

# Zusammenfassung



*vorlesung*

**CLOUD  
COMPUTING**

# Organisatorisches

## **Zulassungsvoraussetzung**

- Das Ergebnis Ihrer ZV sollte Ihnen über Moodle mitgeteilt worden sein.
- Falls Sie kein Feedback erhalten haben, melden Sie sich nach der Vorlesung bei mir.

## **Prüfung**

- Die Prüfung findet am **Dienstag, den 19.07.2022** um **10.30** in Raum **R1.006** statt.
- Die Teilnahme an der Prüfung ist ausschließlich **in Präsenz** möglich.
- Hilfsmittel sind **nicht zugelassen**.



BEGINNT JETZT DER  
ERNST DES LEBENS?



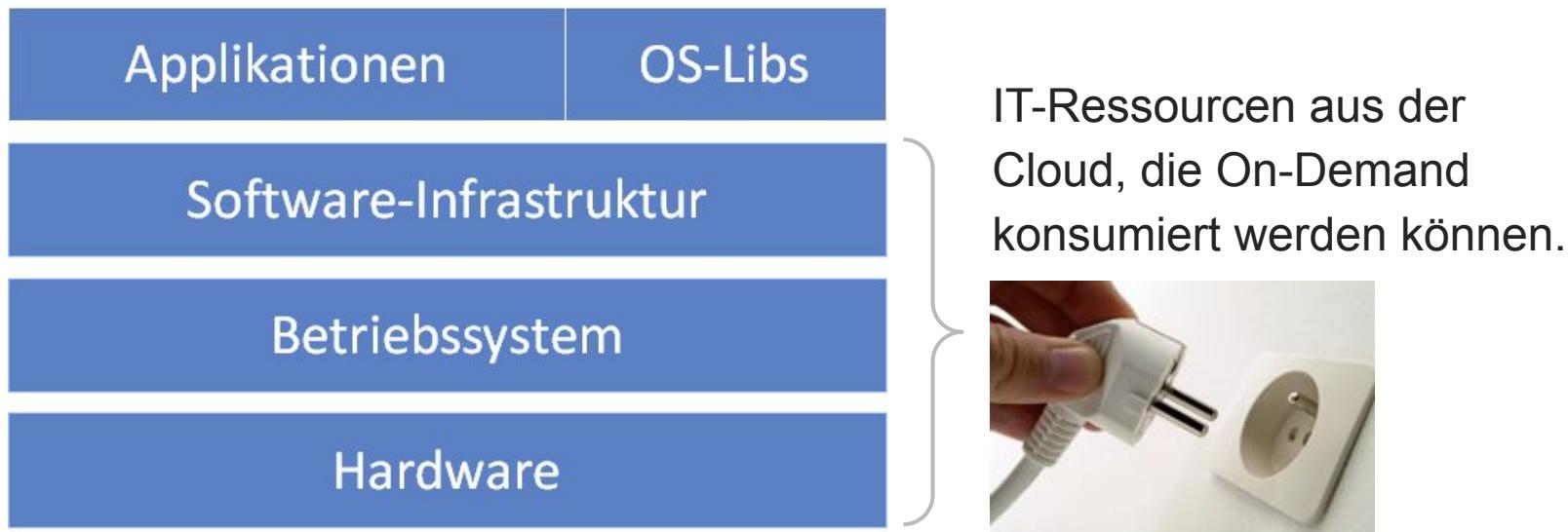
NEIN, DIE FREUDE  
AM CODEN.

BESTE EINSTIEGSCHANCEN FÜR STUDIERENDE BEI QAware.

Erfahre mehr über deine Perspektiven: [qaware.de/entwicklung](http://qaware.de/entwicklung)

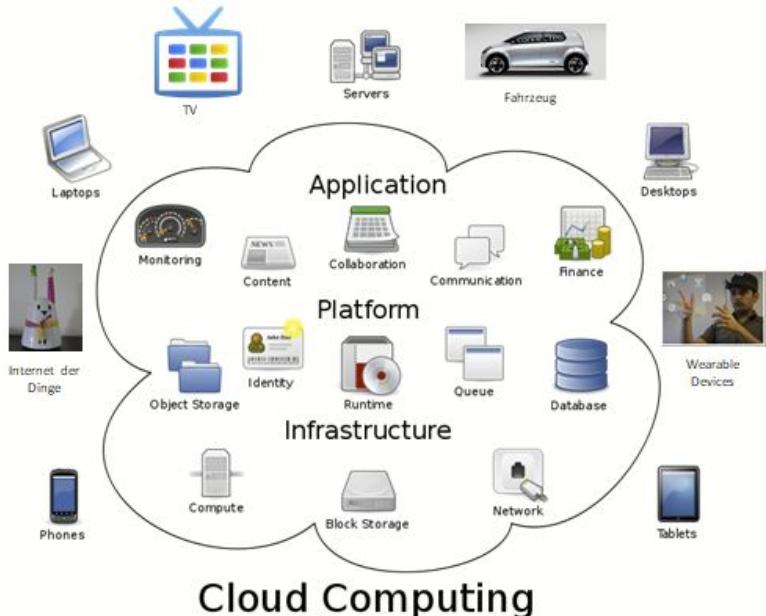
# Kapitel Einführung und Grundlagen

Beim Cloud Computing geht es im Kern 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 verschiedensten Endgeräten aus (über Standard-Protokolle).

# Die 5 Gebote der Cloud

- Everything fails all the time
- Focus on MTTR, not on MTTF
- Respect the eight fallacies of distributed computing
- Scale out, not up
- Treat resources as cattles, not as pets



Quelle: [https://de.wikipedia.org/wiki/Zehn\\_Gebote](https://de.wikipedia.org/wiki/Zehn_Gebote)

# Eight fallacies of distributed computing

DeveloperToArchitect.com

## Software Architecture Monday with Mark Richards Lesson 18 - Fallacies of Distributed Computing



**Mark Richards**

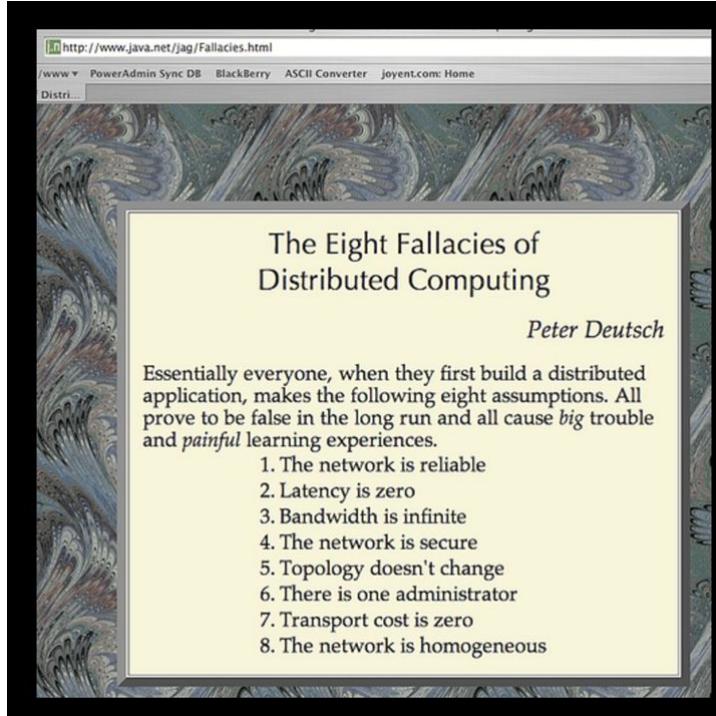
Independent Consultant

Hands-on Software Architect / Published Author / Conference Speaker

Founder, DeveloperToArchitect.com

[www.wmrichards.com](http://www.wmrichards.com)

# Eight fallacies of distributed computing



# 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



The image shows a historical newspaper clipping from the New York Times. At the top, there is a purple circular logo containing a white letter 'Y' followed by an exclamation mark, with the text 'NY Times' to its right. Below this, there is a list of bullet points describing a project. To the right of the list, there is a small graphic of a computer monitor with a cursor arrow. Further down, there is a section titled 'A COMPUTER WANTED' with descriptive text about a civil service examination for a computer position.

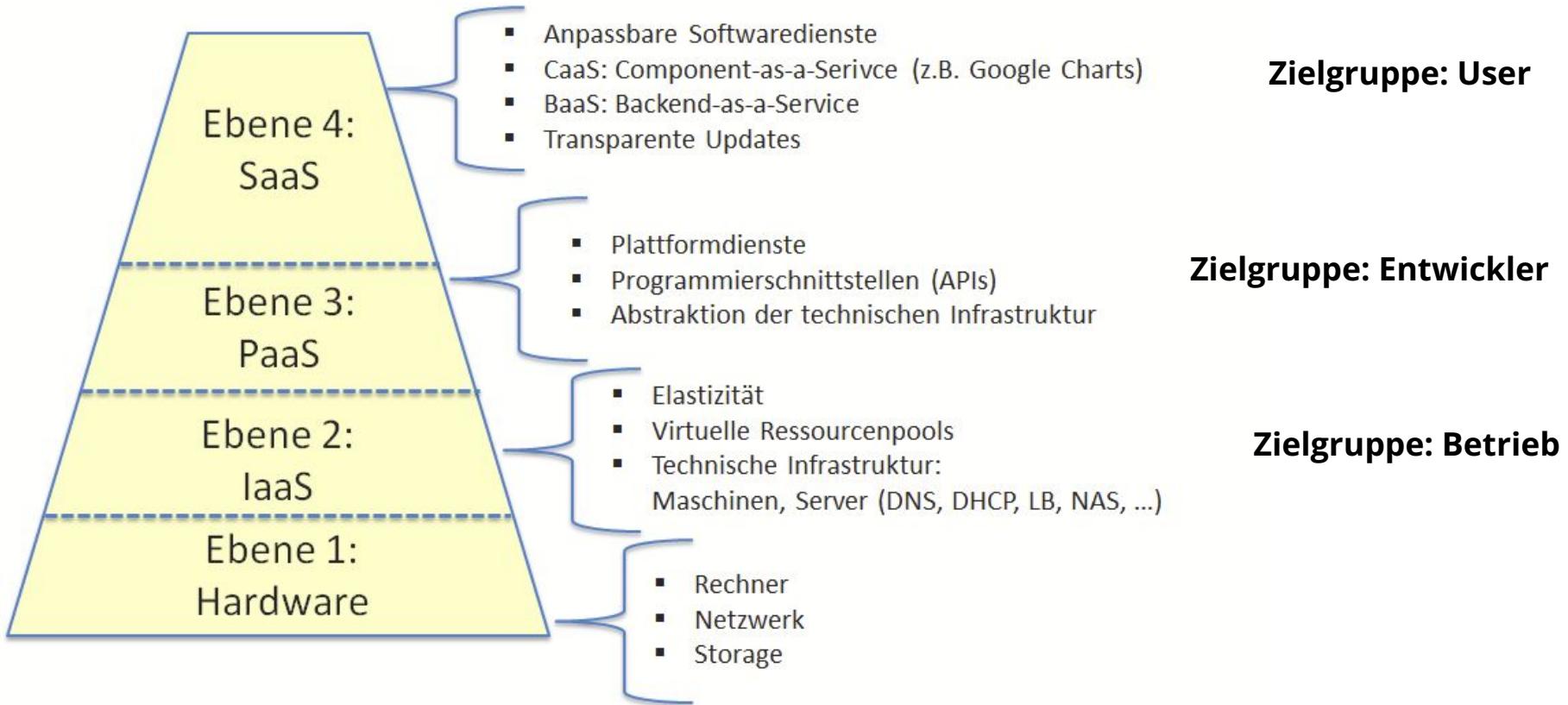
- Needed offline conversion of public domain articles from 1851-1922.
- Used Hadoop to convert scanned images to PDF
- Ran 100 Amazon EC2 instances for around 24 hours
- 4 TB of input
- 1.5 TB of output

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

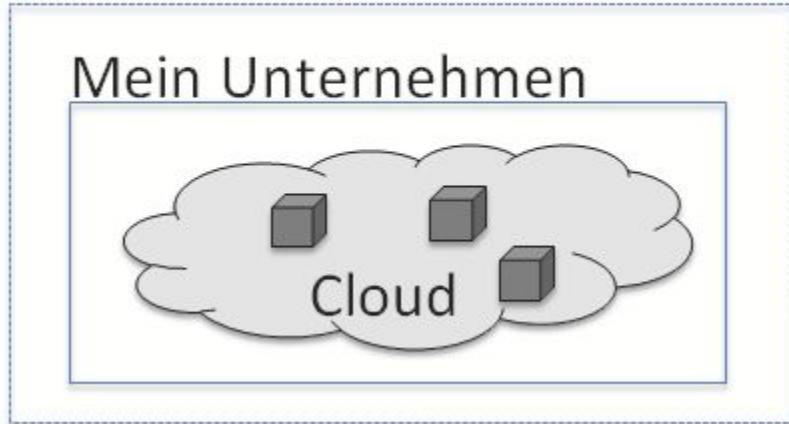
<http://www.slideshare.net/acarlos1000/hadoop-basics-presentation>

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

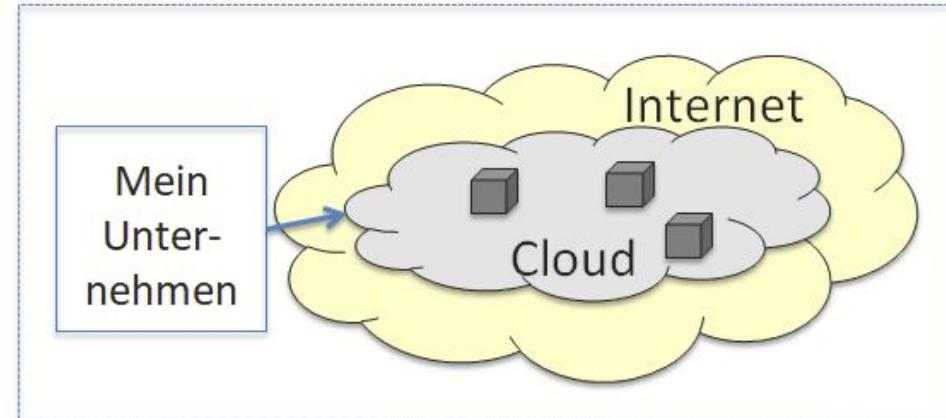


# Öffentliche und private Wolken.

## Private Cloud:

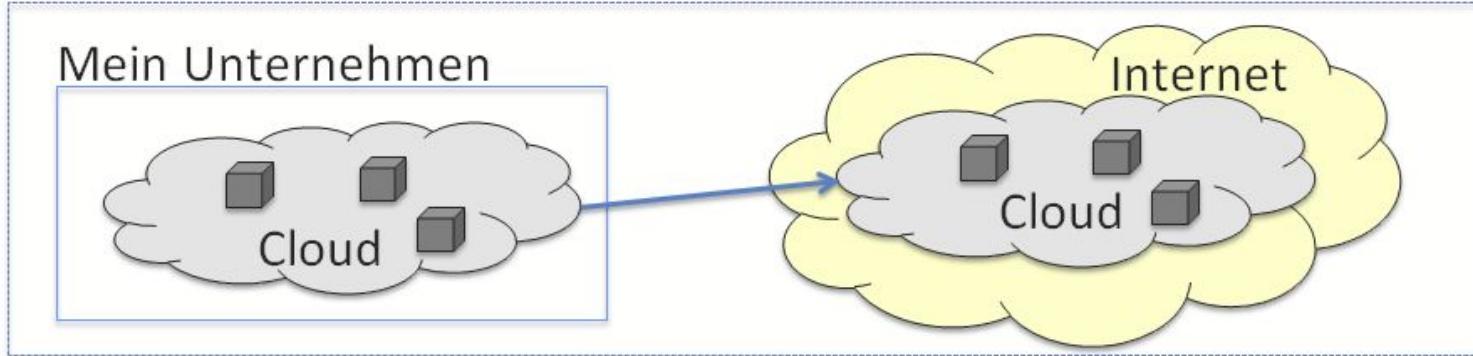


## Public Cloud:

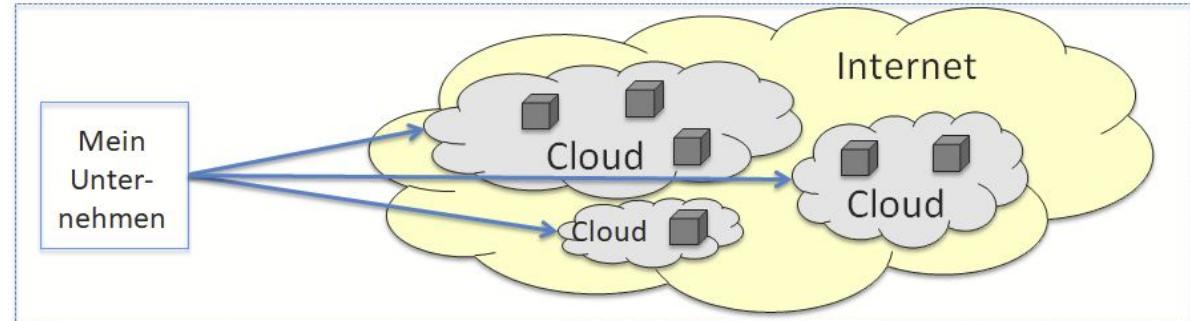


# Hybride und multiple Wolken.

## Hybrid Cloud:



## Multi-Cloud:



# Kapitel Kommunikation

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

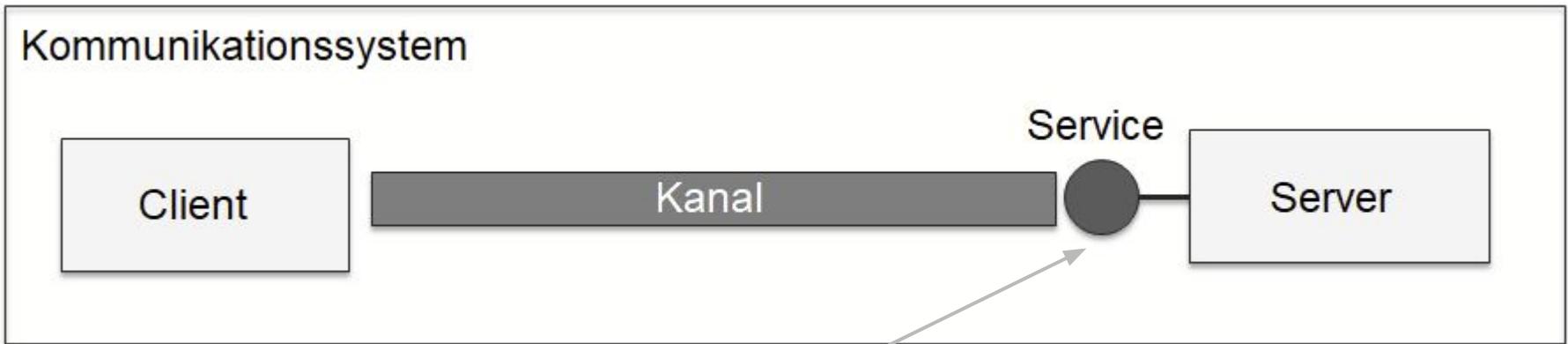
Kommunikationssystem = Infrastruktur für die Übermittlung von Informationen.



Typische Kanaleigenschaften:

- Richtung
- Datenformat/Codierung
- Synchronität (synchron/asynchron)
- Zuverlässigkeit und Garantien
- Sicherheit
- Performance (Latenz, Bandbreite)
- Overhead (Nutzlast / Gesamtlast)

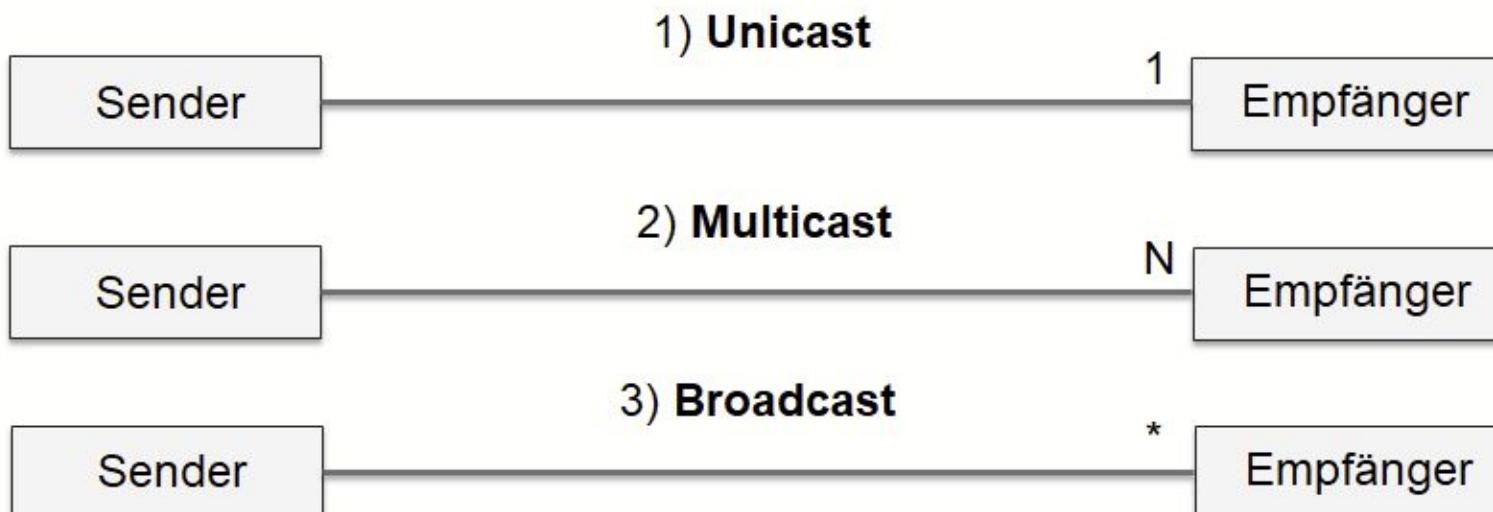
# Service-Orientierung in einem Kommunikationssystem: Client-Server-Kommunikation über Services



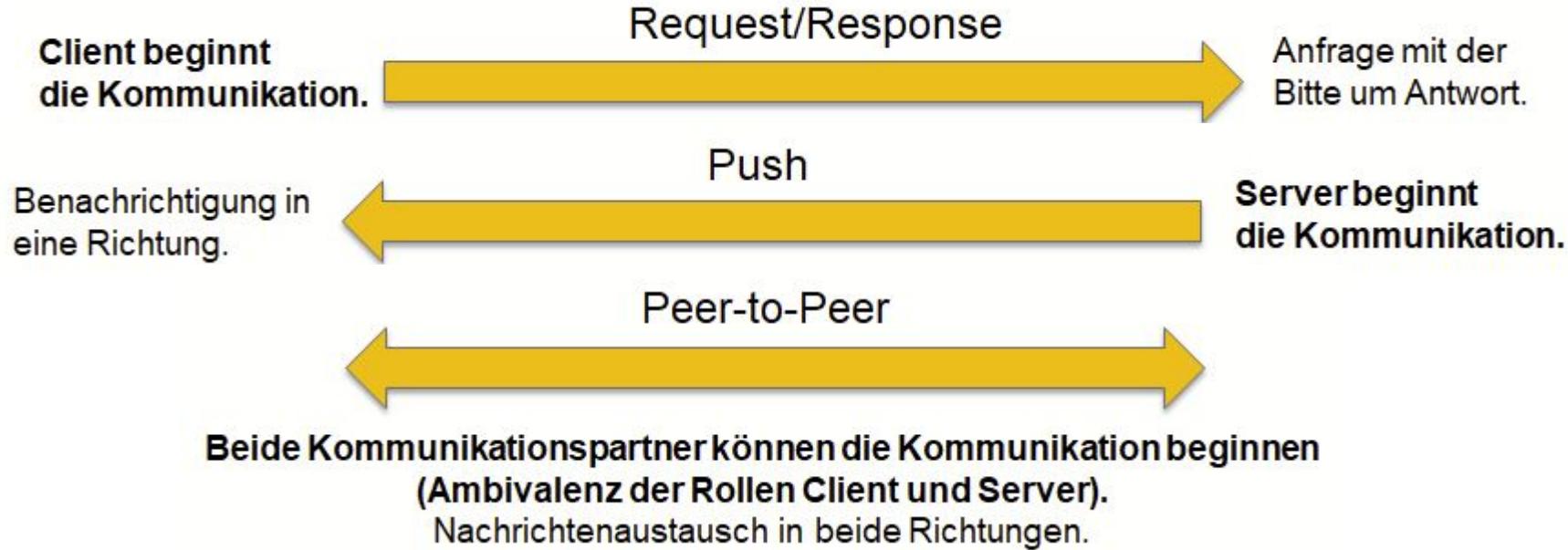
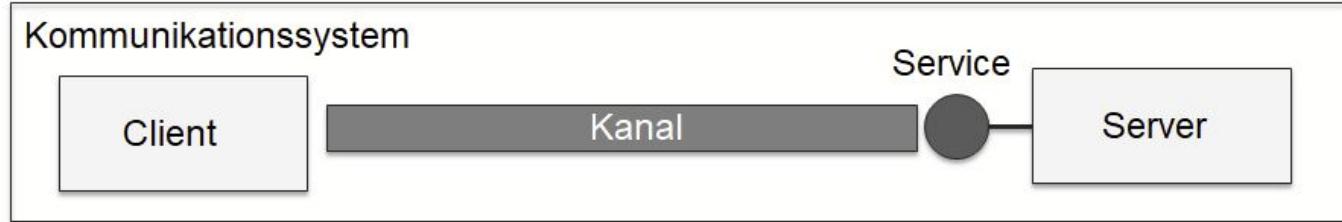
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

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**.

# Kardinalität der Empfänger einer Nachricht.



# Wer beginnt mit der Kommunikation?



# JSON

```
{  
  "Herausgeber": "Xema",  
  "Nummer": "1234-5678-9012-3456",  
  "Deckung": 2e+6,  
  "Währung": "EURO",  
  "Inhaber": {  
    "Name": "Mustermann",  
    "Vorname": "Max",  
    "männlich": true,  
    "Hobbys": [ "Reiten", "Golfen", "Lesen" ],  
    "Alter": 42,  
    "Kinder": [],  
    "Partner": null  
  }  
}
```

- JSON = JavaScript Object Notation (Daten pur).
- Auch in Binärcodierung (BSON – Binary JSON).
- MIME-Typ: *application/json*
- Schema-Sprachen: JSON Schema (<http://json-schema.org>)

Datentypen:

- Nullwert: null
- bool'scher Wert: true, false
- Zahl: 42, 2e+6
- Zeichenkette: "Mustermann"
- Array: [1,2,3]
- Objekt mit Eigenschaften:  
 {"Name": "Mustermann"}

# Protocol Buffers

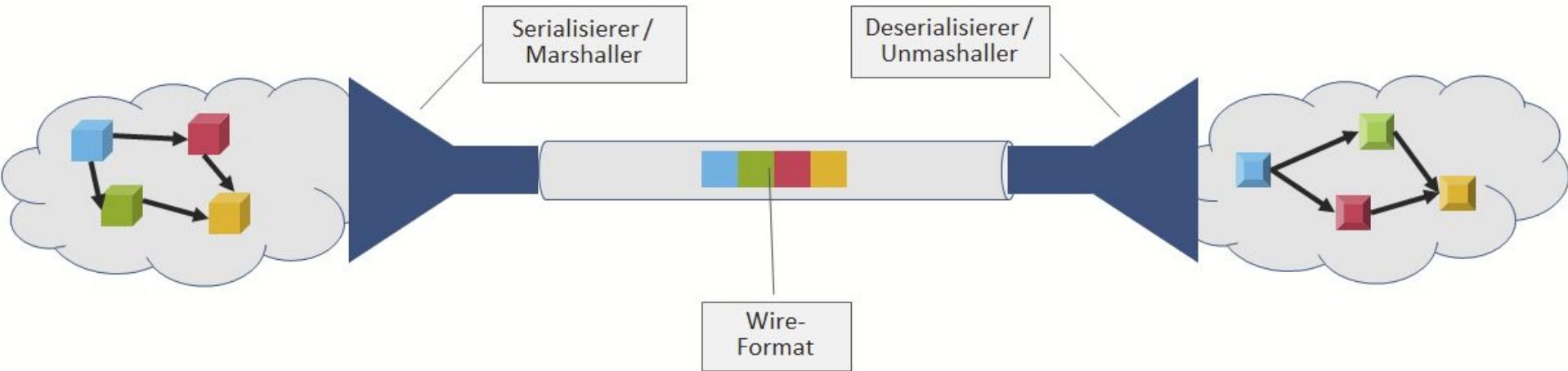
```
syntax = "proto3";

message SearchRequest {
    string query = 1;
    int32 page_number = 2;
    int32 result_per_page = 3;
}
```

- 2008 von Google veröffentlicht
- Sprachneutral
- Platformunabhängig
- Binärkodiert - speichereffizient und schnell
- Protobuf-Source wird von Protobuf-Compiler in Programmiersprache übersetzt

# Für die meisten Schnittstellen (JSON, ProtoBuf) benötigen wir Serialisierung und Deserialisierung.

Die **Serialisierung** ist [...] eine Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.



**Marshalling** (englisch marshal ‚aufstellen‘, ‚ordnen‘) ist das Umwandeln von strukturierten oder elementaren Daten in ein Format, das die Übermittlung an andere Prozesse ermöglicht

# 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 (Verben).
- 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 (POST = erzeugen, PUT = 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.

# Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch



## Entkopplung von Producer und Consumer

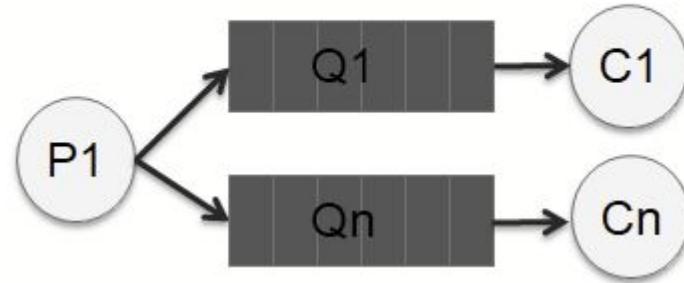
- Serviceschnittstelle: Format
- Messagebroker: keine Einschränkung für Format
- Sende- und Empfangszeitpunkt können beliebig lange auseinander liegen
- Horizontale Skalierbarkeit: Kann an mehrere Consumer ausgeliefert werden
- Lastverteilung: z.B. Auslieferung an Consumer, der am wenigsten zu tun hat
- Lastspitzen: Auslieferung der Nachricht kann verzögert werden, wenn alle Consumer überlastet sind
- Konfigurationsoptionen:
  - TTL der Nachricht
  - Zustellgarantie (min. 1 mal, exakt 1 mal, keine Garantie)
  - Transaktionalität
  - Priorität der Nachricht
  - Einhaltung der Reihenfolge

# Messaging erlaubt mehrere Kommunikationsmuster

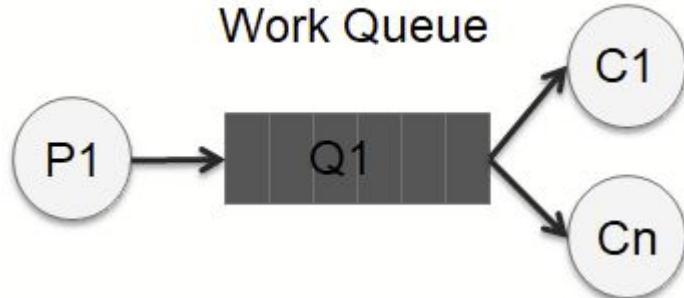
Message Passing



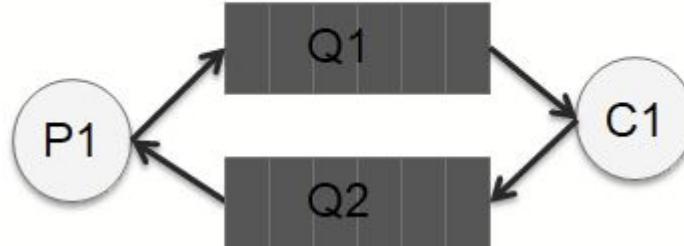
Publish/Subscribe



Work Queue



Remote Procedure Call



# Kapitel 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*)

Das schafft Grundlagen für das Cloud Computing

- Entkopplung von der Hardware für mehr Flexibilität im Betrieb und Robustheit bei Ausfällen
- Normierung von Ressourcen-Kapazitäten auf heterogener und wechselnder Hardware
- Zentrale Steuerung und Bereitstellung von Rechenressourcen über die mit Virtualisierung bereitgestellten Software-Defined-Resources

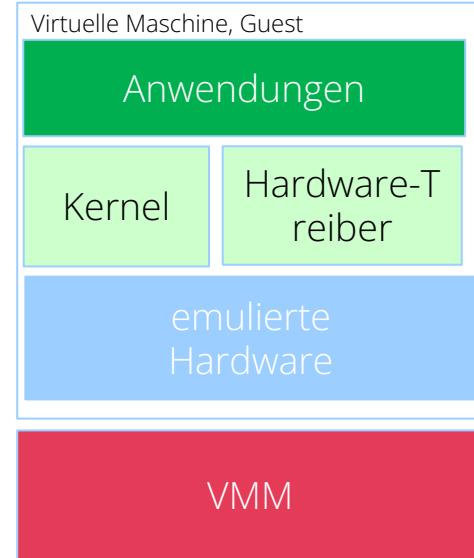
# Was wird virtualisiert?

Hardwarevirtualisierung arbeitet auf Ebene der Rechnerarchitektur.

- **Prozessor**
  - Der State des Prozessors. Im wesentlichen Prozessorregister.
  - Maschinencode
  - Memory Management Unit
- **Hauptspeicher**
  - Linear addressierter physikalischer Speicher
- **Netzwerk**
  - z. B. Input-Output Stream von Ethernet Frames
- **Storage**
  - Blockspeicher (linear addressiert. Lesen und Speichern von Blöcken)
- **Grafikkarte**
  - z. B. Framebuffer (2D-Array mit Pixeldaten)
  - 3D Funktionalität (DirectX, OpenGL), siehe [https://en.wikipedia.org/wiki/GPU\\_virtualization](https://en.wikipedia.org/wiki/GPU_virtualization)
  - Computing (KI, Simulationen) (Zunehmend wichtig für die Cloud)
- Evtl. Peripherie wie USB, Maus, Keyboard
- Timer, Interrupt Controller

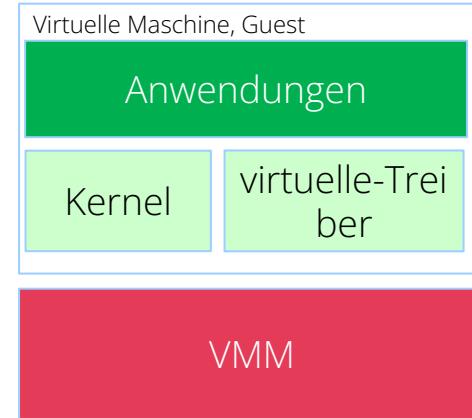
# Voll-Virtualisierung

- Jedem Gastbetriebssystem steht ein eigener virtueller Rechner mit virtuellen Ressourcen wie CPU, Hauptspeicher, Laufwerken, Netzwerkkarten, usw. zur Verfügung.
- Das Gastbetriebssystem muss also nicht angepasst werden. Zum Zeitpunkt des Starts muss das Gastbetriebssystem nicht bekannt sein.
- Die VMM emuliert auch weiterhin echte Hardware wie Storage (SATA) und Netzwerk (Ethernet).
- Die VMM kann aber zur Beschleunigung oder zur besseren Nutzung (Grafik, Mouse) spezielle virtuelle Hardware zur Verfügung stellen.
  - z. B. Einfacher Pass-Through von USB
  - Fließender Übergang zur Paravirtualisierung
  - Leistungsverlust: 1 - 5%

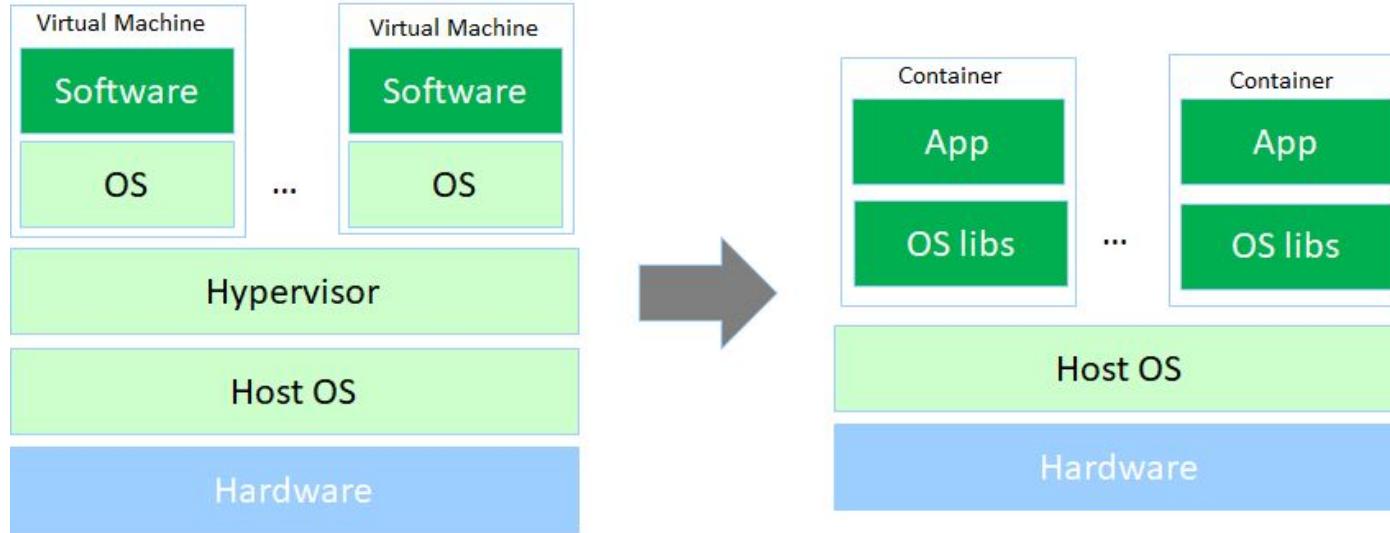


# Para-Virtualisierung

- Dem Gast-Betriebssystem stehen keine direkt low-level virtualisierten Hardware-Ressourcen zur Verfügung sondern eine API.
  - Vereinfacht den Aufbau der VM
- Das Gast-Betriebssystem muss portiert werden.
  - Low Level Prozessorinstruktionen werden erst gar nicht ausgeführt oder durch API Aufrufe abgebildet
  - Virtuelle Treiber (z. B. virtio).
    - Vermeidung von Umformungen und Kopieraktionen durch Verwendung spezieller Treiber
    - Übertragung von IP Paketen und nicht von Ethernet Frames.
- Unterstützte Betriebssysteme und Hardware-Varianten aus Sicht des Gastes eingeschränkt pro Hypervisor-Implementierung.
- Leistungsverlust: 0 - 2%



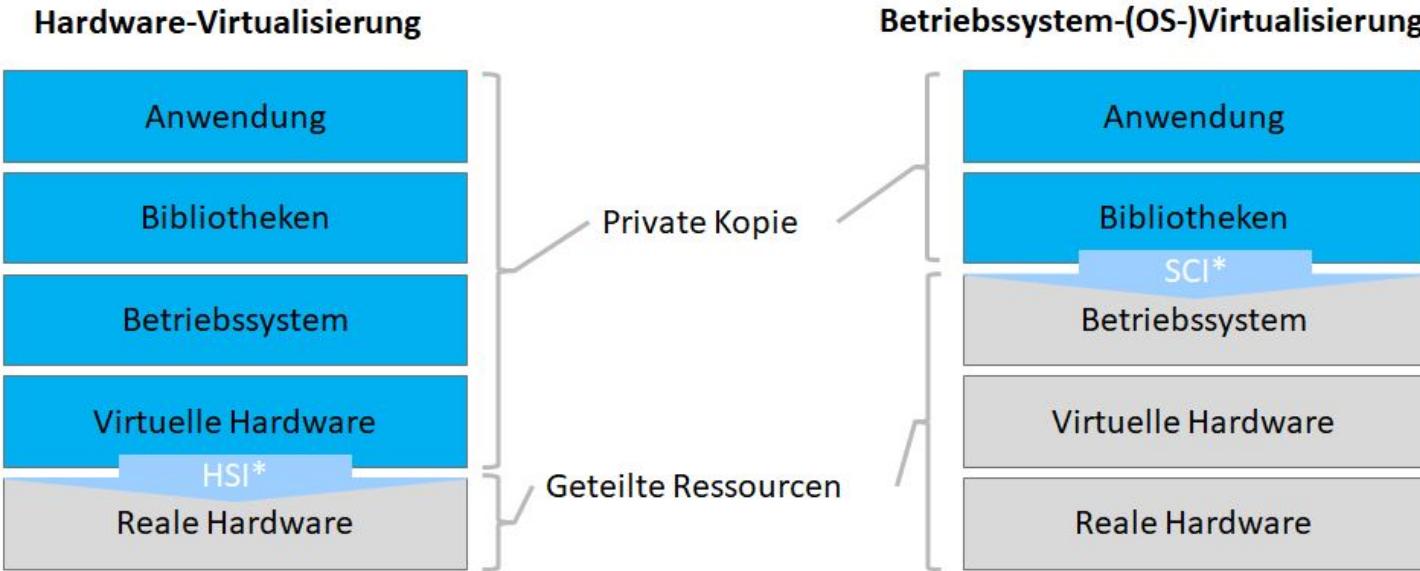
# Betriebssystem-Virtualisierung / Containerisierung



- 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

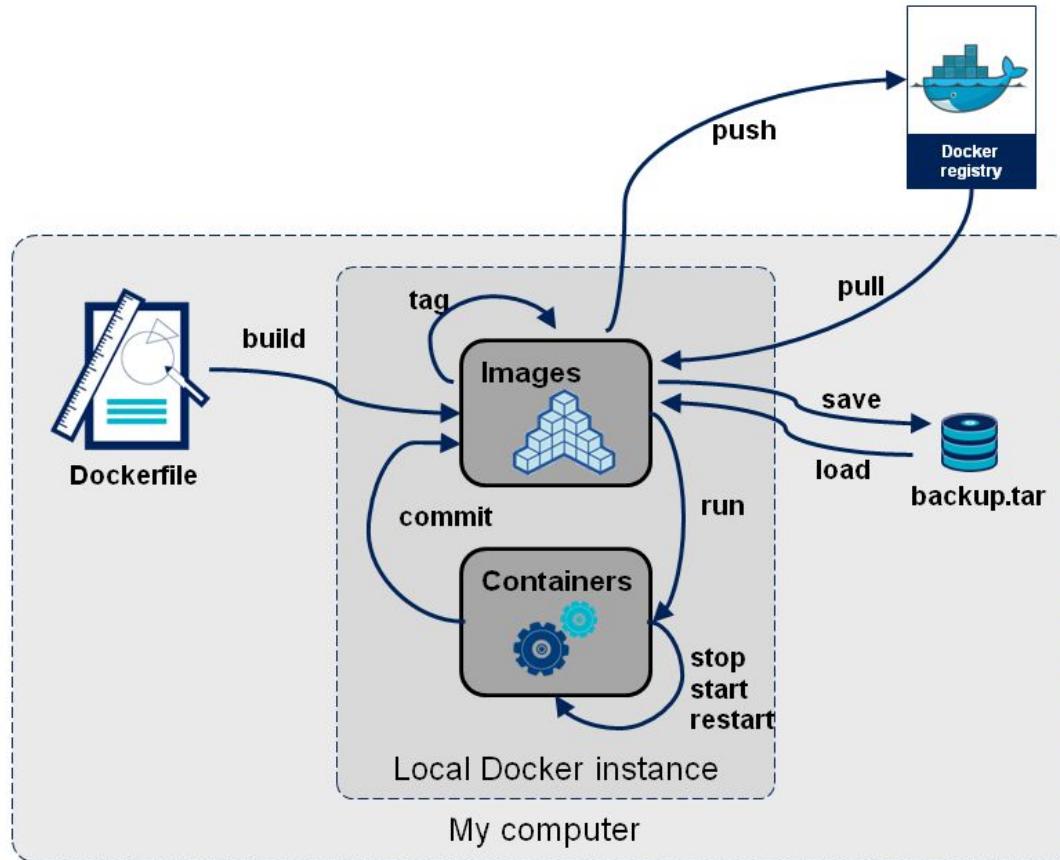
\* HSI = Hardware Software Interface  
SCI = System Call Interface



- Benötigt Hardwareunterstützung
- Höhere Sicherheit. Die HSIs sind einfach.
- Stärkere Isolation.
- Hohes Volumen, Hohe Startzeit
- Unterschiedliche Betriebssysteme

- Ist eine reine Softwarelösung
- Geringere Sicherheit: System Call Interface ist sehr mächtig und komplex
- Geringeres Volumen, Geringerer Overhead, Kürzere Startup-Zeit
- Betriebssystem fest

# Der Docker Workflow



# Das Dockerfile definiert Aufbau und Inhalt des Image.

My Image

Layer 8

Layer 7

Layer 6

Layer 5

Layer 4

Layer 3

Layer 2

Layer 1

```
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"]
```

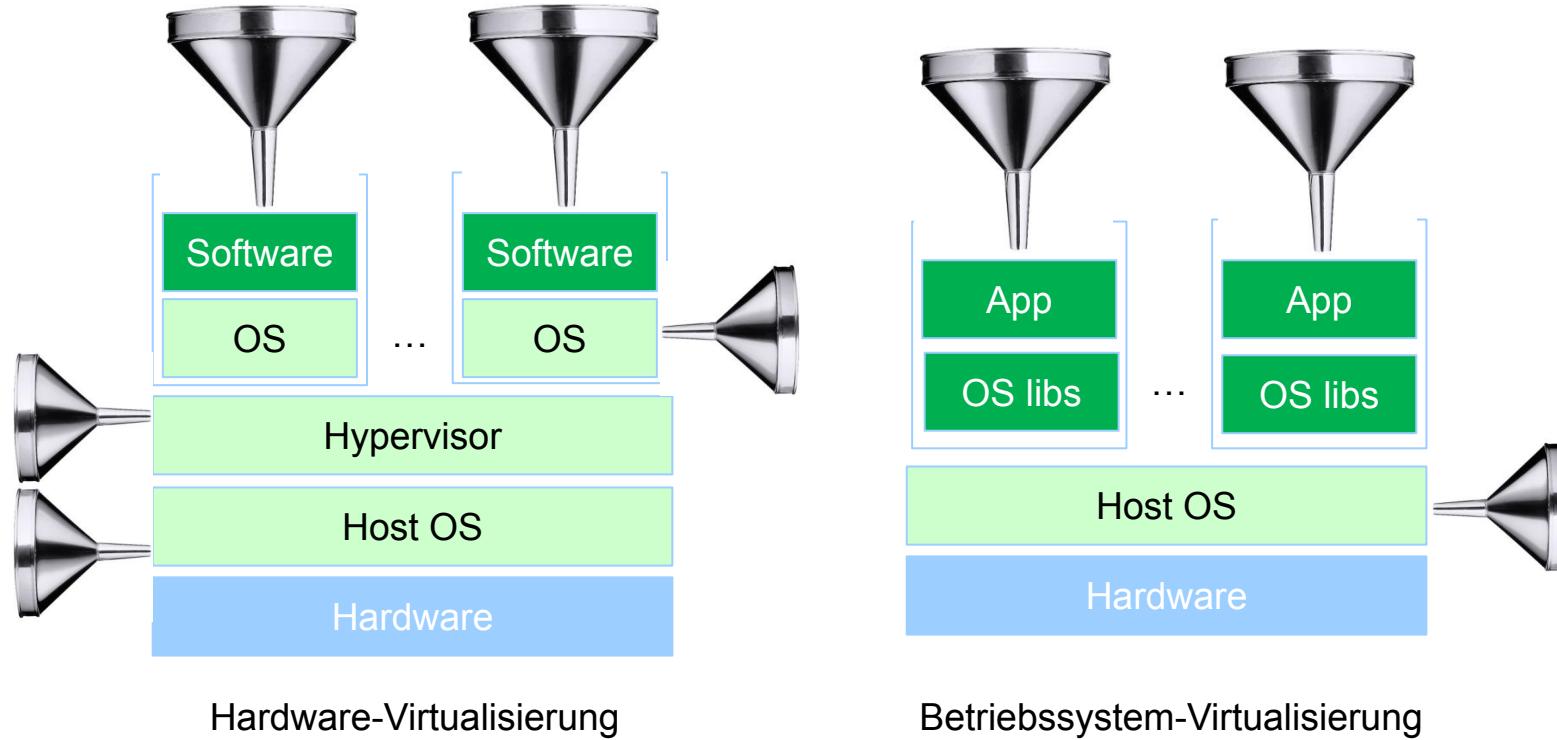
# Typische Kommandos eines Docker Workflows

Command	Action
<code>docker build -t &lt;image&gt; .</code>	Build Docker image from „Dockerfile“ with given tag in current directory
<code>docker images</code>	Prints all local images
<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)
<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

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

# Kapitel Provisionierung

# Provisionierung: Wie kommt Software in die Boxen?



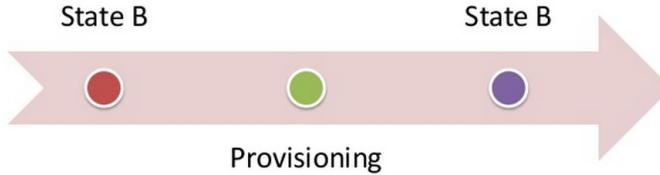
Provisionierung ist die Bezeichnung für die automatisierte Bereitstellung von IT-Ressourcen.

<http://wirtschaftslexikon.gabler.de/Definition/provisionierung.html>

# 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:

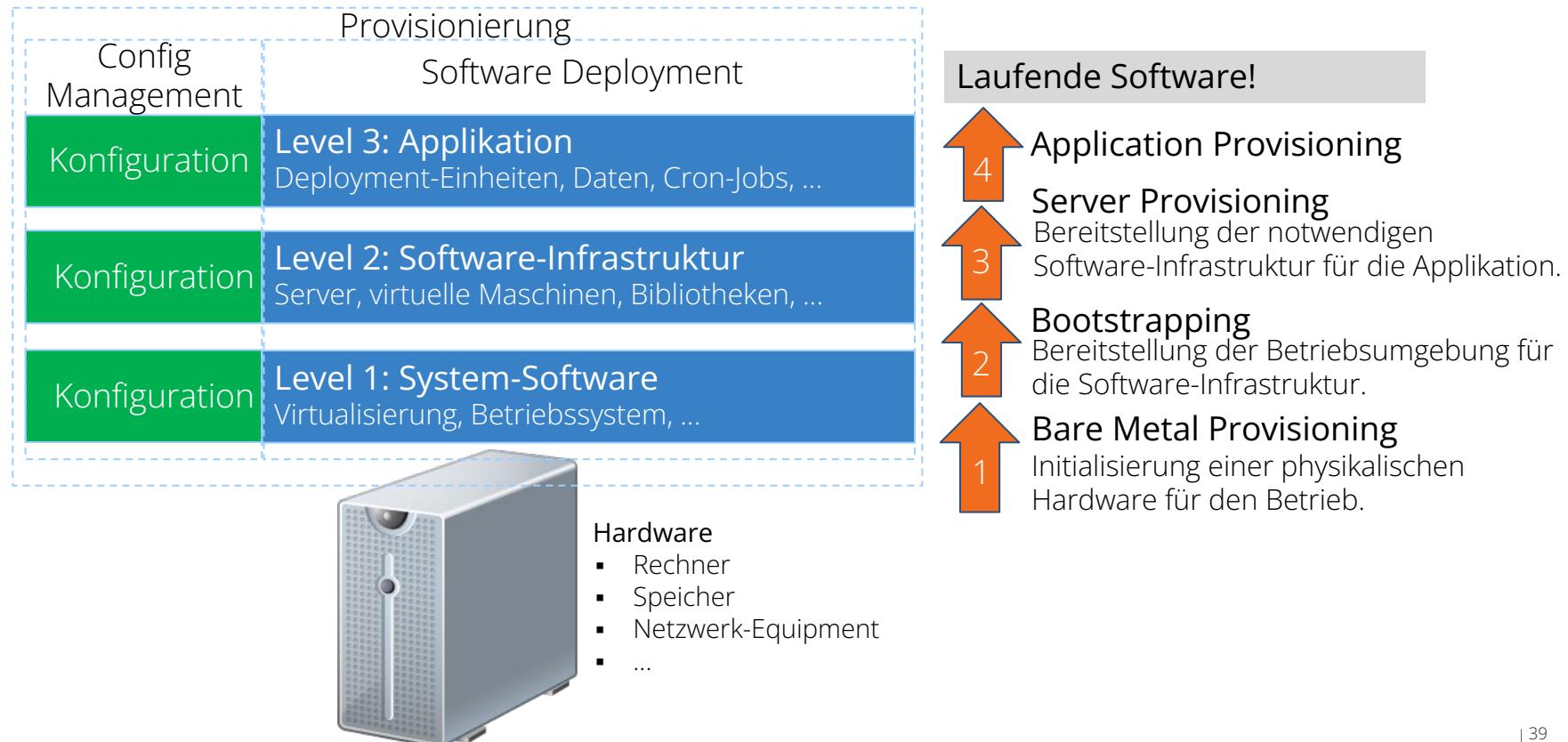
1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggf. Zustand zurücksetzen



**Idempotenz:** Die Fähigkeit eine Aktion durchzuführen und sie dasselbe 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.

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



# Die neue Leichtigkeit des Seins.

Old Style

Beliebiger  
Zustand



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

Ziel-Zustand

New Style

„Immutable Infrastructure / Phoenix Systems“

Basis-Zustand



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

Ziel-Zustand

# Immutable Infrastructure

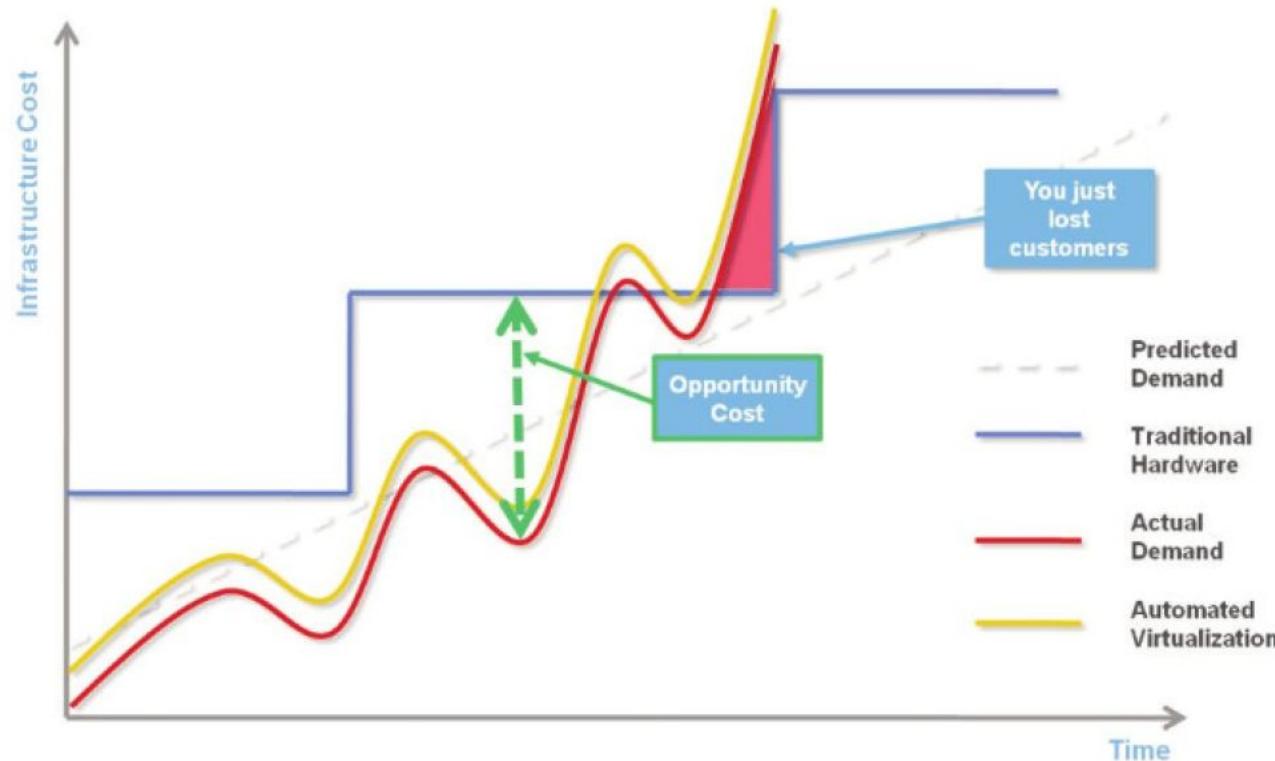
An *immutable infrastructure* is another infrastructure paradigm in which servers are **never modified** after they're deployed. If something needs to be updated, fixed, or modified in any way, **new servers built from a common image with the appropriate changes** are provisioned to replace the old ones. After they're validated, they're put into use and **the old ones are decommissioned**.

The benefits of an immutable infrastructure include **more consistency and reliability** in your infrastructure and a **simpler, more predictable deployment process**. It mitigates or entirely prevents issues that are common in mutable infrastructures, like **configuration drift** and **snowflake servers**. However, using it efficiently often includes comprehensive deployment automation, fast server provisioning in a cloud computing environment, and solutions for handling stateful or ephemeral data like logs.

Quelle: <https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure>

# Kapitel Infrastructure-as-a-Service

Klassische Betriebsszenarien werden bei dynamischer Nachfrage teuer. Hohe Opportunitätskosten.



# 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.

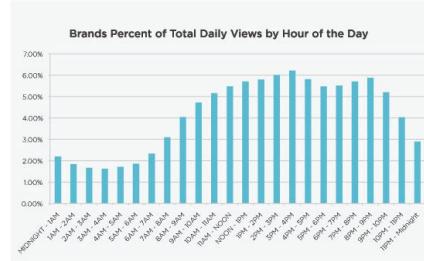
Infrastruktur-Dienste einer IaaS-Cloud:

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

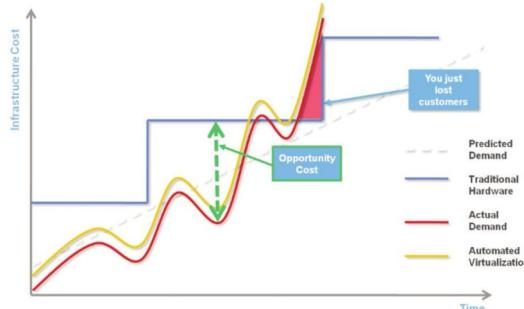
# Skalierbarkeit: Effekte

## ■ Tageszeitliche und saisonale Effekte:

Mittags-Peak, Prime-Time-Peak, Wochenend-Peak,  
Weihnachten, Valentinstag, Muttertag, ...  
(vorhersehbare Belastungsspitzen)

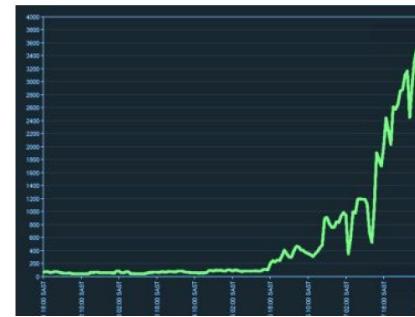


## ■ Kontinuierliches Wachstum

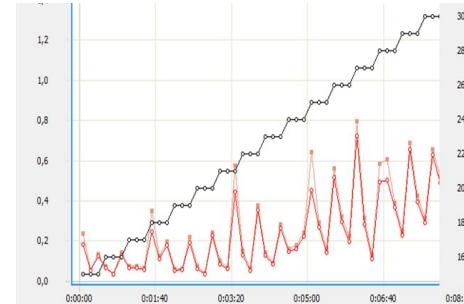


Source: Amazon Web Services

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



## ■ Temporäre Plattformen: Projekte, Tests, Batch...



# Kriterien bei der Auswahl einer passenden IaaS-Cloud

- Unterstützte Cloud-Varianten (Private Cloud, Public Cloud, Hybrid Cloud, ...)
- Zuverlässigkeit / Verfügbarkeit
- Sicherheit und Datenschutz
- Vorhersagbare und stabile Performance
- Preismodell: Fixe und flexible Kosten
- Skalierbarkeit: Grenzen, Automatismen und Reaktionszeiten
- Lock-In der Daten und Anwendungen: Offene APIs
- Haftung
- Support

# Service Level Agreement

## Service Level Objective

Zielwert einer messbaren Metrik die über die Qualität oder Verfügbarkeit eines Dienstes aussage trifft.

## Service Level Agreement

Vertrag über die Bereitstellung von Ressourcen und Dienste mit Zuverlässigkeitzzusagen (SLOs). Häufig verbunden mit Konsequenzen bei nichterfüllung der SLO wie z.B. finanziellen Strafen.

## Verfügbarkeitsklassen:

Availability %	Downtime per Year	Downtime per Month	Downtime per Week
99.9% (three nines)	8.76 hours	43.2 minutes	10.1 minutes
99.95%	4.38 hours	21.56 minutes	5.04 minutes
99.99% (four nines)	52.6 minutes	4.32 minutes	1.01 minutes
99.999% (five nines)	5.26 minutes	25.9 seconds	6.05 seconds
99.9999% (six nines)	31.5 seconds	2.59 seconds	.0605 seconds

## Beispiel: Amazon S3 (Storage)

### Service Commitment

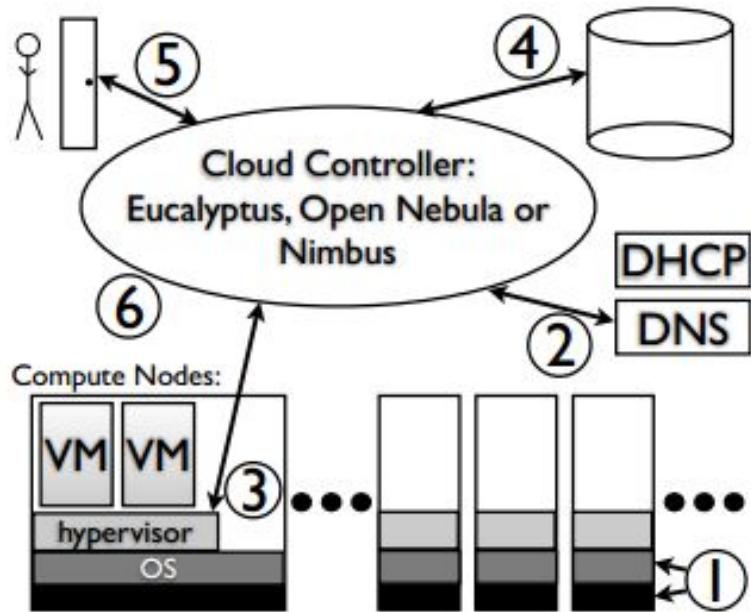
AWS will use commercially reasonable efforts to make Amazon S3 available with a Monthly Uptime Percentage (defined below) of at least 99.9% during any monthly billing cycle (the "Service Commitment"). In the event Amazon S3 does not meet the Service Commitment, you will be eligible to receive a Service Credit as described below.

Monthly Uptime Percentage	Service Credit Percentage
Equal to or greater than 99% but less than 99.9%	10%
less than 99%	25%

# Aspekte der Sicherheit in einer IaaS-Cloud.

- Vertraulichkeit der Daten und Datenkommunikation: Datenverschlüsselung, VPNs
- Nachvollziehbarkeit der Daten: Einhaltung nationaler Gesetze (z.B. EU-Datenschutzbestimmung, US Patriot Act) durch geographische Datenhaltung
- Firewalls und starke Authentifizierungsverfahren
- Backup der VMs, Storages und Datenbanken
- Zertifizierungen: ISO 27001, TÜV IT
- Siehe auch Sopot Memorandum:  
[https://www.datenschutz-berlin.de/fileadmin/user\\_upload/pdf/publikationen/working-paper/2012/2012-WP-Sopot\\_Memorandum-de.pdf](https://www.datenschutz-berlin.de/fileadmin/user_upload/pdf/publikationen/working-paper/2012/2012-WP-Sopot_Memorandum-de.pdf)

# 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.

# Infrastructure as Code

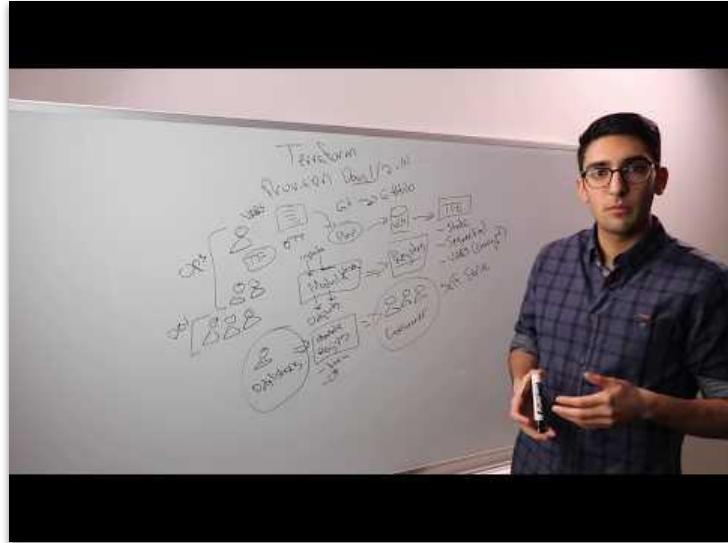
- Provisionieren und Managen ganzer Rechenzentren - nicht nur einzelne virtuelle Maschinen
- Abgrenzung zu Configuration Management (z.B. Ansible):
  - Explizite Erzeugung und Zerstörung der Infrastruktur eines (virtuellen) Rechenzentrums
  - Immutable Infrastructure, statt kontinuierlichem ändern existierender Ressourcen
  - Typischerweise Deklarativ statt Imperativ
- Erstmals für die Cloud in 2010 mit AWS CloudFormation

Vorteile:

- Versionierung des Rechenzentrums und damit einfaches Staging und Rollbacks
- Beschleunigte Auslieferung von Infrastrukturänderungen
- Konsistenz über Umgebungen hinweg
- Dadurch auch Sicherheit und Auditierbarkeit der Infrastruktur im Code
- Wiederverwendbar und Modularisierbar
- Ermöglicht Kollaboration über Code Verwaltung

# Terraform Grundlagen

- **Write:** Beschreibung Zielzustand über eine domänenspezifische Sprache HCL (HashiCorp Configuration Language)
- **Plan (terraform plan):** Ist-Zustand ermitteln. Notwendige Änderungen planen (entsprechend Abhängigkeiten geordnet und parallelisiert, Unterbrechungen möglichst minimal)
- **Apply (terraform apply):** Idempotente Herstellung des Zielzustands. Der Zustand (.tfstate Datei) wird meist in einem Remote Storage (S3, HTTP, ...) gespeichert



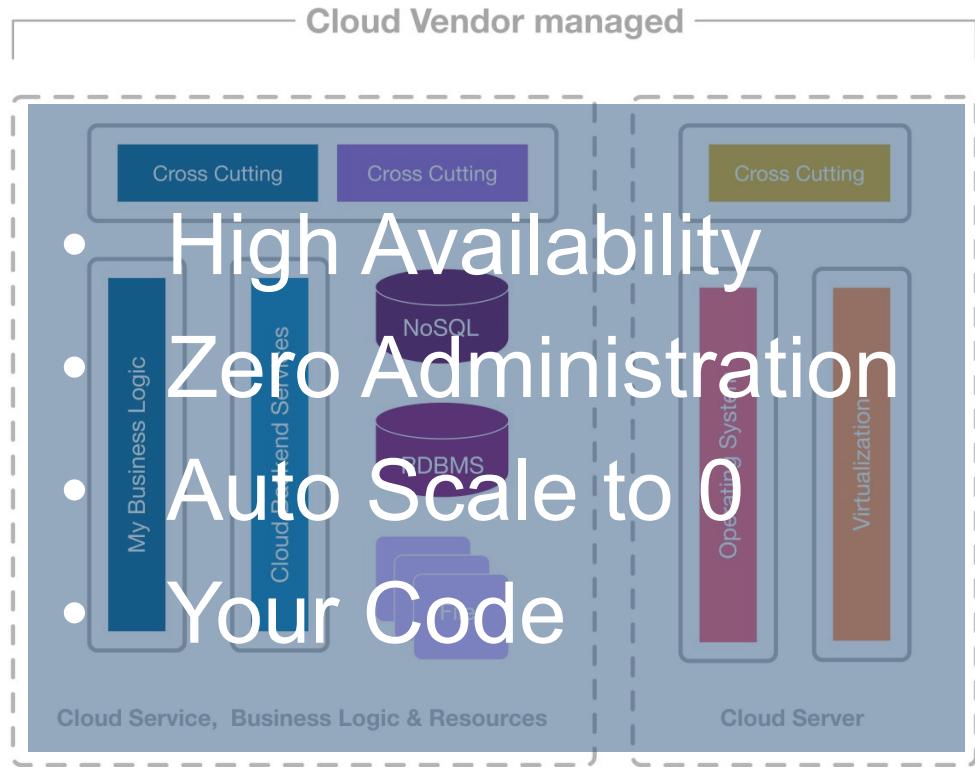
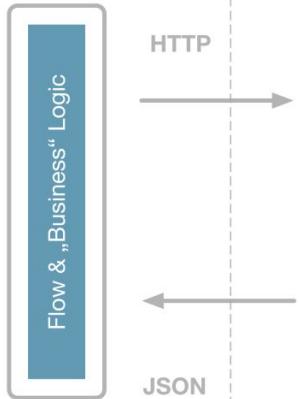
| 51

Video Pause bei 06m:30s, dann weiter bis 10m18s:  
<https://www.youtube.com/watch?v=h970ZBgKING>

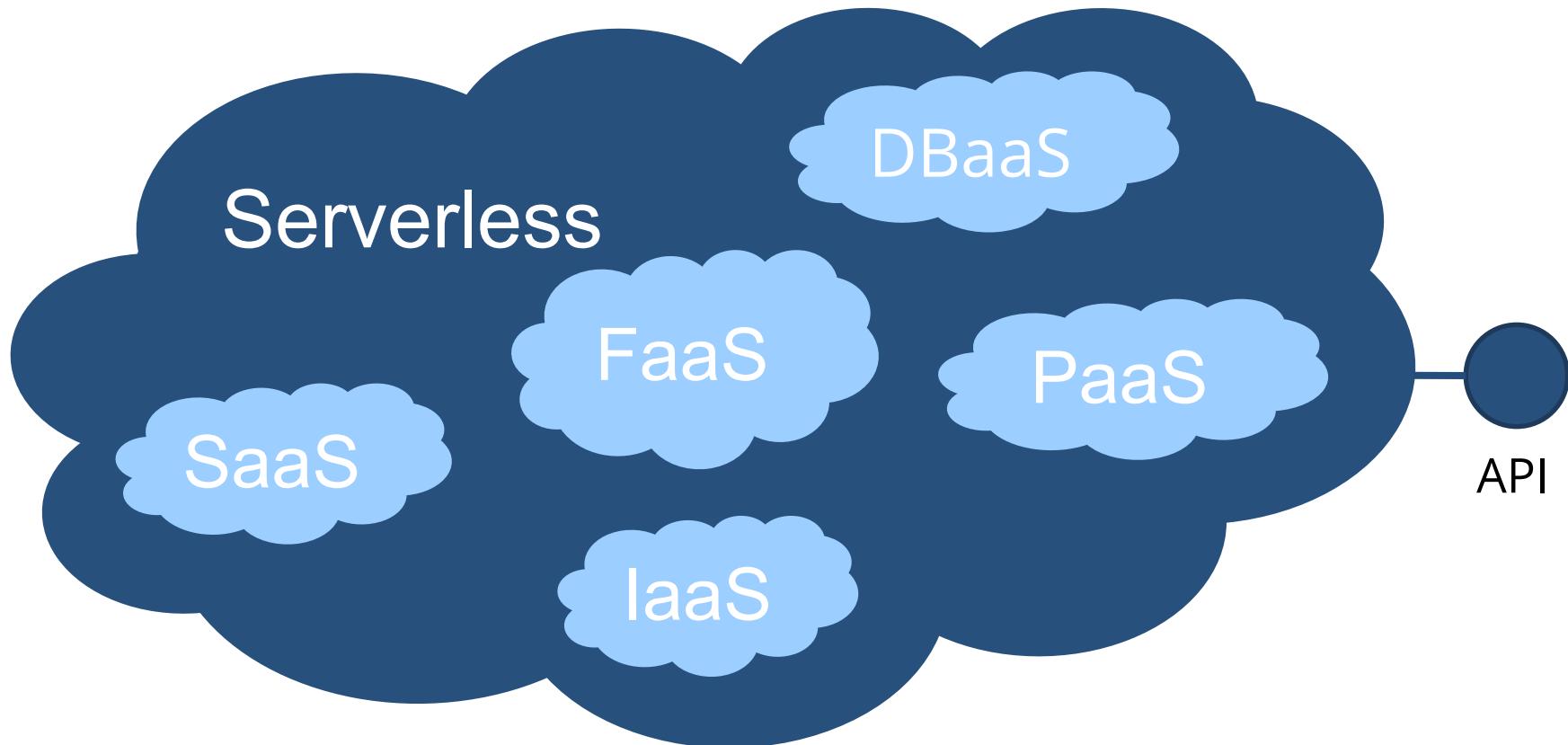
# Kapitel Serverless

# Serverless Anwendungsarchitektur

Run Code, not Servers!



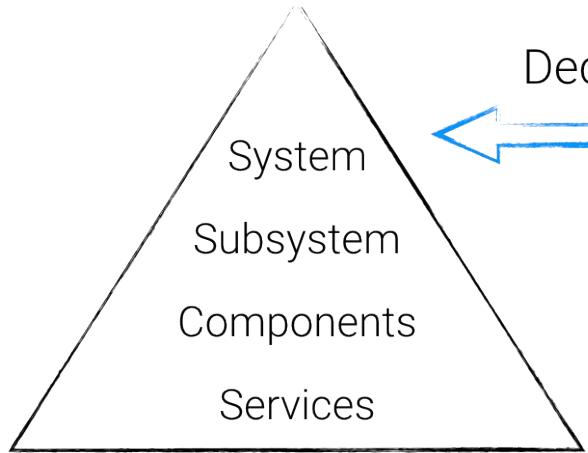
# Out-of the Box Self-scaling Fully Managed Backend



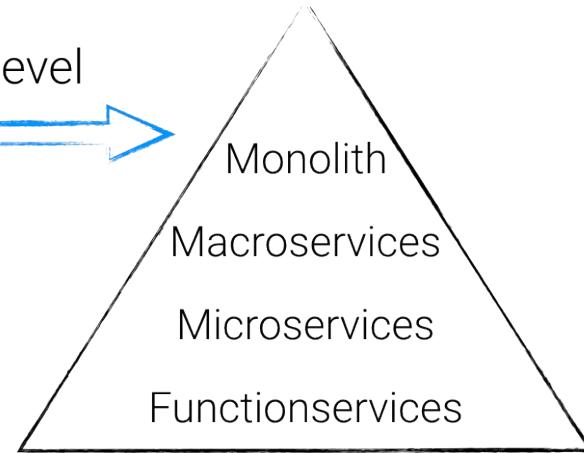
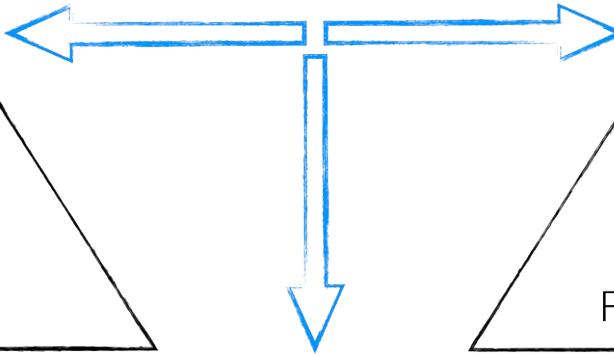
# Dev Components



# Ops Components



Decomposition Level

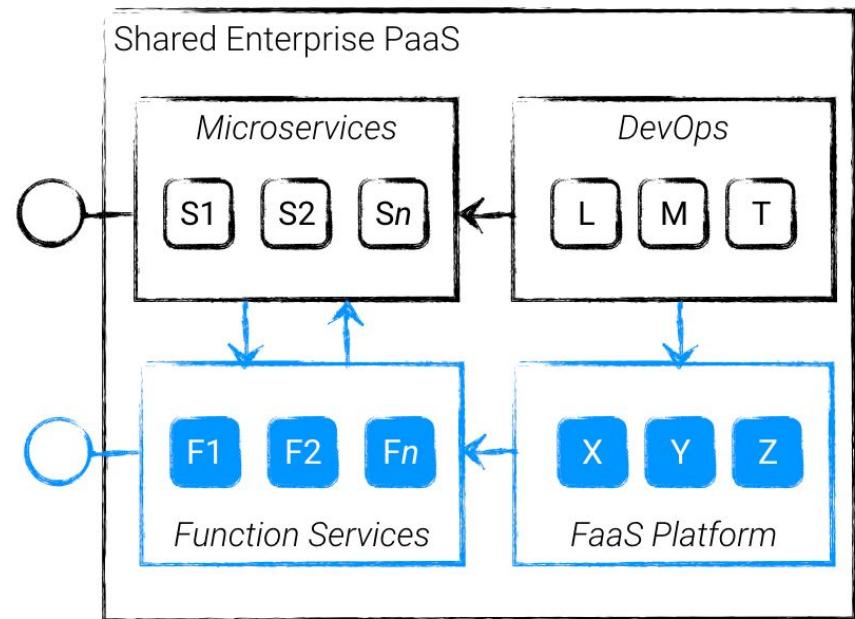


## Decomposition Trade-Offs

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>+ More flexible to scale</li><li>+ Runtime isolation (crash, slow-down, ...)</li><li>+ Independent releases, deployments, teams</li><li>+ Higher resources utilisation</li></ul> | <ul style="list-style-type: none"><li>- Distribution debt: Latency, Consistency</li><li>- Increased infrastructure complexity</li><li>- Increased troubleshooting complexity</li><li>- Increased integration complexity</li></ul> |
|--|---|

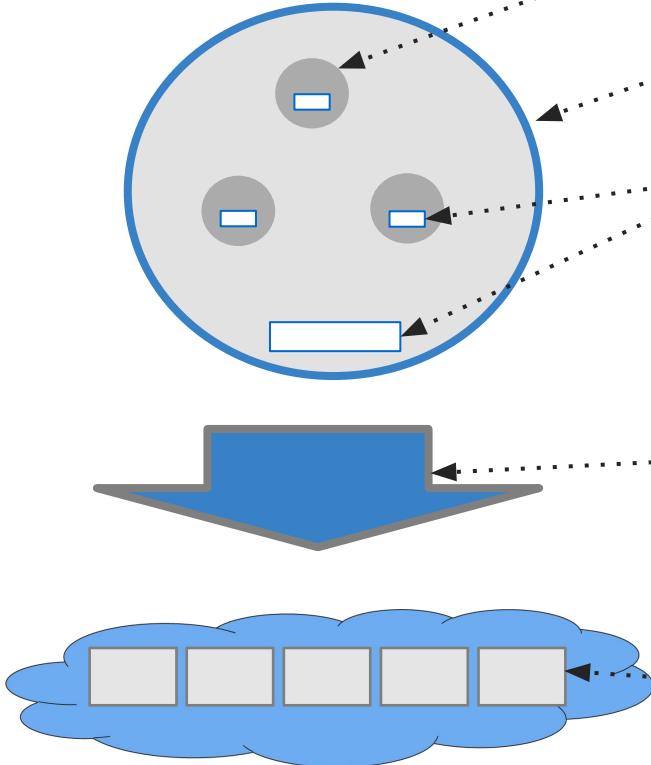
# Use Case Hybrid Architectures

- Kombination von Microservice Architektur mit EDA
- Nutzung von Function Services für Event-getriebene Use Cases
- Reduzierter Ressourcen-Verbrauch per Scale-to-Zero
- Integration in bestehende Enterprise PaaS Umgebung



# Kapitel Cluster Scheduling

# Terminologie



**Task:** Atomare Rechenaufgabe inklusive Ausführungsvorschrift.

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

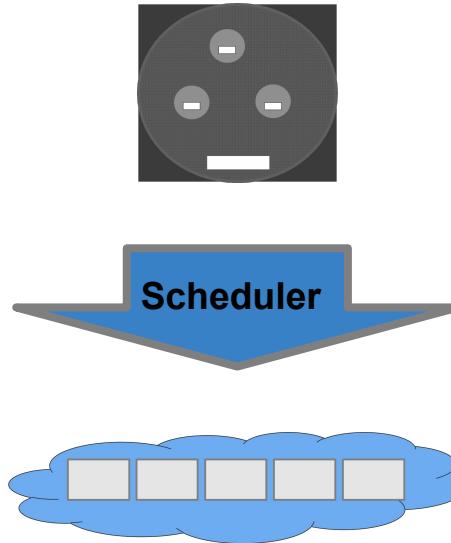
**Properties:** Ausführungsrelevante Eigenschaften der Tasks und Jobs, wie z.B.:

- 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



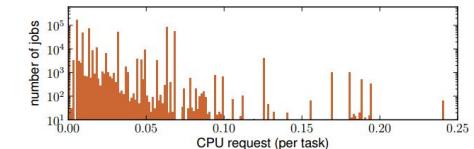
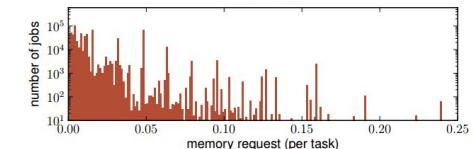
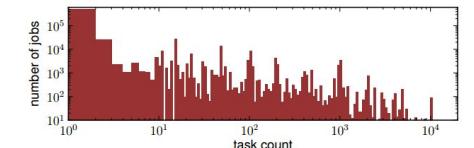
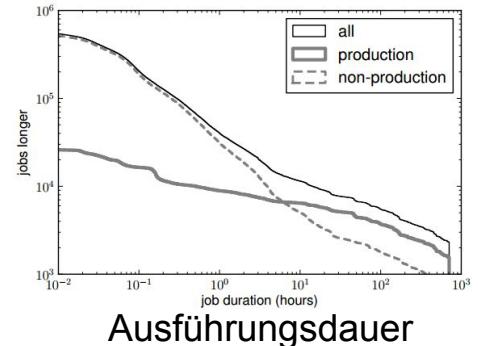
**Cluster Awareness:** Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.

**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.

# 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.
  - Ausführungszeit: 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.



Ressourcenverbrauch

# 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 Task 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.:

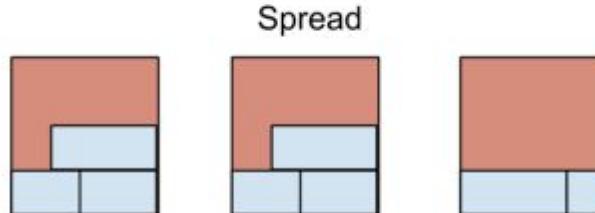
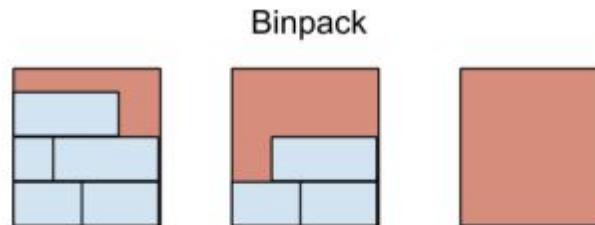
- Fairness: Kein Task sollte unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- Maximaler Durchsatz: So viele Tasks pro Zeiteinheit wie möglich.
- Minimale Wartezeit: Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks.
- Ressourcen-Auslastung: Möglichst 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)

# Scheduling ist eine Optimierungsaufgabe...

- ... und ist NP-vollständig.  
Die Optimierungsaufgabe lässt sich auf das Traveling 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.

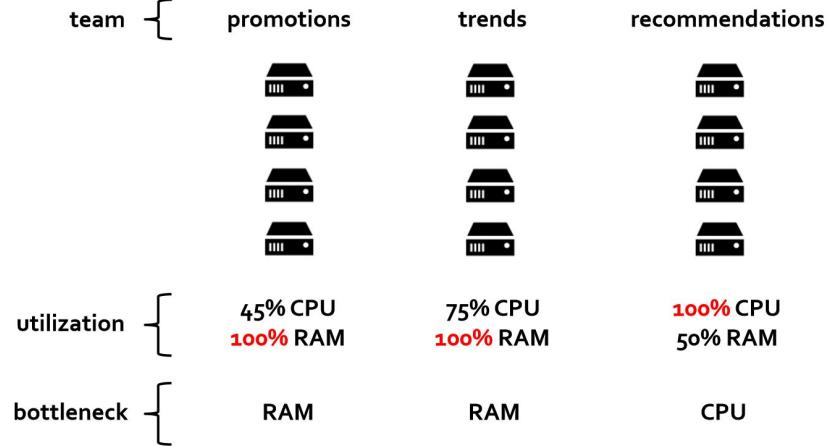
# Einfache Scheduling-Algorithmen

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



- Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSPP) 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>
- <https://stackoverflow.com/questions/39347670/explanation-of-yarns-drf>

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

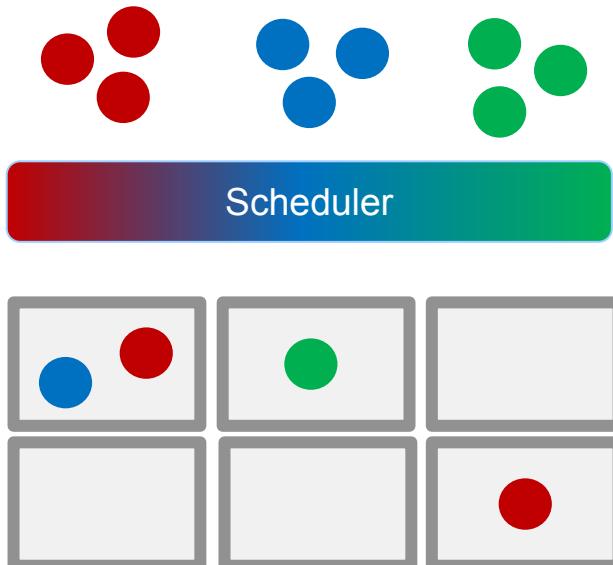
- Es werden Job Queues definiert und zu jeder Queue eine Kapazitätszusage in Ressourcenanteil vom Cluster definiert.
- Fairness-Auffassung: Diese Kapazitätszusage wird stets eingehalten. Der Scheduling-Algorithmus stellt sicher, dass diese Fairness stets sichergestellt wird.
- Damit das Cluster dafür 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.

# Scheduler-Architektur Variante 1: kein Scheduler



Statische Partitionierung

# Scheduler-Architektur Variante 2: Monolithischer Scheduler



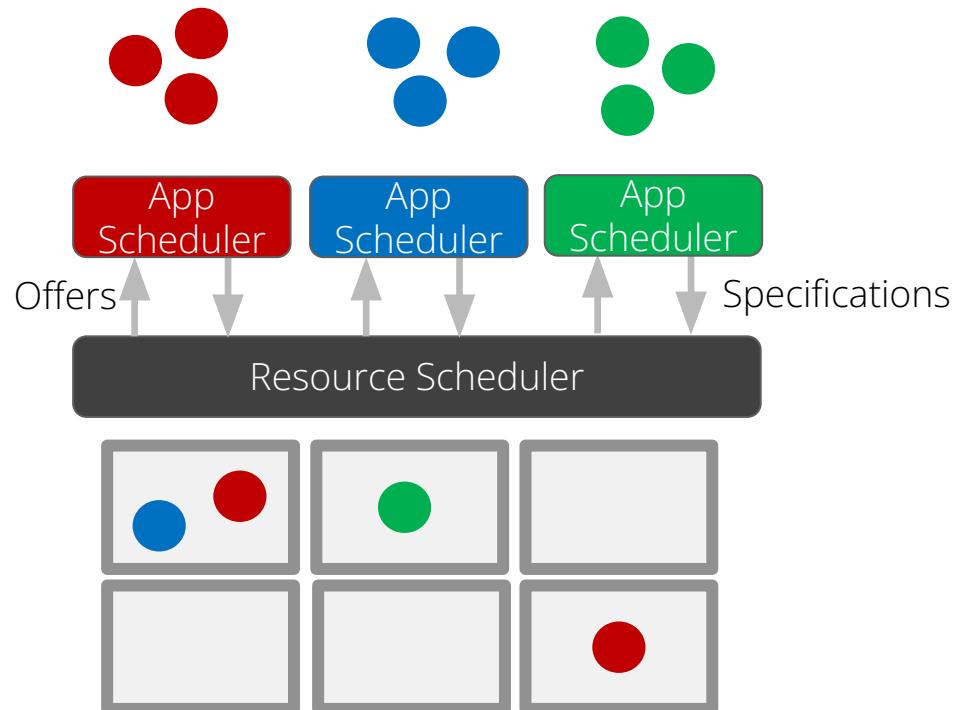
## Vorteile:

- 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.

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



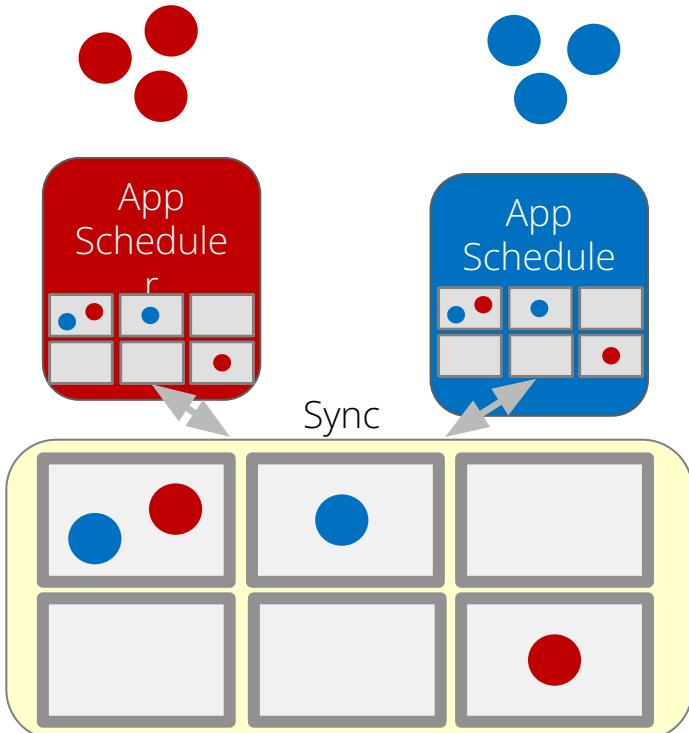
## Vorteile:

- 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



## Vorteile:

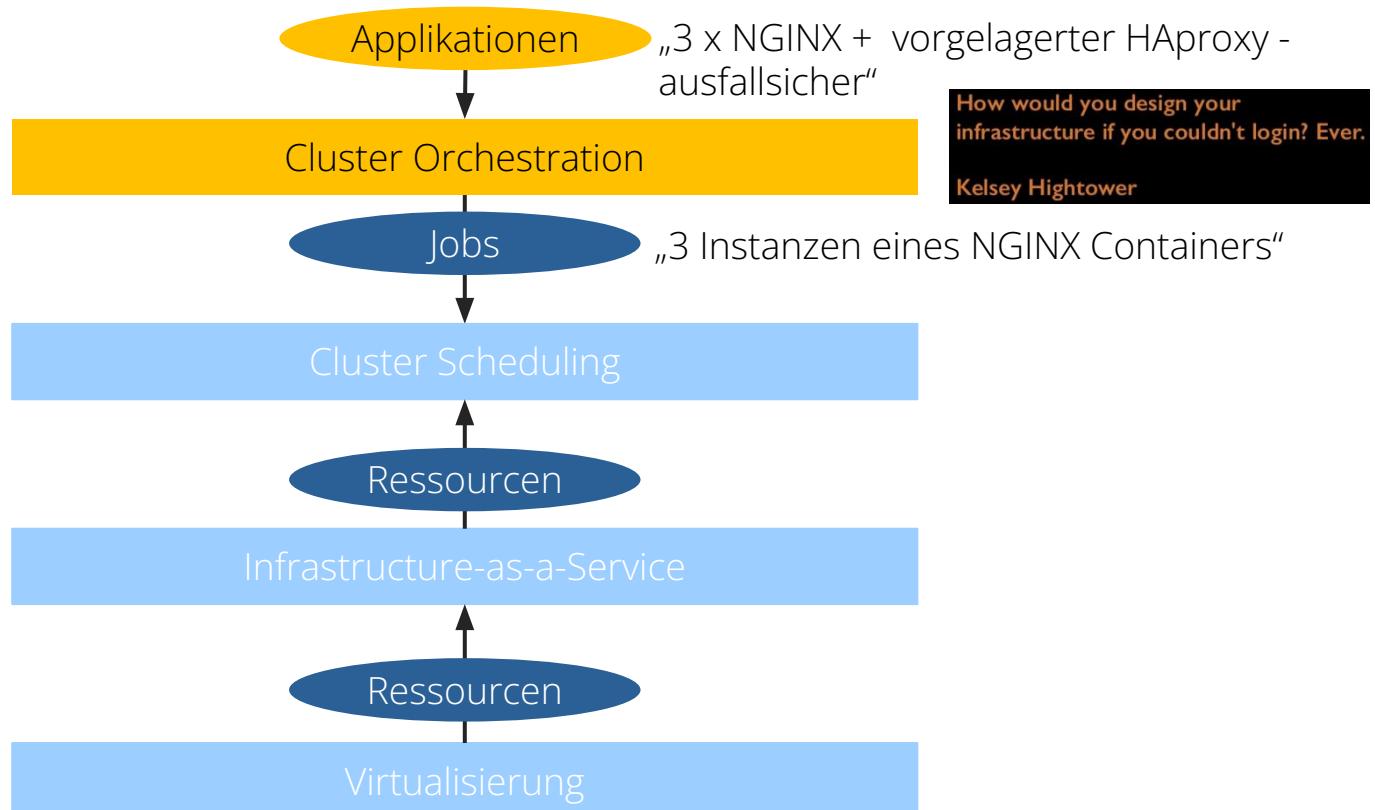
- Tendenziell geringerer Kommunikations-Overhead.

## 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.

# Kapitel Orchestrierung

# Big Picture: Wir sind nun auf Applikationsebene



# Cluster-Orchestrierung

- Ziel: 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 benötigten Schnittstellen und Betriebskomponenten 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.



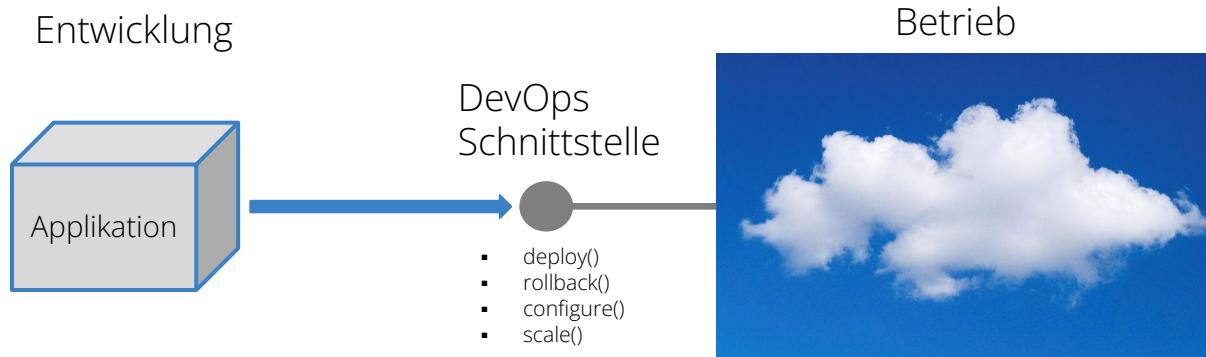
Cluster-Orchestrizer



**Steuerungsaktivitäten im Cluster:**

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

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



# Cluster-Orchestrator automatisieren verschiedene Betriebsaufgaben für Anwendungen auf einem Cluster (1/2):

- 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.

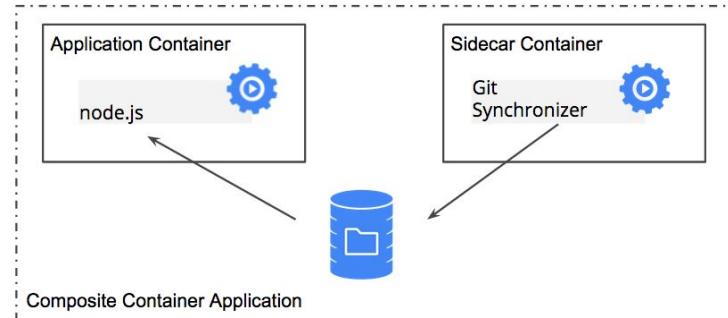
# Cluster-Orchestrator automatisieren verschiedene Betriebsaufgaben für Anwendungen auf einem Cluster (2/2):

- 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.

# Sidecar Containers

Sidecar containers extend and enhance the “main” container, they take existing containers and make them better. As an example, consider a container that runs the Nginx web server. Add a different container that syncs the file system with a git repository, share the file system between the containers and you have built Git push-to-deploy. But you’ve done it in a modular manner where the git synchronizer can be built by a different team, and can be reused across many different web servers (Apache, Python, Tomcat, etc). Because of this modularity, you only have to write and test your git synchronizer once and reuse it across numerous apps. And if someone else writes it, you don’t even need to do that.

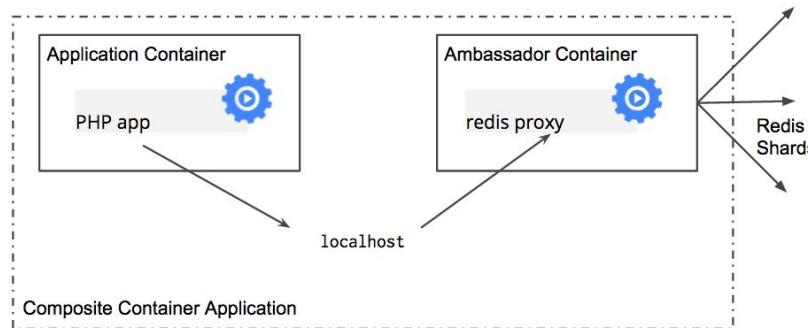
Quelle: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>



# Ambassador containers

Ambassador containers proxy a local connection to the world. As an example, consider a Redis cluster with read-replicas and a single write master. You can create a Pod that groups your main application with a Redis ambassador container. **The ambassador is a proxy is responsible for splitting reads and writes and sending them on to the appropriate servers.** Because these two containers share a network namespace, they share an IP address and your application can open a connection on “localhost” and find the proxy without any service discovery. **As far as your main application is concerned, it is simply connecting to a Redis server on localhost.** This is powerful, not just because of separation of concerns and the fact that different teams can easily own the components, but also because in the development environment, you can simply skip the proxy and connect directly to a Redis server that is running on localhost.

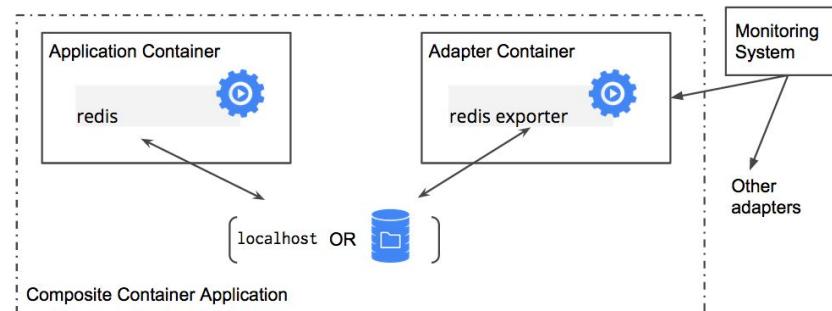
Quelle: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>



# Adapter containers

Adapter containers standardize and normalize output. Consider the task of monitoring N different applications. Each application may be built with a different way of exporting monitoring data. (e.g. JMX, StatsD, application specific statistics) but every monitoring system expects a consistent and uniform data model for the monitoring data it collects. By using the adapter pattern of composite containers, you can transform the heterogeneous monitoring data from different systems into a single unified representation by creating Pods that groups the application containers with adapters that know how to do the transformation. Again because these Pods share namespaces and file systems, the coordination of these two containers is simple and straightforward.

Quelle: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>





# kubernetes

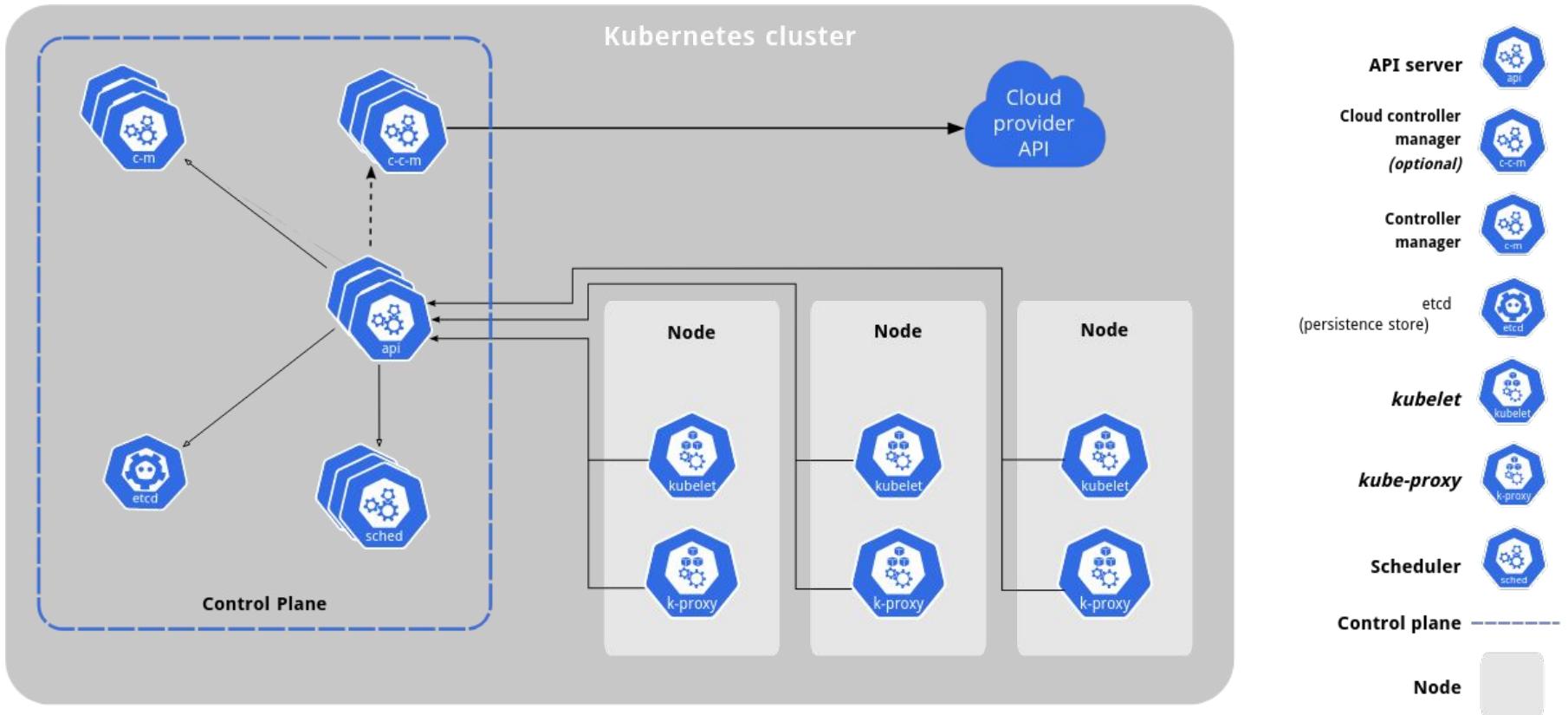
Kubernetes (auch als K8s bezeichnet) ist ein **Open-Source**-System zur **Automatisierung** der **Bereitstellung**, **Skalierung** und **Verwaltung** von **Container-Anwendungen**, das ursprünglich von Google entworfen und an die Cloud Native Computing Foundation (CNCF) gespendet wurde. Es zielt darauf ab, eine „Plattform für ... Anwendungscontainer[n] auf **verteilten Hosts**“ zu liefern. Es unterstützt eine Reihe von Container-Tools, einschließlich **Docker**.

[Quelle: Wikipedia](#)

# Wieso?!

- Ungelöst: Effiziente Nutzung der Hardware in einem Rechenzentrum
  - Ein Server pro Team? Was passiert, wenn ein Team keinen ganzen braucht?
- Virtualisierung und Elastizität löst das Problem der Hardware-Beschaffung
- Ungelöst: Development & Deployment der Anwendungen
- Trend: Weg vom Monolithen, hin zu Microservices
  - Verstärkt das Problem des Deployments
  - Wie finden sich die Microservices gegenseitig?
- Trend: Zero-downtime Deployments
- Lösung: Kubernetes als Standard
  - Anwendung in Container verpacken
  - Deployment beschreiben
  - Kubernetes kümmert sich um den Betrieb: startet neu, skaliert, etc.
  - Zero-Downtime und Rollbacks inkl.

# Architektur von Kubernetes



# Aufgaben der Bausteine auf einem Node

- **Pods** are the smallest deployable units of computing that you can create and manage in Kubernetes.
- Kubernetes runs your workload by placing containers into Pods to run on **Nodes**. A node may be a virtual or physical machine, depending on the cluster.
- **Container runtime**: "The container runtime is the software that is responsible for running containers. Kubernetes supports several container runtimes, one of them is Docker"
- **kubelet**: "An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod."
- A **service** is an abstract way to expose an application running on a set of Pods as a network service.
- **kube-proxy**: "kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept."

# Pods & Deployments

- Pods sind die kleinste schedulbare Einheit in Kubernetes
- Ein Deployment beschreibt, aus welchen Containern der Pod besteht und wie er konfiguriert ist

# Probes

- <https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#container-probes>
- Liveness probes prüfen, ob der Container noch am Leben ist. Falls nicht, wird dieser neu gestartet.
- Readiness probes prüfen, ob der Container bereit ist, Traffic zu bekommen. Falls nicht, wird dieser aus dem Load Balancing genommen.
- Mit einer Startup Probe kann das Problem von langsam startenden Containern mitigiert werden

# Resource constraints

- <https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/>
- Resource Requests helfen dem Scheduler, einen geeigneten Node für den Pod zu finden
- Resource Limits schützen vor amoklaufenden Containern, in dem sie den Container abschießen, sollte dieser die Memory-Limits erreichen
- Limits können auf Speicher und auf CPU gesetzt werden

# Services

- <https://kubernetes.io/docs/concepts/services-networking/service/>
- Services machen Anwendungen in Pods für andere Pods im Kubernetes-Cluster zugreifbar
- Services verwenden Label selectors, um die Pods zu finden, auf die der Service verweist
- Services load-balancen zwischen mehreren Pods

# Ingress

"An API object that manages **external access to the services** in a cluster, typically HTTP. Ingress may provide load balancing, SSL termination and name-based virtual hosting"

<https://kubernetes.io/docs/concepts/services-networking/ingress/>

- Durch einen Ingress kann ein Service von außerhalb des Clusters erreicht werden

# Config Maps

- Config Maps werden verwendet, um eine Anwendung in Kubernetes zu konfigurieren
- Die Werte von Config-Maps können entweder als Umgebungsvariablen oder als Dateien im Container erscheinen

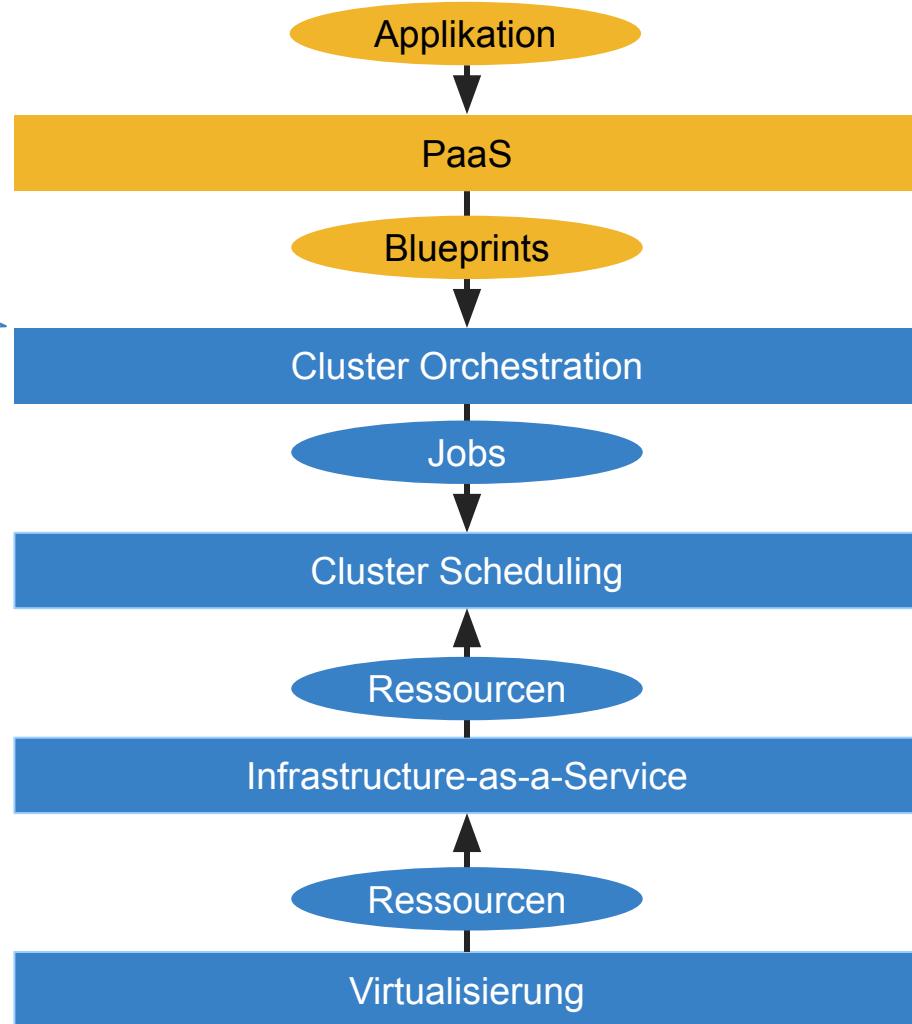
# Volumes

- Persistent Volumes stellen Speicherplatz bereit
  - Diese können entweder statisch oder dynamisch provisioniert werden
  - Dies ist für den Admin des Kubernetes-Clusters interessant
- Persistent Volume Claims fordern ein Persistent Volume an
  - Dies ist für den Entwickler des Kubernetes-Clusters interessant
- Über ein Volume kann ein Pod ein Persistent Volume Claim verwenden
- Volume mounts lassen das Volume in einem Container über einen Linux-Mount erscheinen

# Kapitel Platform-as-a-Service



# Das Big Picture



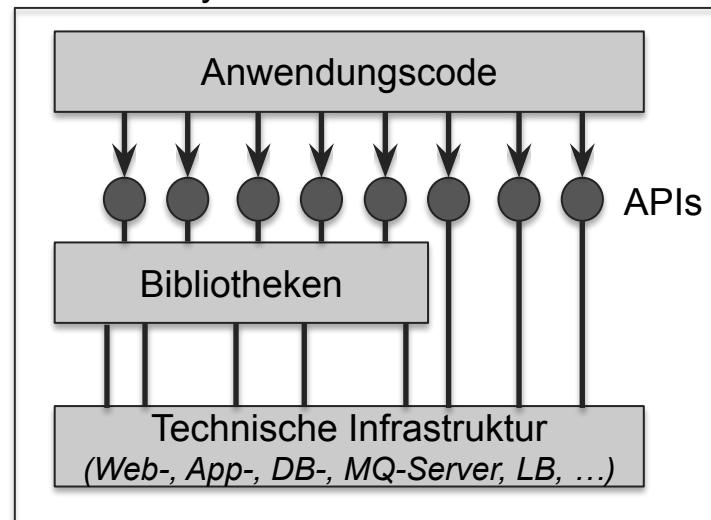
Hier ist man bereits bei 80% einer PaaS. Was noch fehlt:

- Wiederverwendung von Infrastruktur / APIs
- Komfort-Dienste für Entwickler

# Ein Problem: Stovepipe Architecture. Anwendungen aufwändig von Hand verdrahten.

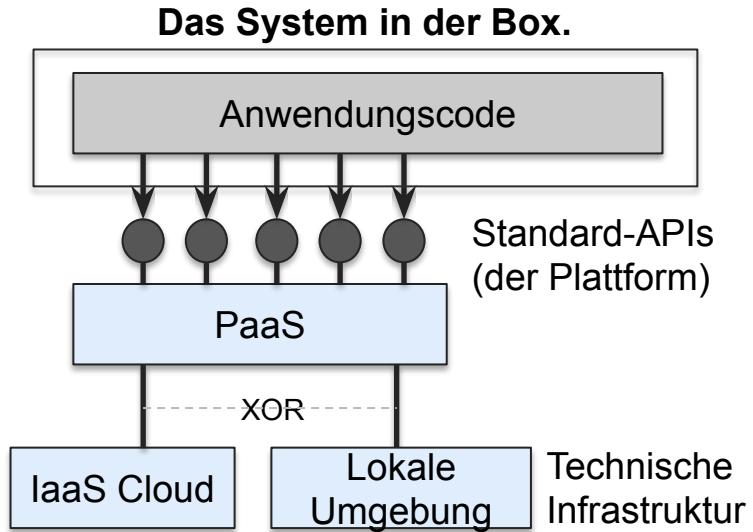


Das System: Mühevoll verdrahtet.

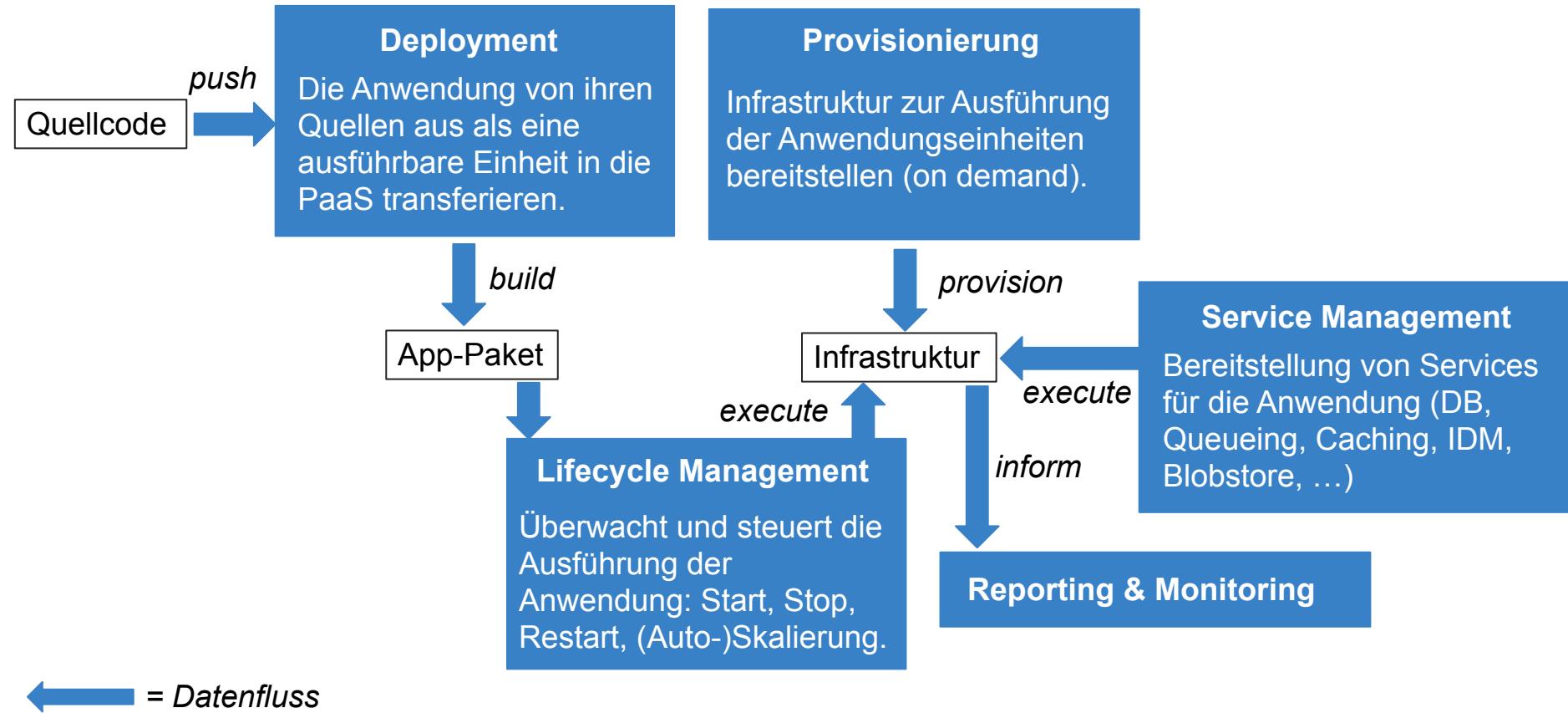


# Lösung: Plattform-as-a-Service bietet eine ad-hoc Entwicklungs- und Betriebsplattform.

- 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.
- Entwicklungswerzeuge (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.

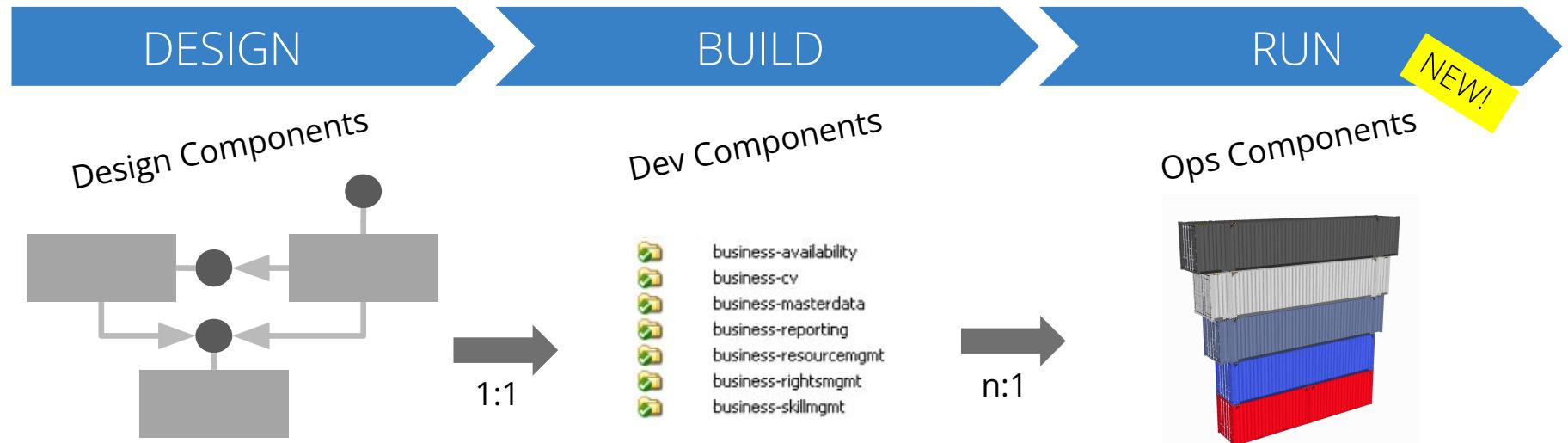


# Die funktionalen Bausteine einer PaaS Cloud



# Kapitel Cloud Native Softwarearchitektur

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

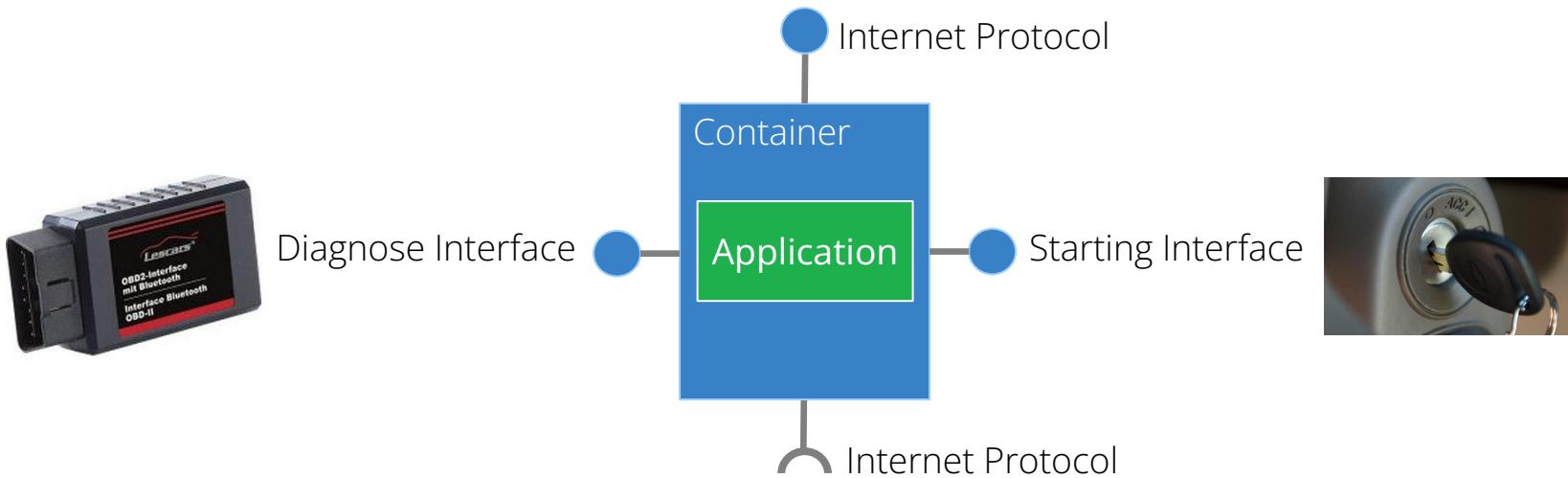


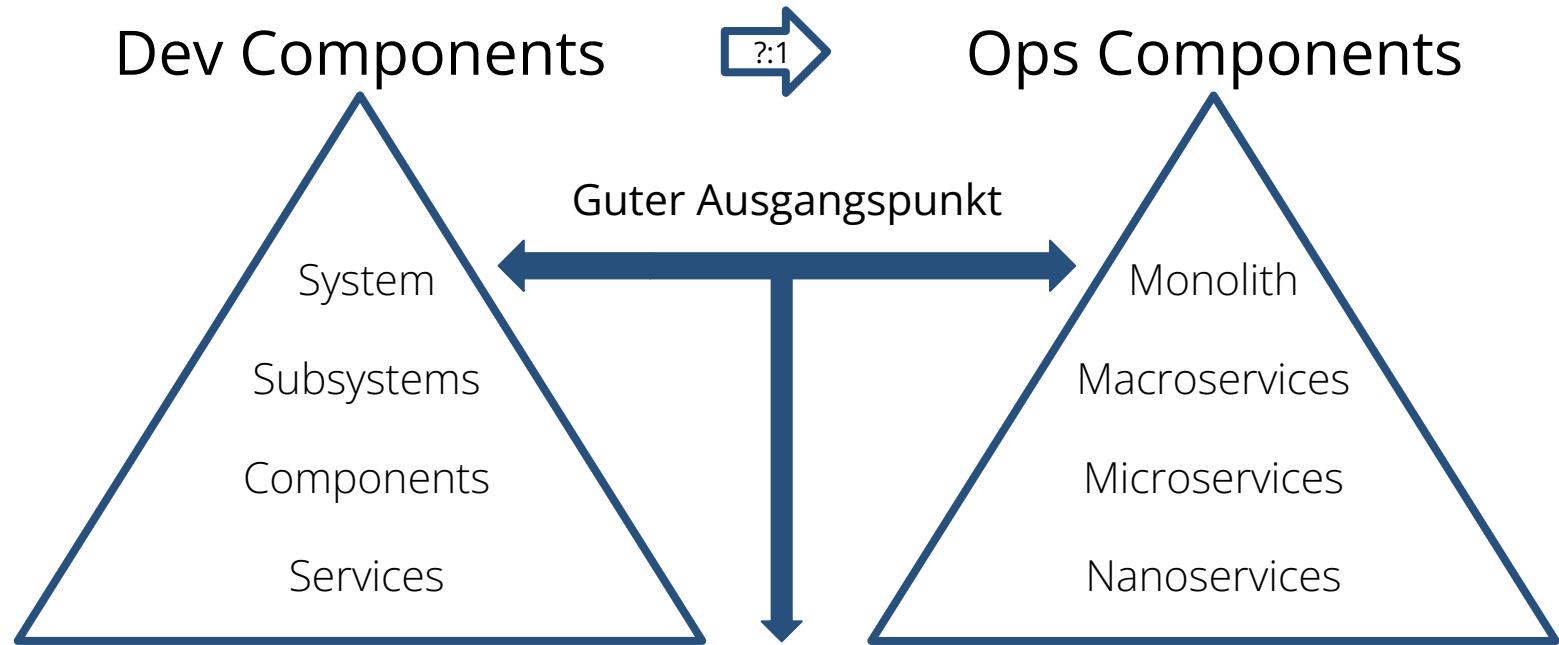
- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

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

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

# Die Anatomie einer Betriebs-Komponente





### Decomposition Trade-Offs

- + More flexible to scale
- + Runtime isolation (crash, slow-down, ...)
- + Independent releases, deployments, teams
- + Higher utilization possible
- Distribution debt: Latency
- Increasing infrastructure complexity
- Increasing troubleshooting complexity
- Increasing integration complexity

# Resilienz

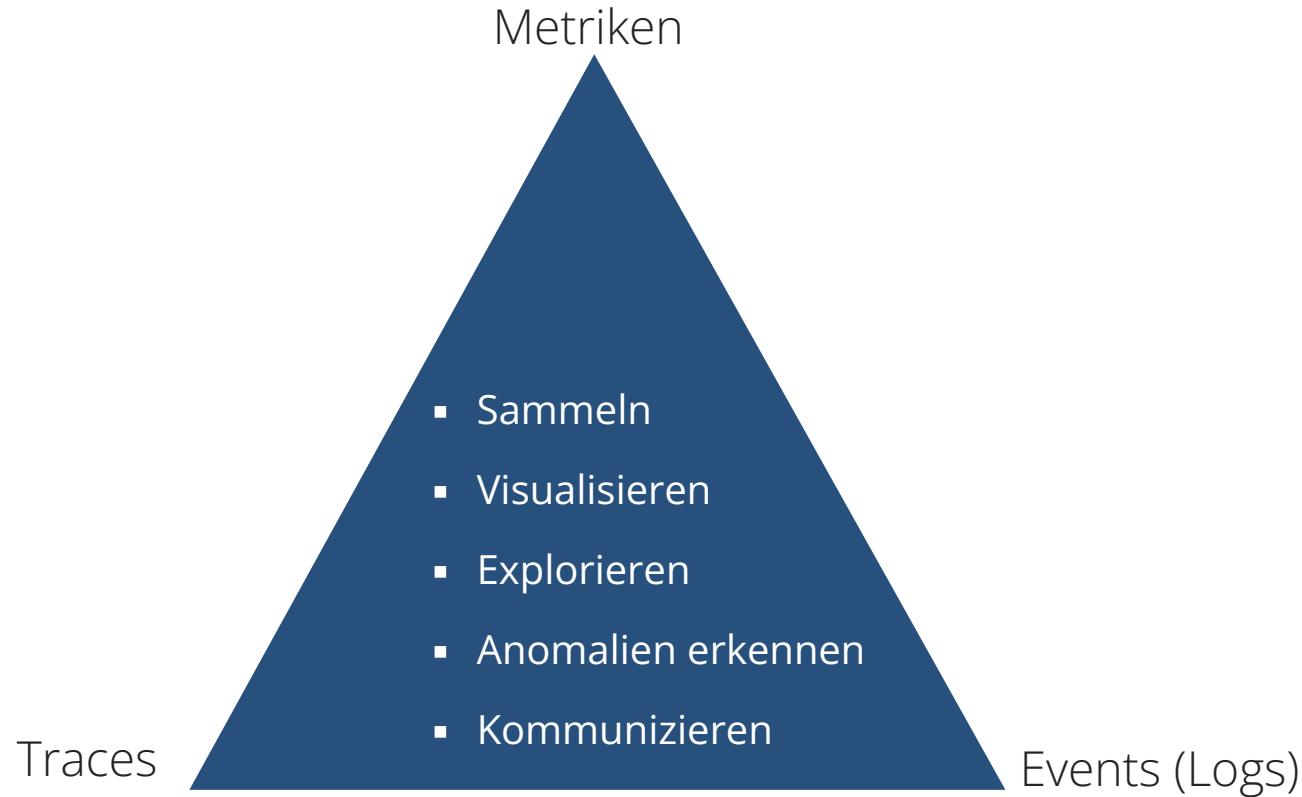
$$\text{Verfügbarkeit} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$



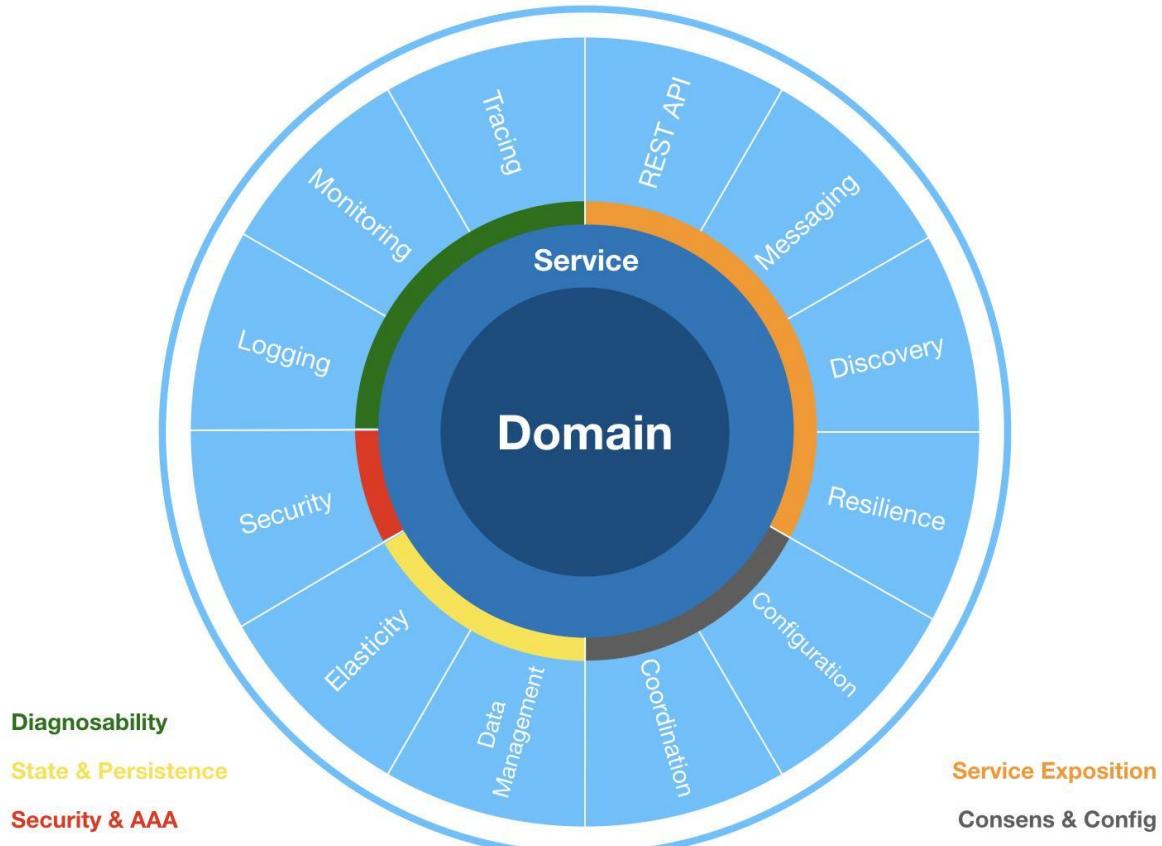
Resilienz: Die Fähigkeit eines Systems mit unerwarteten und fehlerhaften Situationen umzugehen

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

# Diagnostizierbarkeit



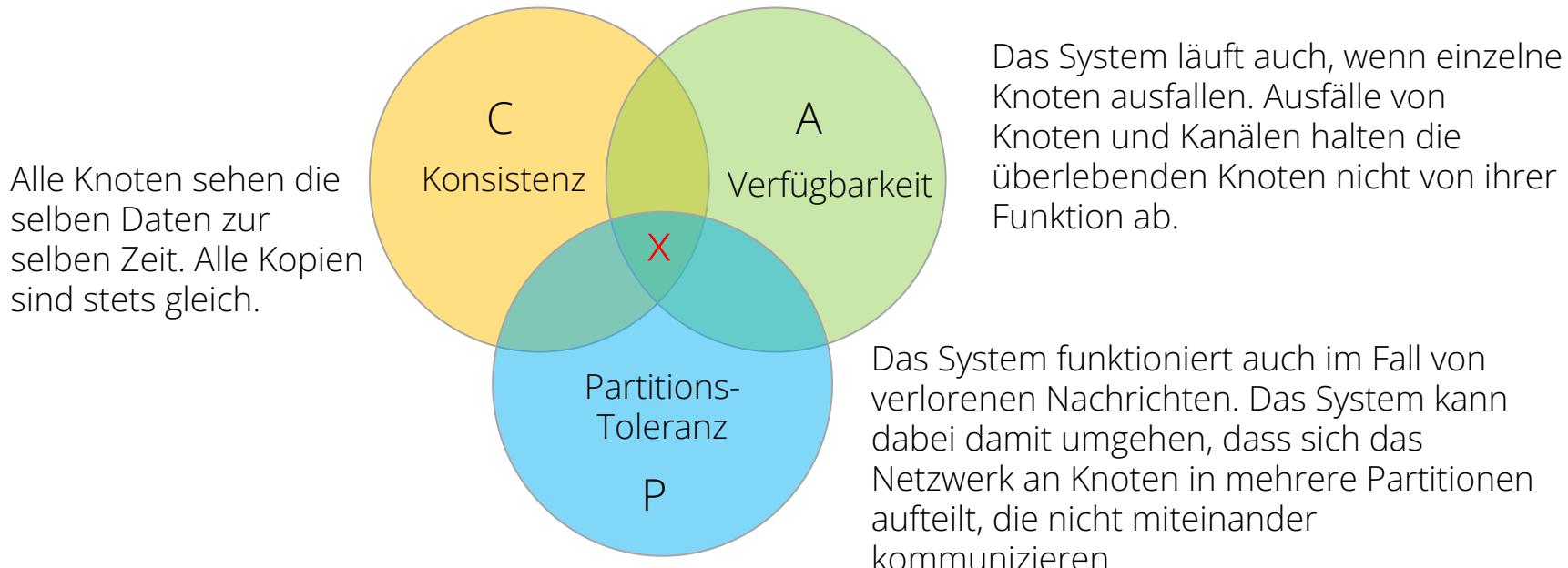
# Technische Aspekte von Microservices



# 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:



# Gossip Protokolle für Hoch-Verfügbarkeit

Grundlage: Ein Netzwerk an Agenten mit eigenem Zustand Agenten verteilen einen Gossip-Strom

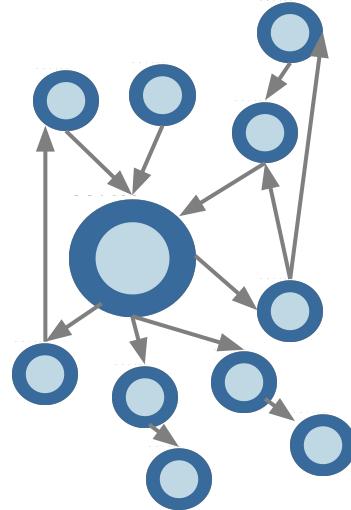
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

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, wenn

- die gleiche Nachricht von mehreren Seiten gehört wurde
- die Nachricht von Knoten stammt, denen der Agent vertraut
- keine aktuellere Nachricht vorhanden sind



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: 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.

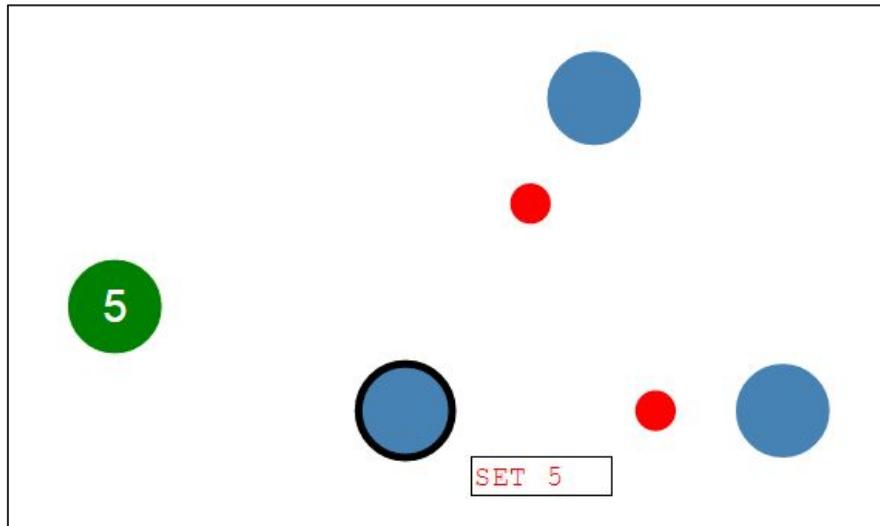
Nachteile:

- 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 Raft Konsens-Protokoll

Ongaro, Diego; Ousterhout, John (2013).

"In Search of an Understandable Consensus Algorithm".

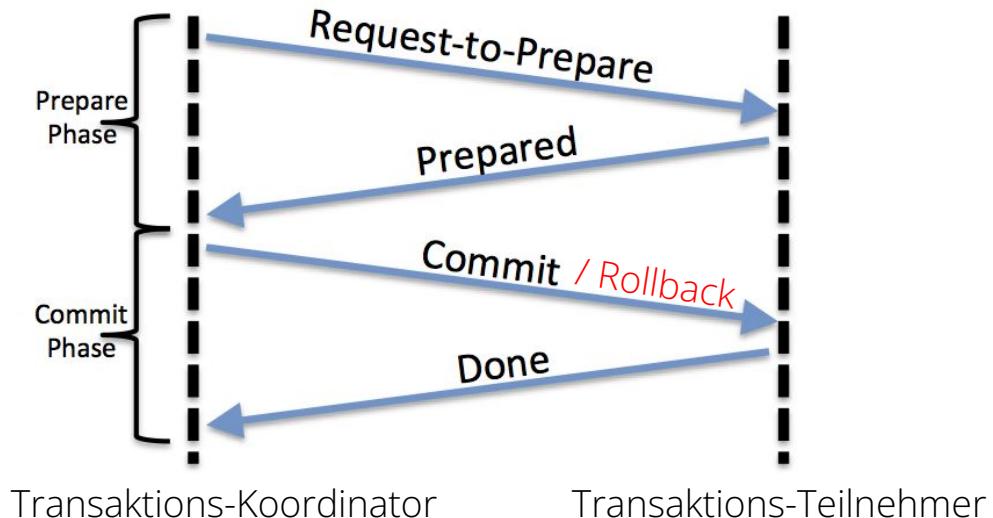


<http://thesecretlivesofdata.com/raft>

<https://raft.github.io/>

# Ist strenge Konsistenz über alle Knoten notwendig, so verbleibt das 2-Phase-Commit Protokoll (2PC)

Ein Transaktionskoordinator verteilt die Änderungen und aktiviert diese erst bei Zustimmung aller. Ansonsten werden die Änderungen rückgängig gemacht.



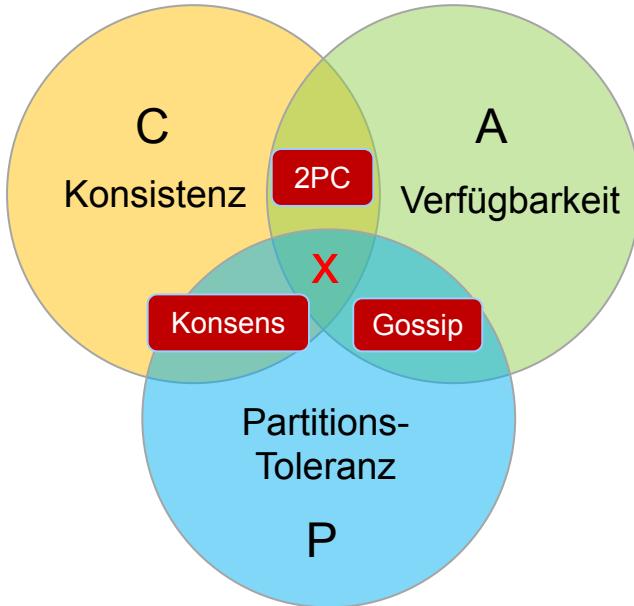
## Vorteil:

- Alle Knoten sind konsistent zueinander.

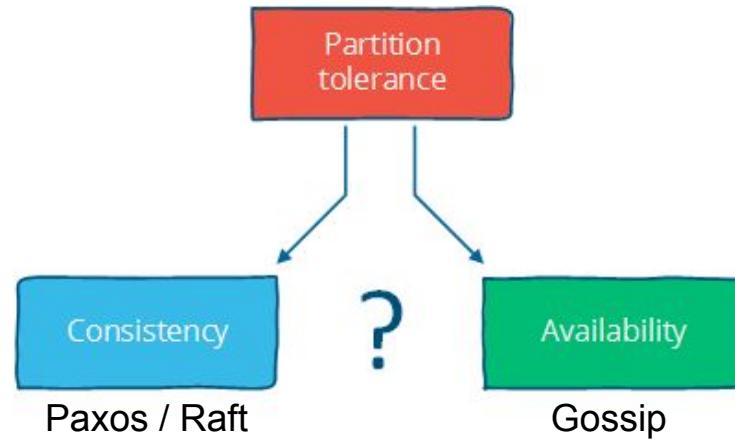
## Nachteile:

- Zeitintensiv, da stets alle Knoten zustimmen müssen.
- Das System funktioniert nicht mehr, sobald das Netzwerk partitioniert ist.

# Die vorgestellten Protokolle und das CAP Theorem



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

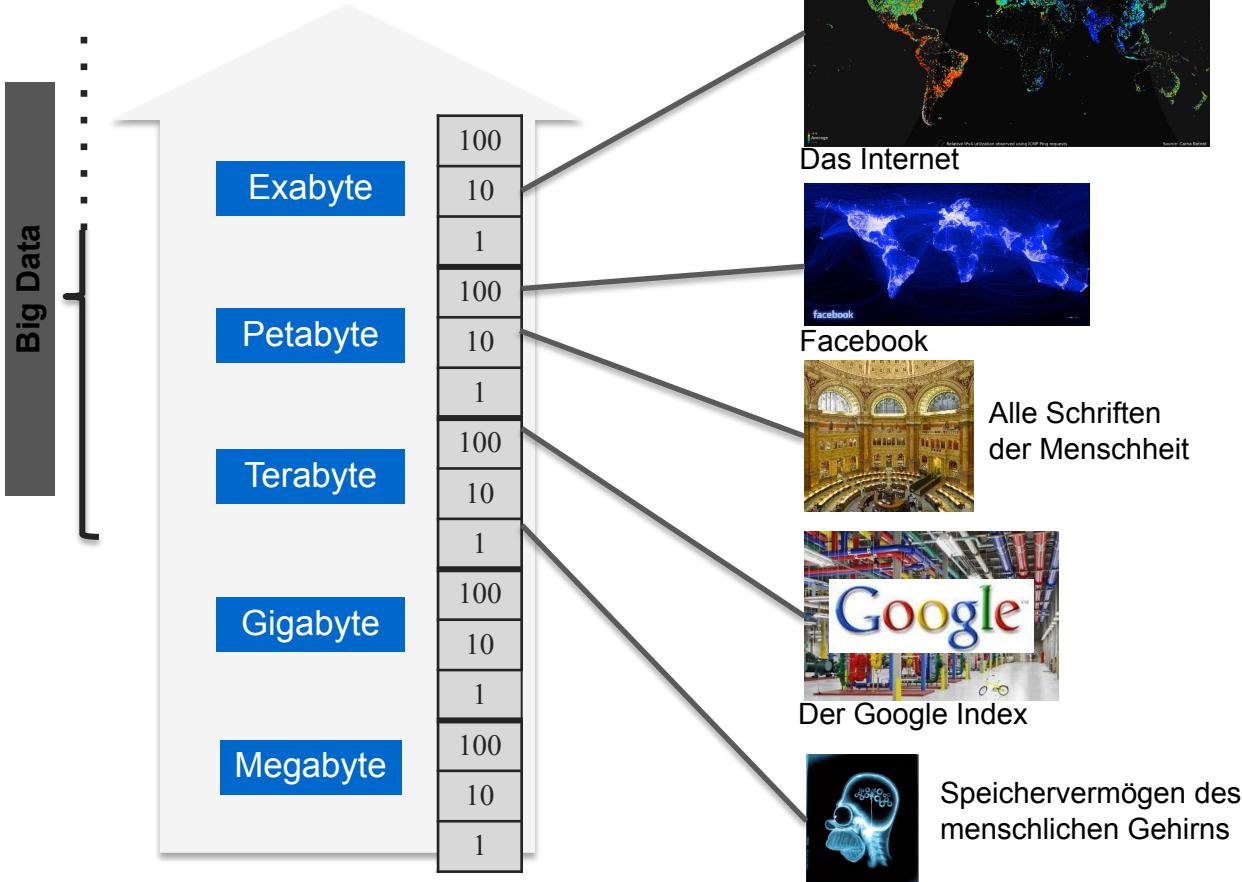


# Kapitel Big Data

# Big Data

Verarbeitung großer Datenmengen durch:

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



# 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.

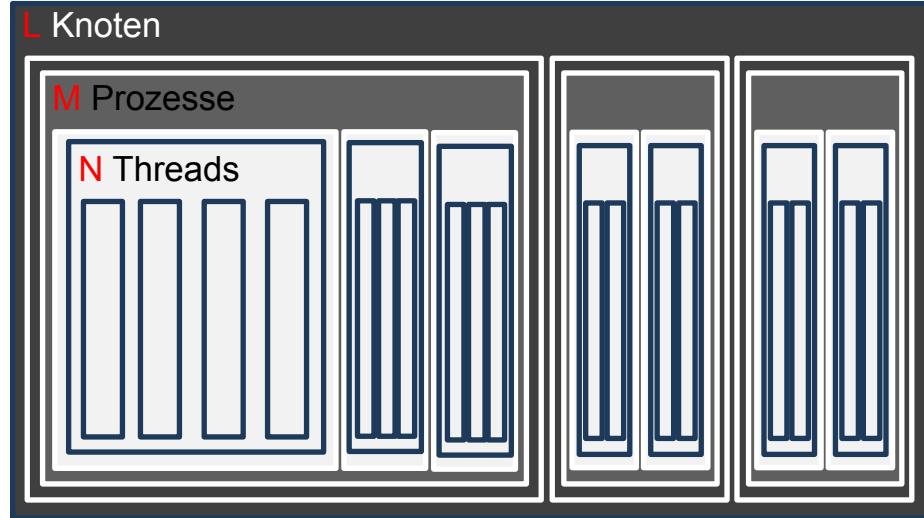
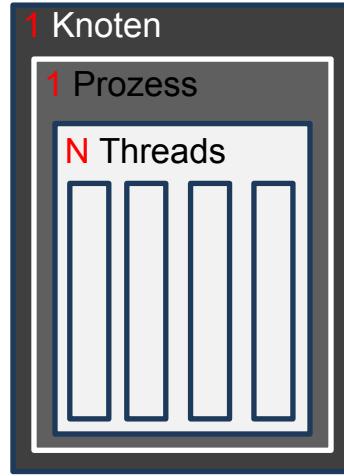
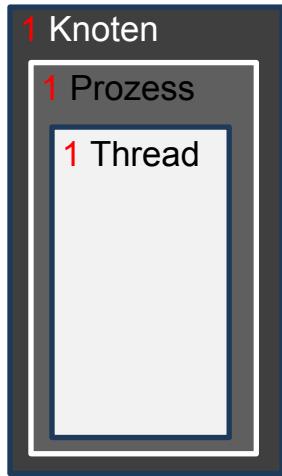
- Funktioniert gut: Quicksort. Aufwand:  $O(n \log n) \square n \times O(\log n)$

```
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. Dabei muss man die vier Paradigmen der Parallelität im Großen beachten:



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.

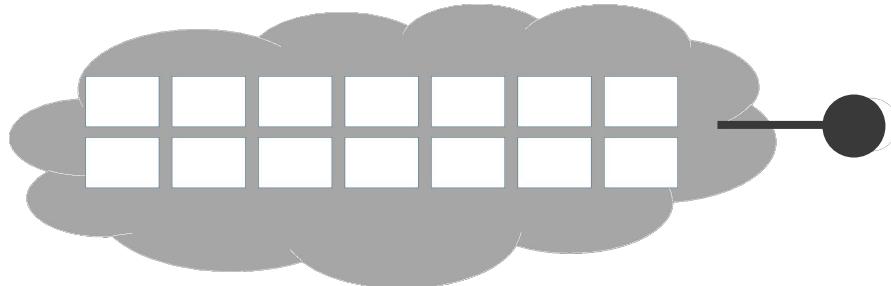
Folgt aus potenziell großer  
Datenmenge und  
Verarbeitungs-geschwindigkeit

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.

# Welche Lösungen gibt es dafür im Cloud Computing?



- **Big Data Engines (low level)**
  - MapReduce
  - RDD (Resilient Distributed Dataset)
- **Big Data Datenbanken (high level)**
  - NoSQL Datenbanken
  - NewSQL Datenbanken (NoSQL + SQL)
- Verteilte Dateisysteme
- In-Memory Data Grids / Elastic Memory

# Die *map* und *reduce* Funktion.

- Die **map** Funktion: Transformation einer Menge von Datensätzen in eine Zwischendarstellung. Erzeugt aus einem Schlüssel und einem Wert eine Liste an Schlüssel-Wert-Paaren.

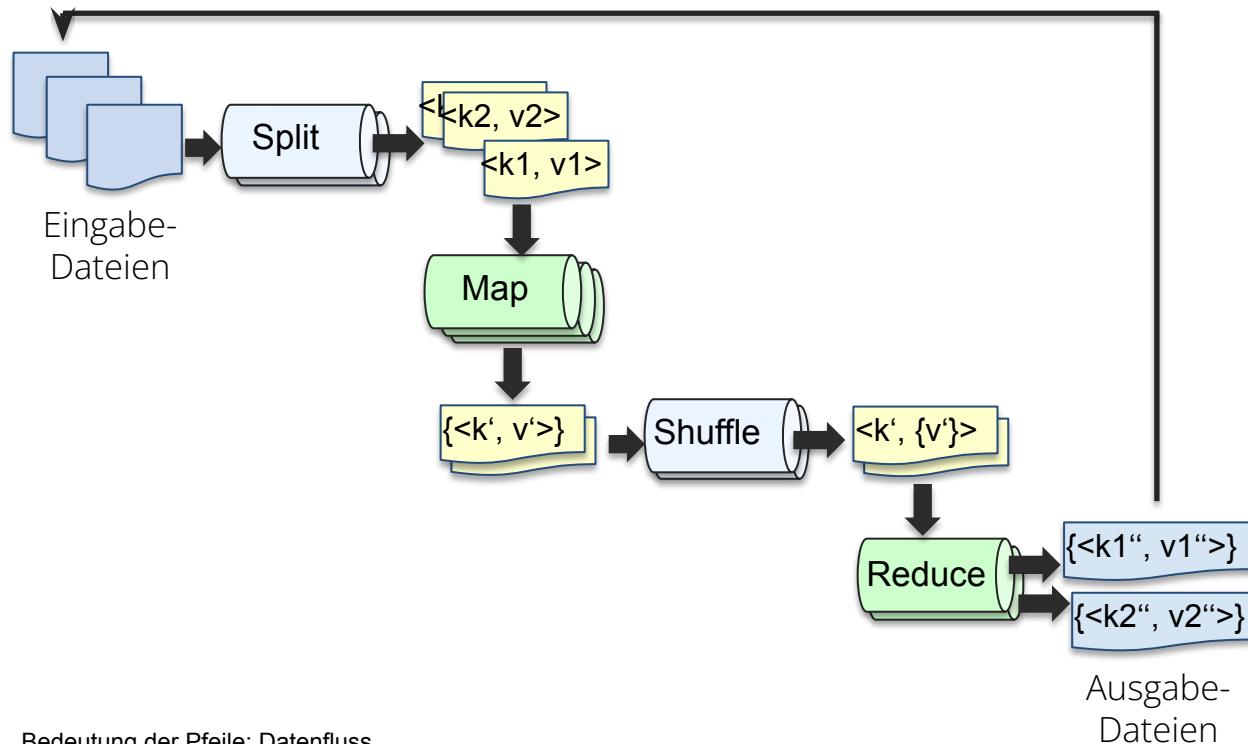
Signatur: **map**(k, v)  $\square$  list(<k', v'>)

- Die **reduce** Funktion: Reduktion der Zwischendarstellung auf das Endergebnis. Verarbeitet alle Werte mit gleichem Schlüssel zu einer Liste an Schlüssel-Wert-Paaren.

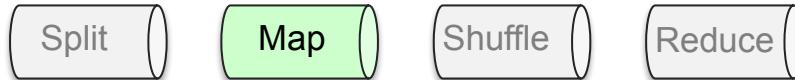
Signatur: **reduce**(k', list(v'))  $\square$  list(<k'', v''>)

- Dabei soll gelten:  $|list(<k'', v''>)| \ll |list(<k', v'>)|$

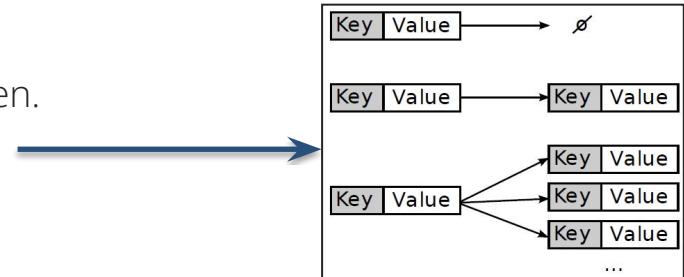
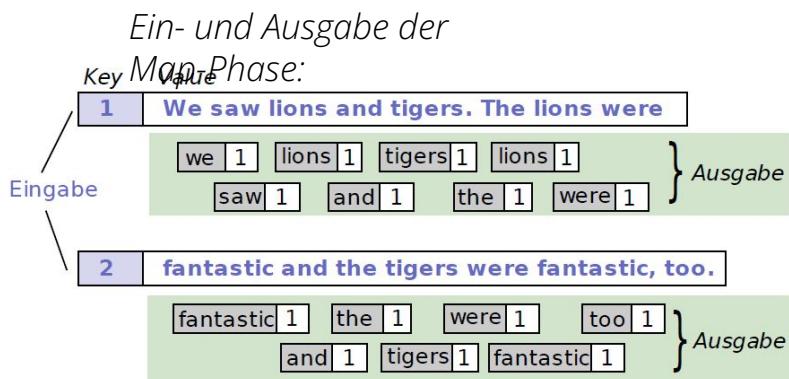
Programme werden in (mehrere) Map-Reduce-Zyklen aufgeteilt. Das Framework übernimmt die Parallelisierung.



# Die Map-Phase



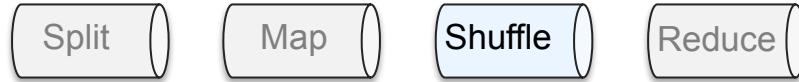
- Parallel Verarbeitung verschiedener Teilbereiche der Eingabedaten.
- Eingabedaten liegen in Form von Schlüssel/Wert-Paaren vor.
- Abbildung auf variable Anzahl von neuen Schlüssel/Wert-Paaren.  
Dabei sind alle Abbildungsvarianten zulässig:
- Beispiel: WordCount



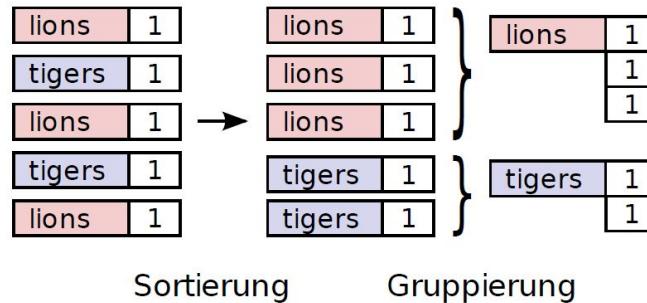
*Pseudocode*  
*Map-Phase:*

```
map(String key, String value):  
    //key: document name  
    //value: document contents  
    for each word in value:  
        EmitIntermediate(word, "1");
```

# Die Shuffle-Phase



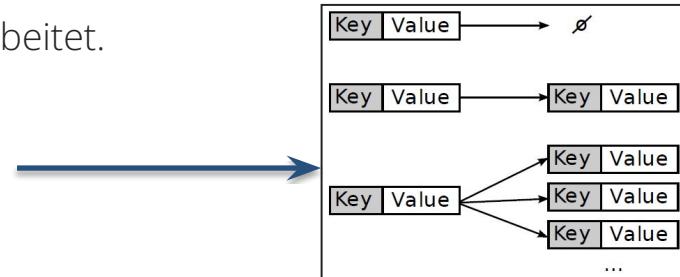
- Verarbeitung der Ergebnisse aus der Map-Phase.
- Ausgaben aus der Map-Phase werden entsprechend ihrem Schlüssel sortiert und gruppiert.
- Im Standard-Fall ist die Shuffle-Phase nicht parallelisiert.
- Sie kann jedoch mittels einer Vor-Sortierung in der Map-Phase über eine Partitionierungsfunktion (z.B. Hash) auf den Schlüssel parallelisiert werden.



# Die Reduce-Phase



- Parallel Verarbeitung von Ergebnis-Gruppen aus der Map-Phase.  
Es wird pro Reduce-Vorgang genau eine dieser Gruppen verarbeitet.
- Eingabedaten liegen in Form von Schlüssel-Wertlisten vor.
- Abbildung auf variable Anzahl an Schlüssel/Wert-Paaren.  
Dabei sind alle Abbildungsvarianten zulässig:



Ein- und Ausgabe der  
Reduce-Phase:

t	l	o	n	s
1				
1				
1				
1				



t	i	g	e	r	s
1					



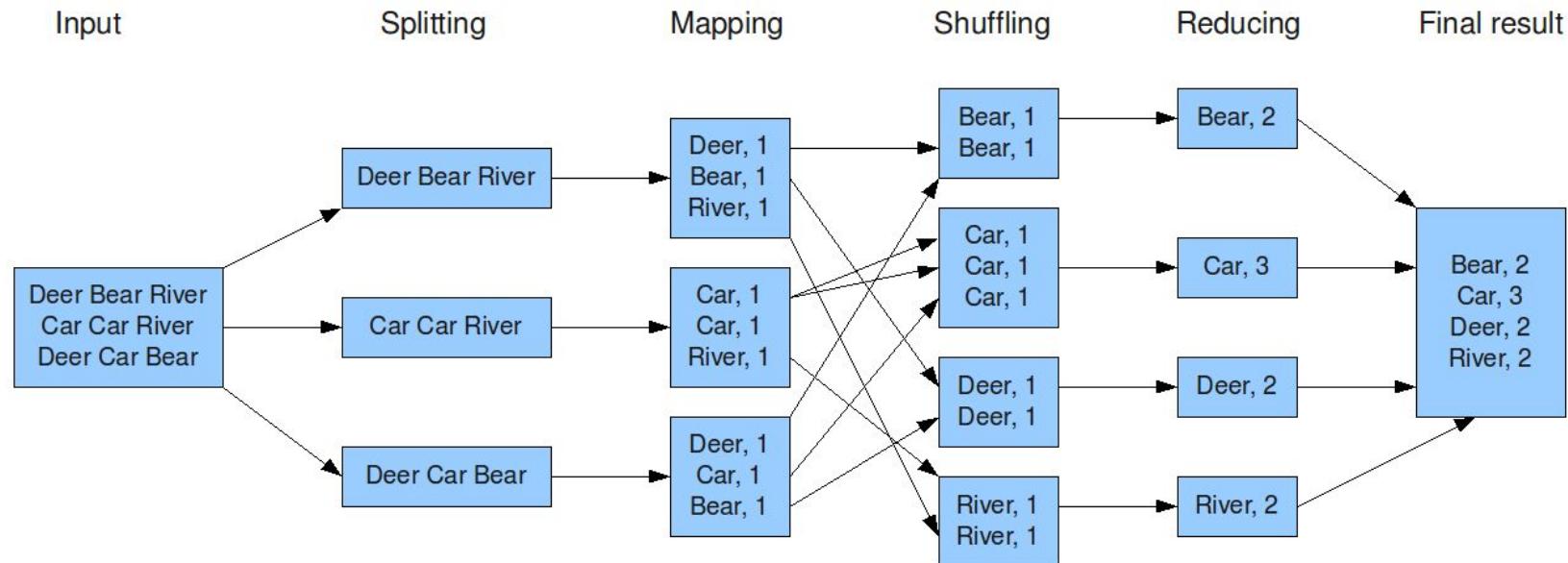
w	e	r	e
1			



Pseudocode  
Reduce-Phase:

```
reduce(String key, Iterator values):  
    //key: a word  
    //values: a list of counts  
    for each value in values:  
        result += ParseInt(value);  
    Emit(AsString(Key +“, “+result));
```

# Übersicht über alle Phasen



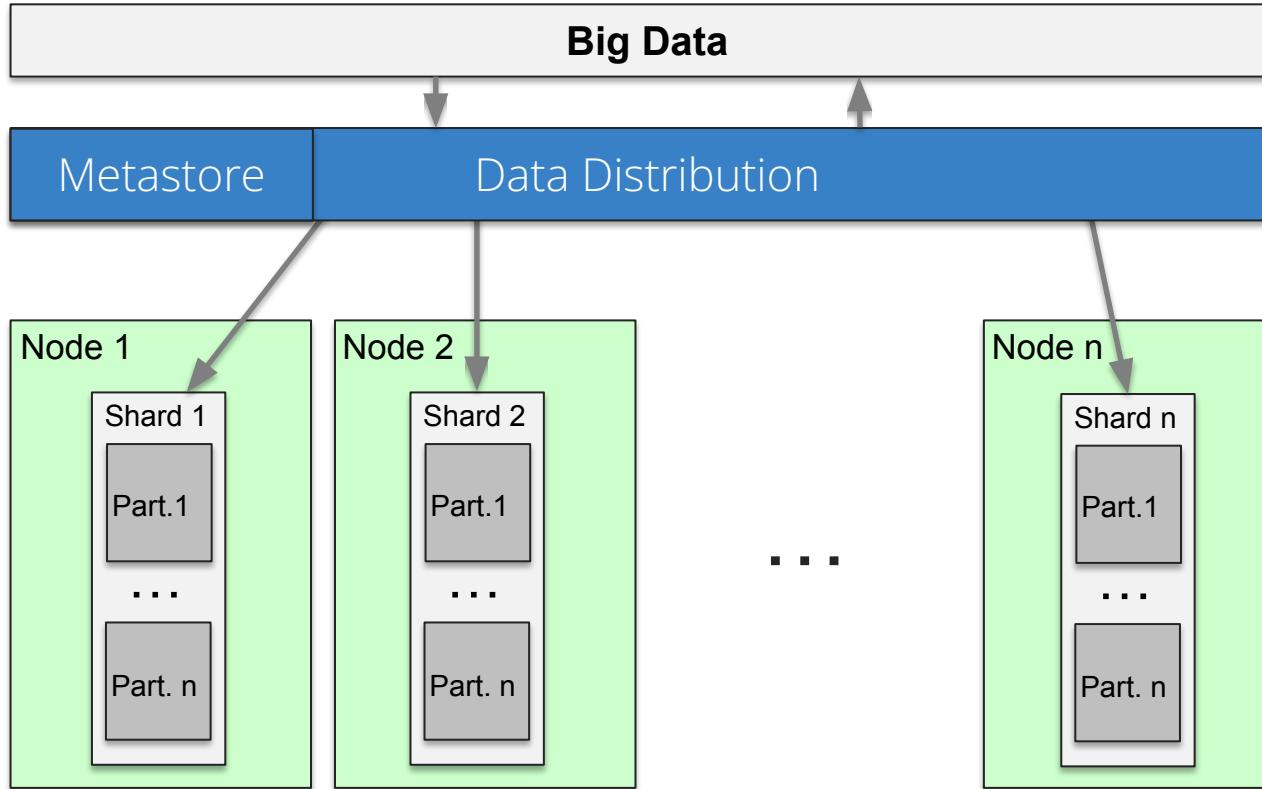
<http://blog.itteam.nl/2009/08/04/introduction-to-hadoop>

Die Resilient Distributed Dataset (RDD) Datenstruktur ist die Abstraktion des Spark Cores.

Eine RDD ist in der Außensicht ein klassischer Collection-Typ mit Transformations- und Aktionsmethoden.

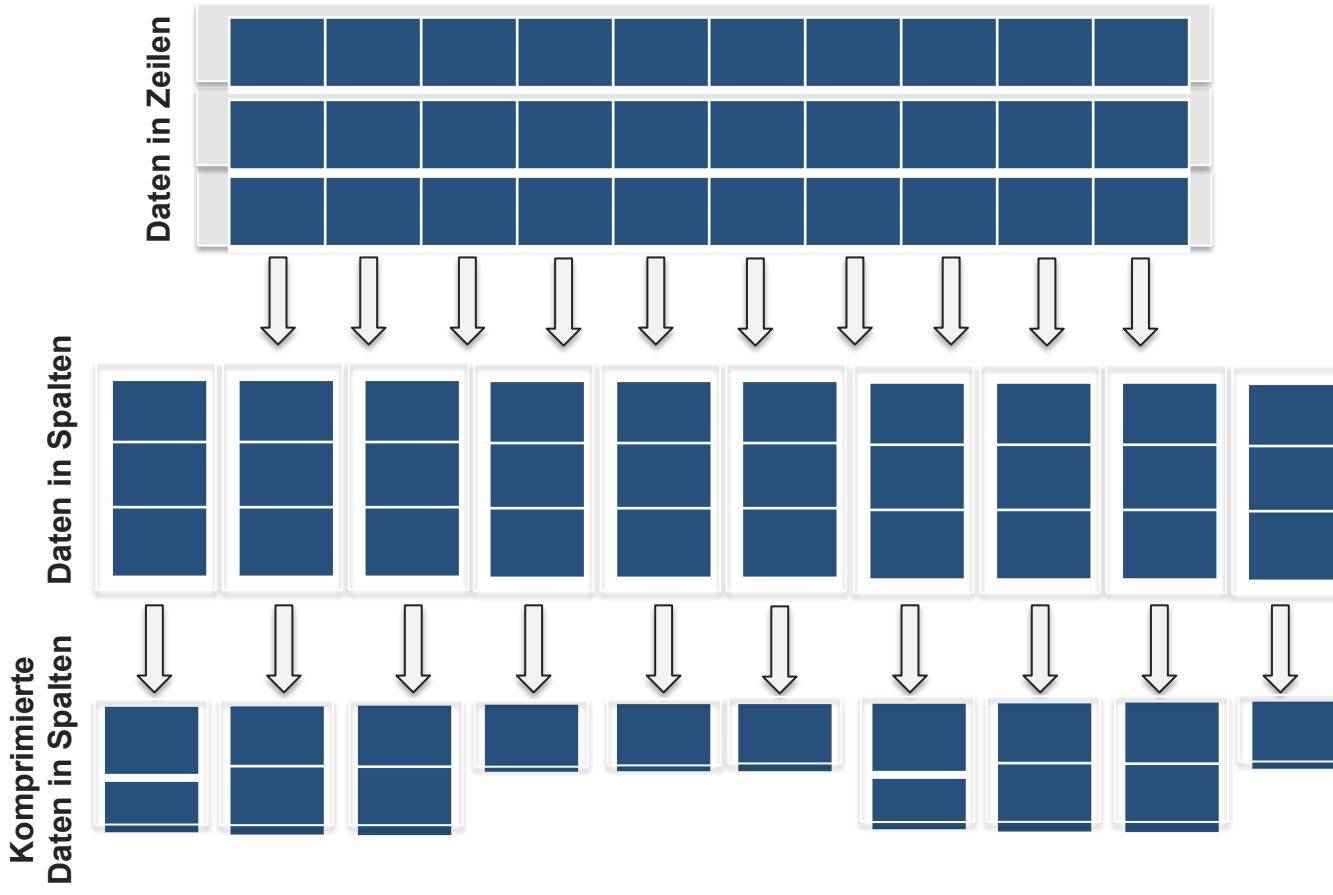
RDD □ RDD	RDD □ skalarer Typ, Collection, Storage
Transformations	Actions
<code>map(func)</code>	<code>take(N)</code>
<code>flatMap(func)</code>	<code>count()</code>
<code>filter(func)</code>	<code>collect()</code>
<code>groupByKey()</code>	<code>reduce(func)</code>
<code>reduceByKey(func)</code>	<code>takeOrdered(N)</code>
<code>mapValues(func)</code>	<code>top(N)</code>
...	...

# Sharding and Partitioning: Verteilung und Stückelung von großen Datenmengen



(Re-) Sharding- und Partitioning-Funktion:  
f(Daten) □ Shard  
f(Daten) □ Partition.  
+ Replikationsstrategie.  
+ Konsistenzstrategie.

# Spalten-orientierte Datenspeicherung



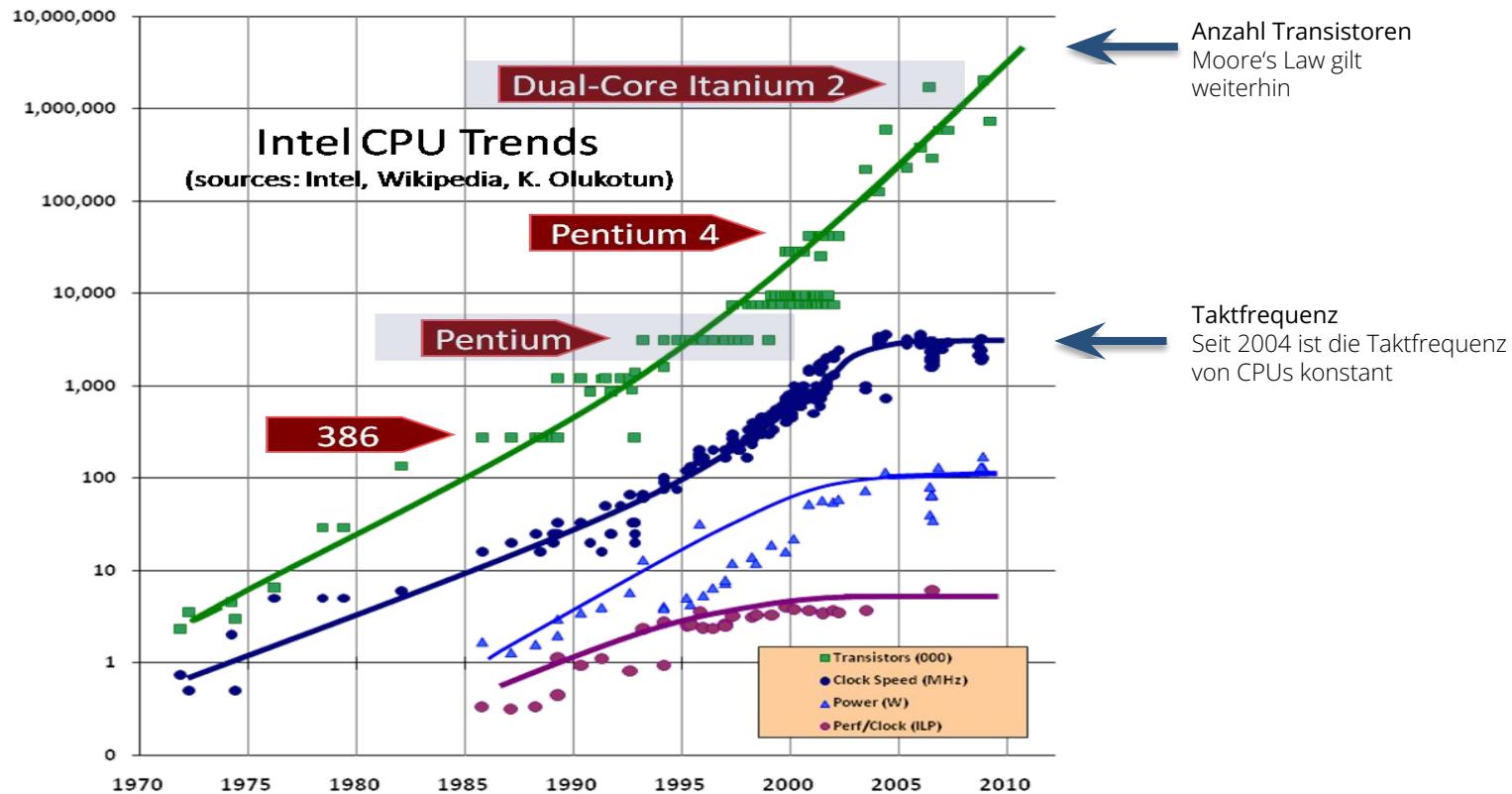
The fastest I/O is the one that never takes place: Es werden nur diejenigen Spalten gelesen, die benötigt werden (gerade bei breiten Tabellen wichtig)

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

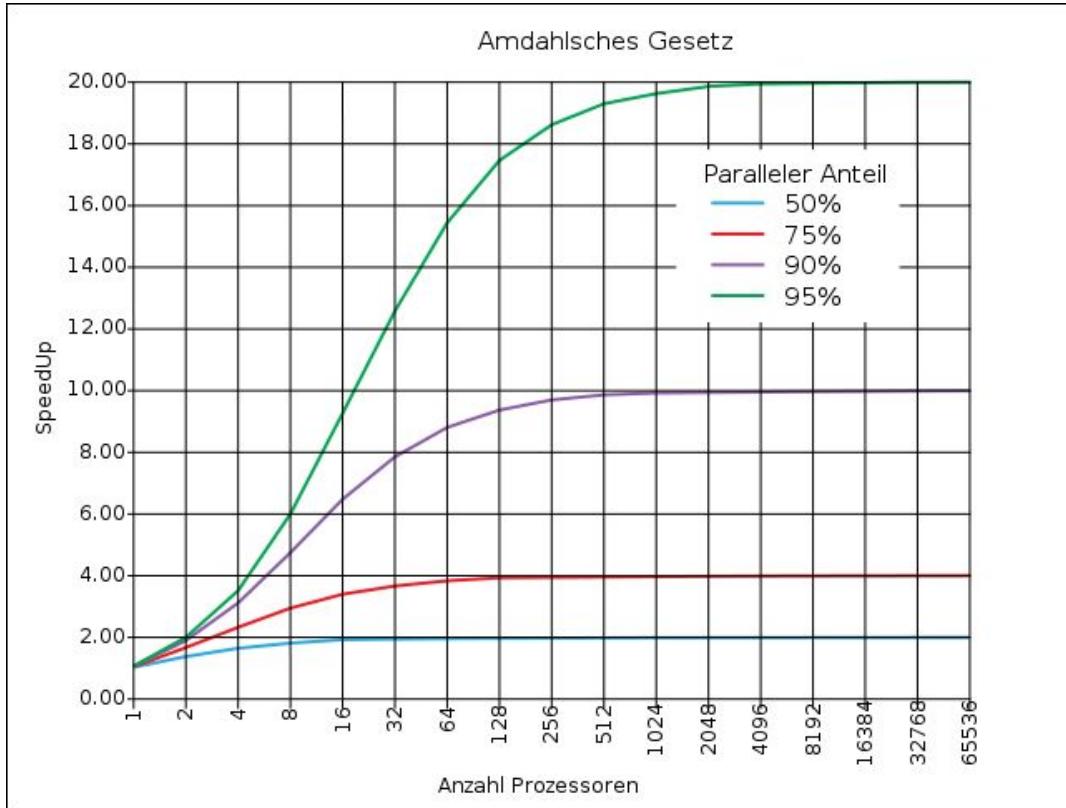
- Datentyp-spezifisch (z.B. Dictionaries)
- Allgemein (z.B. Snappy)
- + ggF. Spalten-Index

# Programmiermodelle

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



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



P = Paralleler Anteil

S = Sequenzieller Anteil

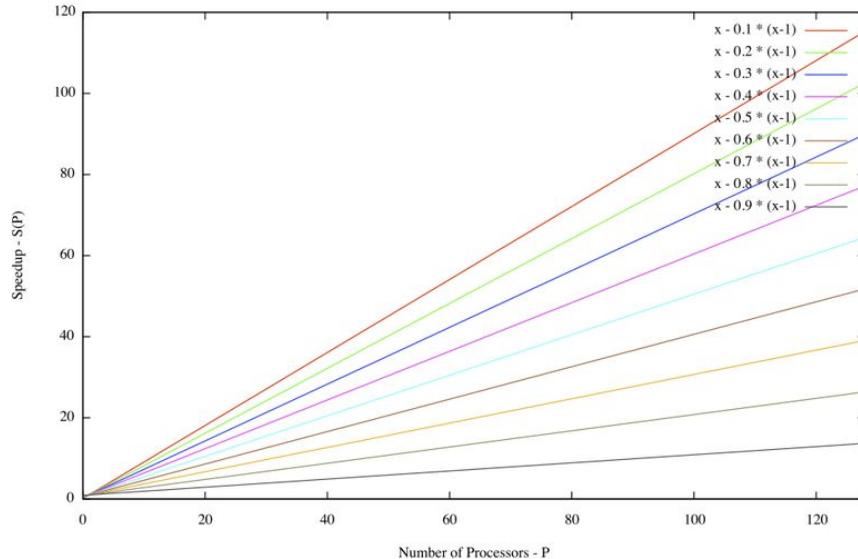
N = Anzahl der Prozessoren

Speedup = Maximale Beschleunigung

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

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

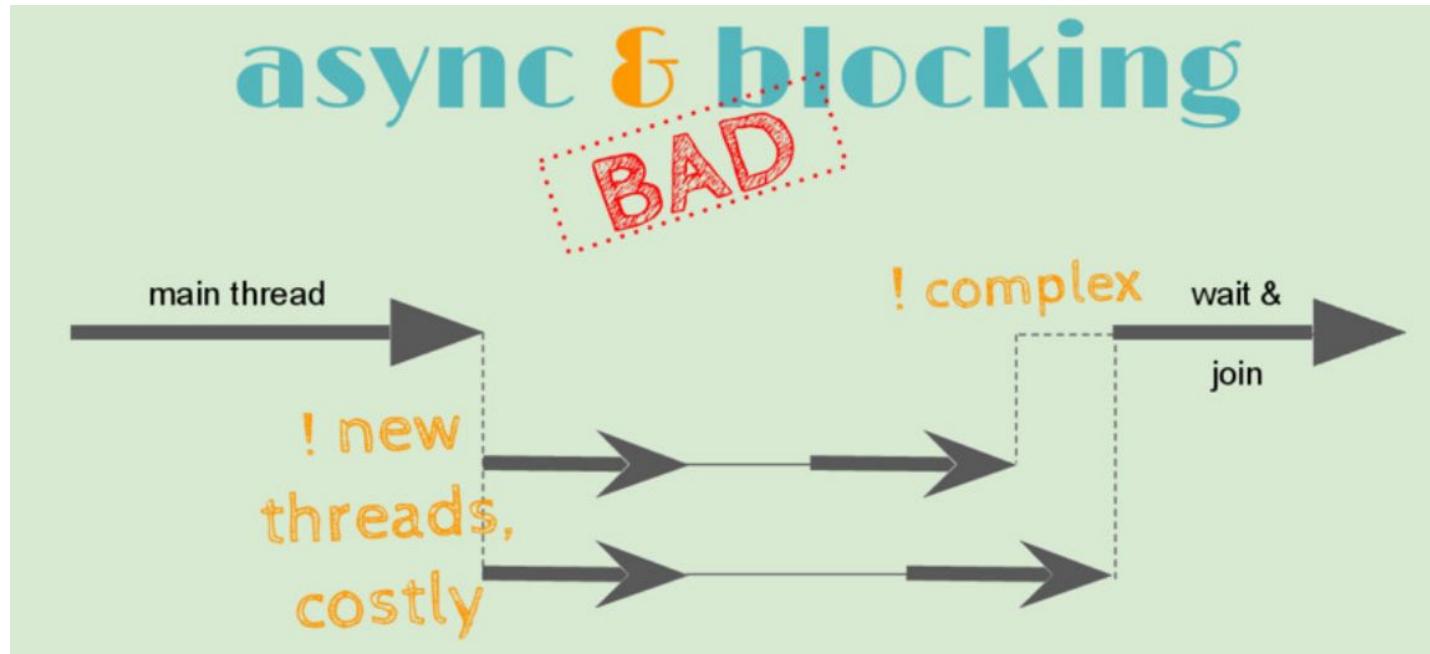
# Gustafsons Gesetz: Ist bei großen Datenmengen jedoch oft passender



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

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

# Parallele Programmierung mit Threads als Parallelisierungseinheit



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

## Dekomposition in Funktionen (auch Aktoren)

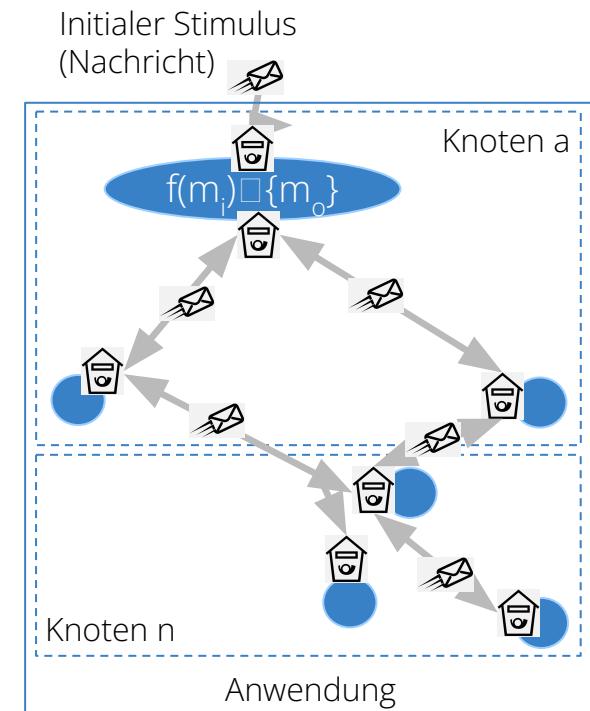
- funktionale Bausteine ohne gemeinsamen Zustand. Jede Funktion ändert nur ihren eigenen Zustand.
- mit wieder aufsetzbarer / idempotenter Logik und abgetrennter Fehlerbehandlung (Supervisor)

## Kommunikation zwischen den Funktionen über Nachrichten

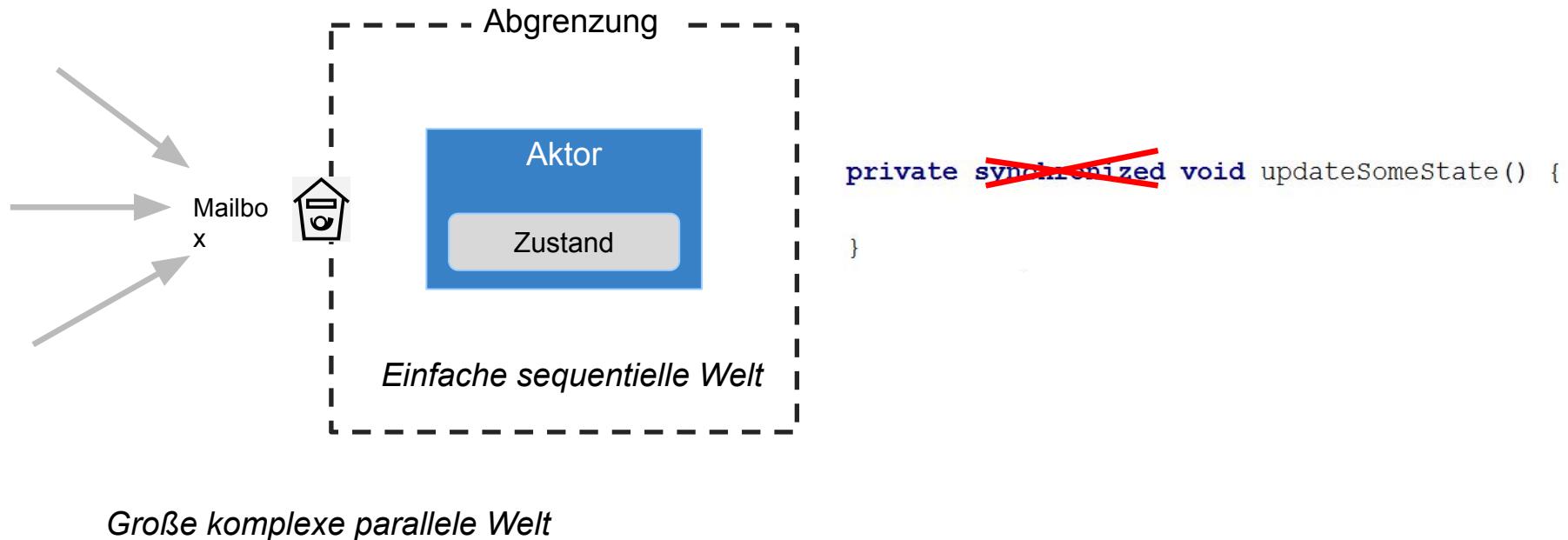
- asynchron und nicht blockierend. Ein Funktion reagiert auf eine Antwort, wartet aber nicht auf sie.
- Mailboxen vor jeder Funktion puffern Nachrichten (Queue mit n Produzenten und 1 Consumer)
- Nachrichten sind das einzige Synchronisationsmittel / Mittel zum Austausch von Zustandsinformationen und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

## Elastischer Kommunikationskanal

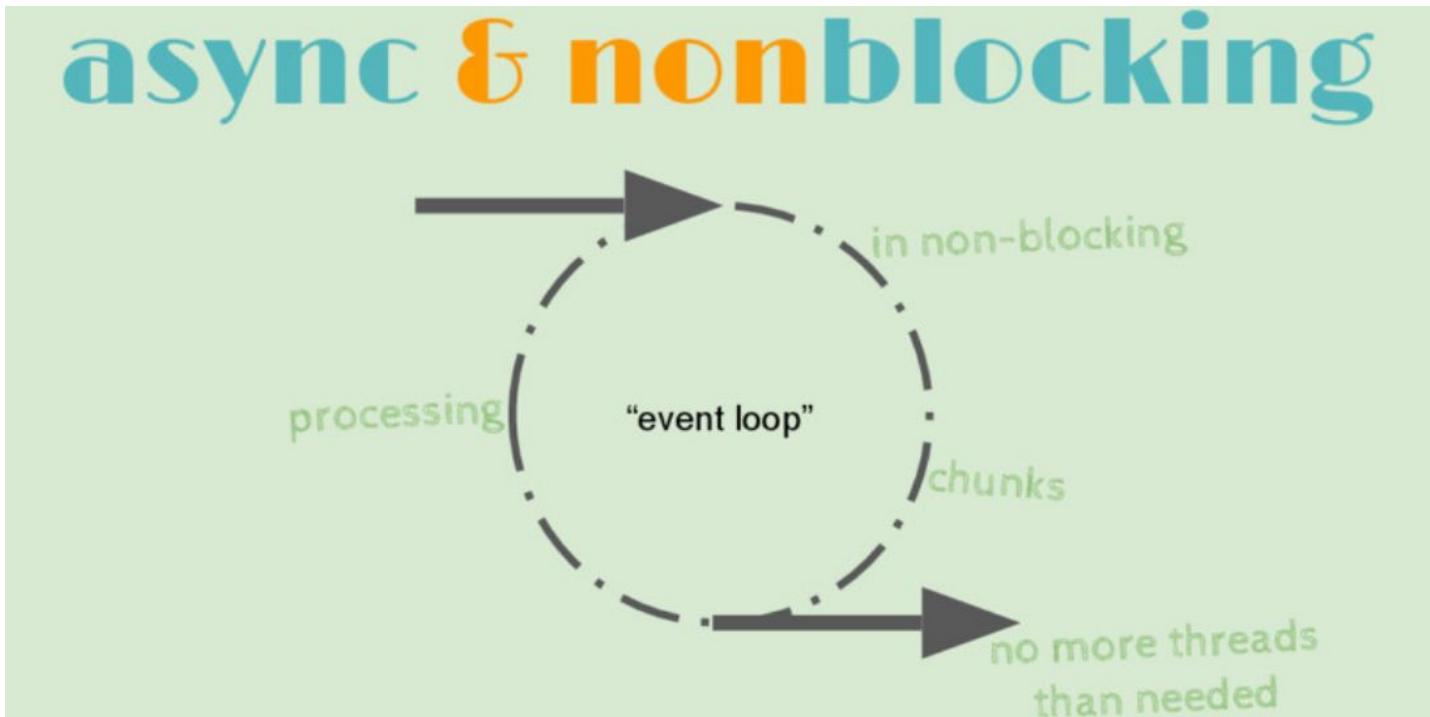
- Effizient: Kanalportabilität (lokal, remote) und geringer Kanal-Overhead
- Load Balancing möglich
- Nachrichten werden (mehr oder minder) zuverlässig zugestellt
- Circuit-Breaker-Logik am Ausgangspunkt (Fail Fast & Reject)



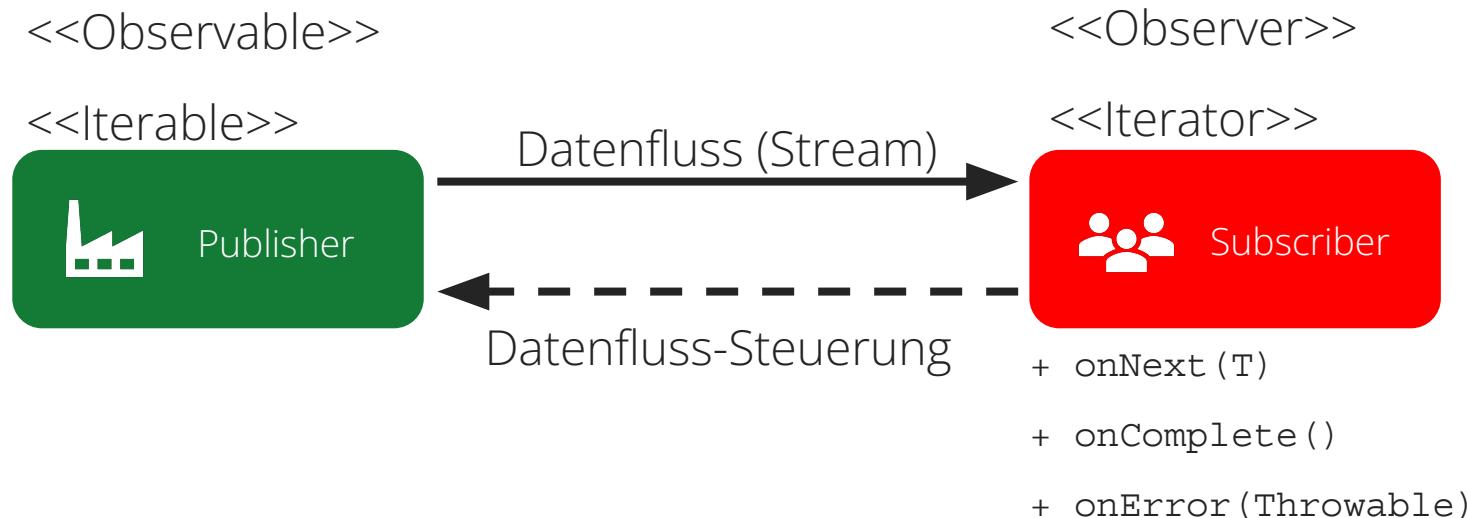
Ein einzelner Aktor ist ein einfaches single-threaded Objekt, das über die Mailbox synchronisiert wird.



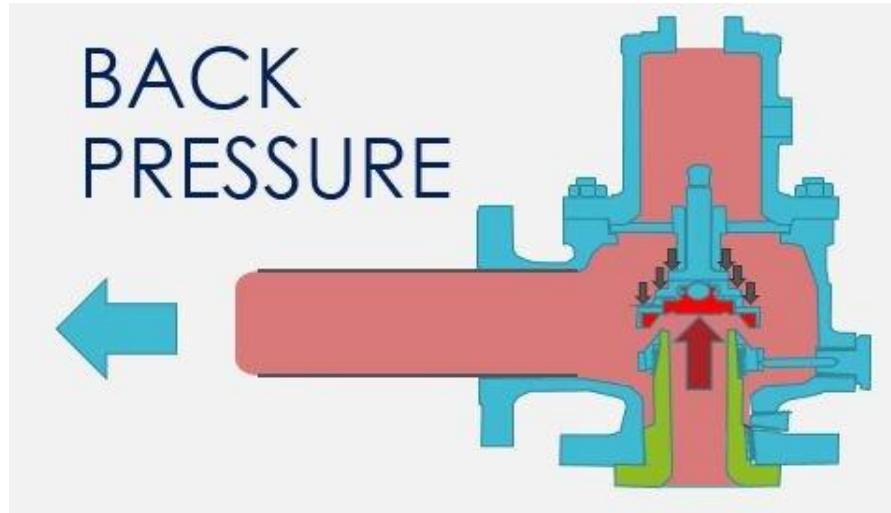
Nachrichten werden über eine Event Loop (aka Scheduler) zugestellt.

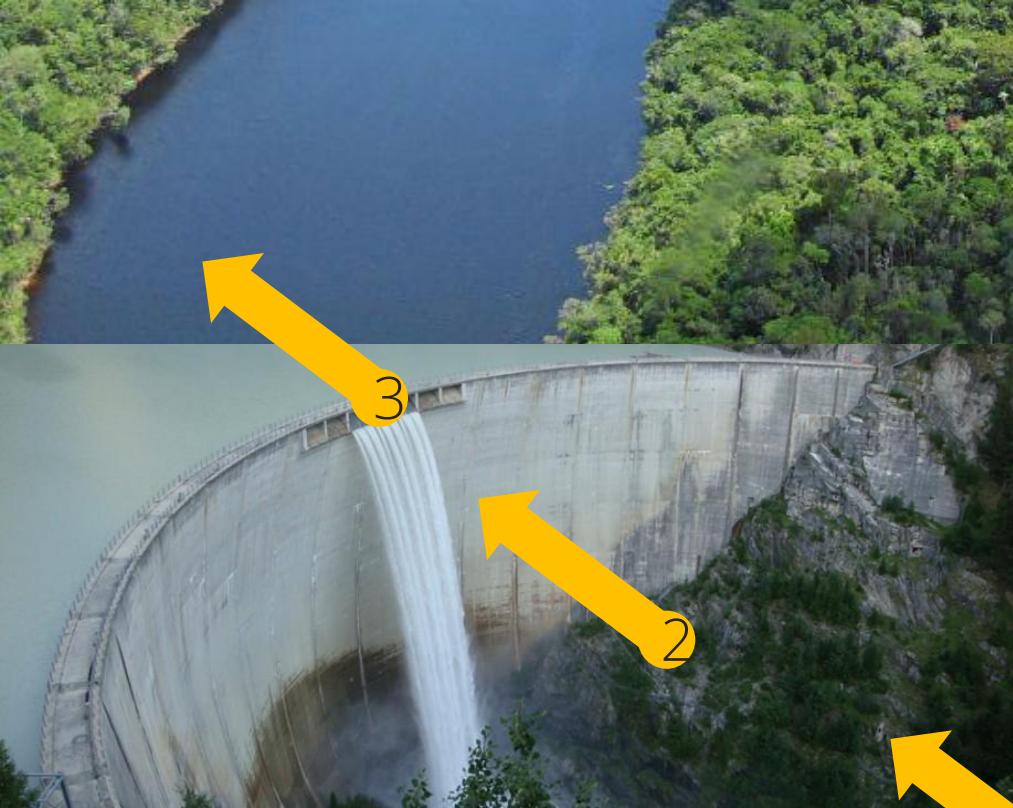


Reactive Streams: Das Fundament ist eine Kombination aus Observer- und Iterator-Pattern.



# Back Pressure: Umgang mit Überlast

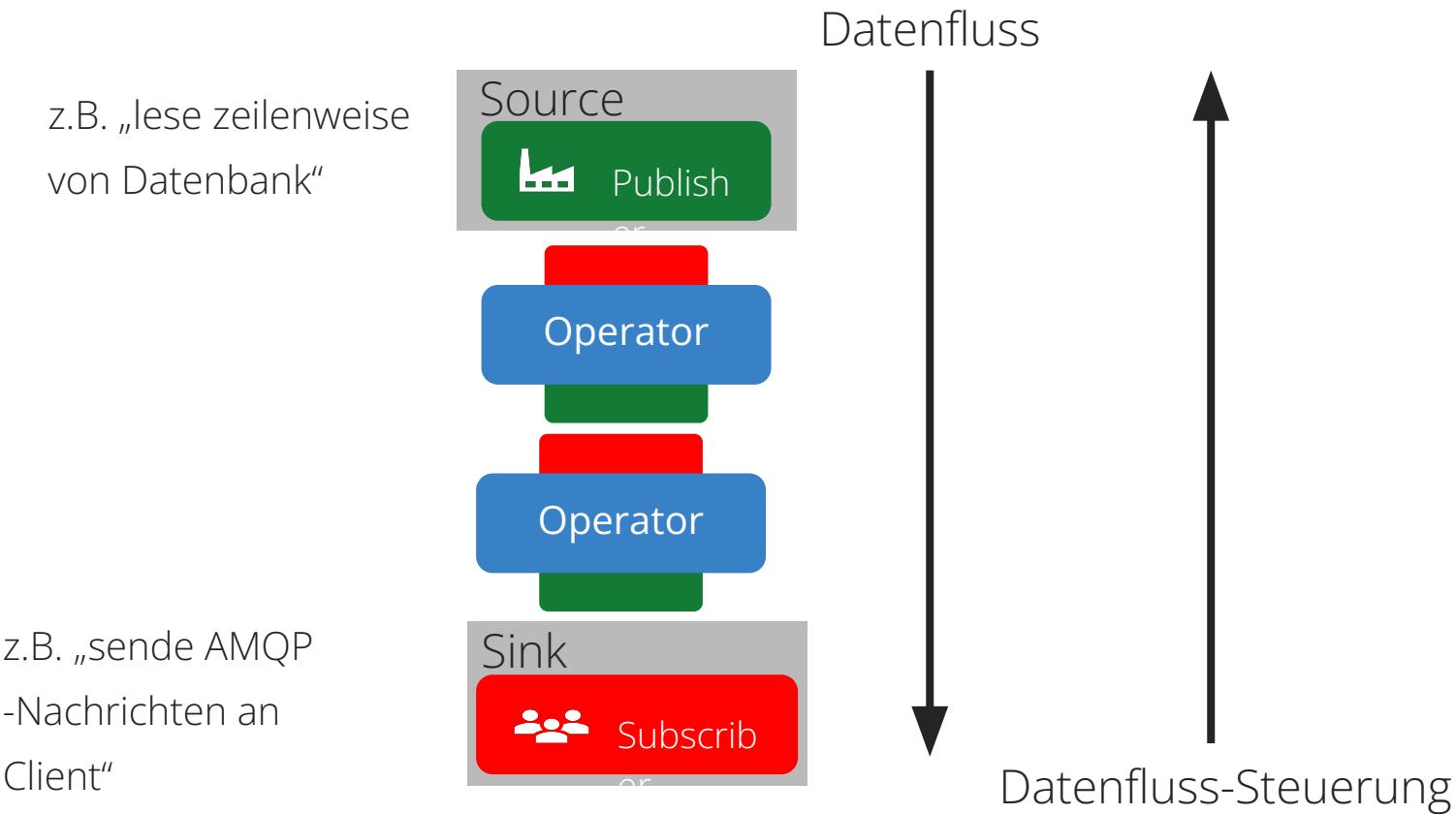




- 3 Staut in einem großen Becken auf
- 2 Reguliert die Schleuse entsprechend, dass der aktuelle max. Durchfluss nie überschritten wird
- 1 Meldet, wie viel max. Durchfluss aktuell möglich ist



# Reactive Streams: Das Programmiermodell



# 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 Programmiermodell
- Over-engineered für Anwendungen mit normaler Last und moderatem Durchsatz
- Es können nur reaktive Libraries verwendet werden (oder Aufwand durch Thread-Pools)

# Kapitel Continuous Delivery

# Abgrenzung zu Continuous X

## Continuous Integration (CI)

- Alle Änderungen werden sofort in den aktuellen Entwicklungsstand integriert und getestet.
- Dadurch wird kontinuierlich getestet, ob eine Änderung kompatibel mit anderen Änderungen ist.

## Continuous Delivery (CD)

- Der Code *kann* zu jeder Zeit deployed werden.
- Er muss aber nicht immer deployed werden.
- D.h. der Code muss (möglichst) zu jedem Zeitpunkt bauen, getestet und ge-debugged sein.

## Continuous Deployment

- Jede stabile Änderung wird in Produktion deployed.
- Ein Teil der Qualitätstests finden dadurch in Produktion statt.
  - Die Möglichkeit mit Fehlern umzugehen muss vorhanden sein (z.B. Canary Release, siehe später)