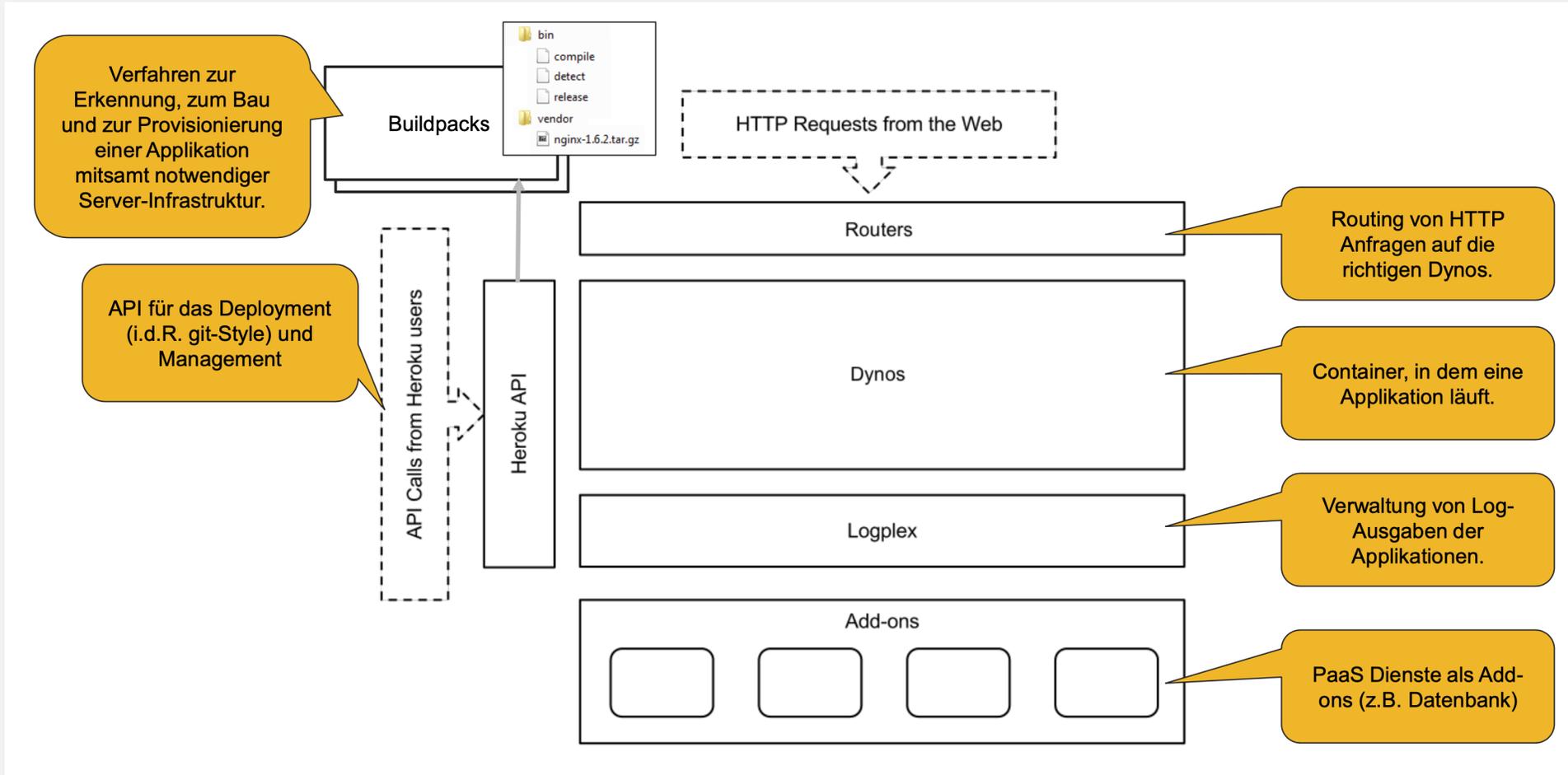
## Minimal Concepts

*Kubernetes*



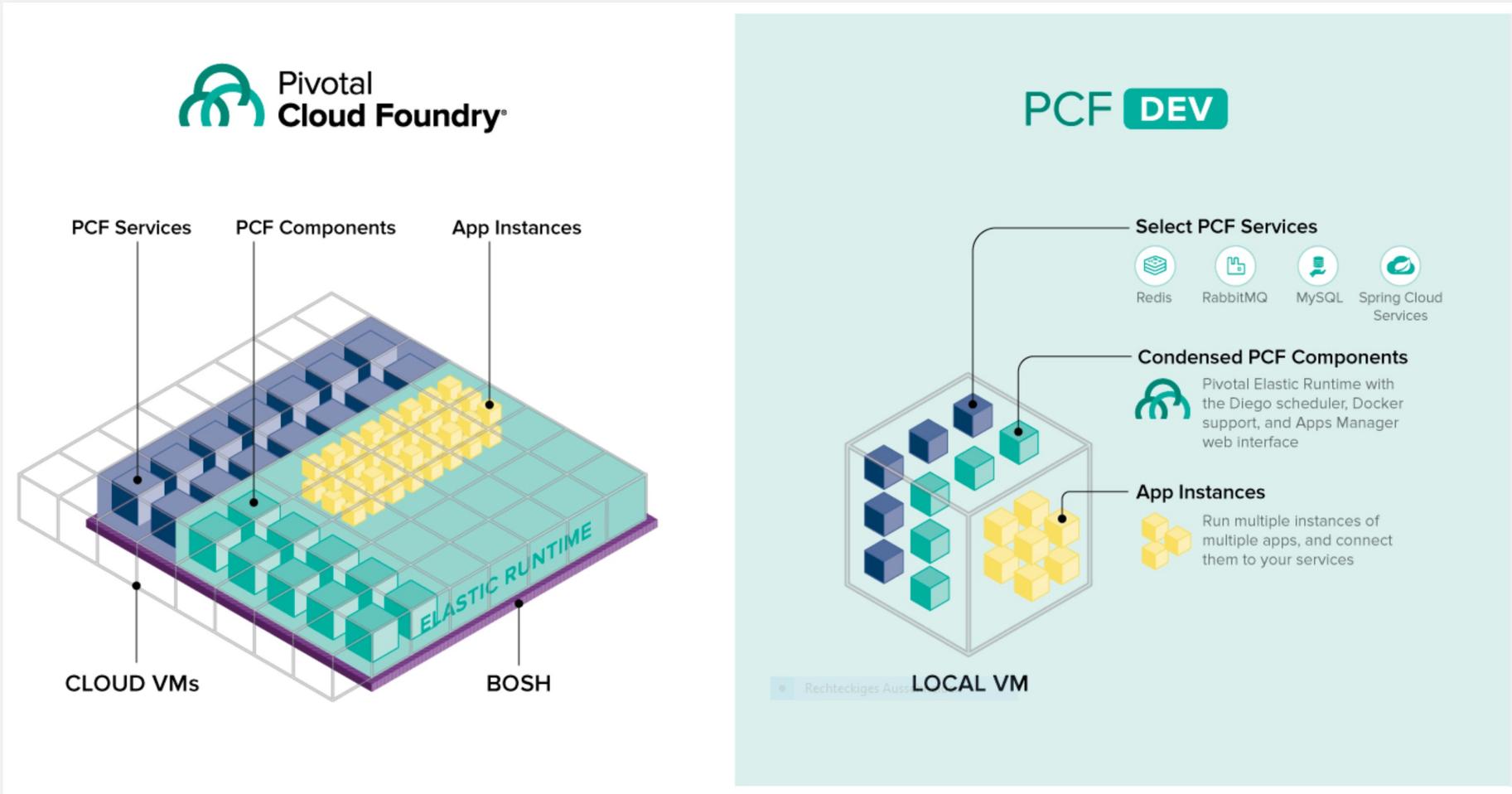
# PAAS HIGH-LEVEL ARCHITEKTUR

Am Beispiel von Heroku



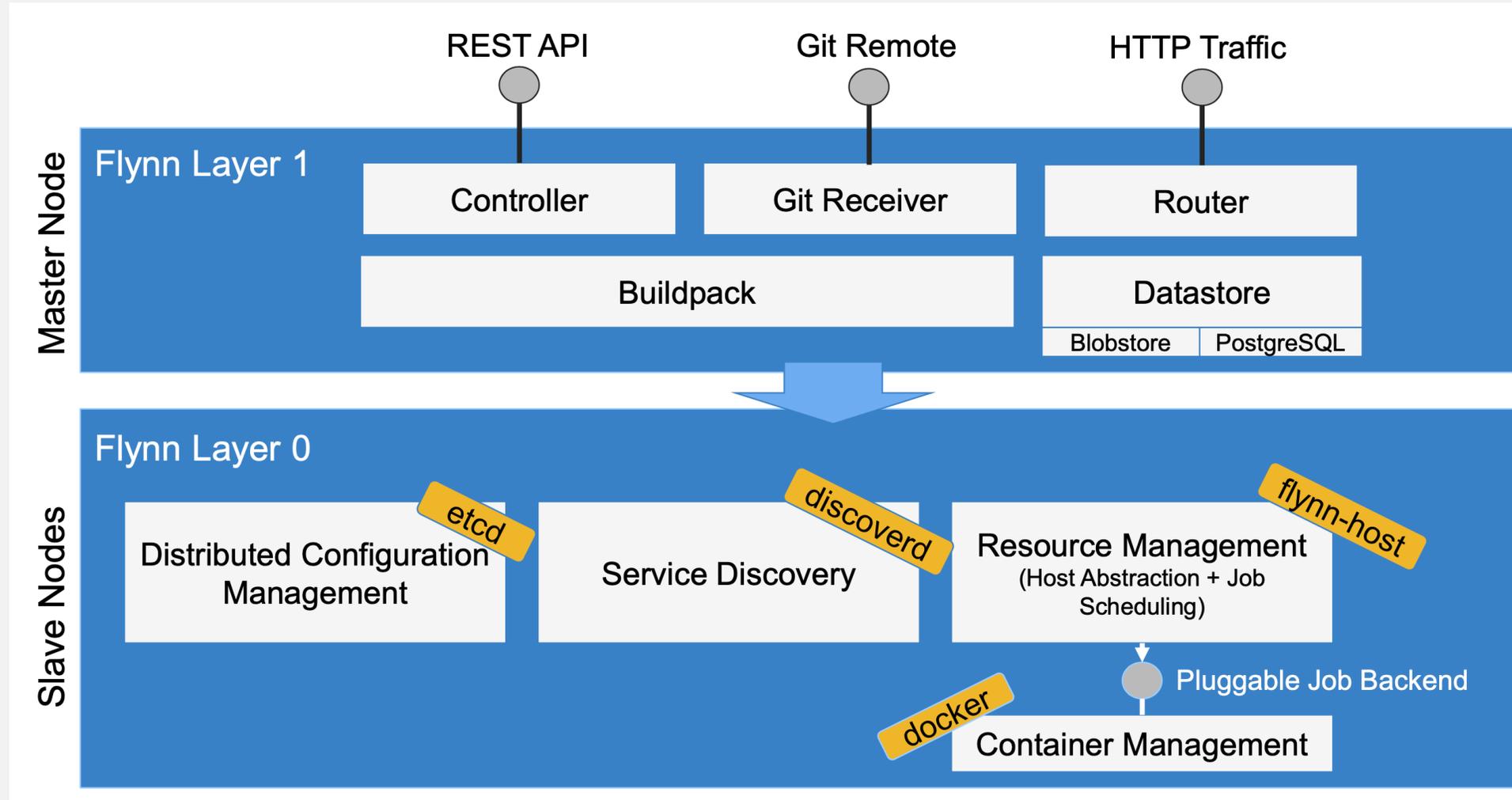
# PRIVATE PAAS CLOUDS

Beispiel: Cloud Foundry



# PRIVATE PAAS CLOUDS

Beispiel: Flynn



# PRIVATE PAAS CLOUDS

## Alternative Private PaaS Clouds

Name	Link	Anmerkung
DEIS	<a href="https://deis.io">https://deis.io</a>	Vglb. Zu Flynn (Docker-basiert)
OpenShift	<a href="https://openshift.com">https://openshift.com</a>	PaaS mit Schwerpunkt JEE von RedHat
CloudFoundry	<a href="https://cloudfoundry.org">https://cloudfoundry.org</a>	PaaS von Pivotal mit breiter Unterstützung aus der Industrie
Stackato	<a href="https://activestate.com/stackato">https://activestate.com/stackato</a>	Private PaaS (kommerziell)
PaaS-TO	<a href="https://github.com/yelp/paasta">https://github.com/yelp/paasta</a>	Open-Source private PaaS auf Basis von Mesos+Marathon
VAMP	<a href="https://vamp.io">https://vamp.io</a>	Leichtgewichtiges Open-Source PaaS ausgelegt auf Microservices (Kuberntes oder Mesos)
Apollo	<a href="https://github.com/capgemini/apollo">https://github.com/capgemini/apollo</a>	Open-Source private PaaS auf Basis von Mesos von Capgemini
Mantl	<a href="https://mantl.io">https://mantl.io</a>	Open-Source private PaaS auf Basis von Mesos von Cisco

# PUBLIC PAAS CLOUDS

*Am Beispiel der Google App Engine*

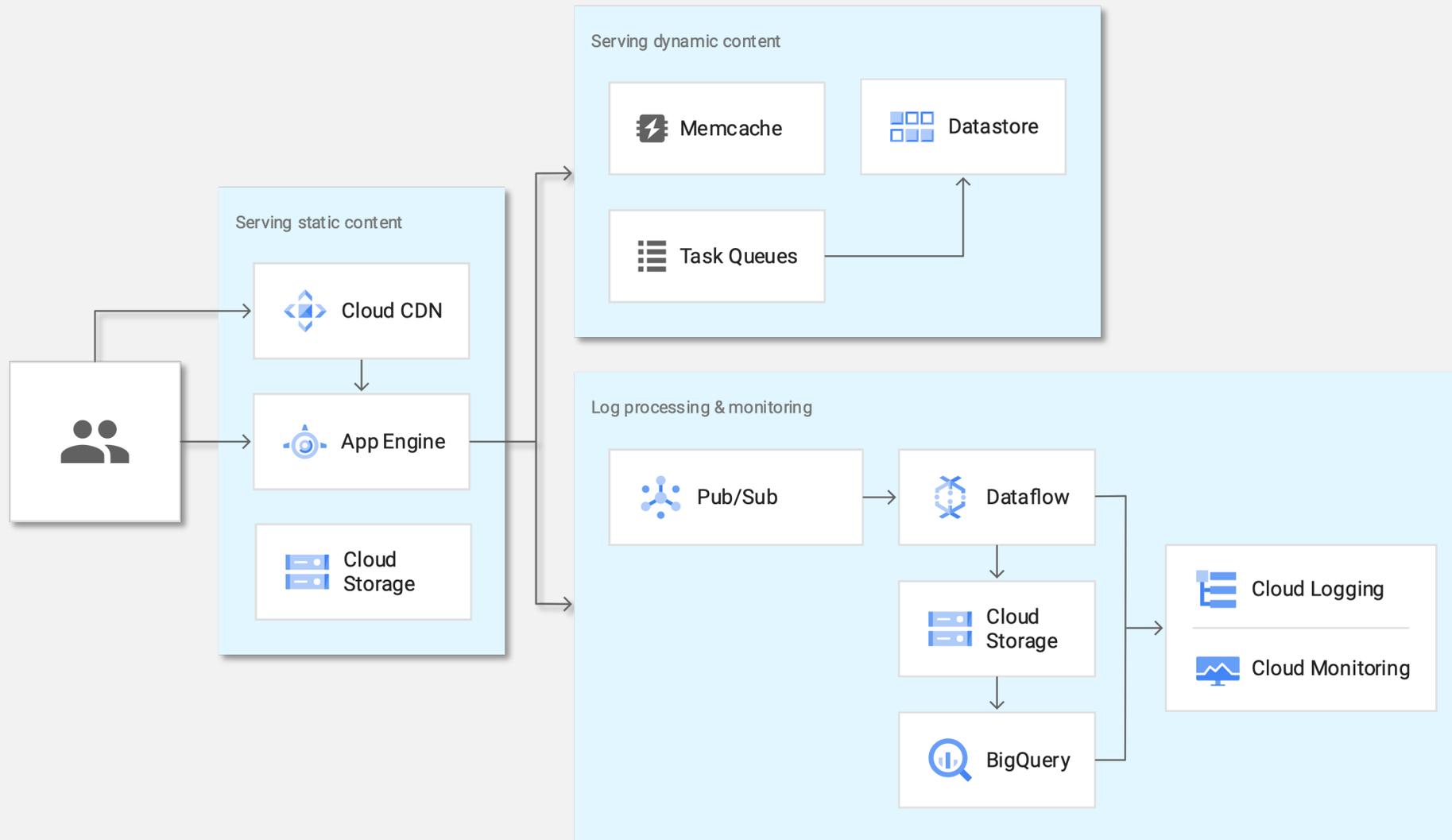
- Die Google App Engine (GAE) ist das PaaS Angebot von Google.
- Anwendungen laufen innerhalb der Google Infrastruktur.
- Der Betrieb der Anwendungen ist innerhalb bestimmter Quoten kostenfrei.
- Unterstützte Sprachen: u.a. JS, Ruby, Go .NET, Java, Python, PHP
- Integrationen in alle gängigen IDEs stehen zur Verfügung



App Engine

# GOOGLE APP ENGINE

## Referenzarchitektur



# GOOGLE APP ENGINE

## *Sandbox-bedingte Einschränkungen*

Eine GAE-Applikation läuft in einer Sandbox, die das Verhalten der Applikation mit dem Ziel einschränkt, die Verarbeitung effizient zu halten und die Infrastruktur im Auto-Scaling zu schützen.

Es dürfen daher nicht alle Klassen von Standardbibliotheken genutzt werden:

- Keine eigenen Threads öffnen
- Kein Zugriff auf die Laufzeitumgebung (z.B. Classloader)

Kommunikation mit anderen Web-Anwendungen nur über URL Fetch, XMPP oder Email

- Anfragen und Antworten dürfen maximal 1MB groß sein
- Web-Hooks als allgemeines Architekturmittel für eingehende Kommunikation. Angestoßen bei Ereignissen (Warmup, Messages, Cron-Jobs)

Alle Requests an eine GAE Anwendung werden nach 60 Sekunden beendet. Es gibt weitere diverse Einschränkungen zu Datenvolumina und Anzahl von Service Aufrufen.



*Einschränkungen dieser Art finden  
Se natürlich auch  
bei anderen  
Anbietern.*

# PAAS

## Das PaaS - Problem

Allein zwischen Oktober 2009 und Oktober 2010 sollen mehr als 100 PaaS-Anbieter den Markt betreten haben.

Alle mit dem Ziel Kunden möglichst viele administrative Aufgaben abzunehmen, **Skalierbarkeit** und **Hochverfügbarkeit** zu ermöglichen, Fixkosten und **Gesamtkosten zu senken**, die Anwender **flexibler** zu machen und um eine schnelle Anwendungsentwicklung und einen baldigen Markteintritt (**Time-to-Market**) zu ermöglichen.

Name	Status	Runtimes	Scaling	Hosting	Infrastructures	
Acquia Cloud	Production	php	↑ ↔	👁	AS EU NA OC	Details
Amazon Elastic Beanstalk	Production	dotnet go java node php python ruby	↑ ↔ ↻	👁	AS EU NA OC SA	Details
Any9ines	Production	groovy java node ruby scala extensible	↑ ↔	👁	EU	Details
App42 PaaS	Production	groovy java node php python ruby	↑ ↔	👁	NA	Details
AppAgile	Production	java node perl php python ruby extensible	↑ ↔ ↻	👁 🖱	EU	Details
AppFog	Production	java node php python ruby extensible	↑ ↔	👁 🔒	NA	Details
AppHarbor	Production	dotnet	↑ ↔	👁	EU NA	Details
Apprenda	Production	docker dotnet java extensible	↑ ↔ ↻	🔒		Details
AppScale	Production	go java php python	↔ ↻	🔒		Details
APPUIO	Production	go java node perl php python ruby scala extensible	↑ ↔ ↻	👁 🖱 🔒	EU	Details
Atos Cloud Foundry	Production	clojure dotnet go groovy hhvm java node php python ruby scala swift extensible	↑ ↔ ↻	👁 🔒	AS EU NA OC SA	Details
Backery	Production	go node php python ruby	↑ ↔	👁	EU	Details
BitNami	Production	java node php python ruby	↑	👁	AS EU NA OC SA	Details
Bluemix	Production	go java node php python ruby extensible	↑ ↔ ↻	👁 🔒	EU NA OC	Details

Quelle: <https://paasfinder.org>

# DAS PAAS-PROBLEM

*„In recent years, the cloud hype has led to a multitude of different offerings across the entire cloud market, from Infrastructure as a Service (IaaS) to Platform as a Service (PaaS) to Software as a Service (SaaS).*

*Despite the high popularity, there are still several problems and deficiencies.*

***Especially for PaaS, the heterogeneous provider landscape is an obstacle for the assessment and feasibility of application portability.“***

Kolb, Stefan. (2018). On the Portability of Applications in Platform as a Service. 10.20378/irbo-54102.



# DAS PAAS-PROBLEM

Am Beispiel zweier Java Getting-Started-Tutorials von Heroku und Google App Engine



heroku

```
import com.heroku.api;
// Further imports skipped for clarity

@Controller
@SpringBootApplication
public class HerokuApplication {

    @Value("${spring.datasource.url}")
    private String dbUrl;

    @Autowired
    private DataSource dataSource;

    public static void main(String[] args) throws Exception {
        SpringApplication.run(HerokuApplication.class, args);
    }

    // Further methods skipped for clarity
}
```

git clone https://github.com/heroku/  
gradle-gettingstarted.git



App Engine

```
import com.google.appengine.api.utils.SystemProperty;
// Further imports skipped for clarity

@WebServlet(name = "HelloAppEngine", value = "/hello")
public class HelloAppEngine extends HttpServlet {

    @Override
    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response
    ) throws IOException {
        Properties properties = System.getProperties();
        response.setContentType("text/plain");
        response.getWriter().println("Hello App Engine");
    }

    // Further methods skipped for clarity
}
```

git clone https://github.com/GoogleCloudPlatform/  
getting-started-java.git

Wesentliche Probleme  
bei PaaS sind u.a.  
folgende fehlende  
Standards:

- **Deployment-Format** der zu hostenden Anwendungen
- **PaaS-Runtime-Schnittstelle**

**Challenge:** Versuchen Sie mal das Heroku Getting-Started auf Google App Engine zum Laufen zu bringen (und anders herum)

# PAAS

## Erkennbarer Trend in PaaS Plattformen



Name	Status	Runtimes	Scaling	Hosting	Infrastructures	
Acquia Cloud	Production	php	↕	👁	AS EU NA OC	Details
Amazon Elastic Beanstalk	Production	dotnet go java node php python ruby	↕ ↻	👁	AS EU NA OC SA	Details
Anyines	Production	groovy java node ruby scala <b>extensible</b>	↕	👁	EU	Details
App42 PaaS	Production	groovy java node php python ruby	↕	👁	NA	Details
AppAgile	Production	java node perl php python ruby <b>extensible</b>	↕ ↻	👁 🗑	EU	Details
AppFog	Production	java node php python ruby <b>extensible</b>	↕	👁 🔒	NA	Details
AppHarbor	Production	dotnet	↕	👁	EU NA	Details
Apprenda	Production	<b>docker</b> dotnet java <b>extensible</b>	↕ ↻	🔒		Details
AppScale	Production	go java php python	↔ ↻	🔒		Details
APPUIO	Production	go java node perl php python ruby scala <b>extensible</b>	↕ ↻	👁 🗑 🔒	EU	Details
Atos Cloud Foundry	Production	clojure dotnet groovy hhvm java node php python ruby scala <b>extensible</b>	↕ ↻	👁 🔒	AS EU NA OC SA	Details
Backery	Production	go node php python ruby	↕	👁	EU	Details
BitNami	Production	java node php python ruby	↕	👁	AS EU NA OC SA	Details
Bluemix	Production	go java node php python ruby <b>extensible</b>	↕ ↻	👁 🔒	EU NA OC	Details

*Laufzeitumgebungen können durch den Kunden erweitert werden.*

*Wie funktioniert das technisch?*

Quelle: <https://paasfinder.org>

## 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

## Automationsumgebungen für Container

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

## Container-Pattern

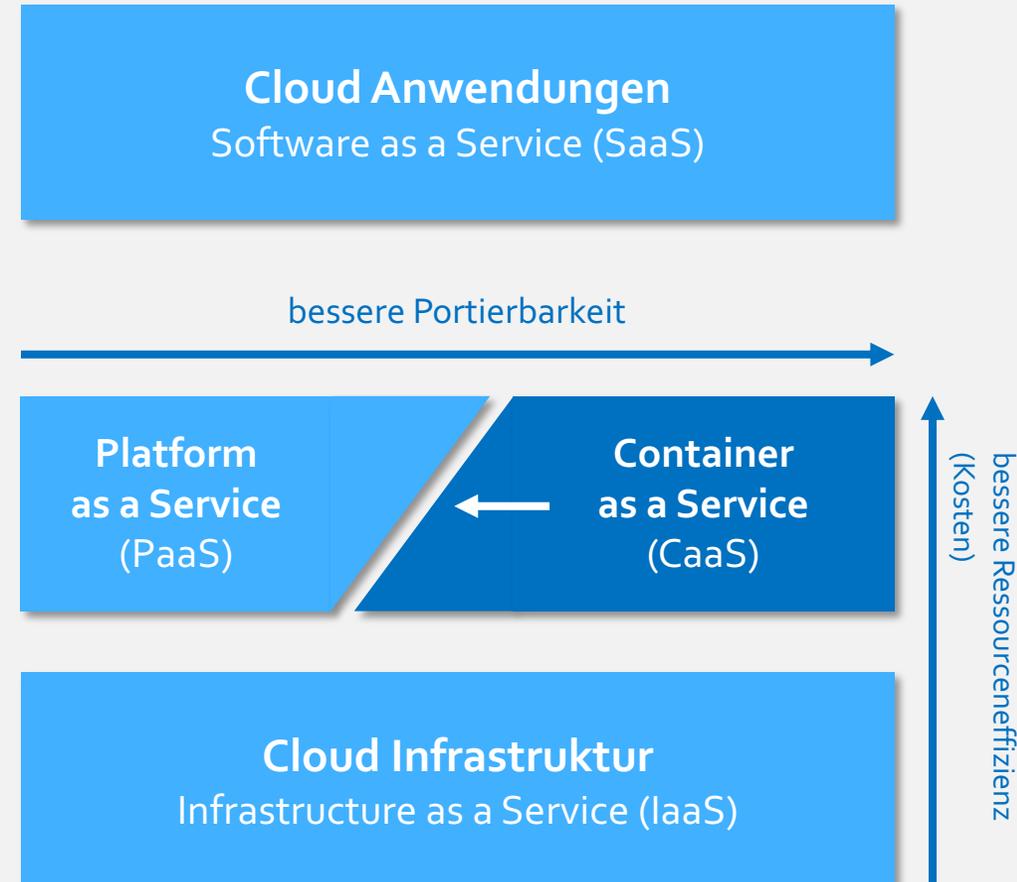
- Container (Anti-)Pattern
- 12-Factor Apps

# VON PAAS ZU CAAS

## Container as a Service

Bei CaaS handelt sich um ein Cloud-Computing-Modell, mit dem sich Container-basierte Virtualisierung als Service aus der Cloud nutzen lässt. Anwender von CaaS benötigen keine eigene Infrastruktur für die Container-Virtualisierung und müssen keine eigene Container-Plattform betreiben und managen. Sämtliche Ressourcen für die Virtualisierung wie Rechenleistung, Speicherplatz, Container-Engine und Orchestrierungssoftware stellt der Cloud-Service-Provider zur Verfügung. Es ist möglich, CaaS als Service aus einer Public Cloud oder aus einer Private Cloud zu beziehen.

Container as a Service liegt konzeptionell zwar zwischen den Modellen Infrastructure as a Service (IaaS) und Platform as a Service (PaaS) verdrängt aber im Cloud-native Umfeld allmählich aufgrund besserer Portierbarkeit und besserer Standardisierung das PaaS Modell.



# CONTAINER

*Die grundsätzliche Idee*

Anstatt „mundgeblasener“  
Speziallösungen



Standardisiert ausbringbare und ausführbare  
Einheiten



*Bei einem  
Container ist egal  
was er beinhaltet.*

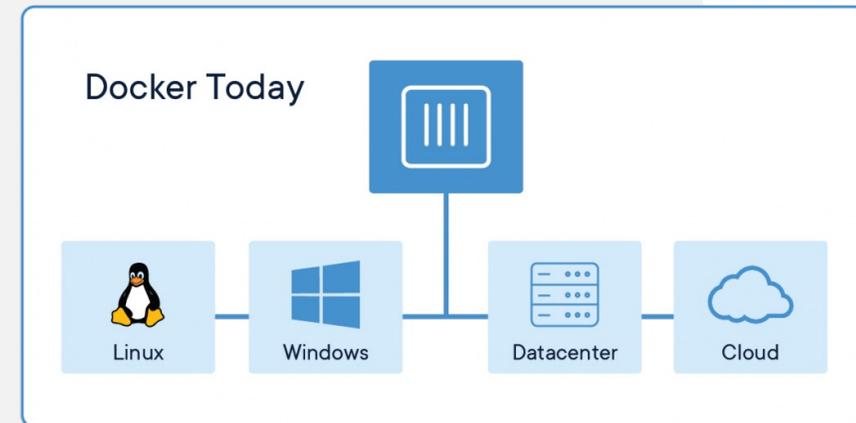
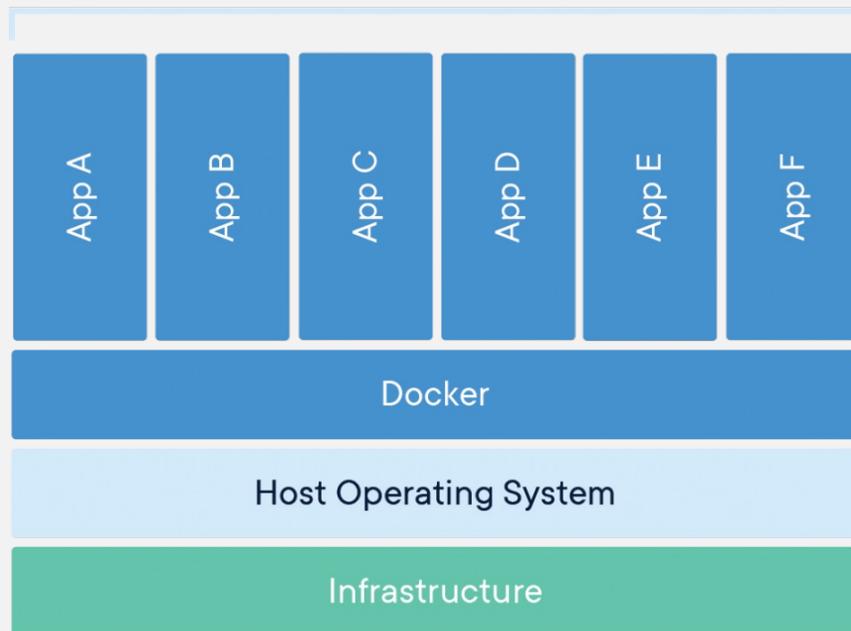
*Er kann  
standardisiert auf  
Schiffen, Zügen  
oder Lastern  
transportiert und  
verladen werden.*

- **Container** = Verpackung für Ops Components
- **Standard-Schnittstellen** für Standard-Betriebsprozeduren
- **Einfach zu transportieren**
- **Schnell zu starten**
- **Wenig Performance-Overhead**

# WHAT IS A CONTAINER?

A standardized unit of software

Containerized Applications



# VIRTUALISIERUNGSARTEN

Wiederholung

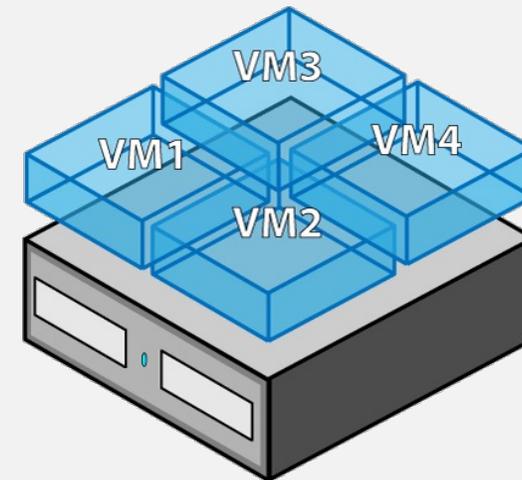
Unter Virtualisierung werden grundsätzlich verschiedene Konzepte und Technologien verstanden.

## Virtualisierung von Hardware-Infrastruktur

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

## Virtualisierung von Software-Infrastruktur

- Betriebssystem-Virtualisierung (Containerization)
- Anwendungs-Virtualisierung (Runtime, z.B. die Java Virtual Machine)

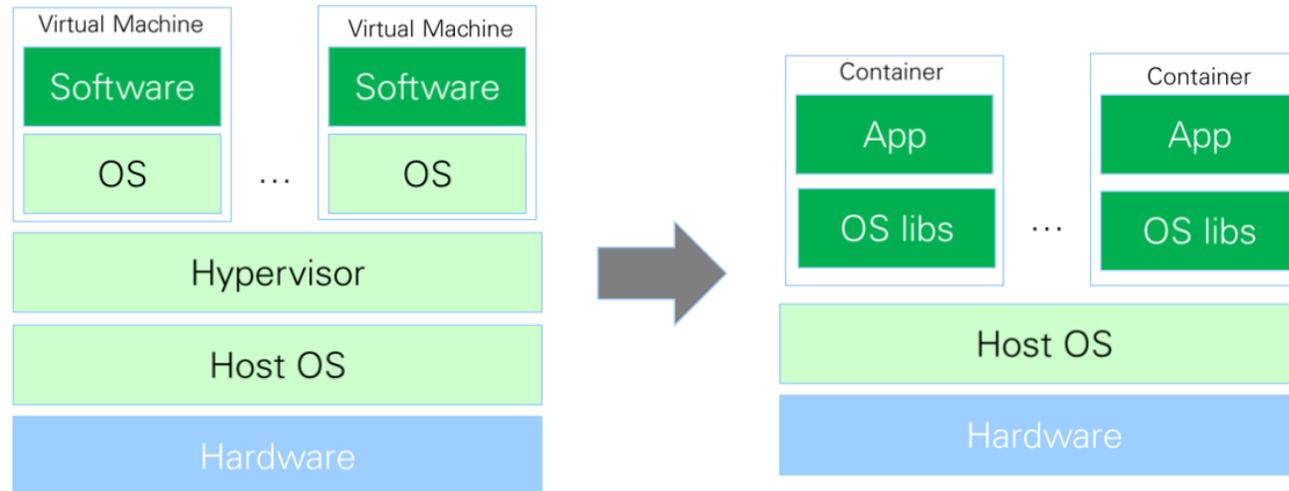


# BETRIEBSSYSTEM-VIRTUALISIERUNG

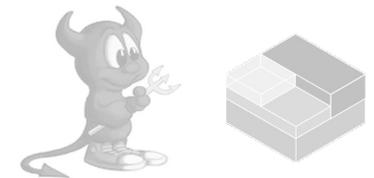
## Wiederholung

Es gibt keinen Hypervisor. Jede Applikation läuft direkt als Prozess im Host-Betriebssystem.

Dieser Prozess ist jedoch mittels OS-Mechanismen isoliert.



Leistungsverlust:  
CPU-/RAM-Overhead in der Regel nicht messbar (~0%)



OpenVZ

SOLARIS

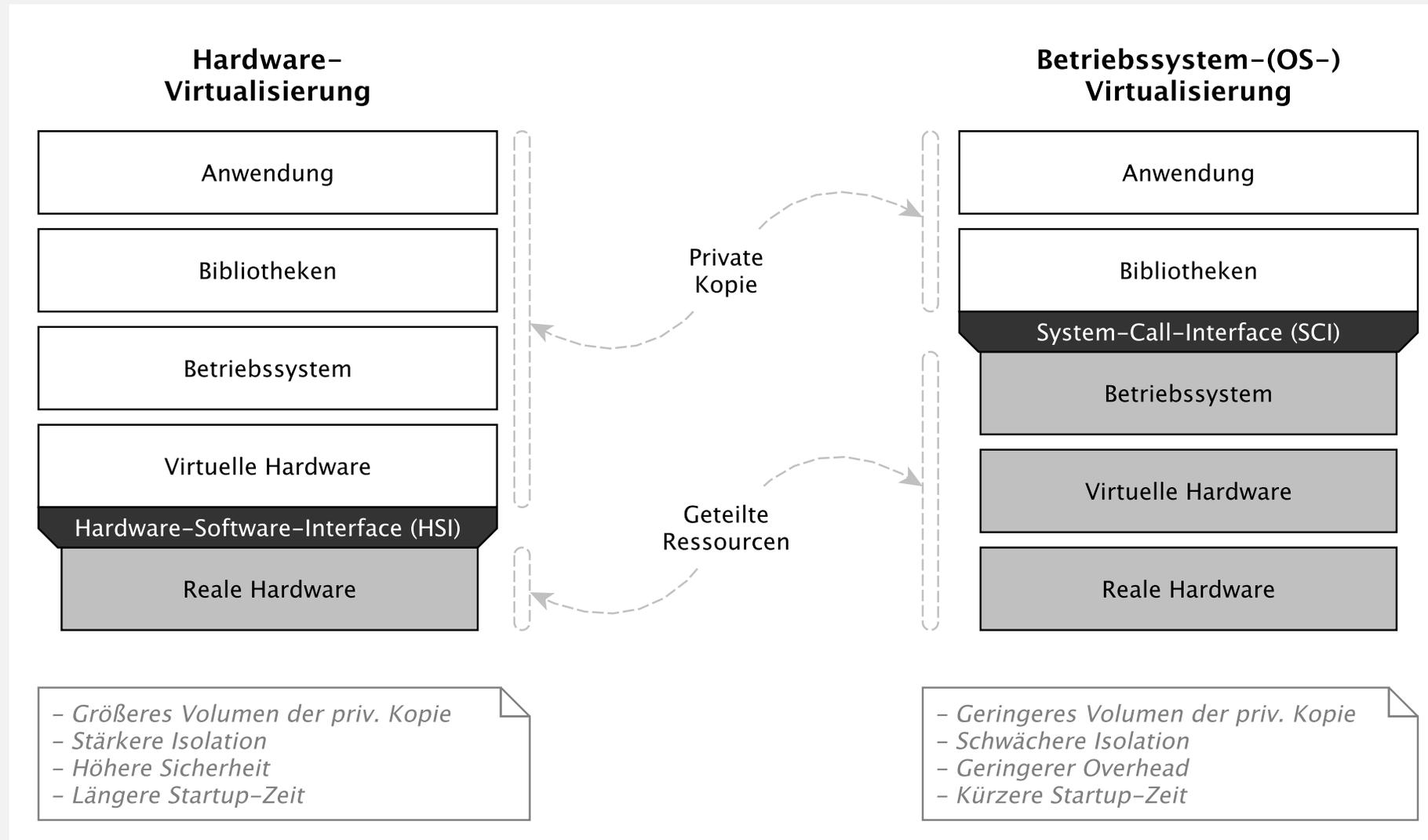
- Free BSD Jails (2000)
- Solaris Zones (2005)
- Linux OpenVZ (2005)
- Linux LXC (2008)
- und mehr ...

- Isolation des Prozesses durch Namespaces (bzgl. CPU, RAM, Disk I/O) und Containments
- Isoliertes Dateisystem
- Eigene Netzwerkschnittstelle
- Startup-Zeit = Startdauer für den ersten Prozess (kein Boot des OS erforderlich!)

Docker v0.0.1  
release: 2013

# HARDWARE- VS. OS-VIRTUALISIERUNG

Wiederholung



# OS-VIRTUALISIERUNG

*Aus dem Blickwinkel des Linux-Kernels*

## Control Groups

Gruppieren von Prozessen (z.B. alle Prozesse einer App, eines Dienstes, etc.) in Gruppen

## Kernel Namespaces

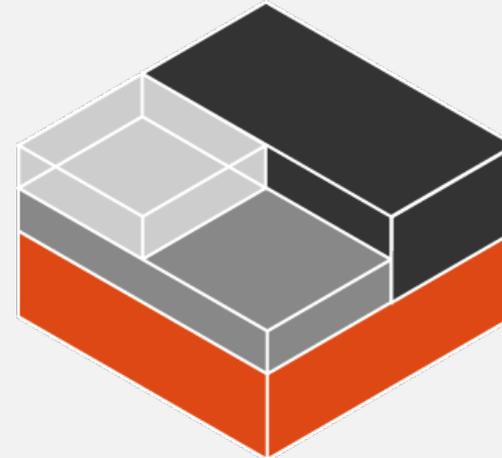
Bereitstellung globaler Systemressourcen (z.B. Netzwerk, Volume mounts) als eigene isoliert erscheinende Ressource für alle Prozesse in einem Namespace.

## Process Capabilities

Capabilities spezielle Attribute im Linux-Kernel, die Prozessen und ausführbaren Binärdateien bestimmte Berechtigungen gewähren, die normalerweise nur Prozessen des Root-Users vorbehalten sind.

## Union File System (inkl. Copy on Write)

Union-Dateisysteme gruppieren Verzeichnisse und Dateien des globalen Dateisystems in Layern. Tiefere Layer werden dabei mit höheren Layern vereinigt (daher Union, engl. Vereinigung).

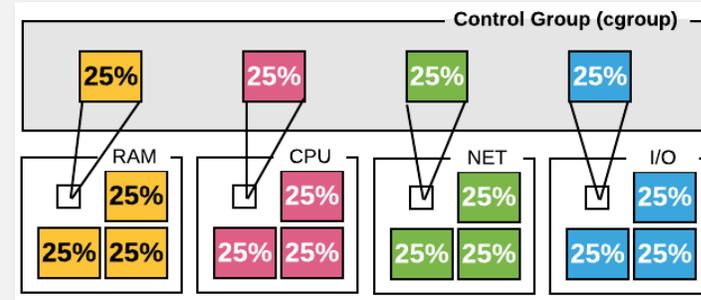


# OS-VIRTUALISIERUNG

## Linux Control Groups (Isolation durch Grenzen)

Control Groups sind eine Methode des Linux Kernels (maßgeblich von Google entwickelt), um Prozesse in Gruppen einzuteilen und Ressourcen-Limits durchzusetzen. Prozessgruppen können geschachtelt (weitere Prozessgruppen enthalten) sein.

- **Ressourcenbeschränkung:** Prozessgruppen können so eingestellt werden, dass sie ein konfiguriertes Speicherlimit nicht überschreiten können.
- **Priorisierung:** Gruppen können einen festgelegten Anteil an der CPU-Auslastung oder dem I/O-Durchsatz zugewiesen werden.
- **Beobachtung:** Der Kernel misst wie viel Ressourcen bestimmte Gruppen verbrauchen. Dies kann beispielsweise für Abrechnungszwecke oder einfach nur Reporting oder Monitoring verwendet werden.
- **Kontrolle:** Der Kernel kann Prozessgruppen bei Überschreiten definierter Grenzen einfrieren (inkl. Checkpointing) und Neustarts von Prozessgruppen veranlassen.



# OS-VIRTUALISIERUNG

*Linux Kernel Namespaces (Isolation durch Sichtbarkeit)*

Ein Namespace umschließt eine globale Systemressource mit einer Abstraktion, die den Prozessen im Namespace den Eindruck vermittelt, dass sie über eine eigene isolierte Instanz der globalen Ressource verfügen. Änderungen an der globalen Ressource sind nur für die Prozesse sichtbar, die Mitglieder des Namespace sind, für andere Prozesse jedoch nicht.

**Eine Verwendung von Namespaces besteht darin, Container zu implementieren und diese voneinander zu isolieren.**

Namespace	Isolierung
Cgroup	Cgroup root directory
IPC	System V IPC (Inter Process Communication), POSIX message queues
Network	Network devices, stacks, ports, etc.
Mount	Mount points
PID	Process IDs
Time	Boot and monotonic clocks
User	User and group IDs
UTS	Hostname and domain name

# OS-VIRTUALISIERUNG

## *Linux Process Capabilities*

Bei der Durchführung von Berechtigungsprüfungen unterscheiden herkömmliche UNIX-Implementierungen zwei Kategorien von Prozessen: privilegierte Prozesse (Superuser oder Root) und nicht privilegierte Prozesse (alle anderen Benutzer). Privilegierte Prozesse umgehen alle Kernel-Berechtigungsprüfungen, während nicht privilegierte Prozesse einer vollständigen Berechtigungsprüfung auf der Grundlage der Anmeldeinformationen des Prozesses unterliegen.

Ab Kernel 2.2 unterteilt Linux die traditionell mit Superuser verbundenen Berechtigungen in verschiedene etwa 40 Einheiten, die als Capabilities bezeichnet werden und pro Thread unabhängig voneinander aktiviert und deaktiviert werden können.

Jeder Linux Prozess hat hierzu sogenannte Capability-Sätze, die es ermöglichen Prozessen „Root-Teilrechte“ zuzuweisen (z.B. Netzwerkmanagement):

- **Effective** (E): Aktivitierte Capabilities definieren, welche Berechtigungen der Prozess tatsächlich hat.
- **Permitted** (P): Erlaubte, aber aktuell nicht genutzte Capabilities. Dies ermöglicht es Prozessen Capabilities an- und abzuschalten (bzw. zu entziehen).
- **Inheritable** (I): Welche Capabilities dürfen an Threads und Child-Prozesse weitergegeben werden.

# OS-VIRTUALISIERUNG

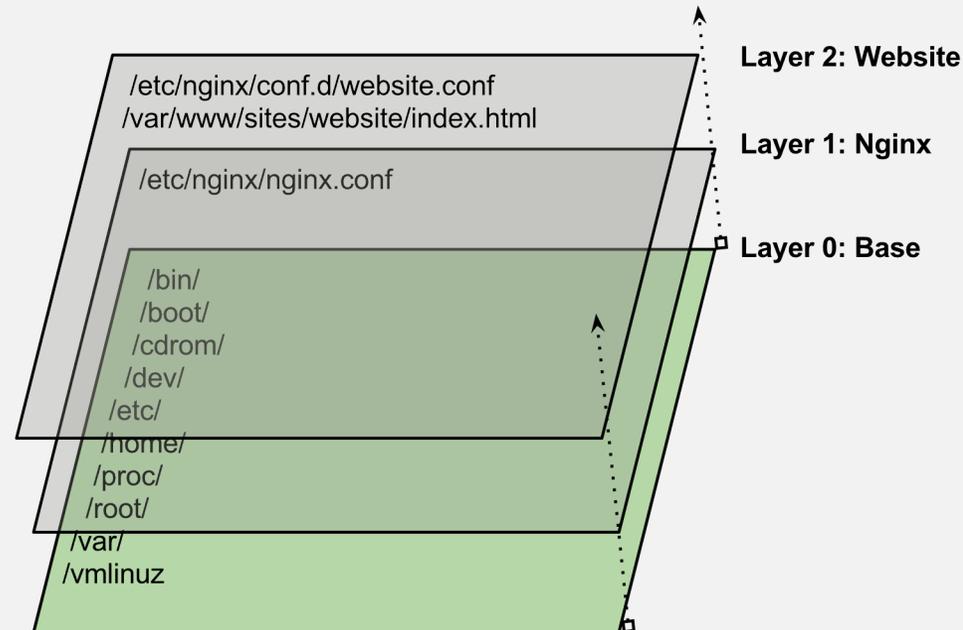
## Union Filesystems

**Union Filesystems** werden in Betriebssystemen dazu verwendet, Prozessen eigene Namensräume innerhalb von Dateisystemen zuzuweisen.

Dadurch können die Dateien verschiedener Dateisysteme zu einem einzigen logischen Dateisystem vereinigt werden.

Dateien, die zwar in getrennten Dateisystemen aber im gleichen Verzeichnis liegen, werden dadurch im selben Verzeichnis angezeigt.

Dabei werden den einzelnen beteiligten Layern Prioritäten zugeordnet, so dass eine eindeutige Zuordnung auch im Falle gleicher Dateinamen gewährleistet ist (höherer Layer überschreibt tieferen Layer).



## Isolation durch Copy-on-Write:

Jeder Container blendet mit Start erforderliche Dateien der Basis als In-Memory-Kopien ein und isoliert so sein Dateisystem vom (persistenten) Host-Dateisystem. Werden diese Dateien durch den Container geändert, erfolgt dies nur In-Memory (nicht auf dem persistenten Storage). Auf diese Weise kann ein Container nicht das Dateisystem des Hosts ändern. Er ist „stateless“ („vergisst“ geschriebene Dateien nach einem Neustart) und isoliert sich so vom Dateisystem des Hosts.

## 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

## Automationsumgebungen für Container

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

## Container-Pattern

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

# CONTAINER RUNTIME ENVIRONMENTS

## Definition und Entstehungsgeschichte

Container sind keine First-Class Objekte im Linux-Kernel. Container bestehen im Wesentlichen aus mehreren zugrunde liegenden Kernel-Grundelementen: Namespaces, Control Groups und LSMs (Linux Security Modules). Zusammen ermöglichen diese Kernel-Grundelemente, sichere, isolierte und überwachte Ausführungsumgebungen für Prozesse einzurichten.

Aber all dies jedes Mal manuell zu tun, wenn ein neuer isolierter Prozess erstellt werden soll, wäre sehr mühsam. Aus diesem Grund sind sogenannte

- Low-Level (Host-fokussiert) und
- High-Level (Multi-Host-/Cluster-fokussiert)

Container Runtime Environments entstanden.

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 eröffnete einige ihrer Containerkomponenten als **libcontainer**-Projekt.

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

# CONTAINER RUNTIME ENVIRONMENTS

## Überblick über Low-Level OCI-Runtimes

Zusätzlich zu nativen Laufzeiten, die den containerisierten Prozess auf demselben Host-Kernel ausführen, gibt es einige Sandbox- und virtualisierte Implementierer 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 Sandbox-Laufzeiten, 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 Angriffsfläche und schützen so den Host.

**runV**, **clearcontainers** und **kata-containers** sind Beispiele für virtualisierte Laufzeiten. 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-Laufzeiten, 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.

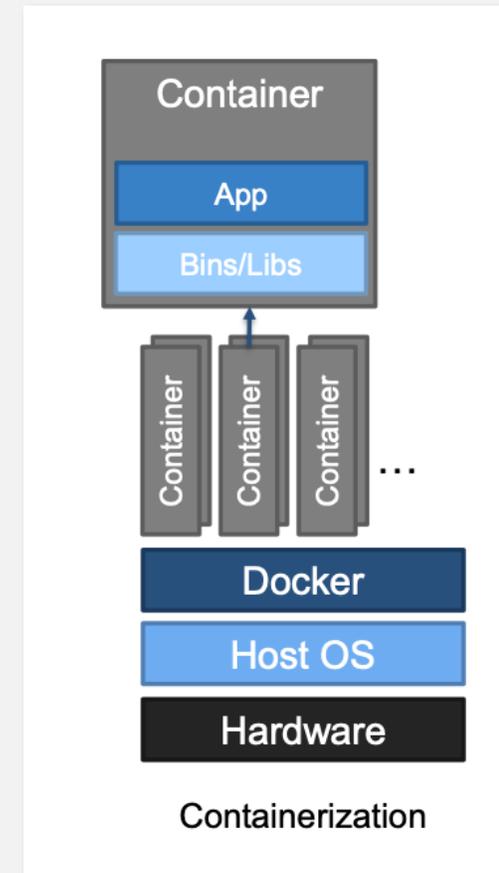
**containerd** ist Dockers High-Level-Runtime. Standardmäßig wird **runC** unter der Haube verwendet. Wie die übrigen Container-Tools, die von Docker stammen, handelt es sich um einen aktuellen De-facto-Standard. Containerd verfügt über ein Plugin-Design, um Low-Level-Runtimes wie Kata mittels containerd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

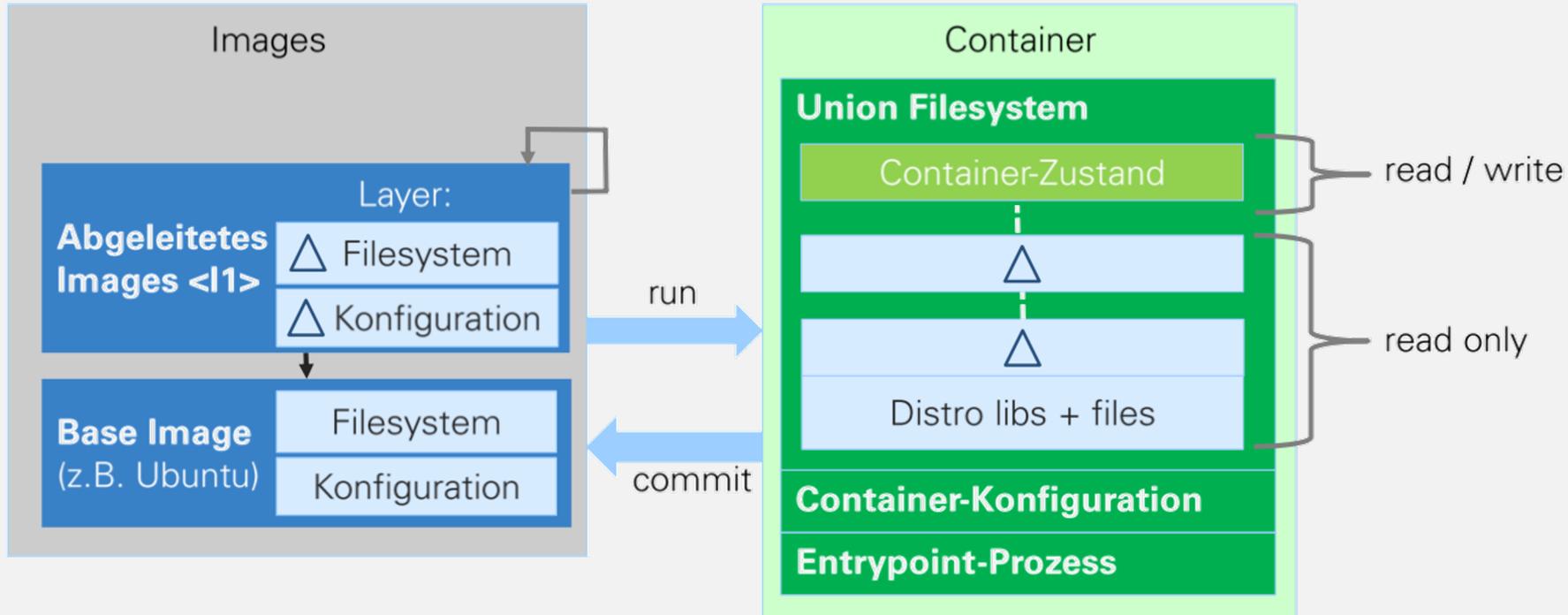
# CONTAINERIZATION MIT DOCKER

- Docker ist eine komplette Automationsumgebung 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



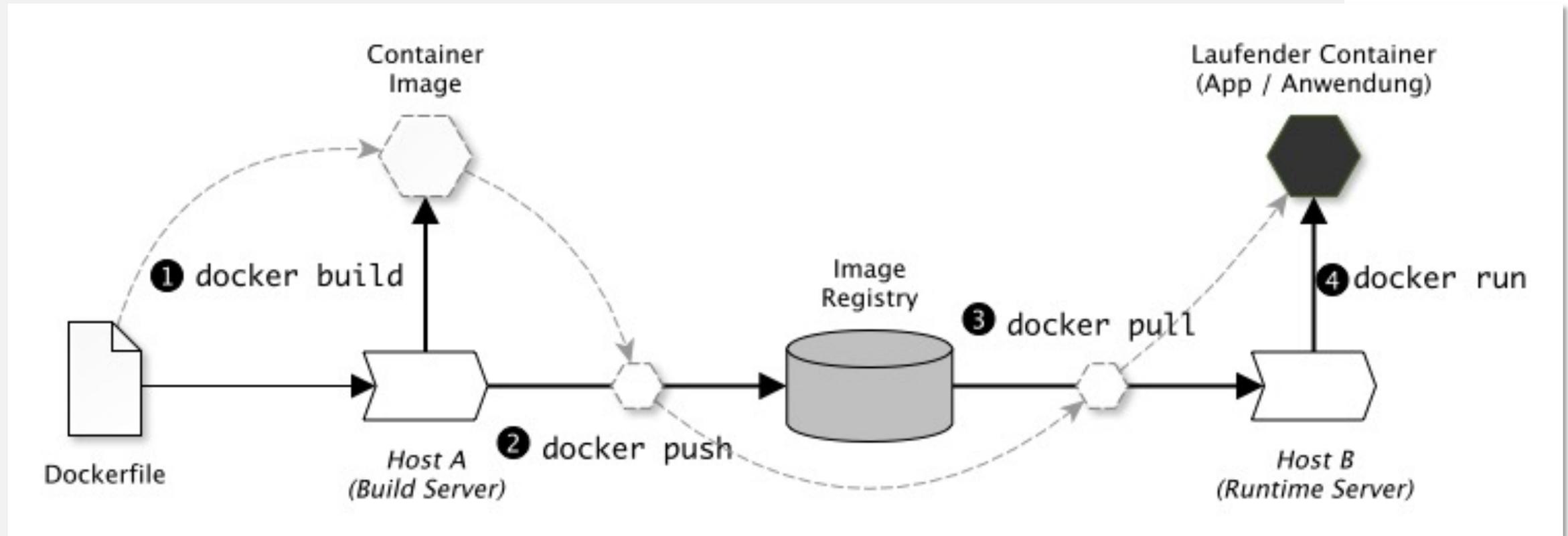
**Ruhender und transportierbarer Zustand**

**Laufender Zustand**

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

# DOCKER

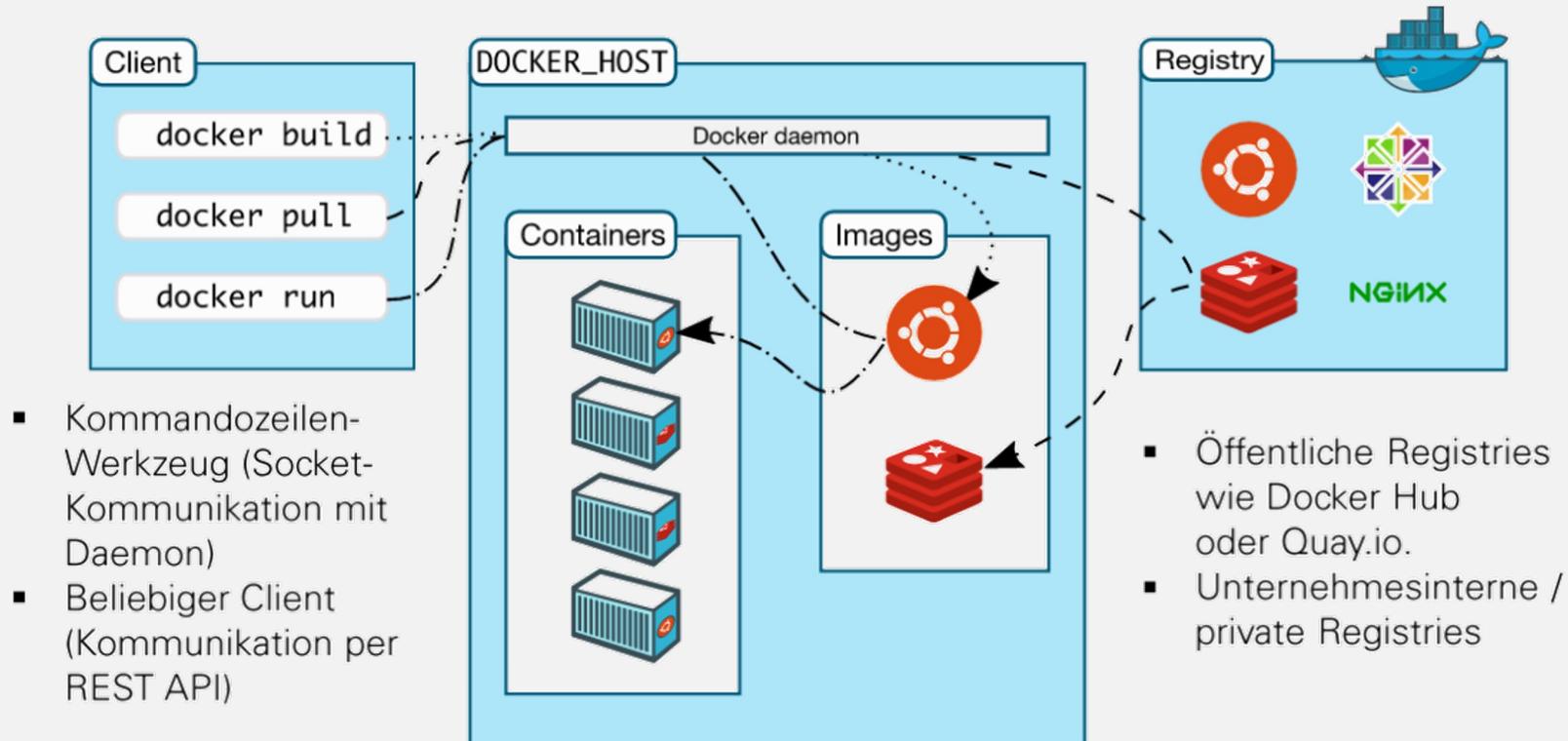
## Workflow



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

# DOCKER

## Architektur



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.

# DOCKER

## Dockerfile

Projekt (Namespace der Registry) + Container + Tag = Basis-Image von dem abgeleitet wird

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

Metainformationen

```
RUN mkdir -p /app
```

Führt ein Command im Container aus (wie in Shell)

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

Kopiert Dateien vom Host in den Container

```
ENV JAVA_OPTS -Xmx256m
```

Setzt Umgebungsvariablen im Container

```
EXPOSE 8080
```

Exponiert einen Port zum Host

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

Command, das ausgeführt wird, wenn Container gestartet wird

# HUB.DOCKER.COM

Die öffentliche Standardregistry für Docker Images



The screenshot shows the Docker Hub search results for 'nginx'. The top navigation bar includes 'dockerhub', a search bar with 'nginx' entered, and links for 'Explore', 'Pricing', 'Sign In', and 'Sign Up'. Below the navigation, there are tabs for 'Docker', 'Containers', and 'Plugins'. The main content area displays search results for 'nginx' with 1 - 25 of 68,455 results. The results are sorted by 'Most Popular'. The first result is 'nginx' by 'NGINX', which is an official image with over 10M downloads and 10K+ stars. It is updated 14 hours ago and is the official build of Nginx. The second result is 'nginx/nginx-ingress' by 'nginx', updated 20 hours ago, with over 10M downloads and 41 stars. The third result is 'nginxdemos/hello' by 'nginxdemos', updated 3 years ago, with over 10M downloads and 61 stars. The fourth result is 'nginxinc/nginx-unprivileged' by 'nginxinc', updated a month ago, with over 10M downloads and 17 stars. On the left side, there are filters for 'Docker Certified', 'Images', 'Categories', and 'Operating Systems'.

## Aber es gibt weitere Registries, z.B.:

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

# DOCKER

## Typische Kommandos eines Workflows

Command	Action
<code>docker build -t &lt;image&gt; .</code>	Build Docker image with given tag in current directory
<code>docker images</code>	Prints all local images
<code>docker run</code> <code>-d</code> <code>-v &lt;volume mounts&gt;</code> <code>-p &lt;host-port&gt;:&lt;container-port&gt;</code> <code>&lt;image&gt; &lt;entrypoint process&gt;</code>	Run a Docker image: Creates and runs a container. <ul style="list-style-type: none"><li>▪ in background</li><li>▪ with host directory mounted into the container</li><li>▪ with port forwarding from host to container</li><li>▪ image name (and optional entrypoint process)</li></ul>
<code>docker run</code> <code>-ti</code> <code>&lt;image&gt; /bin/sh</code>	Run a Docker image and open a shell within the container <ul style="list-style-type: none"><li>▪ ... with forwarding of local terminal</li><li>▪ Image name and shell (or „/bin/bash“)</li></ul>
<code>docker ps -a</code>	Prints all containers (without <code>-a</code> = only running containers)
<code>docker commit &lt;container&gt; qaware/foo</code>	Store container as local image
<code>docker kill &lt;container&gt;</code> <code>docker rm &lt;container&gt;</code>	Terminate container (send SIGKILL to entrypoint process) Remove container
<code>docker rmi -f &lt;image&gt;</code>	Remove local image

# DOCKER

## *Hilfreiche Kommandos für Container Troubleshooting*

Command	Action
<code>docker inspect &lt;container&gt;</code>	Shows container metadata (e.g. IP)
<code>docker logs &lt;container&gt;</code>	Prints container syslog
<code>docker top &lt;container&gt;</code>	Prints all running processes within a container (like <code>ps -a</code> within the container)
<code>docker exec -ti &lt;container&gt; /bin/sh</code>	Connect terminal to running container
<code>docker stats &lt;container&gt;</code>	Shows container runtime statistics (e.g. CPU usage, IO intensity, ...)

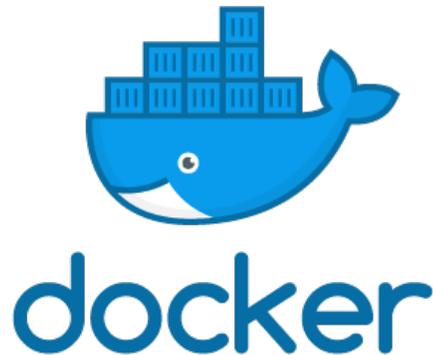
# DOCKER

*Nur Versuch macht kluch ...*



## Klonen Sie dieses Repository:

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



- > NGINX Container
- > Image Shrinking
- > Images in Registries bereitstellen
- > Deployment Pipeline zum Bau von Images

## 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

## Automationsumgebungen für Container

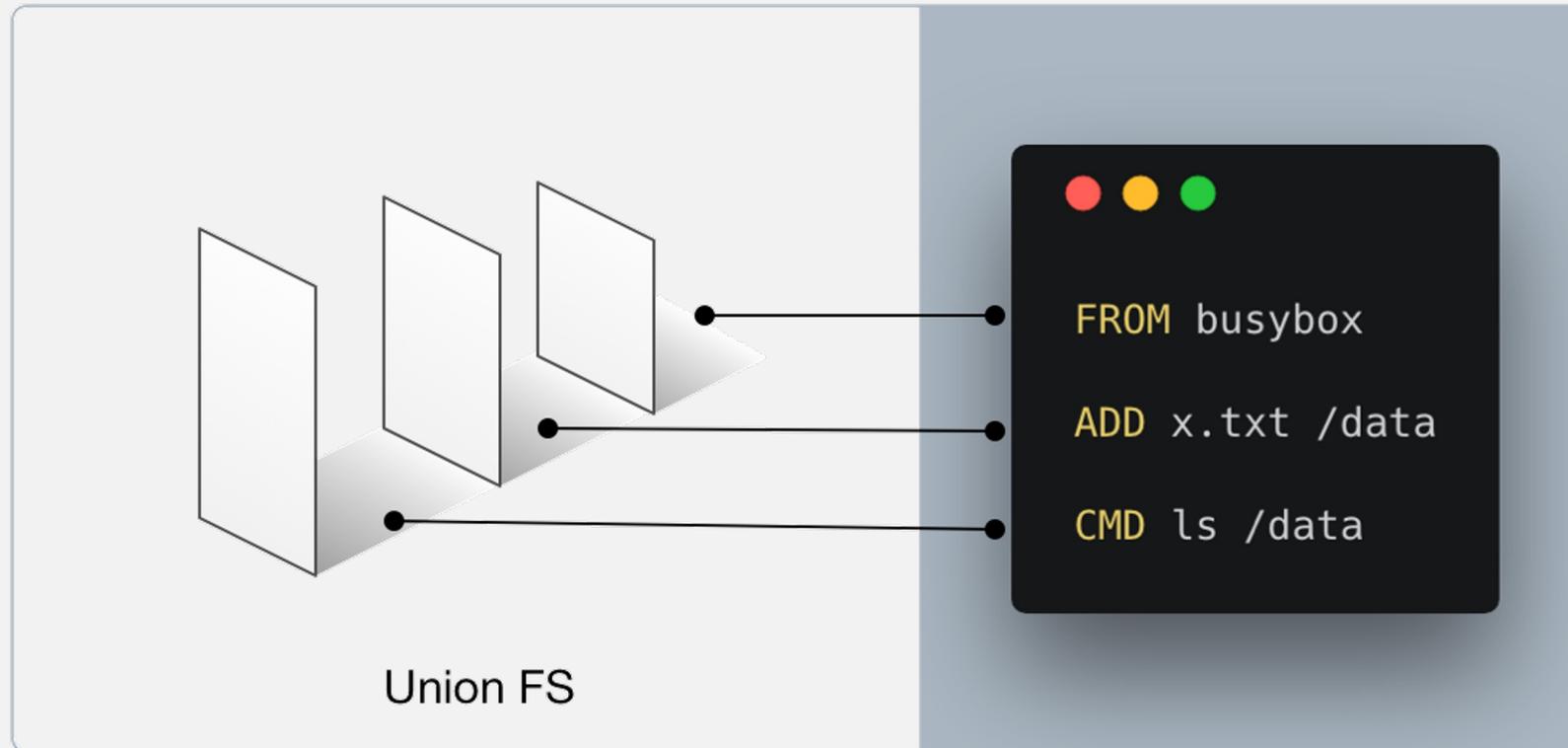
- Container Laufzeitumgebungen und Standards
- Docker
- Image Building und Registries

## Container-Pattern

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

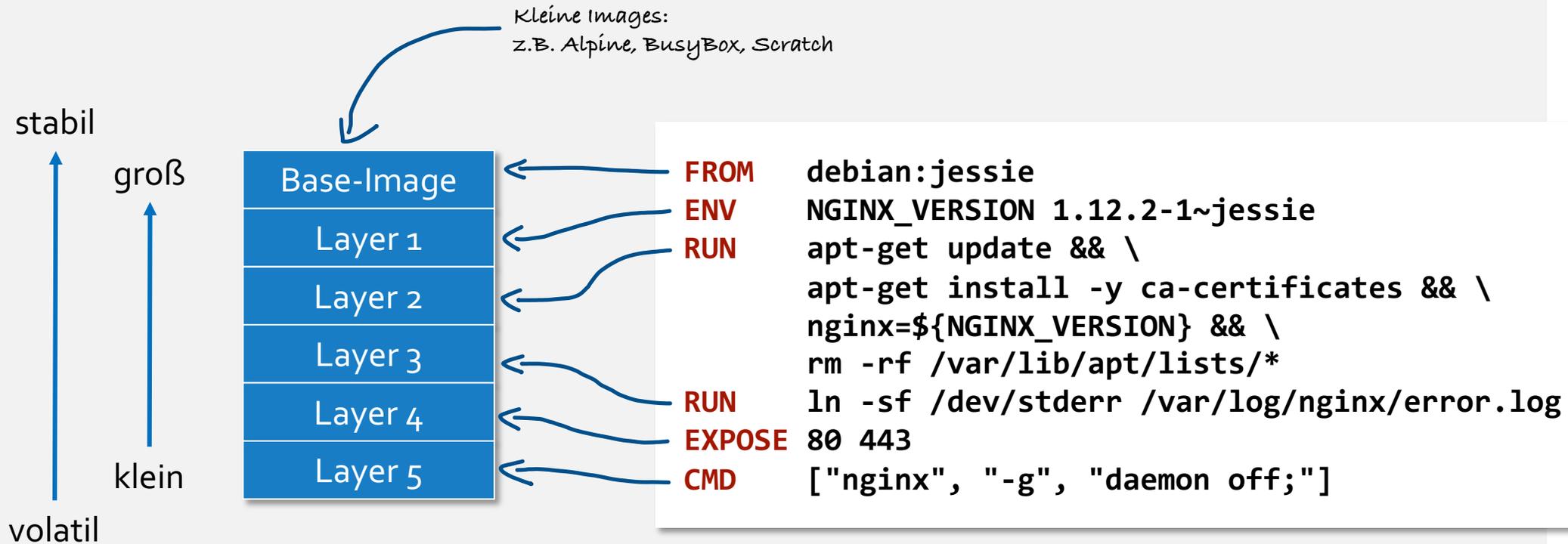
# DOCKER (ANTI-)PATTERNS

*Achtung: (Fast) jede Zeile im Dockerfile erzeugt einen neuen Image Layer*



# DOCKER PATTERN

## Image-Shrinking



Diese Abfolge ermöglicht Images, die gut cachebar sind.

1. Abhängigkeiten
2. Applikation
3. Konfiguration
4. Schnittstelle

Häufige Änderungen nur an den Stellen die kleinen Beitrag zur Imagegröße haben.

### RUN-Chaining:

```
RUN apk add --update wget git && rm -rf /var/cache/apk/*
```

### Layer verschmelzen (nur bei Basis-Images)

```
docker export + docker import
```

### Platzschonende Installation von Paketen

```
RUN apt-get update && \
  apt-get install -y --no-install-recommends apache2 wget && \
  apt-get clean && \
  rm -rf /var/lib/apt/lists/*
```

# DOCKER PATTERN

## Multi-Stage Builds

In einem Dockerfile kann man mehrere Images definieren, die sequentiell nacheinander gebaut werden.

Dies nennt man **Multi-Stage Builds**.



```
1 FROM dart:2.16-sdk AS build
2
3 ENV PATH="${PATH}:.pub-cache/bin"
4 ADD pubspec.yaml pubspec.yaml
5 ADD web/ web/
6 RUN dart pub global activate webdev && \
7     dart pub get
8 RUN webdev build --output web:build
9
10 FROM nginx:alpine
11 COPY --from=build /root/build /usr/share/nginx/html
```

*Name der Stage*

*Definition eines ersten Container Images*

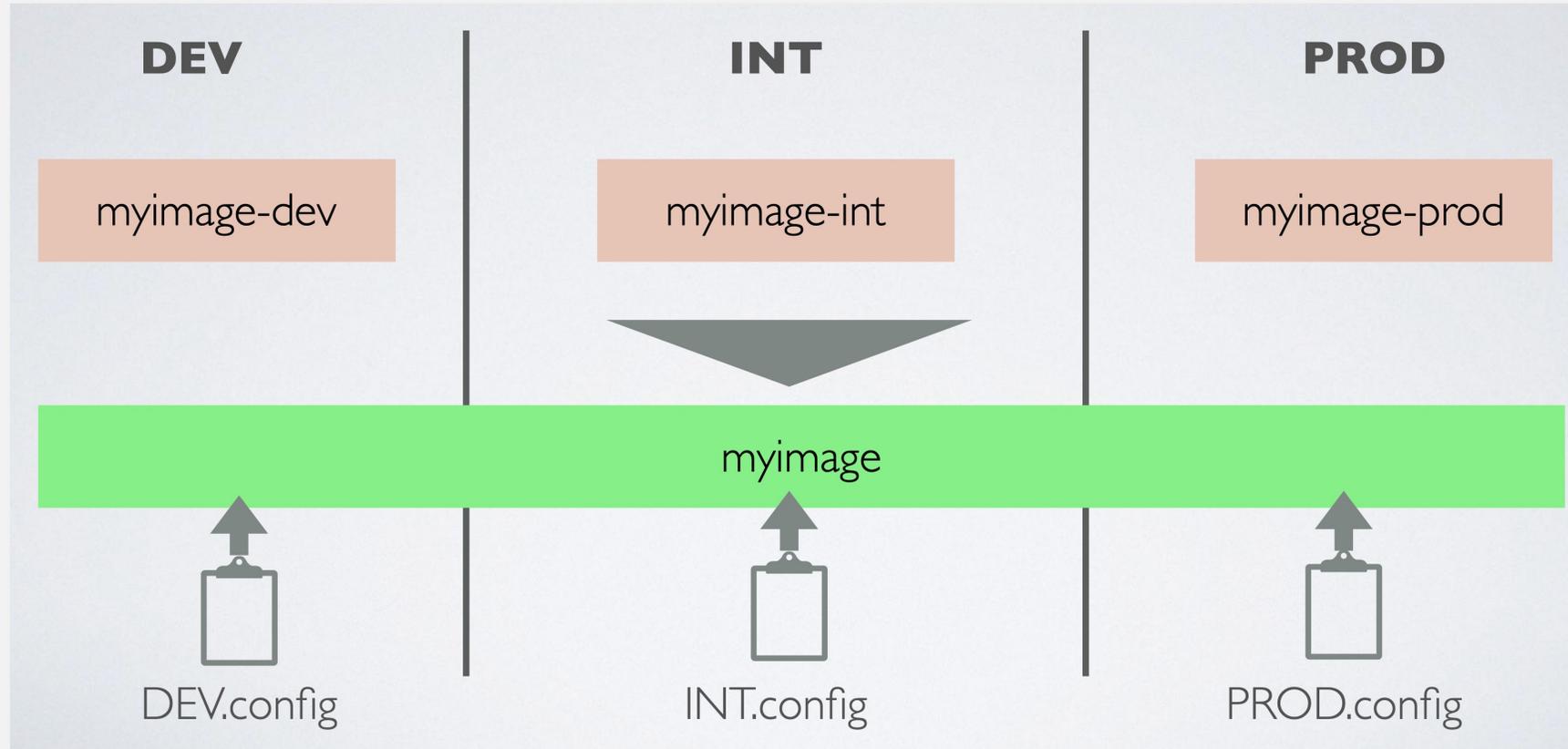
*Definition eines zweiten Container Images, dass Dateien aus dem ersten Image kopiert.*

Multi-Stage Builds können genutzt, um in Pipelines das Weiterreichen von Artefakten zwischen Jobs zu vermeiden/ minimieren.

*Kopieren von Dateien aus einer vorherigen Stage*

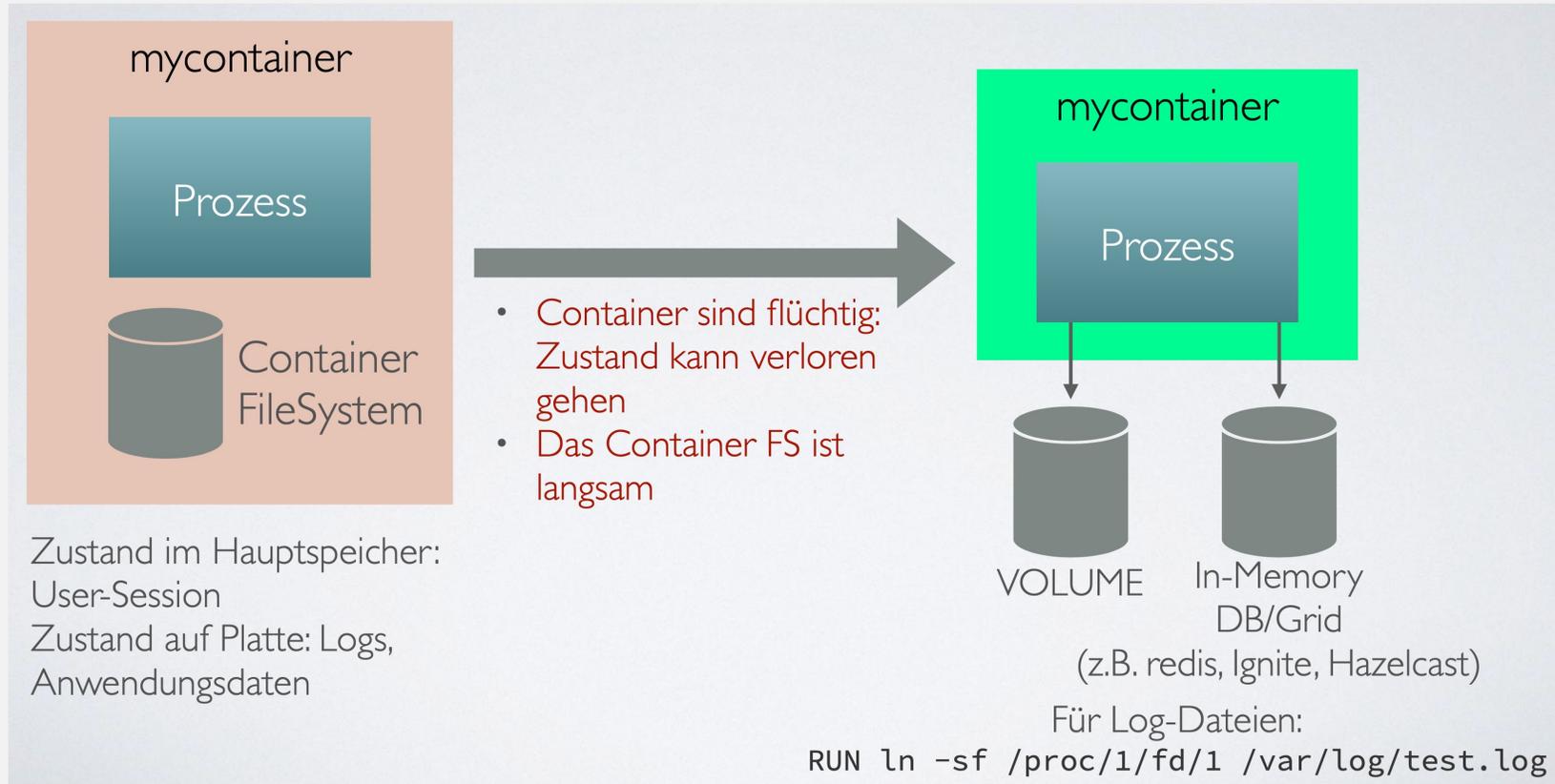
# DOCKER ANTI-PATTERN

## Image Metamorphosis



# DOCKER ANTI-PATTERN

## Stateful Container



# 12 FAKTOREN METHODE

## I. Codebase

*Eine im Versionsmanagementsystem verwaltete Codebase, viele Deployments*

## II. Abhängigkeiten

*Abhängigkeiten explizit deklarieren und isolieren*

## III. Konfiguration

*Die Konfiguration in Umgebungsvariablen ablegen*

## IV. Unterstützende Dienste

*Unterstützende Dienste als angehängte Ressourcen behandeln*

## V. Build, release, run

*Build- und Run-Phase strikt trennen*

## VI. Prozesse

*Die App als einen oder mehrere Prozesse ausführen*



## THE TWELVE-FACTOR APP

## VII. Bindung an Ports

*Dienste durch das Binden von Ports exportieren*

## VIII. Nebenläufigkeit

*Mit dem Prozess-Modell skalieren*

## IX. Einweggebrauch

*Robuster mit schnellem Start und problemlosen Stopp*

## X. Dev-Prod-Vergleichbarkeit

*Entwicklung, Staging und Produktion so ähnlich wie möglich halten*

## XI. Logs

*Logs als Strom von Ereignissen behandeln*

## XII. Admin-Prozesse

*Admin/Management-Aufgaben als einmalige Vorgänge behandeln*

# 12 FAKTOREN METHODE

## 1. Codebase

Eine Zwölf-Faktor-App wird **immer** in einem Versionsmanagementsystem als **Repository** (Codebase) verwaltet (z.B. git).

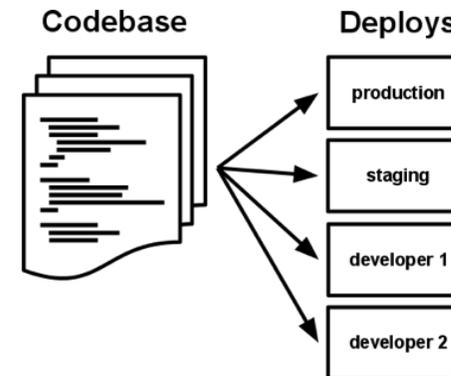
### Jede App hat dabei genau eine Codebase:

- Wenn es mehrere Codebases gibt, ist es keine App – sondern ein verteiltes System. Jede Komponente in einem verteilten System ist eine App, und Jede kann für sich den zwölf Faktoren entsprechen.
- Wenn mehrere Apps denselben Code teilen, verletzt dies die zwölf Faktoren.
- Dies lässt sich durch Codeauslagerungen in Bibliotheken und Abhängigkeitsverwaltung lösen.

**Als Deploy wird eine laufende Instanz der App bezeichnet.** Es gibt zwar nur eine Codebase pro App aber viele mögliche Deploys der App. Meist gibt es Produktionssysteme, Staging-Systeme sowie lokale Entwicklungssysteme von Entwicklern für Deploys.

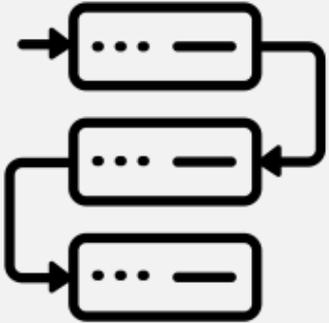
**Die Codebase ist über alle diese Deploys hinweg dieselbe**, auch wenn bei jedem Deploy unterschiedliche Versionen (Branches) aktiv sind.

- So sind in Staging meist mehr Commits als in den Produktion Deploys.
- Aber alle teilen dieselbe Codebase, was sie als verschiedene Deploys derselben App auszeichnet.



# 12 FAKTOREN METHODE

## II. Abhängigkeiten



### Abhängigkeiten explizit deklarieren und isolieren

Die meisten Programmiersprachen bieten Dependency Management Systeme an, um unterstützende Bibliotheken zu verbreiten (Rubygems für Ruby, pip für Python, pub für Dart, usw.).

Eine Zwölf-Faktor-App verlässt sich nie auf die Existenz von systemweiten **Paketen**. Sie deklariert alle Abhängigkeiten vollständig und korrekt über eine Abhängigkeitsdeklaration.

Die **vollständige und explizite Spezifikation der Abhängigkeiten** wird gleichermaßen in Produktion und Entwicklung angewandt.

Zwölf-Faktor-Apps verlassen sich nicht auf die **implizite Existenz irgendwelcher Systemwerkzeuge**.

- Beispiele dafür sind Shell-Aufrufe von ImageMagick oder curl.
- Es gibt nämlich keine Garantie, dass diese auf allen Systemen in einer kompatiblen Version vorhanden sind.
- Wenn die App per Shell auf ein Systemwerkzeug zugreift, muss die App das Werkzeug mitliefern (oder lauffähig installieren).

# 12 FAKTOREN METHODE

## III. Konfiguration



**Konfiguration in Umgebungsvariablen ablegen** (kurz auch env vars oder env), um eine strikte Trennung der Konfiguration vom Code zu erzielen.

- Umgebungsvariablen von Deploy zu Deploy zu ändern ist einfach.
- Im Gegensatz zu Konfigurationsdateien ist es unwahrscheinlich, dass sie versehentlich ins Code Repository eingechekkt werden.
- Im Gegensatz zu speziellen Konfigurationsdateien oder anderen Konfigurationsmechanismen wie den Java Properties sind sie Sprach- und Betriebssystemunabhängig.

**Die Konfiguration einer App ist alles, was sich zwischen den Deploys ändert**

(Staging, Produktion, Entwicklungsumgebungen, usw.). Dies umfasst:

- Resource-Handles für Datenbanken, Memcached und andere unterstützende Dienste
- Credentials für externe Dienste wie Amazon S3 oder Twitter
- Direkt vom Deploy abhängige Werte wie der kanonische Hostname für den Deploy

**Niemals die Konfiguration als Konstanten im Code speichern!**

Die Konfiguration ändert sich deutlich von Deploy zu Deploy, ganz im Gegensatz zu Code.

*Hinweis:*

*Definition von "Konfiguration" umfasst nicht die interne Anwendungskonfiguration, wie bspw. `config/routes.rb` in Rails.*

*Diese Art von Konfiguration ändert sich nämlich nicht von Deploy zu Deploy und gehört daher zum Code!*

# 12 FAKTOREN METHODE

## III. Konfiguration

*Definieren von  
Umgebungsvariablen  
in einem Dockerfile*

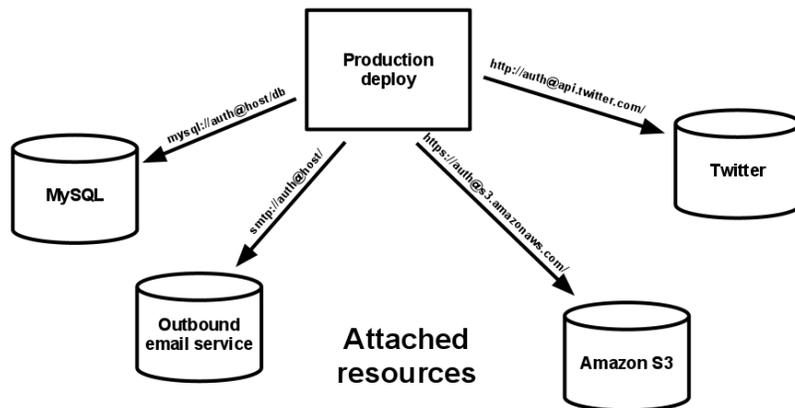
```
1 FROM dart:2.16-sdk AS build
2
3 ENV PATH="${PATH}:.pub-cache/bin"
4 ADD pubspec.yaml pubspec.yaml
5 ADD web/ web/
6 RUN dart pub global activate webdev && \
7     dart pub get
8 RUN webdev build --output web:build
```

```
9
10 FRO
11 COP
1 import wikipedia, structlog, logging, sys, os, redis, datetime, traceback
2 from tqdm import tqdm
3 from pymongo import MongoClient
4
5 BATCH = int(os.environ.get('BATCH', '50')) # Processing n documents until pushing to mongo
6 ROUNDS = int(os.environ.get('ROUNDS', '1000')) # Restart crawler after n rounds
7 PAGES = int(os.environ.get('PAGES', '50')) # Start with n random documents the crawling
8 REDIS = os.environ.get('REDIS', 'redis') # DNS name of Redis service
9 MONGO = os.environ.get('MONGO', 'mongo') # DNS name of MongoDB service
10 LANG = os.environ.get('LANGUAGE', 'de') # Language code
11 WAIT = int(os.environ.get('WAIT', '100')) # Wait in ms between two calls
```

*Auslesen von  
Umgebungs-  
variablen in einem  
Prozess (hier  
Python)*

# 12 FAKTOREN METHODE

## IV. Unterstützende Dienste



**Ein unterstützender Dienst ist jeder Dienst, den die App über das Netzwerk im Rahmen ihrer normalen Arbeit konsumiert** (z.B. MySQL, CouchDB, RabbitMQ, usw.).

Unterstützende Dienste können selbstverwaltet (z.B. selbst gehostete MySQL-Datenbank) oder auch von Dritten verwaltete Dienste sein (z.B. DB-Dienste wie AWS RDS oder auch über eine API zugängliche Dienste (wie Twitter, Google Maps, etc.) und viele mehr.

**Der Code einer Zwölf-Faktor-App macht keinen Unterschied zwischen lokalen Diensten und solchen von Dritten.**

Für die App sind sie beide unterstützende Dienste, zugreifbar über eine URL oder andere Lokatoren/Credentials, die in der Konfiguration gespeichert sind.

Ressourcen können beliebig an Deploys an- und abgehängt werden. Wenn zum Beispiel die Datenbank einer App aufgrund von Hardwareproblemen aus der Rolle fällt, könnte der App-Administrator eine neue Datenbank aus einem Backup aufsetzen. Die aktuelle Produktionsdatenbank könnte abgehängt und die neue Datenbank angehängt werden – ganz ohne Codeänderung.

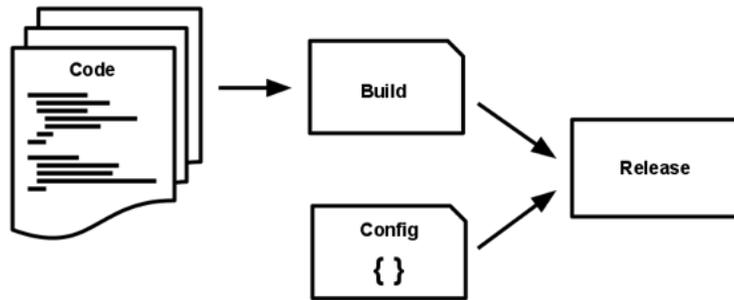
*Merke:*

*Jeder einzelne Dienst ist als unterstützende aber austauschbare Ressource zu behandeln.*

*Ihr Zugriff sollte daher in einer Konfiguration (Lokatoren + Credentials) gespeichert werden können.*

# 12 FAKTOREN METHODE

## V. Build, release, run



Jedes Release sollte eine eindeutige Release-ID haben, wie zum Beispiel einen Zeitstempel des Releases (2011-04-06-20:32:17) oder eine laufende Nummer (v100). Releases werden nie gelöscht und ein Release kann nicht verändert werden, wenn es einmal angelegt ist. Jede Änderung erzeugt einen neuen Release.

Mit rollback Kommandos eines Releasemanagement Werkzeugs kann dann einfach und schnell auf einen früheren Release zurückgesetzt werden.

Eine Codebase wird durch drei Phasen in einen (Nicht-Entwicklungs)-Deploy transformiert:

- Die **Build-Phase** ist eine Transformation, die ein Code-Repository in ein ausführbares Code-Bündel übersetzt, das man auch Build nennt. Ausgehend von einer Code-Version eines Commits werden Binaries und Assets erzeugt.
- Die **Release-Phase** übernimmt den Build von der Build-Phase und kombiniert ihn mit der zum Deploy passenden Konfiguration. Der so erzeugte Release enthält sowohl den Build, als auch die Konfiguration und kann in einer Ausführungsumgebung ausgeführt werden.
- Die **Run-Phase** (auch "Laufzeit" genannt) führt die App in einer Ausführungsumgebung aus, indem sie eine Menge der Prozesse der App in einem Release ausführt.

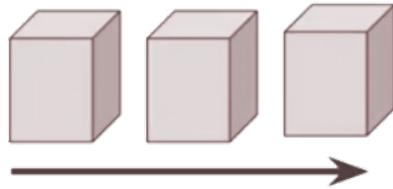
*Hinweise:*

*Die Zwölf-Faktor-App trennt strikt zwischen Build-, Release- und Run-Phase.*

*Es ist nicht möglich, Code-Änderungen zur Laufzeit zu machen, weil es keinen Weg gibt, diese Änderungen zurück in die Build-Phase zu schicken.*

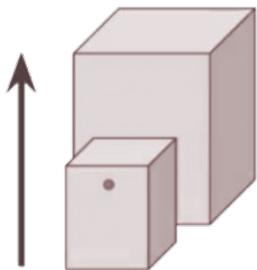
# 12 FAKTOREN METHODE

## VI. Prozesse



Horizontal scaling

*Horizontale Skalierung erfordert über mehrere Requests den Prozessraum zu verlassen.*



Vertical scaling

*Vertikale Skalierung ermöglicht über mehrere Requests in einem Prozessraum bleiben zu können.*

### Die App wird als ein oder mehrere Prozesse ausgeführt

Muss die App mehr Last verarbeiten, werden einfach mehr Prozesse gestartet (**horizontale Skalierung**). Auf schwer zu beherrschende In-App Parallelisierung (bspw. mittels Threads) sollte wenn möglich verzichtet werden.

### Zwölf-Faktor-Apps sind zustandslos und Shared Nothing

Alle Daten werden in unterstützenden Diensten gespeichert, häufig in einer Datenbank.

Der RAM oder das Dateisystem des Prozesses kann zwar als kurzfristiger Cache für die Dauer einer Transaktion verwendet werden.

Zum Beispiel kann ein Prozess eine Datei herunterladen, sie verarbeiten und die Ergebnisse in einer Datenbank speichern. Die Zwölf-Faktor-App geht aber nie davon aus, dass irgendetwas aus dem RAM oder im Dateisystem zwischengespeichertes für einen künftigen Request oder Job verfügbar sein wird.

*Merke:*

*12 Factor Apps werden für horizontale Skalierbarkeit entworfen.*

*Hinweise:*

*Manche Web-Systeme verlassen sich auf "Sticky Sessions" – sie cachten Benutzer-Session-Daten und erwarten, dass künftige Requests desselben Benutzers zum selben Prozess geschickt werden.*

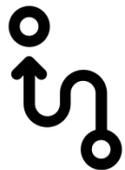
*Sticky Sessions sind eine Verletzung der zwölf Faktoren. Solches Cachen kann mit Prozess-externen Cacheing-Lösungen wie bspw. Memcached oder Redis gelöst werden.*

# 12 FAKTOREN METHODE

## VII. Bindung an Ports



Dienste über  
Ports  
exportieren



Öffentliche  
Requests an  
exportierte  
Ports routen

### Dienste durch Port Binding exportieren

Zwölf-Faktor-Apps sind vollständig eigenständig (self-contained). Web-Apps exportieren bspw. HTTP als Dienst, indem sie sich an einen Port binden und warten an diesem Port auf HTTP-Requests.

Üblicherweise wird dies mittels Abhängigkeitsdeklaration implementiert. Zu der App fügt man eine Webserver-Bibliothek hinzu wie Tornado für Python, Thin für Ruby oder Jetty für Java und andere JVM-basierenden Sprachen. Dies findet vollständig im User Space statt, also im Code der App. Der Vertrag mit der Laufzeitumgebung ist das Binden an einen Port, um Requests zu bedienen.

In einer lokalen Entwicklungsumgebung kann ein Entwickler so über Dienst-URL wie bspw. `http://localhost:5000/` auf den Dienst der App zuzugreifen. Beim Deployment sorgt eine Routing-Schicht dafür, dass Requests von einem öffentlich sichtbaren Hostnamen zu den an die Ports gebundenen Prozessen kommen (ggf. inklusive TLS Termination).

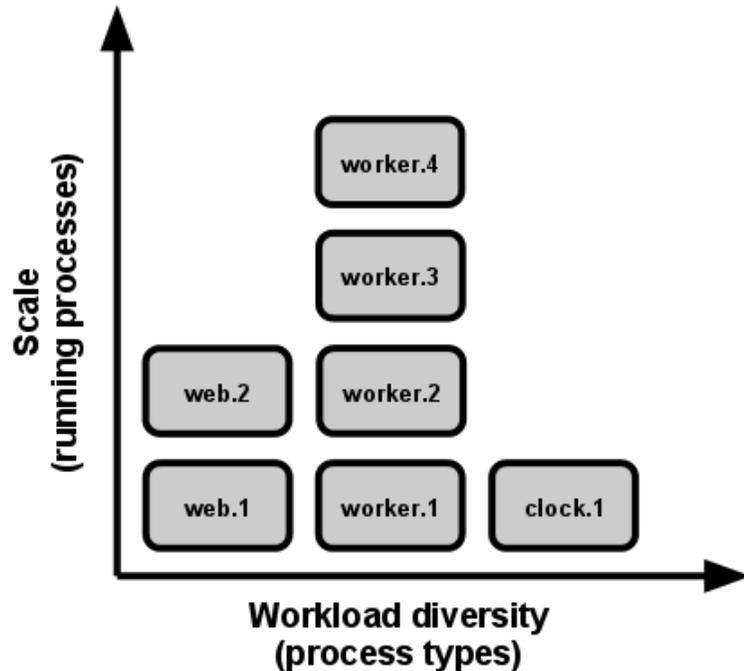
Für andere Protokolle wird analog verfahren.

#### Hinweise:

Durch Port Binding kann eine App ein unterstützender Dienst (siehe Faktor IV) für eine andere App werden, indem die URL der unterstützenden App der konsumierenden App als Resource-Handle zur Verfügung gestellt wird.

# 12 FAKTOREN METHODE

## VIII. Nebenläufigkeit



### Mit dem Prozess skalieren

Die Prozesse der Zwölf-Faktor-App orientieren sich am Unix-Prozess-Modell für Service Daemons. Mit diesem Modell können Entwickler ihre App für die Bearbeitung verschiedenster Aufgaben konzipieren, in dem sie jeder Aufgabe einen Prozessstyp zuweisen. Zum Beispiel können HTTP-Requests durch einen Web-Prozess bedient werden und langlaufende Hintergrundarbeiten durch einen Worker-Prozess.

Diese Shared-Nothing sowie horizontal teilbare Art und Weise der Prozesse der Zwölf-Faktor-App hat zur Folge, dass weitere Nebenläufigkeit einfach und zuverlässig durch Starten neuer Prozesse hinzugefügt werden kann.

Die Prozesse einer Zwölf-Faktor-App sollten nie als Daemons laufen oder PID-Dateien schreiben. Stattdessen sollen sie sich auf den Prozessmanager des Betriebssystems verlassen (wie systemd, den verteilten Prozessmanager einer Cloud-Plattform, Kubernetes o. ähnl.) um Output-Streams zu verwalten, auf abgestürzte Prozesse zu reagieren und mit von Benutzern angestoßenen Restarts und Shutdowns umzugehen.

### Hinweise:

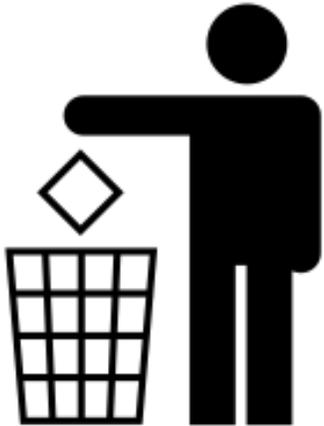
Dieser Faktor verbietet nicht, dass Prozesse ihr internes Multiplexing realisieren. Z.B. mittels Threads in der Laufzeit-VM oder mit dem Async/Event-Modell von Werkzeugen wie EventMachine, Twisted oder Node.js.

Aber eine 12 Faktor App sollte in der Lage sein, dass mehrere Prozesse auf mehreren physikalischen oder virtualisierten Maschinen oder in Containern gestartet werden können (diese können intern multiplexen).

# 12 FAKTOREN METHODE

## IX. Einweggebrauch

*Die Prozesse einer Zwölf-Faktor-App können „weggeworfen“ werden, weil sie schnell gestartet und gestoppt werden können.*



### Prozesse sollten möglichst geringe Startup-Zeiten haben

Idealerweise braucht ein Prozess wenige Sekunden vom Startkommando bis der Prozess läuft und Requests oder Jobs entgegennehmen kann. Kurze Startup-Zeiten geben dem Release-Prozess und der Skalierung mehr Agilität und optimieren die Robustheit, weil ein Prozessmanager bei Bedarf einfacher Prozesse auf neue physikalische Maschinen verschieben kann.

### Mit SIGTERM Prozesse herunterfahren

Für einen Web-Prozess kann ein problemloses Herunterfahren erreicht werden, indem er aufhört an seinem Service-Port zu lauschen (und damit alle neuen Requests ablehnt), die laufenden Requests zuende bearbeitet und sich dann beendet. Für einen Worker-Prozess wird ein problemloser Stopp erreicht, indem er seinen laufenden Job an die Workqueue zurück gibt.

### Prozesse sollten robust gegen plötzlichen Tod (SIGKILL) sein

Prozesse sollten zwar nicht mit SIGKILL beendet werden, aber die zugrundeliegende Hardware kann versagen. Auch wenn dies viel seltener ist als ein reguläres Herunterfahren mit SIGTERM, so kommt es dennoch vor.

#### *Hinweise:*

*Schnelle Starts und problemlose Stopps optimieren die Robustheit.*

*Dieser Faktor erleichtert schnelles elastisches Skalieren, schnelles Deployment von Code oder Konfigurationsänderungen und macht Produktionsdeployments robuster.*

#### *MERKE:*

*Anything fails all the time!*

# 12 FAKTOREN METHODE

## X. Dev-Prod-Vergleichbarkeit

### Entwicklung, Staging und Produktion so ähnlich wie möglich halten, um folgende Lücken zu schließen:

- Die **Zeit-Lücke** Ein Entwickler arbeitet an Code der Tage, Wochen oder sogar Monate braucht um in Produktion zu gehen.
- Die **Personal-Lücke**: Entwickler schreiben Code, Operatoren deployen ihn.
- Die **Werkzeug-Lücke**: Entwickler nutzen vielleicht einen Stack wie Nginx, SQLite und OS X - die Produktion nutzt Apache, MySQL und Linux.



### Zwölf-Faktor-Apps sind auf Continuous Deployment und Minimierung der Lücken ausgelegt

- Die Zeit-Lücke klein halten: Ein Entwickler kann Code schreiben, der Stunden oder sogar Minuten später deployed wird.
- Die Personal-Lücke klein halten: Entwickler die Code schreiben sind intensiv am Deployment und der Überwachung des Verhaltens auf Produktion beteiligt.
- Die Werkzeug-Lücke klein halten: Entwicklung und Produktion so ähnlich wie möglich halten.

### Unterstützende Dienste

Im Bereich der unterstützenden Dienste wie der Datenbank der App, dem Queue-System oder dem Cache ist die Dev-Prod-Vergleichbarkeit wichtig. Viele Sprachen bieten zwar Bibliotheken, die den Zugriff auf die unterstützenden Dienste vereinfachen und ebenso Adapter für unterschiedliche Arten von Diensten.

Kleinste Inkompatibilitäten zwischen den unterstützenden Diensten können bereits Probleme verursachen. Code, der in Entwicklung oder Staging funktioniert und Tests besteht, scheitert dann ggf. in Produktion.

### Merke:

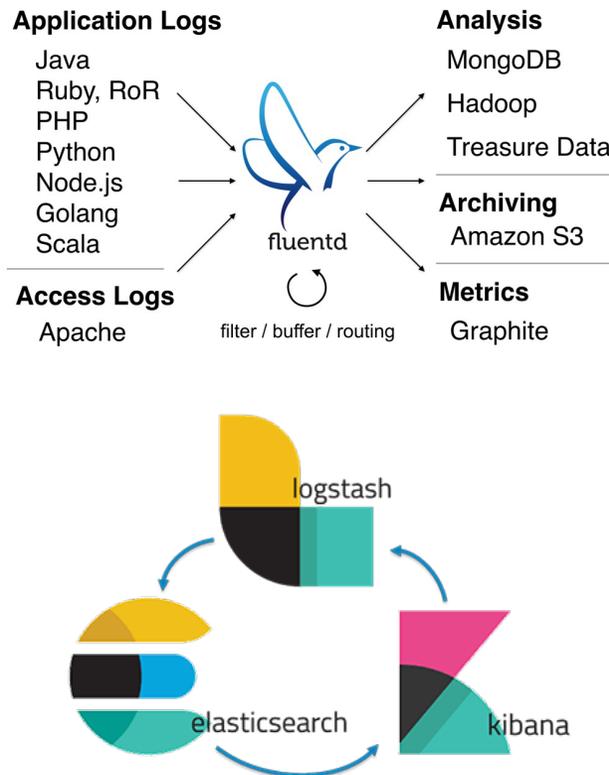
*Die Zwölf-Faktor-Entwicklung widersteht dem Drang, verschiedene unterstützende Dienste in Entwicklung und Produktion zu verwenden*

*Diese Reibungskosten und die Störungen im Continuous Deployment sind sehr hoch, wenn man sie über die Lebensdauer einer App aggregiert.*

# 12 FAKTOREN METHODE

## XI. Logs

### Unified Logging Layer Solutions



### Logs als Strom von Ereignissen behandeln

Logs sind der Stream von aggregierten, nach Zeit sortierten Ereignissen und zusammengefasst aus den Output Streams aller laufenden Prozesse und unterstützenden Dienste. Logs in ihrer rohen Form sind üblicherweise ein Textformat mit einem Ereignis pro Zeile.

### Eine Zwölf-Faktor-App kümmert sich nie um das Routing oder die Speicherung ihres Output Streams

12 Faktor Apps schreiben den Stream von Ereignissen ungepuffert auf stdout. Bei einem lokalen Deployment sieht ein Entwickler diesen Stream im Vordergrund seines Terminals, um das Verhalten der App zu beobachten.

Auf Staging- oder Produktionsdeploys werden die Streams aller Prozesse von der Laufzeitumgebung erfasst, mit allen anderen Streams der App zusammengefasst und zu einem oder mehreren Zielen (z.B. mittels fluentd) geleitet, zur Ansicht oder Archivierung.

Diese Archivierungsziele sind für die App weder sichtbar noch konfigurierbar - sie werden vollständig von der Laufzeitumgebung aus verwaltet.

#### Merke:

*Logs werden in Server-basierten Umgebungen üblicherweise in eine Datei auf der Platte geschrieben (eine Logdatei) - das ist aber nur ein Ausgabeformat und man ist nicht daran gebunden.*

*12-Faktor-Apps schreiben den Stream von Ereignissen ungepuffert auf stdout damit Logs einer Logkonsolidierung möglichst unkompliziert unterworfen werden können.*

# 12 FAKTOREN METHODE

## XI. Logs

```
print("Ich bin ein Log-Entry in Python")
```

*Sehr einfaches Logging (log to stdout), kein Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren zu können)*

```
import json
print(json.dumps({
    "event": "Log-Entry",
    "message": "Ich bin ein Log-Entry",
    "language": "Python"
}))
```

*Sehr einfaches strukturiertes Logging (log JSON to stdout), ebenfalls ohne Einsatz spezifischer Logging-Libraries (reicht schon aus, um Logs konsolidieren und zielgerichteter auswerten zu können in einer Log-Konsolidierung)*

*Merke:  
Komplizierter  
muss es nicht  
unbedingt sein.*

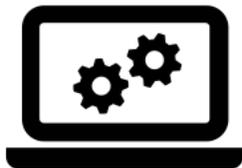
# 12 FAKTOREN METHODE

## XII. Admin Prozesse

### Einmalige / seltene Wartungsaufgaben

Neben Prozessen zur Erledigung der üblichen Aufgaben einer App (wie die Abarbeitung von Web-Requests) müssen jedoch auch einmalige Administrativ- oder Wartungsaufgaben an der App erledigt werden, wie bspw.:

- Datenbank-Migrationen
- Konsole starten
- Einmalig auszuführende Skripte starten (z.B. `fix_bad_records.py`)



### Identische Umgebung

Einmalige Administrationsprozesse sollten in einer Umgebung laufen, die identisch ist zu der Umgebung der üblichen langlaufenden Prozesse.

- Sie laufen gegen einen Release und benutzen dieselbe Codebase und Konfiguration wie jeder Prozess, der gegen einen Release läuft.
- Administrationscode wird mit dem App-Code ausgeliefert, um Synchronisationsprobleme zu vermeiden.
- Der Administrationscode sollte dieselbe Dependency Management Lösung verwenden, wie auch die App (siehe Faktor II)



# KONTAKT

**Nane Kratzke**

☎ +49 451 300-5549

✉ [nane.kratzke@th-luebeck.de](mailto:nane.kratzke@th-luebeck.de)

🌐 [kratzke.mylab.th-luebeck.de](http://kratzke.mylab.th-luebeck.de)

