

Kapitel 13: Zusammenfassung

vorlesung

CLOUD
COMPUTING

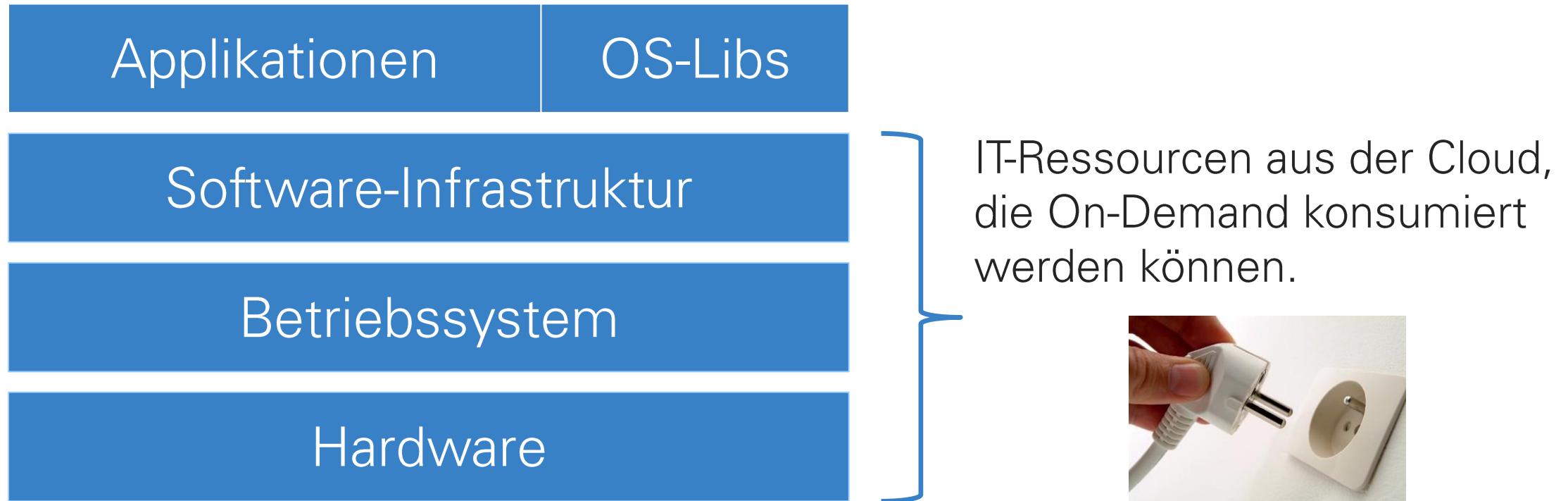
Organisatorisches

Prüfung

- Die Prüfung findet am 21.01.2020 um 15.30 statt.
- Sie ist schriftlich, dauert 90min und es sind keine Unterlagen dafür zugelassen.
- Der Raum der Prüfung ist A3.10.

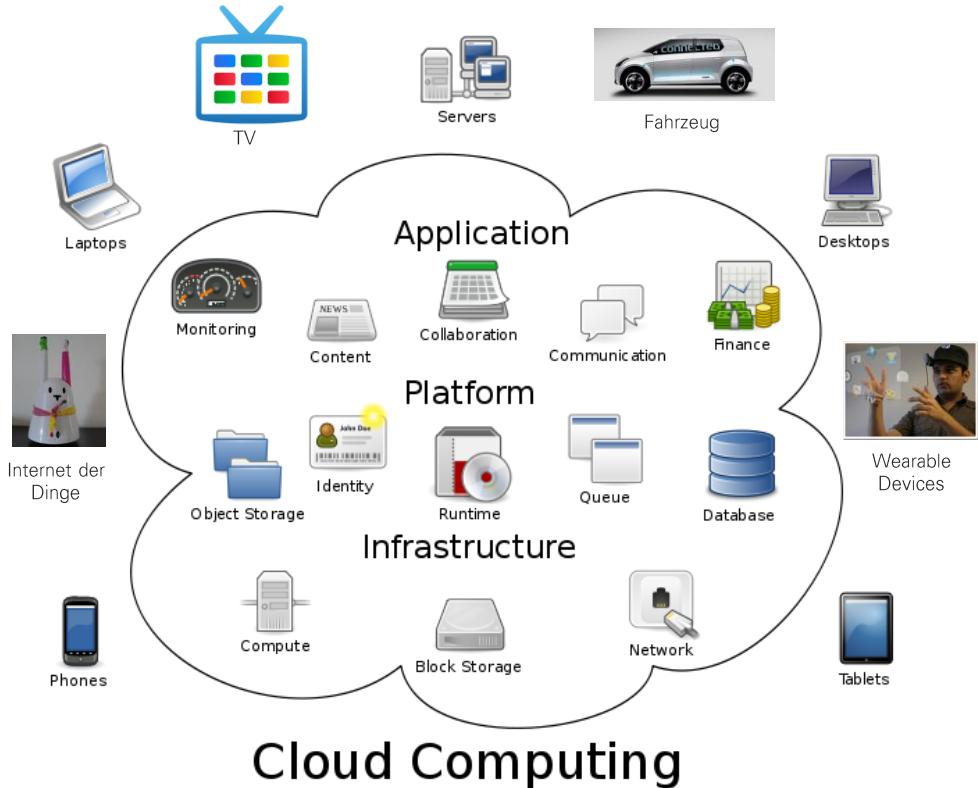
Kapitel 0: Einführung und Grundlagen

Beim Cloud Computing geht es im Kern um eine geringere Verbaustiefe 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://jelastic.com/de/>



NY Times

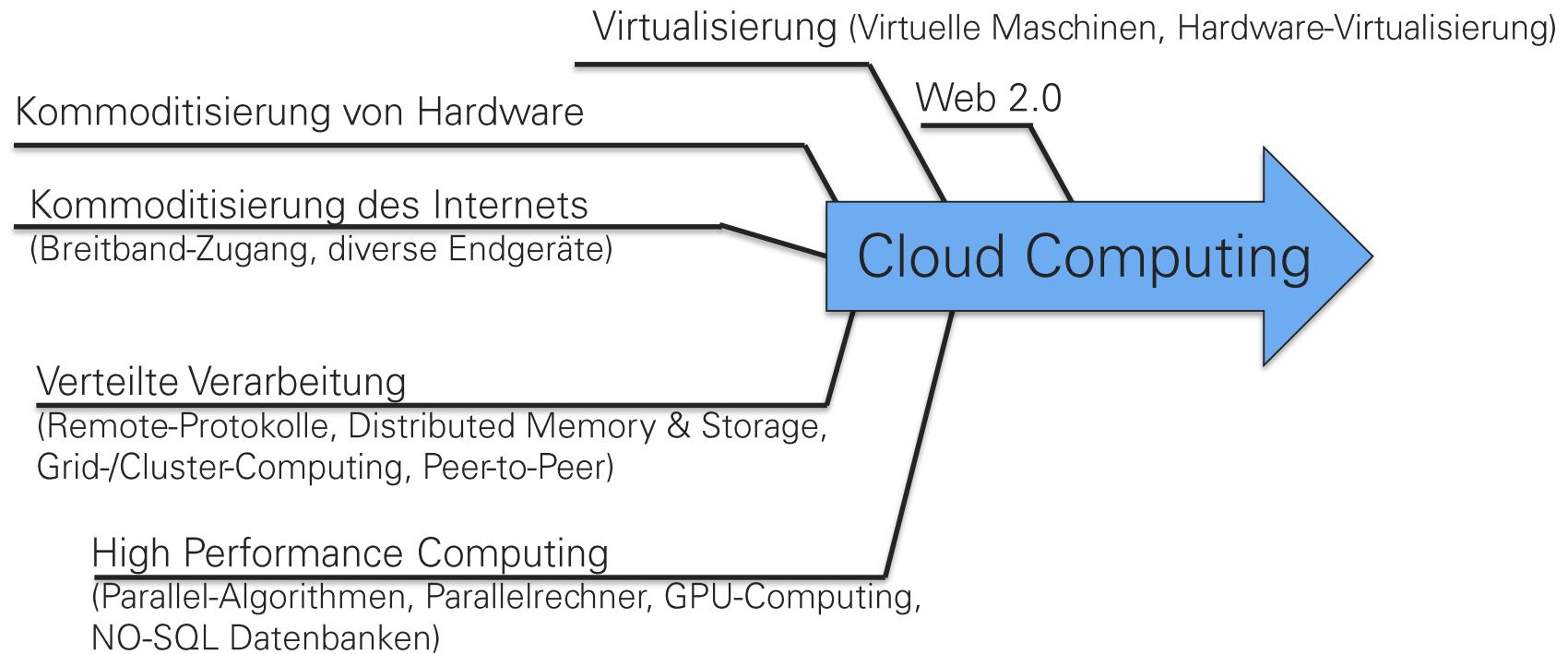
- Needed offline conversion of public domain articles from 1851-1922.
- Used Hadoop to convert scanned images to PDF
- Ran 100 Amazon EC2 instances for around 24 hours
- 4 TB of input
- 1.5 TB of output

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.

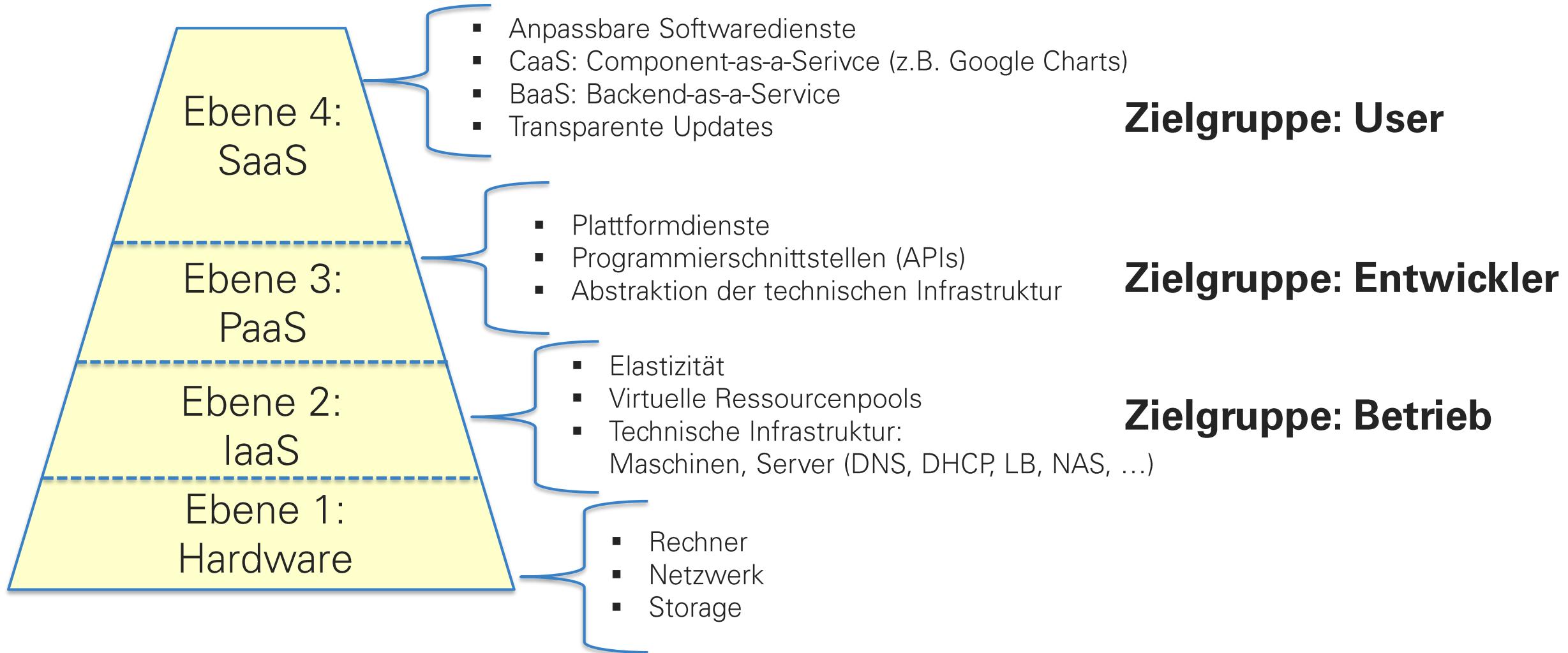
Published 1892, copyright New York Times

[<< http://www.slideshare.net/acarlos1000/hadoop-basics-presentation](http://www.slideshare.net/acarlos1000/hadoop-basics-presentation)

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

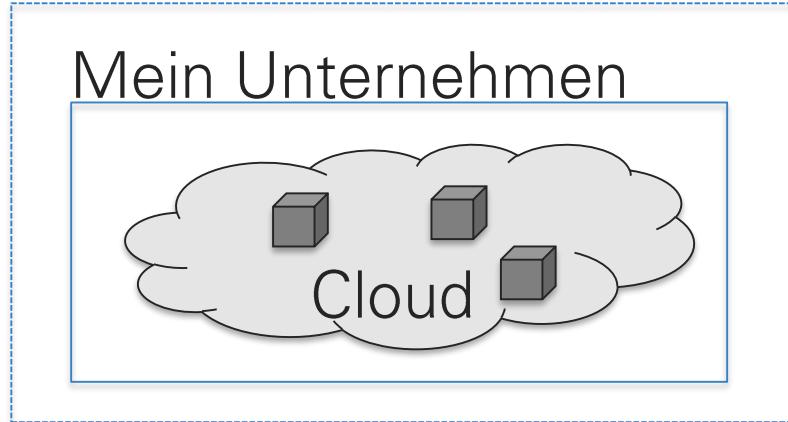


Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.

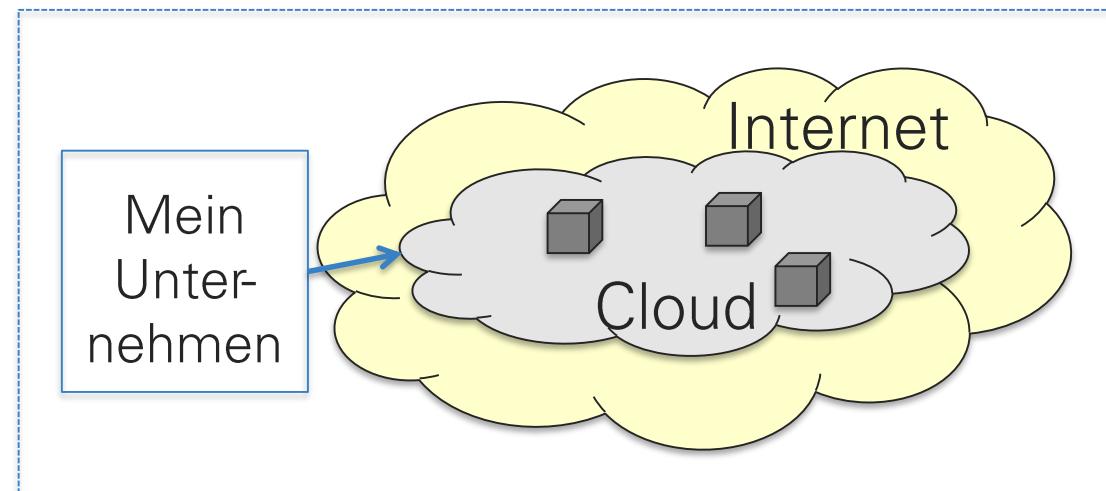


Öffentliche und private Wolken.

Private Cloud:

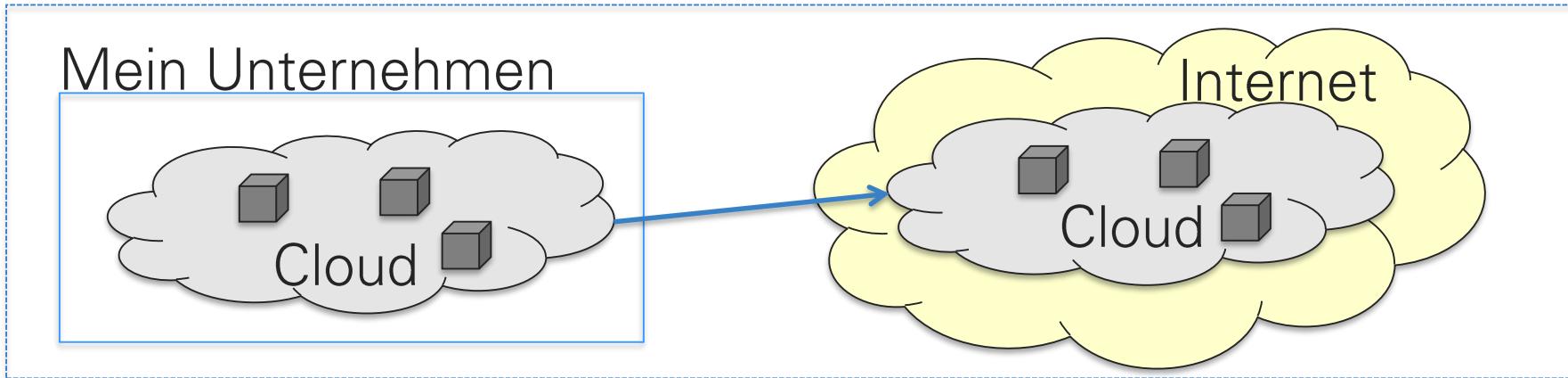


Public Cloud:

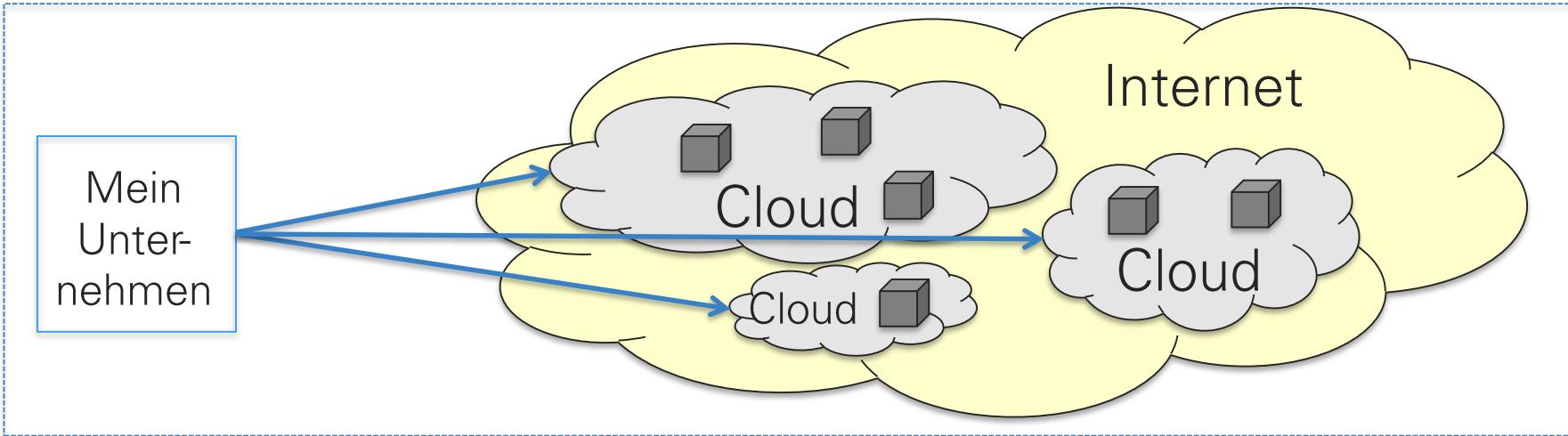


Hybride und multiple Wolken.

Hybrid Cloud:

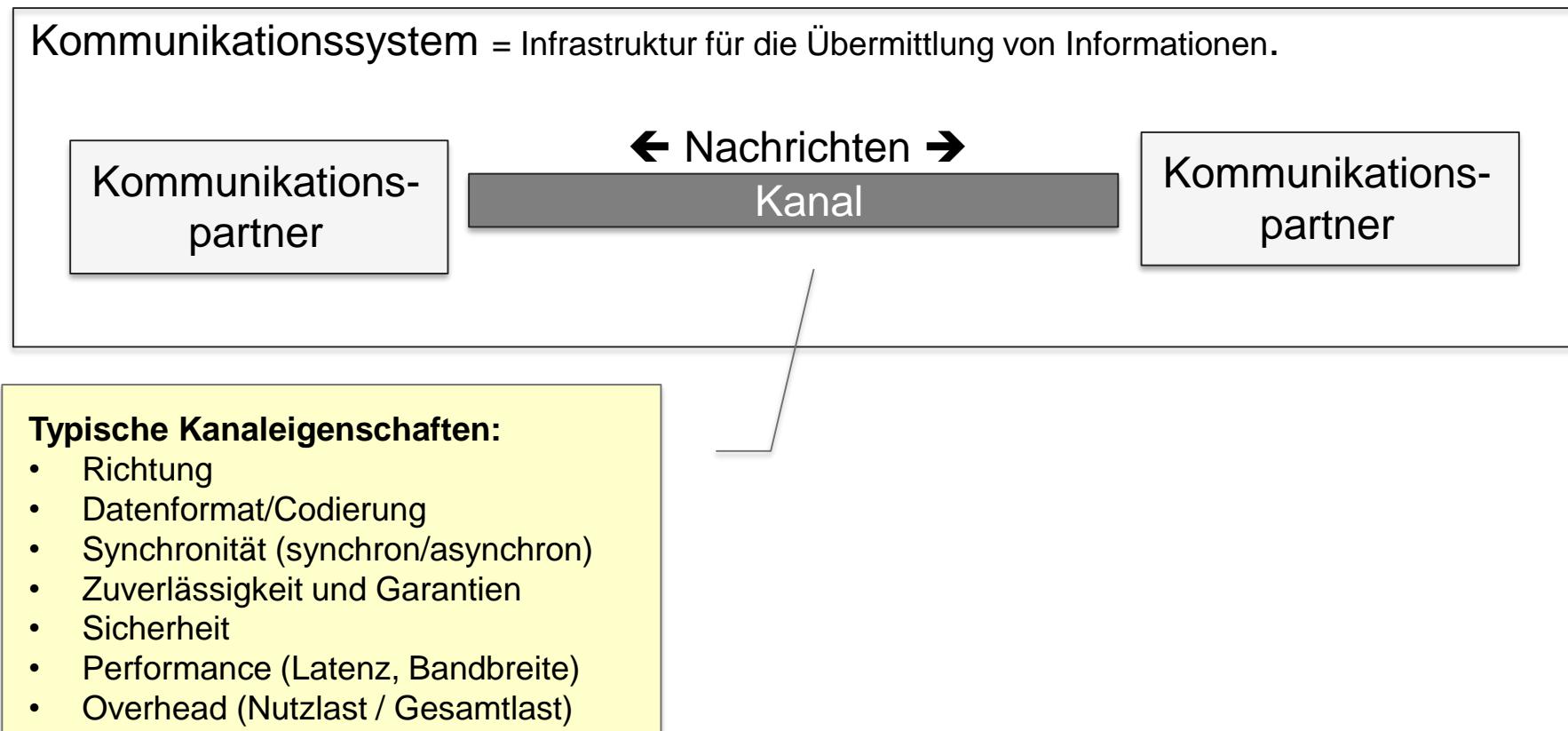


Multi-Cloud:

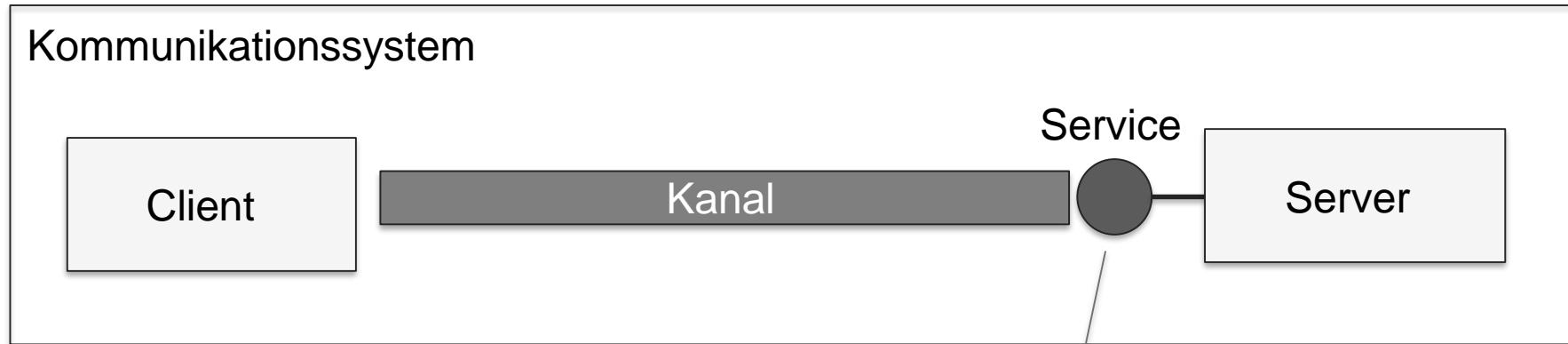


Kapitel 1: Kommunikation

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



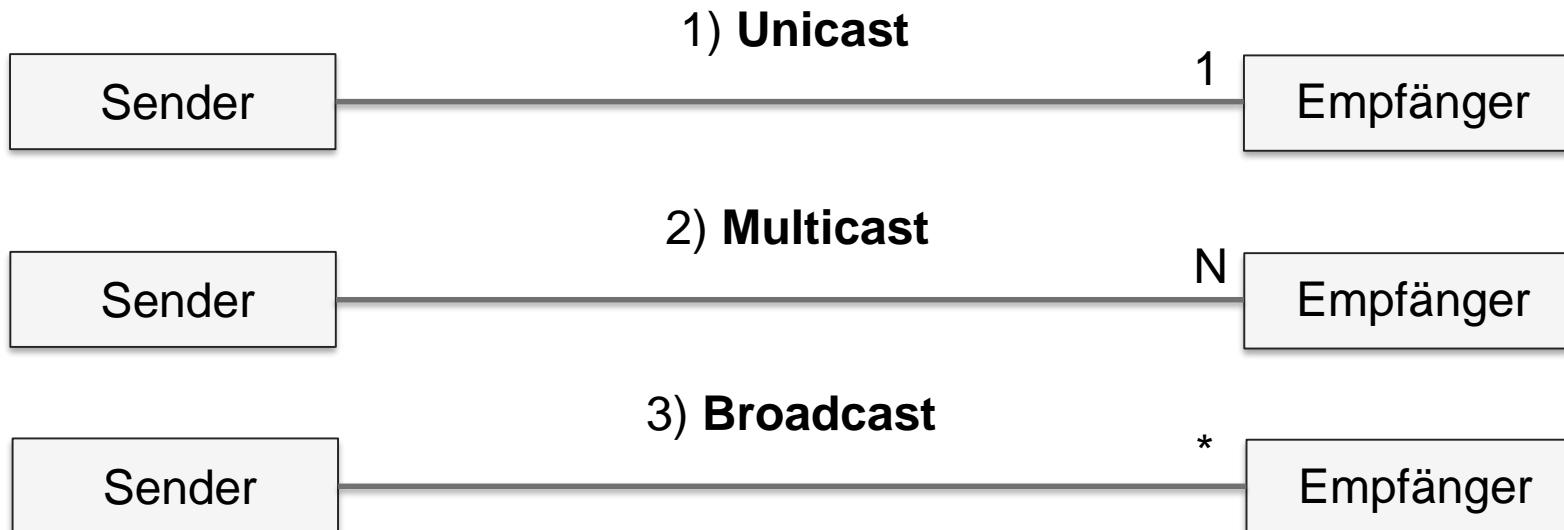
Service-Orientierung in einem Kommunikationssystem: Client-Server-Kommunikation über Services



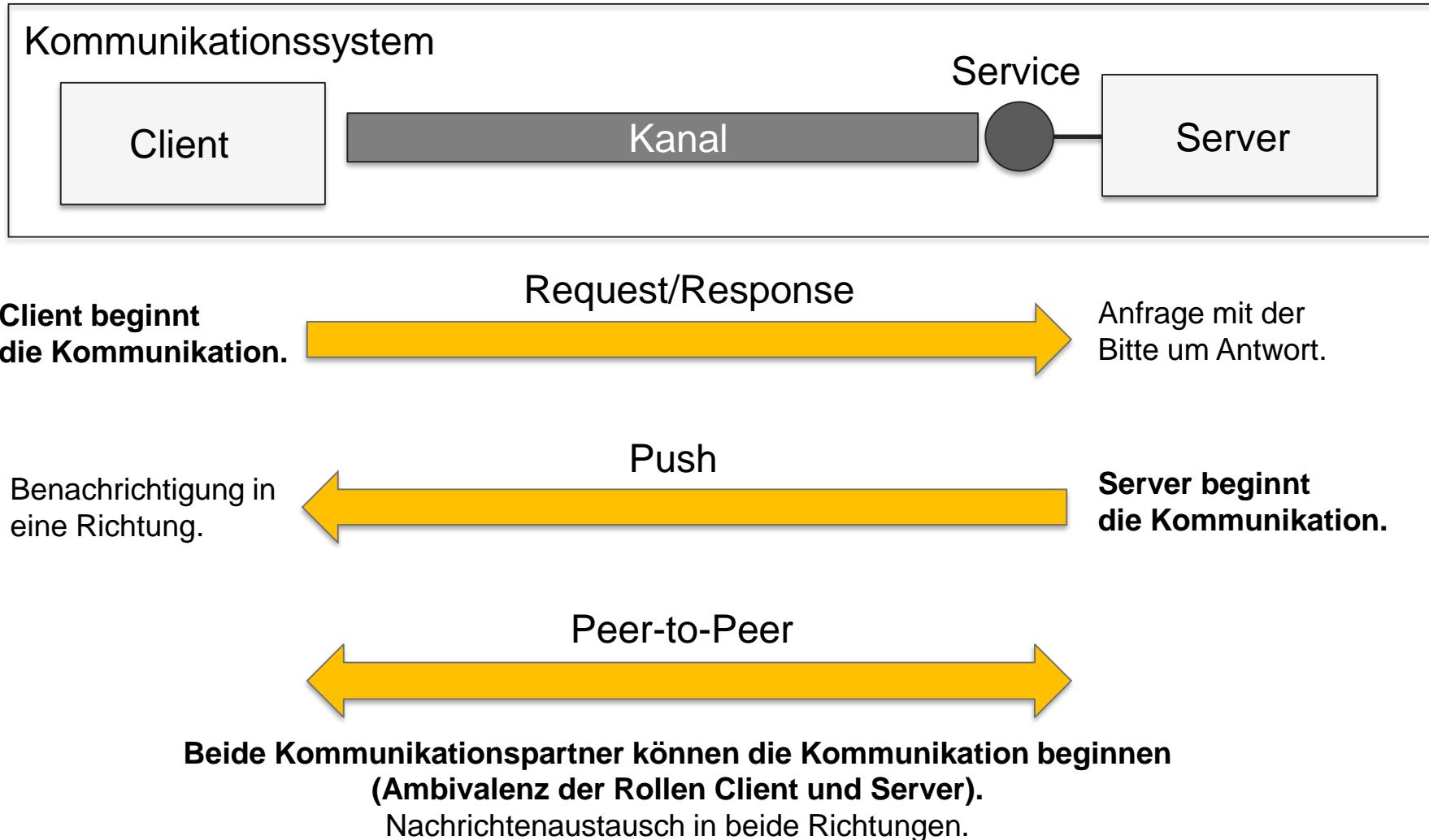
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

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**.

Klassifikation von Kommunikationssystemen: Kardinalität der Empfänger einer Nachricht.



Klassifikation von Kommunikationssystemen: 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 (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 (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.

Beispiele für REST-Aufrufsyntax: Schnittstellenentwurf über Substantive.

- Produkte aus der Kategorie Spielwaren:

<http://www.example.com/produkte/spielwaren>

- Bestellungen aus dem Jahr 2008

<http://www.example.com/bestellungen/2008>

- Liste aller Regionen, in denen der Umsatz größer als 5 Mio. Euro

[http://www.example.com/regionen/umsatz/summe?groesserAls=5
M](http://www.example.com/regionen/umsatz/summe?groesserAls=5M)

- Gib mir die zweite Seite aus dem Produktkatalog

<http://www.example.com/produkte/2>

- Alle Gruppen, in den der Benutzer „josef.adersberger“ Mitglied ist.

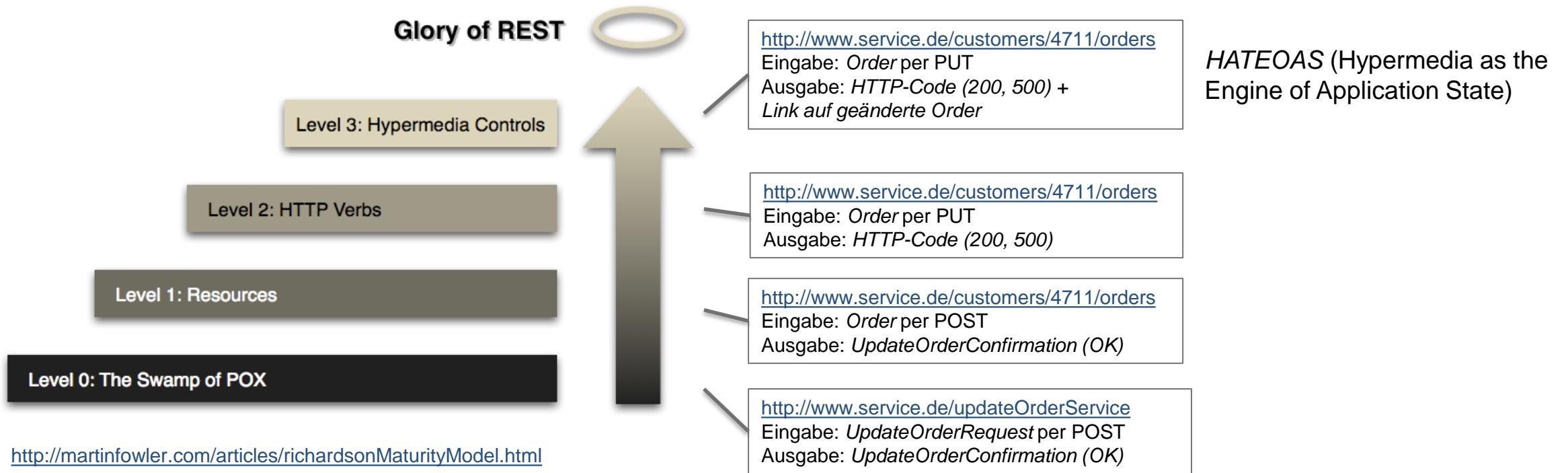
<http://www.example.com/benutzer/josef.adersberger/gruppen>

Gängige Entwurfsregeln:

- Plural, wenn auf Menge an Entitäten referenziert werden soll. Sonst singular.
- Pfad-Parameter, wenn Reihenfolge der Angabe wichtig. Sonst Query Parameter.
- Standard Query Parameter einführen (z.B. für Filter und Abfragen sowie seitenweisen Zugriff) und konsistent halten.
- Pfad-Abstieg, wenn Entitäten per Aggregation oder Komposition verbunden sind.
- Pfad-Abstieg, wenn es sich um einen gängigen Navigationsweg handelt.
- Ids als Pfad-Parameter abbilden.
- Fehler und Ausnahmen über Return Codes abbilden. Einen Standard-Code suchen, der von der Semantik her passt.

Siehe auch: <https://thomashunter.name/posts/2013-12-31-codeplanet-principles-of-good-restful-api-design>

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

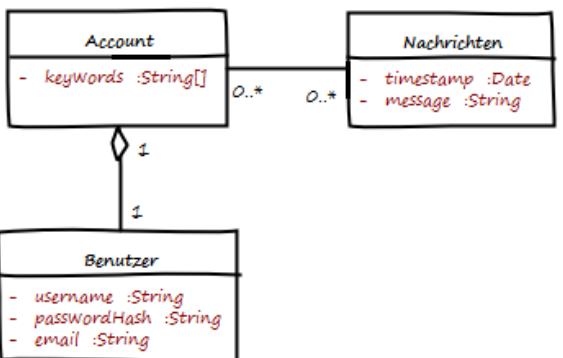


Entwicklung von REST APIs

Anwendungsfälle erheben

- Benutzer authentifizieren
- Nachricht einstellen
- Meine Nachricht löschen
- Meine Nachricht ändern
- Liste aller Nachrichten anzeigen
- Liste meiner Nachrichten anzeigen
- Liste der Nachrichten mit best. Text anzeigen
- Nutzerstatistik anzeigen
- Account erstellen (inkl. Benutzerinfos anlegen)
- Account ändern
- Account löschen

Entitätenmodell erstellen



REST Schnittstelle umsetzen

Var. 1

Top-down Ansatz:

REST Schnittstelle definieren

REST Schnittstelle generieren

REST Schnittstelle implementieren

Var. 2

Bottom-up Ansatz:

REST Schnittstelle implementieren

Definition REST-Schnittstelle generieren

Die effizienten Alternativen: Binärprotokolle

Binärprotokolle sind eine sinnvolle Alternative zu REST, wenn eine effiziente und programmiersprachennahe Kommunikation erfolgen soll.

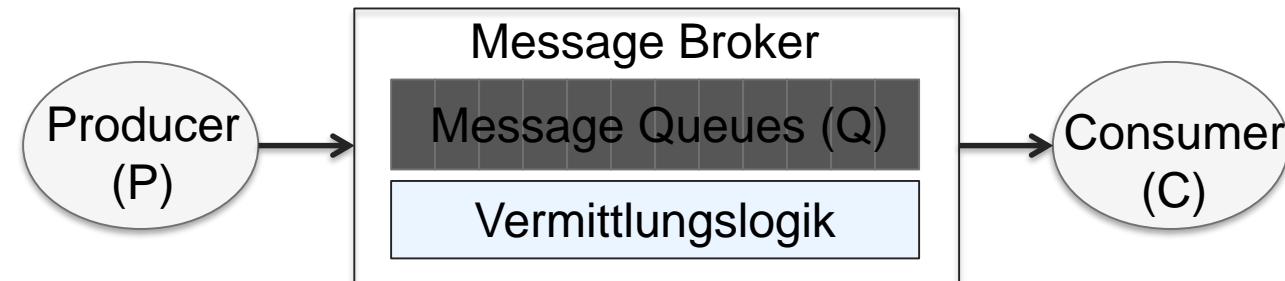
- Encoding der Payload als komprimiertes Binärformat
- Separate Schnittstellenbeschreibungen (IDLs, *Interface Definition Languages*) aus denen dann Client- und Server-Code in mehreren Programmiersprachen generiert werden können

Kandidaten

- gRPC / Protocol Buffers
- Apache Avro
- Apache Thrift
- Hessian

Binärprotokolle können auch mit REST kombiniert werden: Als Content-Type und damit als Payload wird eine Binär-Codierung verwendet. Beispiel: Protocol Buffers over REST.

Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch.



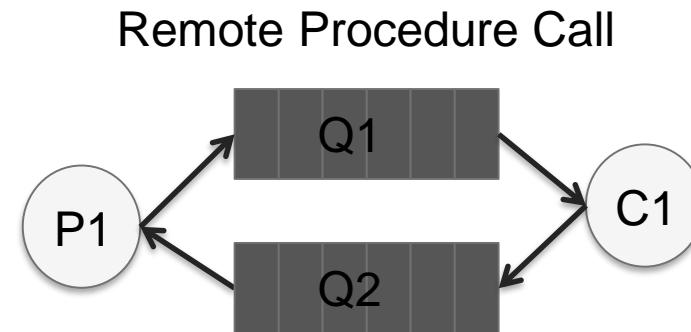
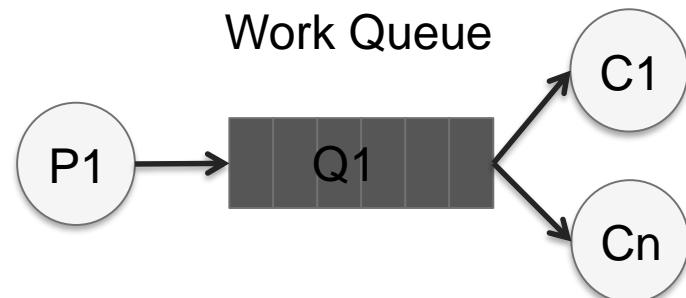
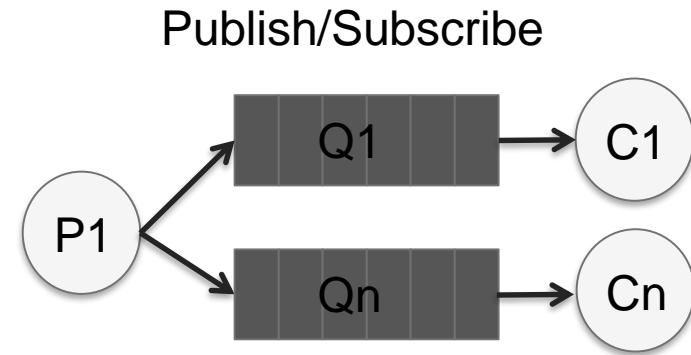
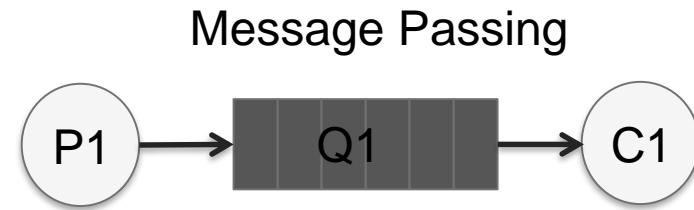
Entkopplung von Producer und Consumer.

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.



Kapitel 2: 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>

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

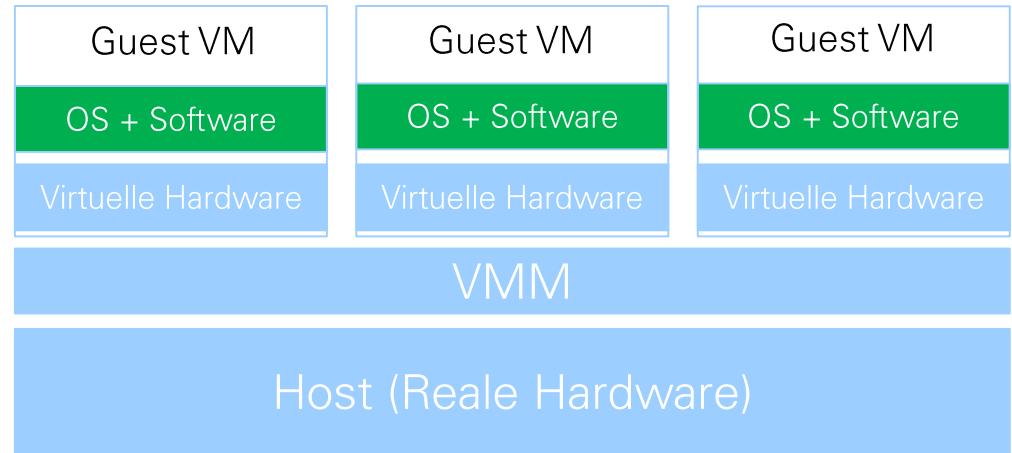
- Virtuelle Netzwerkschnittstellen und virtuelle Netzwerk-Infrastrukturen (VLAN)
- Brücken zwischen virtuellen und realen Netzwerken

■ Storage

- Virtuelle Festplatten-Laufwerke. Abbildung auf Dateien im realen Dateisystem. Volumen entweder vor-allociert 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

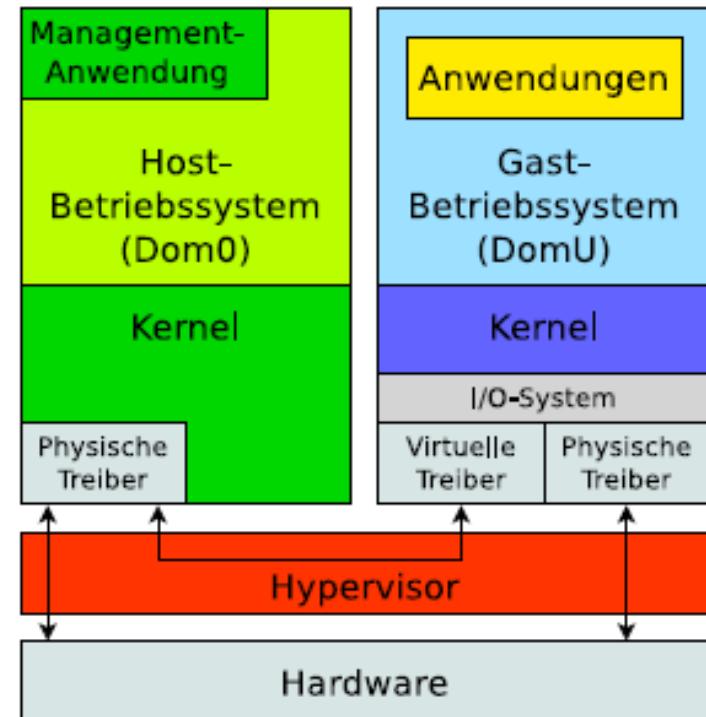
- Eine lauffähige / laufende virtuelle Maschine

VMM (Virtual Machine Monitor)

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

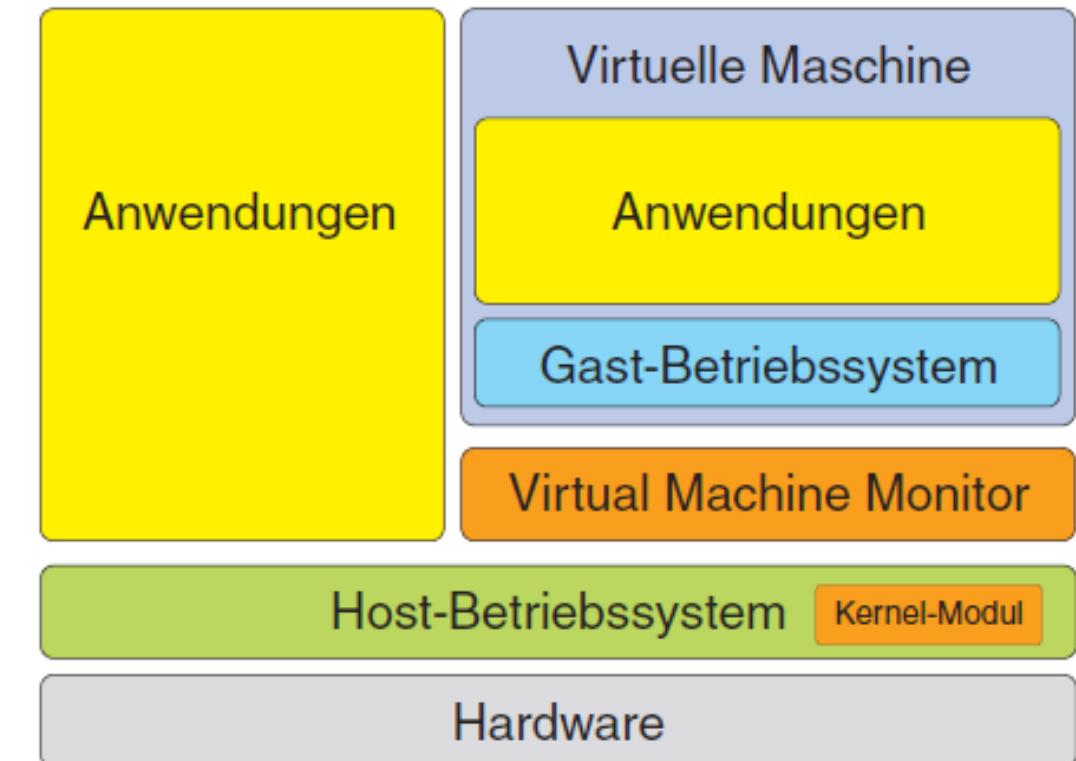
Hardware-Virtualisierung: Para-Virtualisierung

- 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%

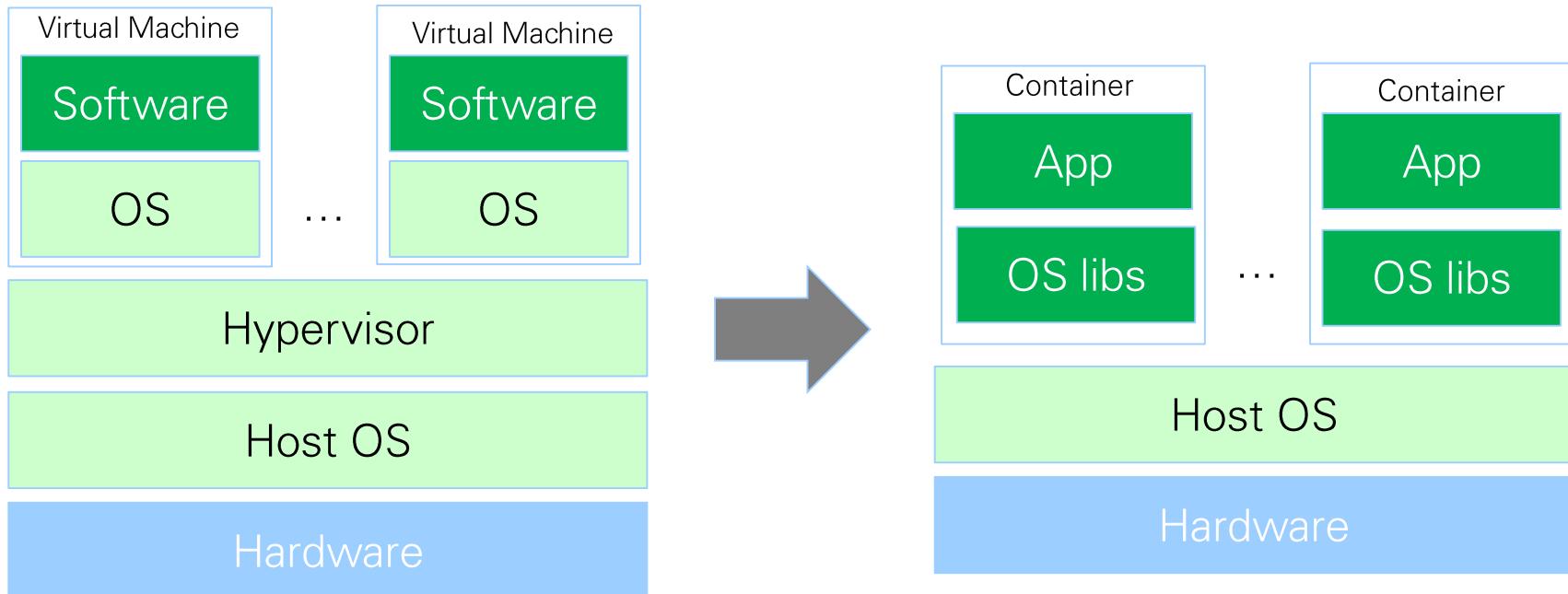


Hardware-Virtualisierung: Voll-Virtualisierung

- 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%.



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

Hardware-Virtualisierung



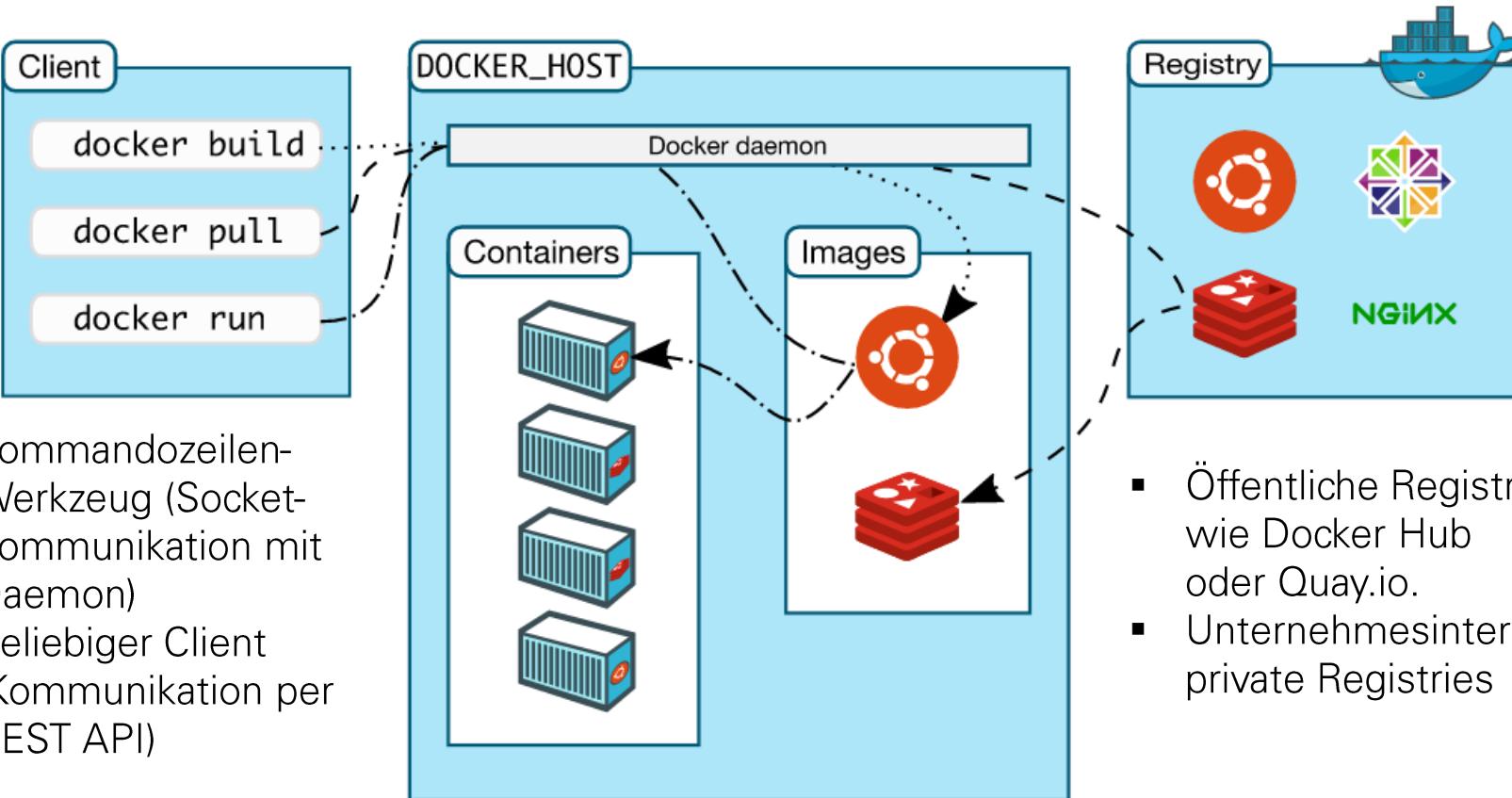
- Stärkere Isolation
- Höhere Sicherheit

Betriebssystem-(OS-)Virtualisierung



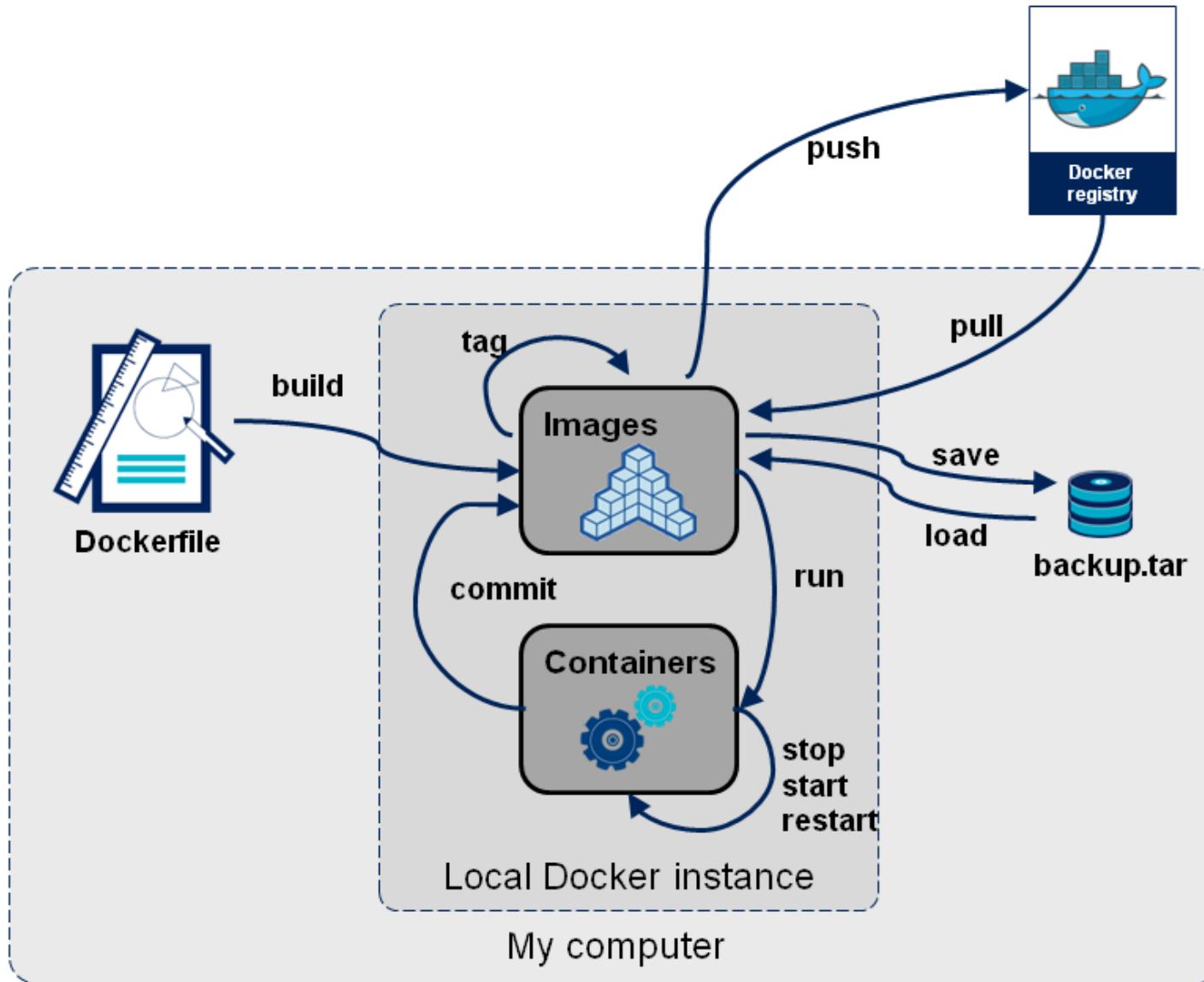
- Geringeres Volumen der privaten Kopie
- Geringerer Overhead
- Kürzere Startup-Zeit

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

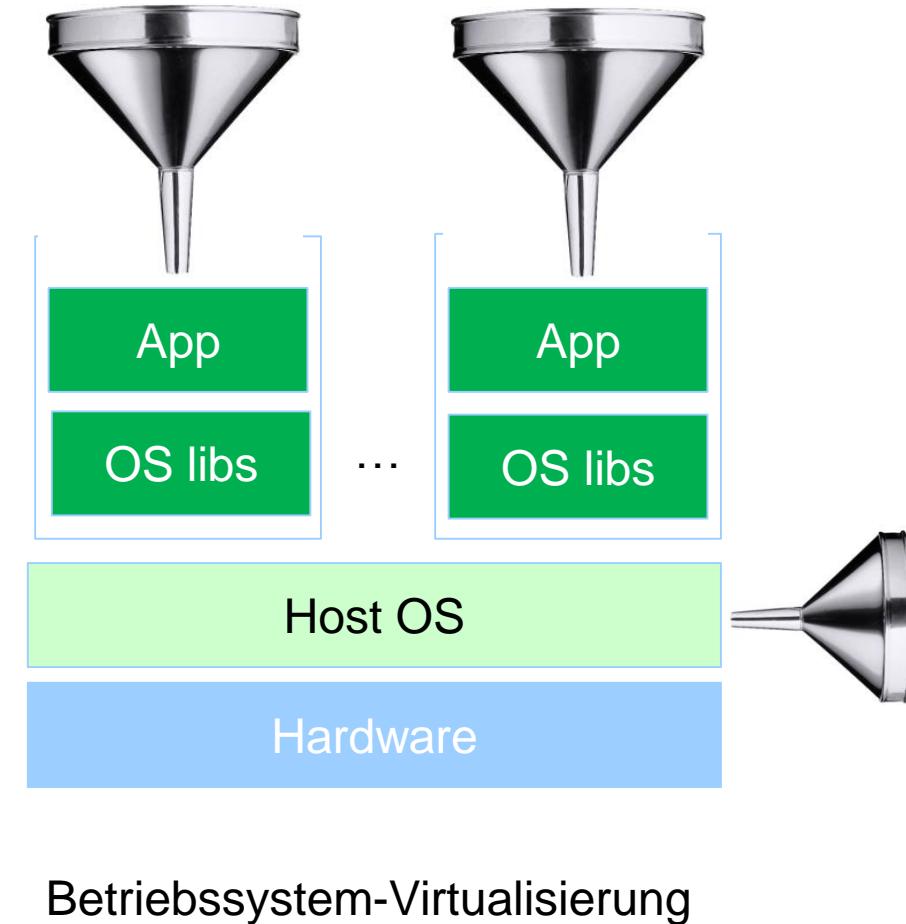
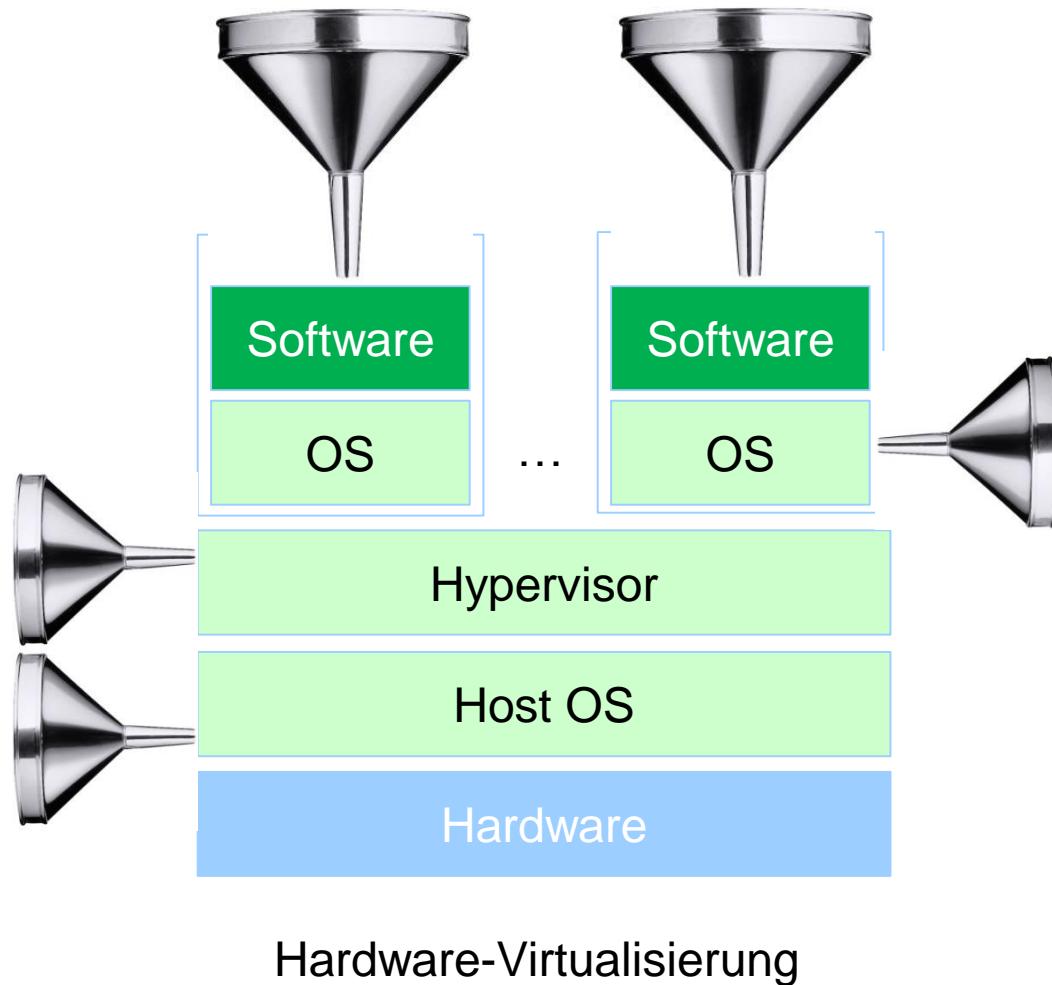
| Command | Action |
|--|---|
| <code>docker build -t <image> .</code> | Build Docker image with given tag in current directory |
| <code>docker images</code> | Prints all local images |
| <code>docker run -d -v <volume mounts> -p <host-port>:<container-port> <image> <entrypoint process></code> | Run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

Hilfreiche Kommandos für Container Troubleshooting

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

Kapitel 3: Provisionierung

Provisionierung: Wie kommt Software in die Boxen?

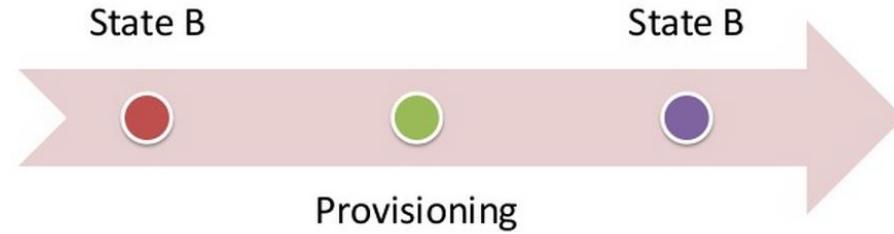


Provisionierung ist die Bezeichnung für die automatisierte Bereitstellung von IT-Ressourcen.
<http://wirtschaftslexikon.gabler.de/Definition/provisionierung.html>

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:

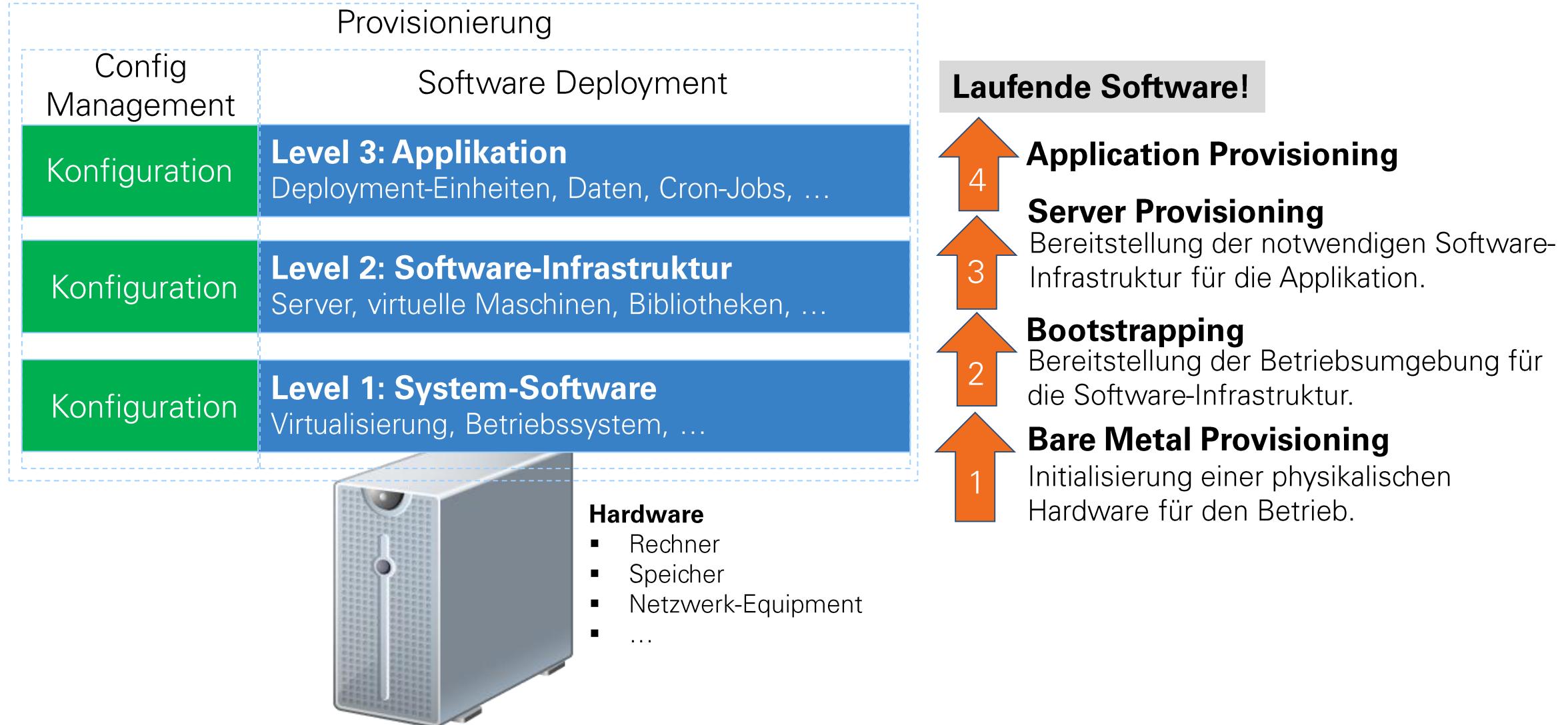
1. Ausgangszustand feststellen
2. Vorbedingungen prüfen
3. Zustandsverändernde Aktionen ermitteln
4. Zustandsverändernde Aktionen durchführen
5. Nachbedingungen prüfen und ggf. Zustand zurücksetzen

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.

Provisierung erfolgt auf drei verschiedenen Ebenen und in vier Stufen.



Die neue Leichtigkeit des Seins.

Old Style



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

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

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 4: Infrastructure-as-a-Service

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.

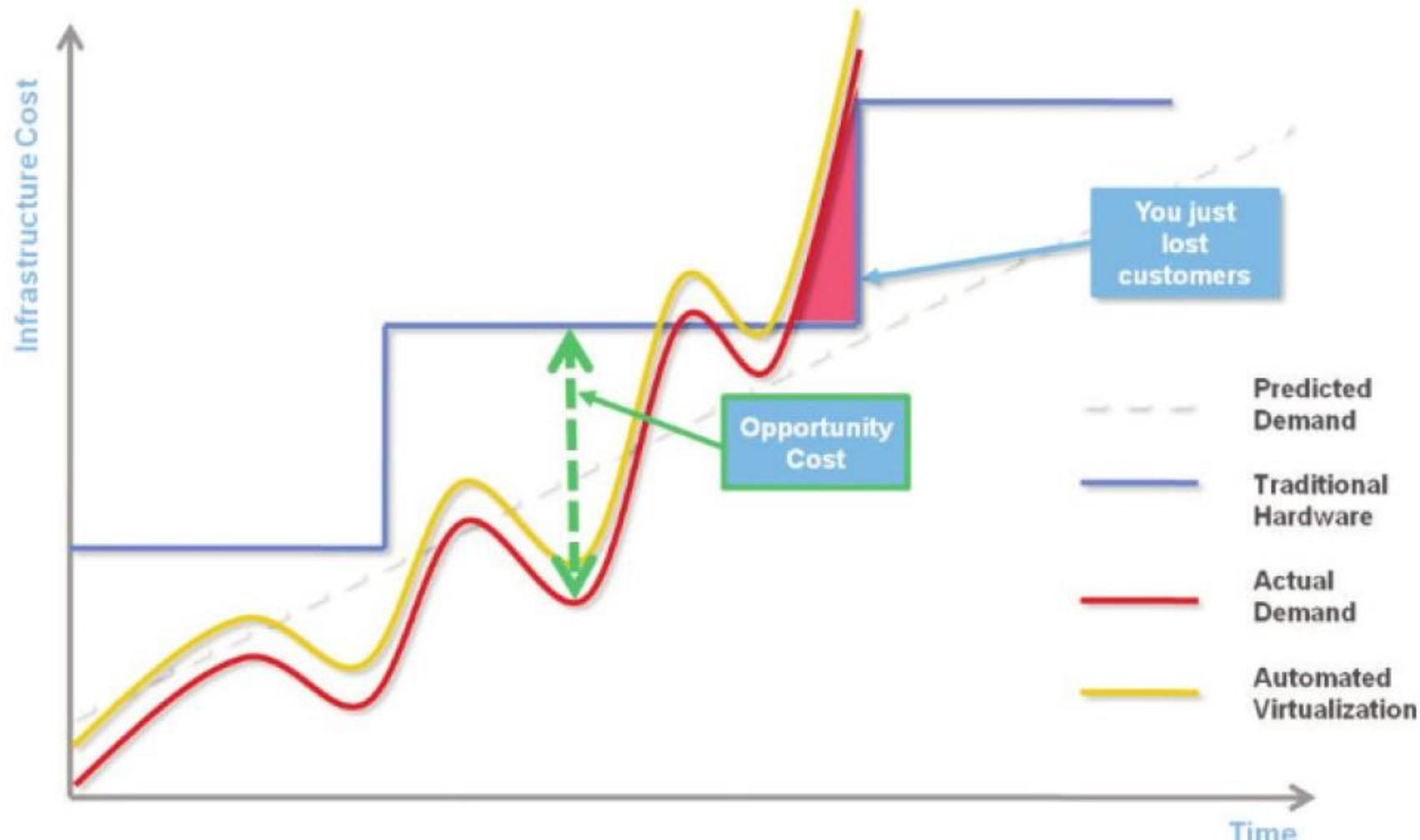
Ressourcen-Typen in einer IaaS-Cloud:

- **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:

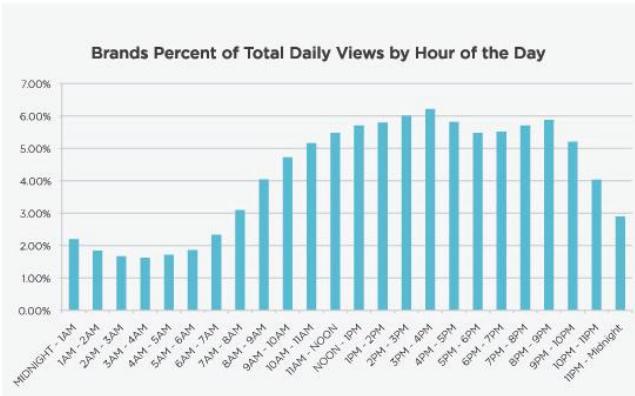
- **Monitoring**
- **Ressourcen-Management**

Klassische Betriebsszenarien werden bei dynamischer Nachfrage teuer. Hohe Opportunitätskosten.

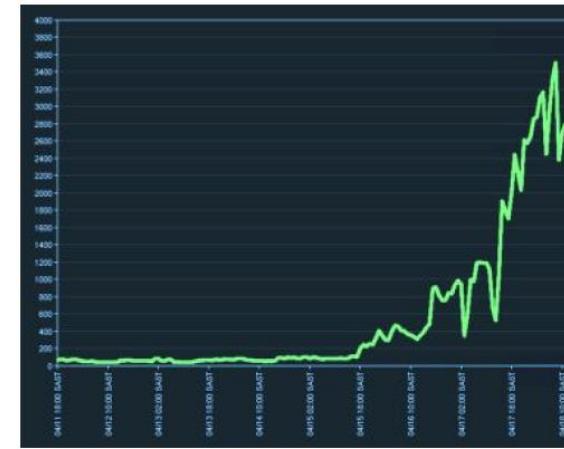


Skalierbarkeit: Effekte

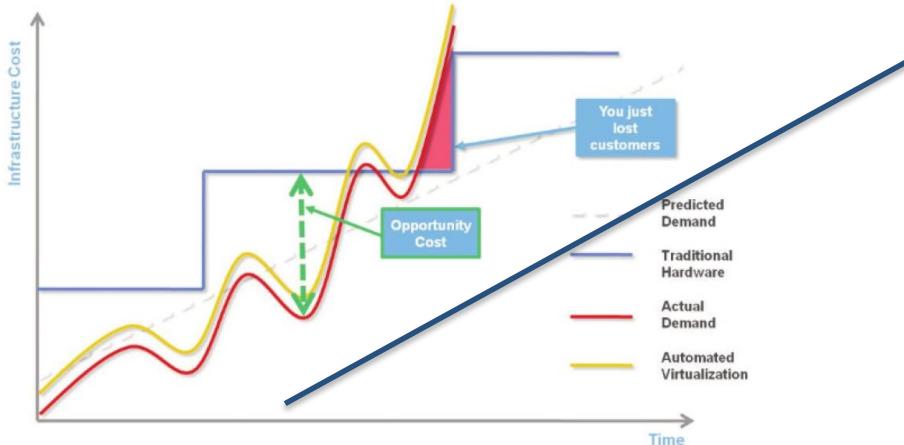
- **Tageszeitliche und saisonale Effekte:** Mittags-Peak, Prime-Time-Peak, Wochenend-Peak, Weihnachten, Valentinstag, Muttertag, ... (vorhersehbare Belastungsspitzen)



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

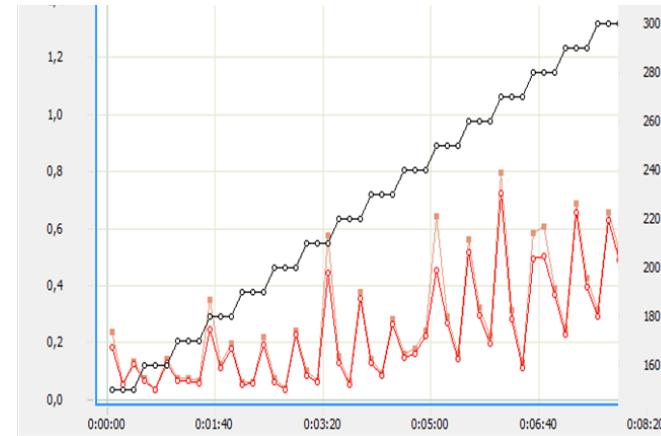


- **Kontinuierliches Wachstum**

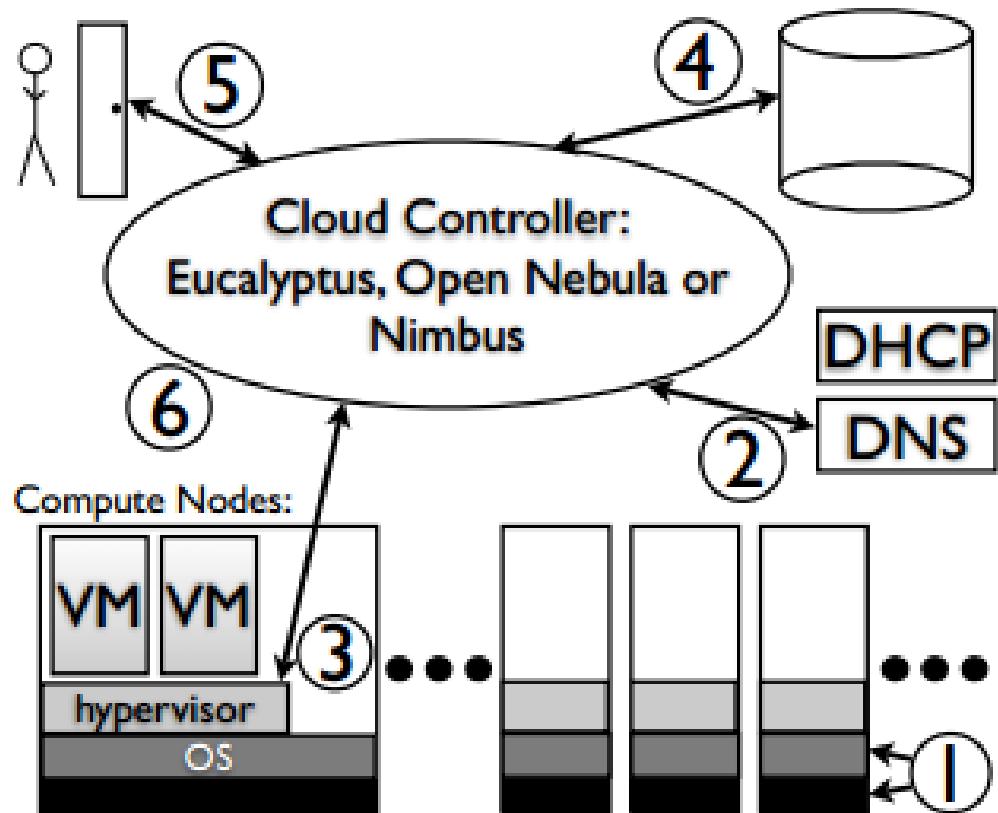


Source: Amazon Web Services

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



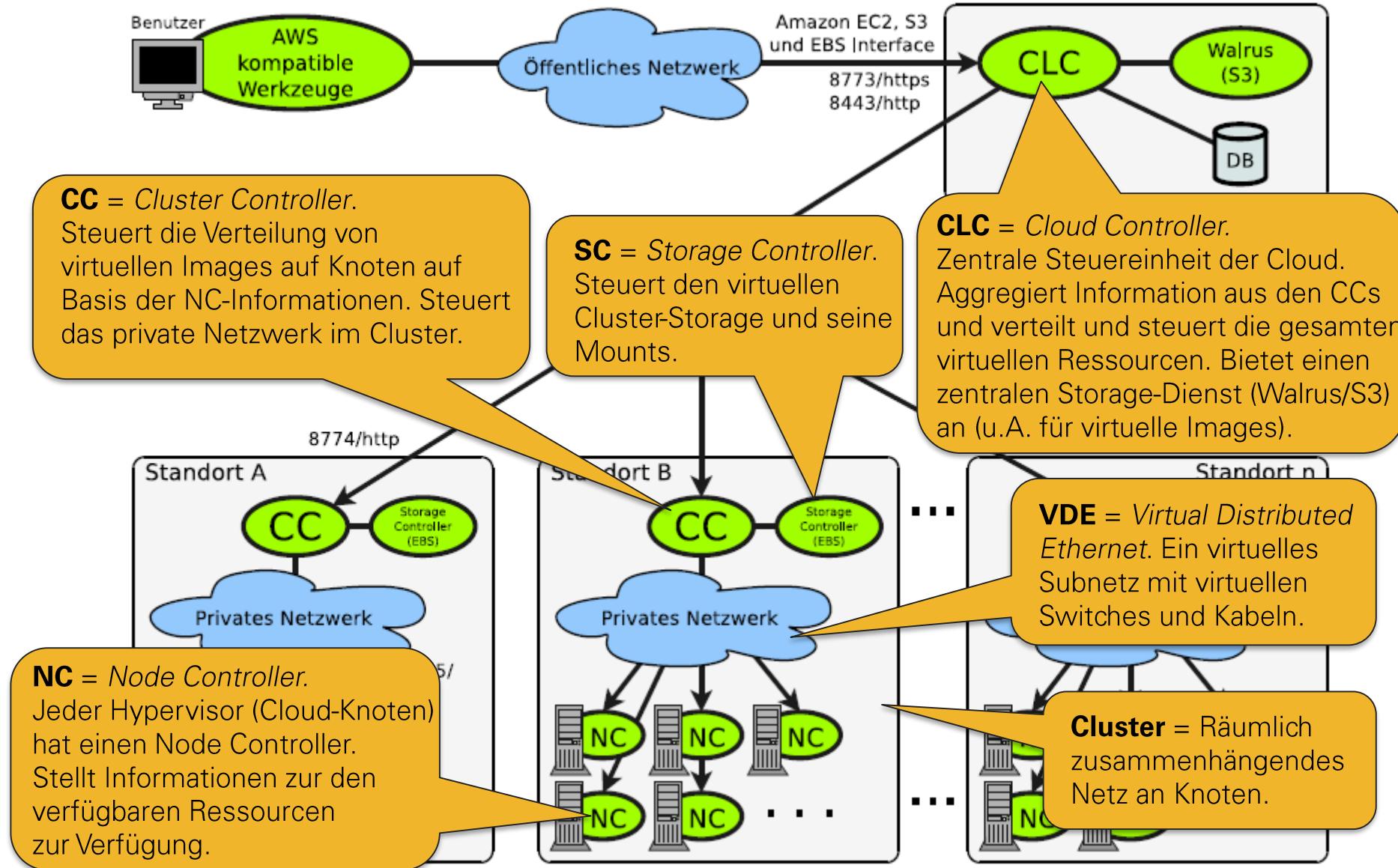
Eine IaaS-Referenzarchitektur.



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.



Es gibt eine Reihe an gängigen 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

Ein Service Level Agreement (SLA) ist ein Vertrag mit Zuverlässigkeitss zusagen für Ressourcen und Dienste.

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)

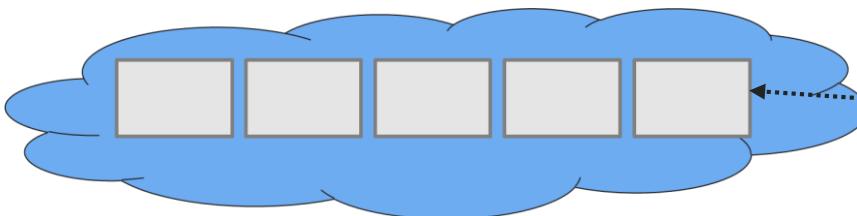
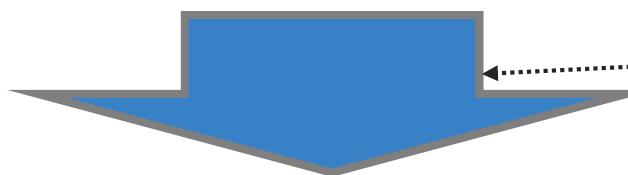
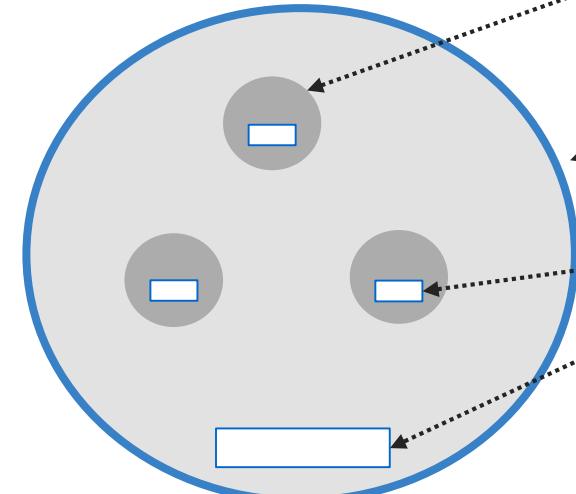
Service Commitment

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% |

Kapitel 5: 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.:

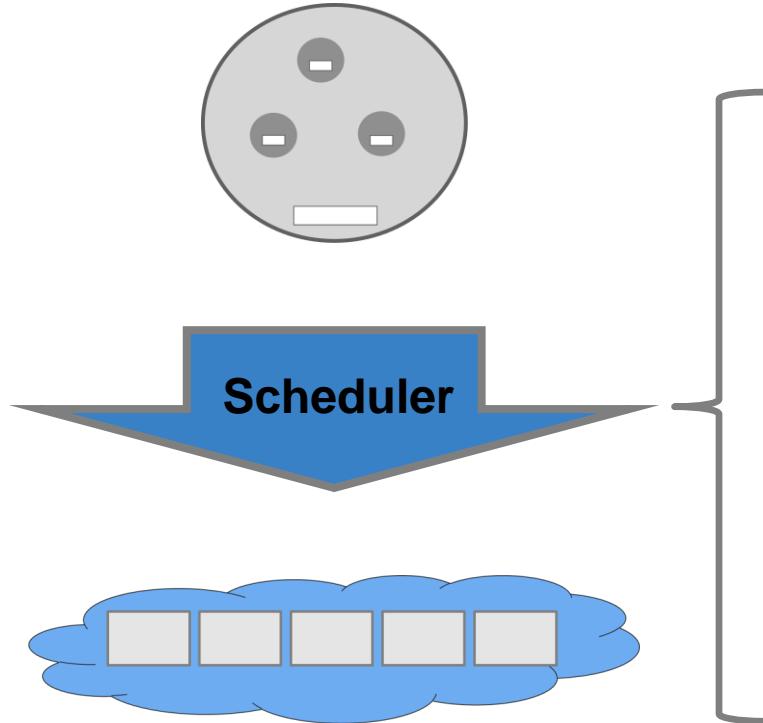
- Task: Ausführungszeitpunkt, Priorität, Ressourcenverbrauch
- Job: Abhängigkeiten der Tasks, Ausführungszeitpunkt

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:

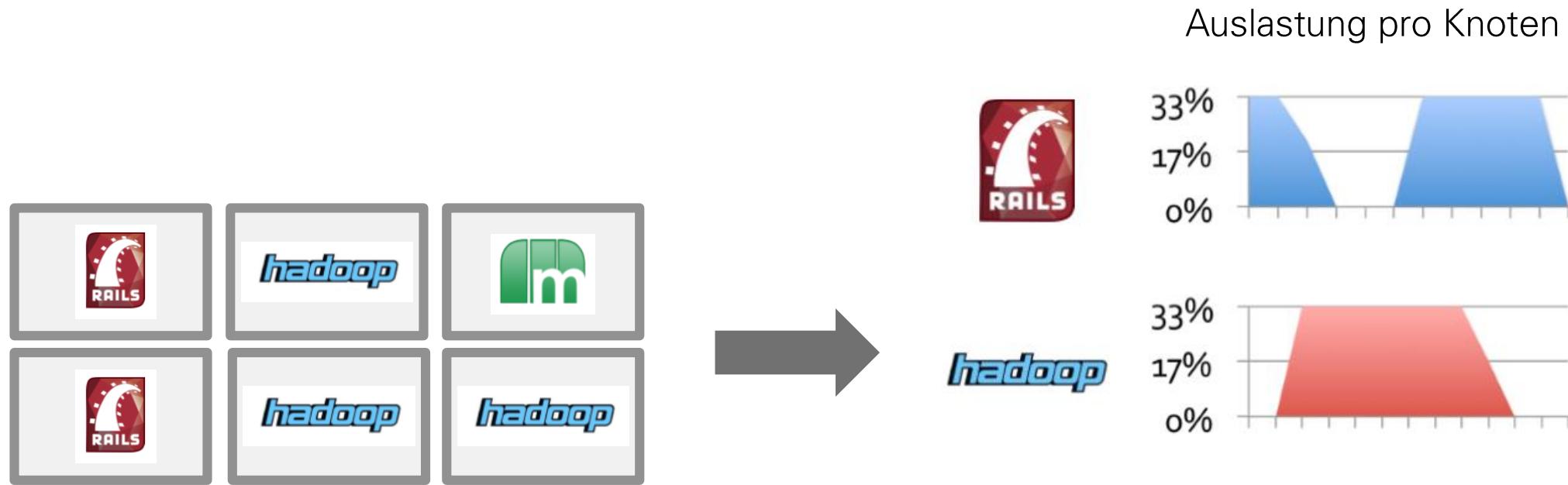


Cluster Awareness: Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.

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:

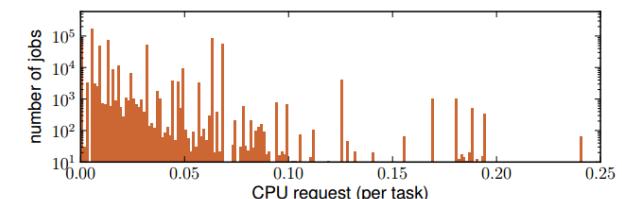
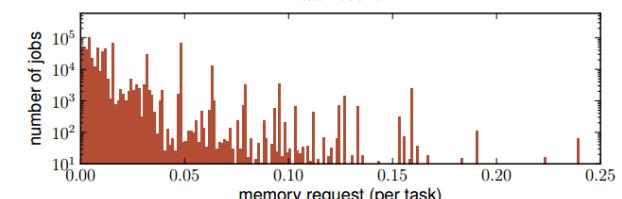
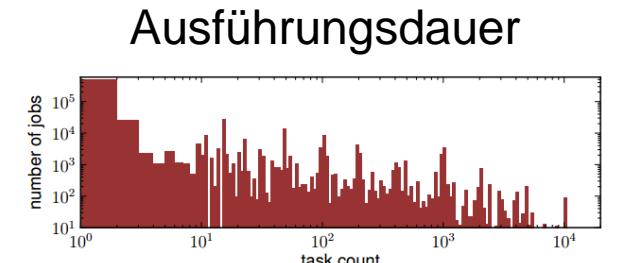
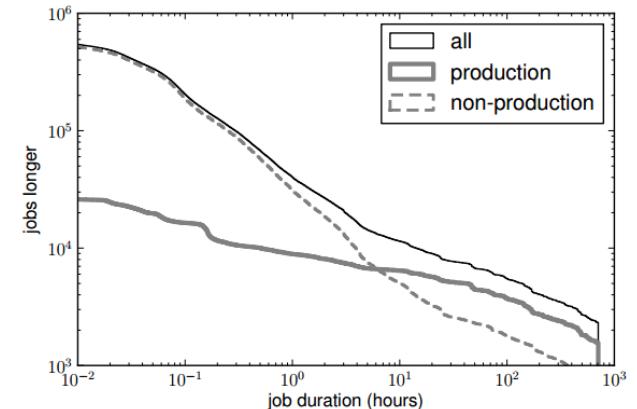
- Nicht flexibel bei geänderten Bedürfnissen
 - Geringere Auslastung
→ hohe Opportunitätskosten



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:
 - Ausführungsdauer: 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.



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 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.:

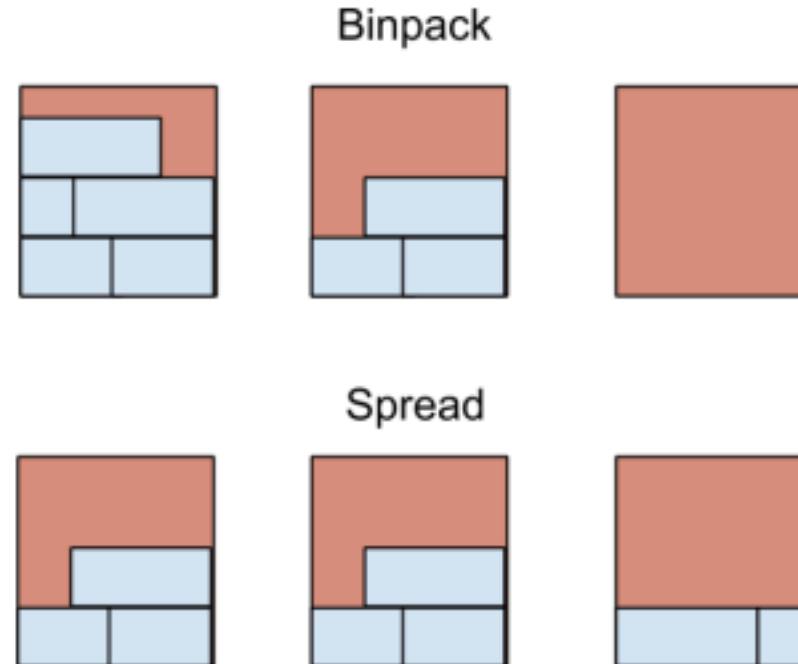
- **Fairness**: Kein Task sollte unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird.
- **Maximaler Durchsatz**: So viele Tasks pro Zeiteinheit wie möglich.
- **Minimale Wartezeit**: Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks.
- **Ressourcen-Auslastung**: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

Eine konzeptionelle Architektur für Cluster-Scheduler.

Job Queue:

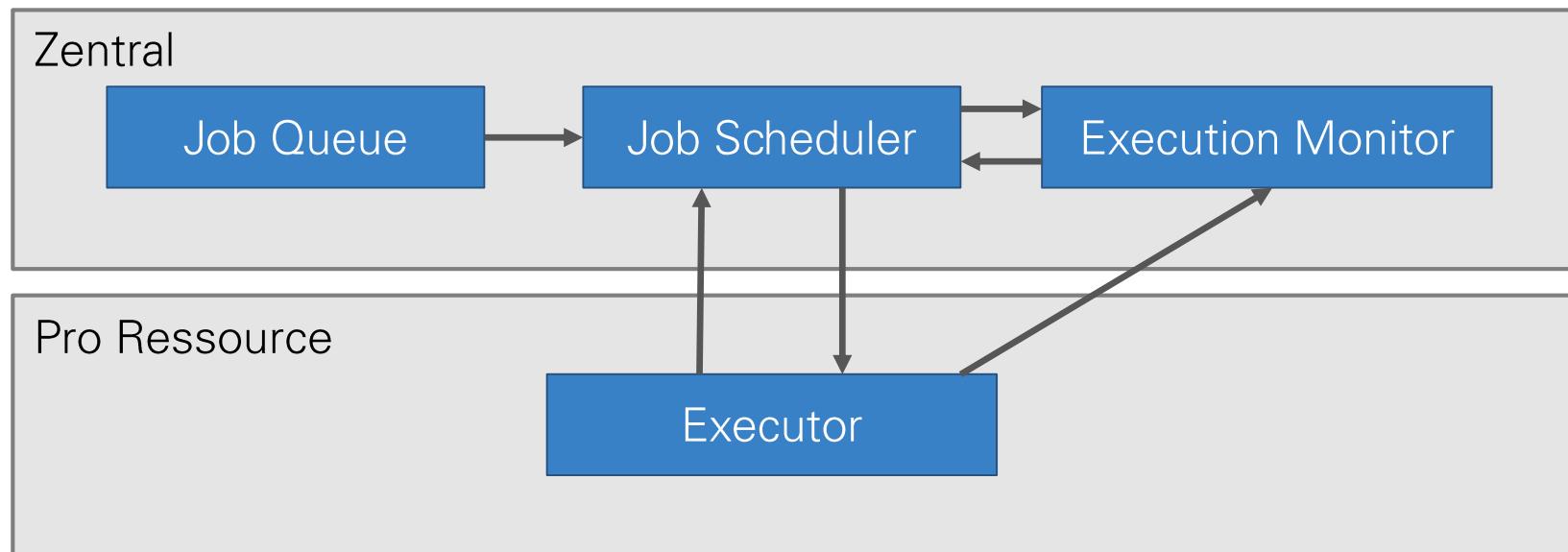
- Eingehende Jobs zur Ausführung
- Events zu eingegangenen Jobs

Job Scheduler:

- Jobs einplanen
- Taskausführung steuern

Execution Monitor:

- Task-Ausführung überwachen
- Ressourcen überwachen



Anforderungen:

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

Executor:

- Task ausführen
- Informationen zur Ressource bereitstellen

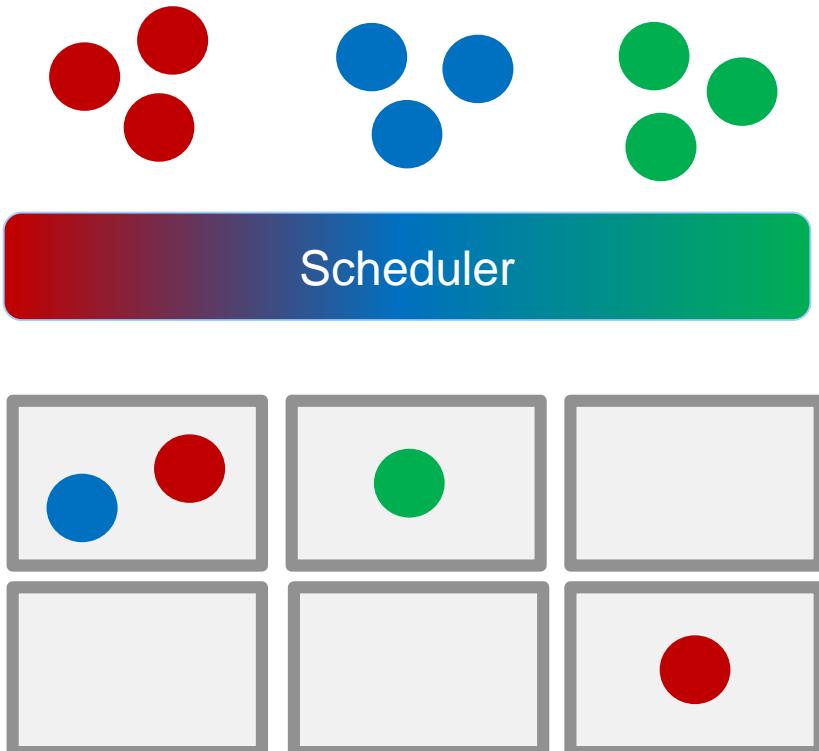
Scheduler-Architektur.

Variante 1: kein Scheduler.



Statische Partitionierung.

Scheduler-Architektur. Variante 2: Monolithischer Scheduler.



Vorteile:

- Globale Optimierungsstrategien einfach möglich.

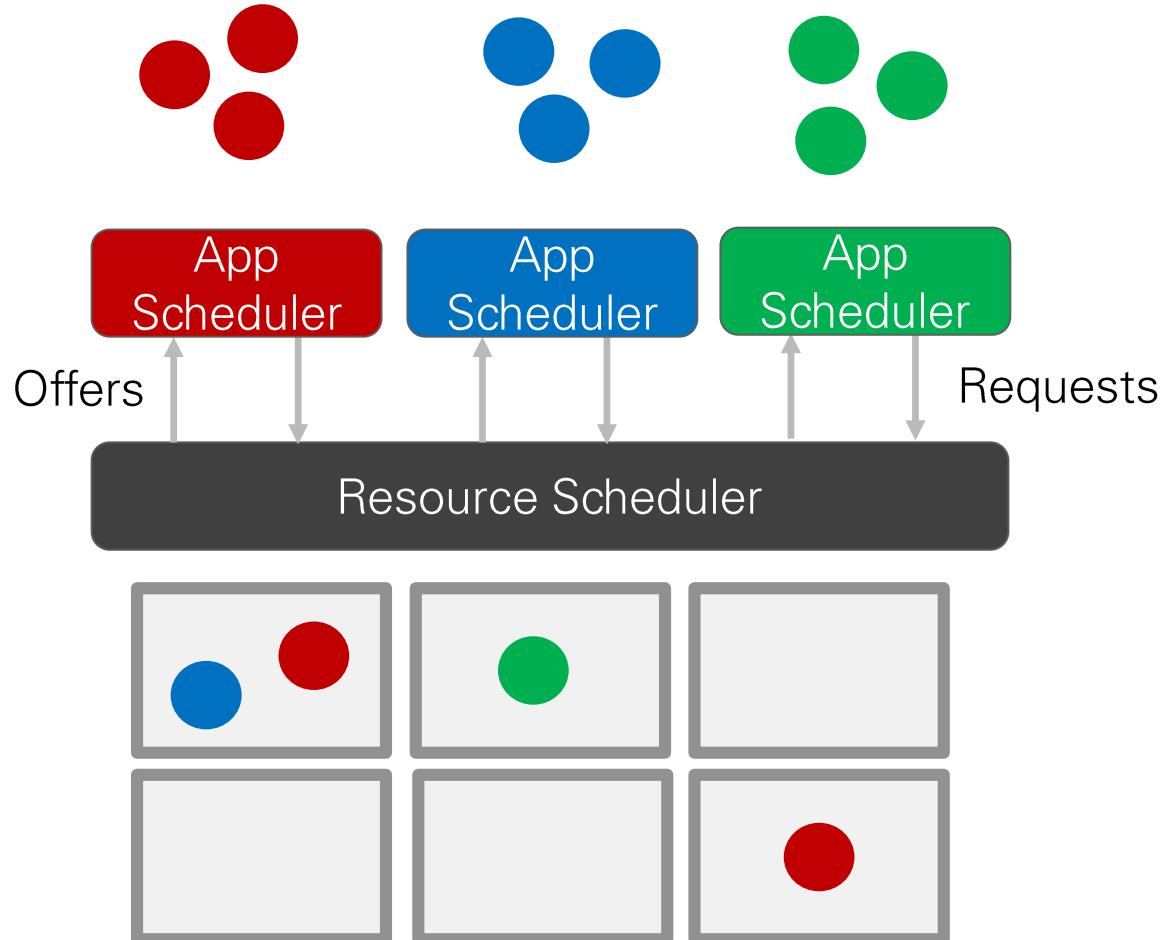
Nachteile:

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

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

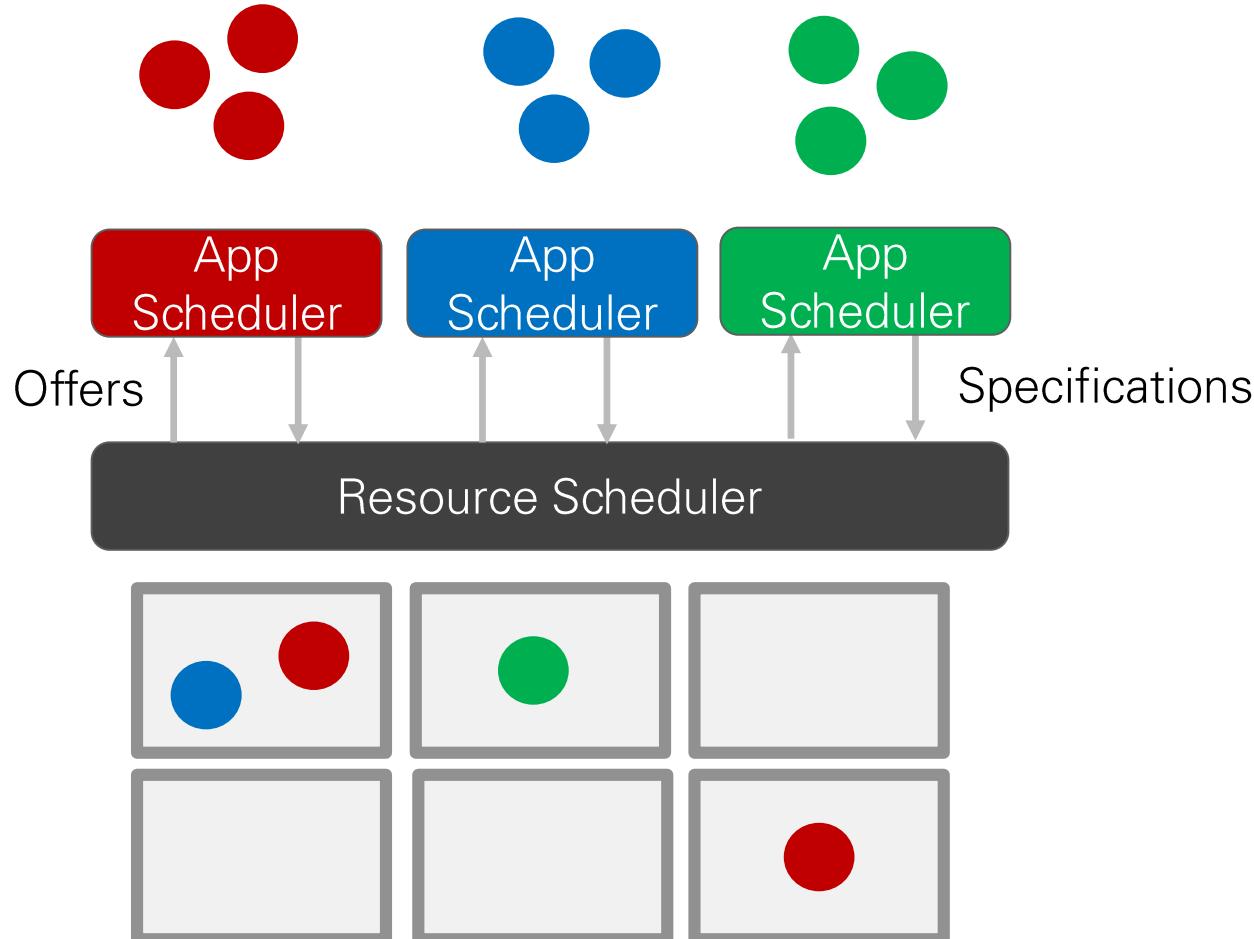
- **Hadoop YARN**
- **Kubernetes**
- **Docker Swarm**

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

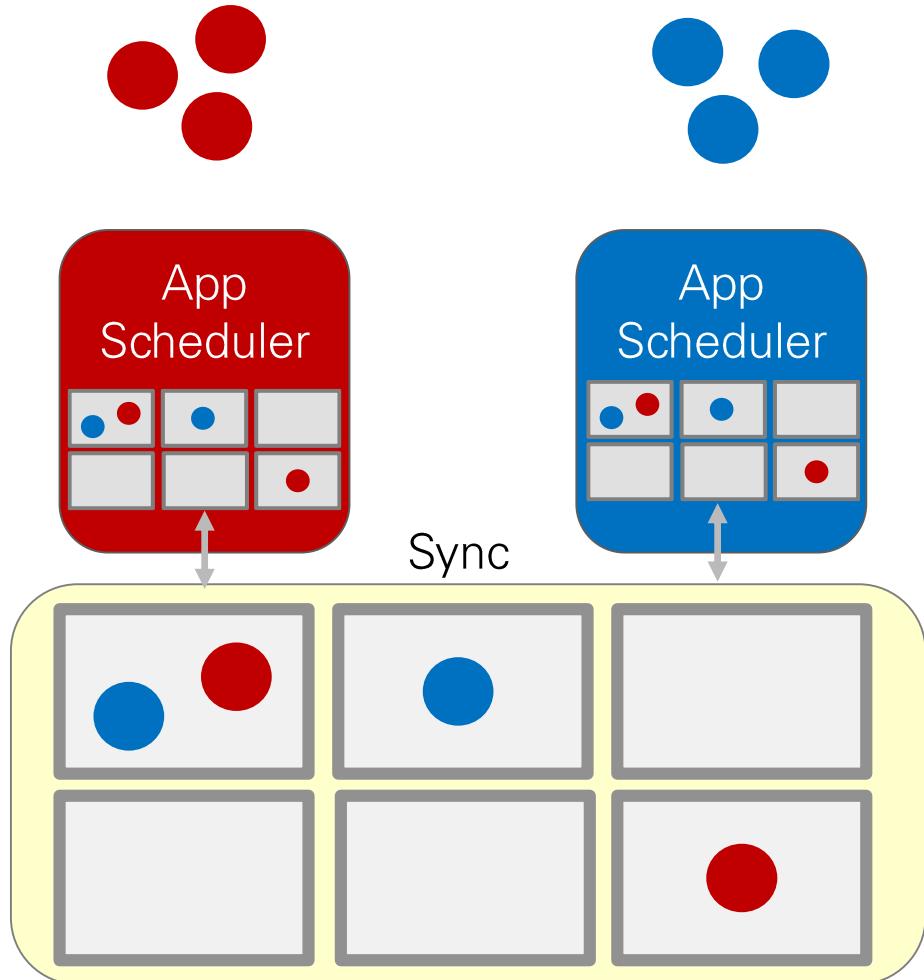
Vorteile:

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

Nachteile:

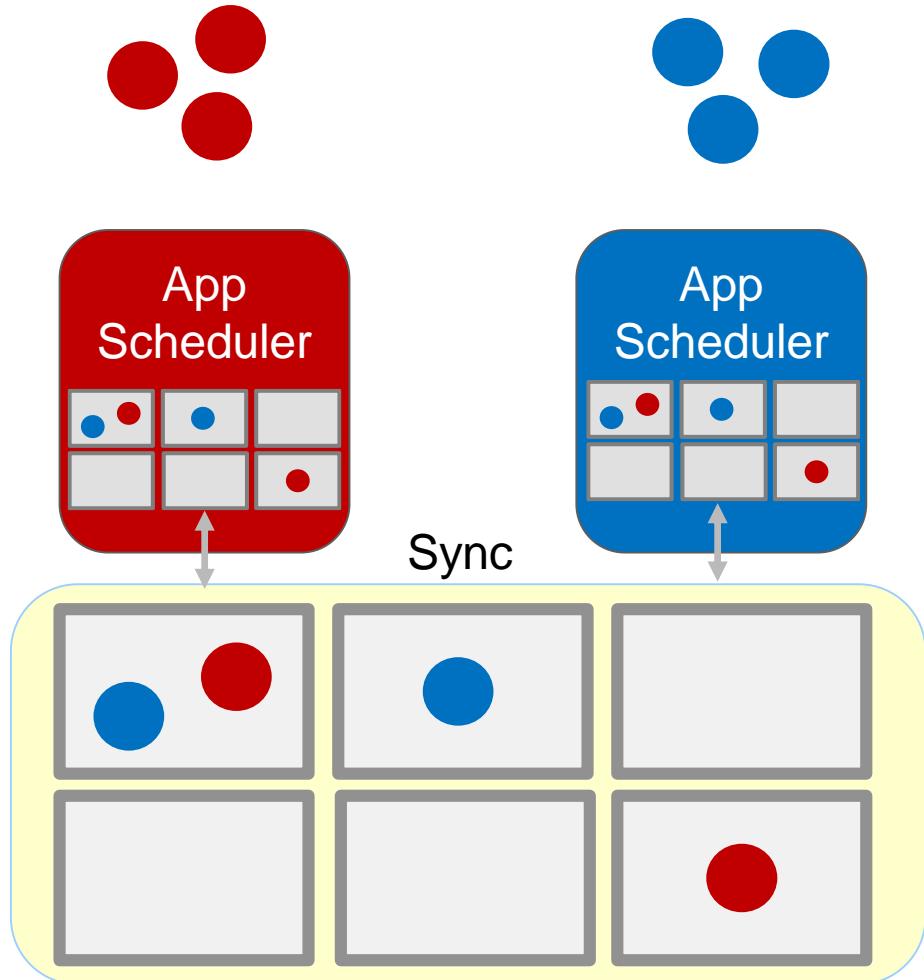
- App-Scheduler übergreifende Logiken nur schwer zu realisieren (z.B. globaler Ausführungsverzicht oder Gang-Scheduling)

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



- Es gibt ausschließlich applicationsspezifische 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

Kapitel 6: 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.



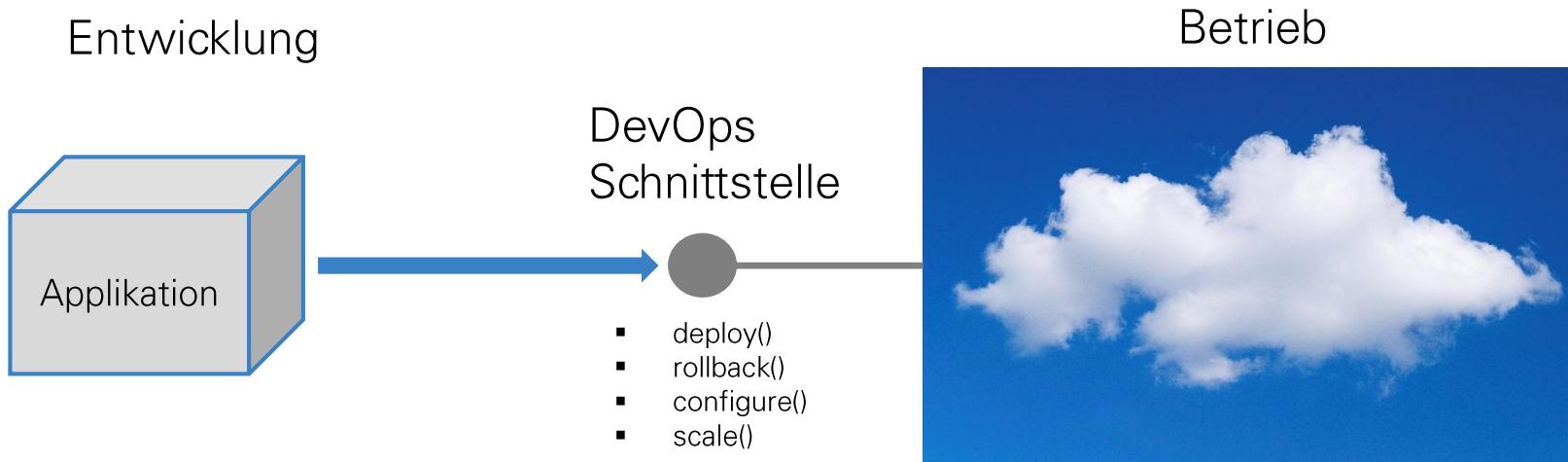
Cluster-Orchestrator



Steuerungsaktivitäten im Cluster:

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

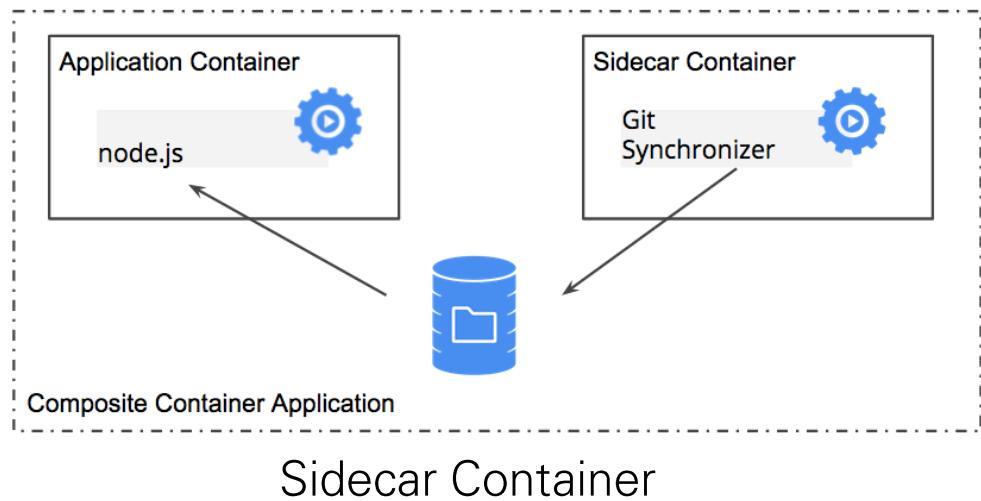
Ein Cluster-Orchestrator bietet eine Schnittstelle zwischen Betrieb und Entwicklung für ein Cluster an.



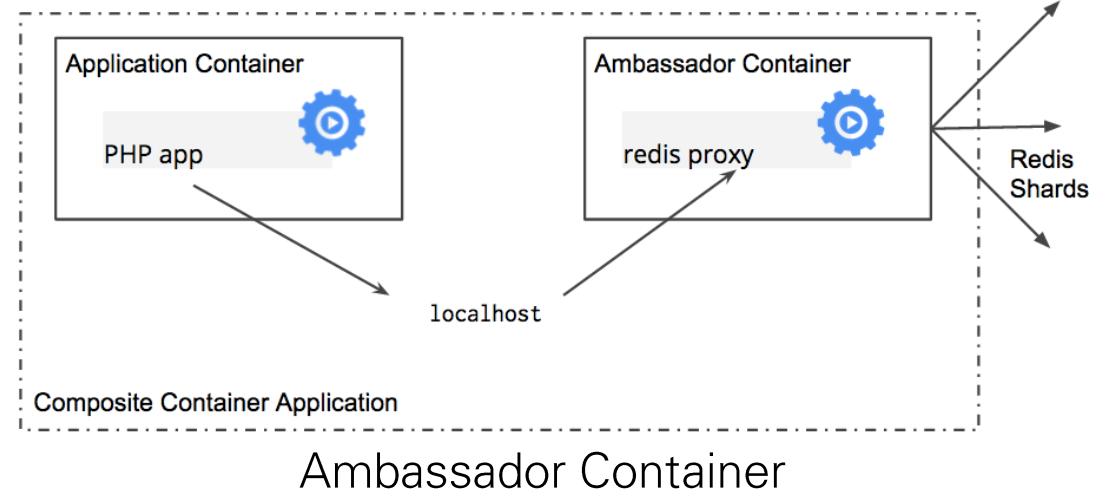
Ein Cluster-Orchestrator automatisiert vielerlei Betriebsaufgaben für Anwendungen auf einem Cluster.

- 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.

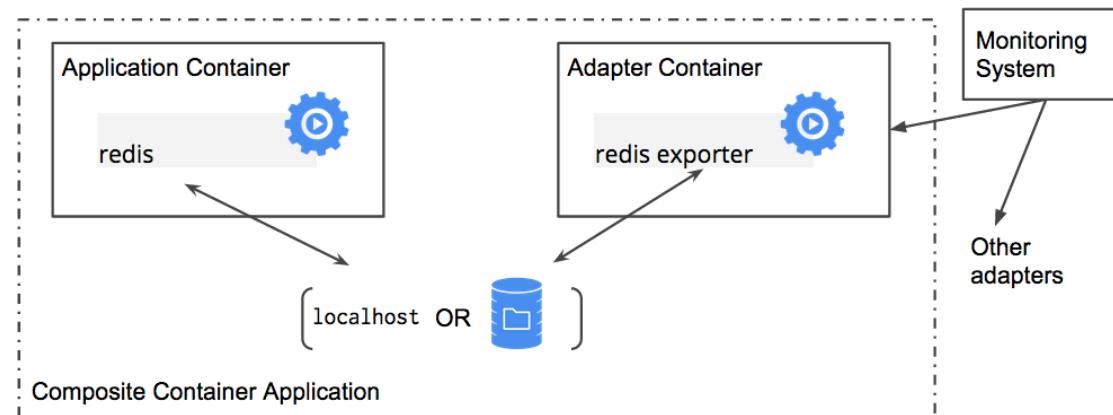
Orchestrierungsmuster – Separation of Concerns mit modularen Containern



Sidecar Container



Ambassador Container

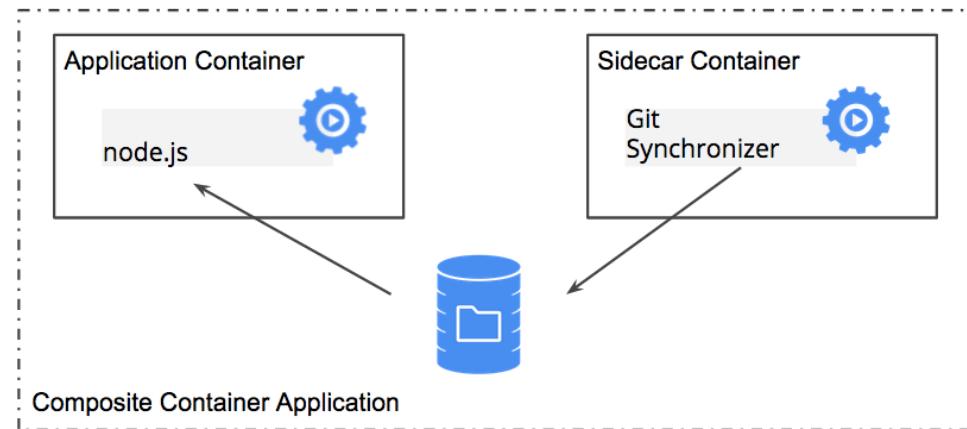


Adapter Container

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.

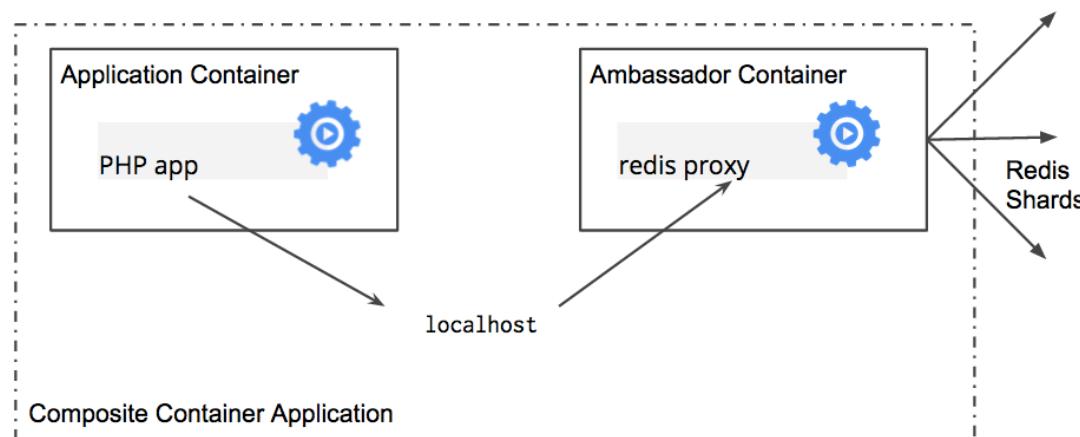
Quelle: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>



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.

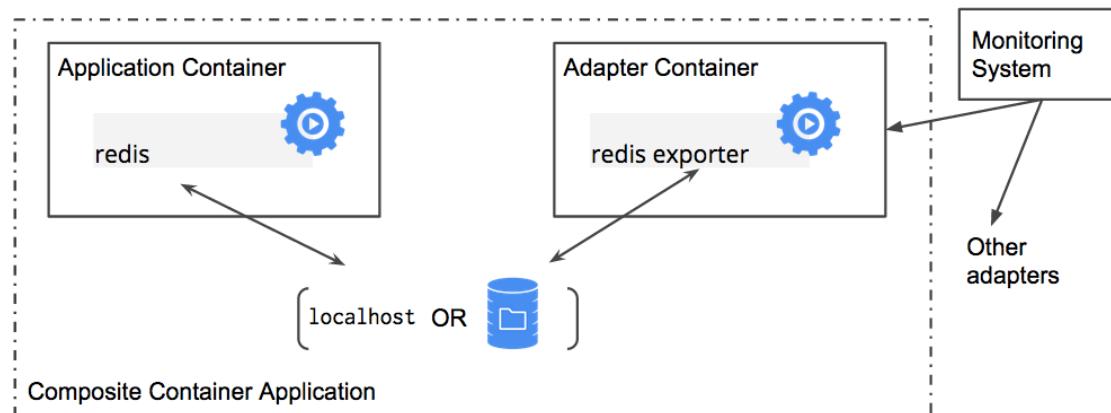
Quelle: <https://kubernetes.io/blog/2015/06/the-distributed-system-toolkit-patterns/>



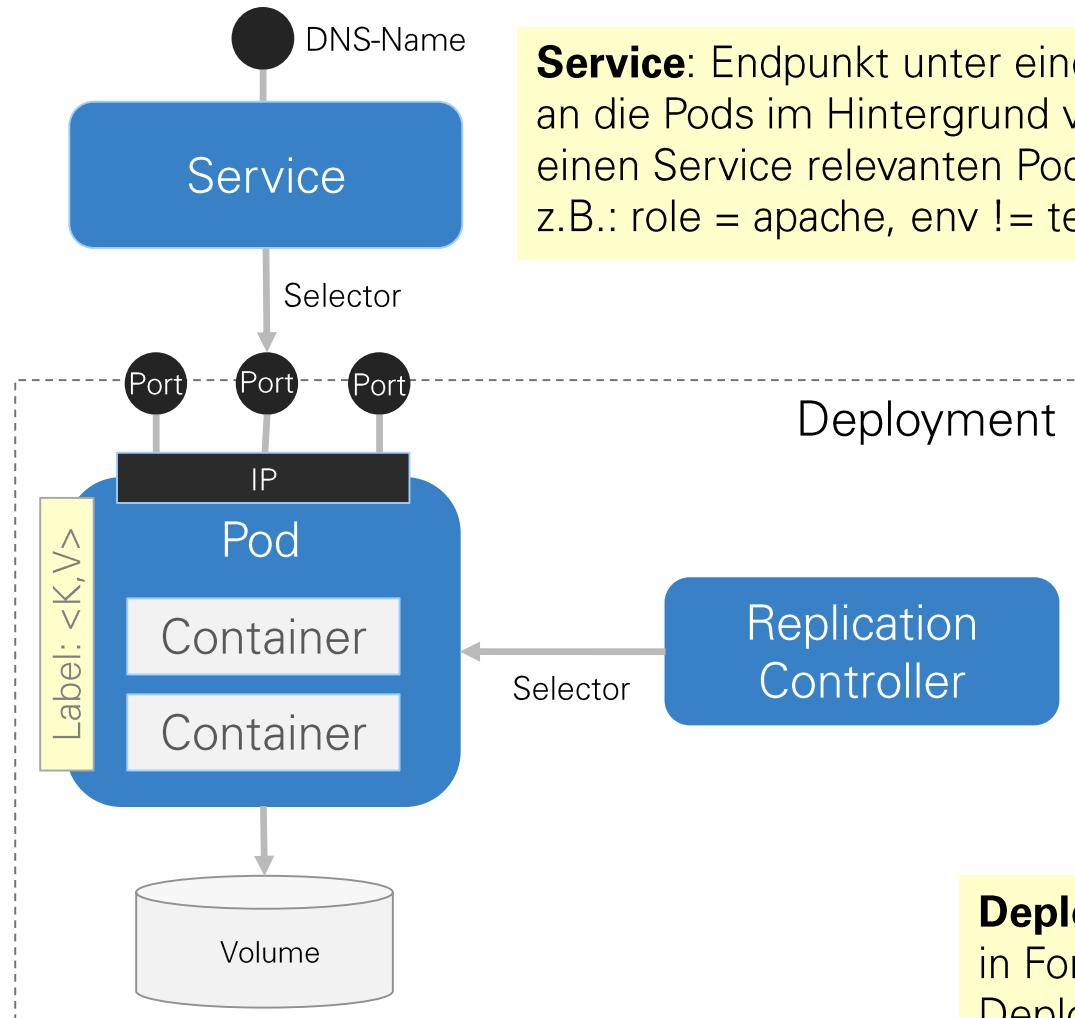
Adapter containers

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/>



Die 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.

Service Definition

```
apiVersion: v1
kind: Service
metadata:
  name: zwitscher-service
  labels:
    zwitscher: service
spec:
  # use NodePort here to be able to access the port on each node
  # use LoadBalancer for external load-balanced IP if supported
  type: NodePort
  ports:
  - port: 8080
  selector:
    zwitscher: service
```

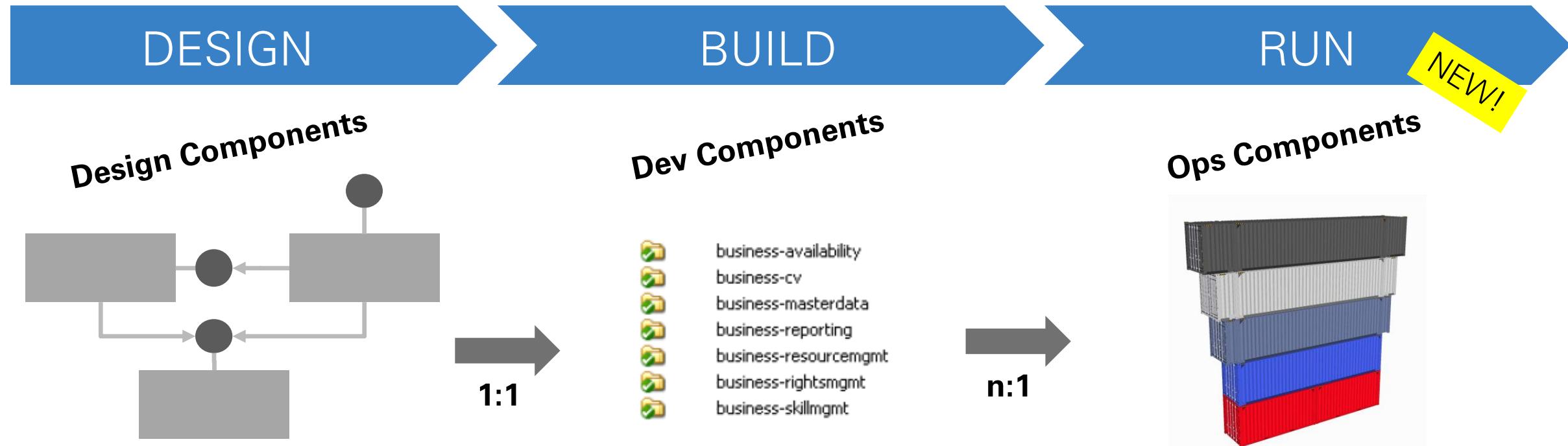
Liveness und Readiness Probes

```
# container will receive requests if probe succeeds
readinessProbe:
  httpGet:
    path: /admin/info
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5

# container will be killed if probe fails
livenessProbe:
  httpGet:
    path: /admin/health
    port: 8080
  initialDelaySeconds: 90
  timeoutSeconds: 10
```

Kapitel 7: 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

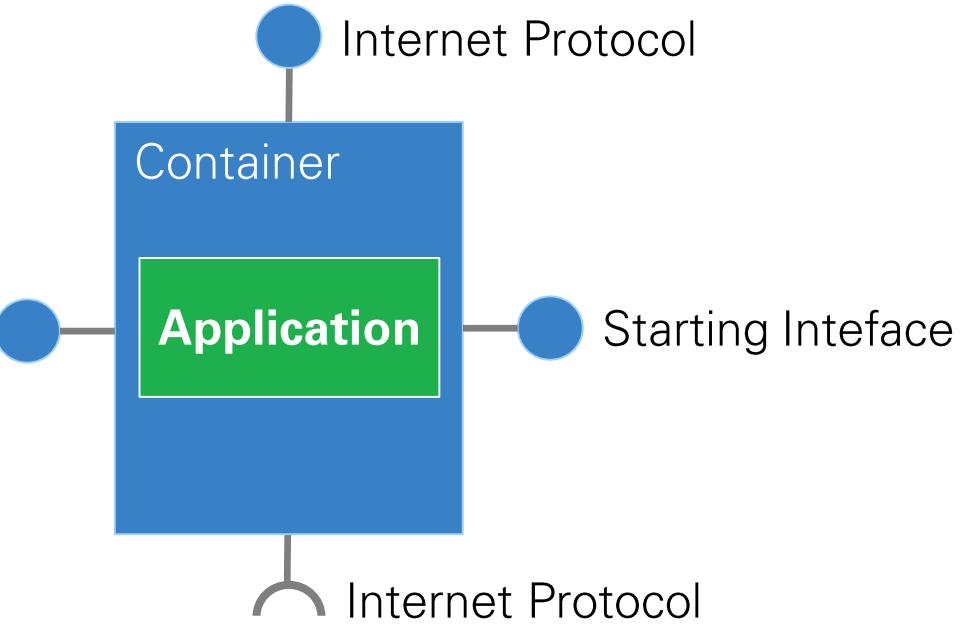
- Planning unit
- Team assignment unit
- Knowledge unit
- Development unit
- Integration unit

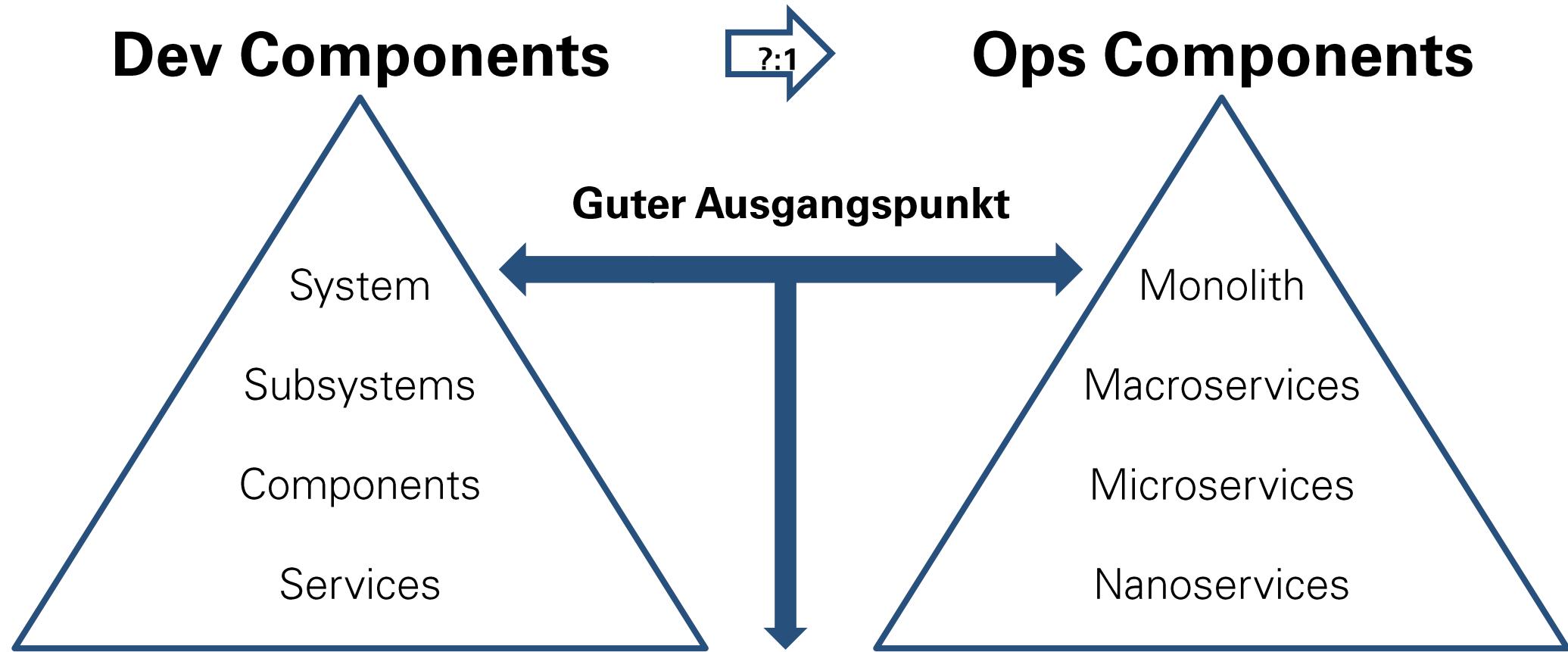
- Release unit
- Deployment unit
- Runtime unit (crash, slow-down, access)
- Scaling unit

Die Anatomie einer Betriebs-Komponente.



Diagnose Interface





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

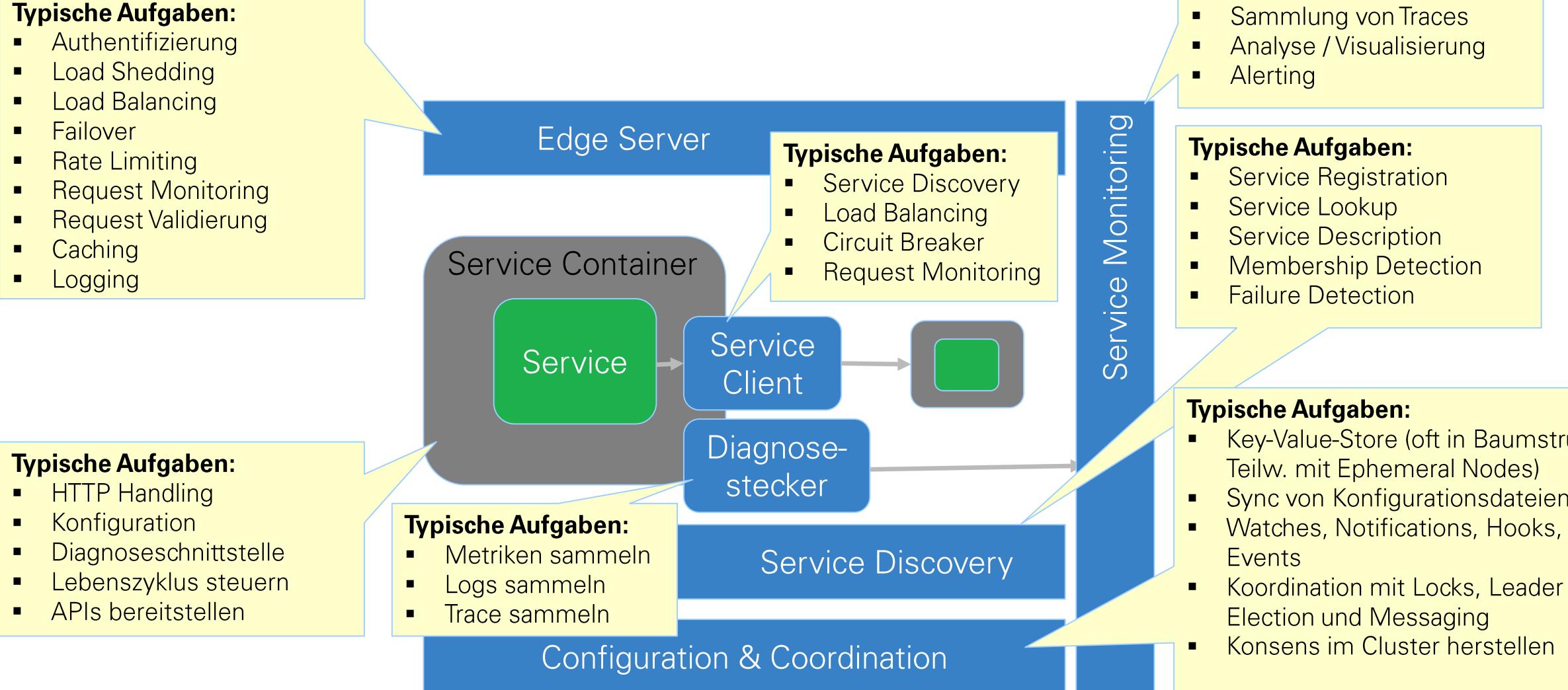
Betriebskomponenten benötigen eine Infrastruktur um sie herum: Eine Micro-Service-Plattform.

Typische Aufgaben:

- Authentifizierung
- Load Shedding
- Load Balancing
- Failover
- Rate Limiting
- Request Monitoring
- Request Validierung
- Caching
- Logging

Typische Aufgaben:

- HTTP Handling
- Konfiguration
- Diagnoseschnittstelle
- Lebenszyklus steuern
- APIs bereitstellen



Typische Aufgaben:

- Aggregation von Metriken
- Sammlung von Logs
- Sammlung von Traces
- Analyse / Visualisierung
- Alerting

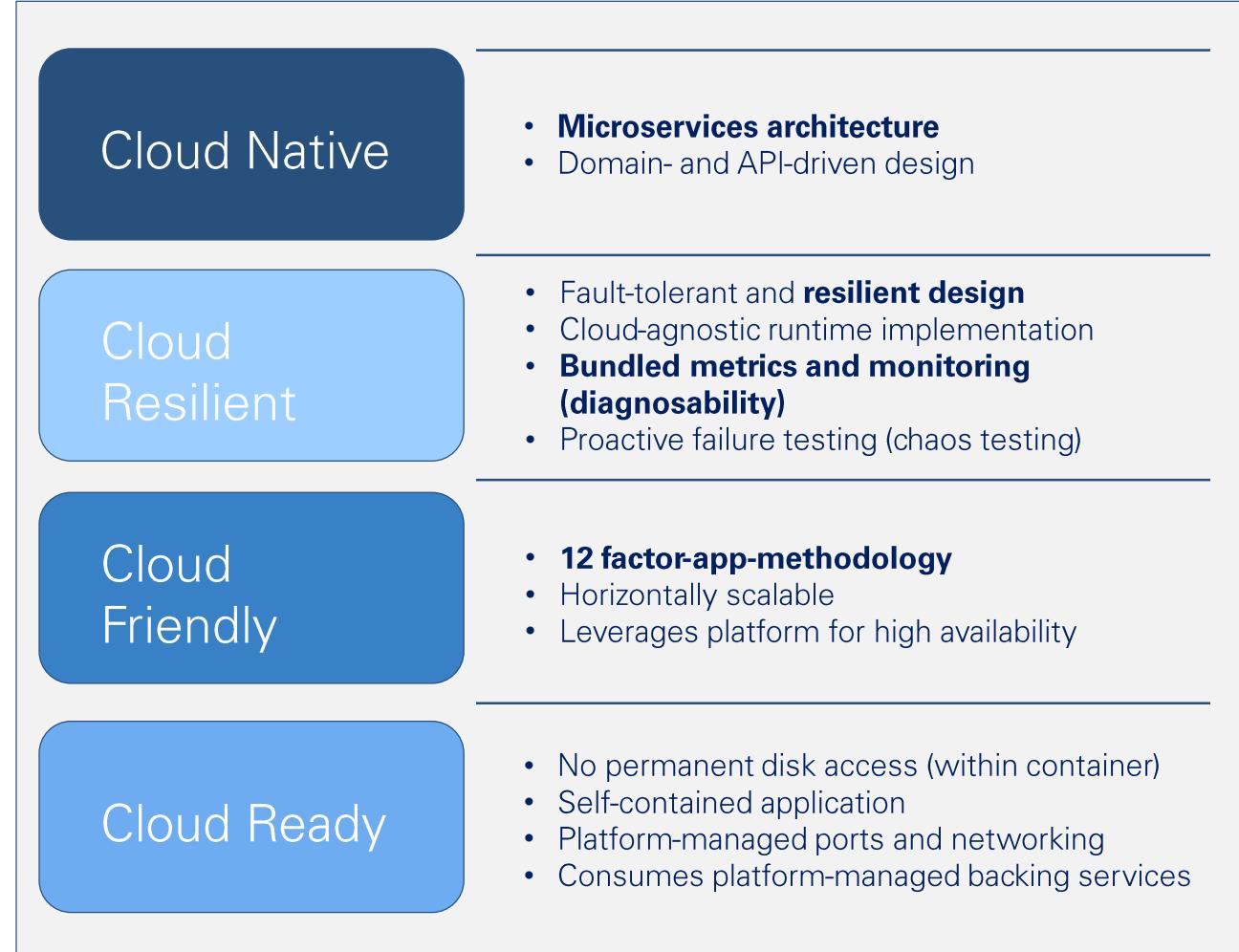
Typische Aufgaben:

- Service Registration
- Service Lookup
- Service Description
- Membership Detection
- Failure Detection

Typische Aufgaben:

- Key-Value-Store (oft in Baumstruktur, Teilw. mit Ephemeral Nodes)
- Sync von Konfigurationsdateien
- Watches, Notifications, Hooks, Events
- Koordination mit Locks, Leader Election und Messaging
- Konsens im Cluster herstellen

Das Cloud Native Application Reifegradmodell

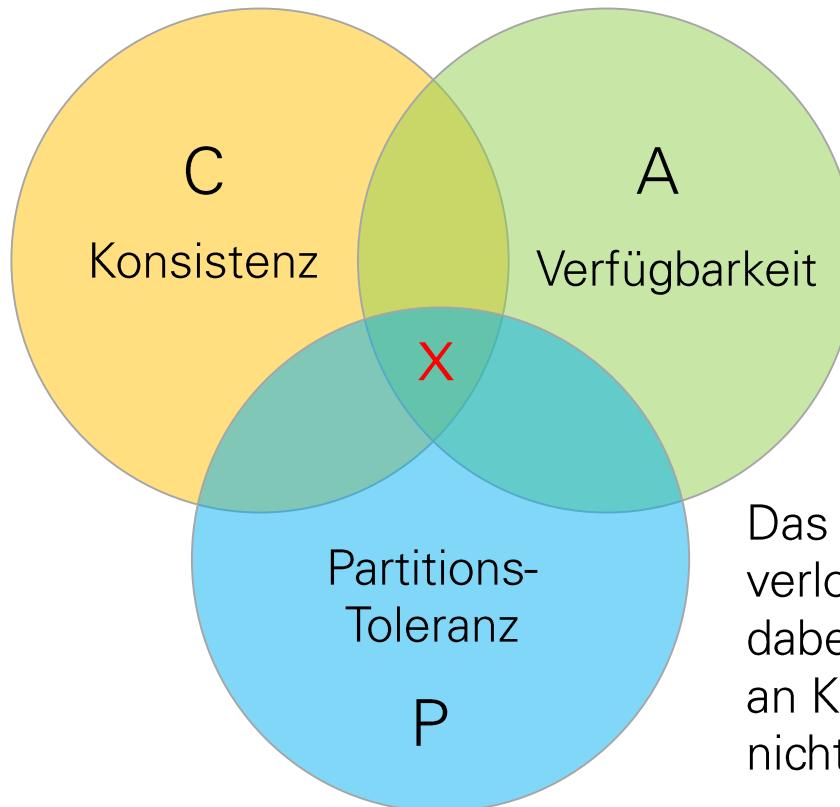


Das CAP Theorem.

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:



Alle Knoten sehen die selben Daten zur selben Zeit. Alle Kopien sind stets gleich.

Das System läuft auch, wenn einzelne Knoten ausfallen. Ausfälle von Knoten und Kanälen halten die überlebenden Knoten nicht von ihrer Funktion ab.

Das System funktioniert auch im Fall von verlorenen Nachrichten. Das System kann dabei damit umgehen, dass sich das Netzwerk an Knoten in mehrere Partitionen aufteilt, die nicht miteinander kommunizieren.

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

Grundlage: Ein Netzwerk an Agenten mit eigenem Zustand

Agenten verteilen einen Gossip-Strom

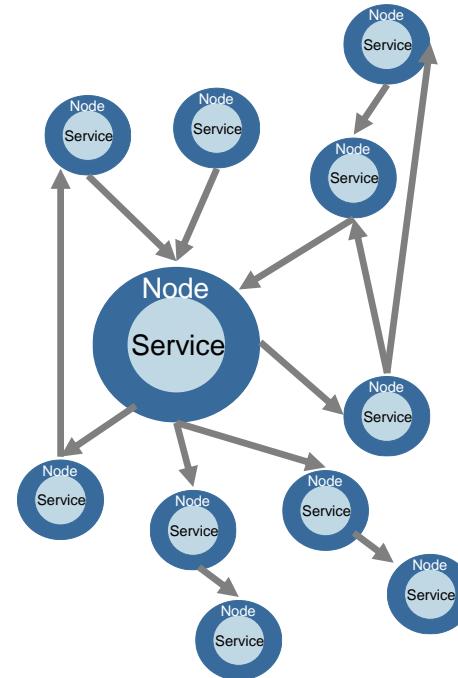
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

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.

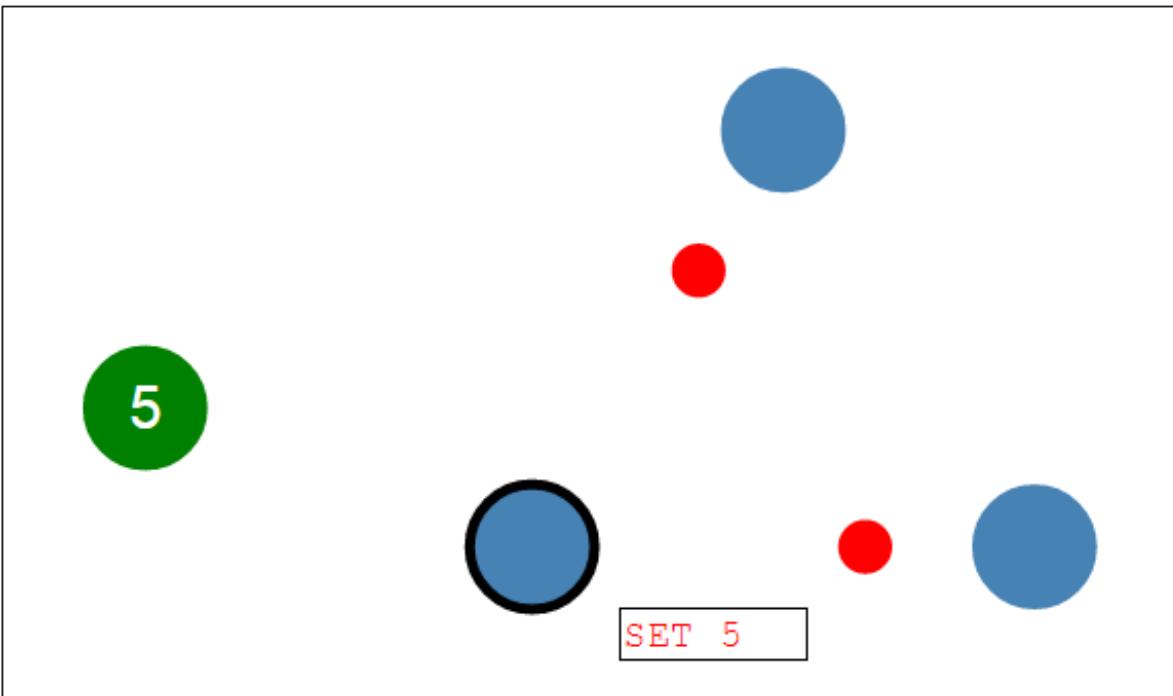
Nachteile:

- 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 Raft Konsens-Protokoll

Ongaro, Diego; Ousterhout, John (2013).

"In Search of an Understandable Consensus Algorithm".

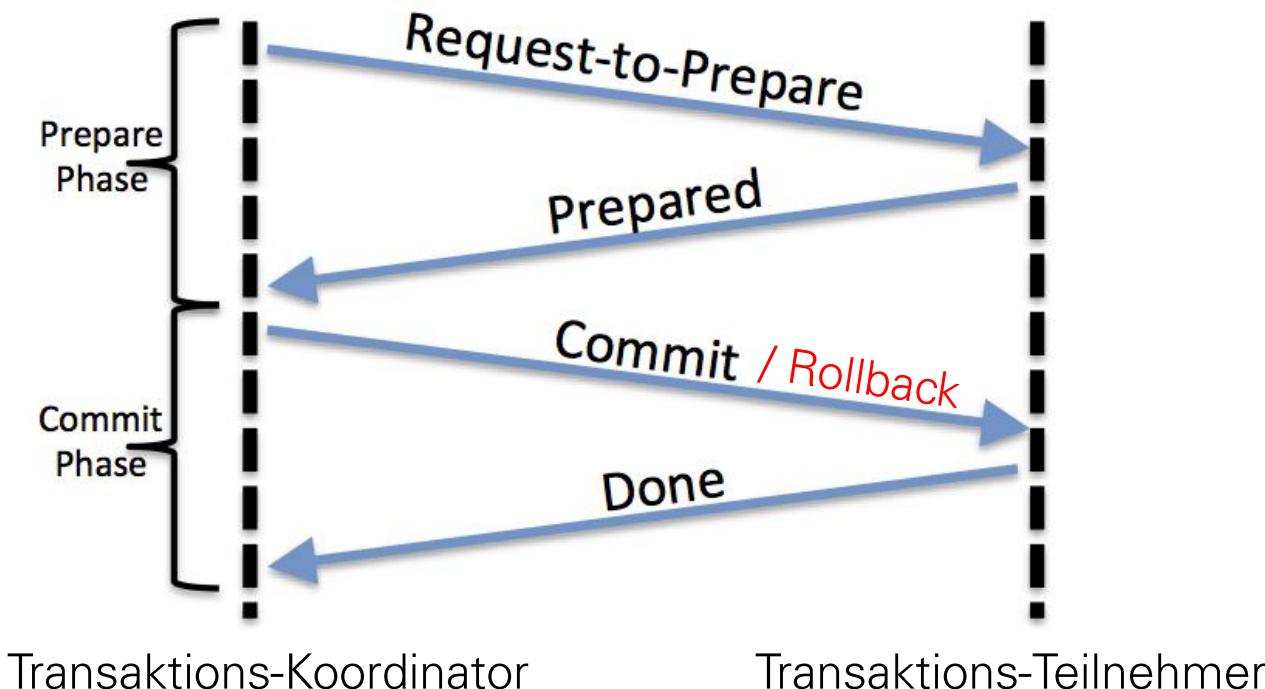


<http://thesecretlivesofdata.com/raft>

<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.



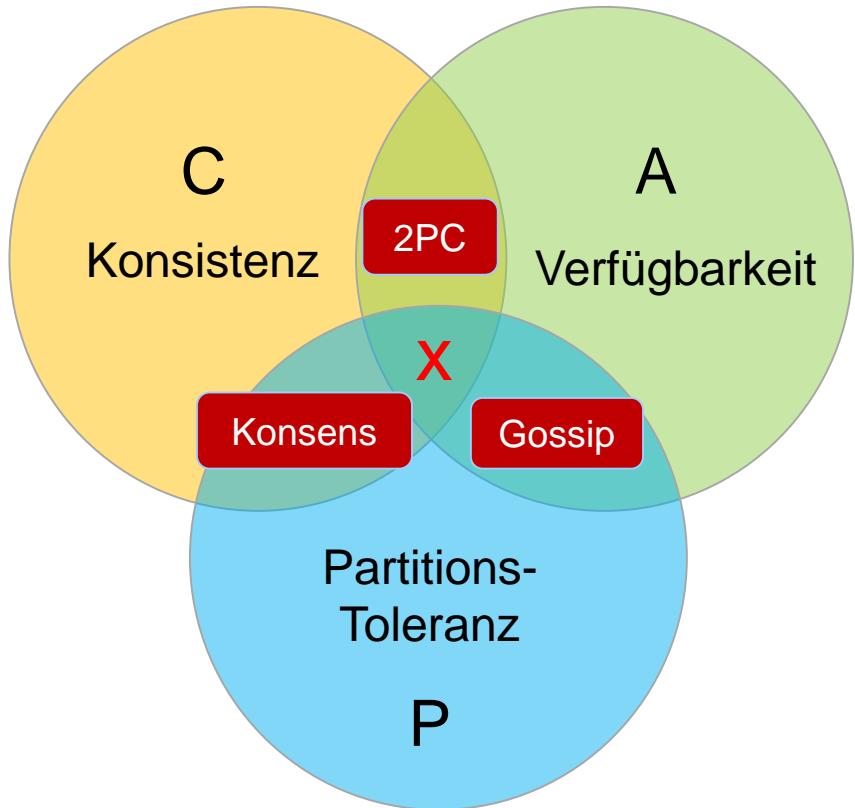
Vorteil:

- Alle Knoten sind konsistent zueinander.

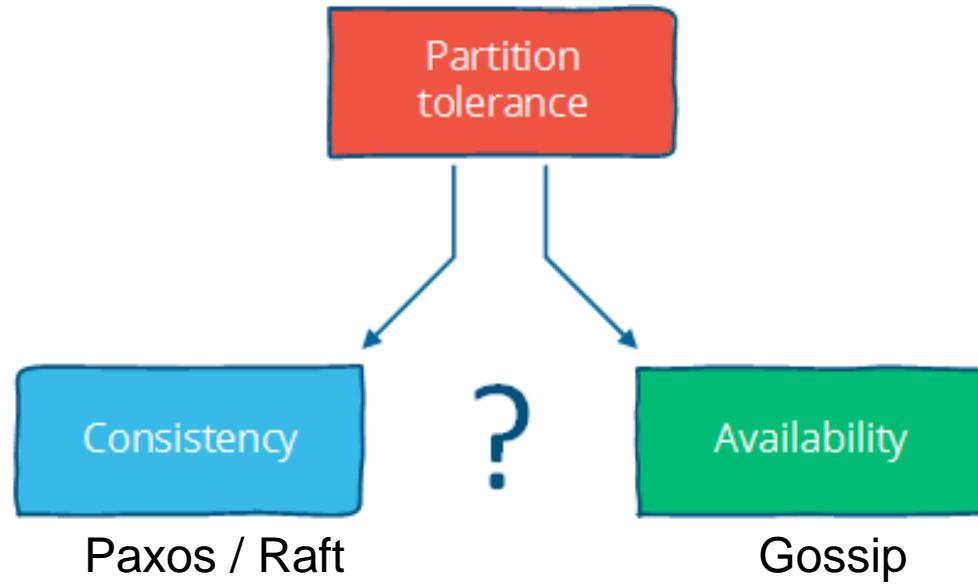
Nachteile:

- 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.

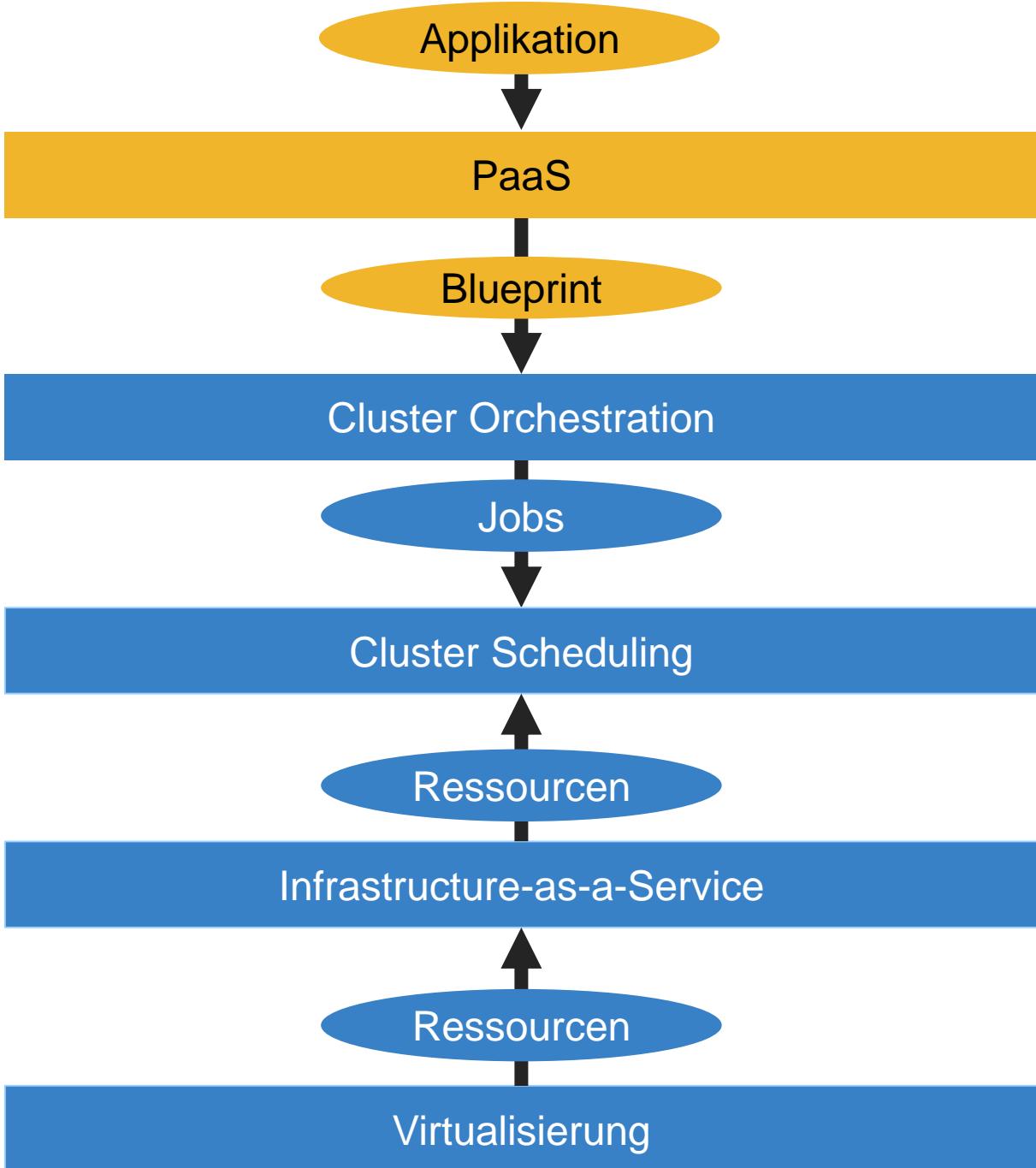


Kapitel 8: Platform-as-a-Service

Das Big Picture

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

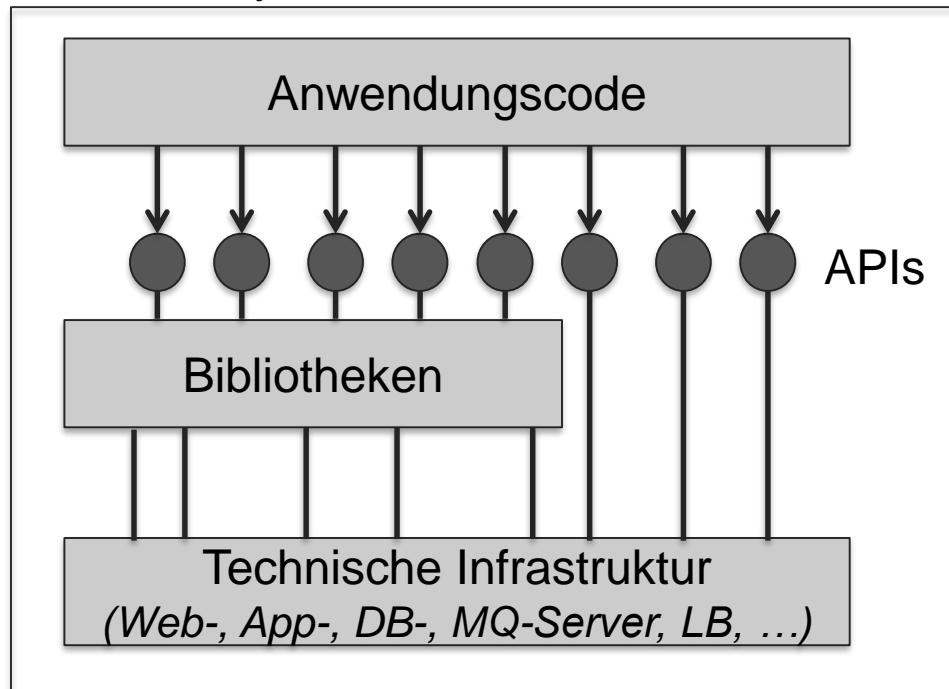
- Wiederverwendung von Infrastruktur / APIs
- Komfort-Dienste für Entwickler



Das Problem: Stovepipe Architecture. Anwendungen aufwändig von Hand verdrahten.

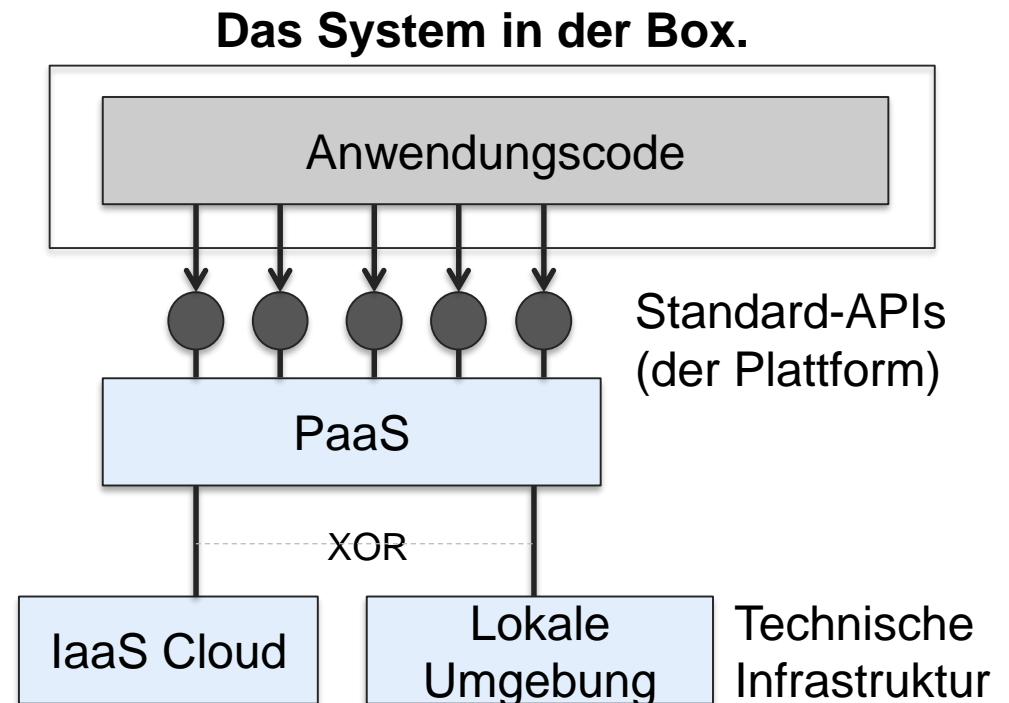


Das System: Mühevoll verdrahtet.

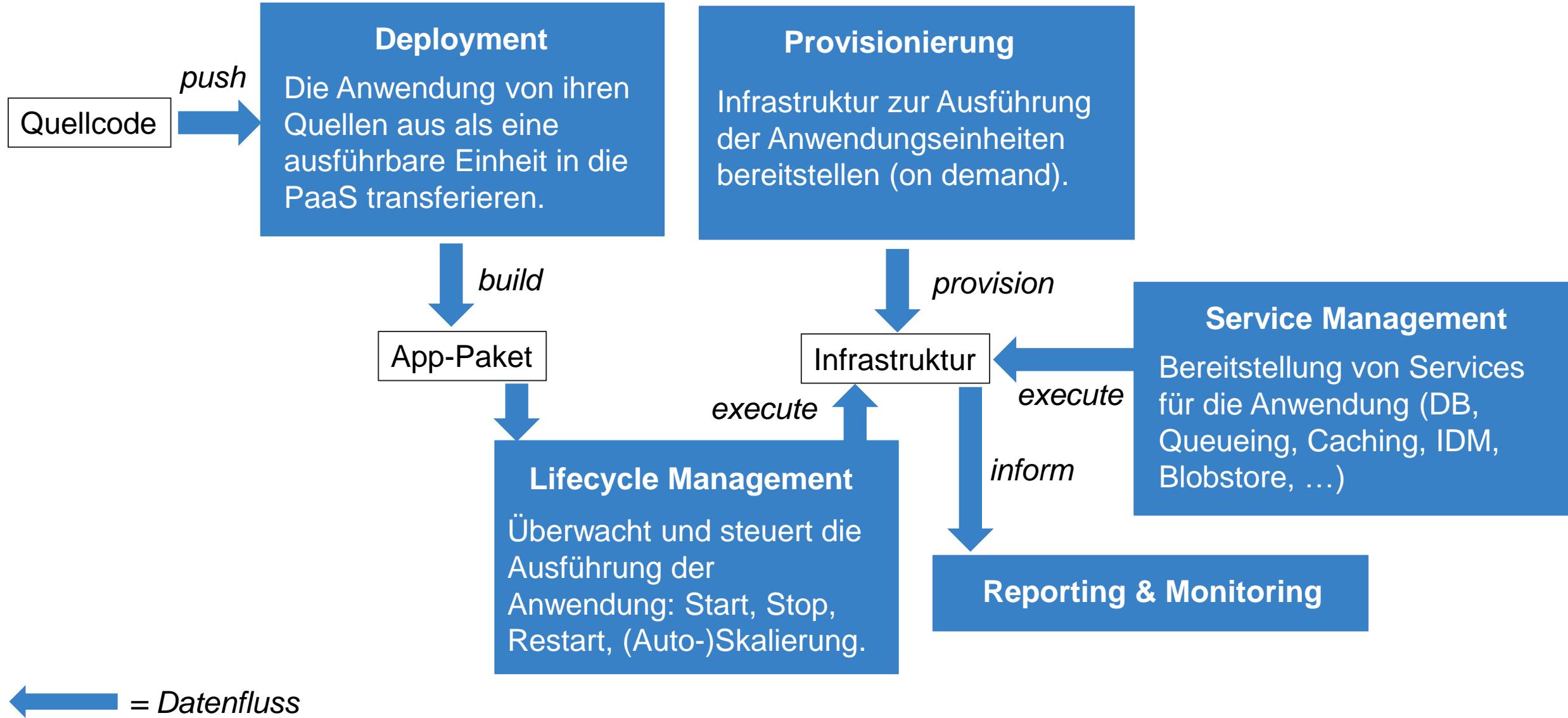


Die Lösung: Plattform-as-a-Service bietet eine ad-hoc Entwicklungs- und Betriebsplattform.

- 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.



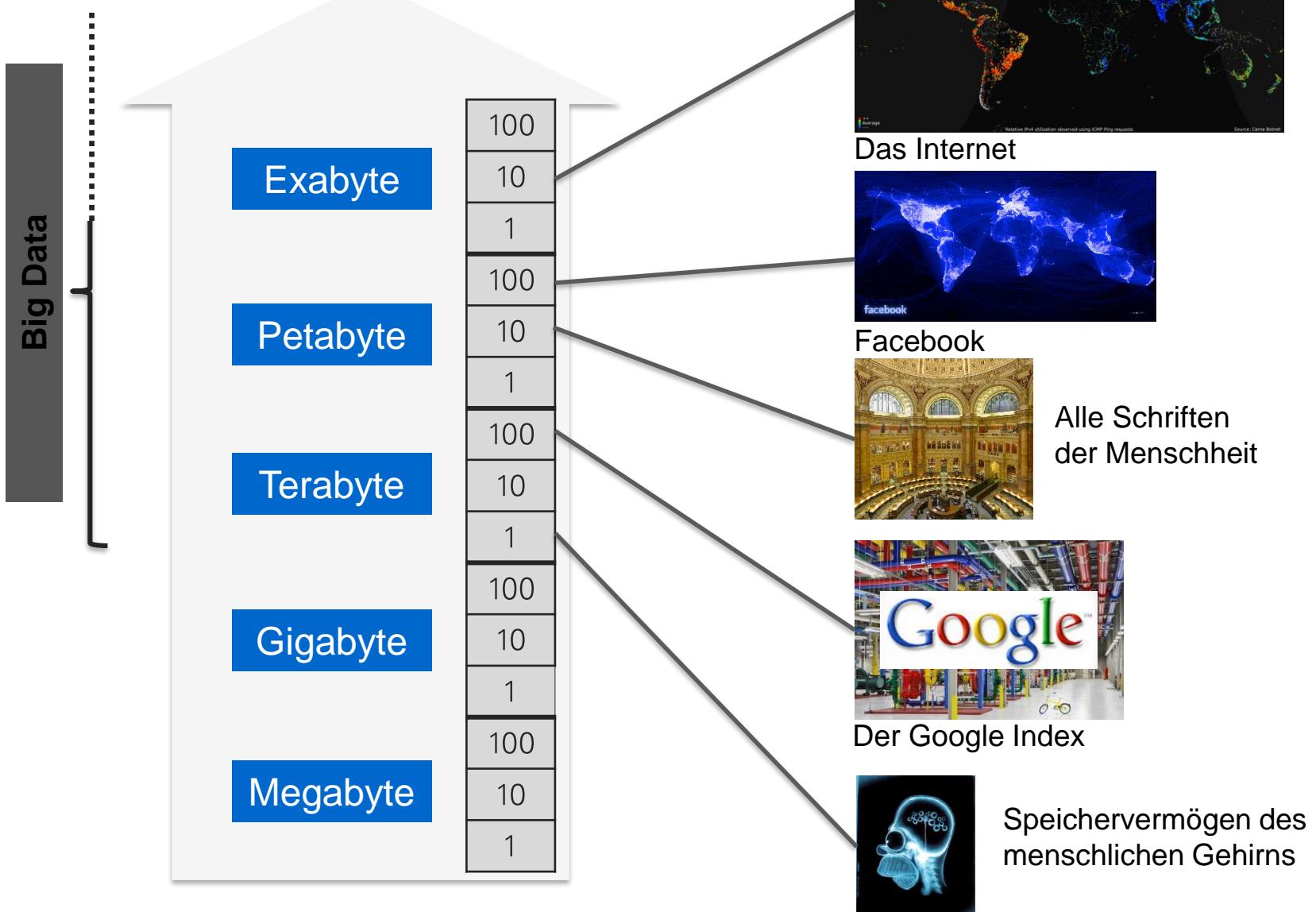
Kapitel 9: Big Data

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: $O(n \log n) \rightarrow n \times O(\log n)$

```
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));
    }
}
```

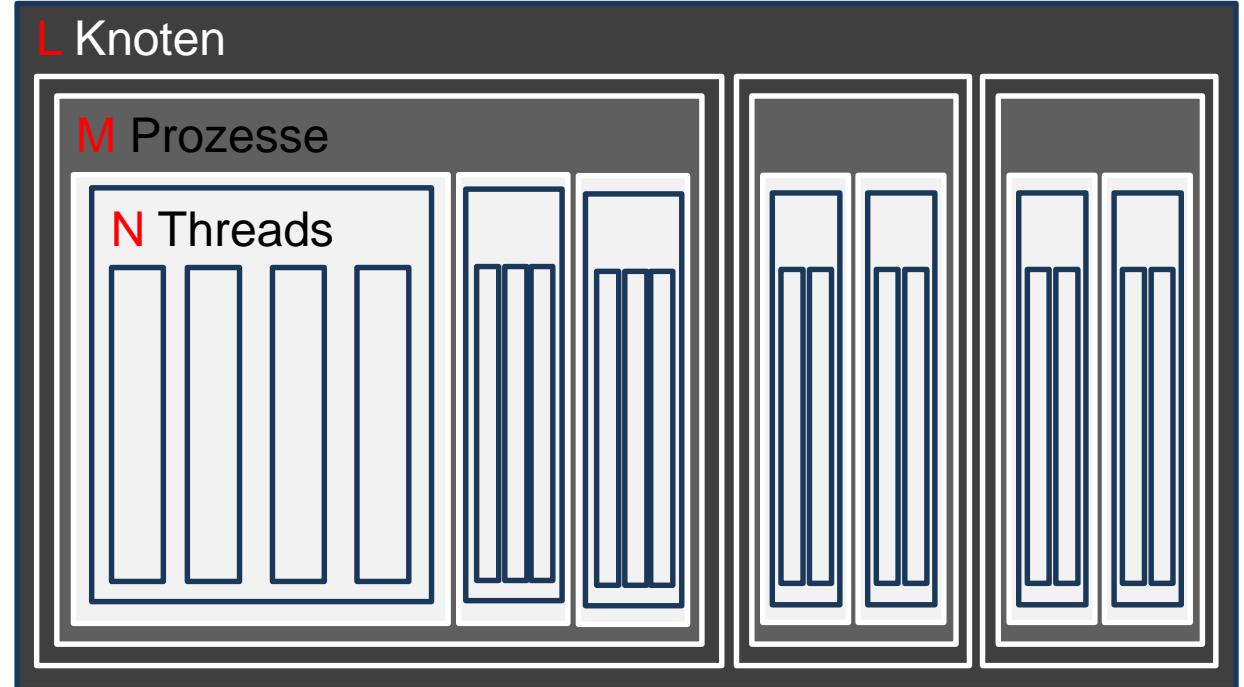
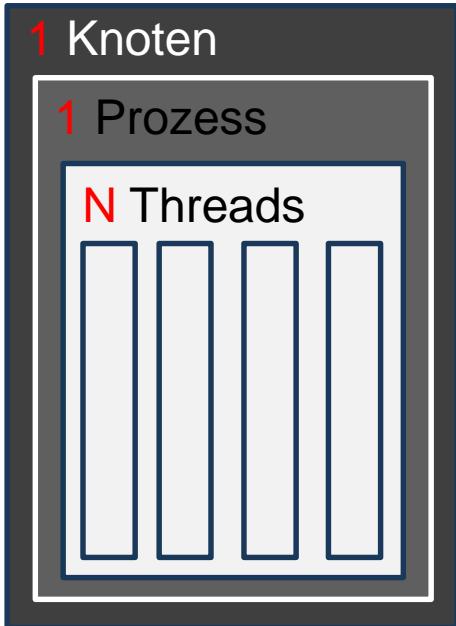
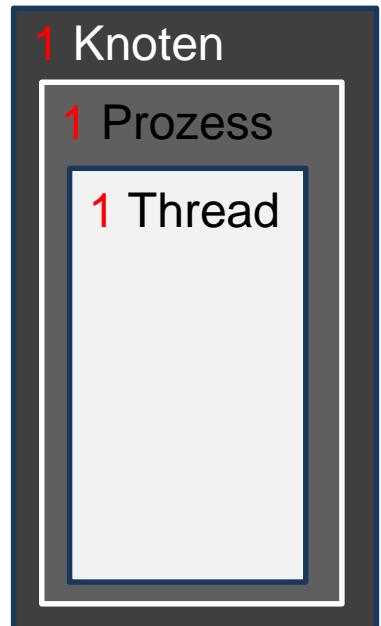
- Funktioniert nicht: Berechnung der Fibonacci-Folge ($F_{k+2} = F_k + F_{k+1}$). Berechnung ist nicht parallelisierbar.

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 basiert oft auf funktionaler Programmierung.

- Ein funktionales Programm besteht (ausschließlich) aus Funktionen.
- Eine Funktion ist die Abbildung von Eingabedaten auf Ausgabedaten:
 $f(E) \rightarrow A$
Eine Funktion ändert die Eingabedaten dabei nicht.
- Funktionen sind idempotent:
 - Sie erzeugen neben den Ausgabedaten keine weiteren Seiteneffekte.
→ Funktionen sind somit ideal parallelisierbar und zur Beschreibung von Tasks geeignet.
 - Sie erzeugen für die gleichen Eingabedaten auch stets die gleichen Ausgabedaten.
→ Funktionen können im Fehlerfall stets neu ausgeführt werden. Parallel Verarbeitung ist aus technischen Gründen oft fehleranfällig. Damit kann eine Fehlertoleranz sichergestellt werden.

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

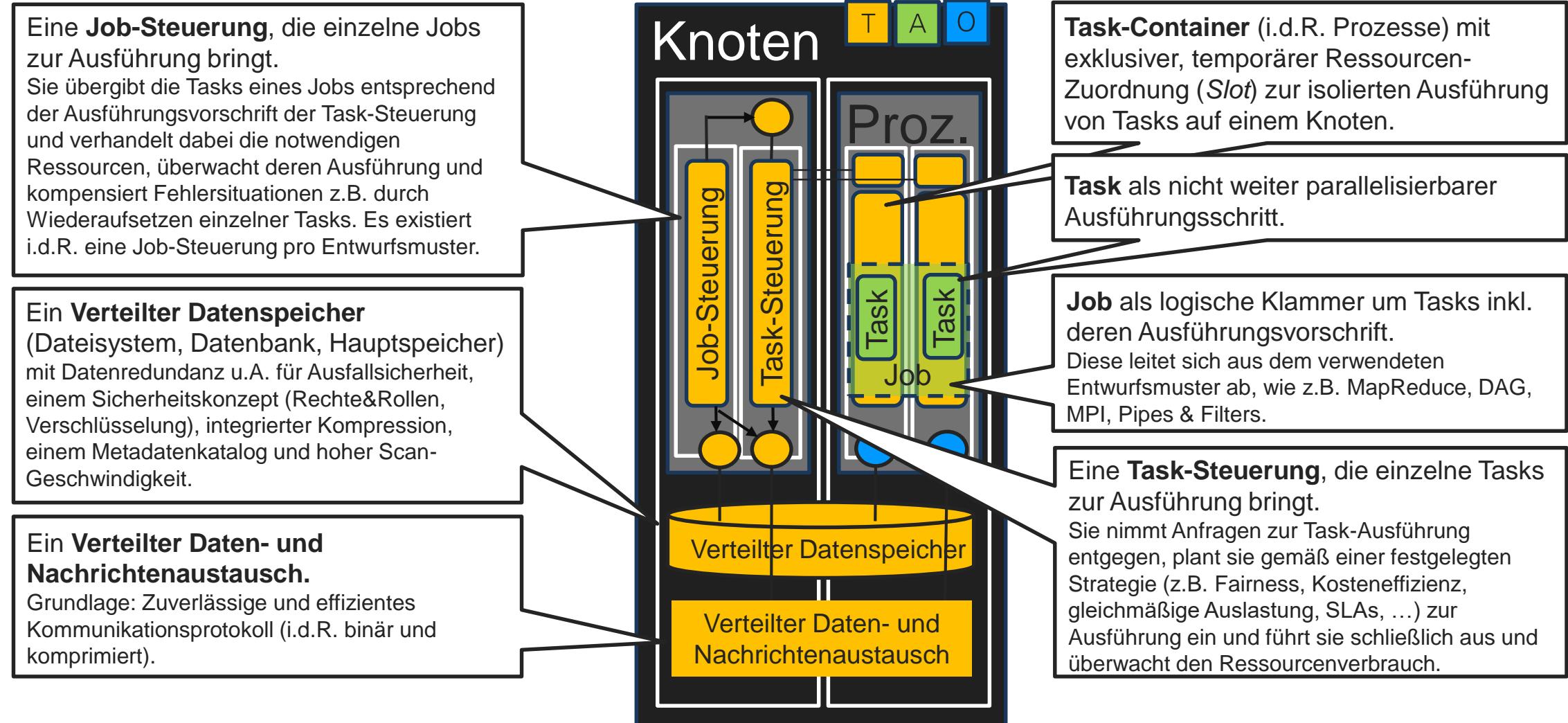
Das Grundprinzip von paralleler
Verarbeitung.

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

Eine Standardarchitektur für Parallelität im Großen



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.

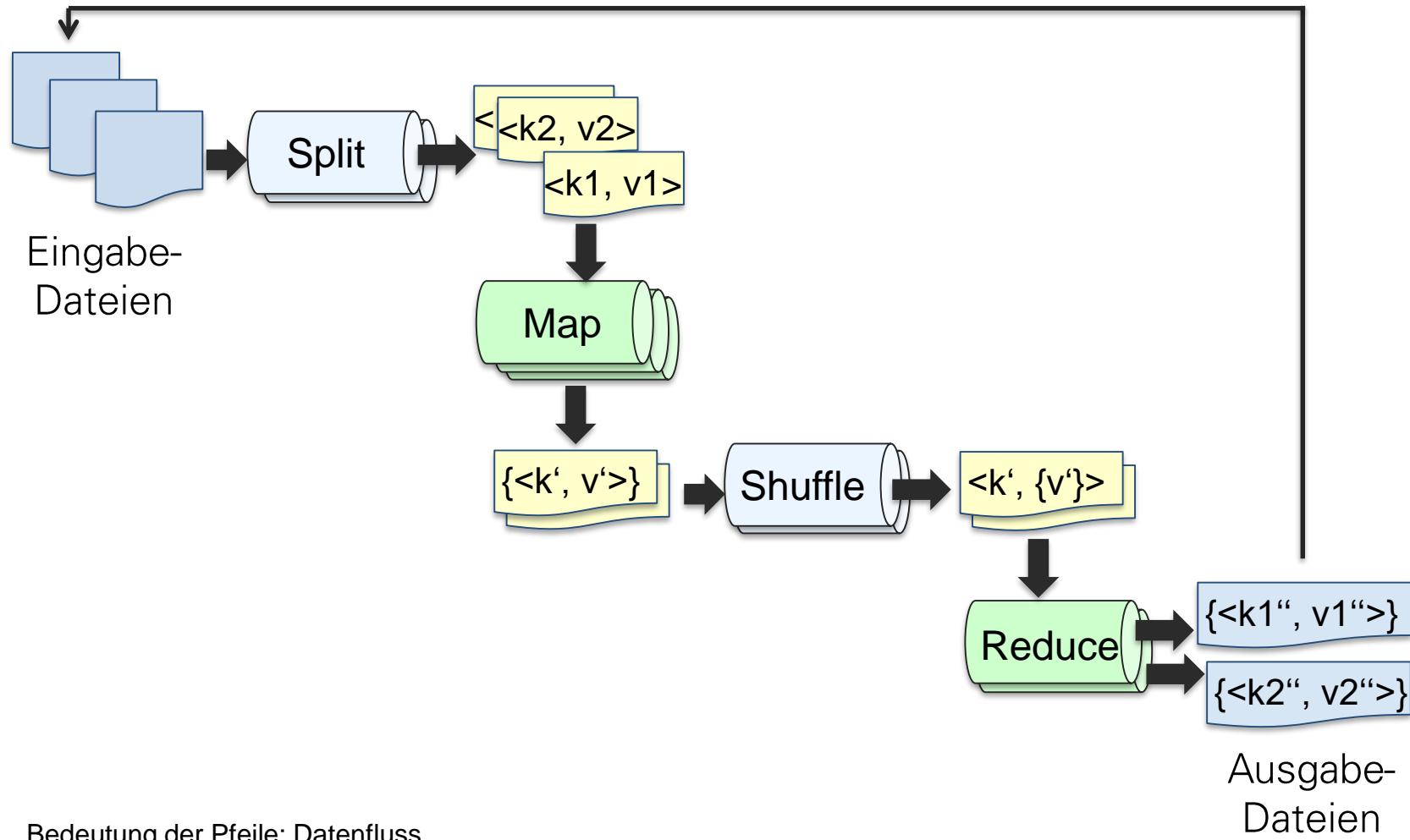
Signatur: **map**(k, v) → list(<k‘, v‘>)

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

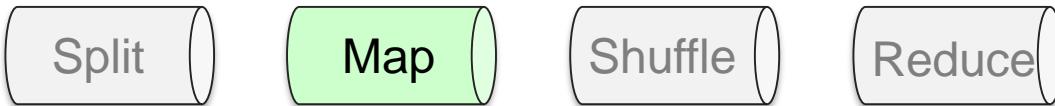
Signatur: **reduce**(k‘, list(v‘)) → list(<k‘‘, v‘‘>)

- Dabei soll gelten: |list(<k‘‘, v‘‘>)| << |list(<k‘, v‘>)|

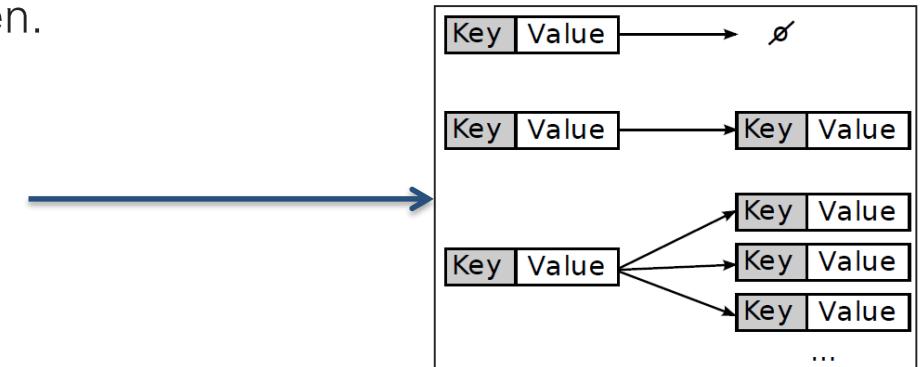
Programme werden in (mehrere) Map-Reduce-Zyklen aufgeteilt. Das Framework übernimmt die Parallelisierung.



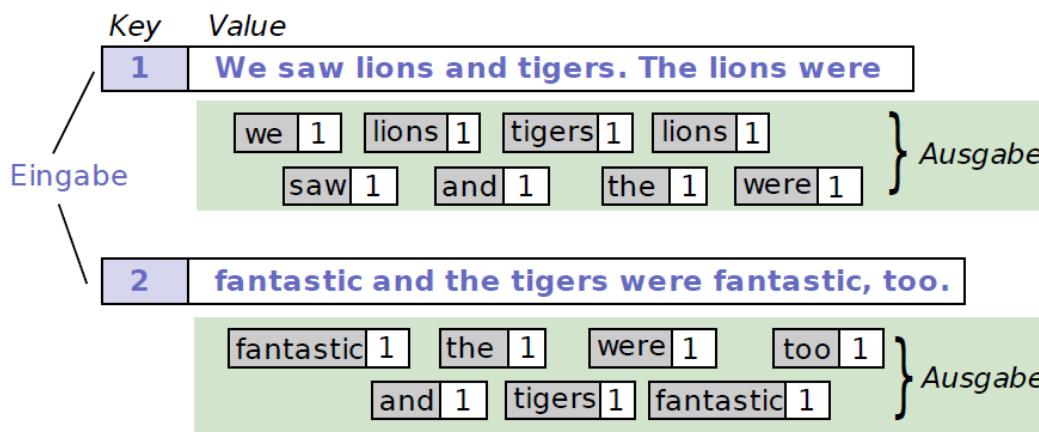
Die Map-Phase



- 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



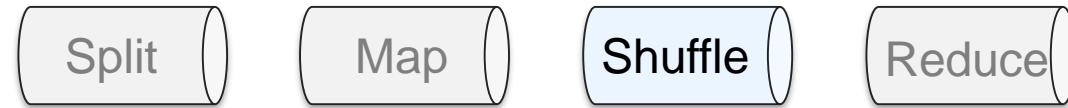
Ein- und Ausgabe der Map-Phase:



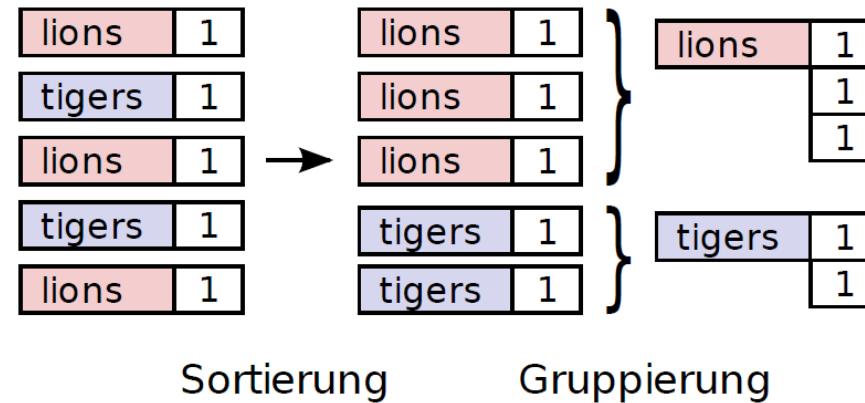
Pseudocode Map-Phase:

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

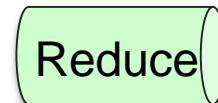
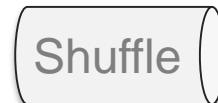
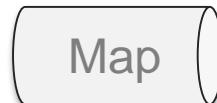
Die Shuffle-Phase



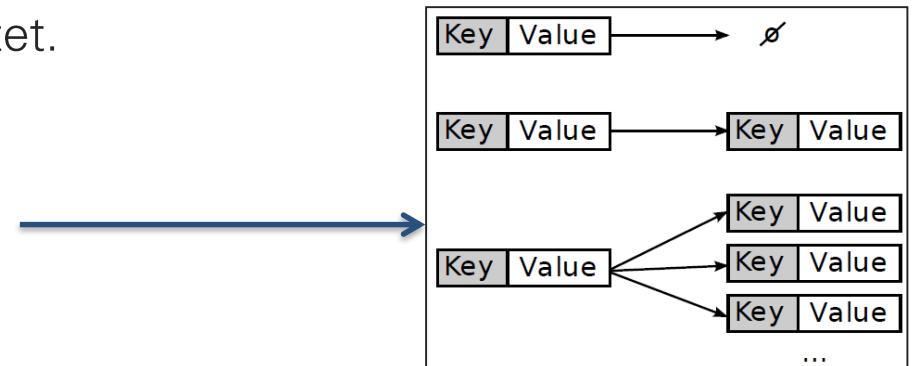
- Verarbeitung der Ergebnisse aus der Map-Phase.
- Ausgaben aus der Map-Phase werden entsprechend ihrem Schlüssel sortiert und gruppiert.
- Im Standard-Fall ist die Shuffle-Phase nicht parallelisiert.
- Sie kann jedoch mittels einer Vor-Sortierung in der Map-Phase über eine Partitionierungsfunktion (z.B.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

| | |
|-------|---|
| lions | 1 |
| | 1 |
| | 1 |
| | 1 |



| | |
|-------|---|
| lions | 4 |
|-------|---|

| | |
|--------|---|
| tigers | 1 |
| | 1 |



| | |
|--------|---|
| tigers | 2 |
|--------|---|

| | |
|------|---|
| were | 1 |
| | 1 |

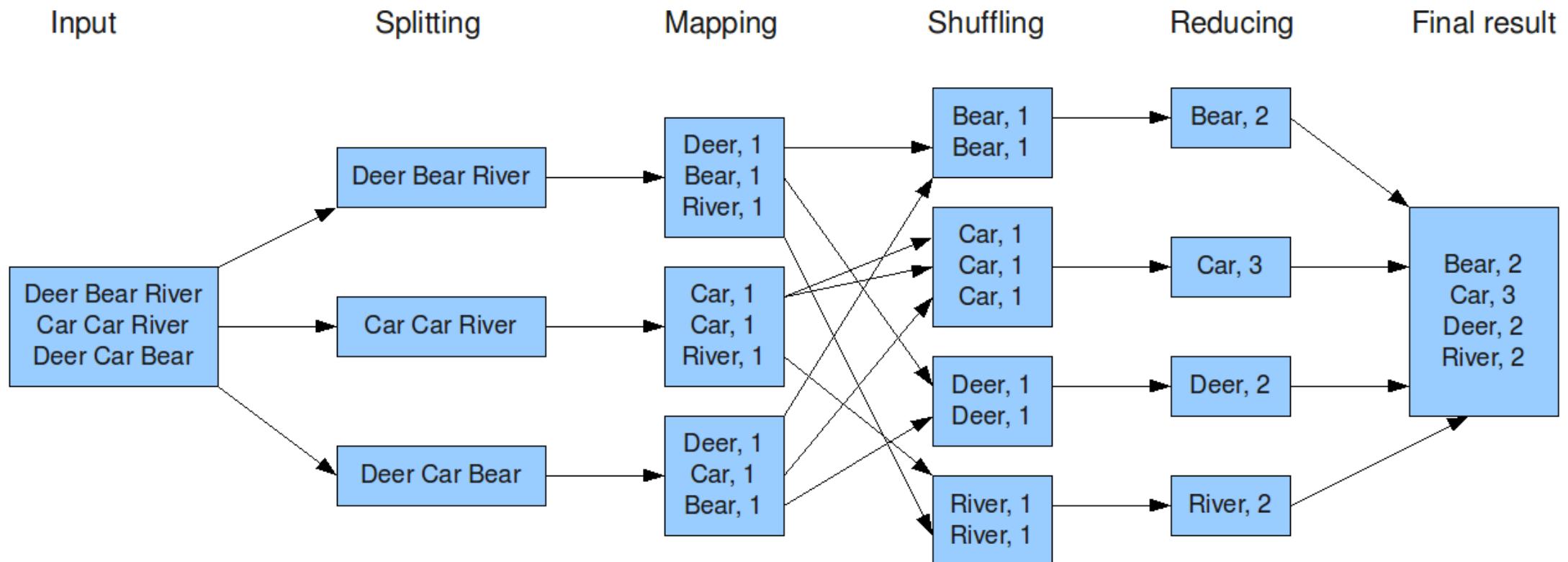


| | |
|------|---|
| were | 2 |
|------|---|

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



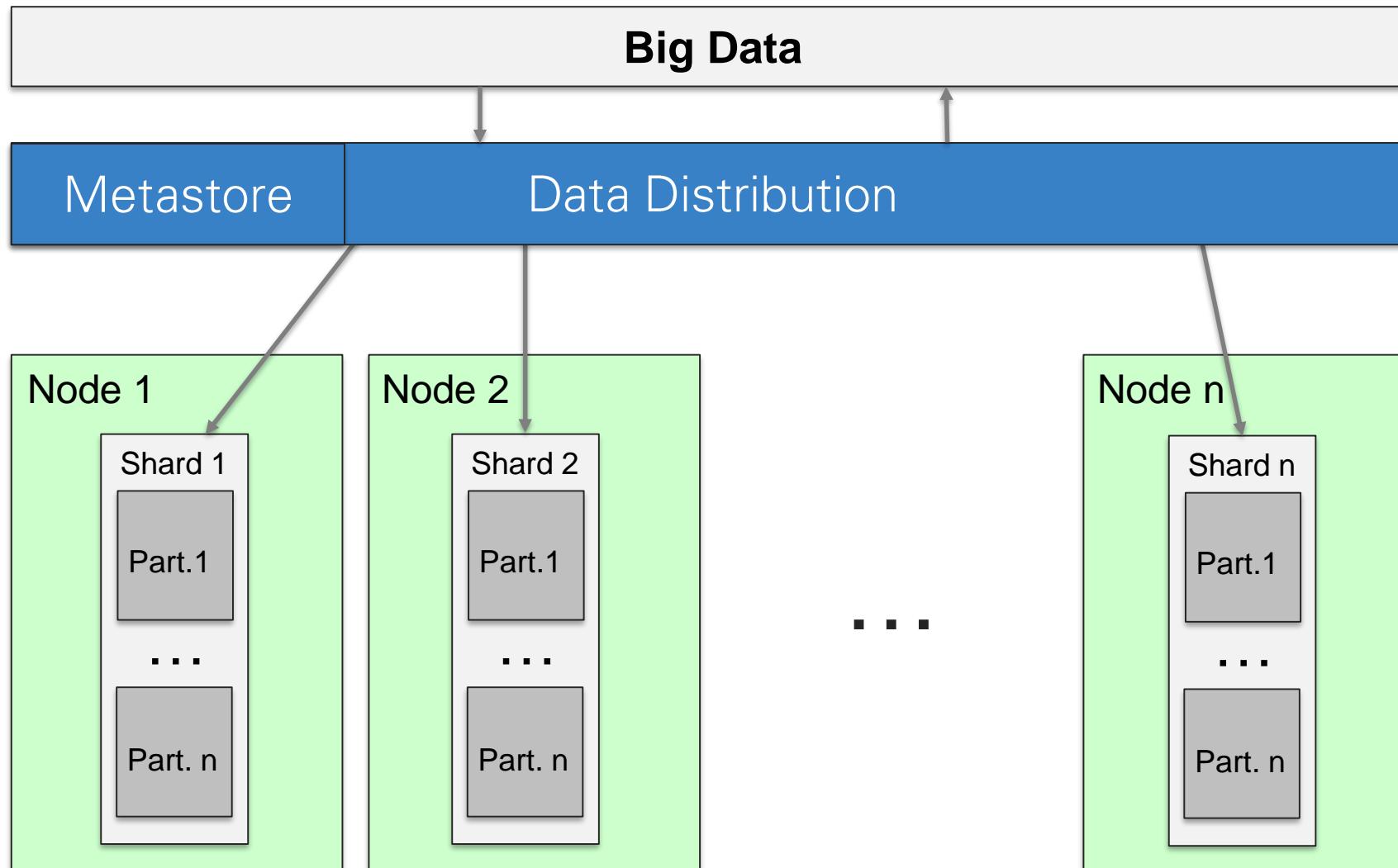
<http://blog.jteam.nl/2009/08/04/introduction-to-hadoop>

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.

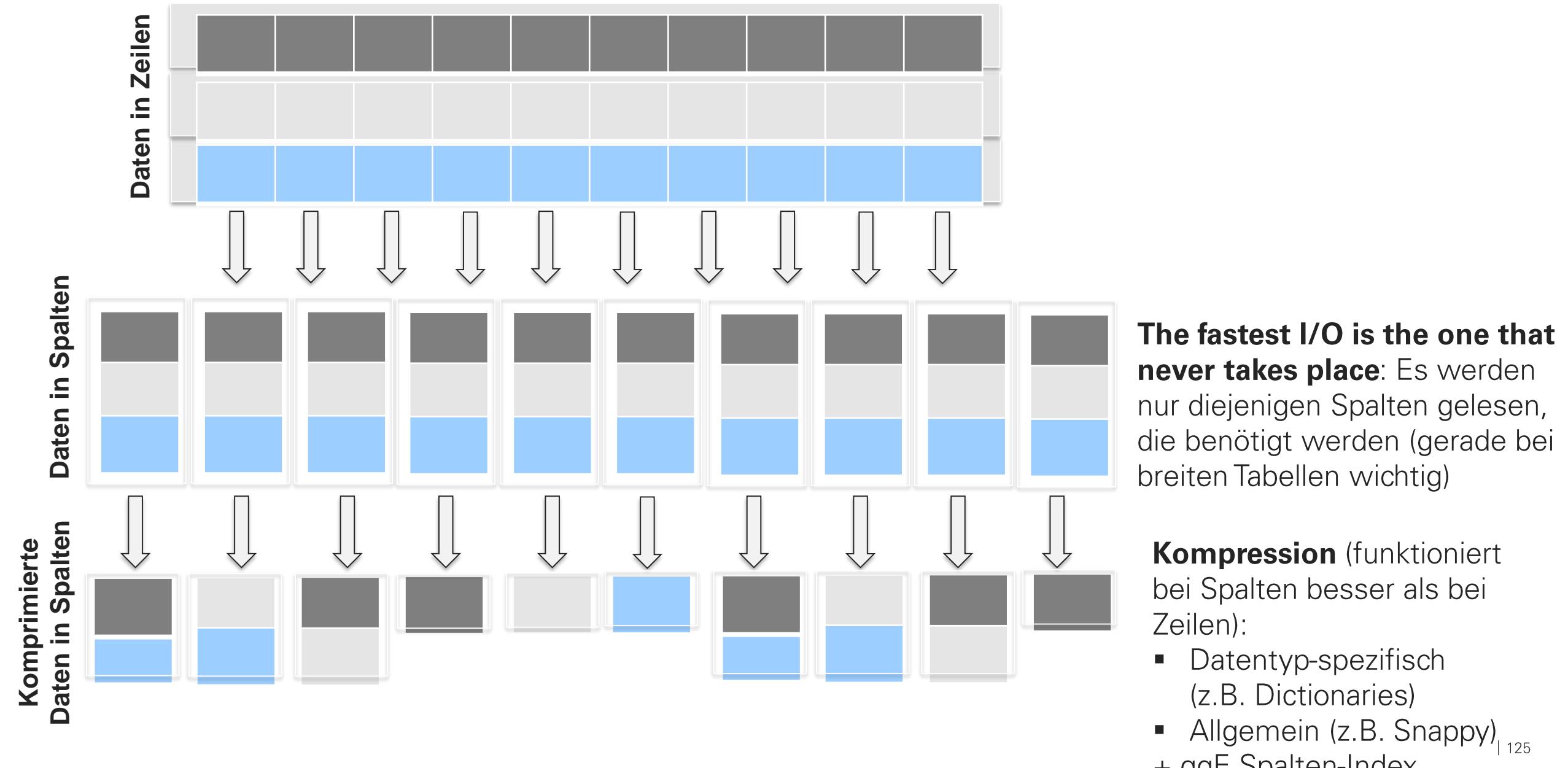


Sharding and Partitioning: Verteilung und Stückelung von großen Datenmengen.



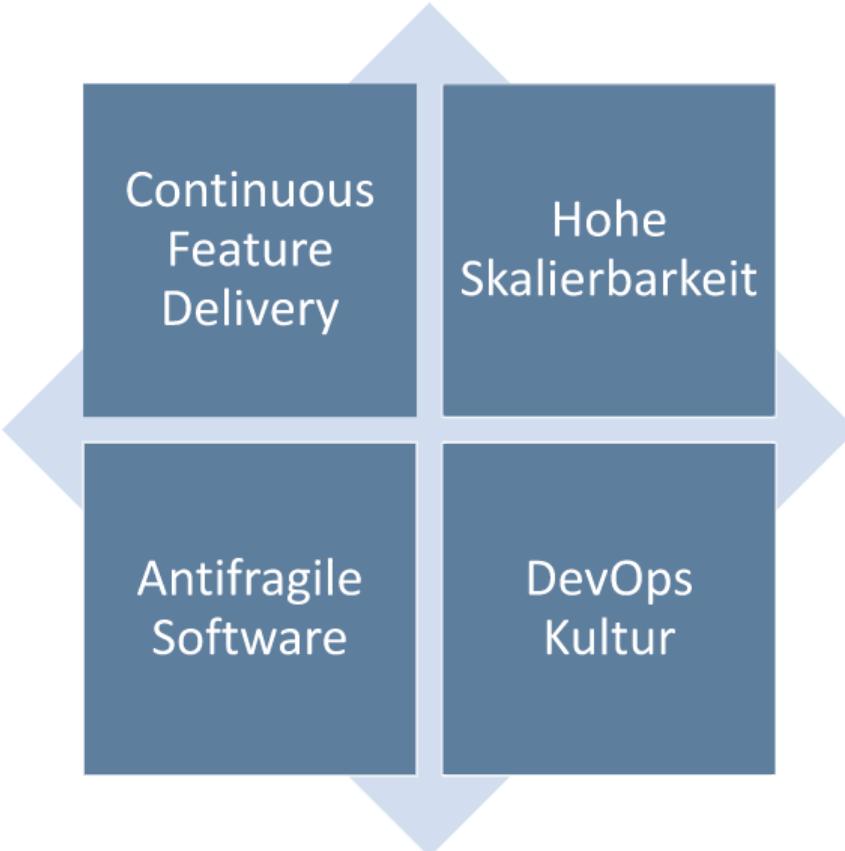
(Re-) Sharding- und
Partitioning-Funktion:
 $f(\text{Daten}) \rightarrow \text{Shard}$
 $f(\text{Daten}) \rightarrow \text{Partition.}$
+
Replikationsstrategie.
+ Konsistenzstrategie.

Spalten-orientierte Datenspeicherung.



Kapitel 10: Continuous Delivery

Treiber für Cloud-native Anwendungen



Continuous Delivery - Definition

ContinuousDelivery



Martin Fowler

30 May 2013

Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time.

martinfowler.com

Continuous delivery

From Wikipedia, the free encyclopedia

Continuous delivery (CD) is a [software engineering](#) approach in which teams produce software in short cycles, ensuring that the software can be reliably released at any time.^[1] It aims at building, testing, and releasing software faster and more frequently. The approach helps reduce the cost, time, and risk of delivering changes by allowing for more incremental updates to applications in production. A straightforward and repeatable deployment process is important for 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 inkompatibel mit anderen Änderungen ist.

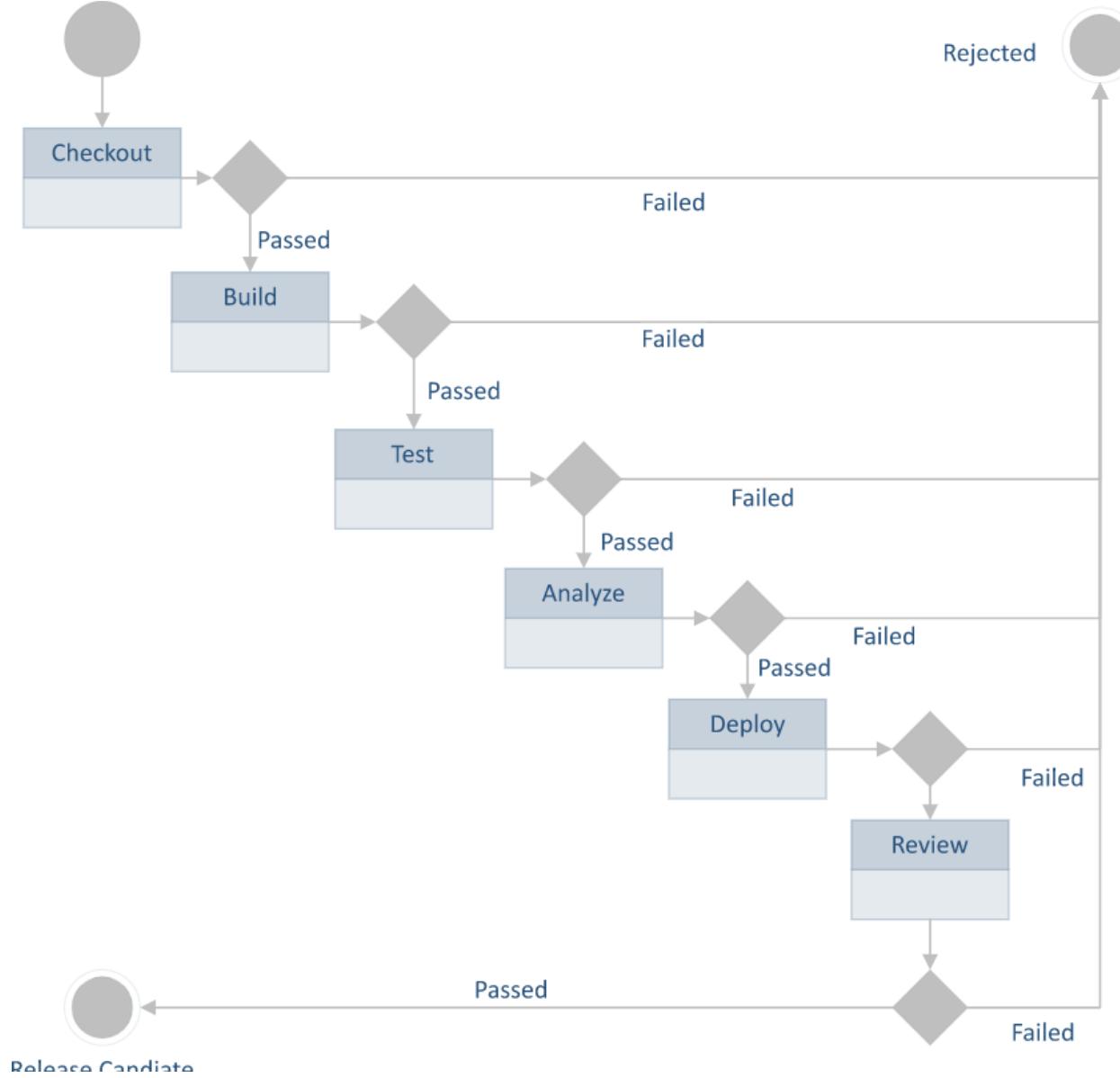
Continuous Delivery (CD)

- 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

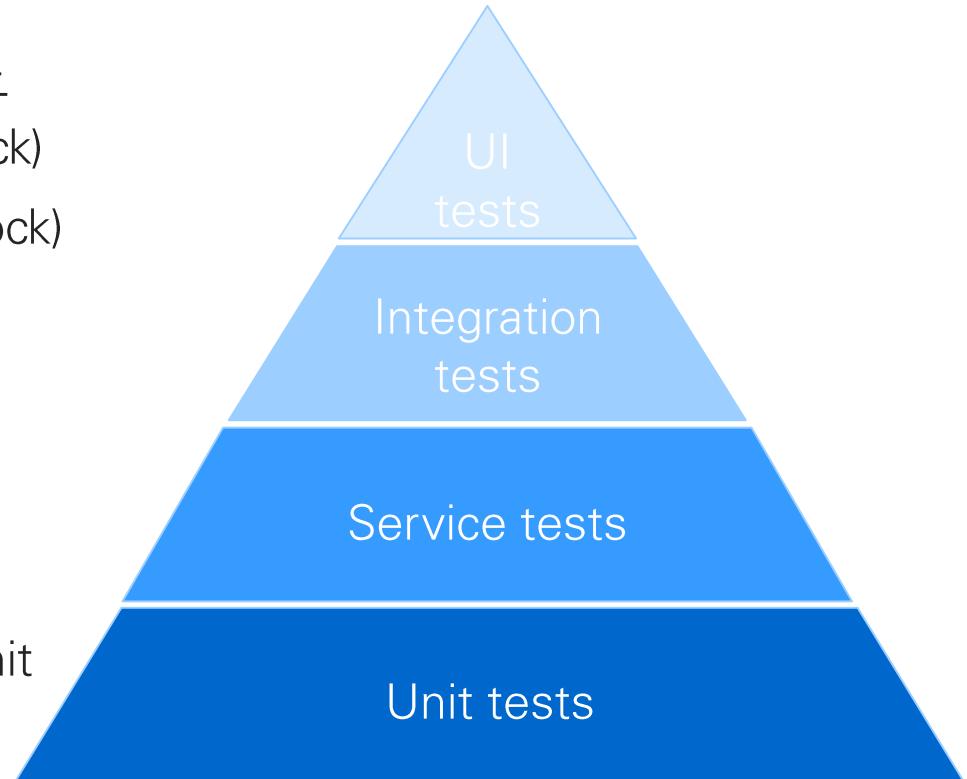
- Jede stabile Änderung wird in Produktion deployed.
- Ein Teil der Qualitätstests finden dadurch in Produktion statt.
 - → Die Möglichkeit mit Fehlern umzugehen muss vorhanden sein (z.B. Canary Release, siehe später)

Die Continuous Delivery Pipeline



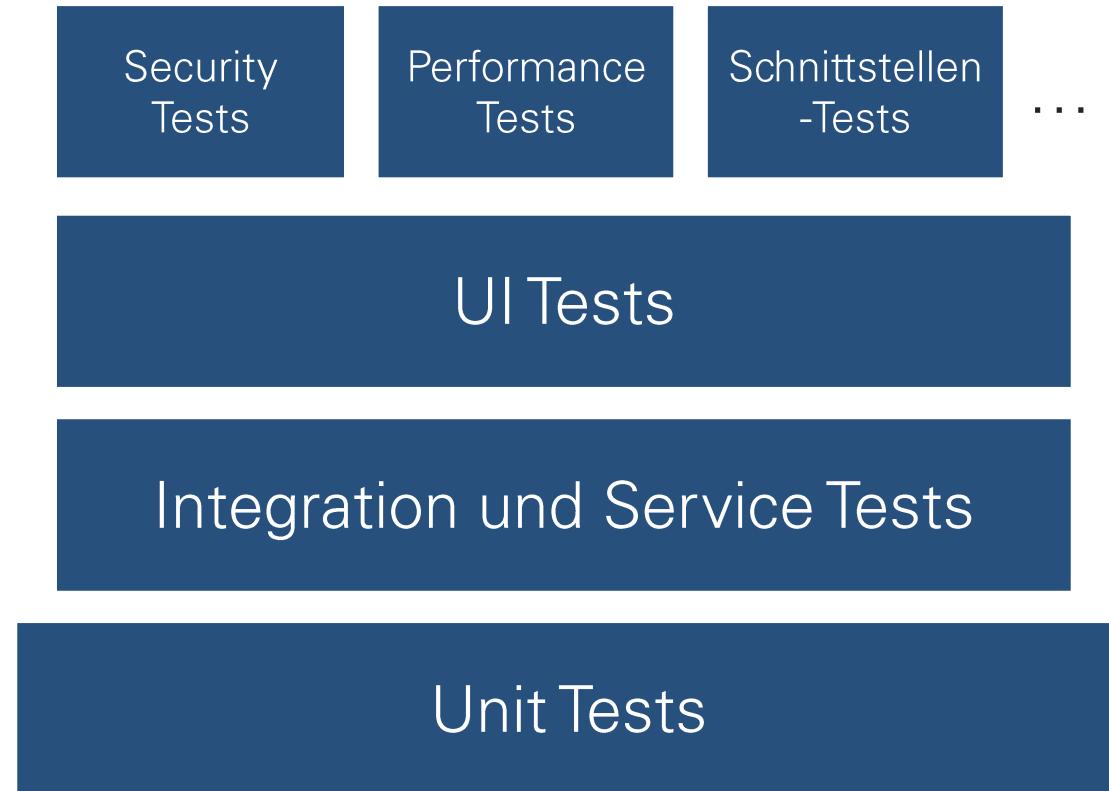
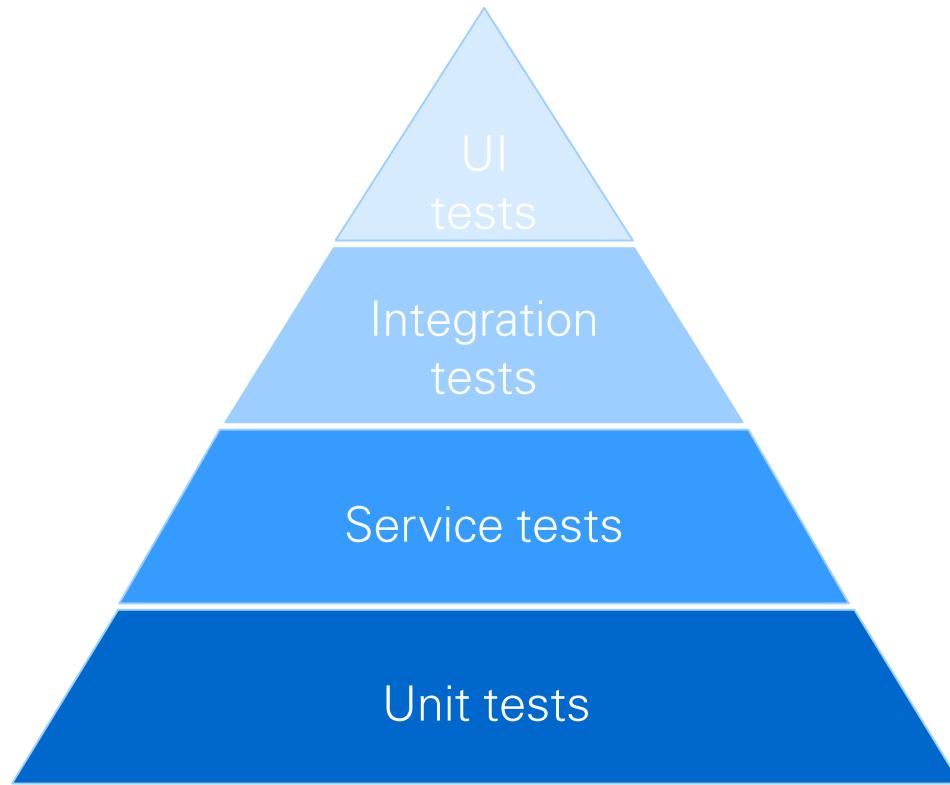
Beispiel-Aufbau einer Test-Pyramide: Sofortige Feedback bei Fehlern

- Unit Tests: Die klassischen Unit Tests (z.B. JUnit, Mockito)
- Service Tests: Tests eines einzelnen Microservices, inkl. der REST-Controller und Client-Calls (z.B. JUnit, Spring MVC Tests, Wiremock)
 - Mocks der anderen Microservices notwendig (z.B. mit Wiremock)
- Integration Tests:
 - Testet die Integration mehrerer Services und deren Interaktion (z.B. JUnit, Spring MVC Tests)
 - Performance Tests: Testet, ob es signifikante Performance-Änderungen gibt (z.B. Gatling)
- UI-Tests: Testet die UI-Funktionalität und deren Zusammenspiel mit dem Backend (z.B. Selenium, Protractor)



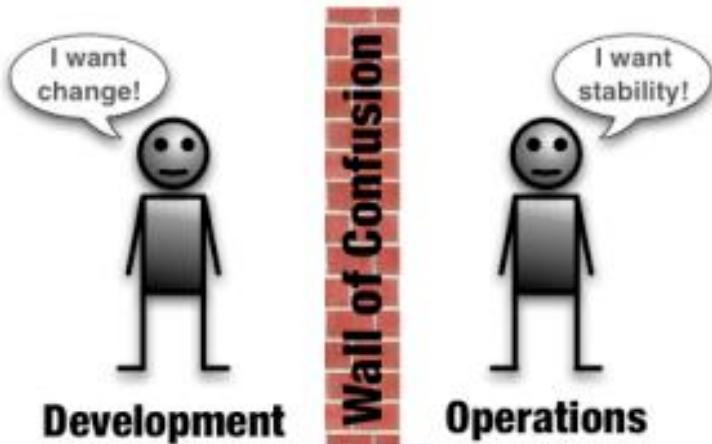
Alle Tests sollten so oft wie möglich ausgeführt werden. Idealerweise bei jedem Commit!

Alle Testebenen sind gut automatisierbar. Anstelle der klassischen Testpyramide kann die Verteilung auch so aussehen:



Was ist eigentlich DevOps?

DevOps ist die **verbesserte Integration** von **Entwicklung und Betrieb** durch mehr **Kooperation und Automation** mit dem Ziel, Änderungen schneller in Produktion zu bringen und die MTTR dort gering zu halten. DevOps ist somit eine Kultur.

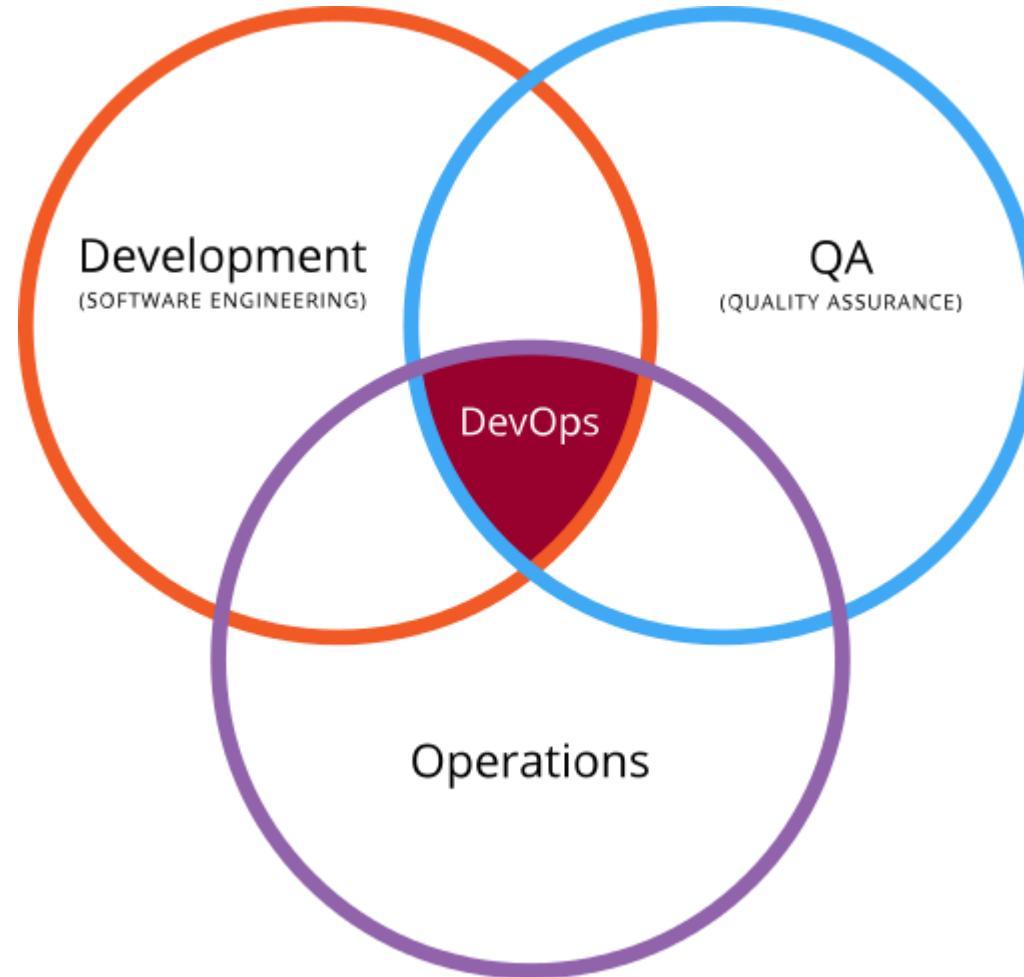


MVP + Feature-Strom

Pro Feature:

- Minimaler manueller Post-Commit-Anteil bis PROD
- Diagnostizierbarkeit des Erfolgs eines Features
- Möglichkeit Feature zu deaktivieren / zurückzurollen

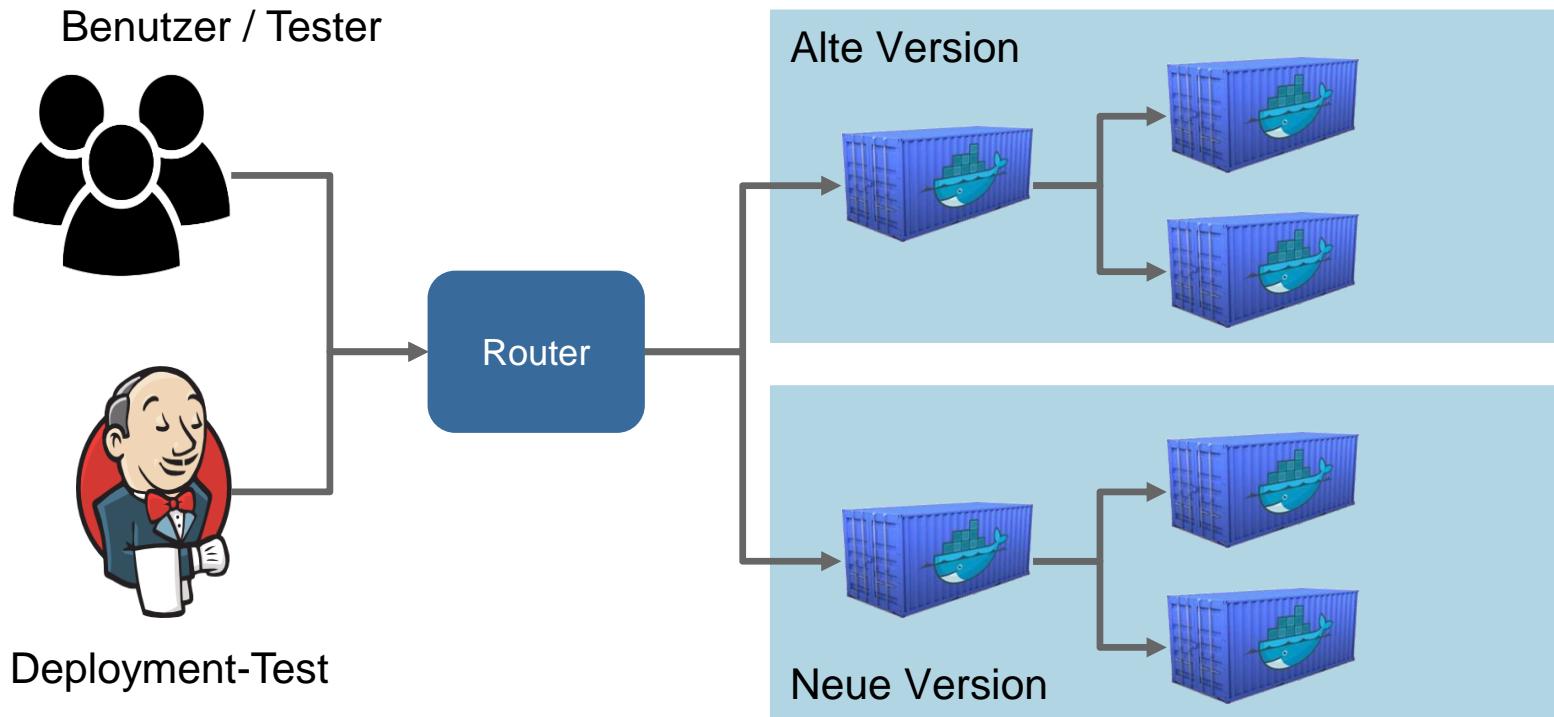
DevOps verbindet DEVelopment, OPerations und Quality Assurance



„Venn diagram showing DevOps“ by Rajiv.Pant/Wylve; Creative Commons 3.0

Wie kommen die Microservices in die Testumgebung?

- Canary-Release mit Vamp (Very Awesome Microservices Platform)
- Grundidee:
 - Neue Deployments werden nur von Testern / einem kleinen Teil der Nutzer benutzt.
 - Der Großteil der Benutzer wird erst auf die neue Version geleitet, falls sich diese als stabil erwiesen hat.



1. Die neue Version wird neben der alten Version deployed. Ein Router steuert, wer welche Version benutzt
2. Nur der Post-Deployment Test aus der Pipeline heraus wird auf die neue Version geleitet.
3. Erst danach wird die erste Teilgruppe der Benutzer / Tester auf umgeleitet.
4. Wenn keine Fehler auftreten, werden alle Benutzer umgeleitet
5. Die alte Version wird offline genommen

Kapitel 11:

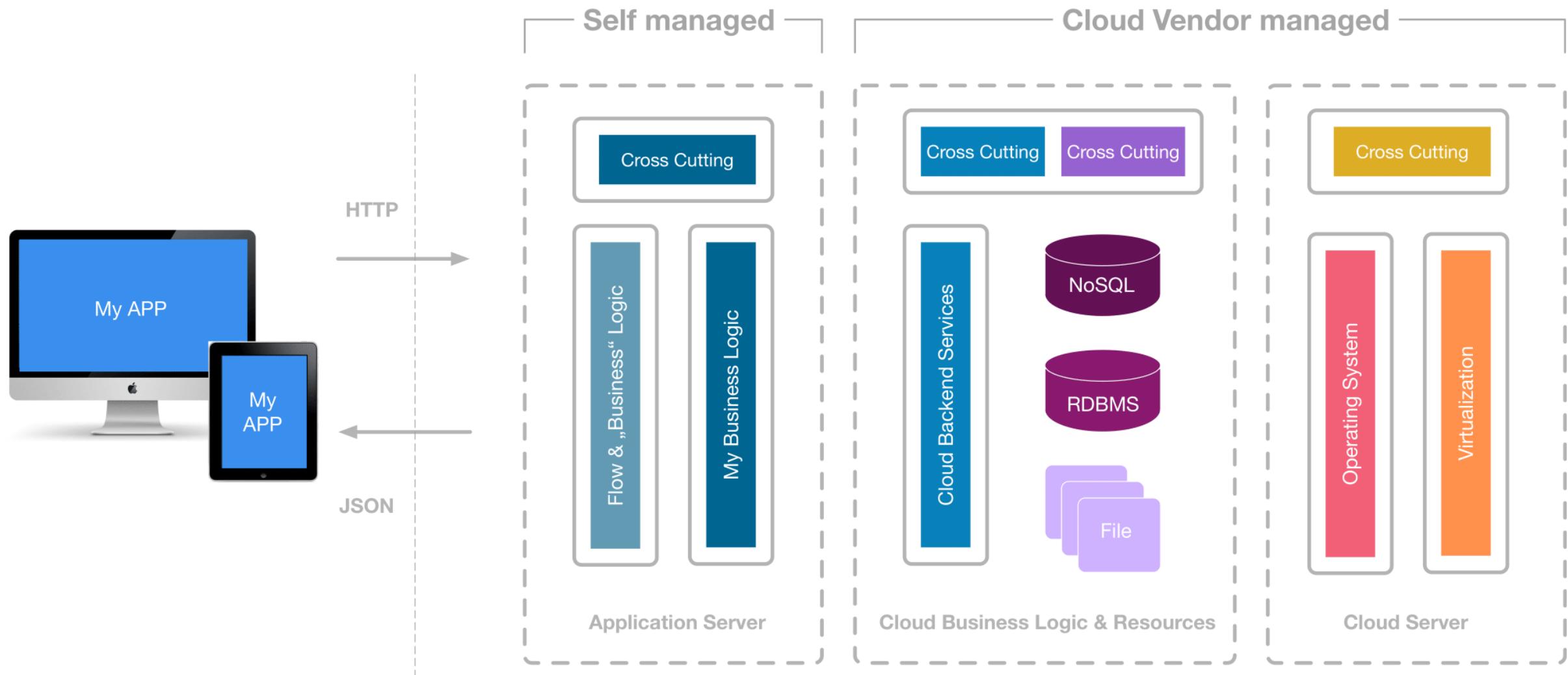
Serverless

Serverless Definition

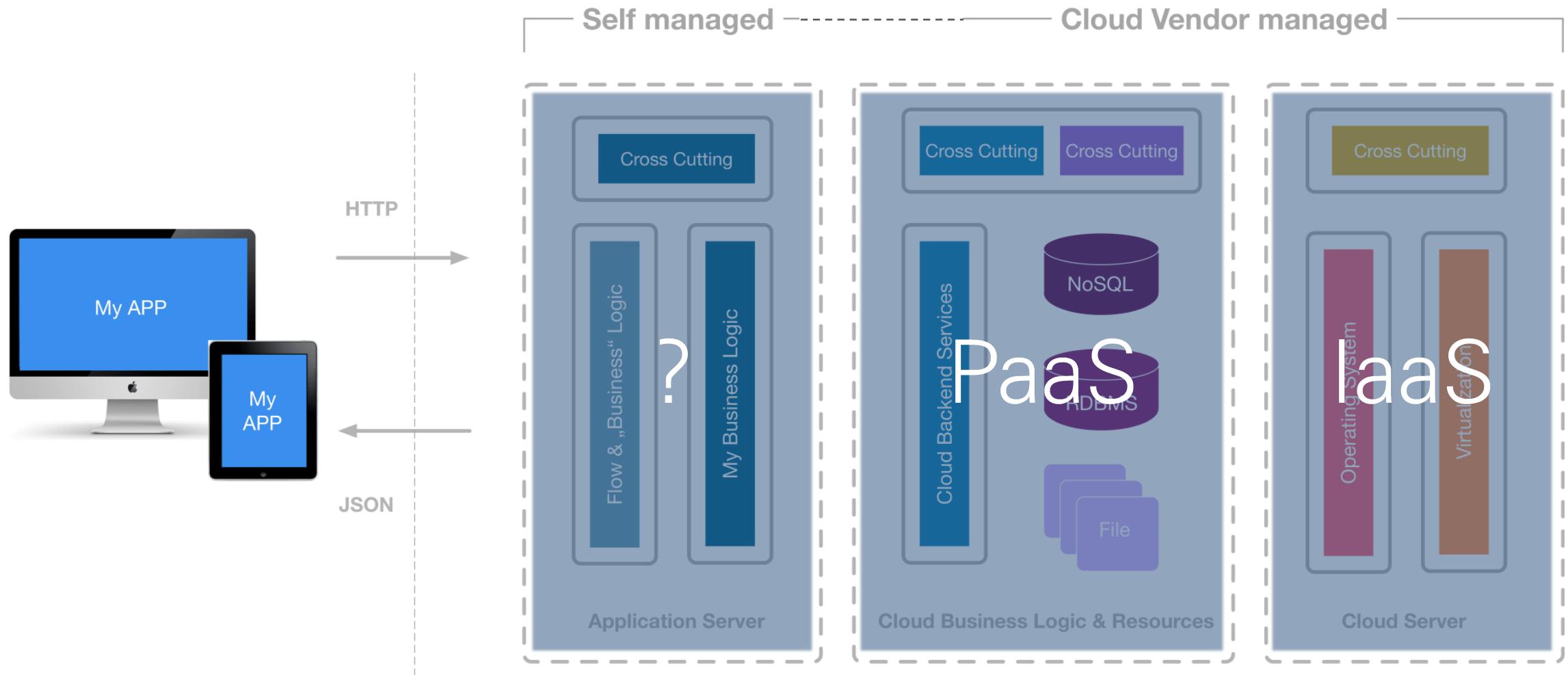
„Run your code **highly-available** in the cloud in **response to events** and scale **without any servers** to manage.“

Serverless computing refers to a new model of cloud native computing, enabled by architectures that do not require server management to build and run applications. It leverages a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment.

Traditionelle Cloud-basierte Anwendungsarchitektur



Traditionelle Cloud-basierte Anwendungsarchitektur

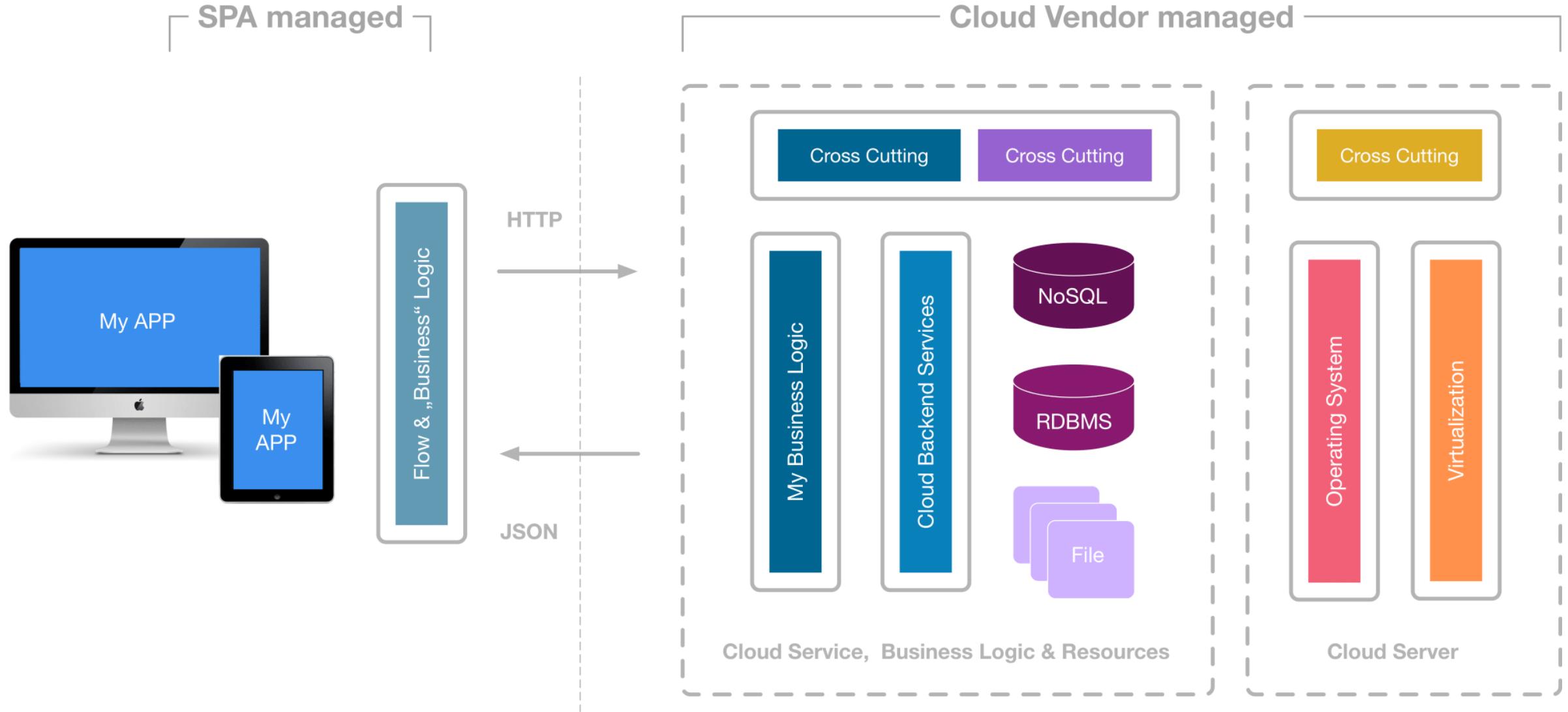


A dark blue background featuring a complex, abstract network graph composed of numerous small, semi-transparent white dots connected by thin white lines, creating a sense of data flow and connectivity.

*Kein Server ist einfacher zu
verwalten, als kein Server!*

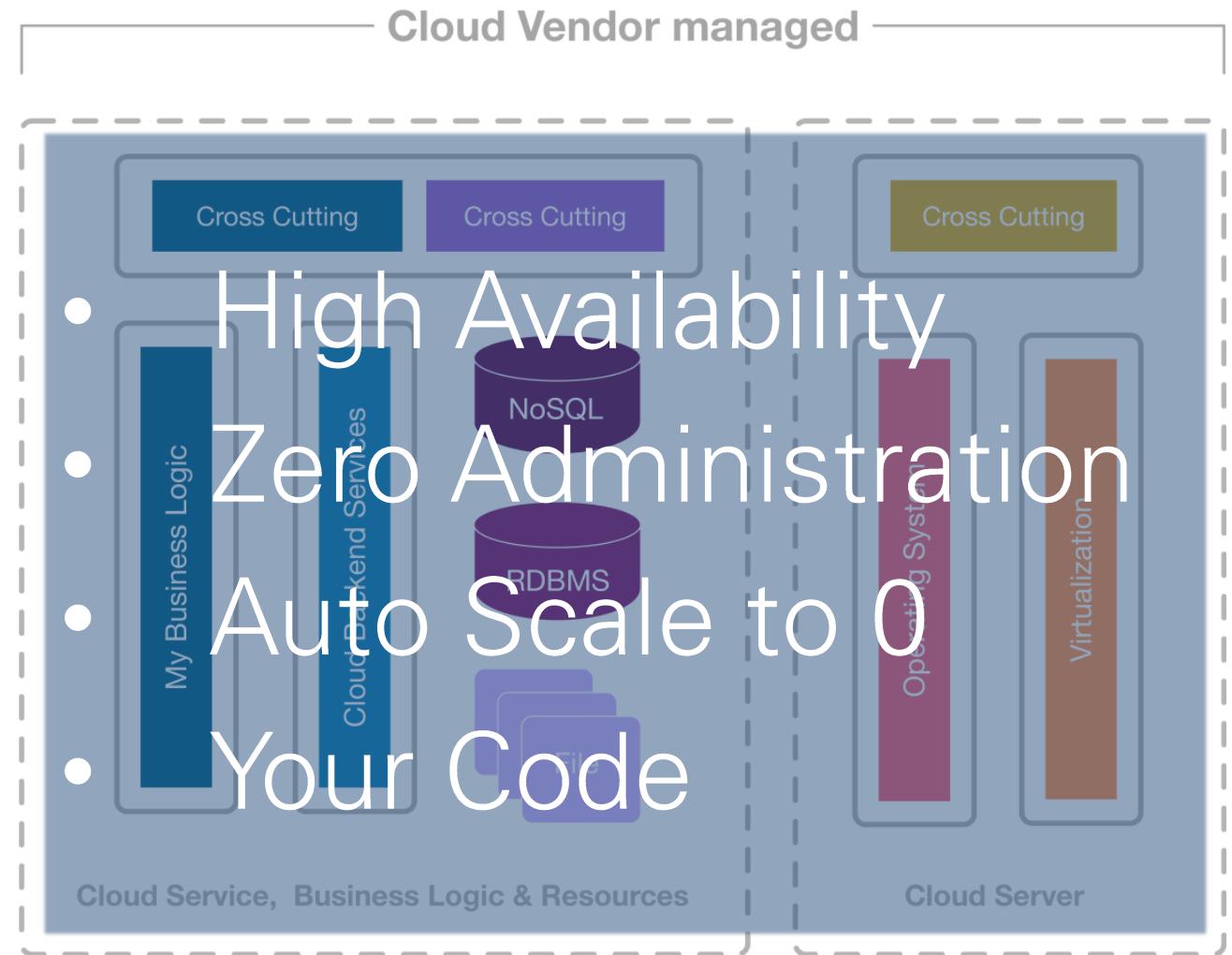
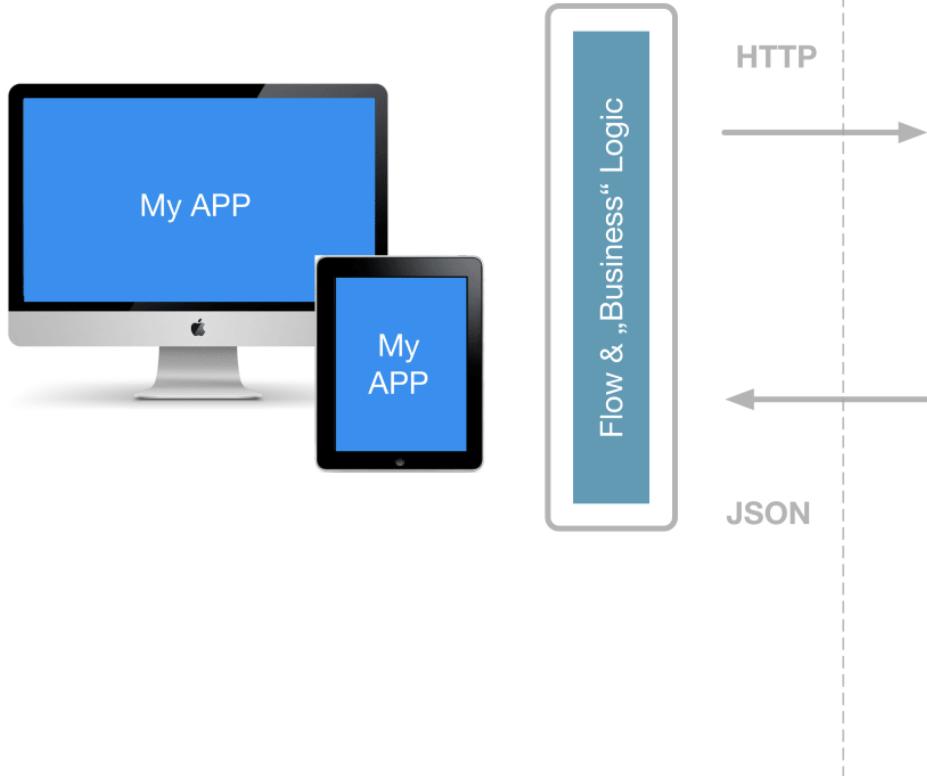
Werner Vogels, CTO, Amazon

Serverless Anwendungsarchitektur

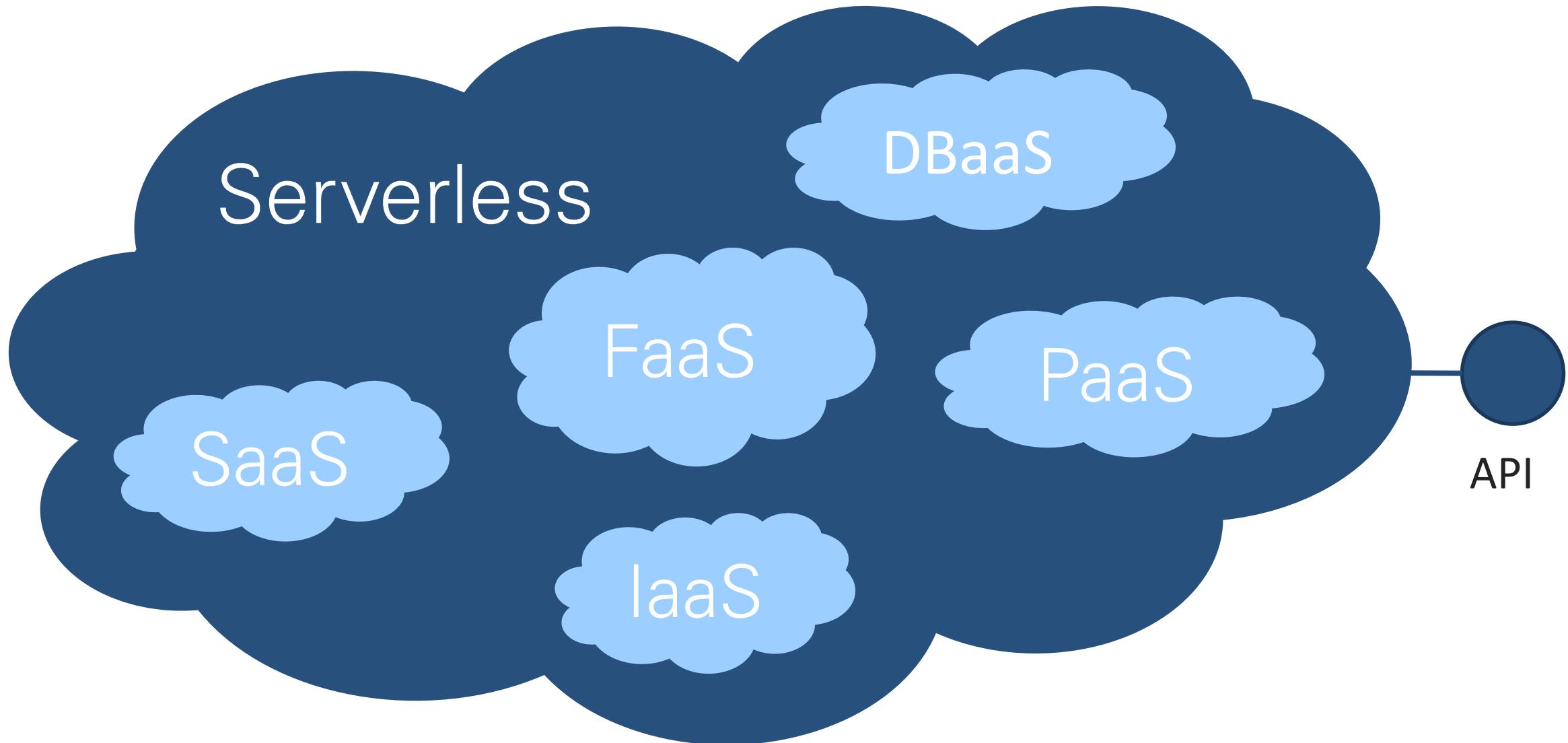


Serverless Anwendungsarchitektur

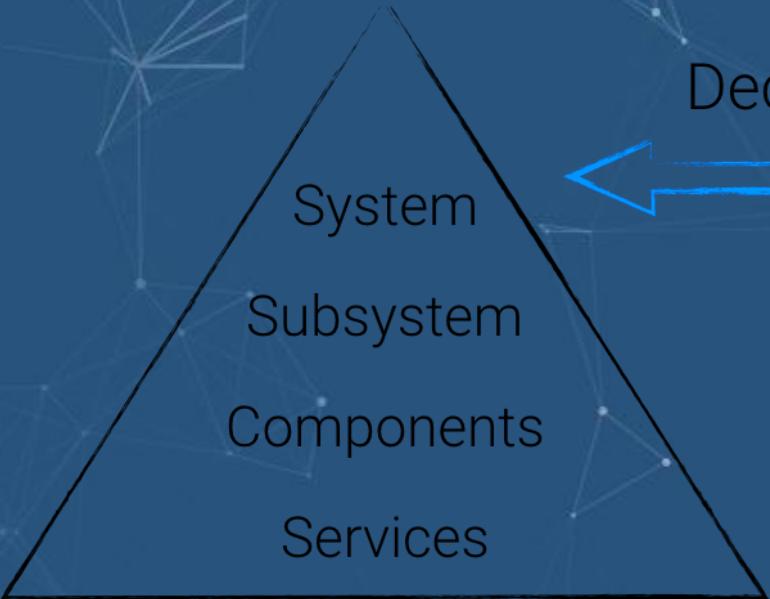
Run Code, not Servers!



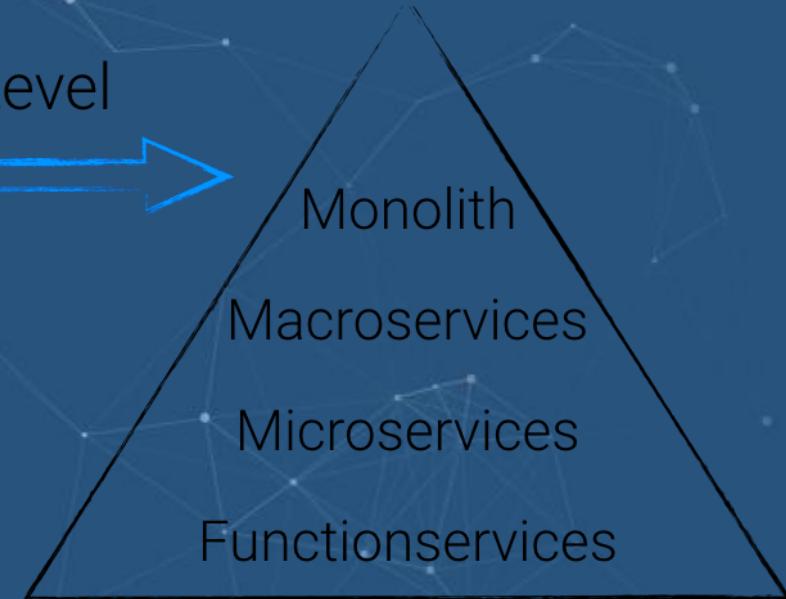
Out-of the Box Self-scaling Fully Managed Backend



Dev Components



Ops Components



Decomposition Level

? : 1



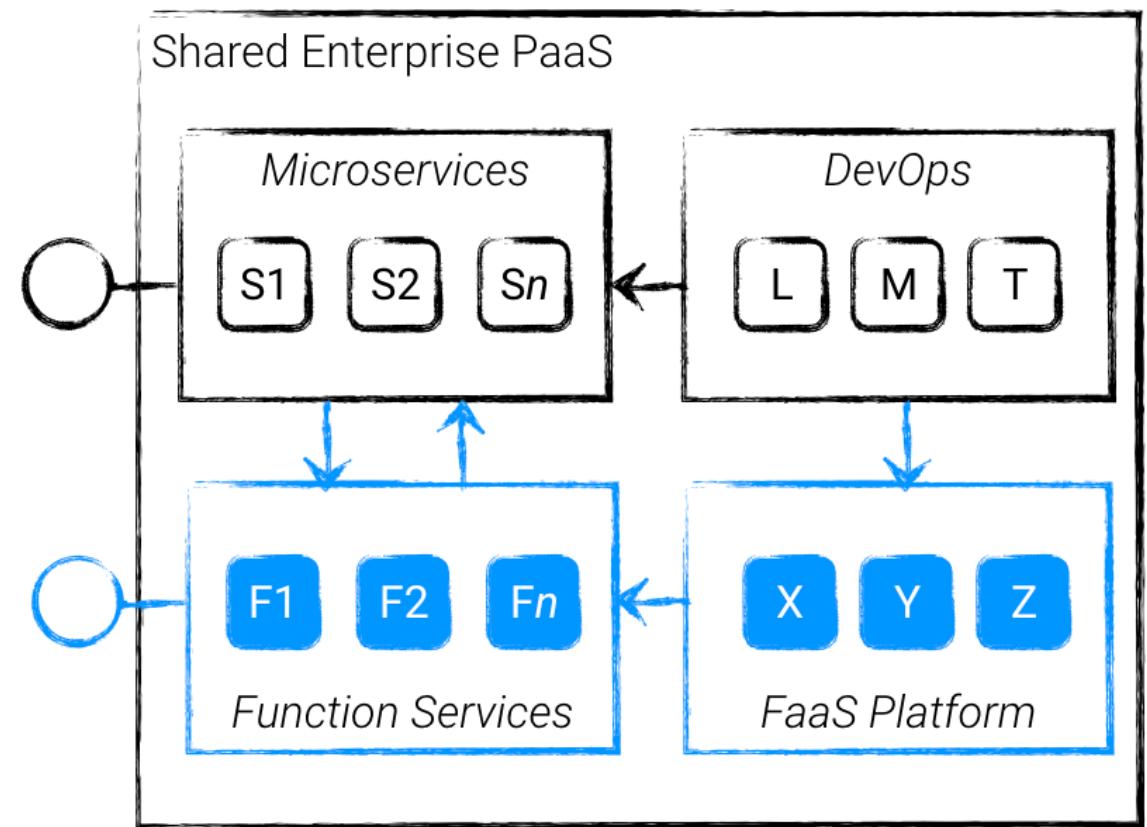
Decomposition Trade-Offs

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

Use Case 1

Hybrid Architectures

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



Use Case 2

Edge und Fog Computing

- Anbindung unserer LoRaWan Raum-Sensoren mittels Serverless Backend
 - Couch Projekt: Nutzung von FaaS auf Low Power Devices
 - Unterstützung von leichtgewichtigen Cluster Scheduler wie Docker Swarm



Die Klausur

Die Klausur.

- Prüfungszeit: 21.01.2020, 15:30 in Raum A3.10
- 90-minütige schriftliche Prüfung.
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!