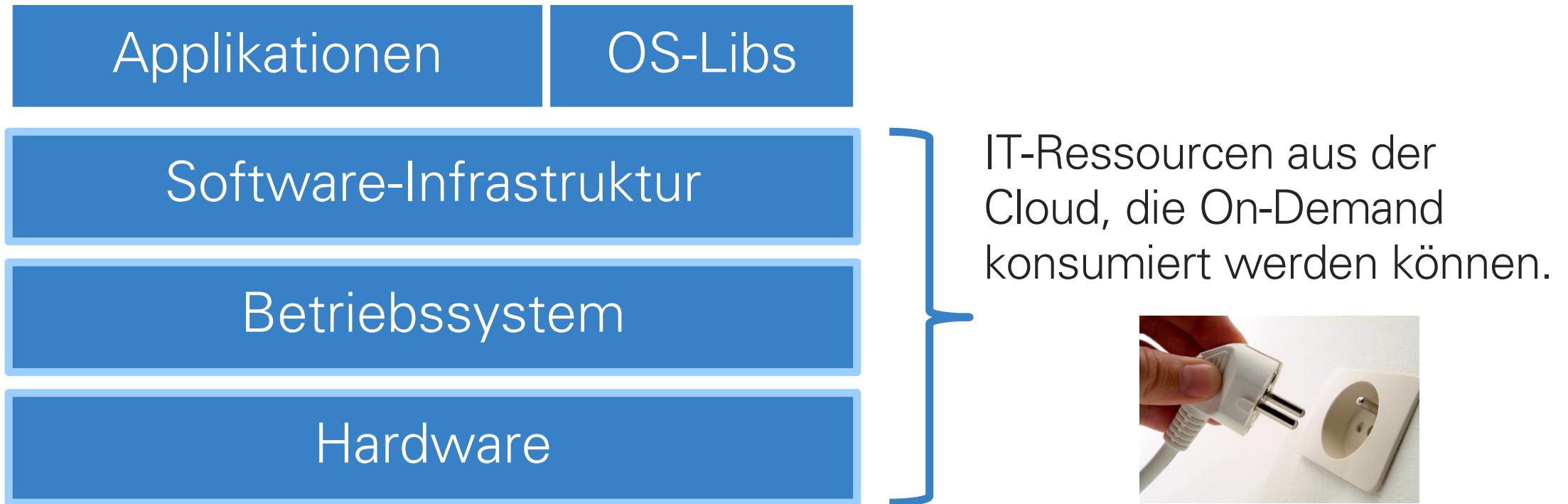


# Kapitel 0: Einführung

Im Kern geht es um eine geringere Verbauungstiefe bei der Systementwicklung & dem Betrieb.



“computation may someday be organized as a public utility”, John McCarthy, 1961

# Die Cloud ist dynamisch, elastisch und omnipräsent.



## Die wichtigsten Eigenschaften von Cloud Computing:

- **X as a Service:** On-Demand Charakter; Bereitstellung von Rechenkapazitäten, Plattform-Diensten und Applikationen auf Anfrage und in Echtzeit.
- **Ressourcen-Pools:** Verfügbarkeit von scheinbar unbegrenzten Ressourcen, die Anfragen verteilt verarbeiten.
- **Elastizität:** Dynamische Zuweisung von zusätzlichen Ressourcen bei Bedarf (Selbst-Adaption). Keine Kapazitätsplanung aus Sicht des Nutzers mehr nötig.
- **Pay-as-you-go Modell → Economy of Scale;** Die Kosten skalieren mit dem Nutzen.
- **Omnipräsenz:** Zugriff auf die Cloud über das Internet und von verschiedenen Endgeräten aus (über Standard-Protokolle).

# Die 5 Gebote der Cloud.

1. Everything Fails All The Time.
2. Focus on MTTR and not on MTTF.
3. Respect the Eight Fallacies of Distributed Computing.
4. Scale out, not up.
5. Treat resources as cattle, not pets.

# Nutzen der Cloud.

## Temporäre Server

- Projekt-Server
- Test-Server
- Server für Prototypen

## Einfaches Deployment

- Automatisches Deployment von Anwendungen
- Automatischer Aufbau verschiedener Deployment-Varianten

## Skalierbare Applikationen

- Dynamische Skalierung, je nach Anfragelast

## Umfangreiche Berechnungen

- Analyse von Transaktionen
- Aggregation von Daten
- Data-Warehousing

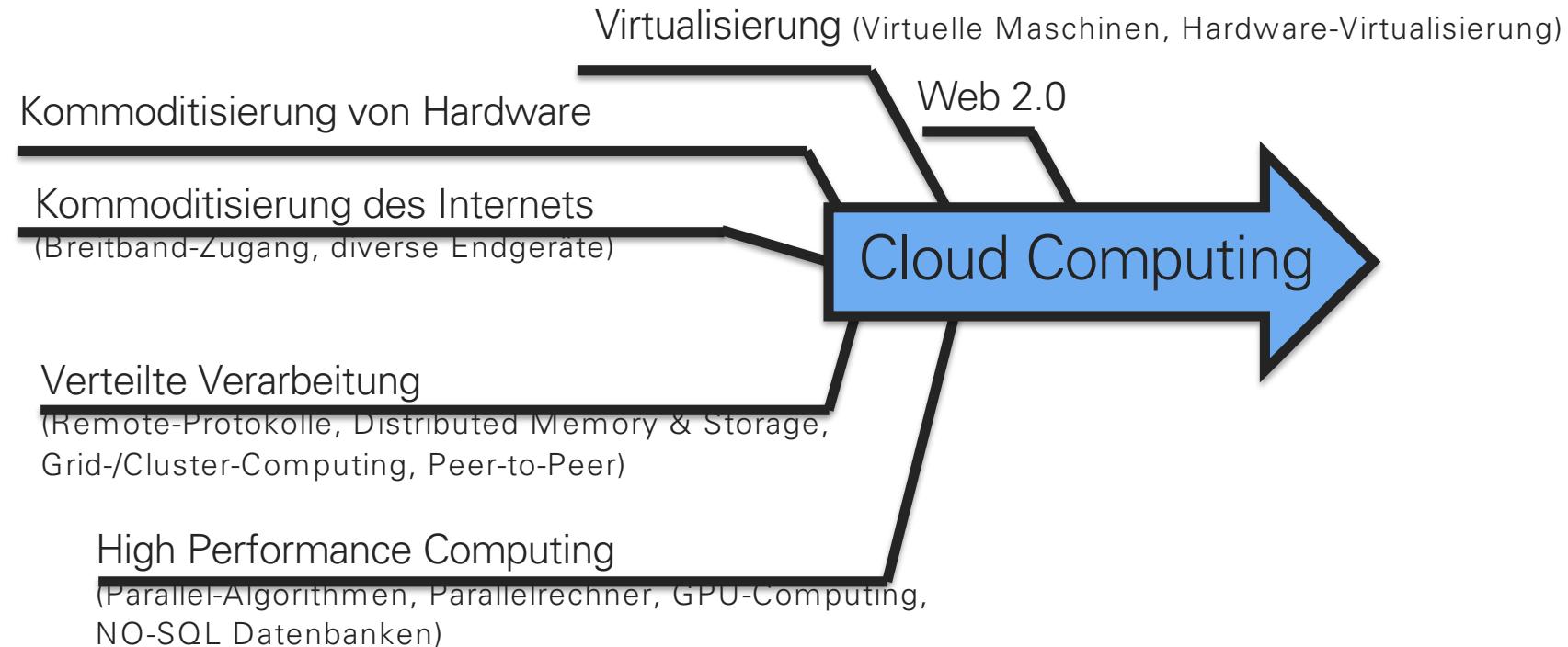


<http://ielastic.com/de/>

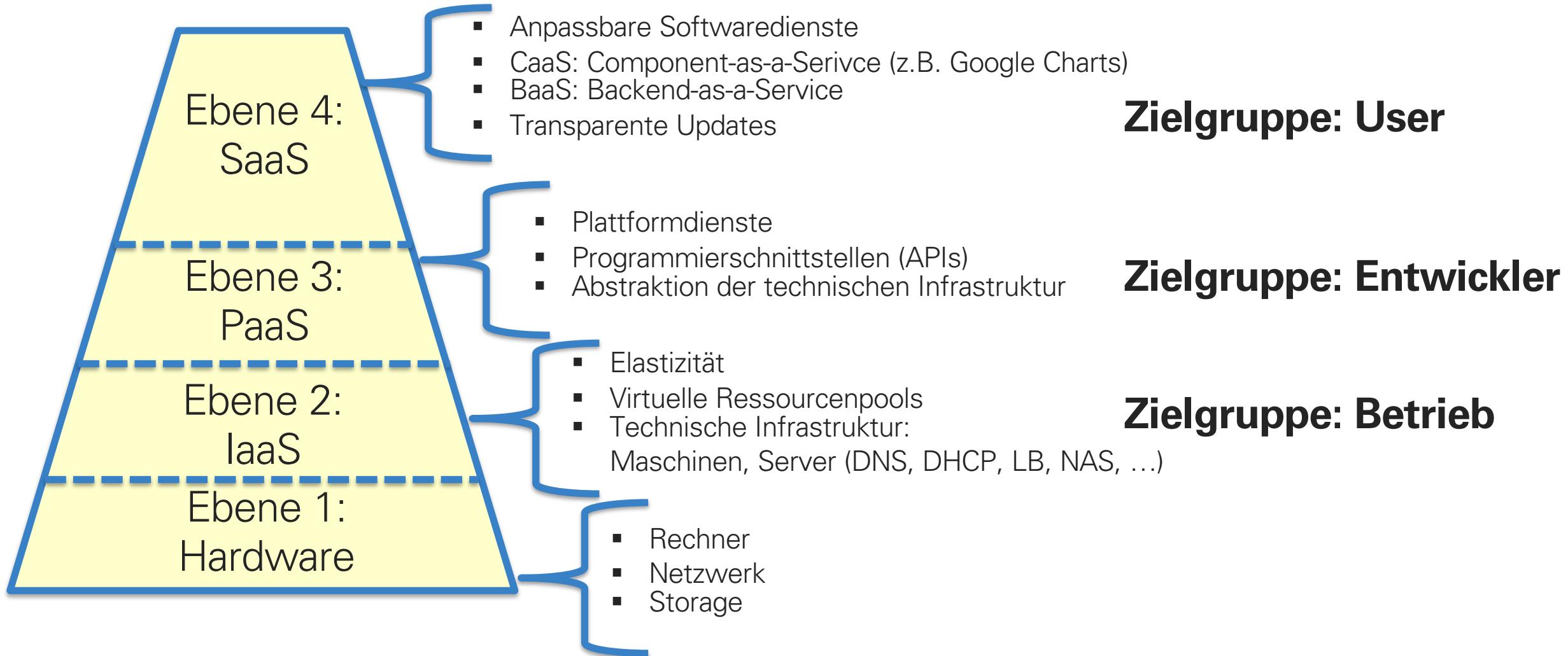
The screenshot shows the New York Times homepage. In the upper right corner, there is a purple "Y!" logo followed by the text "NY Times". Below this, a sidebar contains a historical advertisement for a computer examination. The ad reads:  
A COMPUTER WANTED.  
WASHINGTON, May 1.—A civil service examination will be held May 18 in Washington, and, if necessary, in other cities, to secure eligibles for the position of computer in the Nautical Almanac Office, where two vacancies exist—one at \$1,000, the other at \$1,400. The examination will include the subjects of algebra, geometry, trigonometry, and astronomy. Application blanks may be obtained of the United States Civil Service Commission.  
Published 1892, copyright New York Times

<http://www.slideshare.net/acarlos1000/hadoop-basics-presentation<<>

# Cloud Computing ist keine Überraschung, sondern auf den Schultern von Giganten entstanden.



# Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.

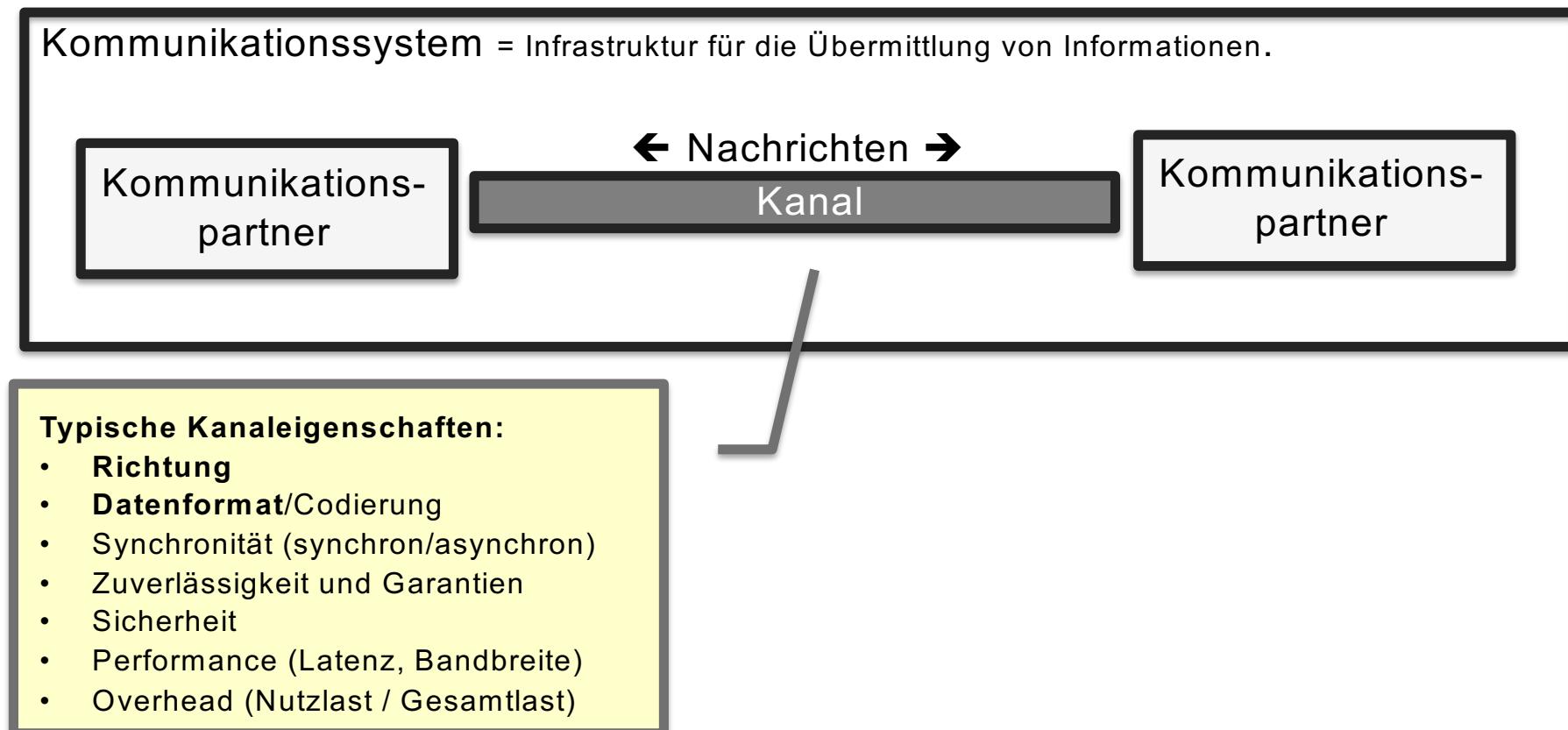


# Mögliche Klausurfragen

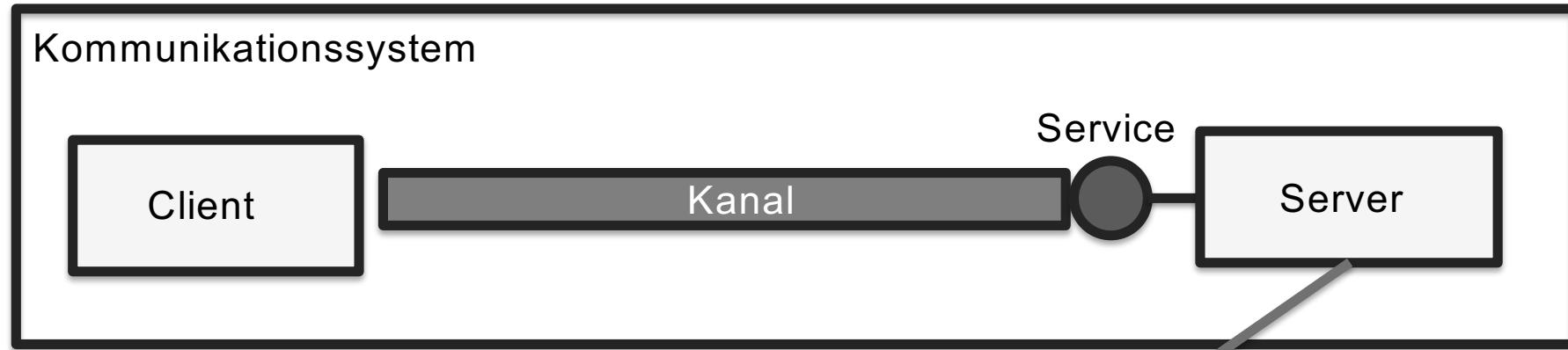
- Was versteht man im Cloud Computing unter Ressourcen Pools?
- Skizzieren und erläutern sie das Schichtenmodell des Cloud Computing. Wer sind die Zielgruppen der jeweiligen Schicht?
- Was sind zwei mögliche Gründe für Unternehmen, einen Multi-Cloud-Ansatz zu nutzen? Nenne mindestens zwei Gründe.

# Kapitel 1: Kommunikation

# Ein allgemeines Kommunikationsmodell im Internet. Angelehnt an das Modell von Shannon/Weaver.



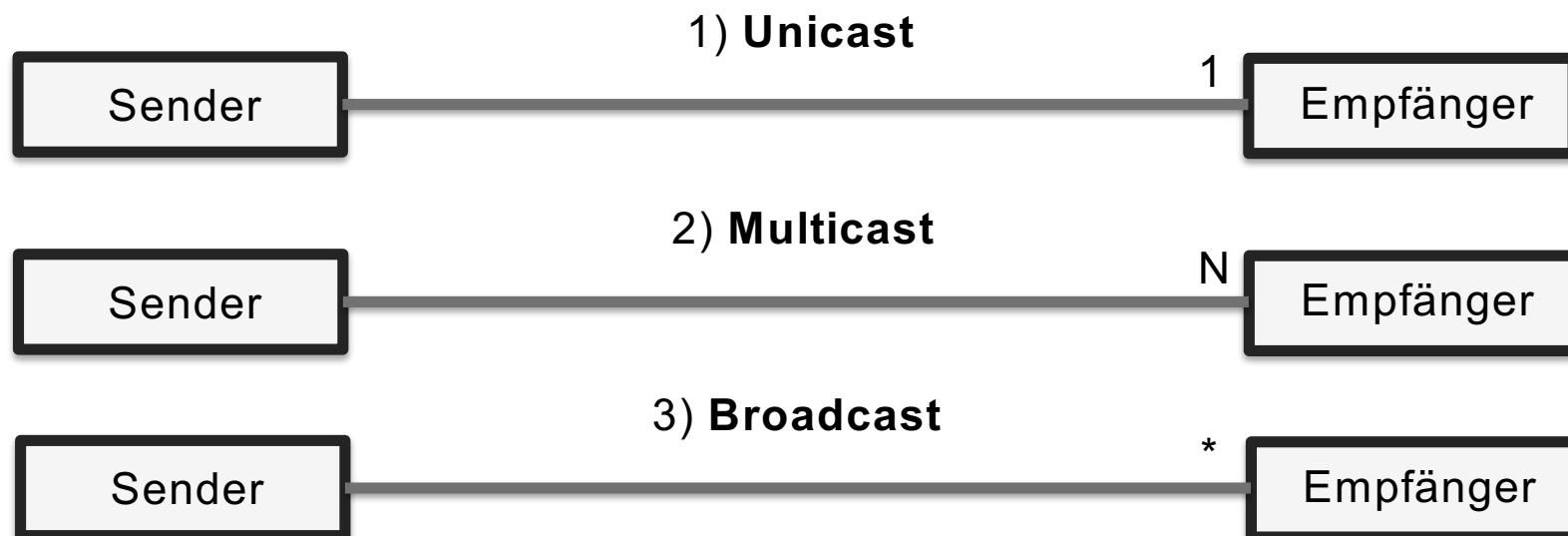
# Service-Orientierung in einem Kommunikationssystem.



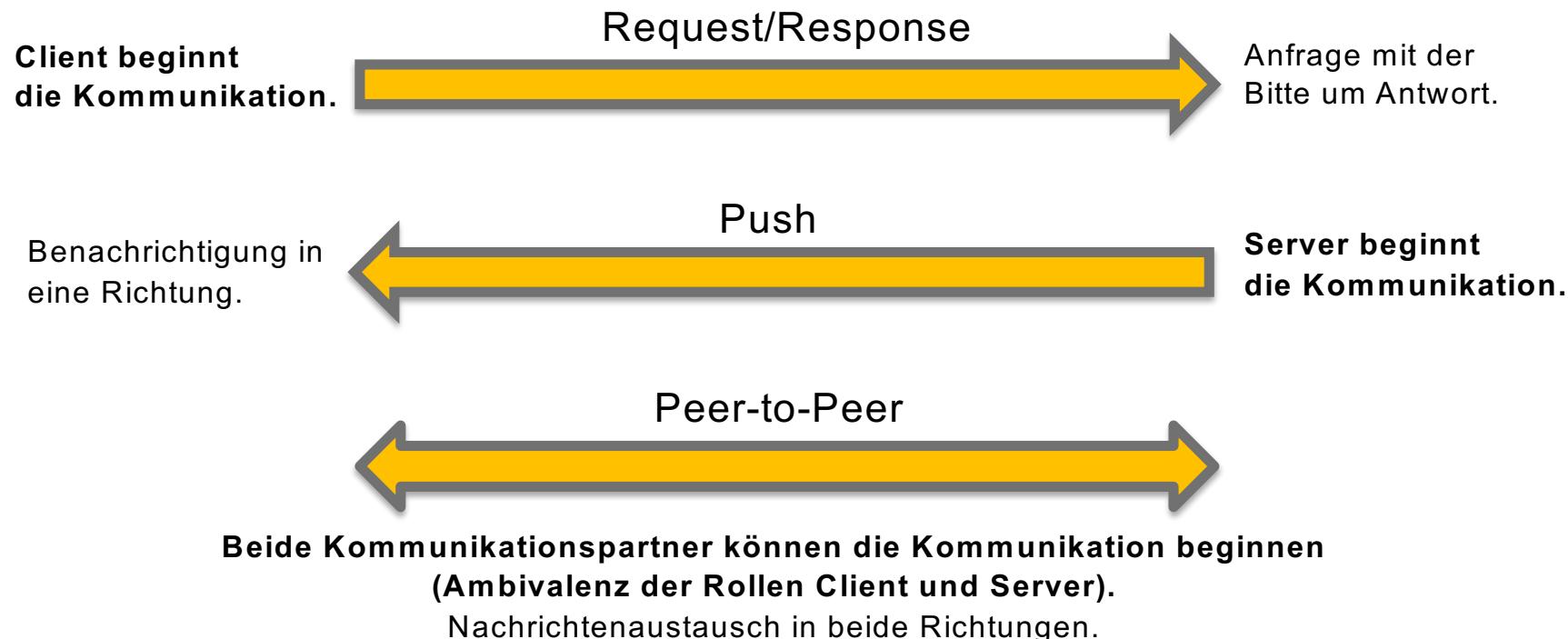
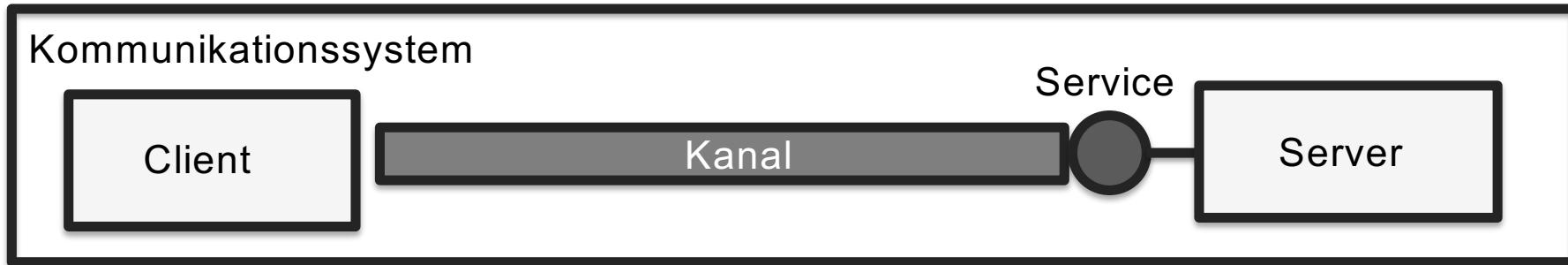
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung stellt. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

Eine **Serviceschnittstelle** ist ein Vertrag zwischen Nutzer und Anbieter über Syntax und Semantik der Service-Nutzung und enthält optional Zusicherungen in Hinblick auf den **Quality of Service**.

# Klassifikation von Kommunikationssystemen: Kardinalität der Empfänger einer Nachricht.



# Klassifikation von Kommunikationssystemen: (B) Wer beginnt mit der Kommunikation?



# REST ist ein Paradigma für Anwendungsservices auf Basis des HTTP-Protokolls.

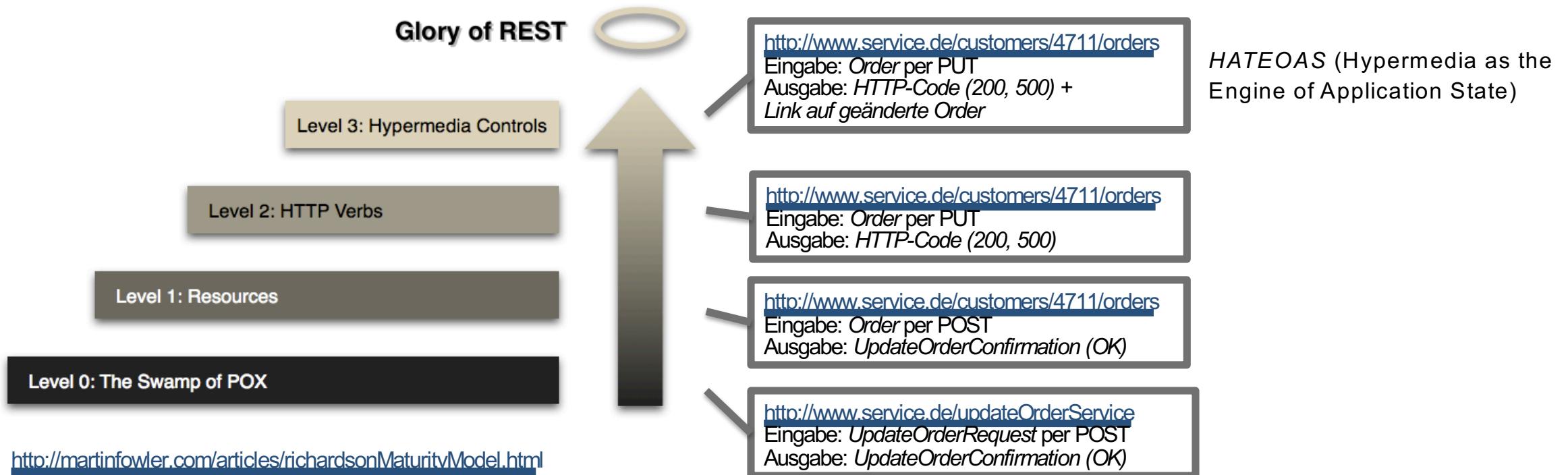
- REST ist eine Paradigma für den Schnittstellenentwurf von Internetanwendungen auf Basis des HTTP-Protokolls.
- Dissertation von Roy Fielding: „Architectural Styles and the Design of Network-based Software Architectures“, 2000, University of California, Irvine.

## Grundlegende Eigenschaften:

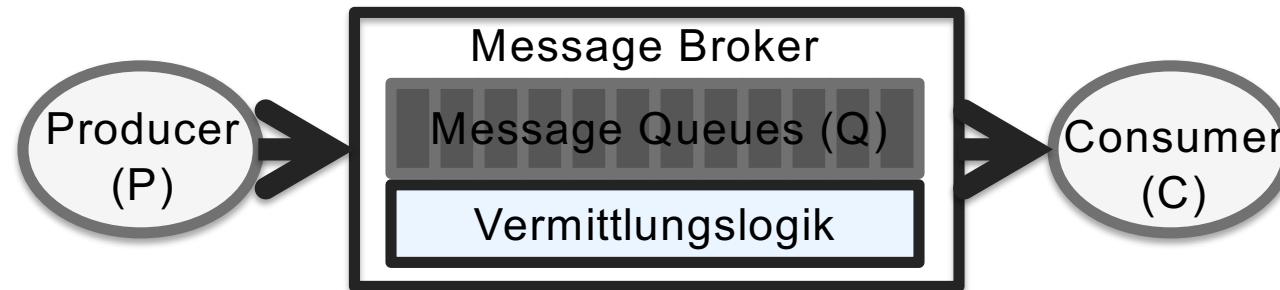
- **Alles ist eine Ressource:** Eine Ressource ist eindeutig adressierbar über einen URI, hat eine oder mehrere Repräsentationen (XML, JSON, bel. MIME-Typ) und kann per Hyperlink auf andere Ressourcen verweisen. Ressourcen sind, wo immer möglich, hierarchisch navigierbar.
- **Uniforme Schnittstellen:** Services auf Basis der HTTP-Methoden (PUT = erzeugen, POST = aktualisieren oder erzeugen, DELETE = löschen, GET = abfragen). Fehler werden über die HTTP Codes zurückgemeldet. Services haben somit eine standardisierte Semantik und eine stabile Syntax.
- **Zustandslosigkeit:** Die Kommunikation zwischen Server und Client ist zustandslos. Ein Zustand wird im Client nur durch URLs gehalten.
- **Konnektivität:** Basiert auf ausgereifter und allgegenwärtiger Infrastruktur: Der Web-Infrastruktur mit wirkungsvollen Caching- und Sicherheitsmechanismen, leistungsfähigen Servern und z.B. Web-Browser als Clients.



# Mit dem REST Maturity Model kann bewertet werden, wie RESTful ein HTTP-basierter Service ist.



# Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch.



## Entkopplung von Producer und Consumer.

Die Serviceschnittstelle ist lediglich das Format der Nachricht. Message Broker machen zum Format keinen Einschränkungen. Sende-Zeitpunkt und Empfangs-Zeitpunkt können beliebig lange auseinander liegen.

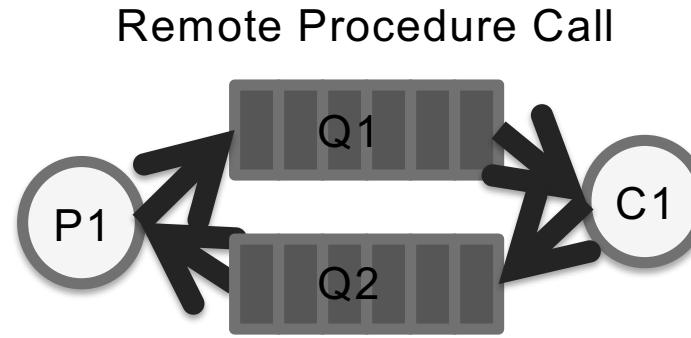
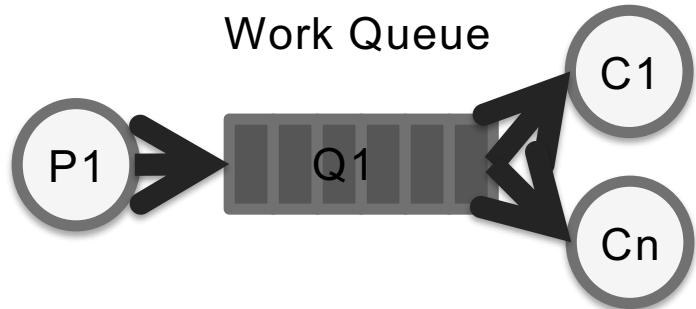
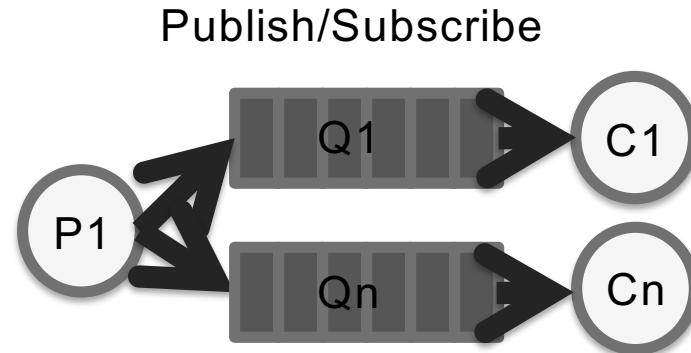
**Skalierbarkeit.** Die Vermittlungslogik entscheidet zentral ...

- ... an wie viele Consumer die Nachricht ausgeliefert wird (horizontale Skalierbarkeit),
- an welchen Consumer die Nachricht ausgeliefert wird (Lastverteilung),
- wann eine Nachricht ausgeliefert wird (Pufferung von Lastspitzen),

auf Basis von konfigurierten Anforderungen an die Vermittlung:

- Maximale Zustelldauer bzw. Lebenszeit der Nachricht
- Geforderte Zustellgarantie (mindestens 1 Mal, exakt 1 Mal, an alle) und Transaktionalität
- Priorität der Nachricht
- Notwendige Einhaltung der Zustellreihenfolge

Messaging ist eine flexible Kommunikationsart, mit der sich vielfältige Kommunikationsmuster umsetzen lassen.

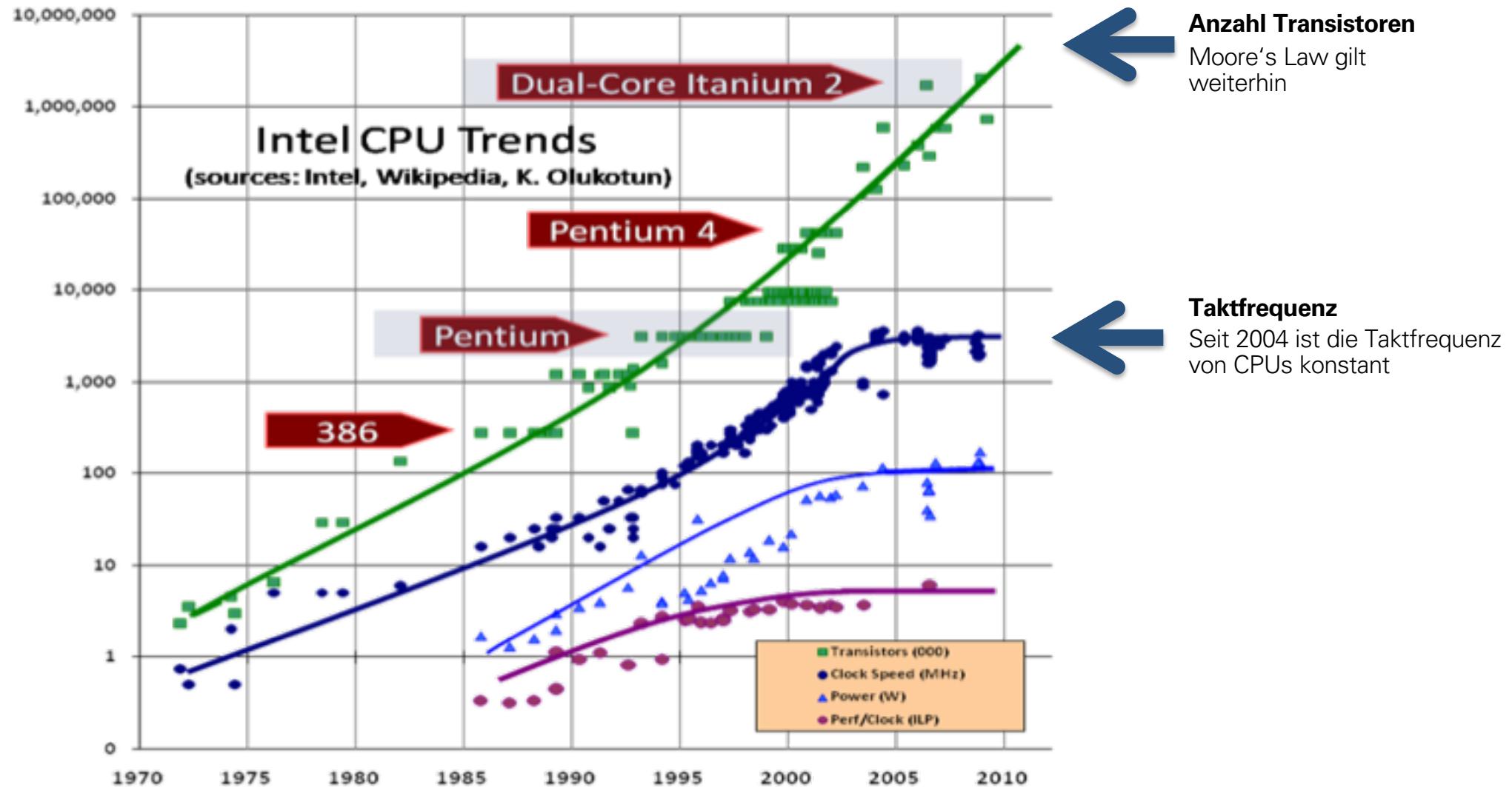


# Mögliche Klausurfragen

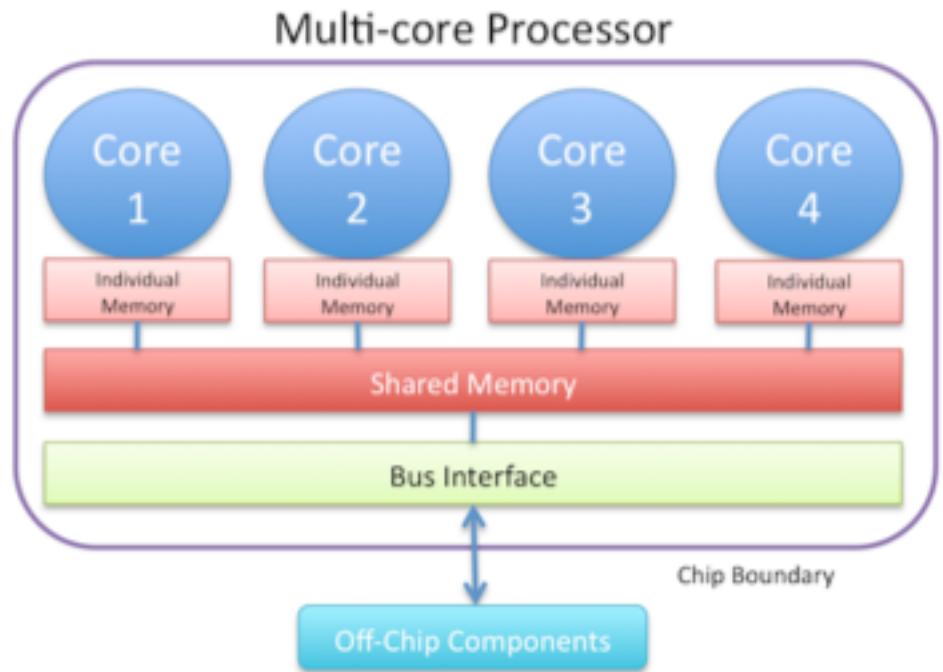
- Was ist Push-Kommunikation?
- Was versteht man bei der Internet-Kommunikation unter einem Service?
- Nennen sie die 4 grundlegenden Eigenschaften des REST Paradigmas.
- Stellen sie das Messaging-Kommunikationsmuster *Publish/Subscribe* grafisch dar und beschreiben sie einen typischen Anwendungsfall für dieses Muster.

# Kapitel 2: Programmiermodelle

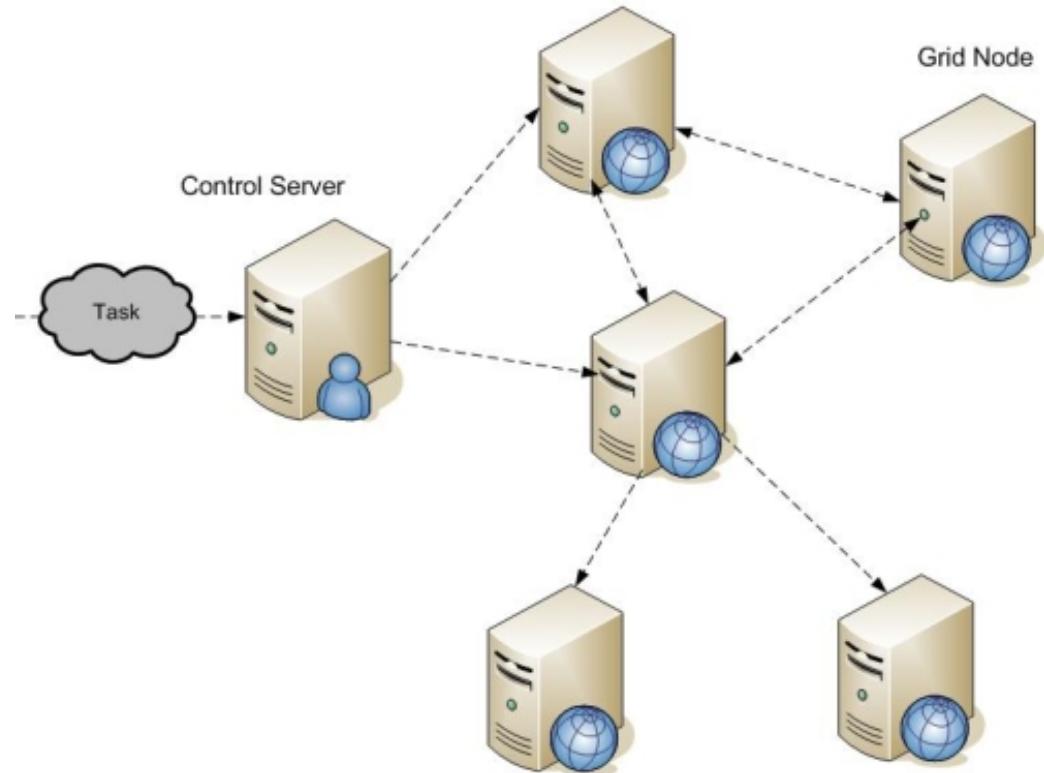
# „The free lunch is over“: Es gibt keine kostenlose Performancesteigerung mehr – Nebenläufigkeit zählt



# Nebenläufigkeit kann im Kleinen und im Großen betrieben werden

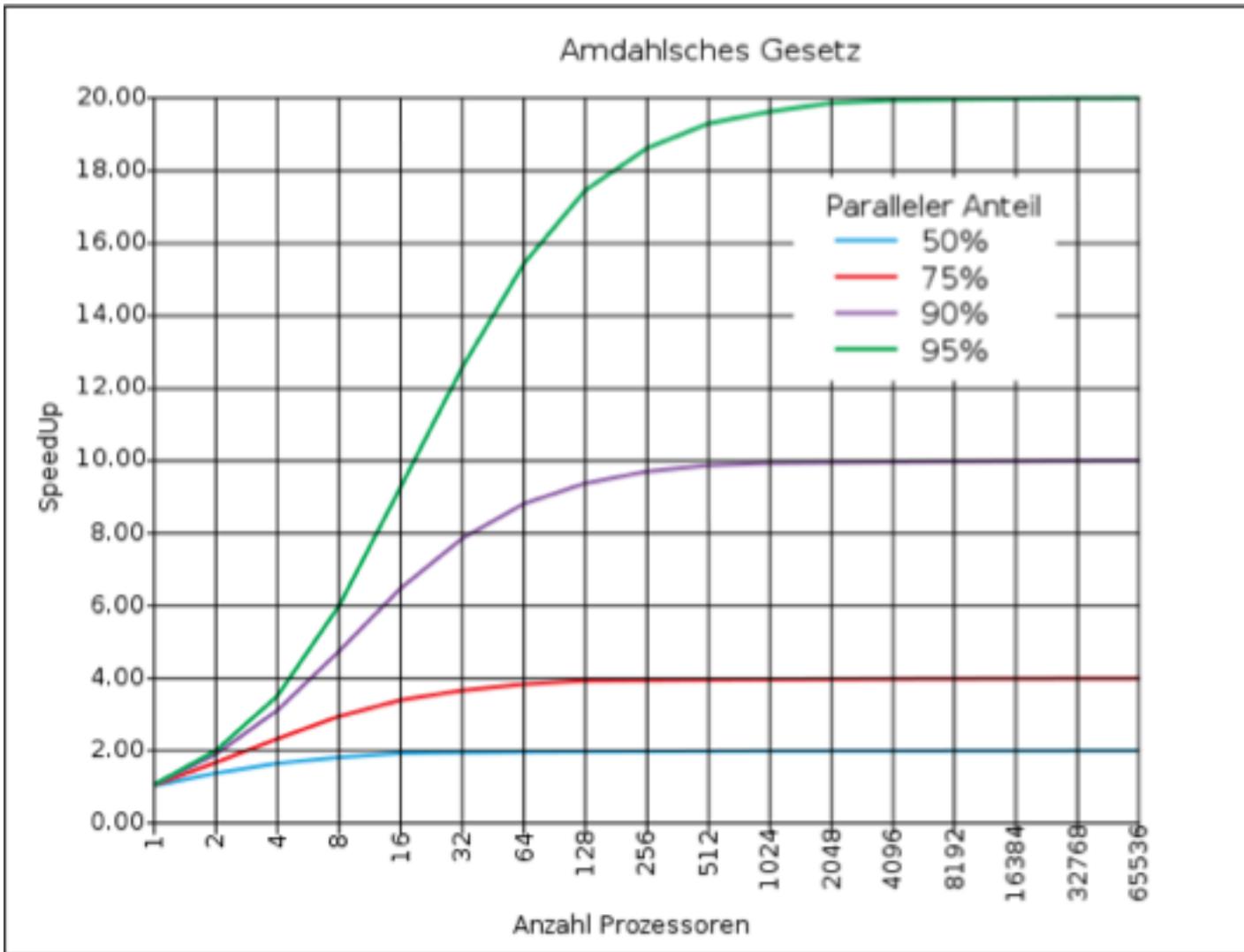


Multi Core



Multi Node  
(Cluster, Grid, Cloud)

# Das Amdahlsche Gesetz: Die Grenzen der Performance-Steigerung über Nebenläufigkeit



**P** = Paralleler Anteil

**S** = Sequenzieller Anteil

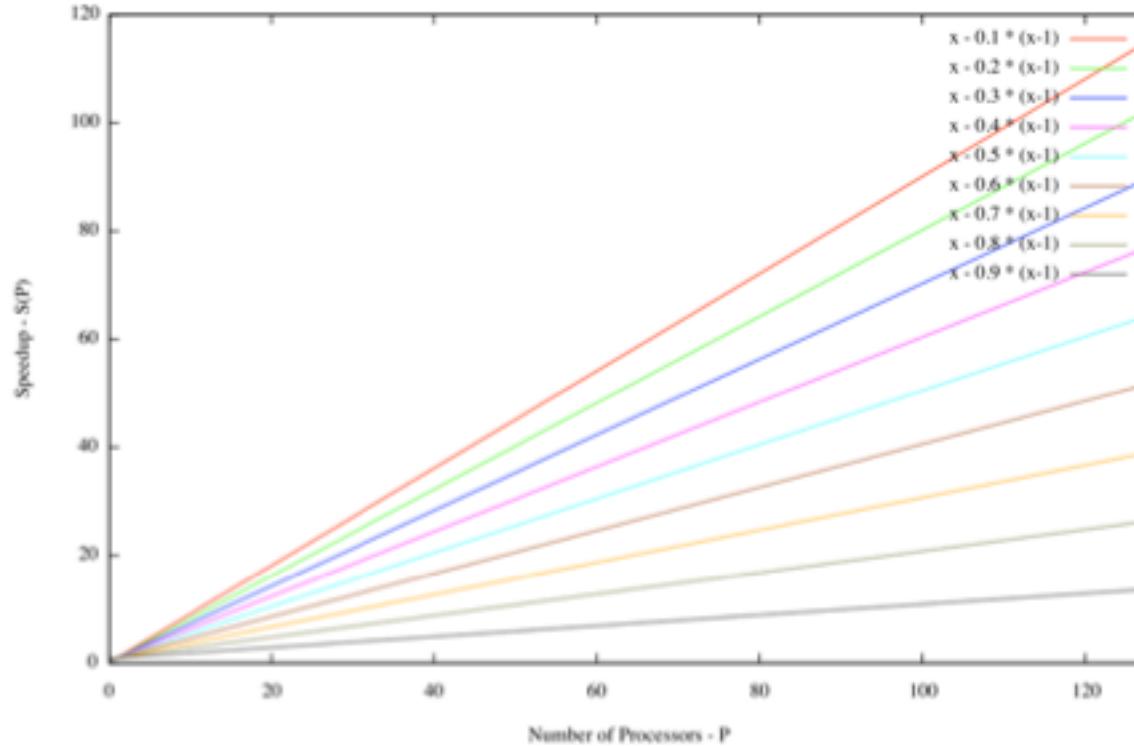
**N** = Anzahl der Prozessoren

**Speedup** = Maximale Beschleunigung

$$\text{Speedup} = \frac{1}{1-P} \quad \text{für } N = \infty$$

$$\text{Speedup} = \frac{1}{\frac{P}{N} + S}$$

# Gustafsons Gesetz: Ist bei großen Datenmengen jedoch oft passender



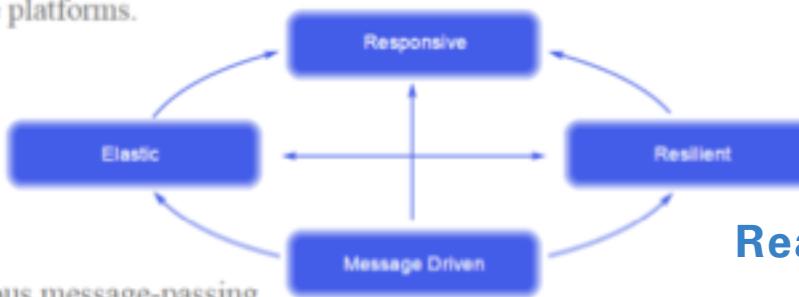
$$\text{Speedup} = \frac{1}{\frac{P}{N} + \cancel{x}}$$

- **Annahme:** Der parallele Anteil P ist linear abhängig von der Problemgröße (i.W. der Datenmenge), der sequenzielle Anteil hingegen nicht.
- Beispiel: Mehr Bilder → Mehr parallele Konvertierung
- Gesetz: Steigt der parallele Anteil P mit der Problemgröße, so wächst auch der Speedup linear

# Das Reactive Manifesto

## React to load

**Elastic:** The system stays responsive under varying workload. Reactive Systems can react to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. This implies designs that have no contention points or central bottlenecks, resulting in the ability to shard or replicate components and distribute inputs among them. Reactive Systems support predictive, as well as Reactive, scaling algorithms by providing relevant live performance measures. They achieve elasticity in a cost-effective way on commodity hardware and software platforms.



## React to events / messages

**Message Driven:** Reactive Systems rely on asynchronous message-passing to establish a boundary between components that ensures loose coupling, isolation, location transparency, and provides the means to delegate errors as messages. Employing explicit message-passing enables load management, elasticity, and flow control by shaping and monitoring the message queues in the system and applying back-pressure when necessary. Location transparent messaging as a means of communication makes it possible for the management of failure to work with the same constructs and semantics across a cluster or within a single host. Non-blocking communication allows recipients to only consume resources while active, leading to less system overhead.

## React to users

**Responsive:** The system responds in a timely manner if at all possible. Responsiveness is the cornerstone of usability and utility, but more than that, responsiveness means that problems may be detected quickly and dealt with effectively. Responsive systems focus on providing rapid and consistent response times, establishing reliable upper bounds so they deliver a consistent quality of service. This consistent behaviour in turn simplifies error handling, builds end user confidence, and encourages further interaction.

## React to failures

**Resilient:** The system stays responsive in the face of failure. This applies not only to highly-available, mission critical systems — any system that is not resilient will be unresponsive after a failure. Resilience is achieved by replication, containment, isolation and delegation. Failures are contained within each component, isolating components from each other and thereby ensuring that parts of the system can fail and recover without compromising the system as a whole. Recovery of each component is delegated to another (external) component and high-availability is ensured by replication where necessary. The client of a component is not burdened with handling its failures.

# Reactive Programming: Das Programmiermodell mit dem das Reactive Manifesto umgesetzt werden kann

## Dekomposition in Funktionen (auch **Aktoren**)

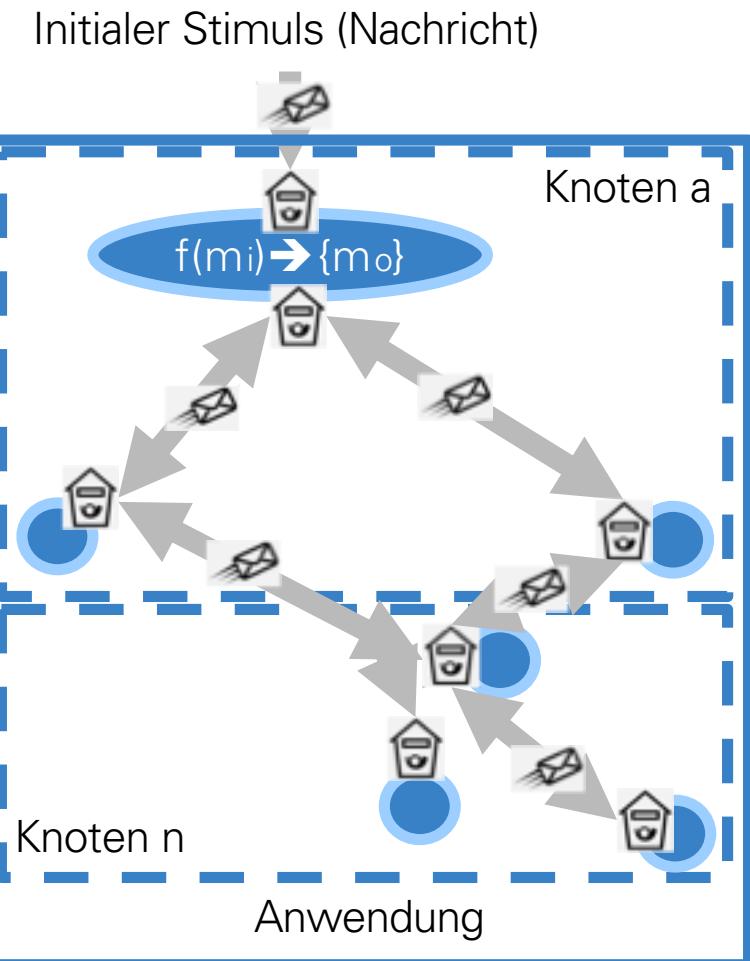
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wiederholbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)

## Kommunikation zwischen den Funktionen über Nachrichten

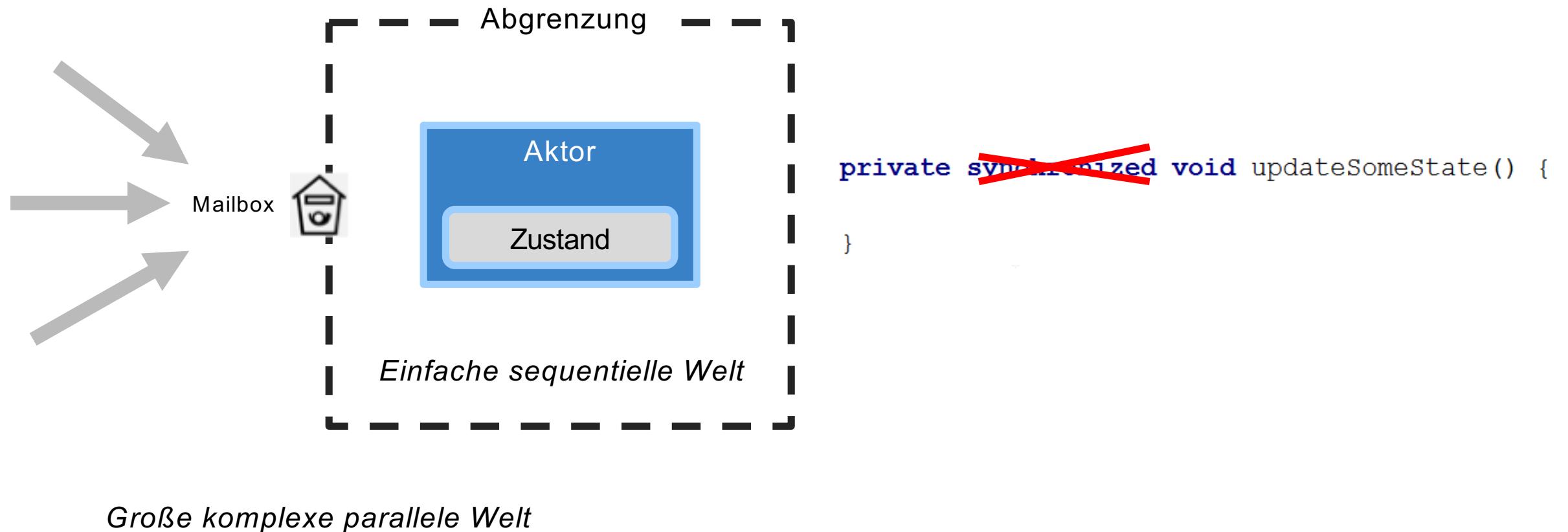
- asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber nicht auf sie.
- Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Produzenten und 1 Consumer)
- Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und sind unveränderbar

## Elastischer Kommunikationskanal

- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden (mehr oder minder) zuverlässig zugestellt
- Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)



Ein einzelner Aktor ist ein einfaches single-threaded Objekt, das über die Mailbox synchronisiert wird.

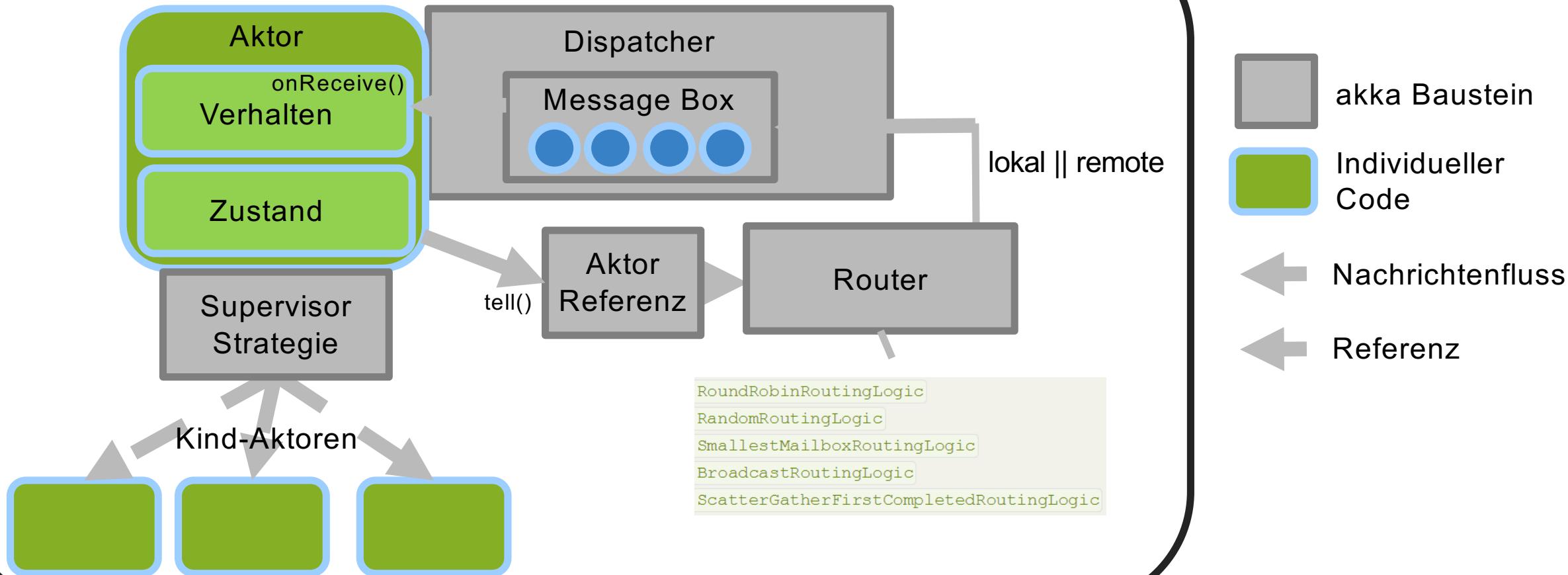


*Große komplexe parallele Welt*

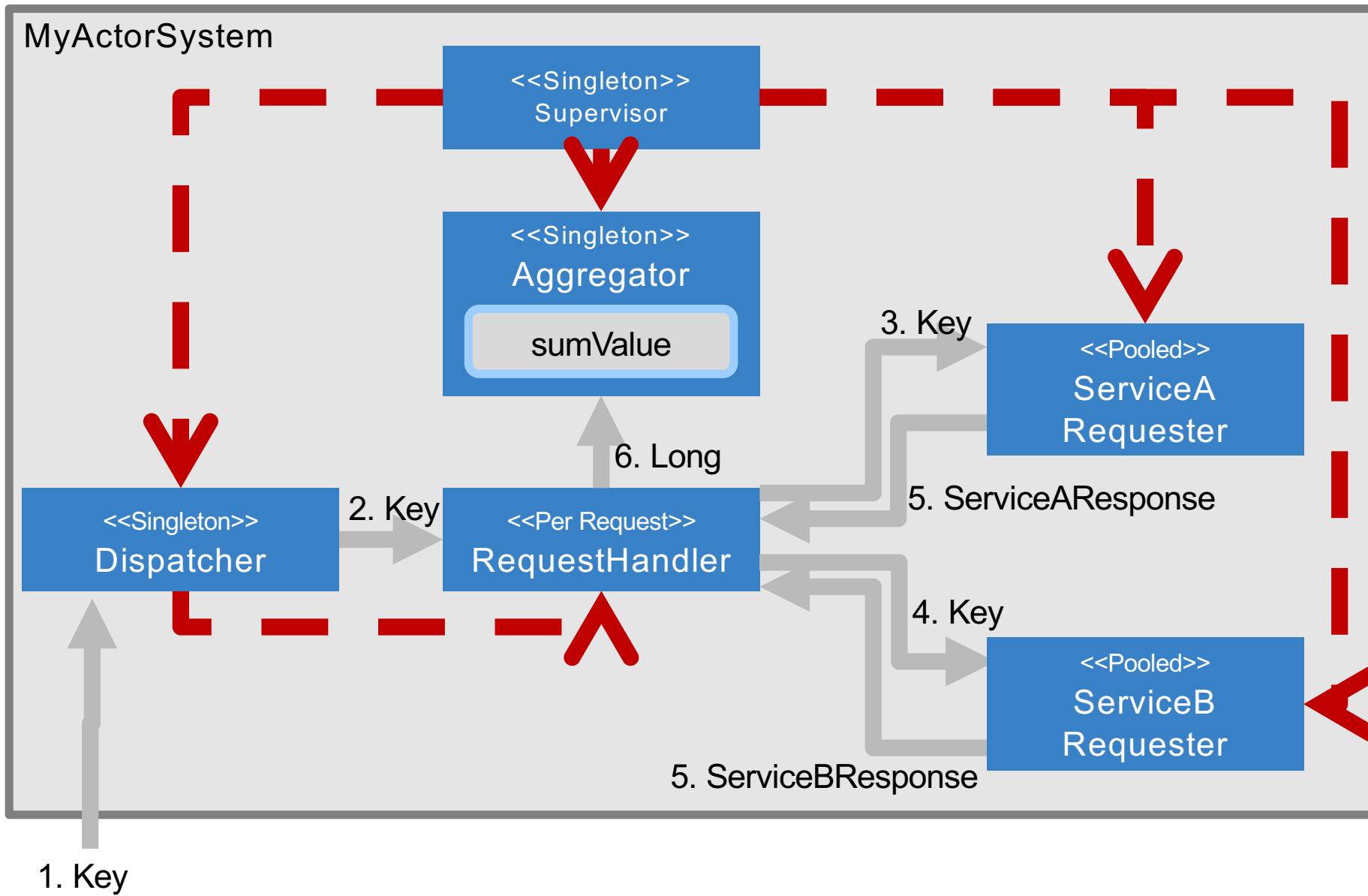
# Die Grundkonzepte von akka

**Konfiguration Aktor** = Aktor-Klasse + Aktor-Name + Supervisor-Strategie + Dispatcher + Lokalität  
**Konfiguration Aktor Referenz** = Aktor-Name + Router

**Aktorenystem:** Kennt die Akteure und ihre Konfigurationen



# Ein Aktorensystem als Bild



Legende:

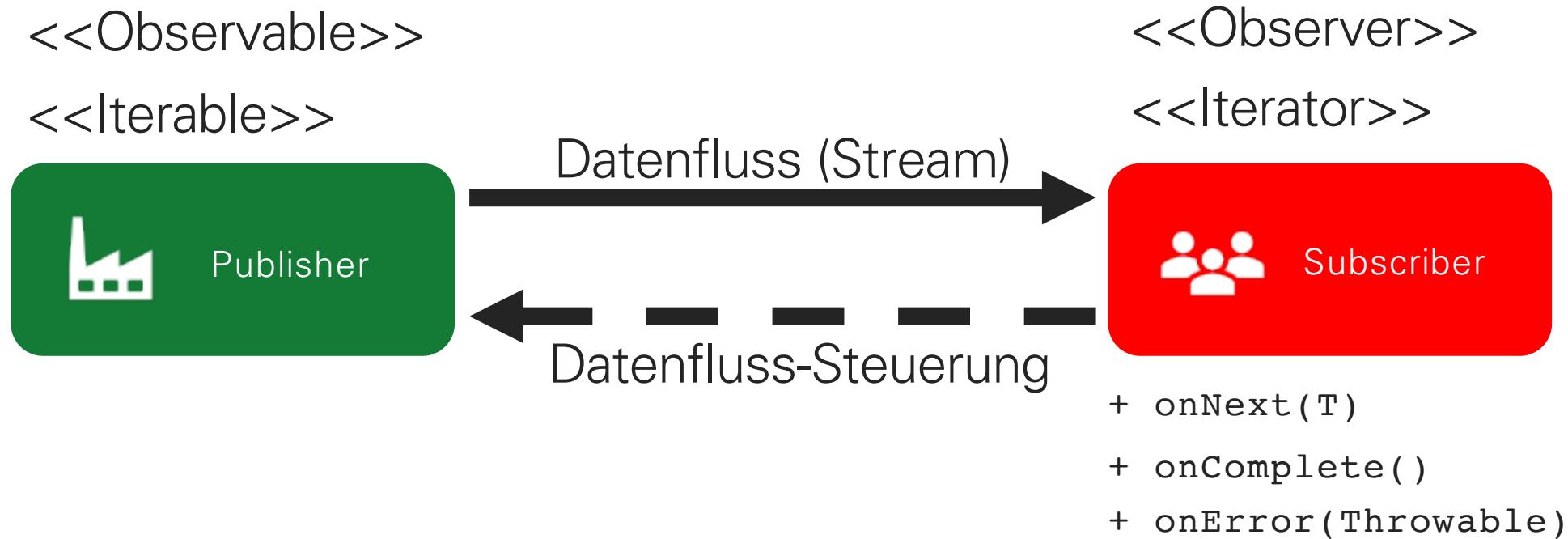
`<<Kardinalität>>`  
Aktor

Reihenfolge.  
Nachrichten-Typ  
Nachrichtenfluss

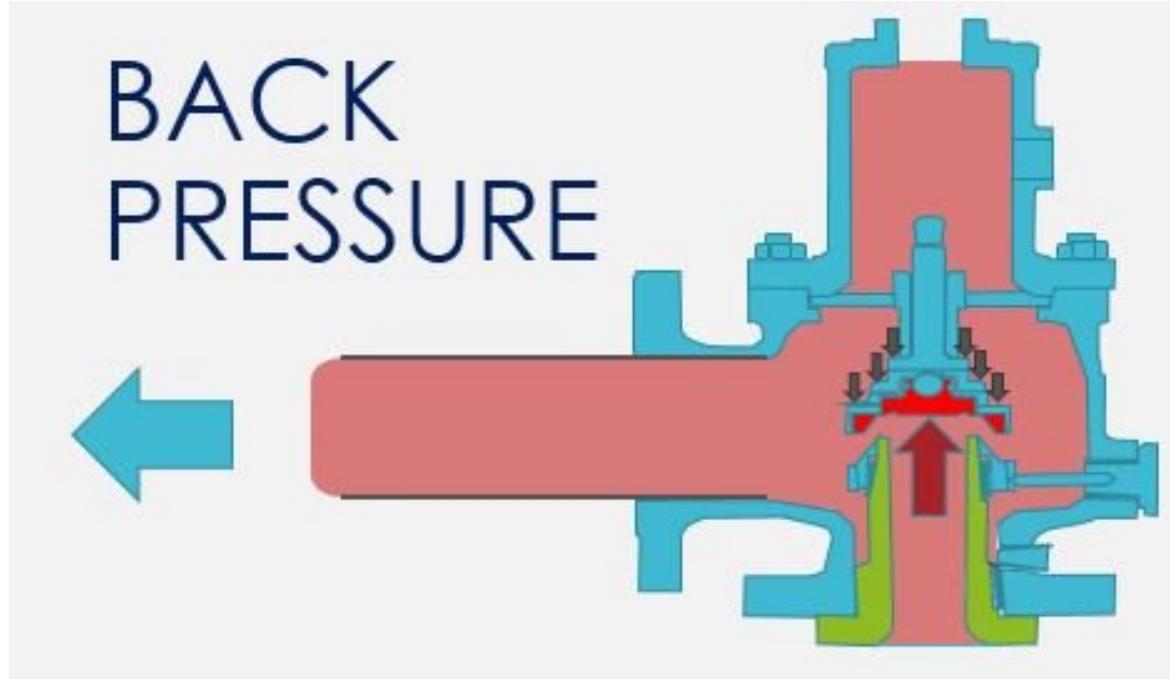
Bei Bedarf erweitert um  
Informationen zur  
Kommunikationsart (lokal,  
entfernt) und dem Routing.

Parent → Child  
Aktorenhierarchie  
(Supervision)

# Reactive Streams: Das Fundament ist eine Kombination aus Observer- und Iterator-Pattern.

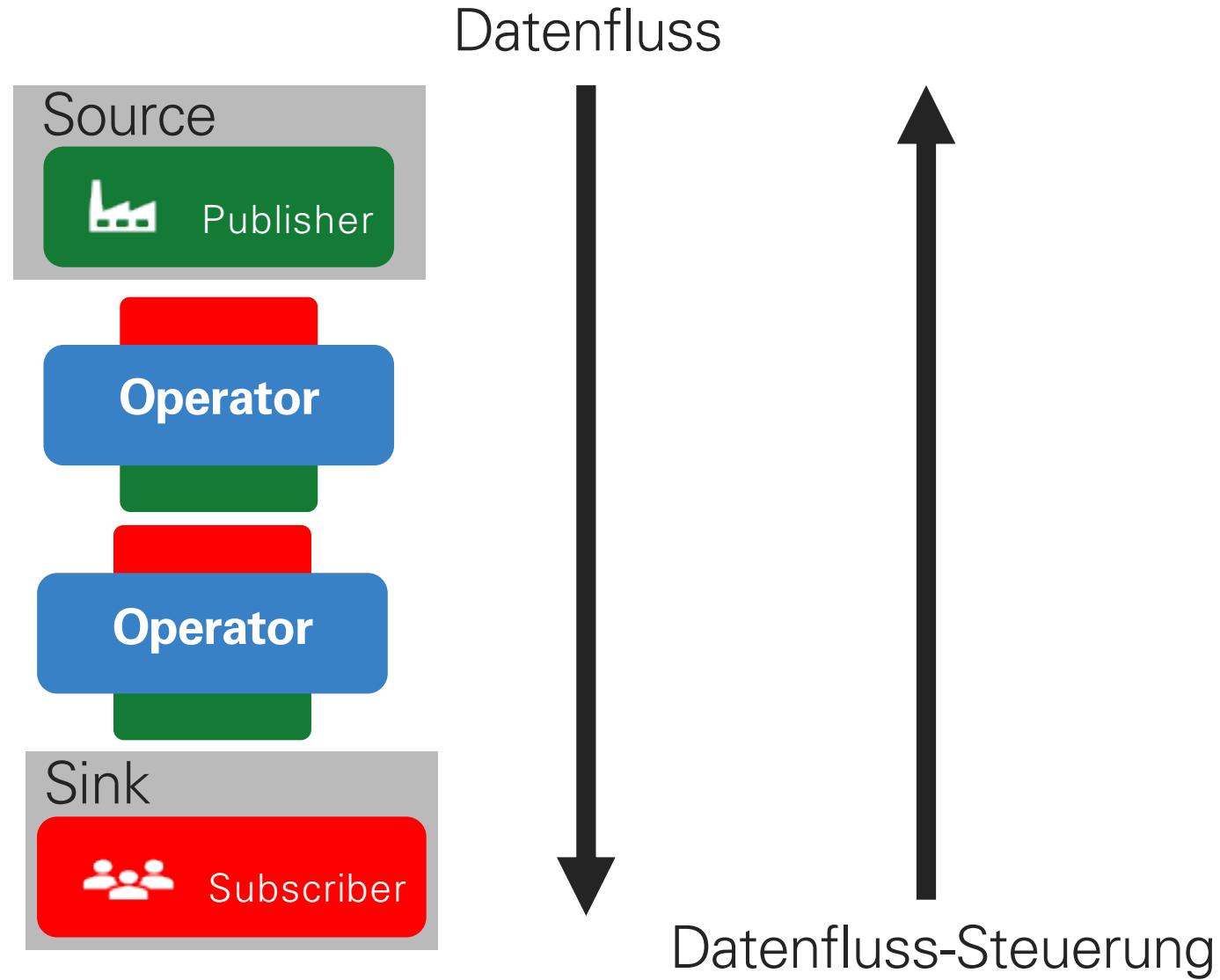


# Back Pressure: Umgang mit Überlast



# Reactive Streams: Das Programmiermodell

z.B. „lese zeilenweise von Datenbank“



z.B. „sende AMQP-Nachrichten an Client“

# Bewertung

## Vorteile

- Höherer Durchsatz bei IO-intensiven Anwendungen
  - Der Prozessor befindet sich weniger Zeit in Wartezuständen (IO-Wait, Lock-Wait, ...)
  - Der Hauptspeicherverbrauch ist niedriger durch häppchenweise Streams
  - Die CPU und der Hauptspeicher sind für weiteren Durchsatz frei
- Toleranteres Verhalten im Hochlastbereich
  - Back Pressure
  - Leichtgewichtige Skalierungsressourcen (Aktoren, Scheduler)

## Nachteile

- Performance-Einbußen bei CPU-intensiven Anwendungen
- Ungewohntes aber gut lesbares Programmiermodell
- Over-engineered für Anwendungen mit normaler Last und moderatem Durchsatz

# Mögliche Klausurfragen

- Was sind die vier wesentlichen Forderungen des Reactive Manifesto?
- Welche Vorteile bietet FRP gegenüber synchronen Systemen?
- Stellen sie ein Aktorensystem grafisch dar, das folgende Funktion erfüllt
  - Ein Aufrufer übergibt eine Zeichenkette nach der er suchen will
  - Das System stößt parallel zueinander zwei Service-Aufrufe an. Einen an System A, den anderen an System B. Beide Systeme können zu einer Zeichenkette eine Liste an Ergebnissen liefern.
  - Die Liste der Ergebnisse wird zusammengefasst an den Aufrufer übergeben.

# Kapitel 3: Virtualisierung

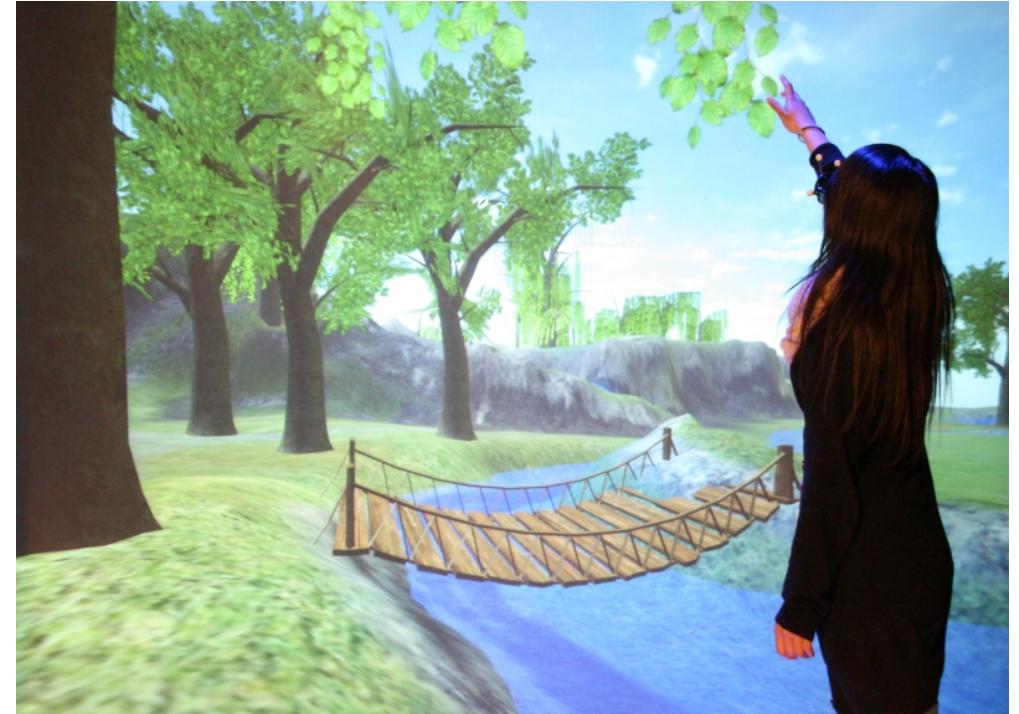


# Virtualisierung

**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 bereitgestellten Software-Defined-Resources.

# 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

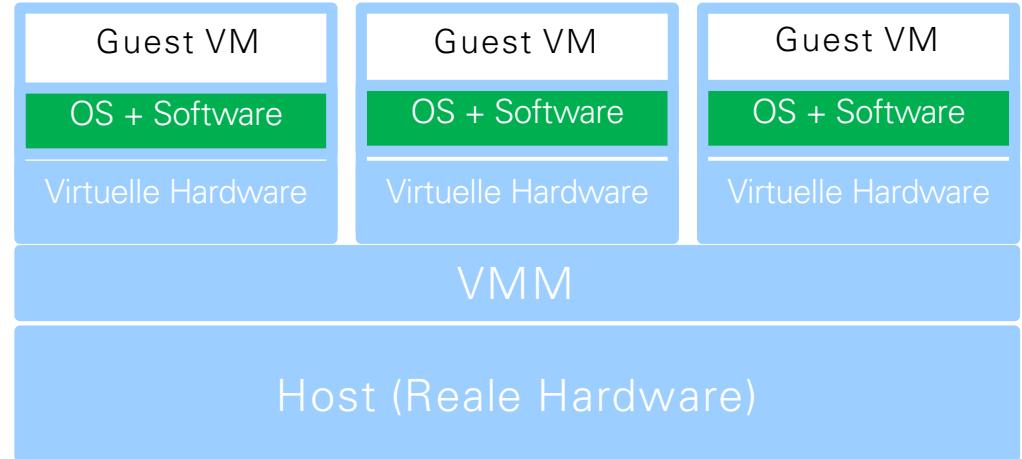
- Virtuelle Netzwerkschnittstellen und virtuelle Netzwerk-Infrastrukturen (VLAN)
- Brücken zwischen virtuellen und realen Netzwerken

## ■ 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

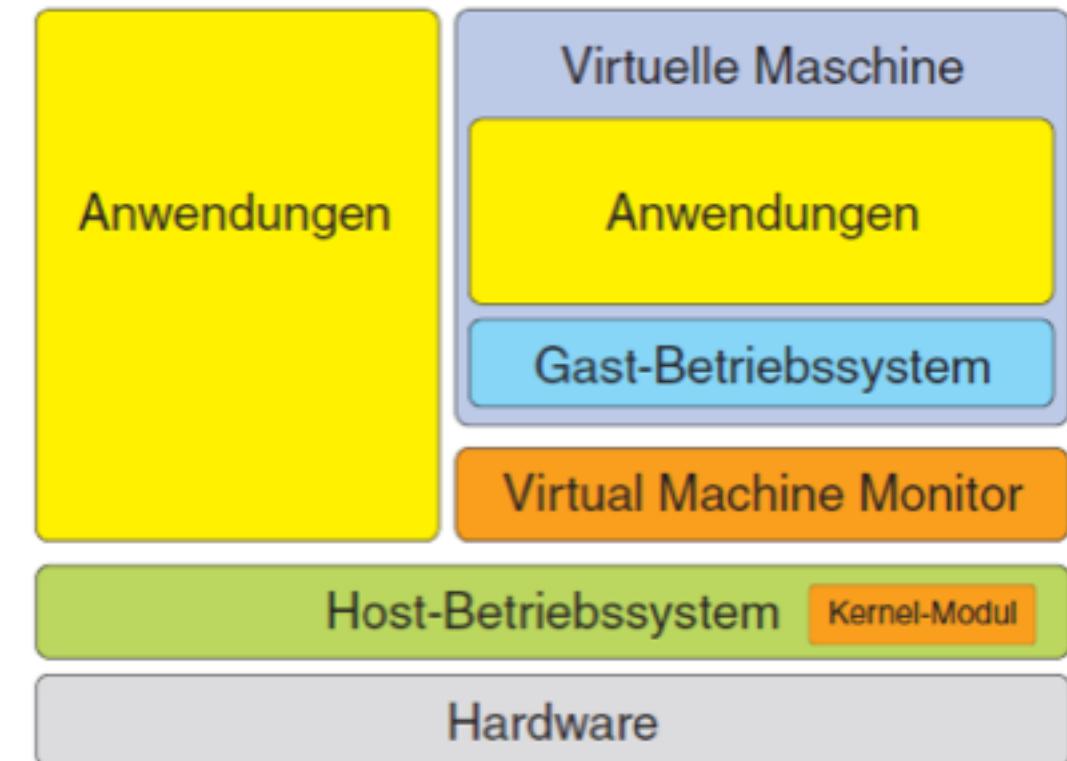
- Eine lauffähige / laufende virtuelle Maschine

## VMM (Virtual Machine Monitor)

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

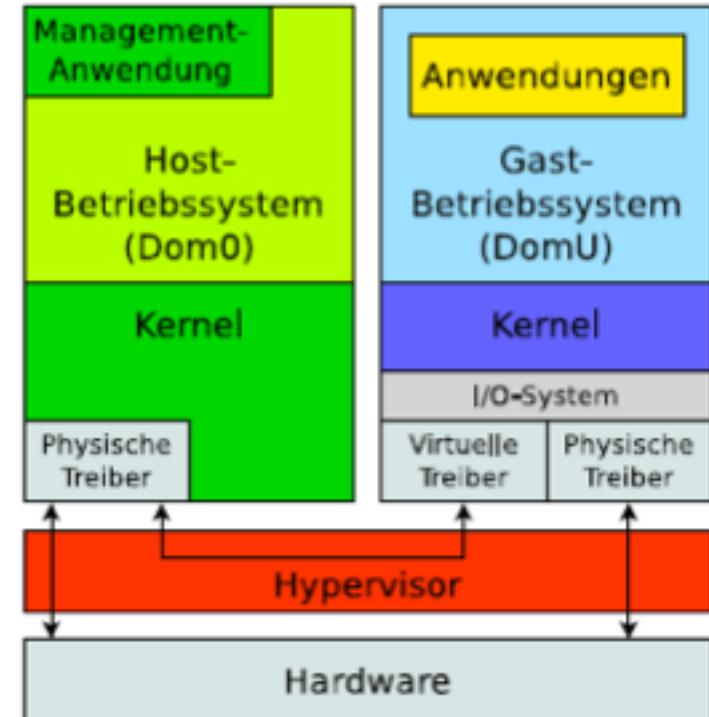
# Hardware-Virtualisierung: Voll-Virtualisierung

- 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% .

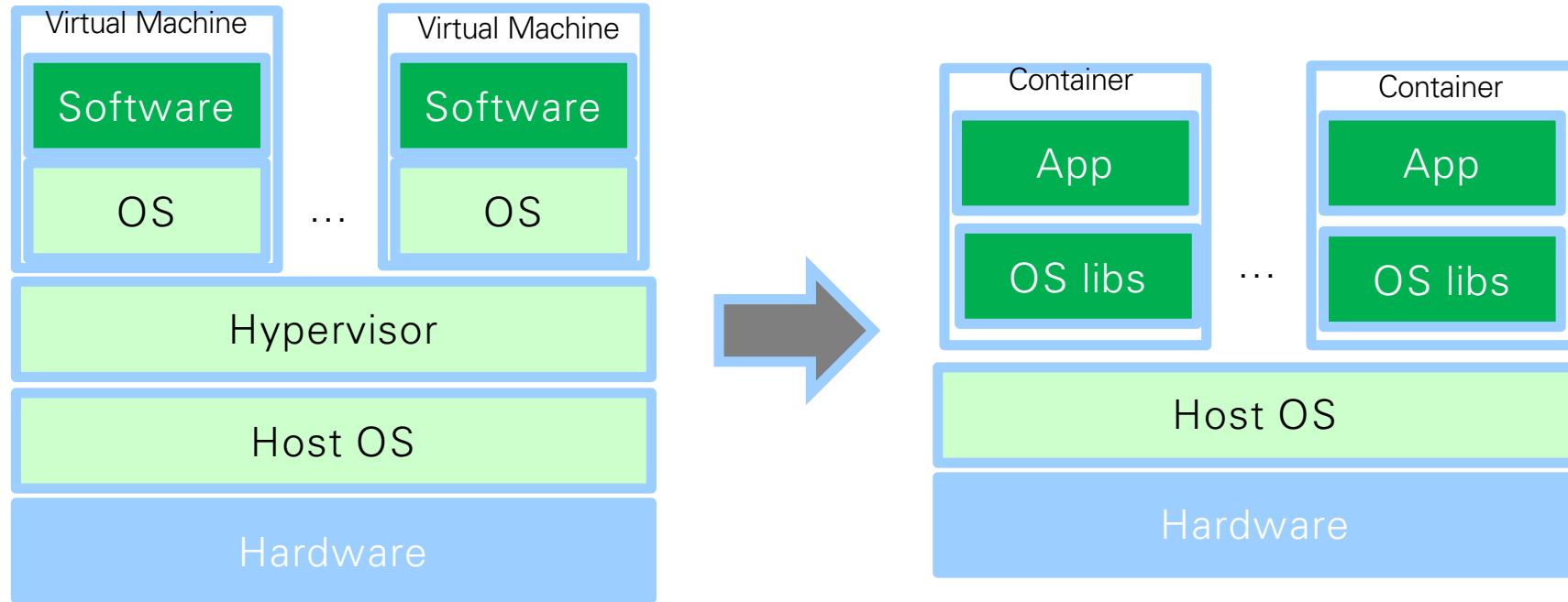


# Hardware-Virtualisierung: Para-Virtualisierung

- 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.
- Leistungsverlust: 2-3%



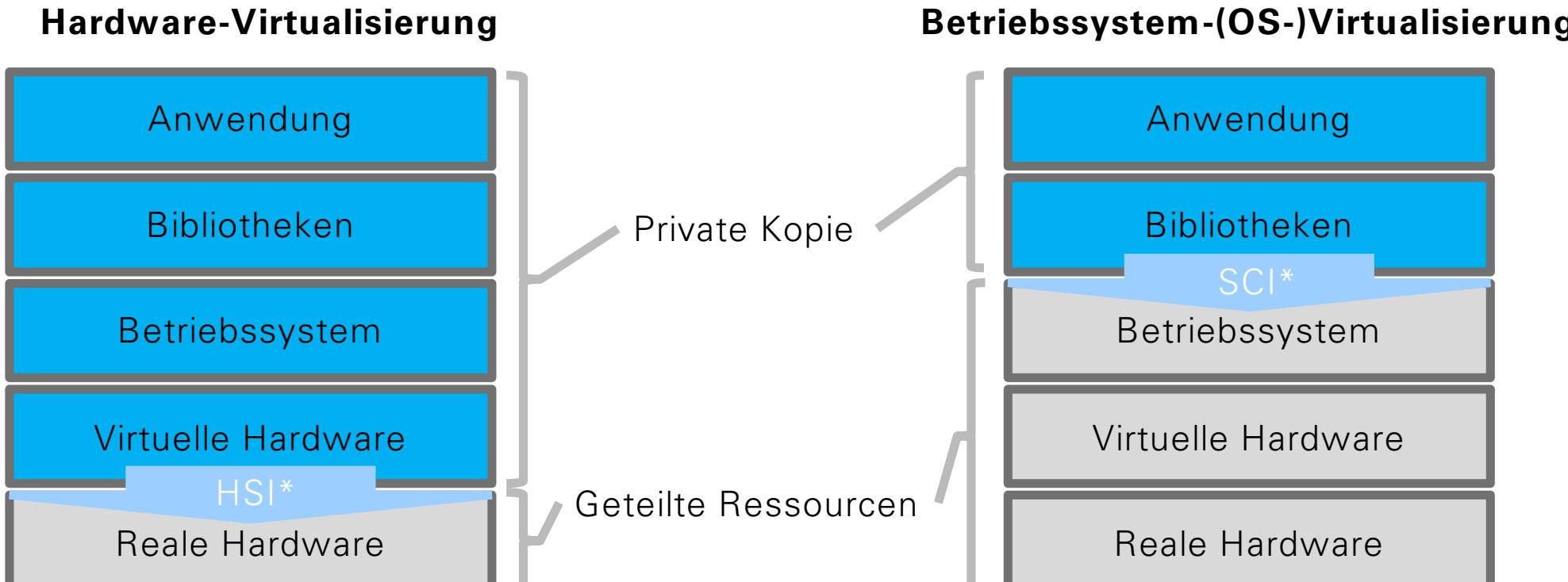
# Betriebssystem-Virtualisierung



- 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

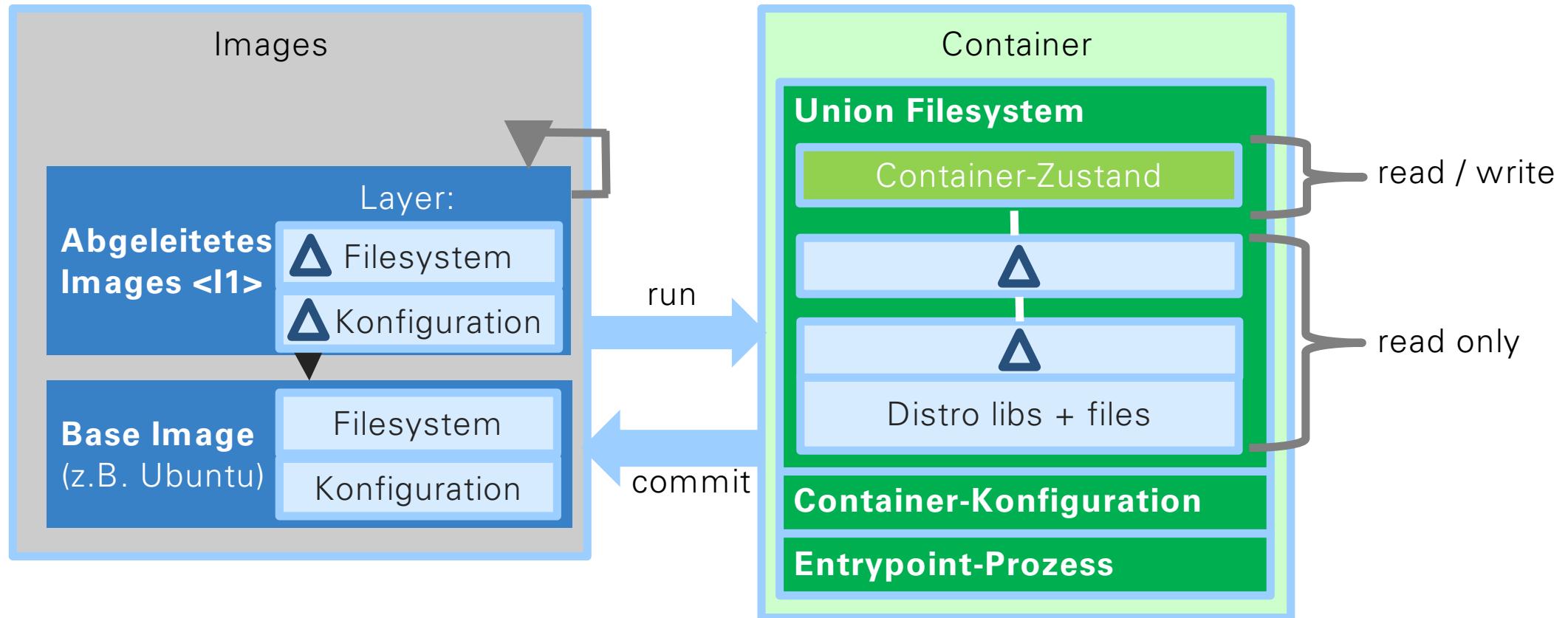
\*) HSI = Hardware Software Interface  
SCI = System Call Interface



- Stärkere Isolation
- Höhere Sicherheit

- Geringeres Volumen der privaten Kopie
- Geringerer Overhead
- Kürzere Startup-Zeit

# Im Zentrum von Docker stehen Images und Container.

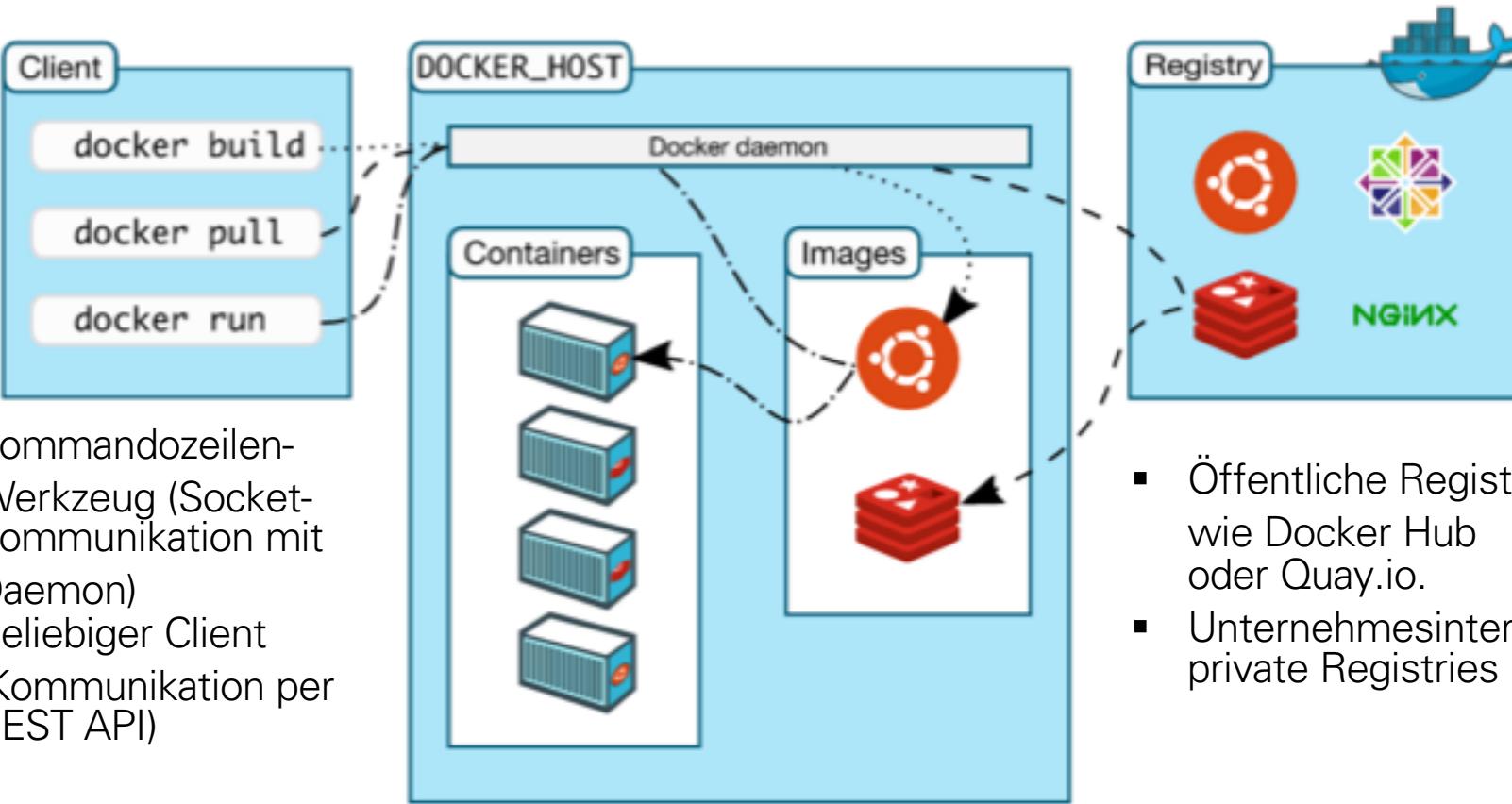


**Ruhender und transportierbarer Zustand**

**Laufender Zustand**

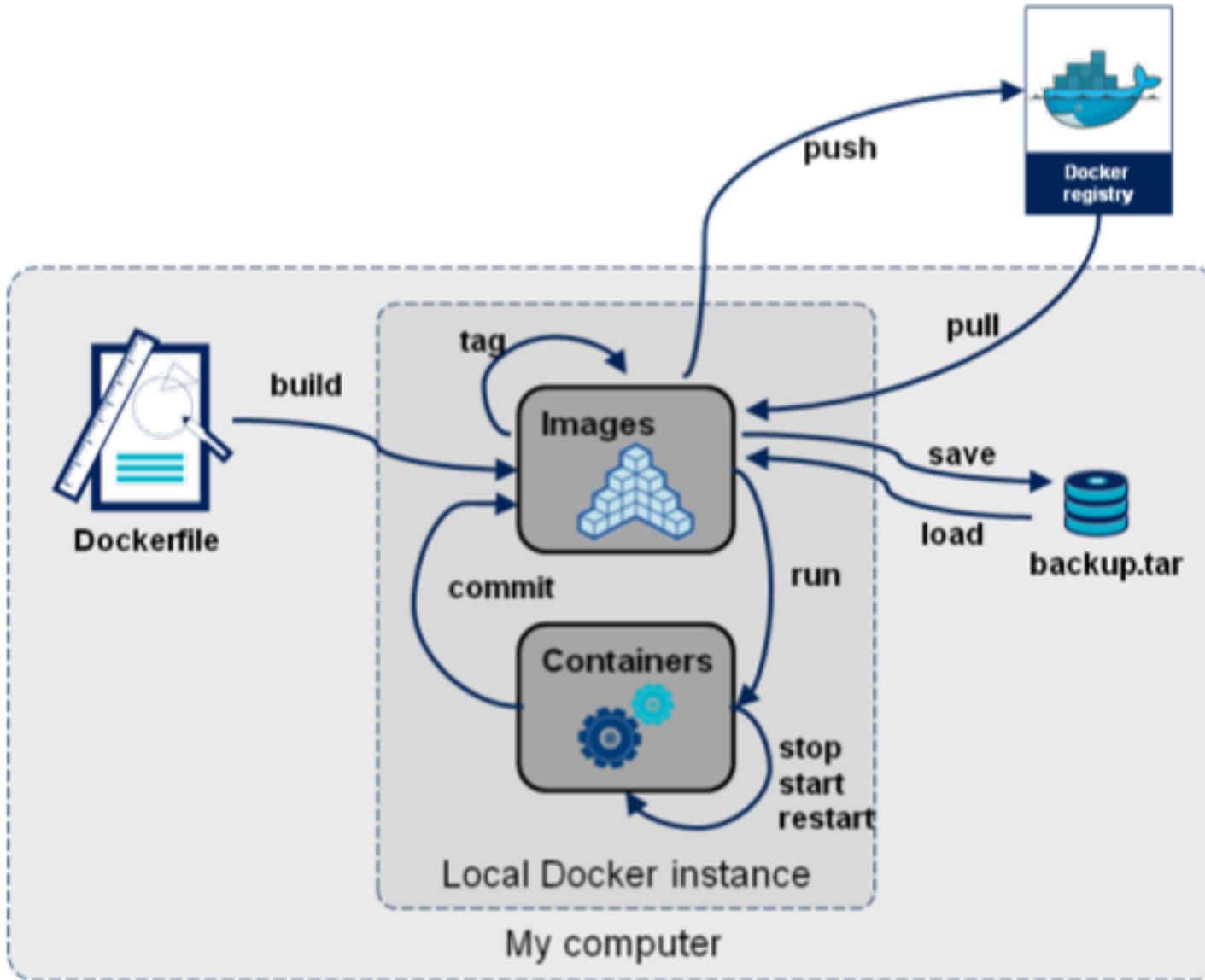
Ein Container läuft so lange wie sein Entrypoint-Prozess im Vordergrund läuft. Docker merkt sich den Container-Zustand.

# Die 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.

# Der Docker Workflow.



# Das Dockerfile definiert Aufbau und Inhalt des Image.

```
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"]
```

# Typische Kommandos eines Docker Workflows

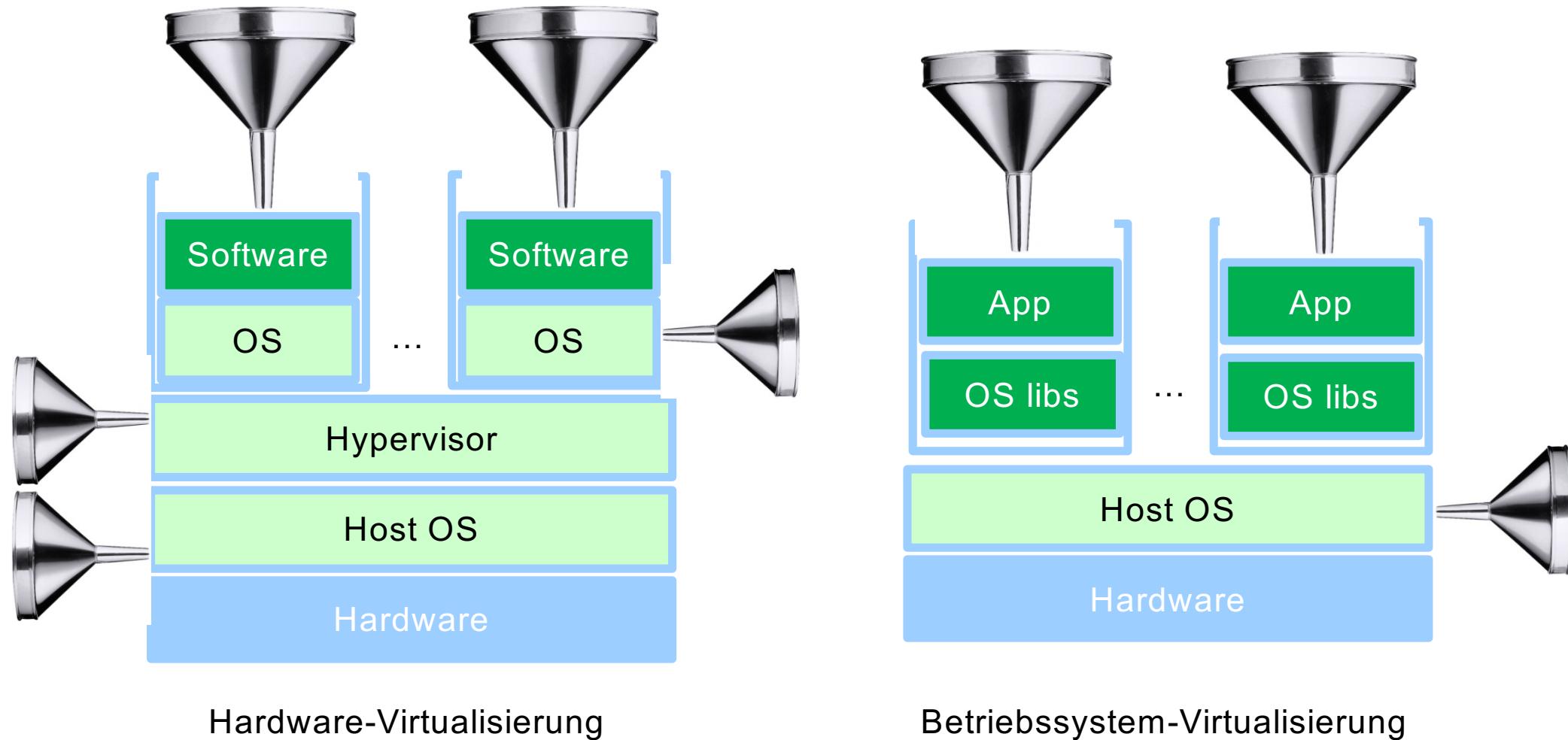
Command	Action
<code>docker build -t &lt;image&gt; .</code>	Build Docker image with given tag in current directory
<code>docker run -d -v &lt;volume mounts&gt; -p &lt;host-port&gt;:&lt;container-port&gt; &lt;image&gt; &lt;entrypoint process&gt;</code>	Run a Docker image: Creates and runs a container. <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 -ti &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 -a = only running containers)

# Mögliche Klausurfragen

- Was ist der allgemeine Zweck von Virtualisierung?
- Welche Arten der Virtualisierung gibt es und wie unterscheiden sich diese?
- Welche vier Ressourcen-Arten werden mindestens virtualisiert?
- Was ist bei Docker der Unterschied zwischen Images und Containern?
- Containerisierung einer Java Web-Anwendung.
  - Schreiben sie ein Dockerfile. Verwenden sie tomee:8-jre-7.0.4-webprofile als Basis Image und kopieren sie die WAR Datei nach /usr/local/tomee/webapps/
  - Wie lautet der Befehl zum Bauen des Image?
  - Wie lautet der Befehl zum Starten des Image im Hintergrund mit Port Binding 8080?

# Kapitel 4: Provisionierung

# Provisionierung: Wie kommt Software in die Boxen?

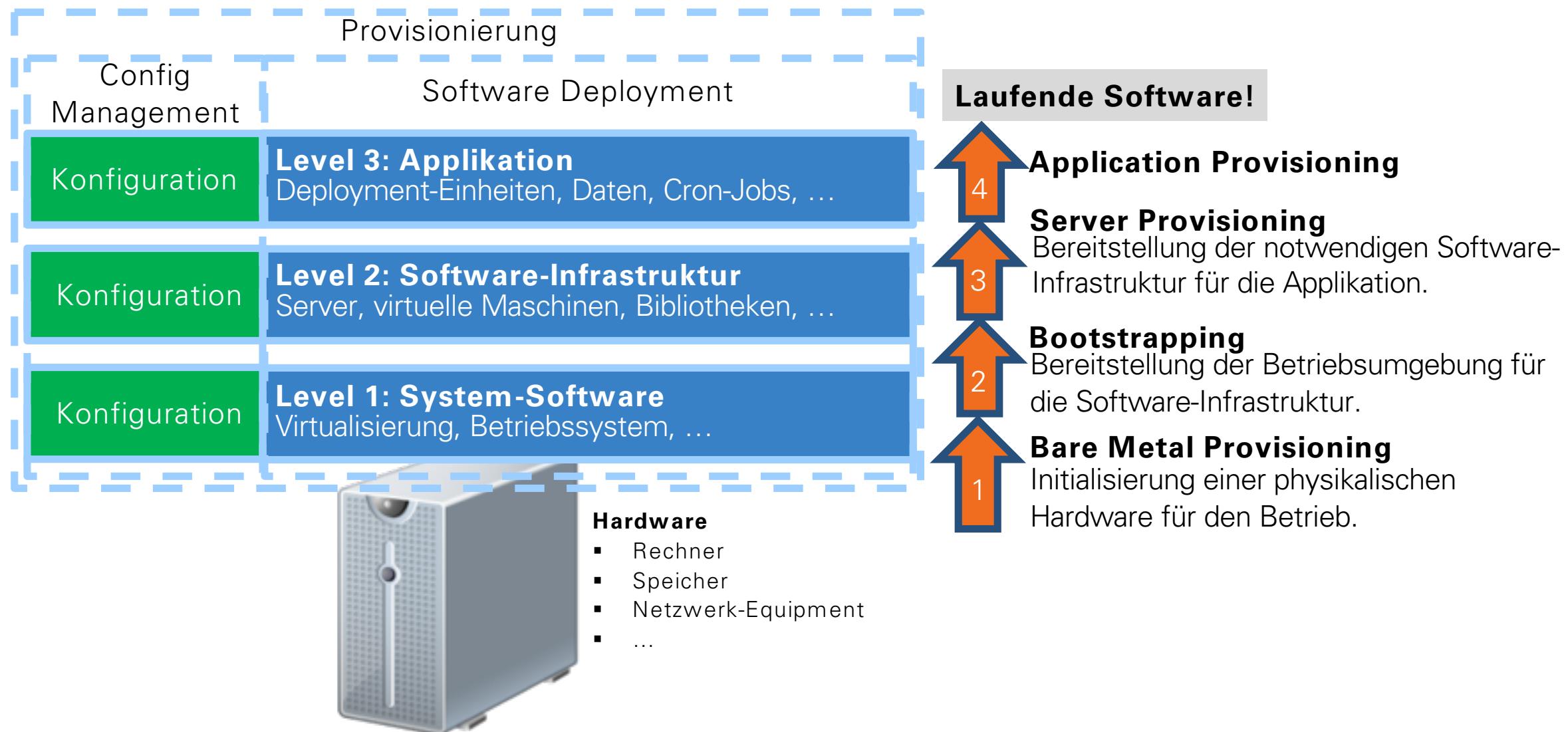


Hardware-Virtualisierung

Betriebssystem-Virtualisierung

Provisionierung ist die Bezeichnung für die automatisierte Bereitstellung von IT-Ressourcen.  
<http://wirtschaftslexikon.aabler.de/Definition/provisionierung.html>

# Provisierung erfolgt auf drei verschiedenen Ebenen und in vier Stufen.



# Konzeptionelle Überlegungen zur Provisionierung.

**Systemzustand** := Gesamtheit der Software, Daten und Konfigurationen auf einem System.

**Provisionierung** := Überführung von einem System in seinem aktuellen Zustand auf einen Ziel-Zustand.



Was ein Provisionierungsmechanismus leisten muss:

1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggf. Zustand zurücksetzen

Zusicherungen

**Idempotenz**: Die Fähigkeit eine Aktion durchzuführen und sie das selbe Ergebnis erzeugt, egal ob sie einmal oder mehrfach ausgeführt wird.

**Konsistenz**: Nach Ausführung der Aktionen herrscht ein konsistenter Systemzustand. Egal ob einzelne, mehrere oder alle Aktionen gescheitert sind.

# Die neue Leichtigkeit des Seins.

## Old Style



1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggF. Zustand zurücksetzen

## New Style

„Immutable Infrastructure / Phoenix Systems“



1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggF. Zustand zurücksetzen

# Provisionierung mit DockerFile und Docker Compose

## Deployment-Ebenen

### Level 3: Applikation

Deployment-Einheiten, Daten, Cron-Jobs, ...

### Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

### Level 1: System-Software

Virtualisierung, Betriebssystem, ...

## Docker-Image-Kette

### Applikations-Image

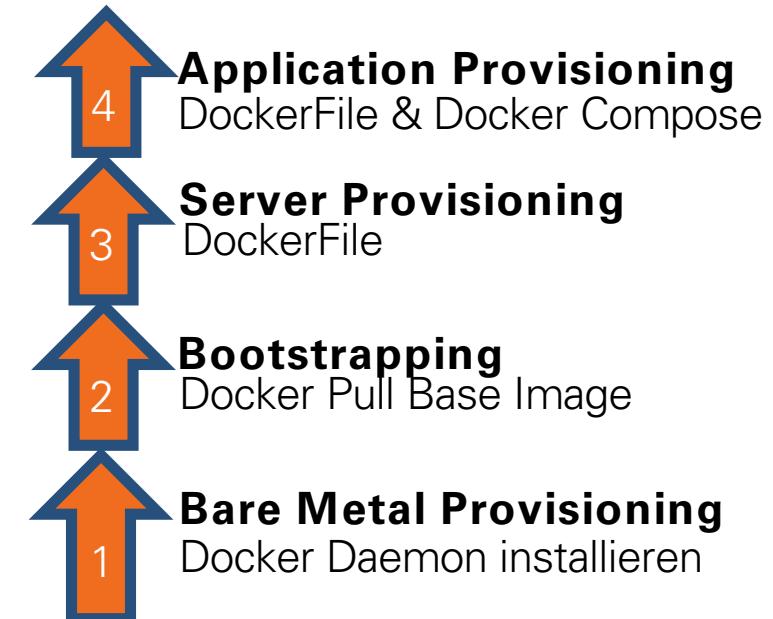
(z.B. www.qaware.de)

### Server Image

(z.B. NGINX)

### Base Image

(z.B. Ubuntu)



# A Docker build must be repeatable.

- A build at a later time must produce an identical image.
- Keep care with versions
  - All files for the image are stored in the repository of the Dockerfile
  - No LATEST tag, use explicit versions instead
  - Always define a version when installing software

```
RUN apt-get update && apt-get install -y ruby1.9.1
```

# Concatenate associated commands in the `RUN` command

- Every RUN command produces a Layer
- Less Layers are better for building and contributing images
- Concatenate commands with \

Installation of several software packages

```
RUN apt-get update && apt-get install -y wget \
    git-core=1:1.9.1-1 \
    subversion=1.8.8-1ubuntu3.2 \
    ruby=1:1.9.3.4 && \
    apt-get clean
```

# Remove temporary files

- Remove all temporary files of the build process to produce small Docker Images
- Use the clean command
- Don't use the clean command in a separate RUN command (it is not possible to clean a different Layer)

## Installation of a Linux Package with YUM

```
RUN yum -y install mypackage1 && \
    yum -y install mypackage2 && \
    yum clean all -y
```

# Publish important ports with **EXPOSE**

- EXPOSE makes a port accessible for the host system or other containers
- Exposed Ports
  - are shown by the docker ps command
  - are executed in the image meta data by the docker inspect command
  - will be connected automatically by linked containers

```
EXPOSE 12340
```

# Define Environment Variables

- Visible in Dockerfile
- Can be used during Build and Execution
- Can be overwritten at the start of a container

```
ENV JAVA_HOME /opt/java-oracle/jdk1.8.0_92
```

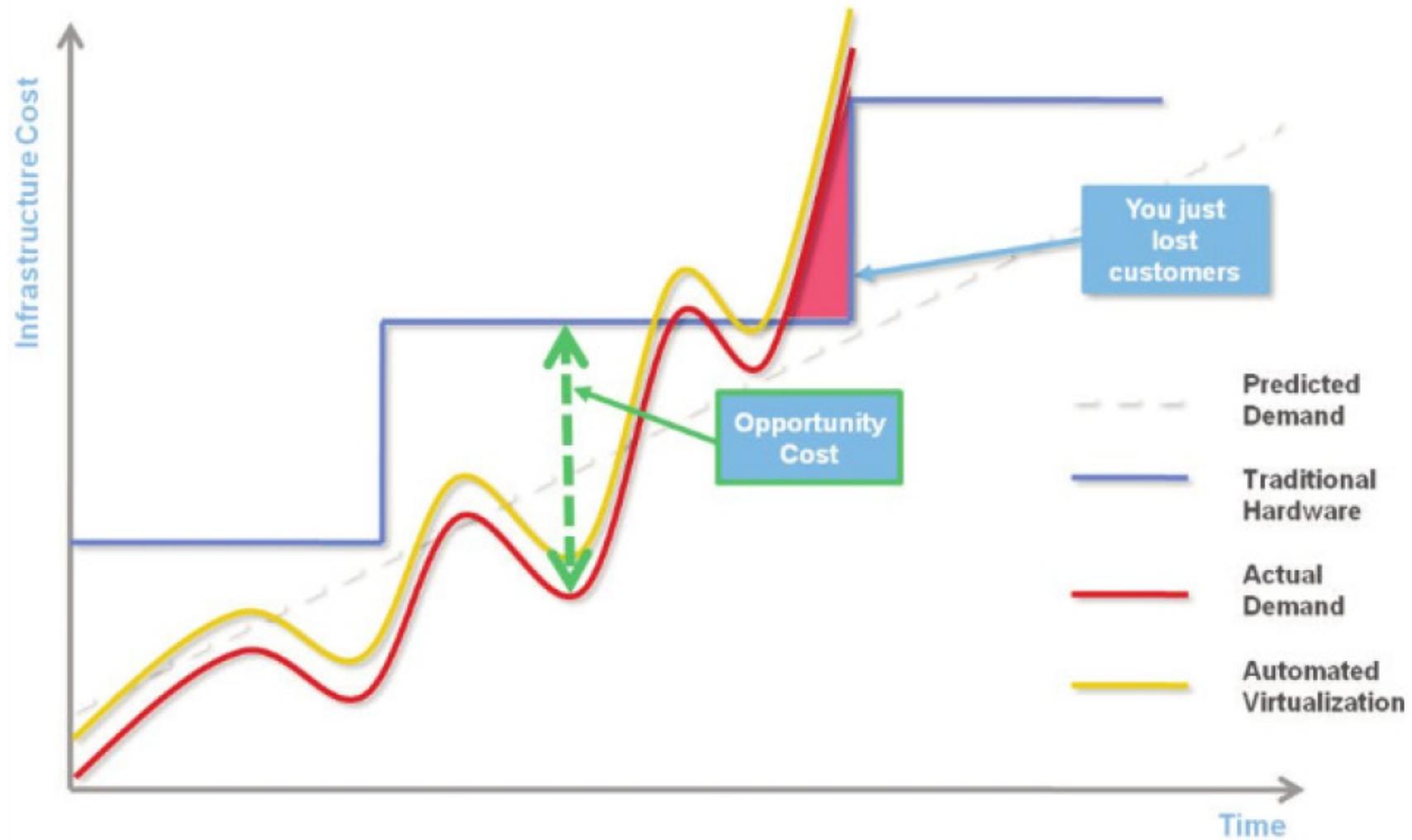
```
ENV MAVEN_HOME /usr/share/maven
```

# Mögliche Klausurfragen

- Was versteht man unter dem Begriff Provisionierung? Auf welchen Ebenen und Stufen kann Provisionierung erfolgen?
- Welche Zusicherungen muss ein Provisionierungsmechanismus machen? Erläutern sie diese Zusicherungen. Welchen Zweck verfolgen diese Zusicherungen aus ihrer Sicht?
- Was versteht man unter „Immutable Infrastructure“? Welche Vorteile bietet dieser Ansatz ihrer Meinung nach?

# Kapitel 5: Infrastructure-as-a-Service

Klassische Betriebsszenarien werden bei dynamischer Nachfrage teuer. Hohe Opportunitätskosten.



Source: Amazon Web Services

# Definition IaaS

Unter *IaaS* versteht man ein Geschäftsmodell, das entgegen dem klassischen Kaufen von Rechnerinfrastruktur vorsieht, diese je nach Bedarf anzumieten und freizugeben.

Eigenschaften einer IaaS-Cloud:

- **Ressourcen-Pools:** Verfügbarkeit von scheinbar unbegrenzten Ressourcen, die Anfragen verteilt verarbeiten.
- **Elastizität:** Dynamische Zuweisung von zusätzlichen Ressourcen bei Bedarf.
- **Pay-as-you-go Modell:** Abgerechnet werden nur verbrauchte Ressourcen.

Ressourcen-Typen in einer IaaS-Cloud:

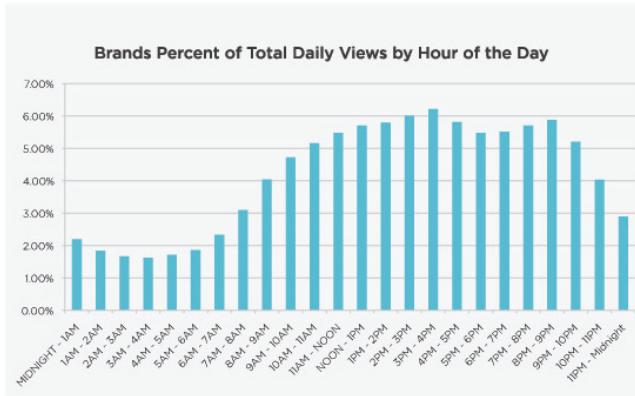
- **Rechenleistung:** Rechner-Knoten mit CPU, RAM und HD-Speicher.
- **Speicher:** Storage-Kapazitäten als Dateisystem-Mounts oder Datenbanken.
- **Netzwerk:** Netzwerk und Netzwerk-Dienste wie DNS, DHCP, VPN, CDN und Load Balancer.

Infrastruktur-Dienste einer IaaS-Cloud:

- **Monitoring**
- **Ressourcen-Management**

# Skalierbarkeit: Effekte

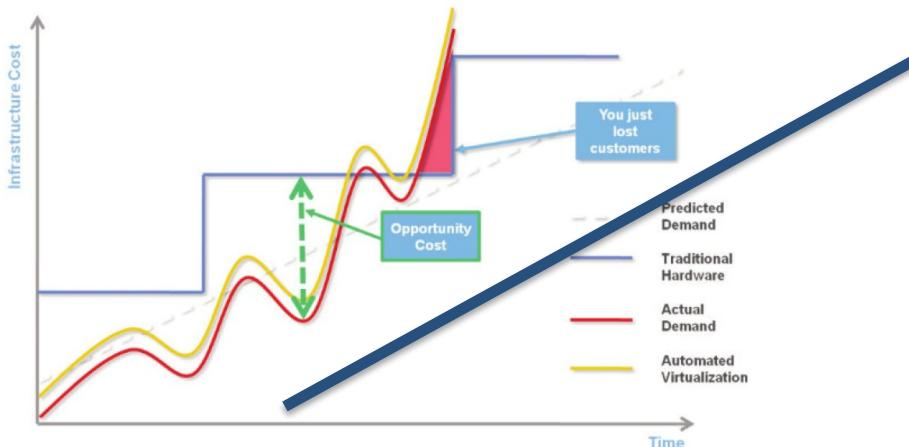
- **Tageszeitliche und saisonale Effekte:** Mittags-Peak, Prime-Time-Peak, Wochenend-Peak, Weihnachten, Valentinstag, Muttertag, ... (vorhersehbare Belastungsspitzen)



- **Sondereffekte:** z.B. Slashdot-Effekt (unvorhersehbare Belastungsspitzen)

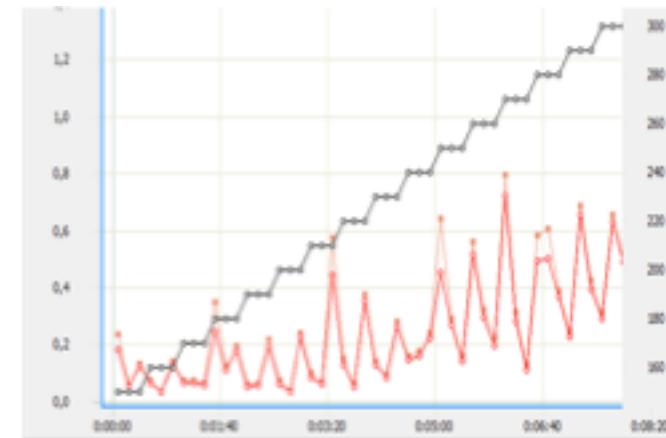


- **Kontinuierliches Wachstum**



Source: Amazon Web Services

- **Temporäre Plattformen:** Projekte, Tests, ...



# Elastizitätsarten

**Nachfrageelastizität:** Die allokierten Ressourcen steigen / sinken mit der Nachfrage.

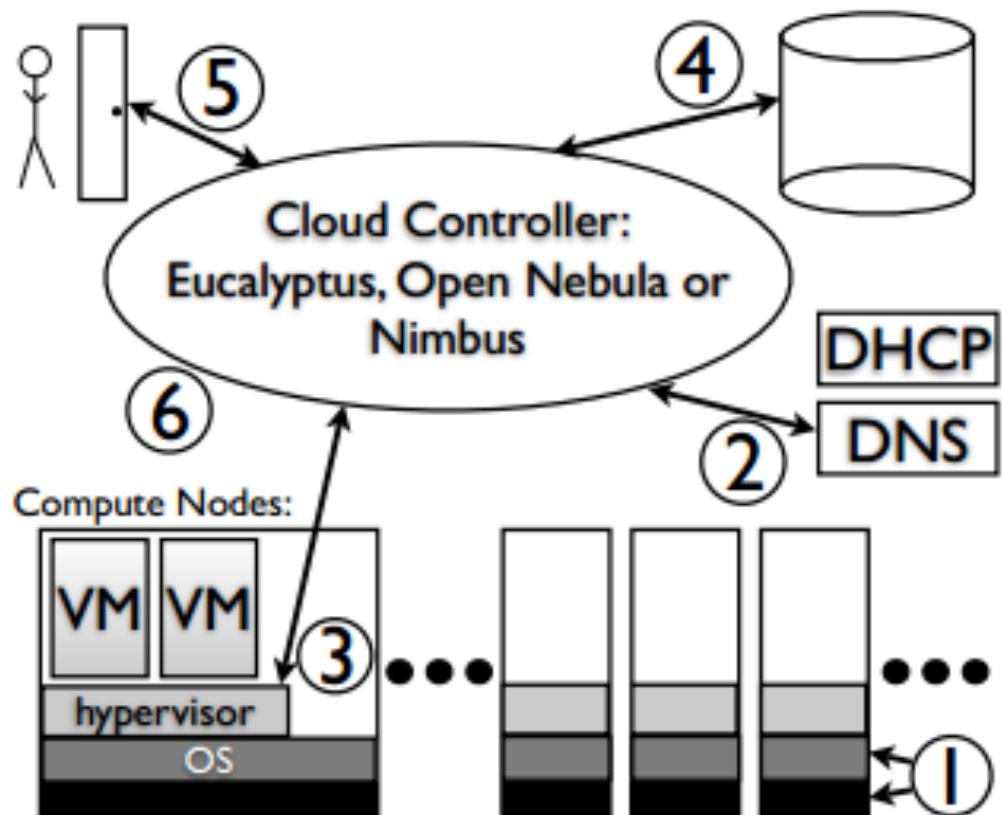
- Pseudo-Elastizität: Schneller Aufbau. Kurze Kündigungsfrist.
- Echtzeit-Elastizität: Allokation und Freigabe von Ressourcen innerhalb von Sekunden. Automatisierter Prozess mit manuellen Triggern oder nach Zeitplan.
- Selbstadaptive Elastizität: Automatische Allokation und Freigabe von Ressourcen in Echtzeit auf Basis von Regeln und Metriken.

**Angebotselastizität:** Die allokierten Ressourcen steigen / sinken mit dem Angebot.

- Dies ist das typische Verhalten eines Grids: Alle verfügbaren Rechner werden allokiert.
- Es sind auch Varianten verfügbar, bei denen man für freie Ressourcen bieten kann.

**Einkommenselastizität:** Die allokierten Ressourcen steigen / sinken mit dem Einkommen bzw. dem Budget.

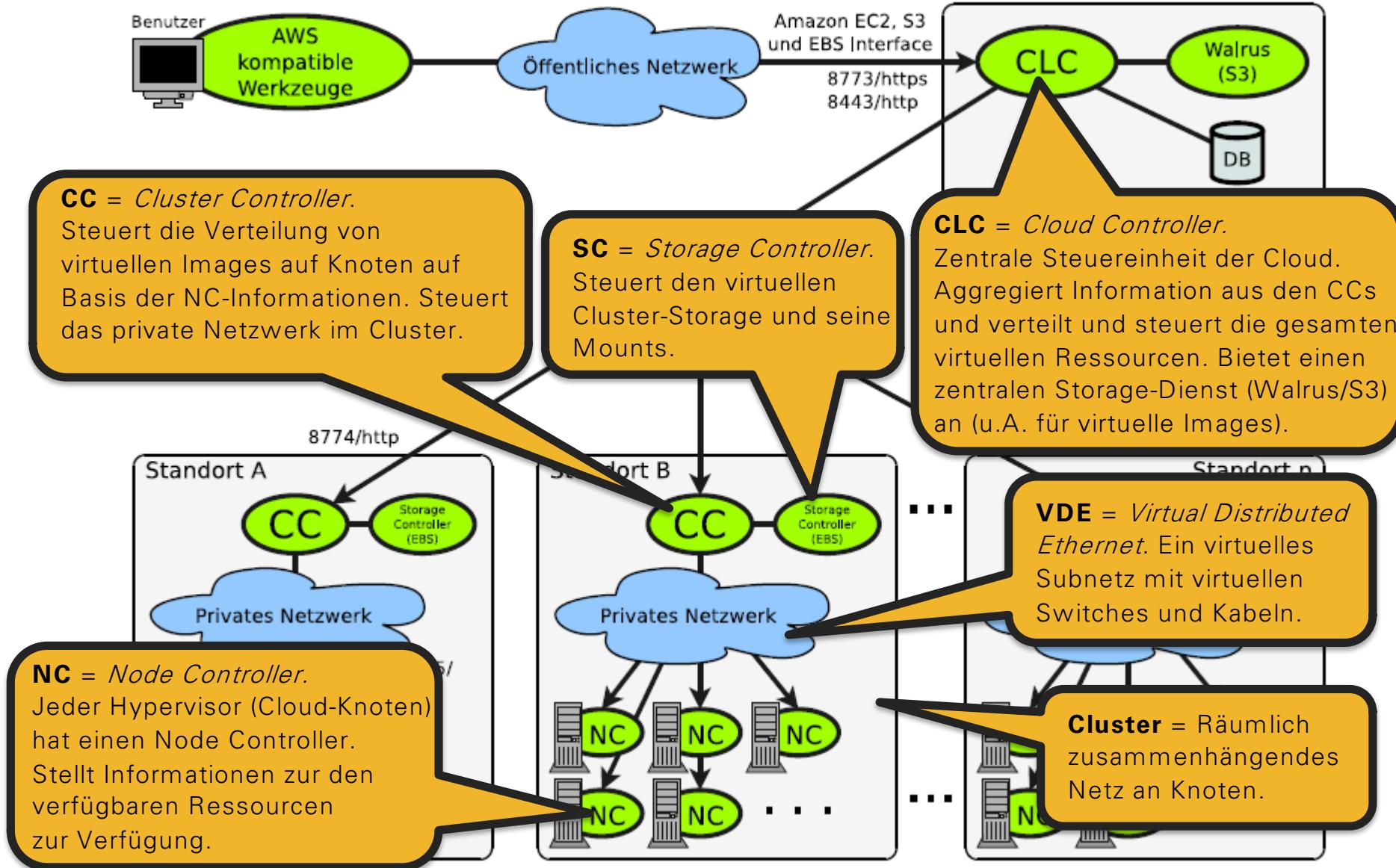
# Eine IaaS-Referenzarchitektur.



1. Hardware und Betriebssystem
2. Virtuelles Netzwerk und Netzwerkdienste
3. Virtualisierung
4. Datenspeicher und Image-Verwaltung
5. Managementschnittstelle für Administratoren und Benutzer
6. Cloud Controller für das mandantenspezifische Management der Cloud-Ressourcen

Peter Sempolinski and Douglas Thain,  
**"A Comparison and Critique of Eucalyptus, OpenNebula and Nimbus"**,  
IEEE International Conference on Cloud Computing Technology and Science, 2010.

# Der interne Aufbau einer IaaS-Cloud am Beispiel Eucalyptus.

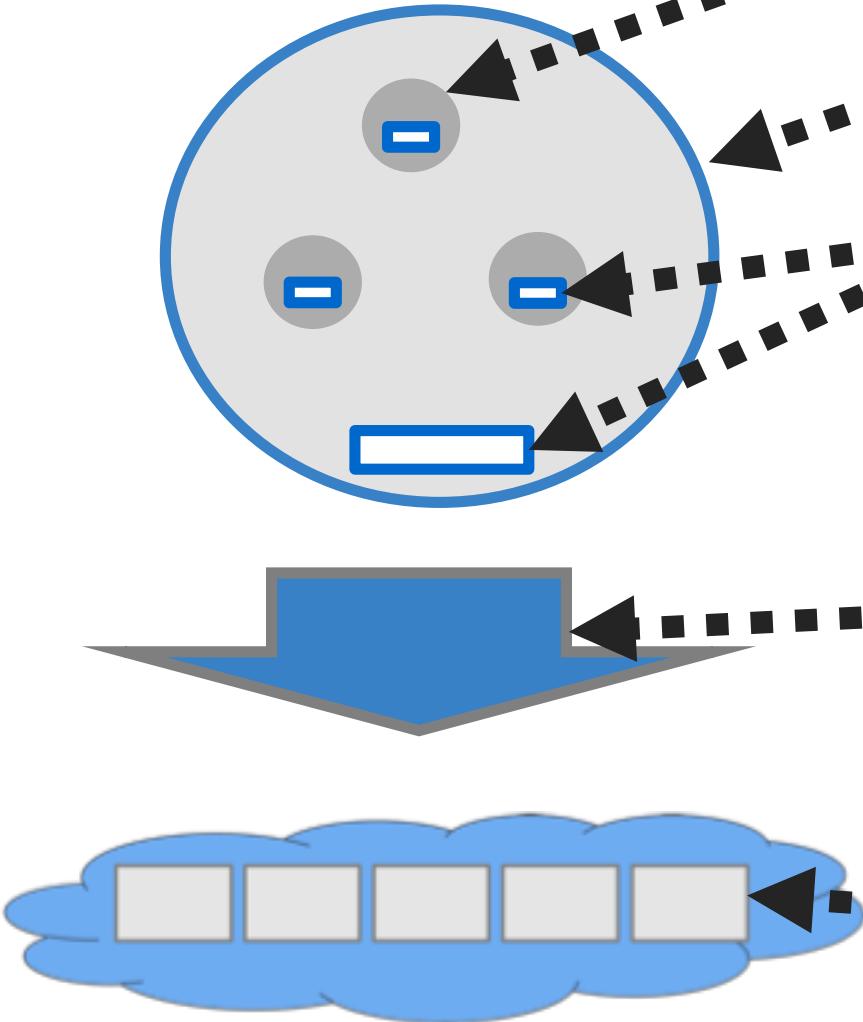


# Mögliche Klausurfragen

- Was versteht man unter Opportunitätskosten in Bezug auf die Betriebskosten von physischer Hardware?
- Gegeben sei eine IaaS-Architektur entsprechend dem Beispiel von Eucalyptus: Was passiert, wenn der Cluster Controller vollständig ausfällt? Was ist weiterhin in der IaaS Cloud noch möglich?

# Kapitel 6: Cluster Scheduling

# Terminologie



**Task:** Atomare Rechenaufgabe inklusive Ausführungsvorschrift.

**Job:** Menge an Tasks mit gemeinsamen Ausführungsziel. Die Menge an Tasks ist i.d.R. als DAG mit Tasks als Knoten und Ausführungsabhängigkeiten als Kanten repräsentiert.

**Properties:** Ausführungsrelevante Eigenschaften der Tasks und Jobs, wie z.B.:

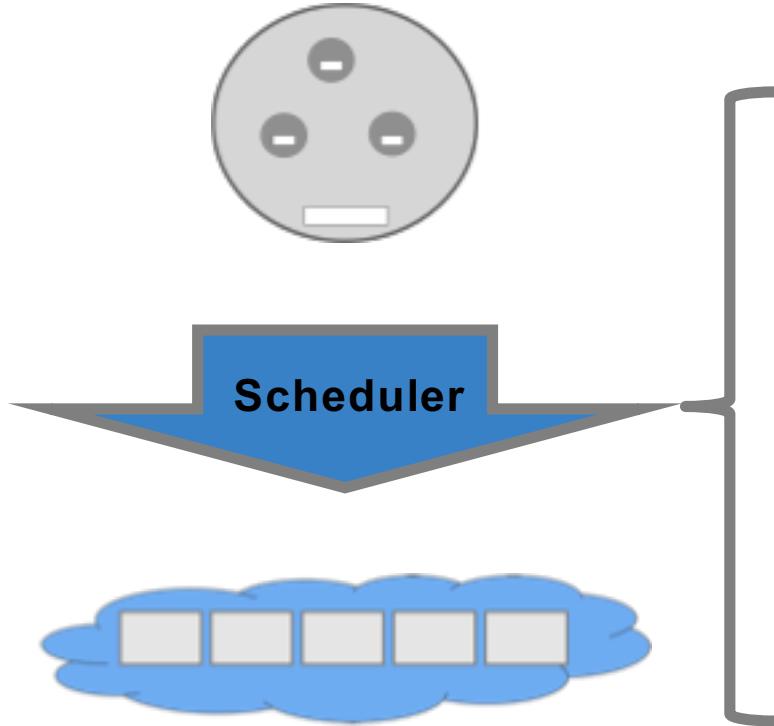
- Task: Ausführungszeitpunkt, Priorität, Ressourcenverbrauch
- Job: Abhängigkeiten der Tasks, Ausführungszeitpunkt

**Scheduler:** Ausführung von Tasks auf den verfügbaren Resources unter Berücksichtigung der Properties und gegebener

**Scheduling-Ziele** (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann **präemptiv** sein, also die Ausführung von Tasks unterbrechen und neu aufsetzen können.

**Resources:** Cluster an Rechnern mit CPU-, RAM-, HDD-, Netzwerk-Ressourcen. Ein Rechner stellt seine Ressourcen temporär zur Ausführung eines oder mehrerer Tasks zur Verfügung (**Slot**). Die parallele Ausführung von Tasks ist isoliert zueinander.

# Aufgaben eines Cluster-Schedulers:

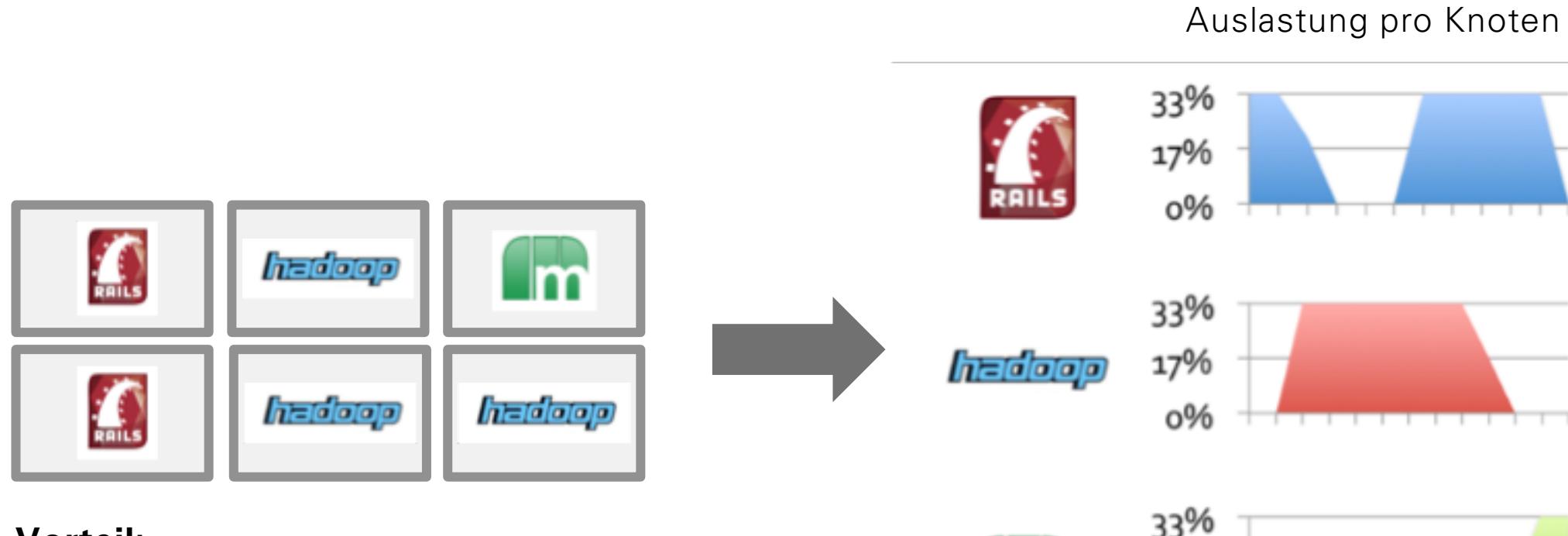


**Cluster Awareness:** Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.

**Job Allocation:** Zur Ausführung eines Services die passende Menge an Ressourcen für einen bestimmten Zeitraum bestimmen und allozieren.

**Job Execution:** Einen Service zuverlässig ausführen und dabei isolieren und überwachen.

# Die einfachste Form des Scheduling: Statische Partitionierung

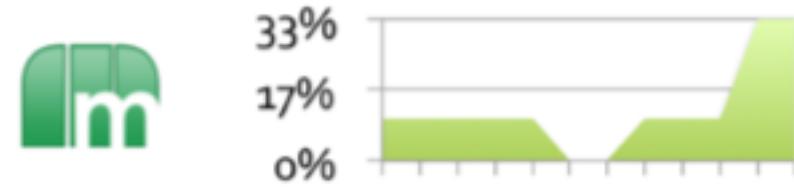
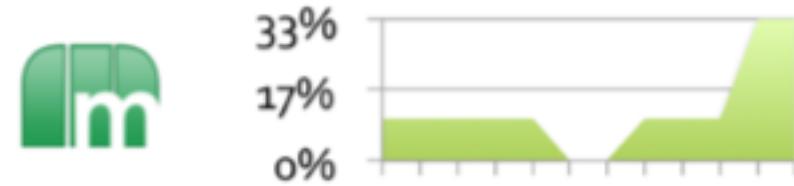


## Vorteil:

- Einfach zu realisieren

## Nachteile:

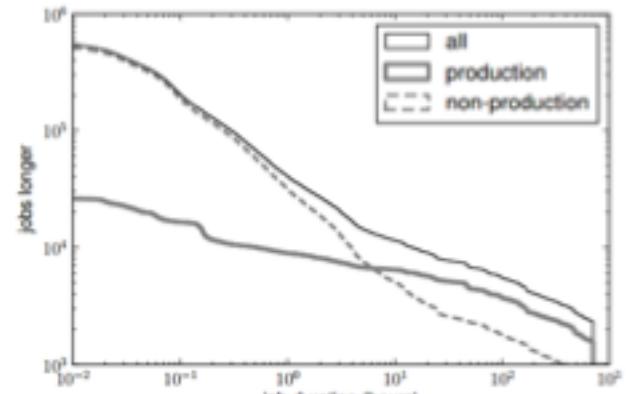
- Nicht flexibel bei geänderten Bedürfnissen
- Geringere Auslastung  
➔ hohe Opportunitätskosten



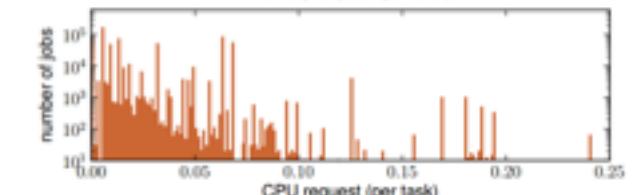
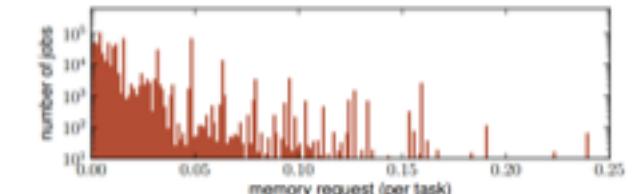
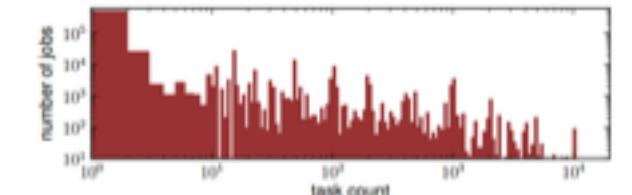
Bildquelle: Practical Considerations for Multi-Level Schedulers,  
Benjamin Hindman, 19th Workshop on Job Scheduling  
Strategies for Parallel Processing (JSSPP) 2015

# Heterogenität im Scheduling

- In typischen Clustern ist die Workload an Jobs sehr heterogen.
- Charakteristische Unterschiede sind:
  - Ausführungszeit: min, h, d, INF.
  - Ausführungszeit: sofort, später, zu einem Zeitpunkt.
  - Ausführungszweck: Datenverarbeitung, Request-Handling.
  - Ressourcenverbrauch: CPU-, RAM-, HDD-, NW-dominant.
  - Zustand: zustandsbehaftet, zustandslos.
- Zu unterscheiden sind mindestens:
  - **Batch-Jobs:** Ausführungszeit im Minuten- bis Stundenbereich. Eher niedrige Priorität und gut unterbrechbar. Müssen i.d.R. bis zu einem bestimmten Zeitpunkt abgeschlossen sein. Zustandsbehaftet.
  - **Service-Jobs:** Sollen auf unbestimmte Zeit unterbrechungsfrei laufen. Haben hohe Priorität und sollten nicht unterbrochen werden. Teilweise zustandslos.

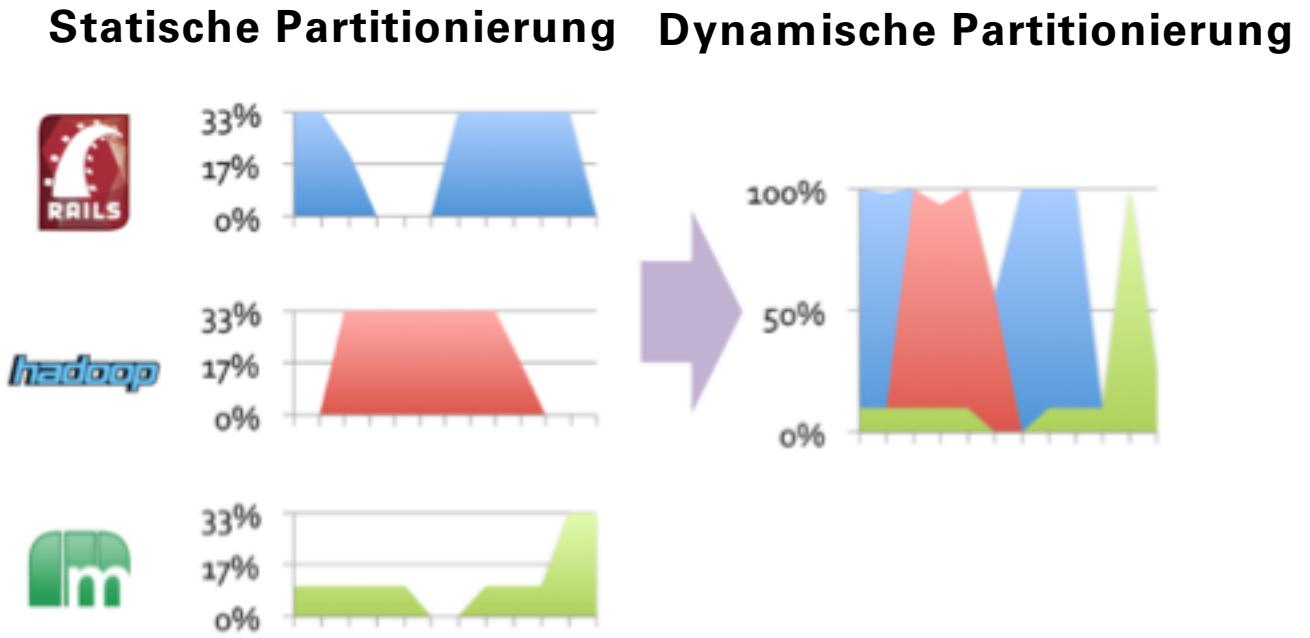
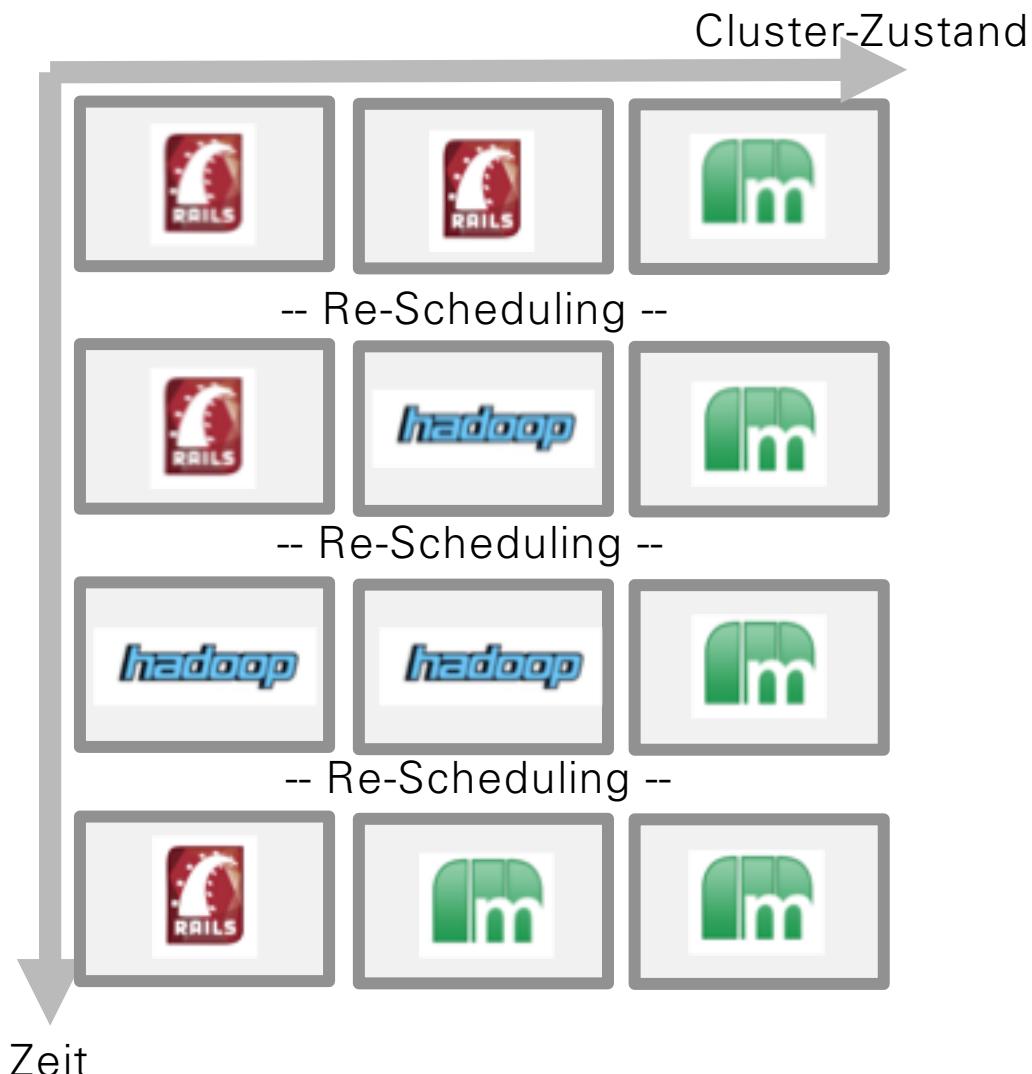


Ausführungszeit



Ressourcenverbrauch

# Bestehende Ressourcen einer Cloud können durch dynamische Partitionierung wesentlich effizienter genutzt werden.



## Vorteile der dynamischen Partitionierung:

- Höhere Auslastung der Ressourcen → weniger Ressourcen notwendig → geringere Betriebskosten
- Potenziell schnellere Ausführung einzelner Tasks, da Ressource opportun genutzt werden können.

# Ein Cluster-Scheduler: Eingabe, Verarbeitung, Ausgabe.

**Eingabe eines Cluster-Schedulers** ist Wissen über die Jobs und Tasks (Properties) und über die Ressourcen:

- **Resource Awareness:** Welche Ressourcen stehen zur Verfügung und wie ist der entsprechende Bedarf des Tasks?
- **Data Awareness:** Wo sind die Daten, die ein Task benötigt?
- **QoS Awareness:** Welche Ausführungszeiten müssen garantiert werden?
- **Economy Awareness:** Welche Betriebskosten dürfen nicht überschritten werden?
- **Priority Awareness:** Wie ist die Priorität der Task zueinander?
- **Failure Awareness:** Wie hoch ist die Wahrscheinlichkeit eines Ausfalls? (z.B. da ein Rack oder eine Stromvers.)
- **Experience Awareness:** Wie hat sich ein Tasks in der Vergangenheit verhalten?



**Ausgabe eines Cluster-Schedulers:**

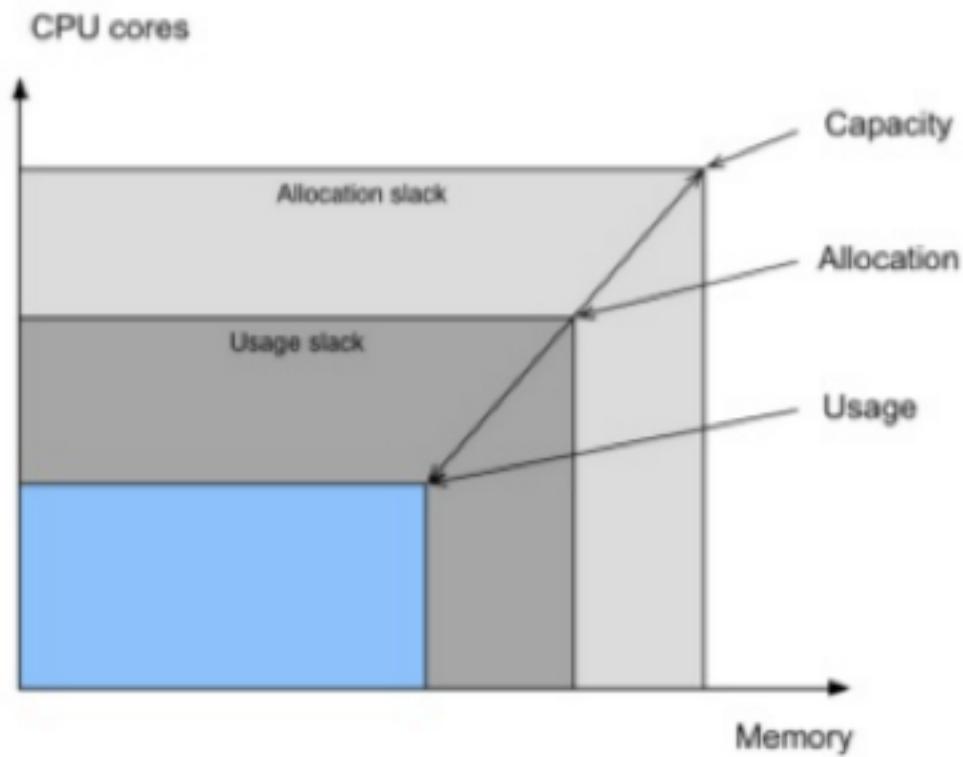
Placement Decision als

- **Slot-Reservierungen**
- **Slot-Stornierungen** (im Fehlerfall, Optimierungsfall, Constraint-Verletzung)

**Verarbeitung im Cluster-Scheduler: Scheduling-Algorithmen** entsprechend der jeweiligen **Scheduling-Ziele**, wie z.B.:

- **Fairness:** Kein Task sollte unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- **Maximaler Durchsatz:** So viele Tasks pro Zeiteinheit wie möglich.
- **Minimale Wartezeit:** Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks.
- **Ressourcen-Auslastung:** Möglichst hohe Auslastung der verfügbaren Ressourcen.
- **Zuverlässigkeit:** Ein Task wird garantiert ausgeführt.
- **Geringe End-to-End Ausführungszeit** (z.B. durch Daten-Lokalität und geringe Kommunikationskosten, syn. Makespan)

Hauptziel ist es oft, die Ressourcen-Auslastung zu optimieren. Das spart Opportunitätskosten.

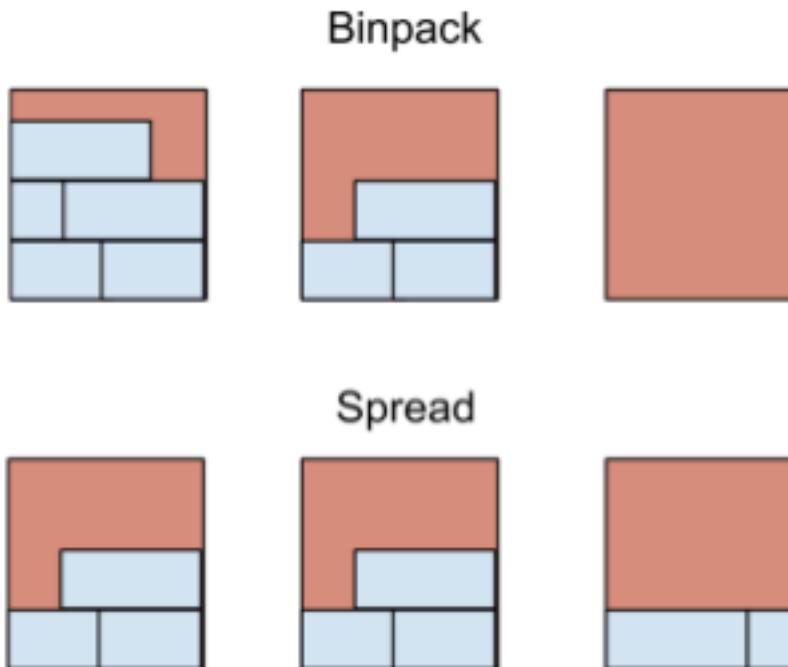


# Scheduling ist eine Optimierungsaufgabe, die NP-vollständig ist.

- ... und ist NP-vollständig.  
Die Optimierungsaufgabe lässt sich auf das Travelling Salesman Problem zurückführen.
- Das bedeutet:
  - Es ist kein Algorithmus bekannt, der eine optimale Lösung garantiert in Polynomialzeit erzeugt.
  - Algorithmus muss für tausende Jobs und tausende Ressourcen skalieren. Optimale Algorithmen, die den Lösungsraum komplett durchsuchen sind nicht praktikabel, da deren Entscheidungszeit zu lange ist für große Eingabemengen ( $|Jobs| \times |Ressourcen|$ ).
  - Praktikable Scheduling-Algorithmen sind somit Algorithmen zur näherungsweisen Lösung des Optimierungsproblems (Heuristiken, Meta-Heuristiken).
- Darüber hinaus kommen Job-Anfragen kontinuierlich an, so dass selbst bei optimalem Algorithmus der Re-Organisationsaufwand pro Job unverhältnismäßig hoch werden kann.

# Einfache Scheduling-Algorithmen

- Optimieren das Scheduling von Tasks oft in genau einer Dimension (z.B. CPU-Auslastung) bzw. wenigen Dimensionen (CPU-Auslastung und RAM).
- Populäre Algorithmen:
  - Binpack
  - Spread (Round Robin)



# Multidimensionaler Scheduling-Algorithmus mit Fokus auf Fairness: Dominant Resource Fairness (DRF).

- Aufteilung der Ressourcen an verschiedene „Teams“ (Applikationen, Jobs).
- Ausgangslage: Würden die Ressourcen gleichmäßig statisch an N Teams verteilt, so hat jedes Team eine dominante Ressource, die besonders intensiv genutzt wird. Diese dominante Ressource kann durch Beobachtung ermittelt werden und balanciert sich über alle Teams hinweg aus.
- Fairness-Auffassung: Jedes Team bekommt mindestens  $1/N$  aller Ressourcen der dominanten Ressourcen. Der Scheduling-Algorithmus ist darauf ausgelegt, die minimal verfügbaren dominanten Ressourcen pro Team zu maximieren.
- Die Fairness kann justiert werden. Jedem Team kann ein gewichteter Anteil der Ressourcen in der statischen Ausgangslage zugesprochen werden. Die Fairness-Auffassung ist dann entsprechend gewichtet.

team	{	promotions	trends	recommendations
utilization	{	45% CPU 100% RAM	75% CPU 100% RAM	100% CPU 50% RAM
bottleneck	{	RAM	RAM	CPU

- Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015
- Dominant Resource Fairness: Fair Allocation of Multiple Resource Types, Ghodsi et al., 2011
- DRF ist eine Generalisierung des Min-Max Algorithmus für mehrere Ressourcen:  
<http://www.ece.rutgers.edu/~marsic/Teaching/CCN/minmax-fairsh.html>

# Scheduling-Algorithmus mit Fokus auf Fairness: Capacity Scheduler (CS).

- Es werden Job Queues definiert und zu jeder Queue eine Kapazitätszusage in Ressourcenanteil vom Cluster definiert.
- Fairness-Auffassung: Diese Kapazitätszusage wird stets eingehalten. Der Scheduling-Algorithmus stellt sicher, dass diese Fairness stets sichergestellt wird.
- Damit das Cluster dafür nicht statisch partitioniert werden muss, ist ein sog. Over-Commitment von Ressourcen erlaubt.
- Wird durch ein Over-Commitment aber eine Kapazitätszusage gefährdet, werden die over-committeten Ressourcen entzogen. Hierfür ist also ein präemptiver Scheduler notwendig.

# Eine konzeptionelle Architektur für Cluster-Scheduler.

## Job Queue:

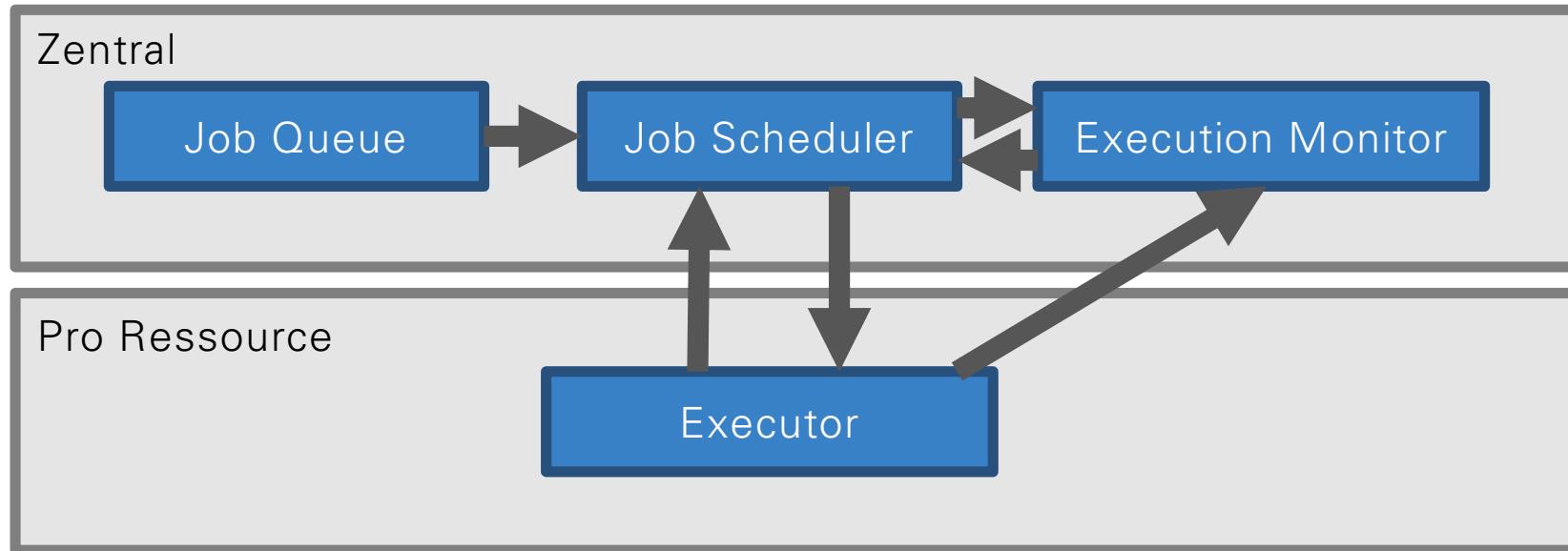
- Eingehende Jobs zur Ausführung
- Events zu eingegangenen Jobs

## Job Scheduler:

- Jobs einplanen
- Taskausführung steuern

## Execution Monitor:

- Task-Ausführung überwachen
- Ressourcen überwachen



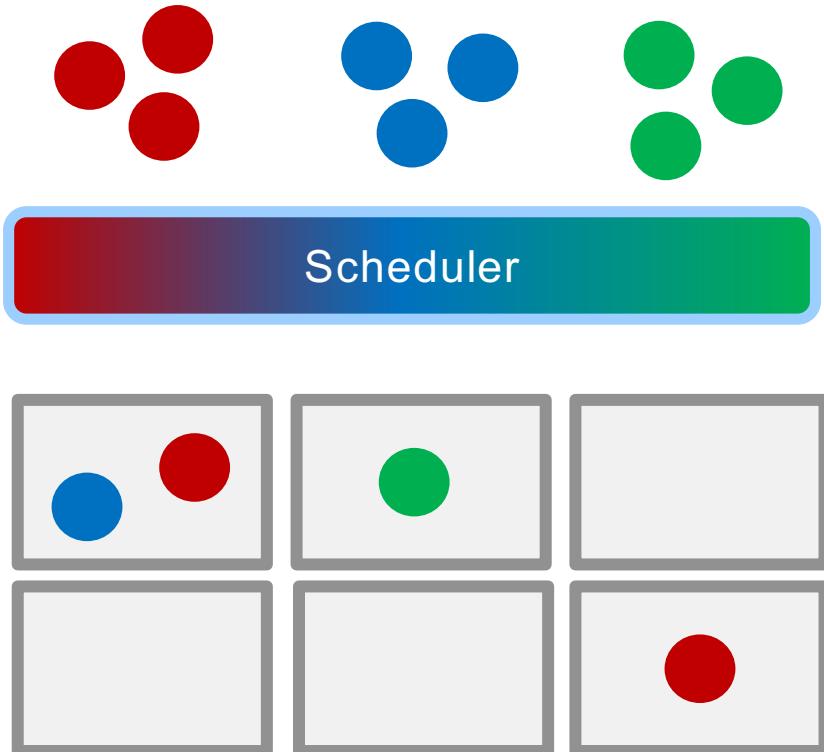
## Executor:

- Task ausführen
- Informationen zur Ressource bereitstellen

## Anforderungen:

- Performance
  - Geringe Queuing-Time
  - Geringe Decision-Time
  - Geringe Ausführungslatenz
- Hoch-Verfügbarkeit und Fehlertoleranz
- Skalierbarkeit bzgl. Anzahl an Jobs und verfügbaren Ressourcen.

# Scheduler-Architektur. Variante 2: Monolithischer Scheduler.



## Vorteile:

- Globale Optimierungsstrategien einfach möglich.

## Nachteile:

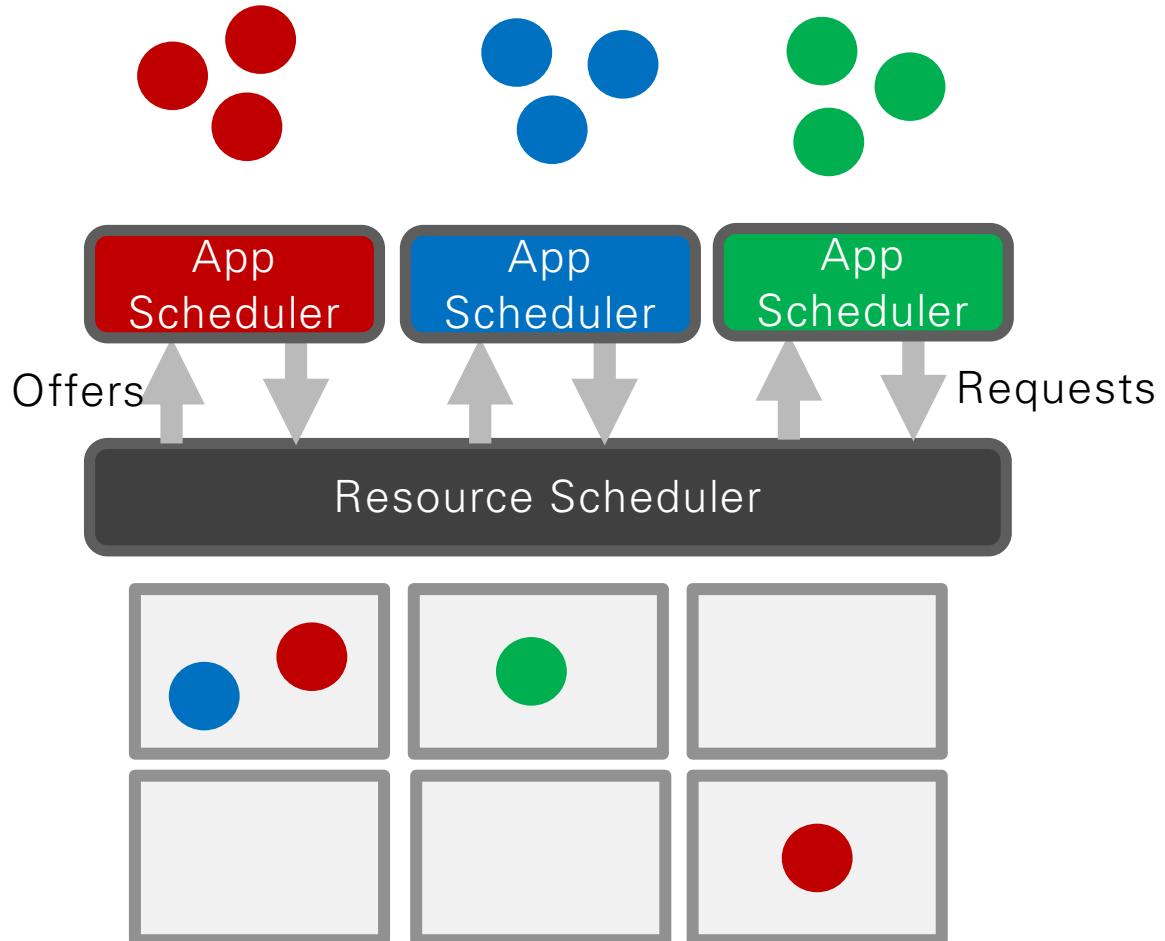
- Heterogenes Scheduling für heterogene Jobs schwierig
  - Komplexe und umfangreiche Implementierung notwendig
  - ... oder homogenes Scheduling von geringerer Effizienz.
- Potenzielles Skalierbarkeits-Bottleneck.

▪ **Google Borg**  
Large-scale cluster management at Google  
with Borg, Verma et al., 2015

- **Hadoop YARN**
- **Kubernetes**
- **Docker Swarm**

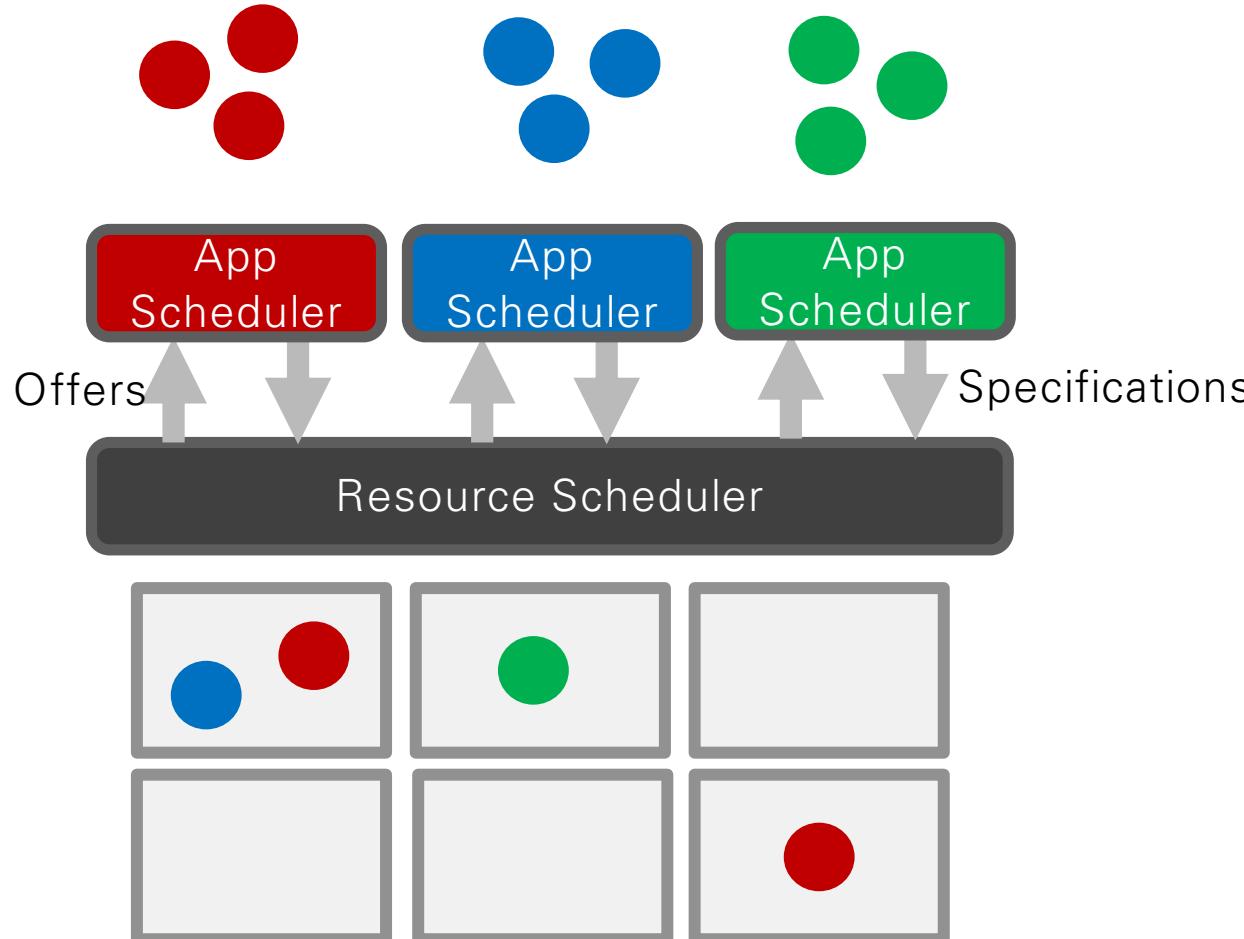
# Scheduler-Architektur.

## Variante 3: 2-Level-Scheduler.



- Auftrennung der Scheduling-Logik in einen Resource Scheduler und einen App Scheduler.
  - Der **Resource Scheduler** kennt alle verfügbaren Ressourcen und darf diese allozieren. Er nimmt Ressourcen-Anfragen (Requests) entgegen und unterbreitet entsprechend einer Scheduling-Policy (definierte Scheduling-Ziele) Ressourcen-Angebote (Offers).
  - Der **App Scheduler** nimmt Jobs entgegen und „übersetzt“ diese in Ressourcen-Anfragen und wählt applikationsspezifisch die passenden Ressourcen-Angebote aus.
- Offers sind eine zeitlich beschränkte Allokation von Ressourcen, die explizit angenommen werden muss.
- Grundsätzlich **pessimistische Strategie**: Disjunkte Offers. I.d.R. sind aber auch optimistische Offers verfügbar, bei denen eine gewisse Überschneidung erlaubt ist.
- Im Sinne der Fairness kann ein prozentualer Anteil der Ressourcen für einen App Scheduler garantiert werden.

# Scheduler-Architektur. Variante 3: 2-Level-Scheduler.



**Apache Mesos**  
Mesos: A Platform for Fine-Grained Resource Sharing  
in the Data Center, Hindman et al., 2010

## Vorteile:

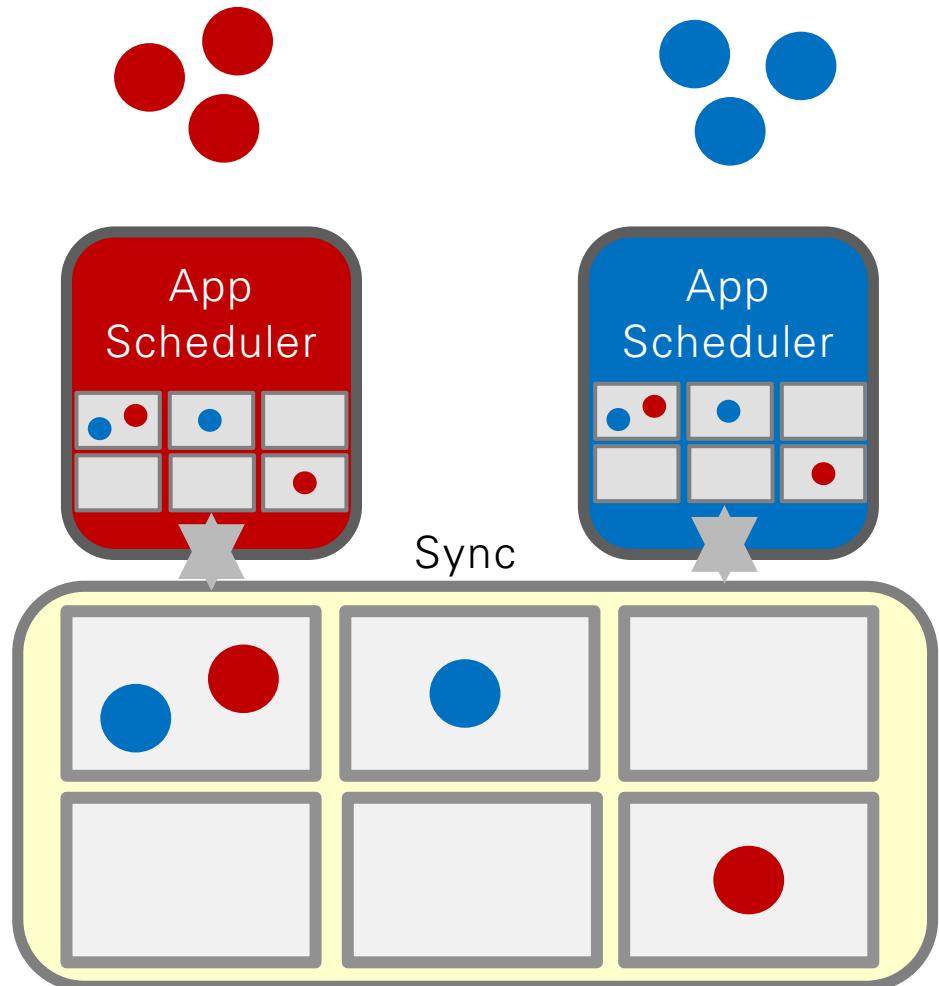
- Nachgewiesene Skalierbarkeit auf tausende von Knoten (z.B. Twitter, Airbnb, Apple Siri).
- Flexible Architektur für heterogene Scheduling-Logiken.

## Nachteile:

- App-Scheduler übergreifende Logiken nur schwer zu realisieren (z.B. globaler Ausführungsverzicht oder Gang-Scheduling)

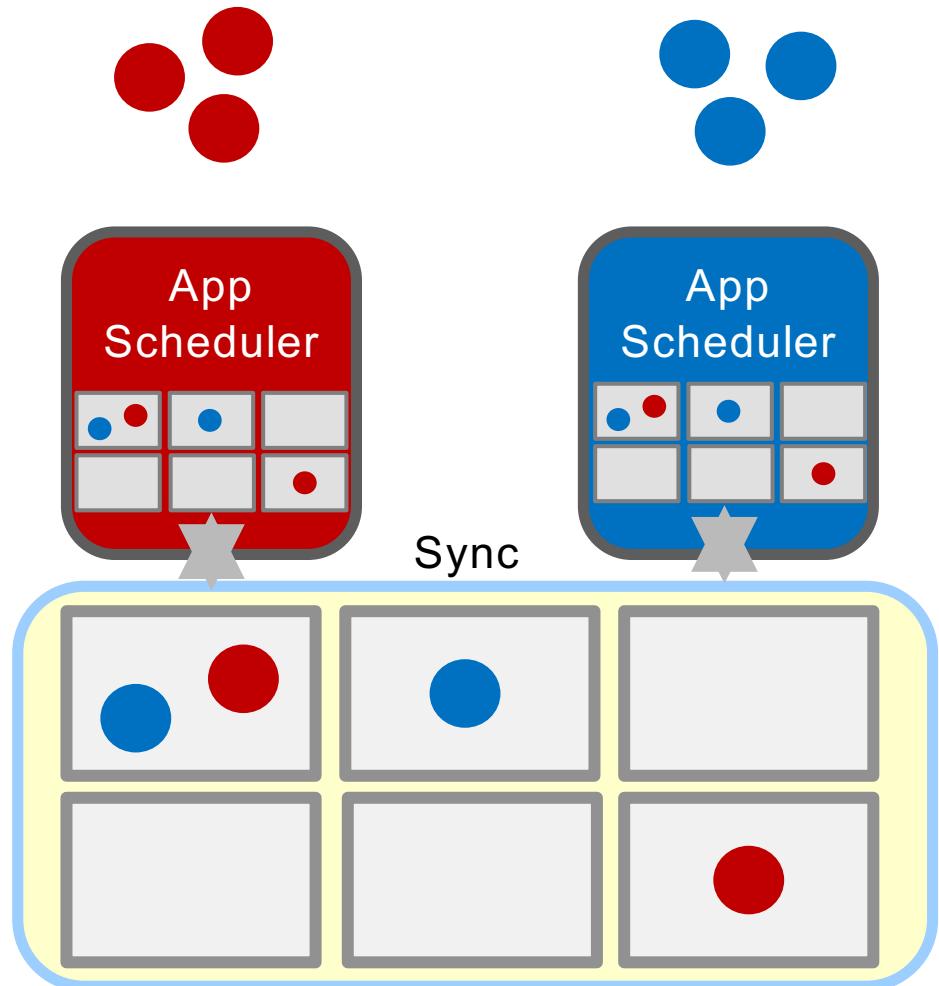
# Scheduler-Architektur.

## Variante 4: Shared-State-Scheduler.



- Es gibt ausschließlich applikationsspezifische Scheduler.
- Die App Scheduler synchronisieren kontinuierlich den aktuellen Zustand des Clusters (Cluster-Zustand: Job-Allokationen und verfügbare Ressourcen).
- Jeder App Scheduler entscheidet die Platzierung von Tasks auf Basis seines aktuellen Cluster-Zustands.
- Optimistische Strategie: Ein zentraler Koordinationsdienst erkennt Konflikte im Scheduling und löst diese auf, in dem er Zustands-Änderungen nur für einen der beteiligten App Scheduler erlaubt und für die anderen App Scheduler einen Fehler meldet.

# Scheduler-Architektur. Variante 4: Shared-State-Scheduler.



## Vorteile:

- Tendenziell geringerer Kommunikations-Overhead.

## Nachteile:

- Komplettes Scheduling muss pro App Scheduler entwickelt werden.
- Keine globalen Scheduling-Ziele (z.B. Fairness) möglich.
- Skalierbarkeit in großen Clustern unklar, da noch nicht in der Praxis erprobt und insbesondere Auswirkung bei hoher Anzahl an Konflikten ungeklärt.

**Google Omega**  
Omega: flexible, scalable schedulers for large  
compute clusters, Schwarzkopf et al., 2013

# Mögliche Klausurfragen

- Was ist der Unterschied zwischen statischer und dynamischer Partitionierung im Scheduling?
- Erläutern sie die beiden mindestens zu unterscheidenden Job-Arten im Cluster-Scheduling.
- Was versteht man unter einer dominanten Ressource beim DRF-Scheduling-Algorithmus?

# Kapitel 7: Cluster Orchestrierung

# Cluster-Orchestrierung

- Eine Anwendung, die in mehrere Betriebskomponenten (Container) aufgeteilt ist, auf mehreren Knoten laufen lassen.  
„Running Containers on Multiple Hosts“.  
DockerCon SF 2015: Orchestration for Sysadmins
- Führt Abstraktionen zur Ausführung von Anwendungen mit ihren Services in einem großen Cluster ein.
- Orchestrierung ist keine statische, einmalige Aktivität wie die Provisionierung sondern eine dynamische, kontinuierliche Aktivität.
- Orchestrierung hat den Anspruch, alle Standard-Betriebsprozeduren einer Anwendung zu automatisieren.

**Blaupause der Anwendung**, die den gewünschten Betriebszustand der Anwendung beschreibt: Betriebskomponenten (Container), deren Betriebsanforderungen sowie die angebotenen und benötigten Schnittstellen.



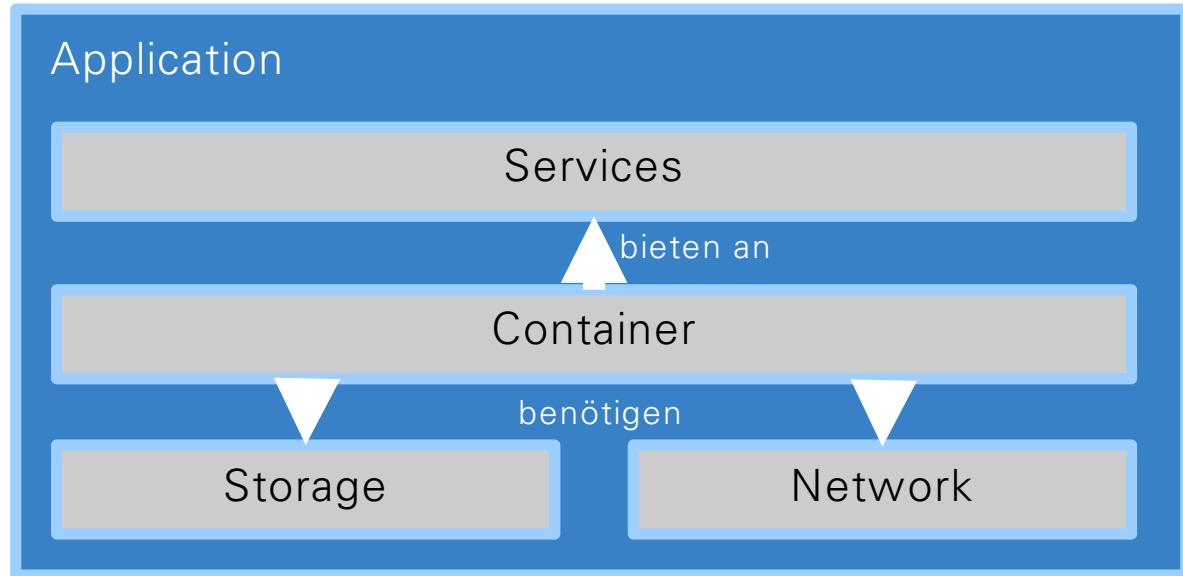
## Cluster-Orchestrator



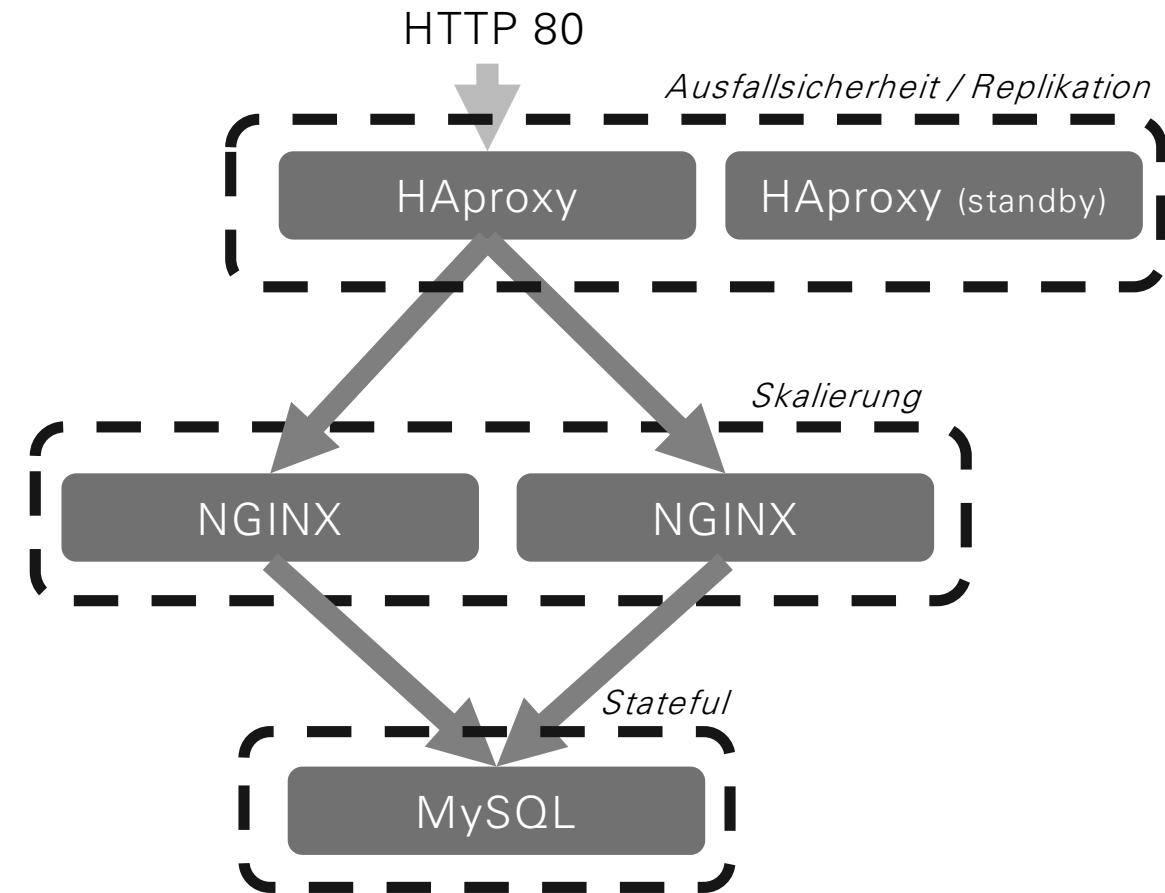
### Steuerungsaktivitäten im Cluster:

- Start von Containern auf Knoten (→ Scheduler)
- Verknüpfung von Containern
- ...

# Blaupause einer Anwendung (vereinfacht)

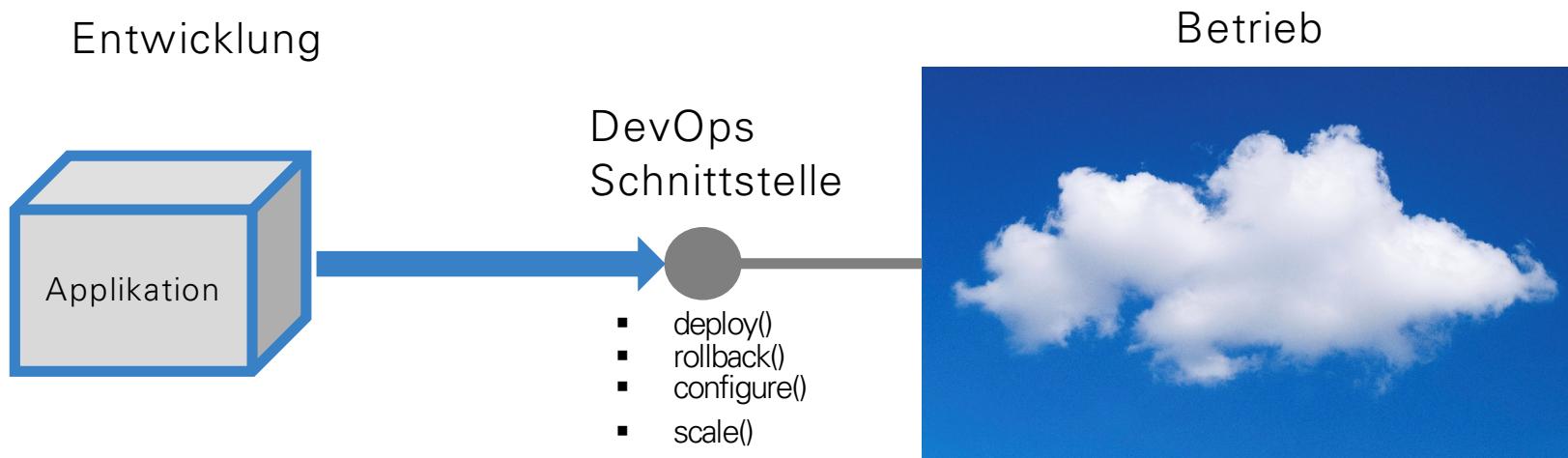


**Metamodell**



**Modell**

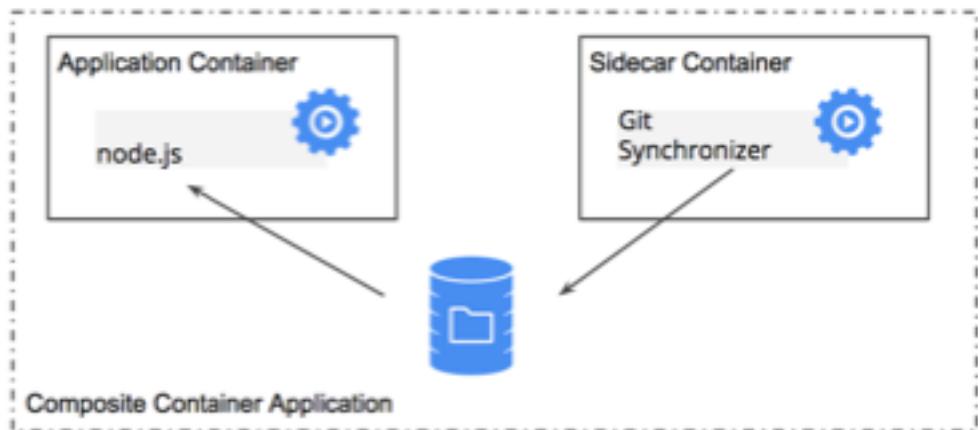
# Ein Cluster-Orchestrator bietet eine Schnittstelle zwischen Betrieb und Entwicklung für ein Cluster an.



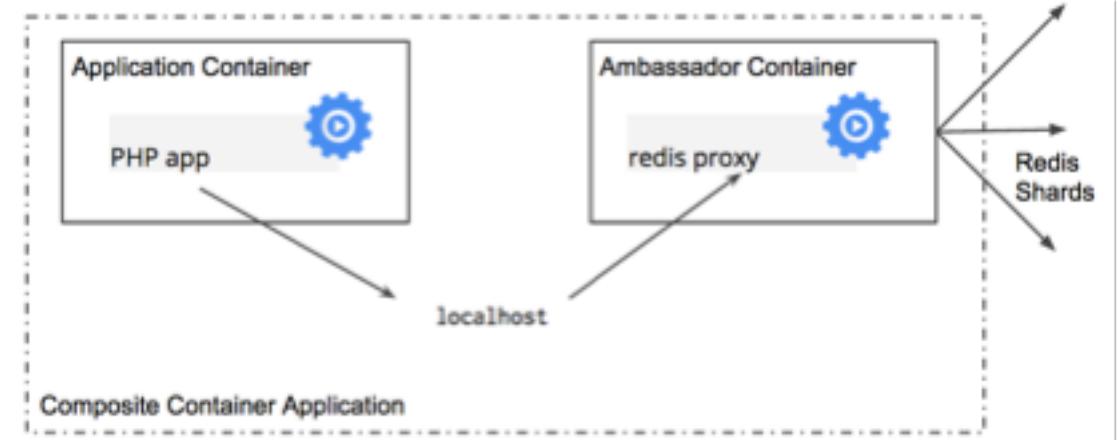
# Ein Cluster-Orchestrator automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster.

- Scheduling von Containern mit applikationsspezifischen Constraints (z.B. Deployment- und Start-Reihenfolgen, Gruppierung, ...)
- Aufbau von notwendigen Netzwerk-Verbindungen zwischen Containern.
- Bereitstellung von persistenten Speichern für zustandsbehaftete Container.
- (Auto-) Skalierung von Containern.
- Re-Scheduling von Containern im Fehlerfall (Auto-Healing) oder zur Performance-Optimierung.
- Container-Logistik: Verwaltung und Bereitstellung von Containern. Package-Management: Verwaltung und Bereitstellung von Applikationen.
- Bereitstellung von Administrationsschnittstellen (Remote-API, Kommandozeile).
- Management von Services: Service Discovery, Naming, Load Balancing.
- Automatismen für Rollout-Workflows wie z.B. Canary Rollout.
- Monitoring und Diagnose von Containern und Services.

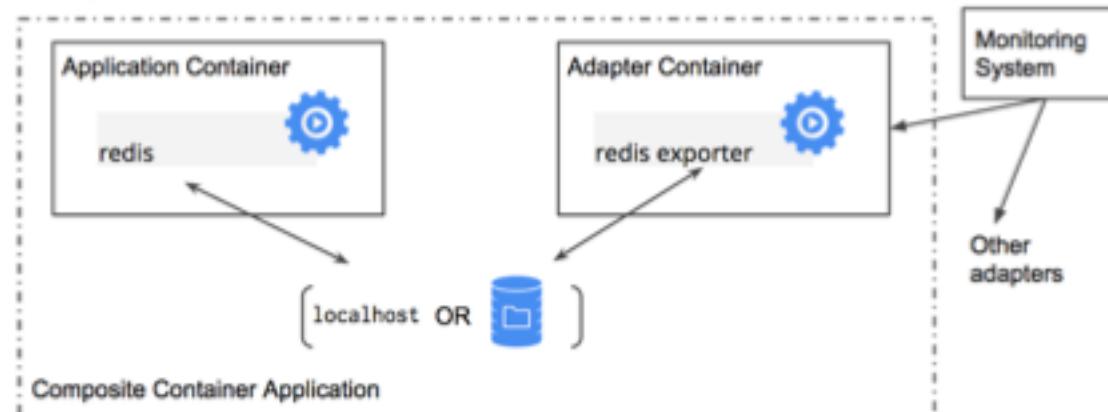
# Orchestrierungsmuster



Sidecar Container

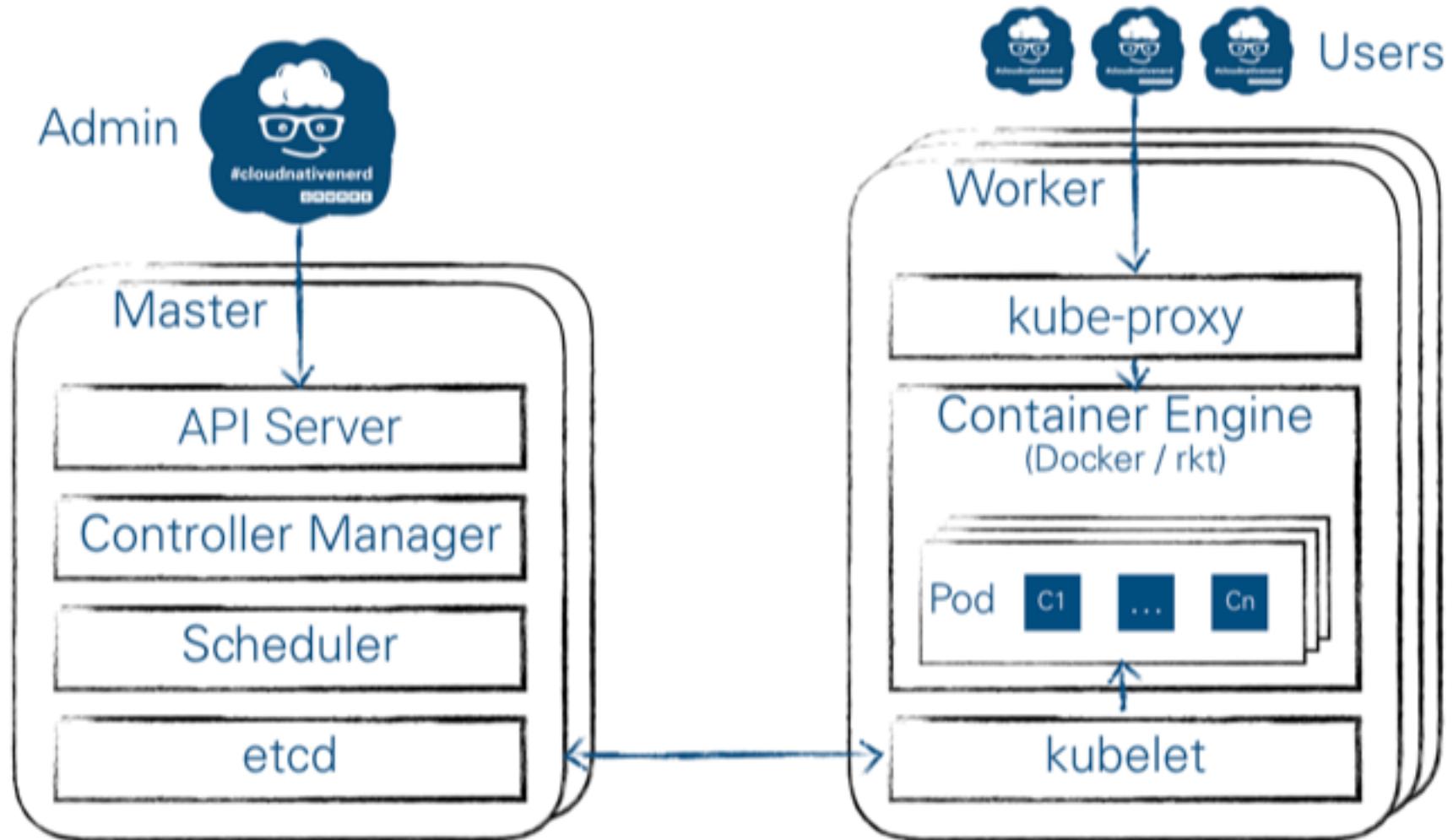


Ambassador Container

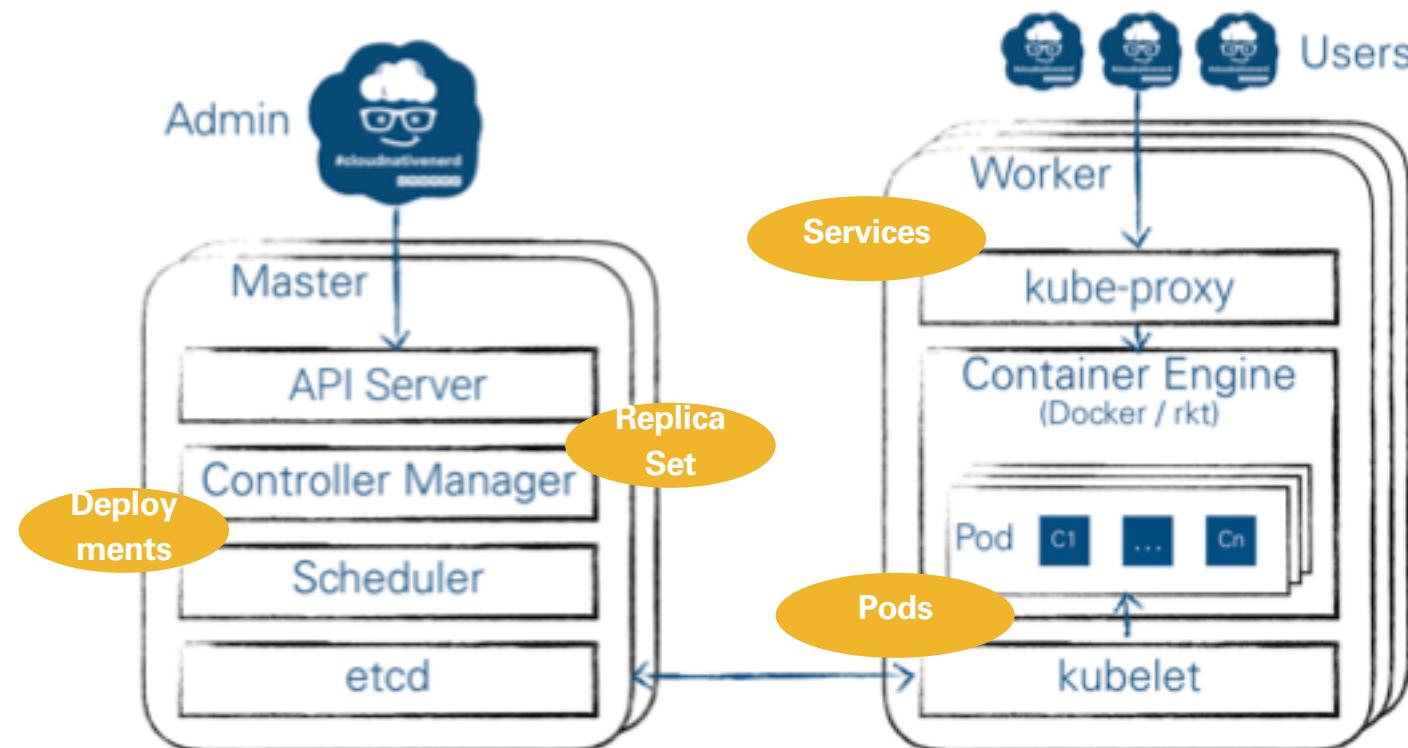


Adapter Container

# Architektur von Kubernetes.

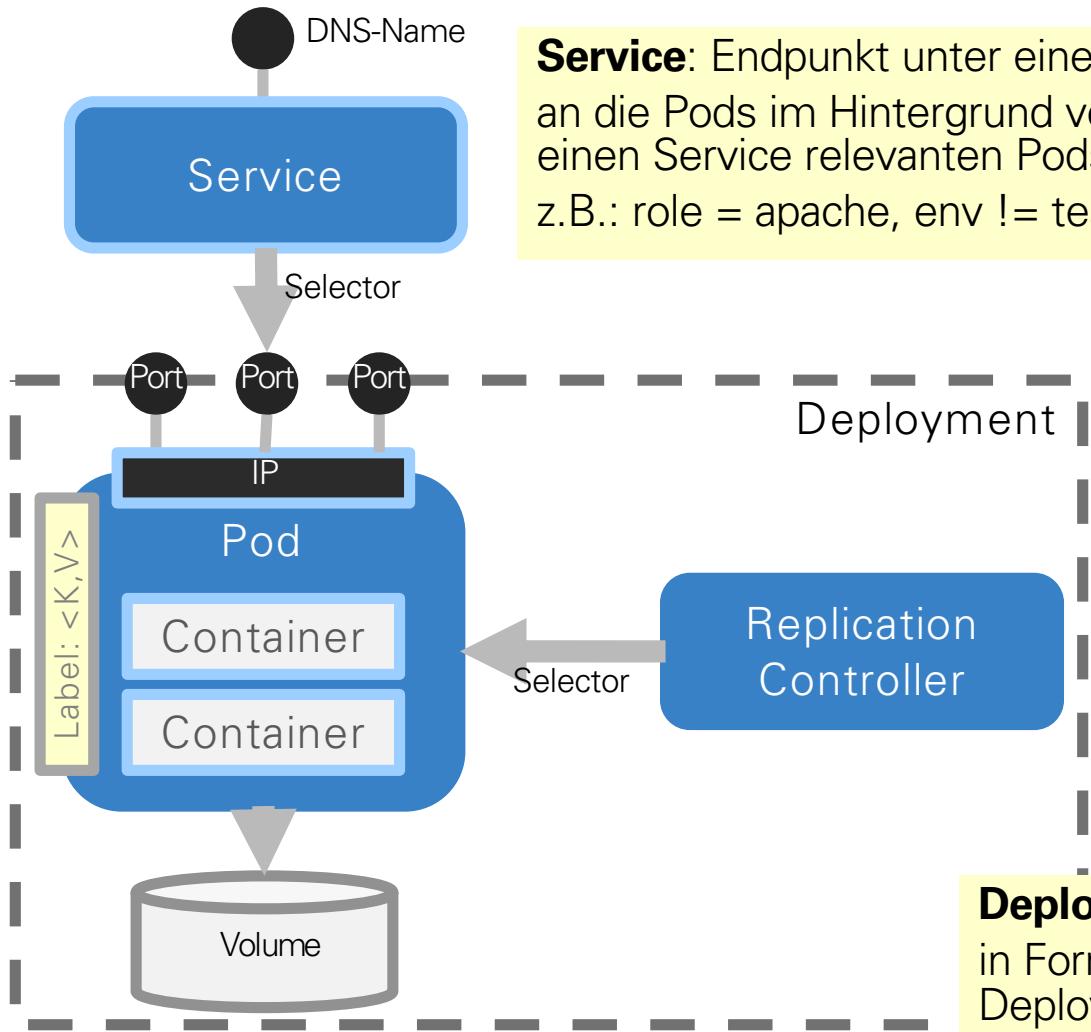


# Aufgaben der Kubernetes Bausteine.



- **API Server:** Stellt die REST API von Kubernetes zur Verfügung (Admin-Schnittstelle)
- **Controller Manager:** Verwaltet die Replica Sets / Replication Controller (stellt Anzahl Instanzen sicher) und Node Controller (prüfen Maschine & Pods)
- **Scheduler:** Cluster-Scheduler.
- **etcd:** Stellt einen zentralen Konfigurationsspeicher zur Verfügung.
- **Kubelet:** Führt Pods aus.
- **Container Engine:** Betriebssystem-Virtualisierung.
- **kube-proxy:** Stellt einen Service nach Außen zur Verfügung.

# Der Kern-Abstraktionen von Kubernetes.



**Service**: Endpunkt unter einem definierten DNS-Namen, der Aufrufe an die Pods im Hintergrund verteilt (Load Balancing, Failover). Die für einen Service relevanten Pods werden über ihre Labels selektiert, z.B.: role = apache, env != test, tier in (web, app)

**Pod**: Gruppe an Containern, die auf dem selben Knoten laufen und sich eine Netzwerk-Schnittstelle inklusive einer dedizierten IP, persistente Volumes und Umgebungsvariablen teilen. Ein Pod ist die atomare Scheduling-Einheit in K8s. Ein Pod kann über sog. *Labels* markiert werden, das sind frei definierbare Schlüssel-Wert-Paare.

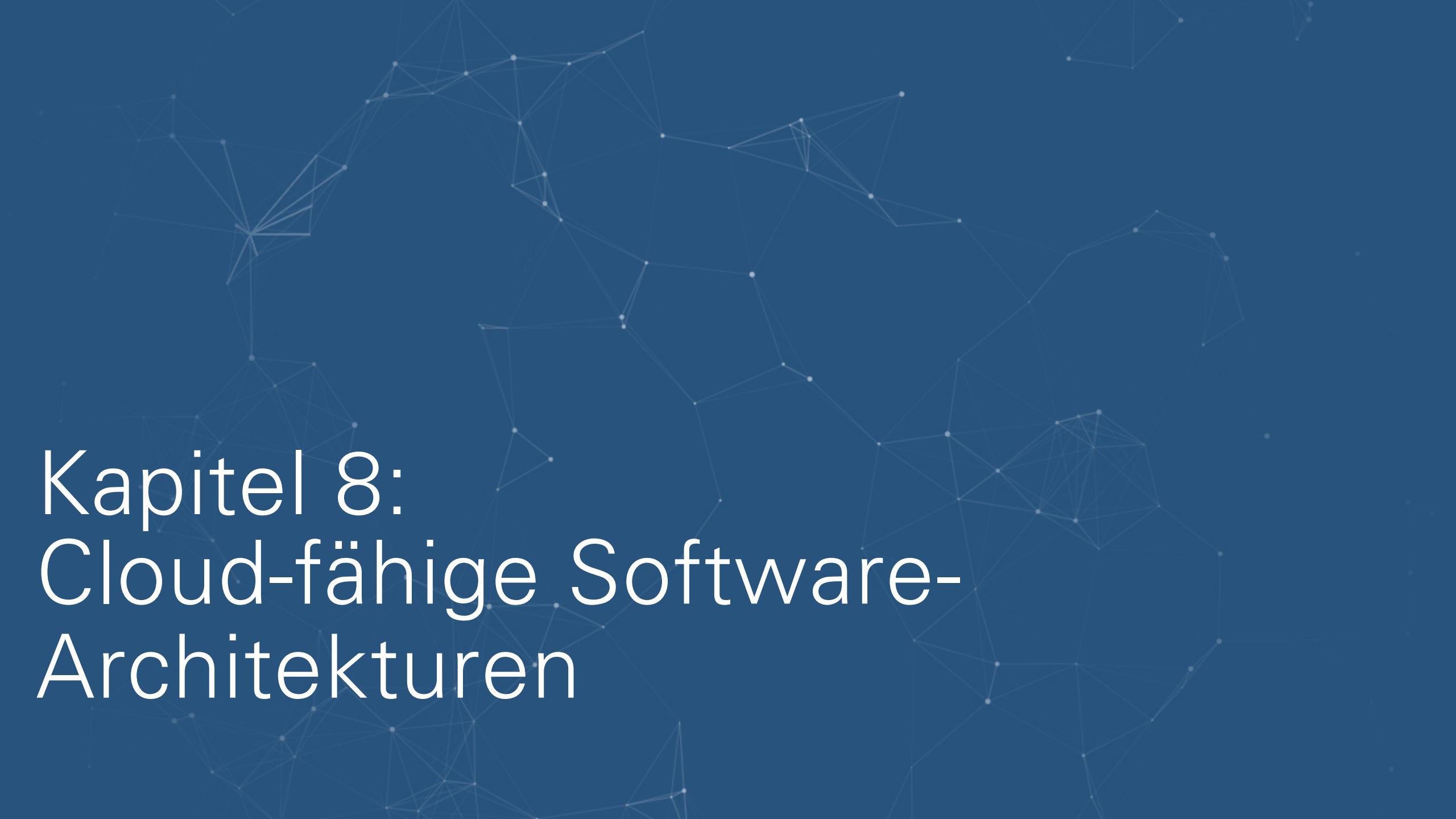
**Replication Controller**: stellen sicher, dass eine spezifizierte Anzahl an Instanzen pro Pod ständig läuft. Ist für Reaktionen im Fehlerfall (Re-Scheduling), Skalierung und Rollouts (Canary Rollouts, Rollout Tracks, ..) zuständig.

**Deployment**: Klammer um einen gewünschten Zielzustand im Cluster in Form eines Pods mit dazugehörigem Replication Controller. Ein Deployment bezieht sich nicht auf Services, da diese in der K8s-Philosophie einen von Pods unabhängigen Lebenszyklus haben.

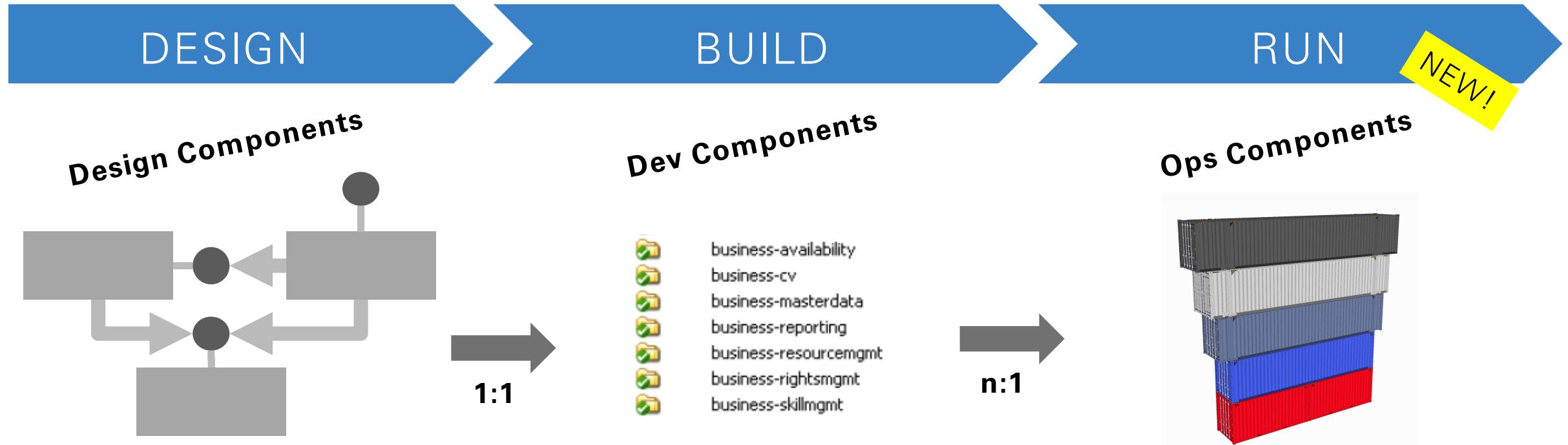
# Mögliche Klausurfragen

- In welchen typischen Fällen führt ein Orchestrator ein Re-Scheduling von Containern durch?
- Welche Orchestrierungsmuster kennen sie? Geben sie jeweils ein Beispiel für einen möglichen Use Case.
- Erläutern sie das Konzept eines Replication Controllers in Kubernetes.
- Was versteht man unter Auto-Skalierung?

# Kapitel 8: Cloud-fähige Software- Architekturen



# Cloud Native Application Development: Components All Along the Software Lifecycle.

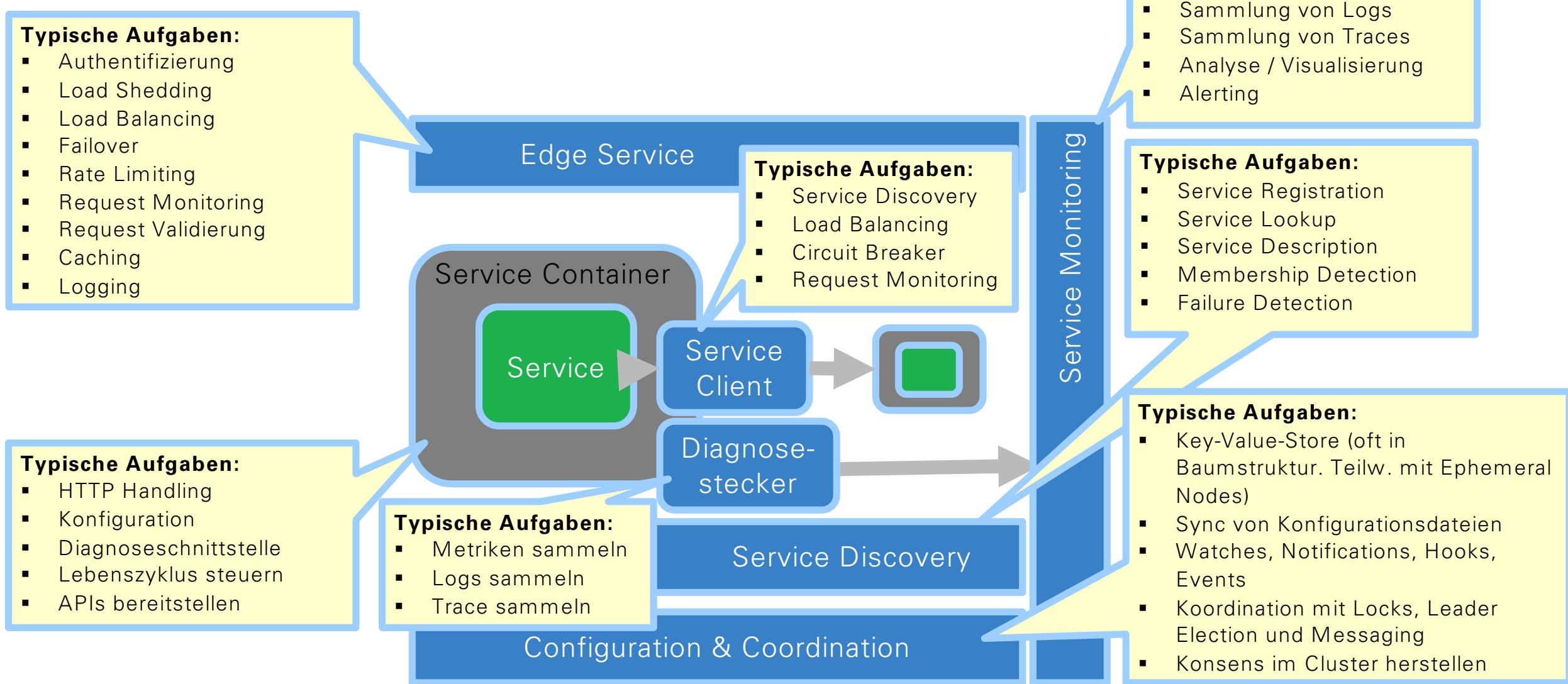


- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

- Planning unit
- Team assignment unit
- Knowledge unit
- Development unit
- Integration unit

- Release unit
- Deployment unit
- Runtime unit  
(crash, slow-down, access)
- Scaling unit

# Betriebskomponenten benötigen eine Infrastruktur um sie herum: Eine Micro-Service-Plattform.



# Gossip Protokolle: Inspiriert von der Verbreitung von Tratsch in sozialen Netzwerken.

Grundlage: Ein Netzwerk an Agenten mit eigenem Zustand

Agenten verteilen einen Gossip-Strom

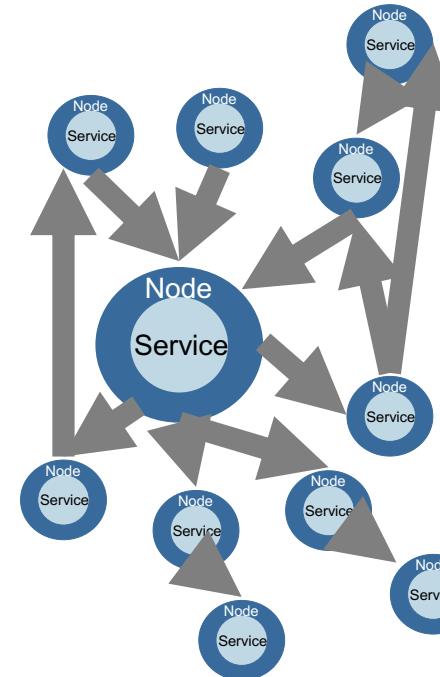
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

Virale Verbreitung des Gossip-Stroms

- Knoten, die mit mir in einer Gruppe sind, bekommen auf jeden Fall eine Nachricht
- Die Top x% an Knoten, die mir Nachrichten schicken bekommen eine Nachricht

Nachrichten, denen vertraut wird, werden in den lokalen Zustand übernommen

- Die gleiche Nachricht wurde von mehreren Seiten gehört
- Die Nachricht stammt von Knoten, denen der Agent vertraut
- Es ist keine aktuellere Nachricht vorhanden



Vorteile:

- Keine zentralen Einheiten notwendig.
- Fehlerhafte Partitionen im Netzwerk werden umschifft. Die Kommunikation muss nicht verlässlich sein.

Nachteile:

- Der Zustand ist potenziell inkonsistent verteilt (konvergiert aber mit der Zeit)
- Overhead durch redundante Nachrichten.

# Protokolle für verteilten Konsens sind im Gegensatz zu Gossip-Protokollen konsistent aber nicht hoch-verfügbar.

Grundlage: Netzwerk an Agenten

Prinzip: Es reicht, wenn der Zustand auf einer einfachen Mehrheit der Knoten konsistent ist und die restlichen Knoten ihre Inkonsistenz erkennen.

Verfahren:

- Das Netzwerk einigt sich per einfacher Mehrheit auf einen Leader-Agenten – initial und falls der Leader-Agent nicht erreichbar ist. Eine Partition in der Minderheit kann keinen Leader-Agenten wählen.
- Alle Änderungen laufen über den Leader-Agenten. Dieser verteilt per Multicast Änderungsnachrichten periodisch im festen Takt an alle weiteren Agenten.
- Quittiert die einfache Mehrheit an Agenten die Änderungsnachricht, so wird die Änderung im Leader und (per Nachricht) auch in den Agenten aktiv, die quittiert haben. Ansonsten wird der Zustand als inkonsistent angenommen.

Konkrete Konsens-Protokolle: Raft, Paxos

Vorteile:

- Fehlerhafte Partitionen im Netzwerk werden toleriert und nach Behebung des Fehlers wieder automatisch konsistent.
- Streng konsistente Daten.

Nachteile:

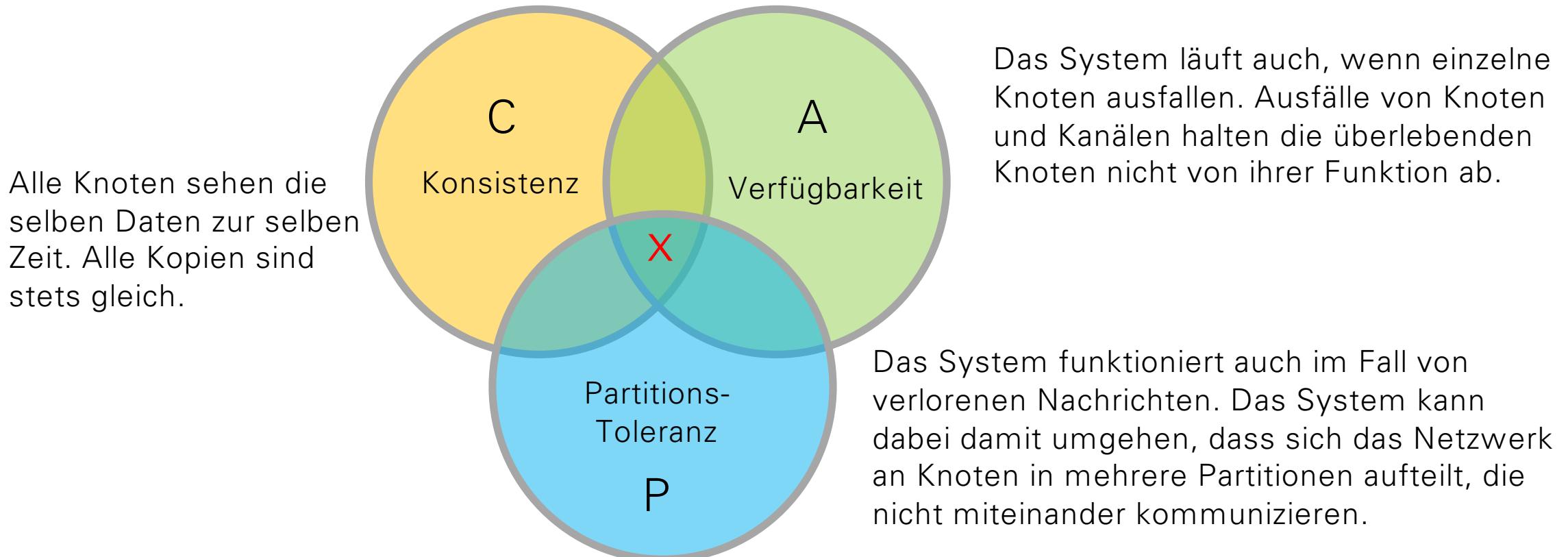
- Der zentrale Leader-Agent limitiert den Durchsatz an Änderungen.
- Nicht hoch-verfügbar: Bei einer Netzwerk-Partition kann die kleinere Partition nicht weiterarbeiten. Ist die Mehrheit in keiner Partition, so kann insgesamt nicht weiter gearbeitet werden.

# Das CAP Theorem.

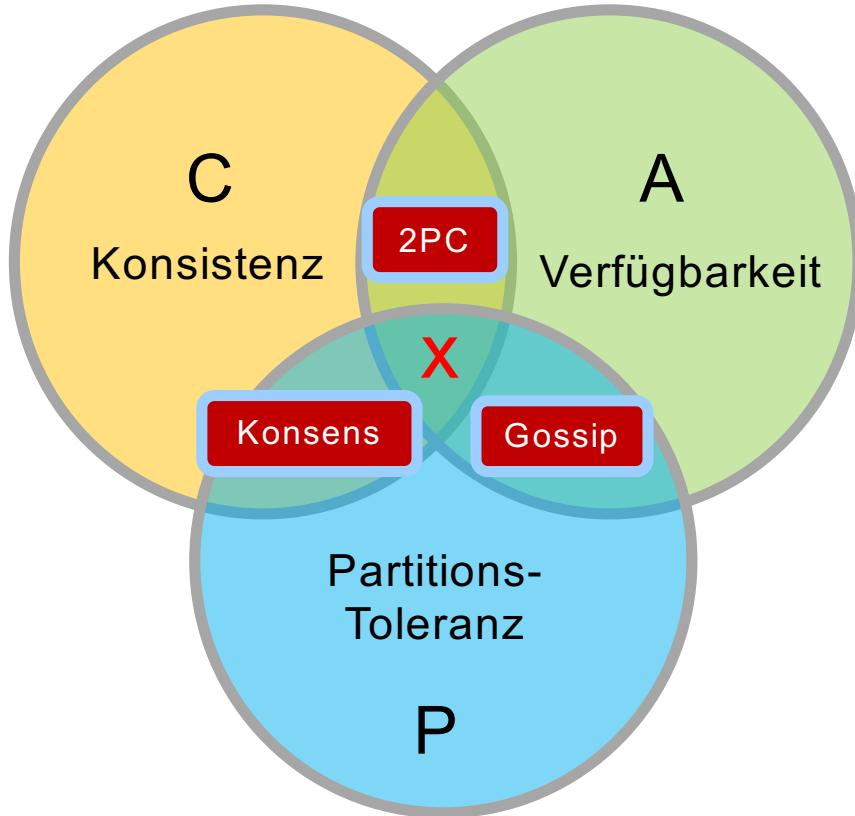
Theorem von Brewer für Eigenschaften von zustandsbehafteten verteilten Systemen – mittlerweile auch formal bewiesen.

Brewer, Eric A. "Towards robust distributed systems." *PODC*. 2000.

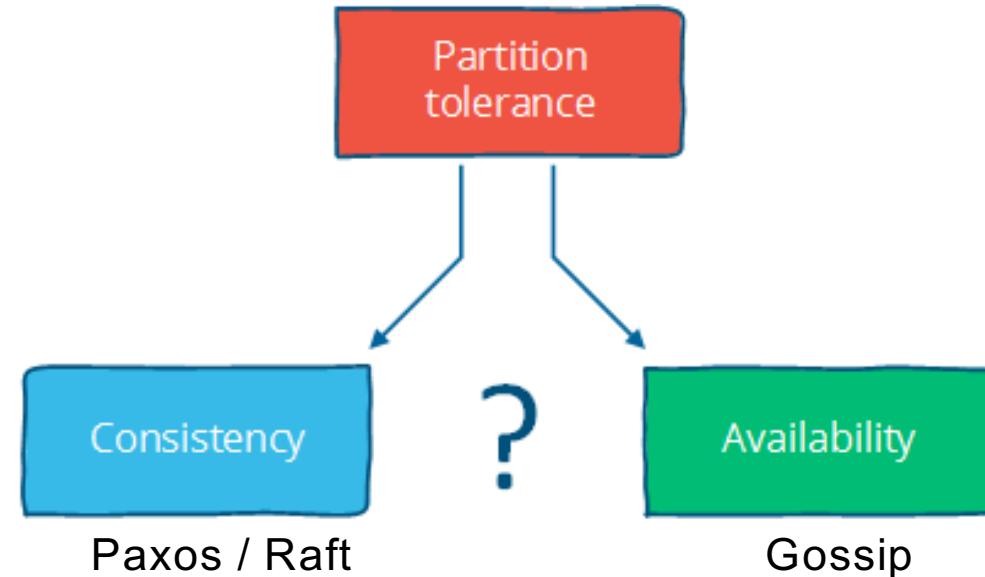
Es gibt drei wesentliche Eigenschaften, von denen ein verteiltes System nur zwei gleichzeitig haben kann:



# Die vorgestellten Protokolle und das CAP Theorem.



In der Cloud müssen Partitionen angenommen werden.  
Damit ist die Entscheidung binär zwischen Konsistenz und  
Verfügbarkeit.



**HYPERSCALE**  
(traffic, data, features)

**ANTIFRAGILITY**  
(resiliency & autonomy)

# CLOUD NATIVE APPLICATIONS

**PRODUCTIVITY**  
(continuous delivery & devops)

**OPEX SAVINGS**  
(utilization & automation)

BUILD AND COMPOSED  
AS MICROSERVICES

# CLOUD NATIVE APPLICATIONS

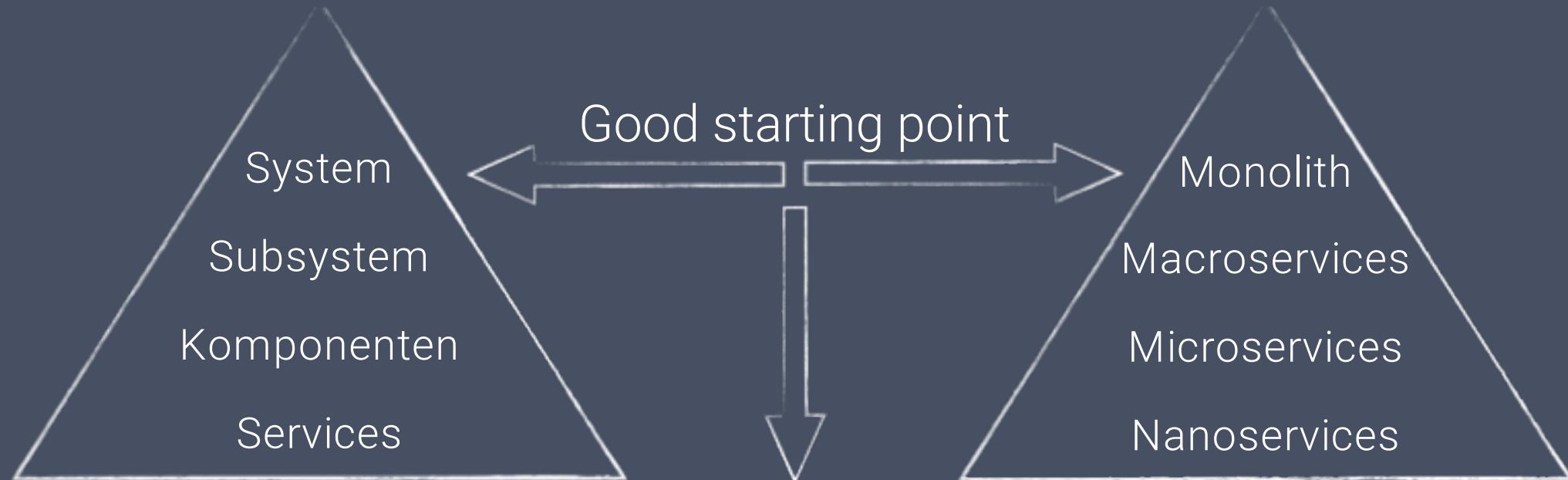
PACKAGED AND  
DISTRIBUTED AS CONTAINERS

DYNAMICALLY  
EXECUTED IN THE CLOUD

# Dev Components



# Ops Components

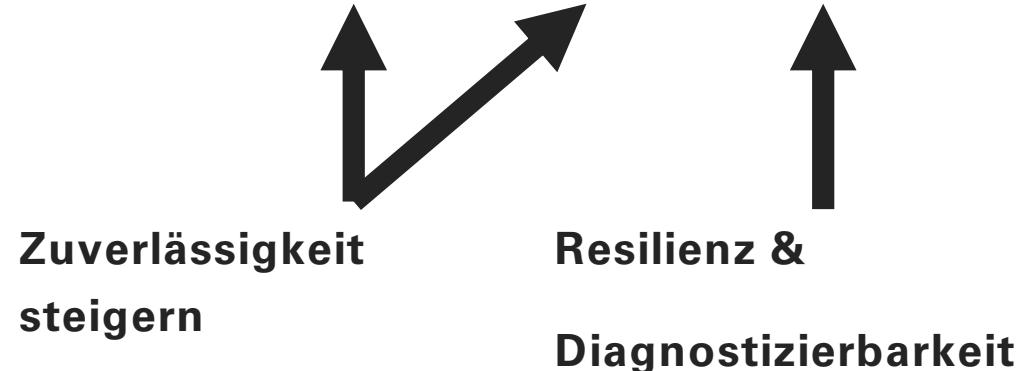


## Decomposition Trade-Offs

- |  |  |
|--|--|
| + Independent releases, deployments, teams | - Distribution debt                      |
| + Short roundtrips                         | - Increased infrastructure complexity    |
| + Runtime isolation (crash, slow-down)     | - Increased troubleshooting complexity   |
| + More flexible to scale                   | - Increased integration complexity       |
| + Higher resources utilisation             | - Higher resource overhead (potentially) |

# Resilienz

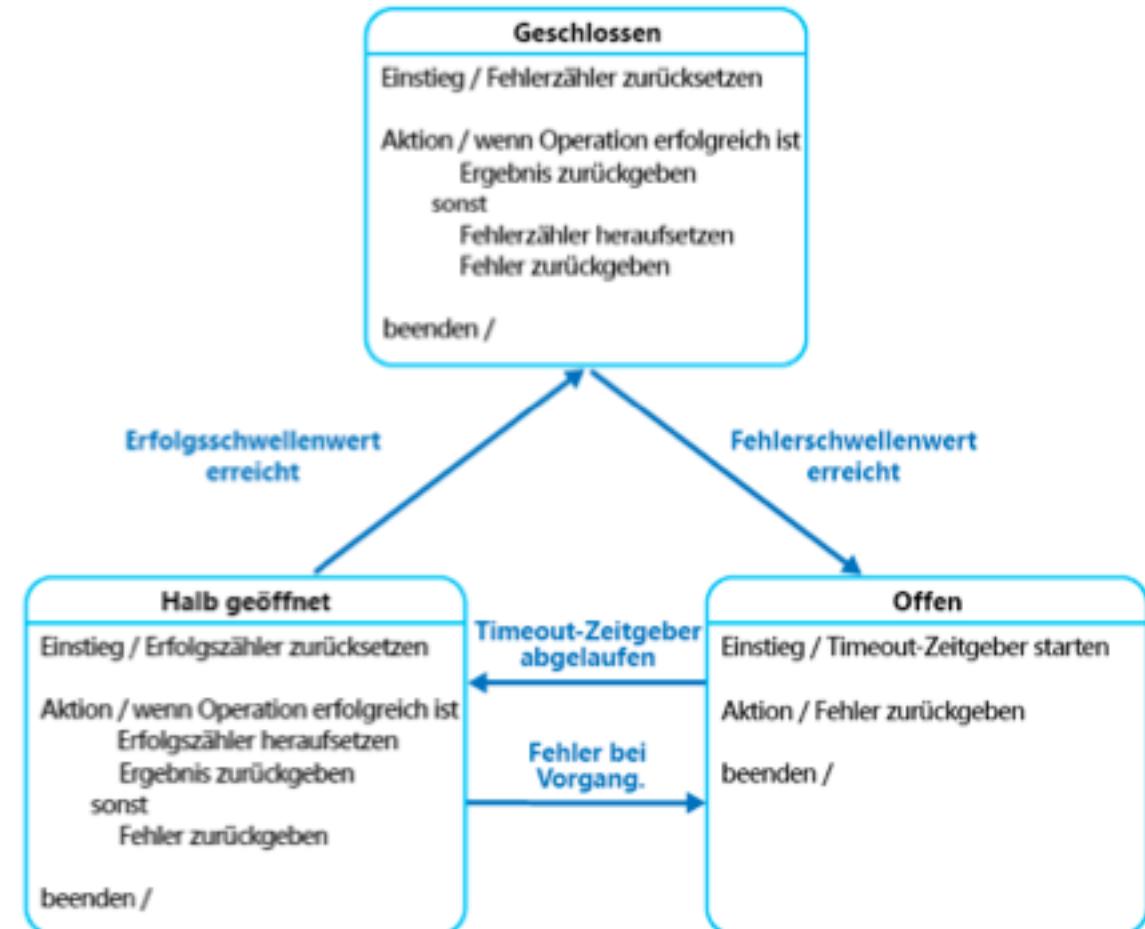
$$\text{Verfügbarkeit} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$



**Resilienz:** Die Fähigkeit eines Systems mit unerwarteten und fehlerhaften Situationen umzugehen

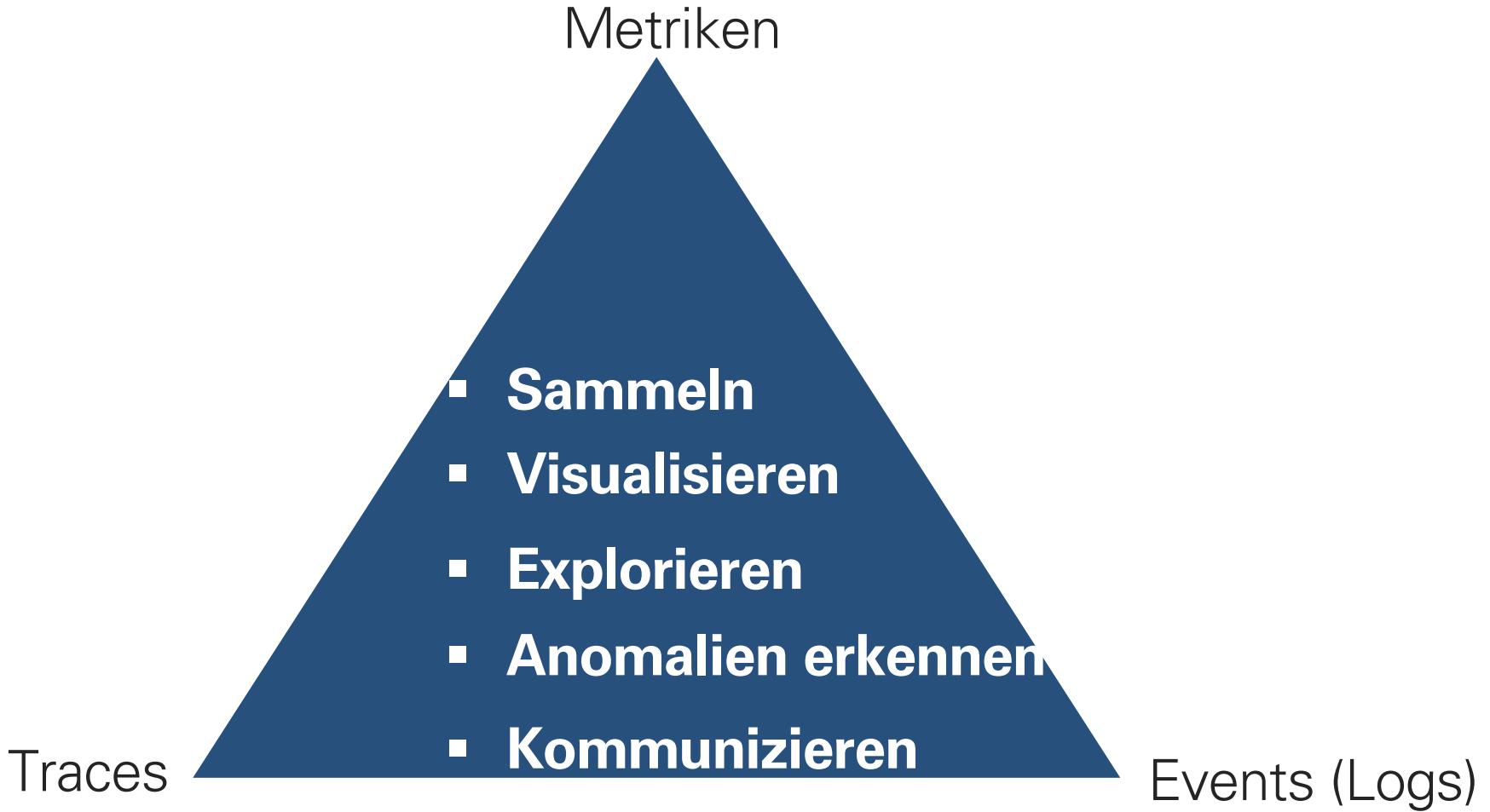
- Ohne dass es der Nutzer merkt (Bestfall)
- Mit einer „graceful degradation“ des Services (schlechtester Fall)

Resilienz-Pattern: Circuit Breaker



Weitere Patterns: <https://docs.microsoft.com/de-de/azure/architecture/patterns/category/resiliency>

# Dagnostizierbarkeit



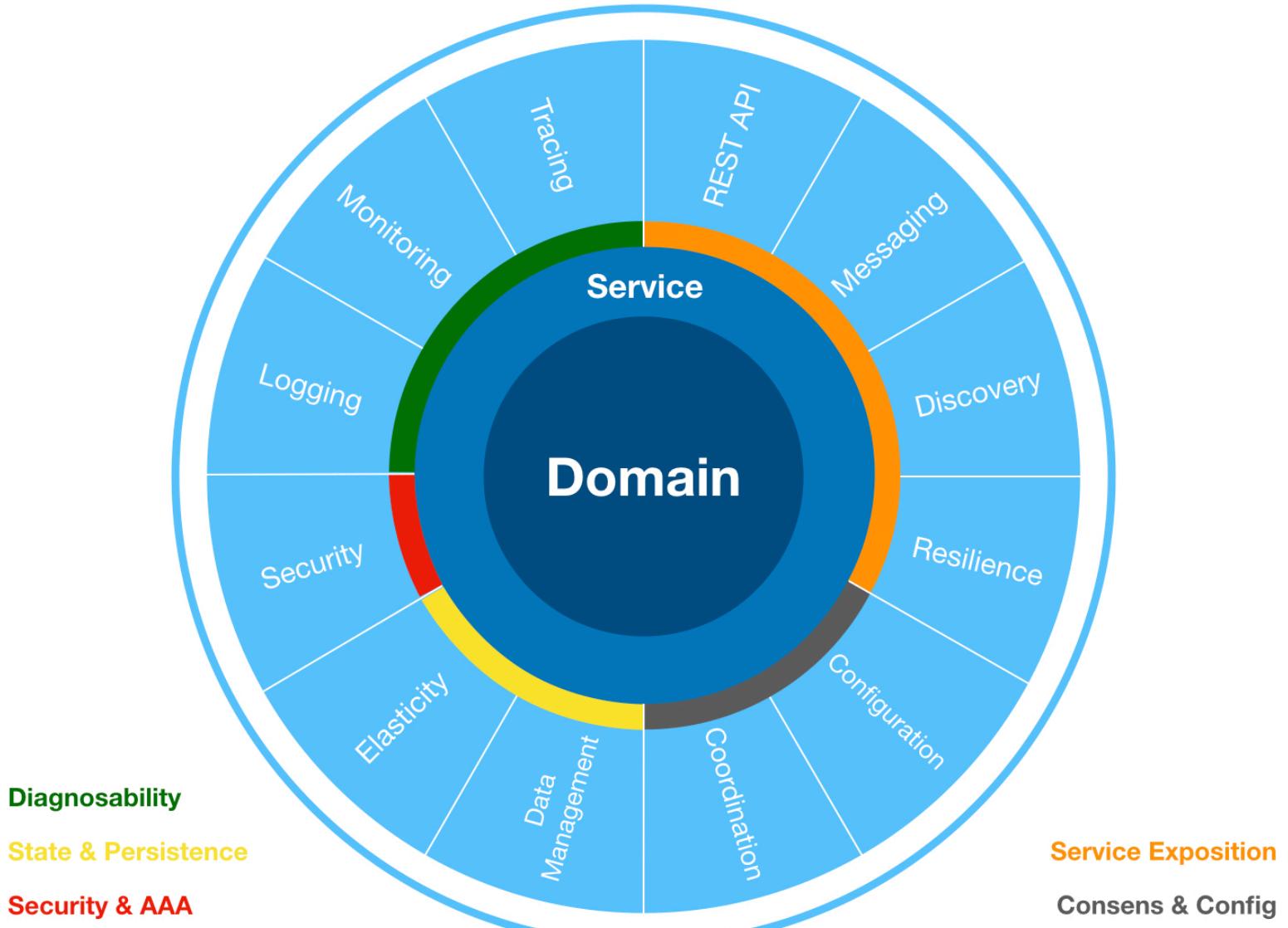
# 12 Factor App

1	<b>Codebase</b> One codebase tracked in revision control, many deploys.	7	<b>Port binding</b> Export services via port binding.
2	<b>Dependencies</b> Explicitly declare and isolate dependencies.	8	<b>Concurrency</b> Scale out via the process model.
3	<b>Configuration</b> Store config in the environment.	9	<b>Disposability</b> Maximize robustness with fast startup and graceful shutdown.
4	<b>Backing Services</b> Treat backing services as attached resources.	10	<b>Dev/Prod Parity</b> Keep development, staging, and production as similar as possible
5	<b>Build, release, run</b> Strictly separate build and run stages.	11	<b>Logs</b> Treat logs as event streams.
6	<b>Processes</b> Execute the app as one or more stateless processes.	12	<b>Admin processes</b> Run admin/management tasks as one-off processes.

<https://12factor.net/de>

<https://www.slideshare.net/Alicanakku1/12-factor-apps>

# Technische Aspekte von Microservices



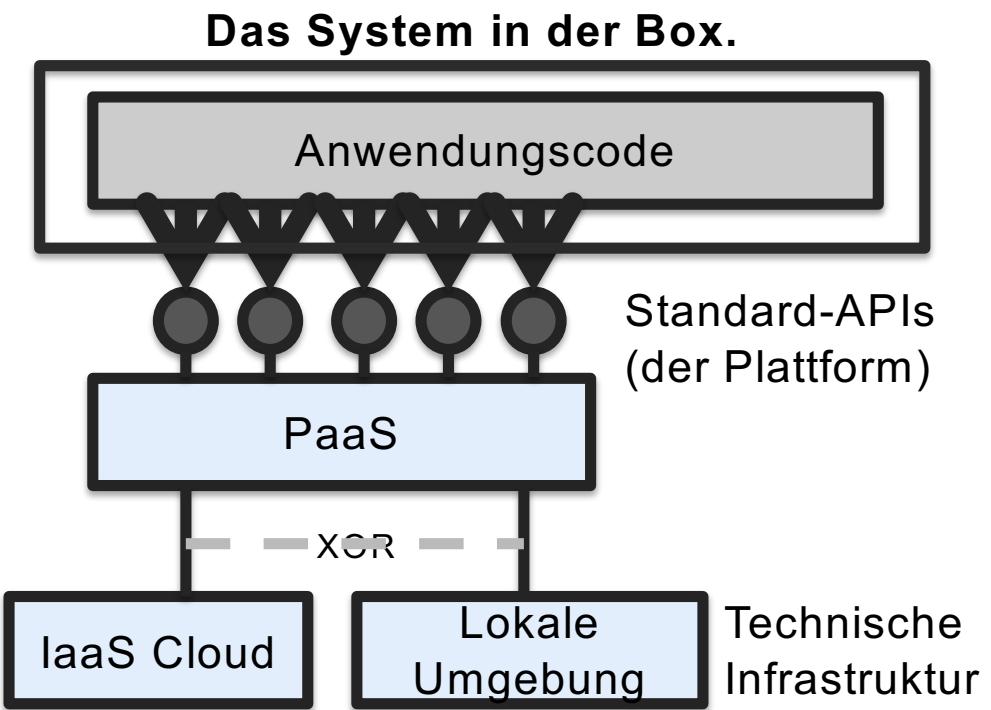
# Mögliche Klausurfragen

- Welche typischen Aufgaben übernimmt ein Edge Service in einer Cloud Architektur? Nennen sie mindestens 3 Aufgaben.
- Was besagt das CAP-Theorem?
- Nennen sie die Protokollart für einen verteilten Konfigurationsspeicher, die sowohl konsistent als auch partitions-tolerant ist.

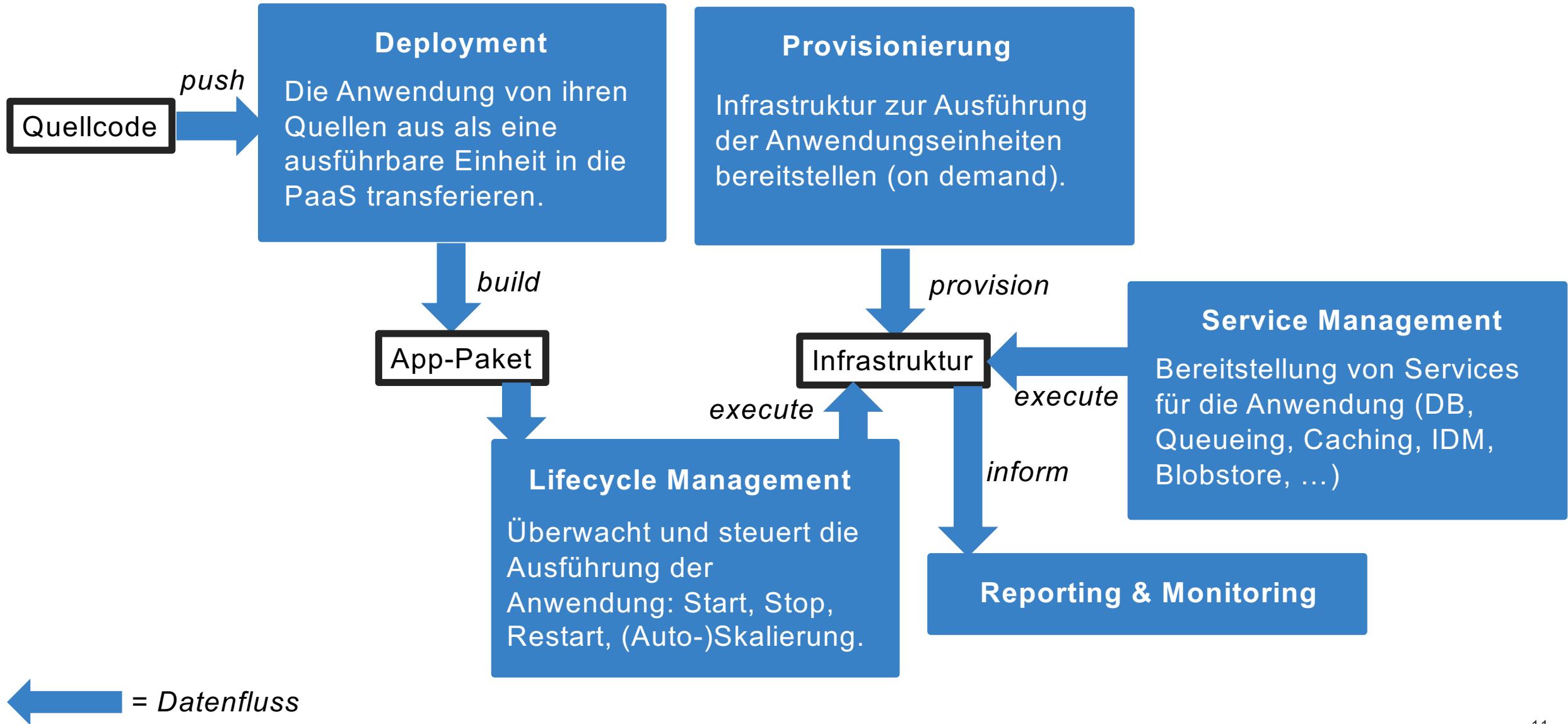
# Kapitel 9: Platform-as-a-Service

# Die Lösung: Plattform-as-a-Service 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. „Engine and Operating System should not matter....“.
- Es erfolgt eine automatische Skalierung der Anwendung.
- Entwicklungswerkzeuge (insb. Plugins für IDEs und Buildsysteme sowie eine lokale Testumgebung) stehen zur Verfügung: „deploy to cloud“.
- Die Plattform bietet eine Schnittstelle zur Administration und zum Monitoring der Anwendungen.



# Die funktionalen Bausteine einer PaaS Cloud.



# Mögliche Klausurfragen

- Wie wird eine Anwendung auf einer PaaS deployed?
- Wie sind die Begriffe Deployment und Provisionierung in Bezug auf die Building Blocks einer PaaS Cloud zu unterscheiden?

# Kapitel 10:

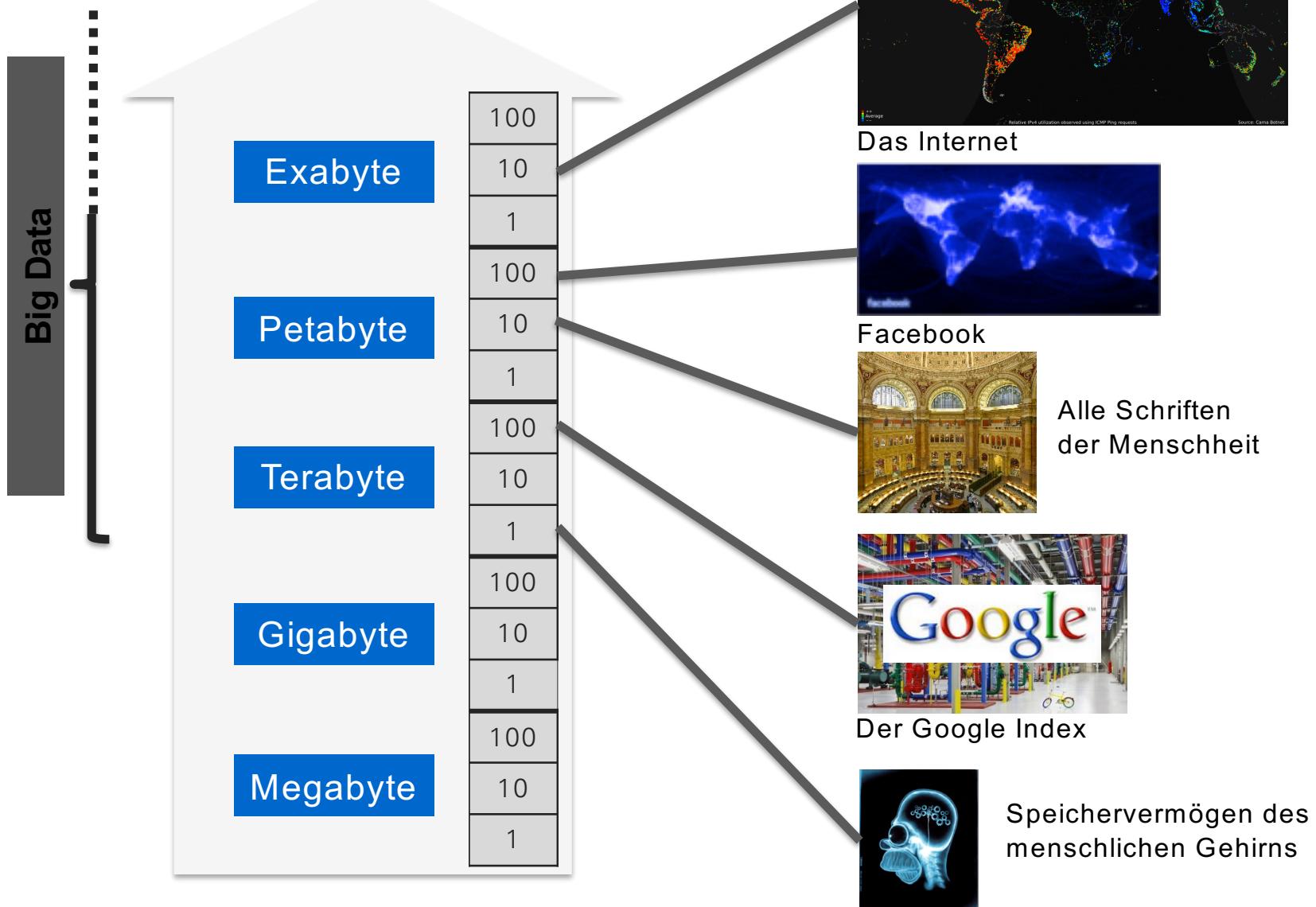
# Big Data

# Big Data

## Big Data

Verarbeitung großer Datenmengen durch:

- verteilte und hochgradig parallelisierte Verarbeitung.
- verteilte und effizient organisierte Datenablagen.



# Wie verwalte und erschließe ich große Datenmengen?

**Die Cloud Computing Antwort:**  
Ich verteile sie auf viele Rechner  
in der Cloud und schaffe eine  
übergreifende  
Zugriffsschnittstelle.



# Große Datenmengen können effizient nur von parallelen Algorithmen verarbeitet werden.

**Ein Algorithmus ist genau dann parallelisierbar, wenn er in einzelne Teile zerlegt werden kann, die keine Seiteneffekte zueinander haben.**

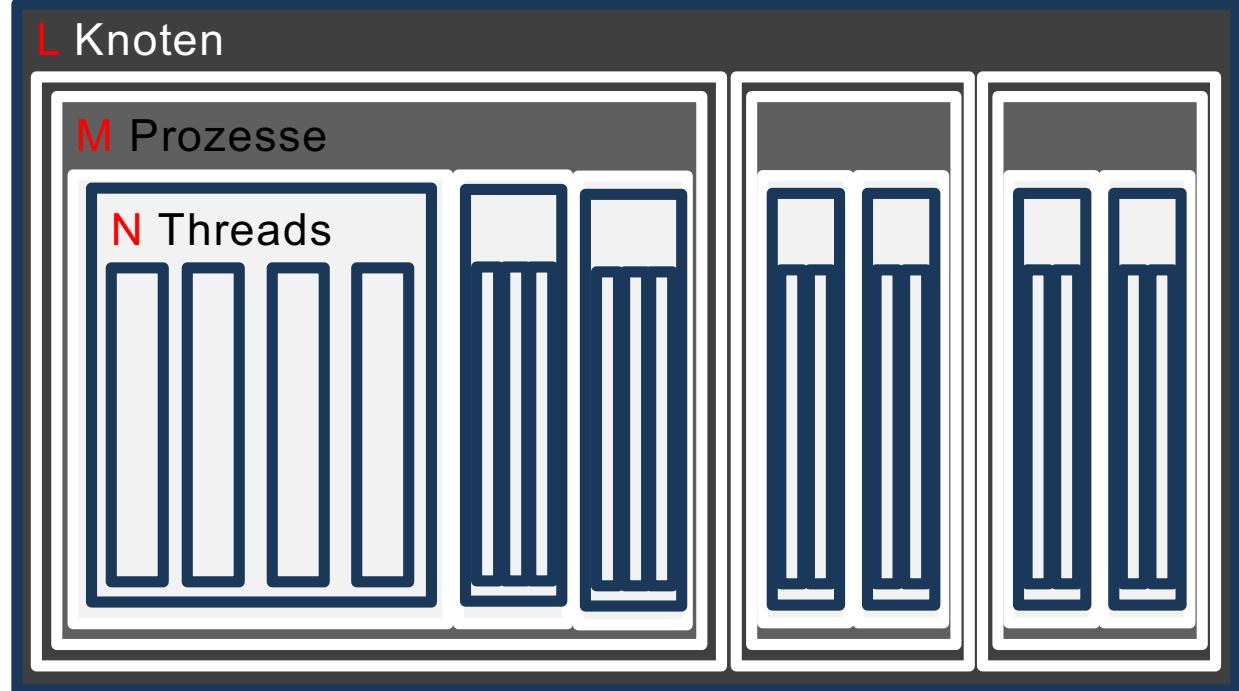
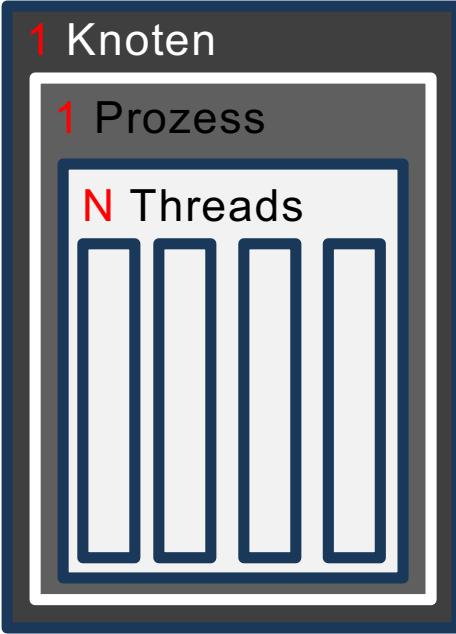
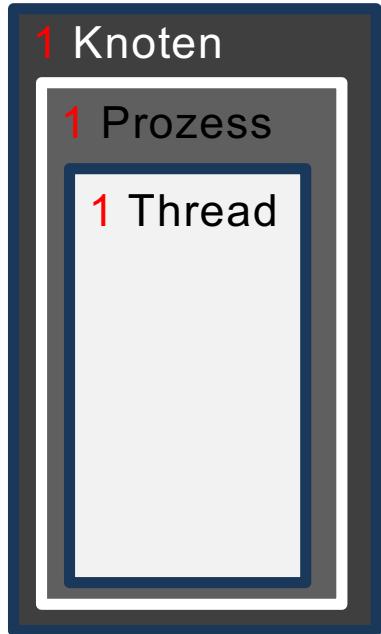
- Funktioniert gut: Quicksort. Aufwand:  $O(n \log n) \rightarrow O(\log n)$

```
private void QuicksortParallel<T>(T[] arr, int left, int right)
where T : IComparable<T>
{
    if (right > left)
    {
        int pivot = Partition(arr, left, right);
        Parallel.Do(
            () => QuicksortParallel(arr, left, pivot - 1),
            () => QuicksortParallel(arr, pivot + 1, right));
    }
}
```

- Funktioniert nicht: Berechnung der Fibonacci-Folge ( $F_{k+2} = F_k + F_{k+1}$ ). Berechnung ist nicht parallelisierbar.

**Ein paralleler Algorithmus (Job) ist aufgeteilt in sequenzielle Berechnungsschritte (Tasks), die parallel zueinander abgearbeitet werden können.** Der Entwurf von parallelen Algorithmen folgt oft dem Teile-und-Herrsche Prinzip.

# Parallele Programmierung kann sowohl im Kleinen als auch im Großen betrieben werden.



Keine Parallelität



Parallelität im Kleinen

Vorteile im Vergleich:

- Höherer Durchsatz
- Bessere Auslastung der Hardware
- Vertikale Skalierung möglich



Parallelität im Großen

Vorteile im Vergleich:

- Höherer Durchsatz
- Horizontale Skalierung möglich (Scale Out).
- Keine hardwarebedingte Limitierung des Datenvolumens (→ Big Data ready).

# Big Data erfordert Parallelität im Großen. Die vier Paradigmen der Parallelität im Großen:



Folgt aus Datenmenge  
im Vergleich zur  
Programmgröße

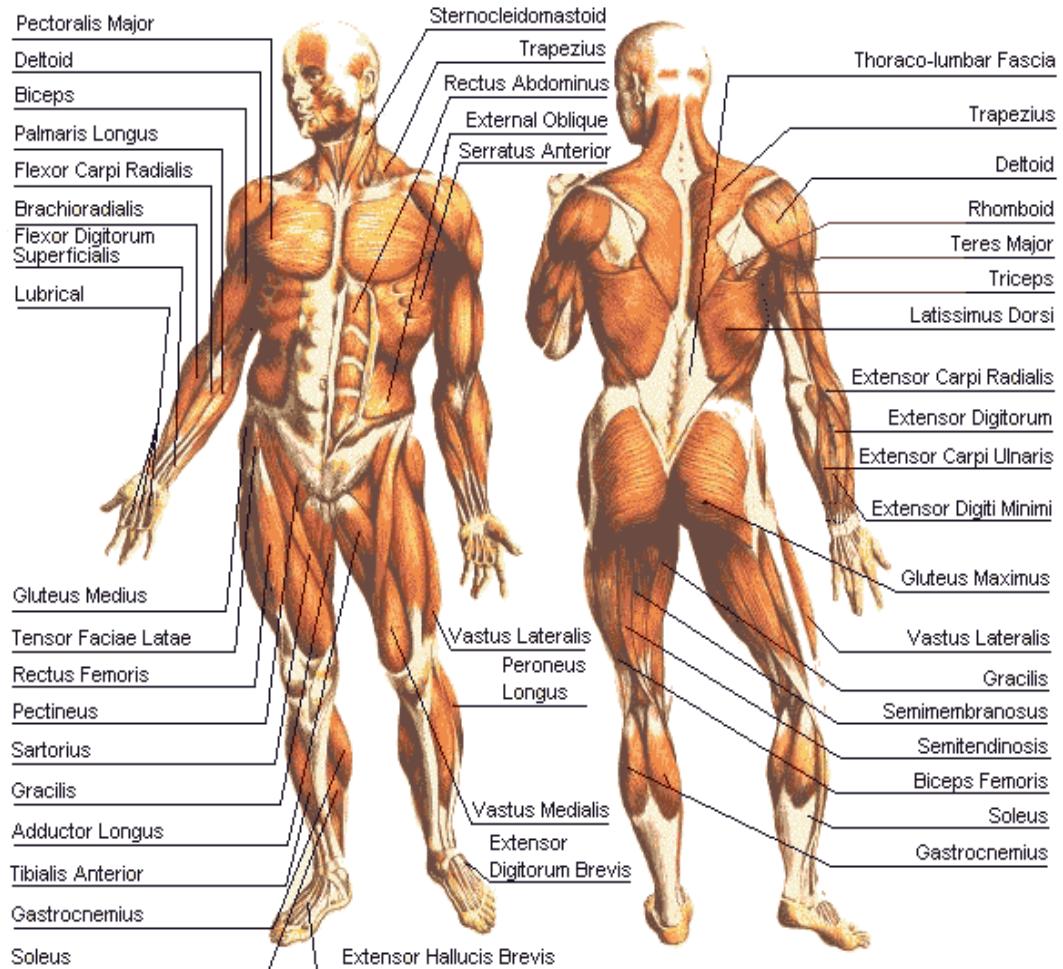
Das Grundprinzip von  
paralleler Verarbeitung.

Folgt aus Praxisanforderung:  
Viele Knoten  
bedeutet  
viele Ausfallmöglichkeiten

1. Die Logik folgt den Daten.
2. Falls Datentransfer notwendig, dann so schnell wie möglich:  
In-Memory vor lokaler Festplatte vor Remote-Transfer.
3. Parallelisierung über *Tasks* (seiteneffektfreie Funktionen) und *Jobs* (Ausführungsvorschrift für Tasks) sowie entsprechend partitionierter Daten (*Shards*).
4. Design for Failure: Ausführungsfehler als Standardfall ansehen und verzeihend und kompensierend sein.

Folgt aus potenziell großer  
Datenmenge und  
Verarbeitungs-  
geschwindigkeit

# Die Anatomie von Big Data Datenbanken

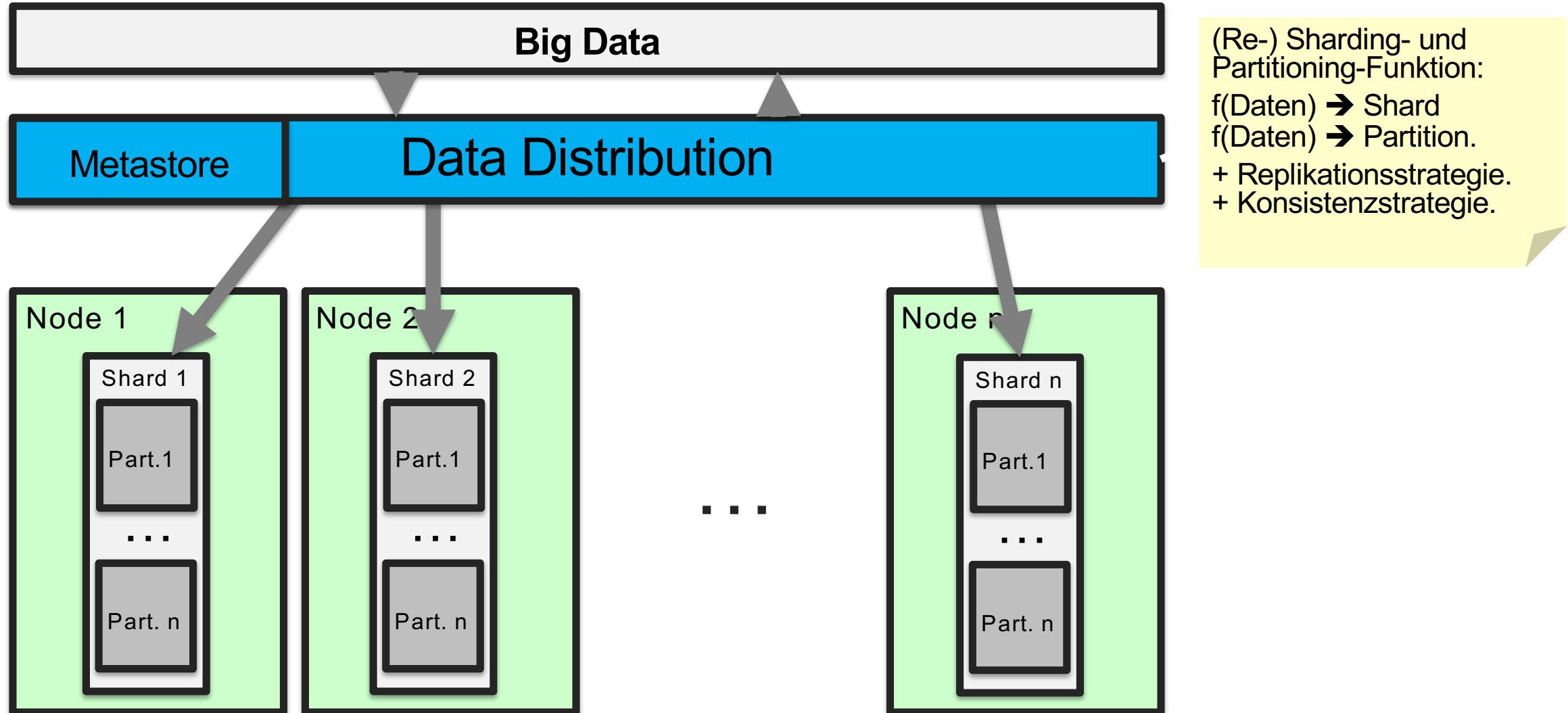


Query Distribution

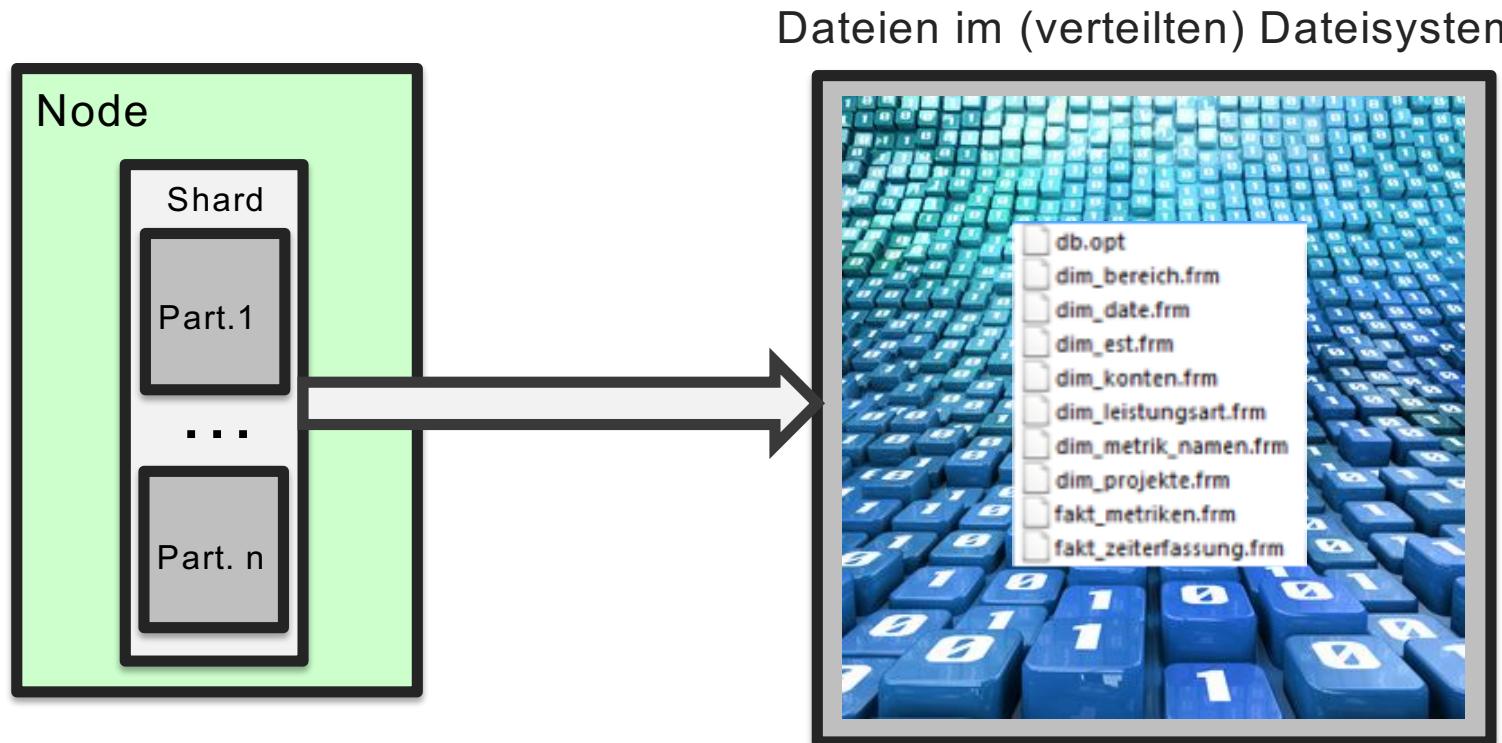
Data Distribution

Data Persistence

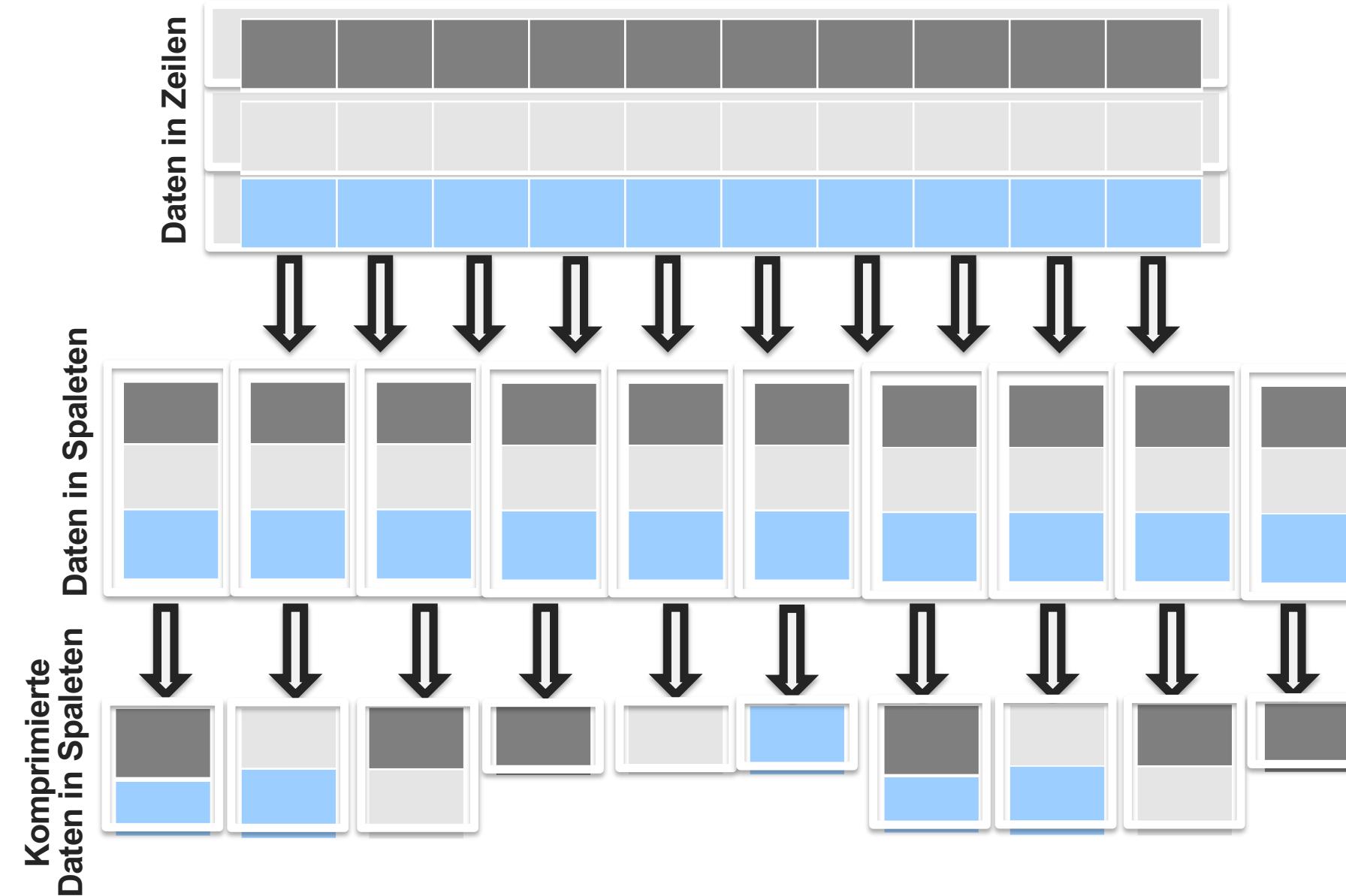
# Sharding and Partitioning: Verteilung und Stückelung von großen Datenmengen.



# Wie werden große Datenmengen technisch so gespeichert, dass eine schnelle Scan-Geschwindigkeit erreicht wird?



# Spalten-orientierte Datenspeicherung.

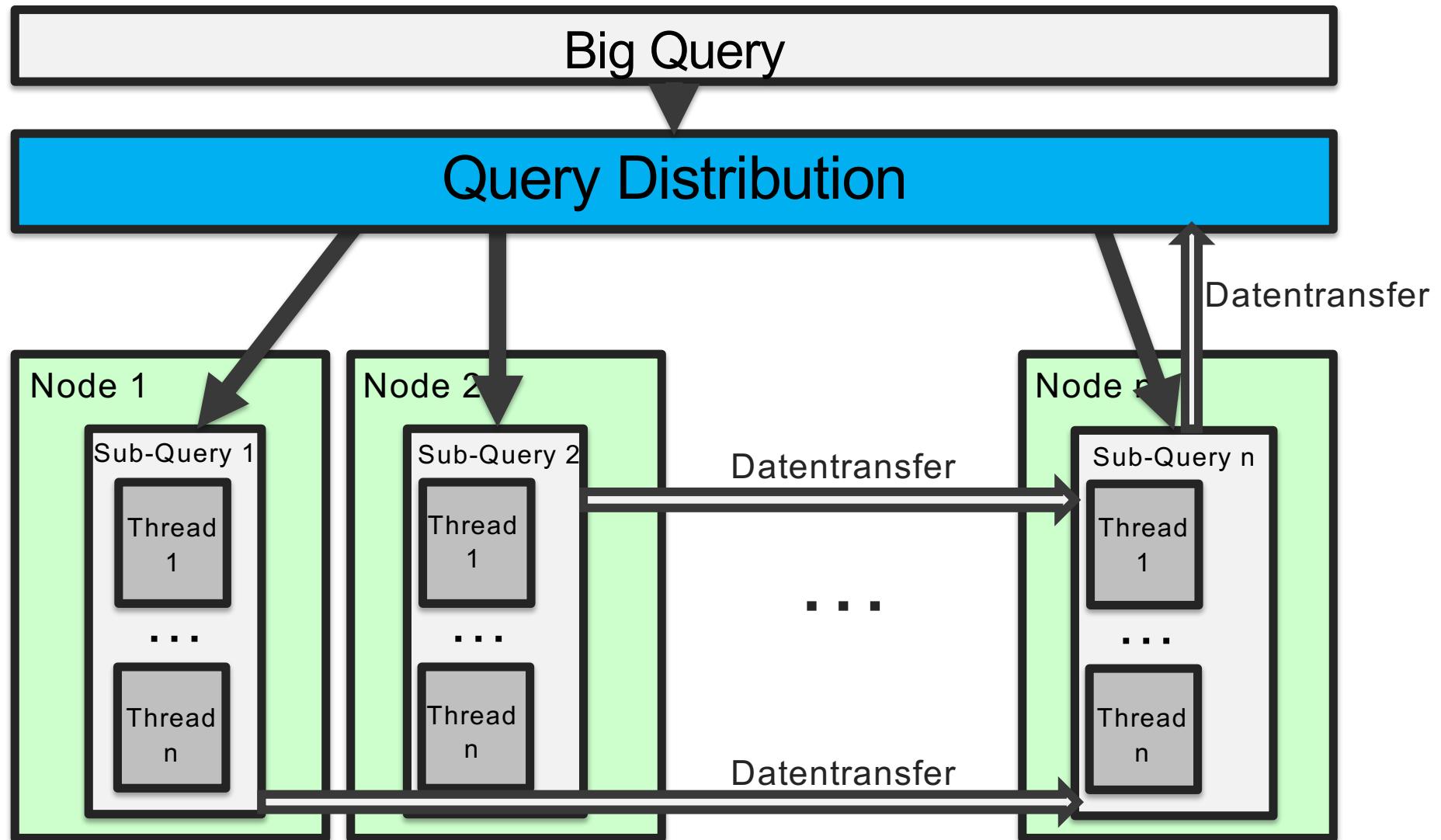


**The fastest I/O is the one that never takes place:** Es werden nur diejenigen Spalten gelesen, die benötigt werden (gerade bei breiten Tabellen wichtig)

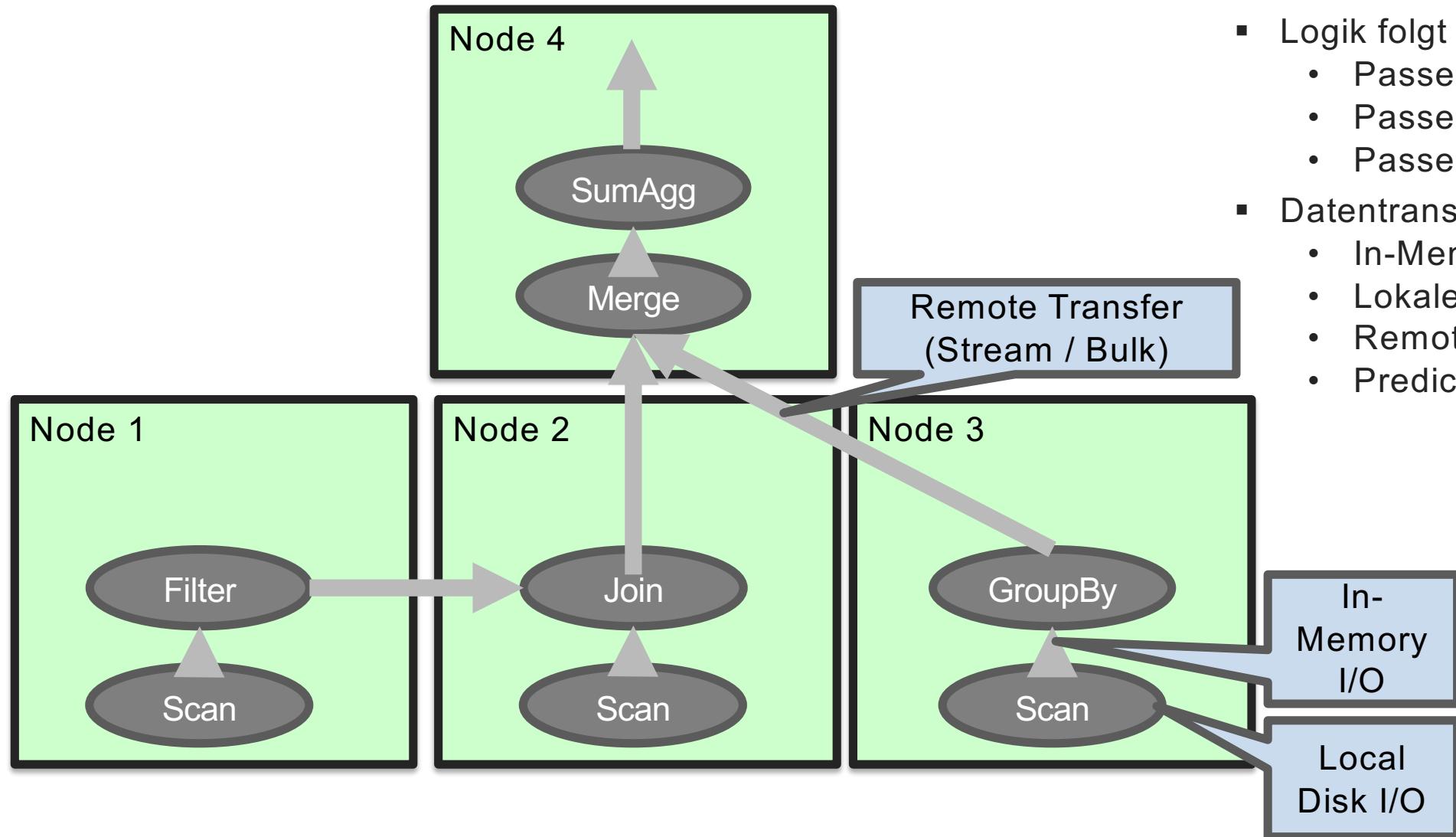
**Kompression** (funktioniert bei Spalten besser als bei Zeilen):

- Datentyp-spezifisch (z.B. Dictionaries)
- Allgemein (z.B. Snappy)
- + ggF. Spalten-Index

# Verteilte und parallelisierte Ausführung von Abfragen.



# Ein verteilter Ausführungsplan: Ein azyklischer Funktionsgraph.



- Logik folgt den Daten:
  - Passende Sharding-Funktion
  - Passende Partitioning-Funktion
  - Passende Replikation
- Datentransfer-Optimierung:
  - In-Memory vor ...
  - Lokaler Disk I/O vor ...
  - Remote-Transfer.
  - Predicate Pushdown.

# Mögliche Klausurfragen

- Ab welcher Datenmenge spricht man in der Regel von Big Data?
- Welche Vorteile bietet die Parallelität im Großen im Vergleich zur Parallelität im Kleinen?

# Die Klausur



# Die Klausur.

- Prüfungszeit: 12.07.2018 ab 14:30 (bitte vorher noch einmal online prüfen!)
- Schriftliche Prüfung über 90min. Daumenregel: Bewertungspunkt einer Frage = 1 Minute Bearbeitungszeit.
- Es sind keine Hilfsmittel zugelassen.
- Bitte rechtzeitig erscheinen! Die Prüfung startet pünktlich.
- Bitte Studentenausweis und Personalausweis mitbringen.
- Viel Erfolg!