



Hochschule
München
University of
Applied Sciences



QA|WARE
SOFTWARE ENGINEERING

Cloud Computing Zusammenfassung

Johannes Ebke, Felix Kampfer, Bernhard Schaidhammer, Victor Wolf

10.07.2025

Organisatorisches

Zulassungsvoraussetzung

- Das Ergebnis Ihrer ZV sollte Ihnen über Moodle mitgeteilt worden sein.

Prüfung

- Die Prüfung findet am **Freitag, den 18.07.2025** um **12.30** in Raum **R1.046** 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.

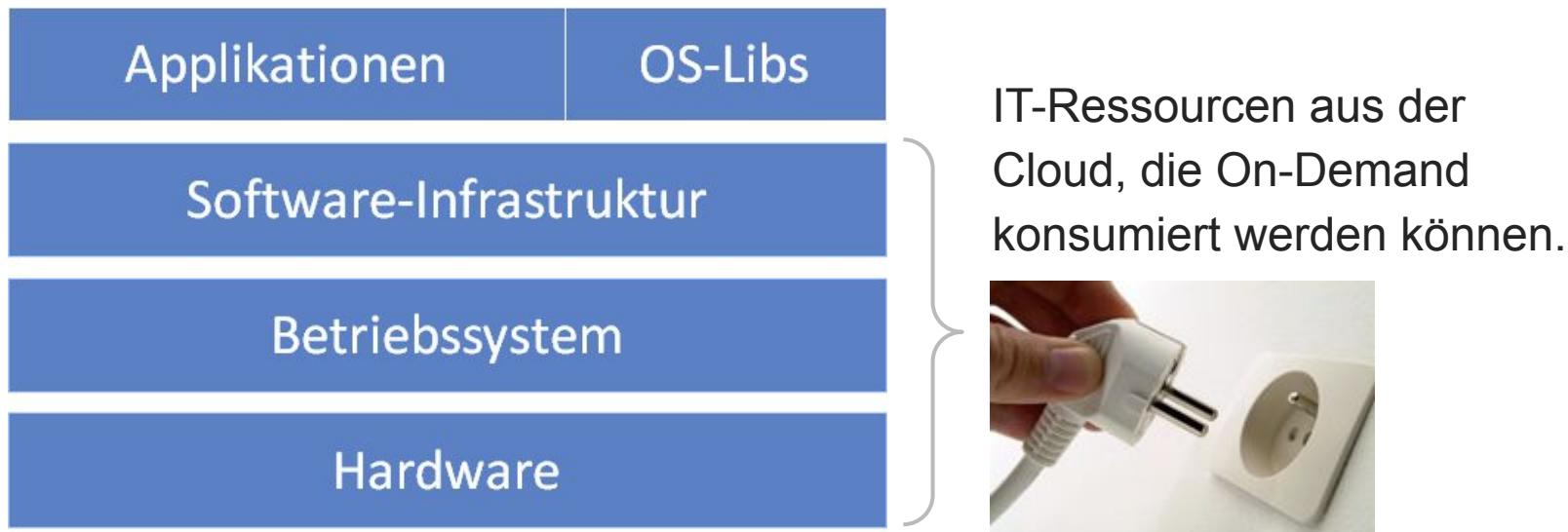
BESTE EINSTIEGSCHANCEN FÜR STUDIERENDE BEI QAware.

Erfahre mehr über deine Perspektiven: qaware.de/entwicklung



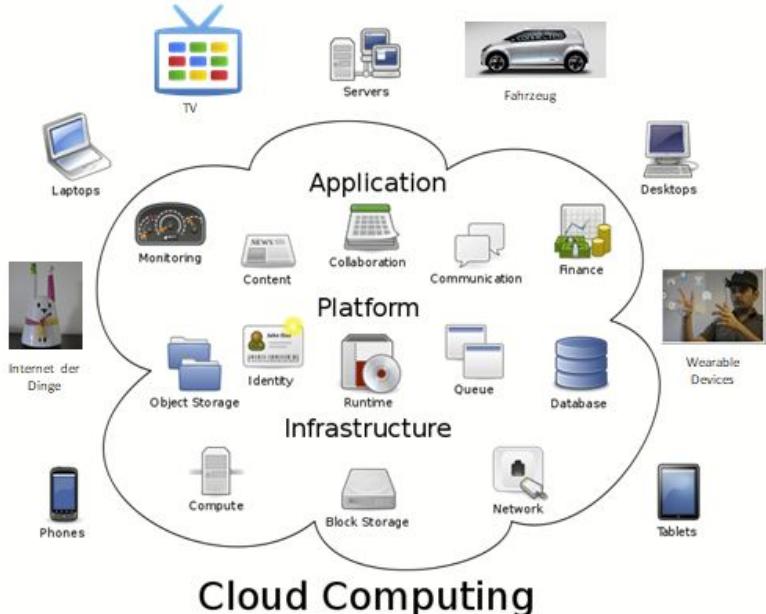
Kapitel Einführung und Grundlagen

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

Eight fallacies of distributed computing

DeveloperToArchitect.com

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



Mark Richards

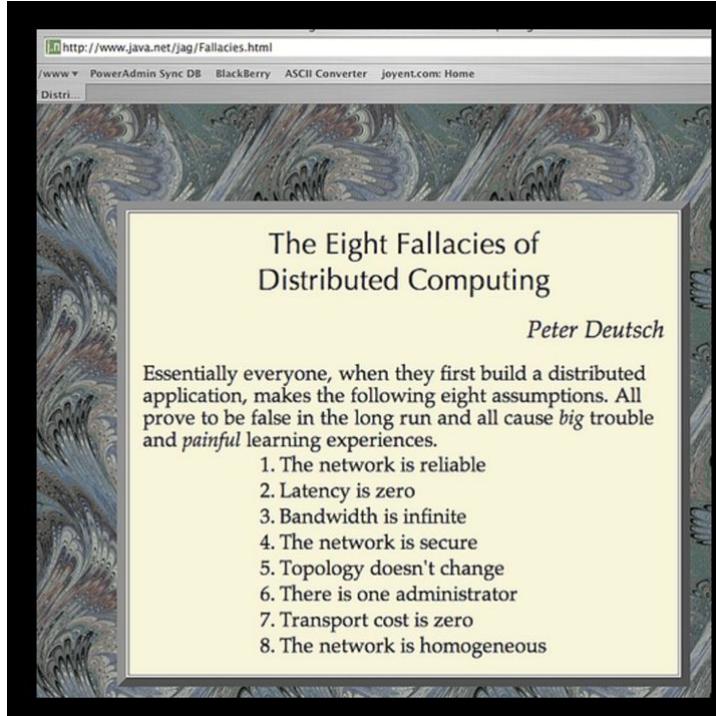
Independent Consultant

Hands-on Software Architect / Published Author / Conference Speaker

Founder, DeveloperToArchitect.com

www.wmrichards.com

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.

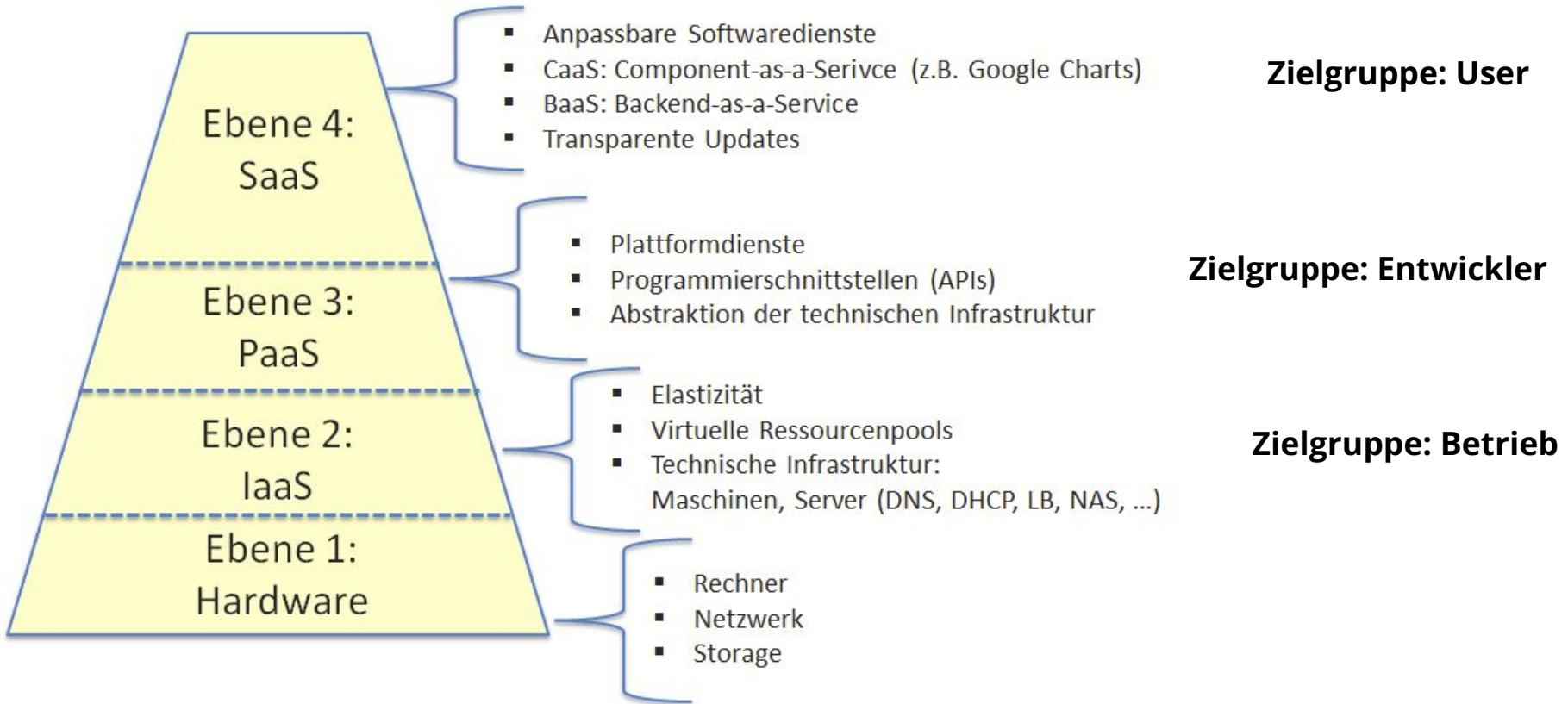
- Needed offline conversion of public domain articles from 1851-1922.
- Used Hadoop to convert scanned images to PDF
- Ran 100 Amazon EC2 instances for around 24 hours
- 4 TB of input
- 1.5 TB of output

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

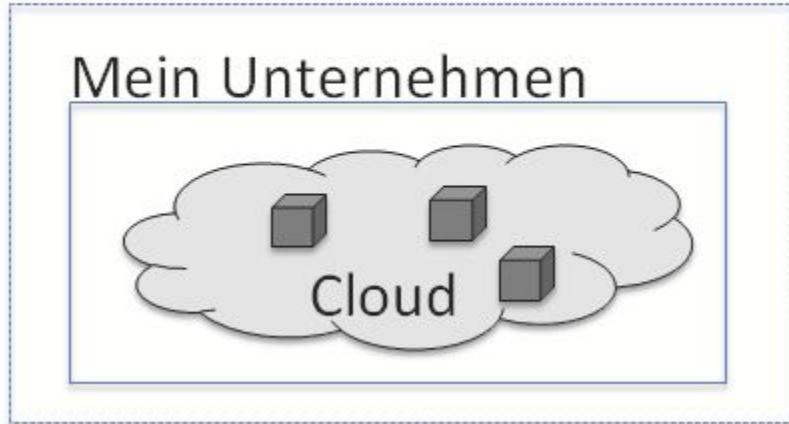
<http://www.slideshare.net/acarlos1000/hadoop-basics-presentation>

Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.

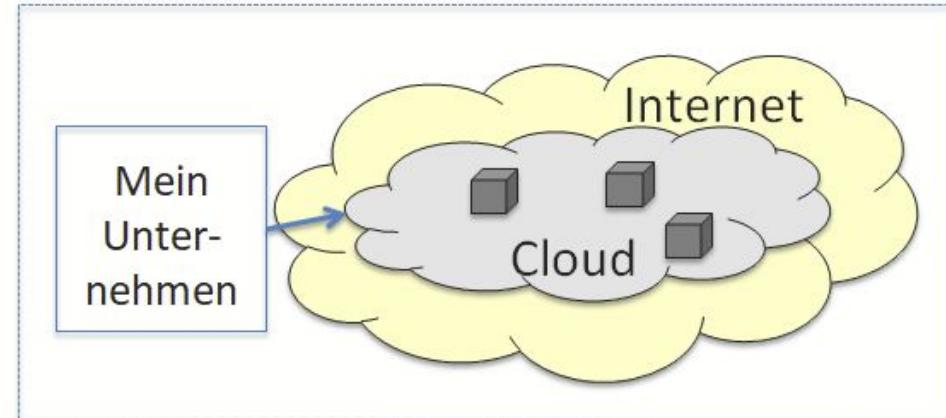


Öffentliche und private Wolken.

Private Cloud:

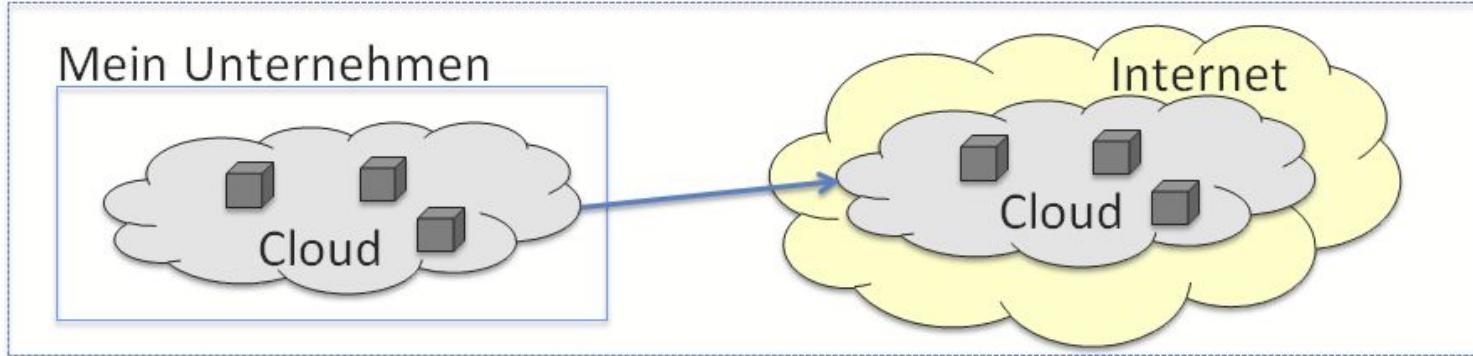


Public Cloud:

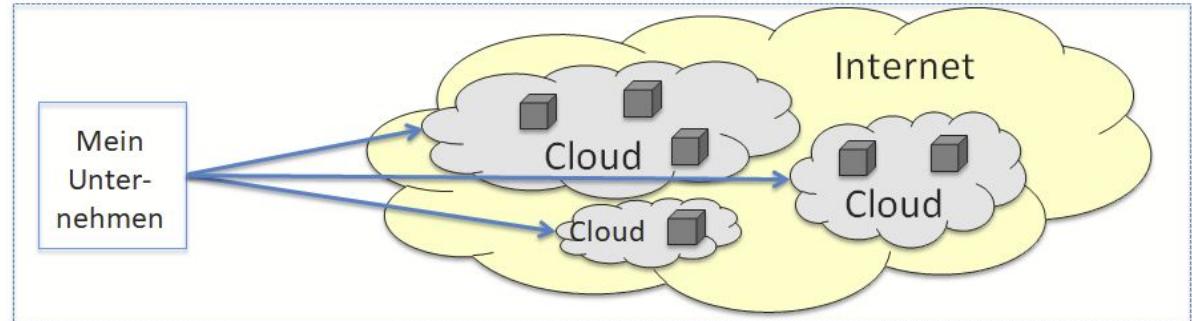


Hybride und multiple Wolken.

Hybrid Cloud:



Multi-Cloud:



Kapitel Kommunikation

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

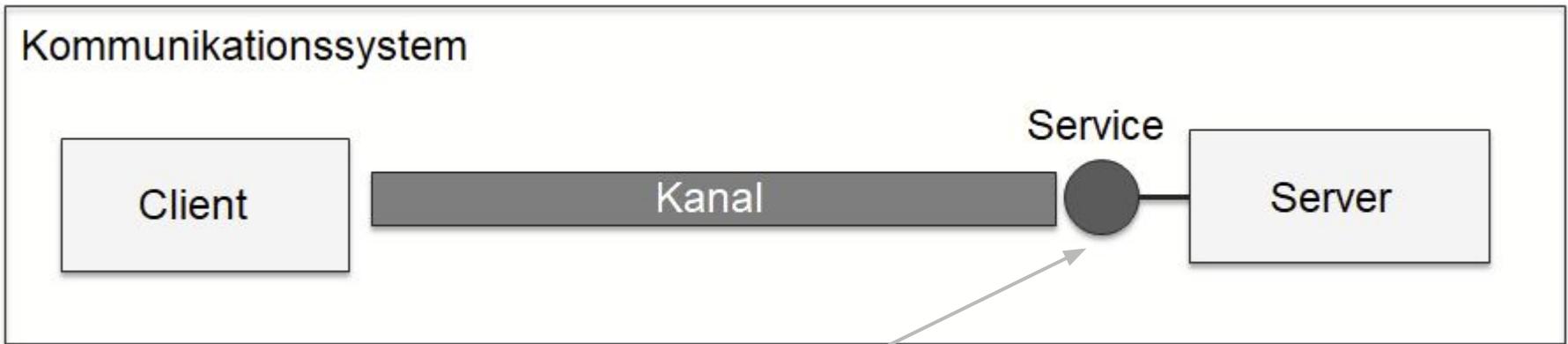
Kommunikationssystem = Infrastruktur für die Übermittlung von Informationen.



Typische Kanaleigenschaften:

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

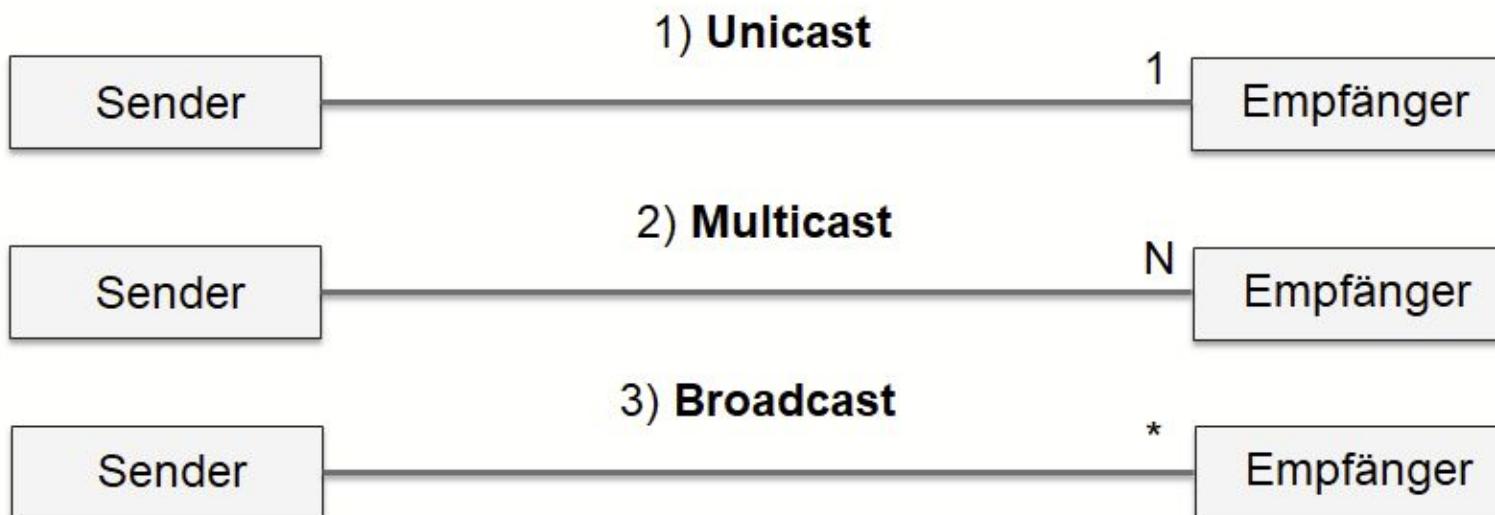
Service-Orientierung in einem Kommunikationssystem: Client-Server-Kommunikation über Services



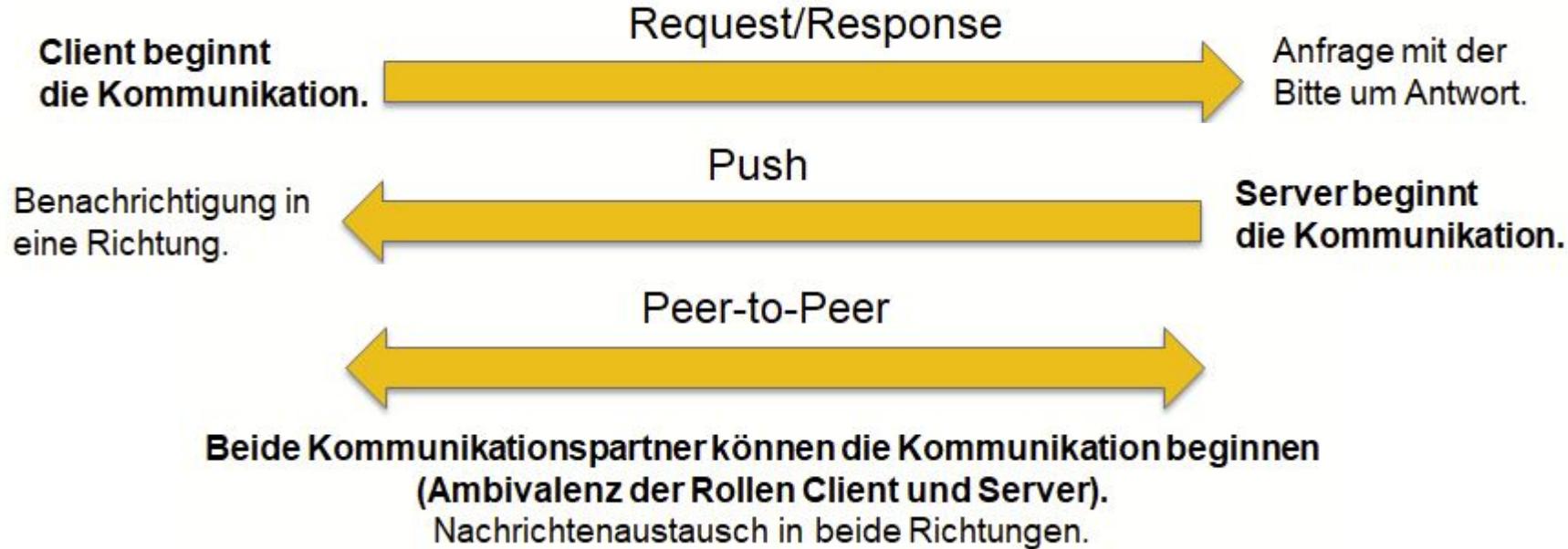
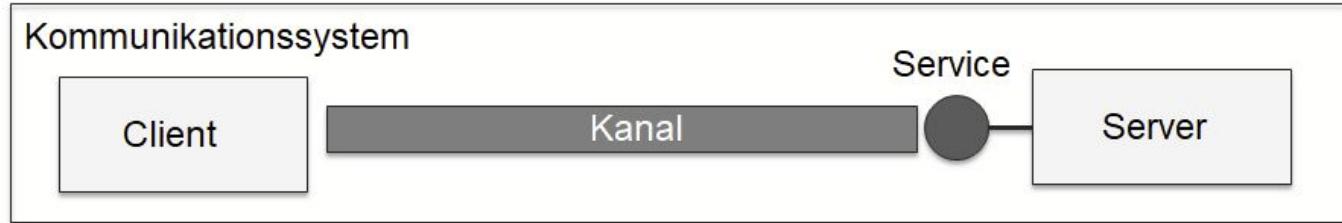
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

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

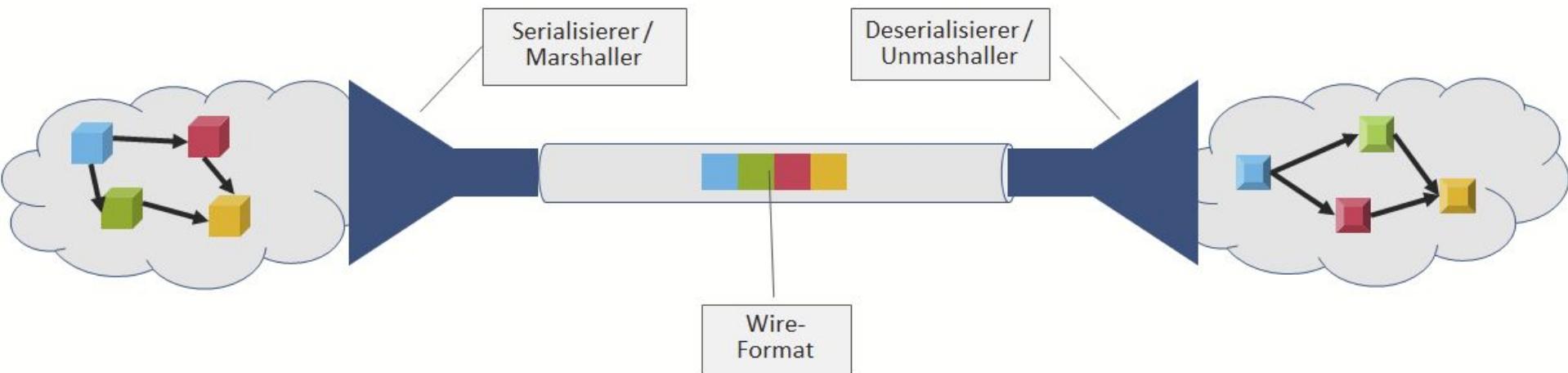
```
syntax = "proto3";

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

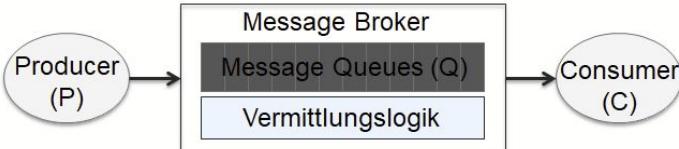
REST ist ein Paradigma für Anwendungsservices auf Basis des HTTP-Protokolls.

- 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

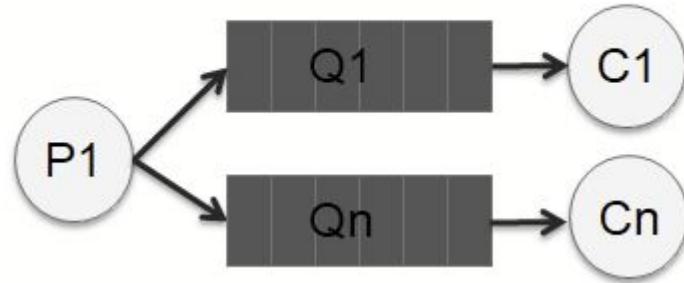
- Serviceschnittstelle: Format
- Messagebroker: keine Einschränkung für Format
- Sende- und Empfangszeitpunkt können beliebig lange auseinander liegen
- Horizontale Skalierbarkeit: Kann an mehrere Consumer ausgeliefert werden
- Lastverteilung: z.B. Auslieferung an Consumer, der am wenigsten zu tun hat
- 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

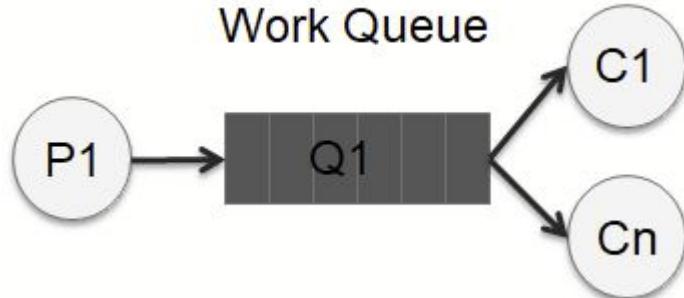
Message Passing



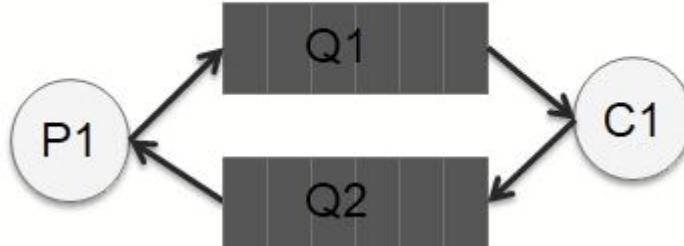
Publish/Subscribe



Work Queue

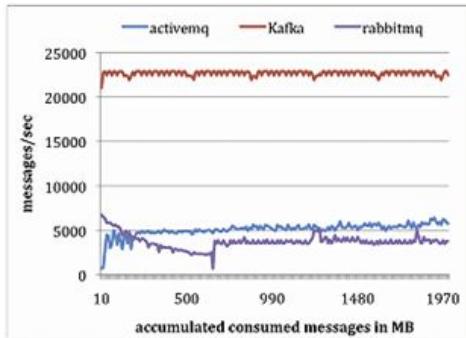


Remote Procedure Call



Apache Kafka

- Entwickelt bei LinkedIn und 2011 als Open Source Projekt veröffentlicht. Seit 2012 bei Apache Foundation.
- Kafka hat sich zum de-facto Standard in der Cloud für Messaging entwickelt, da Kafka hochgradig verteilbar und deutlich schneller als vergleichbare Lösungen ist:

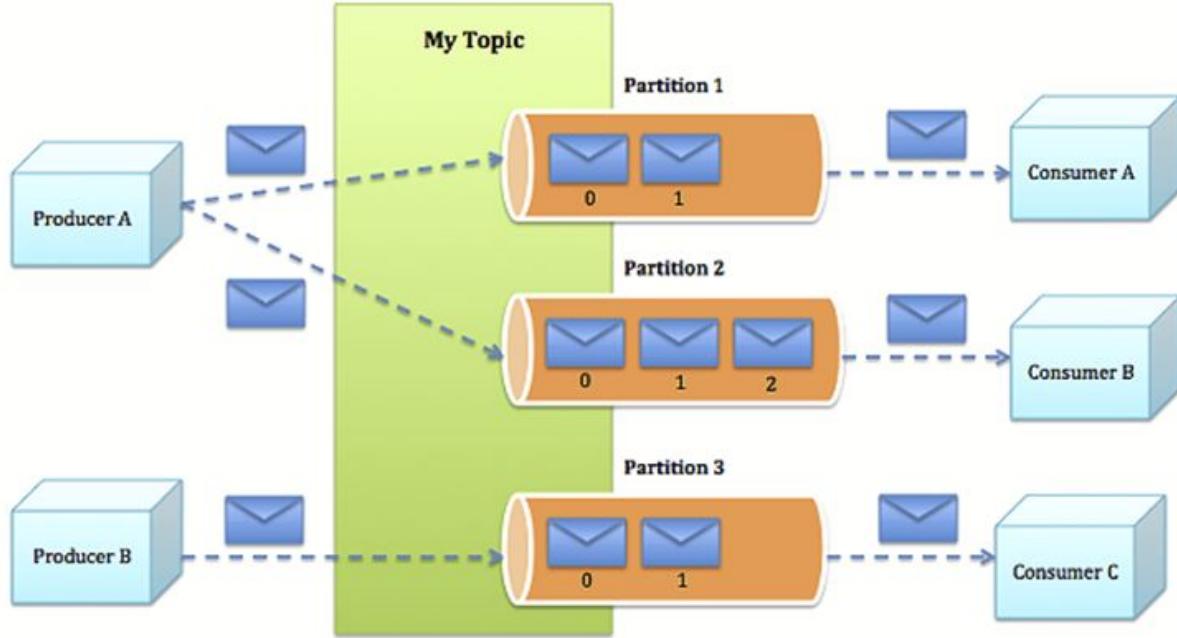


- Kafka ist so schnell, da es Betriebssystem-Mittel intelligent nutzt, ein effizientes Codierungsformat für Nachrichten besitzt und den Auslieferungszustand in den Clients hält.
- Kafka ist in Java und Scala geschrieben.
- Die Kafka API ist proprietär und orientiert sich an keinem Messaging-Standard.

Kafka basiert auf dem Konzept eines Event-Logs. Jeder Consumer hat einen eigenen Lese-Zeiger im Log.



Der Event-Log in Kafka ist hochgradig verteilt.



- Die Events in einem Topic werden aufgeteilt in Partitionen
- Die Partitionen werden verteilt auf die verfügbaren Broker-Instanzen
- Partitionen werden zur Fehlertoleranz repliziert

siehe:

- <http://www.michael-noll.com/blog/2013/03/13/running-a-multi-broker-apache-kafka-cluster-on-a-single-node>
- <http://www.infoq.com/articles/apache-kafka>

Kapitel Virtualisierung

Cloud Computing lebt von der Skalierung

- Erinnerung: NIST-Definition - "Resource Pooling"
- Organisation der Ressourcen in einem Rechenzentrum:
 - **Effiziente** Nutzung der gegebenen Ressourcen zur Minimierung der Kosten
 - **Isolation** der Ressourcen. Kunden sollen andere nicht sehen und auch nicht von ihnen beeinflusst werden. Seiteneffekte sollten vermieden werden, Security ist Ziel.
 - **Entkopplung von der Hardware** für mehr Flexibilität im Betrieb und Robustheit bei Ausfällen
 - Ressourcen sollen **flexibel** vergeben werden. Steuerung mittels "software defined resources"
- **Virtualisierung** löst diese Anforderungen und macht Cloud Computing erst möglich:
 - Virtualisierung erzeugt "**virtuelle Realitäten**", die die gewünschten **Eigenschaften** und **Fähigkeiten** haben, und bildet diese auf die **physische Realität** ab

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

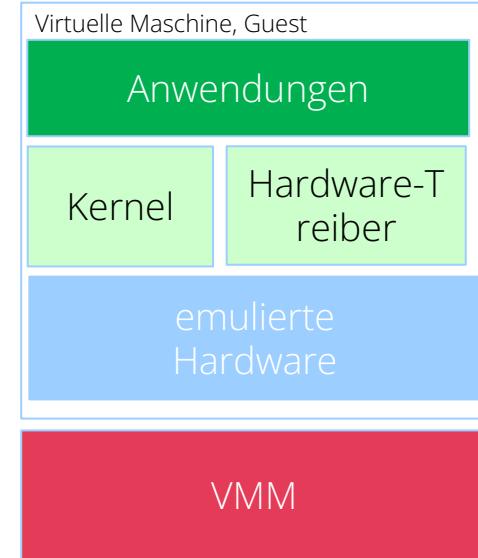
Was wird virtualisiert?

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
 - Computing (KI, Simulationen) (Zunehmend wichtig für die Cloud)
- Evtl. Peripherie wie USB, Maus, Keyboard
- Timer, Interrupt Controller

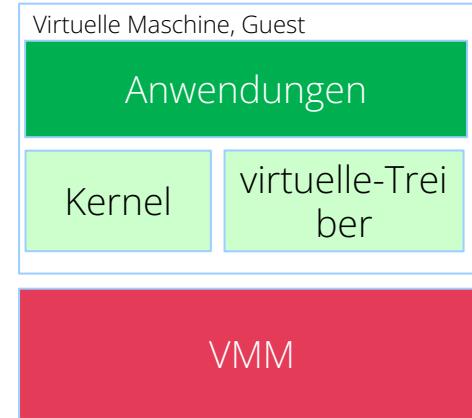
Voll-Virtualisierung

- 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%

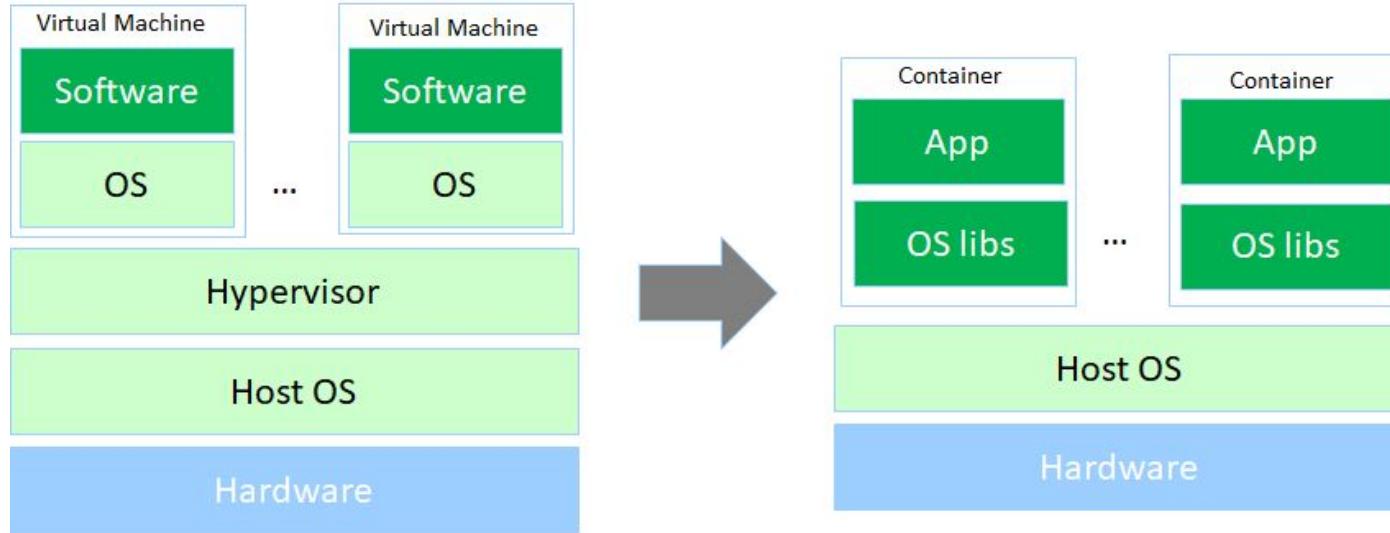


Para-Virtualisierung

- 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%



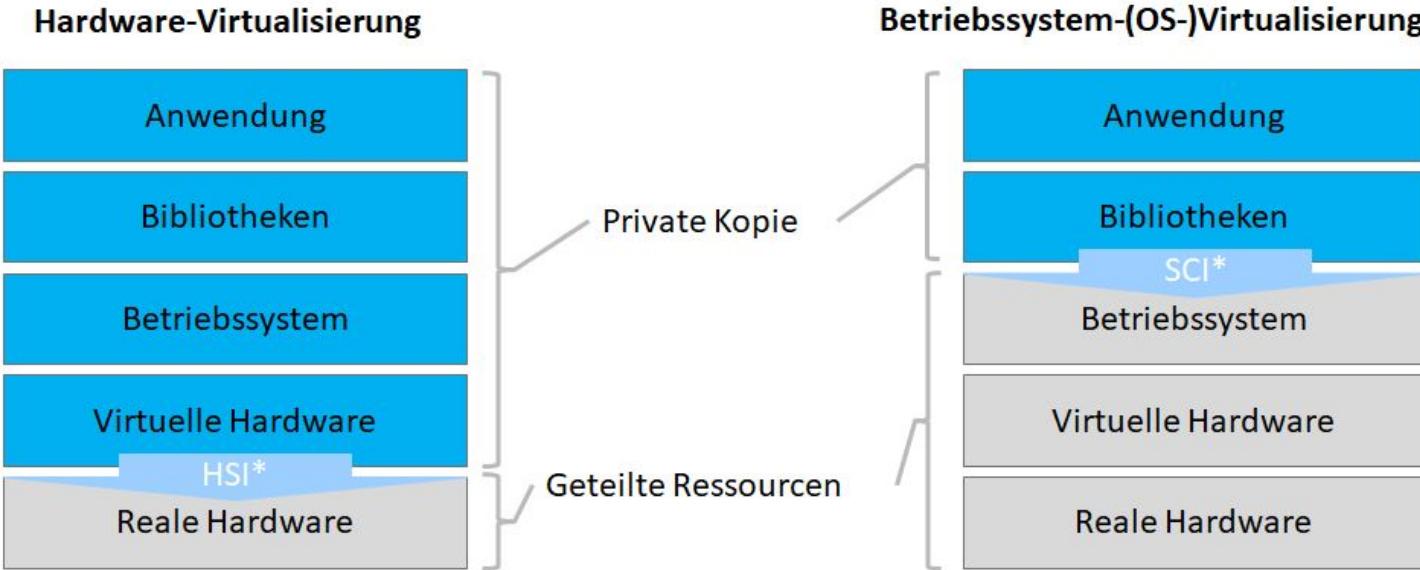
Betriebssystem-Virtualisierung / Containerisierung



- 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

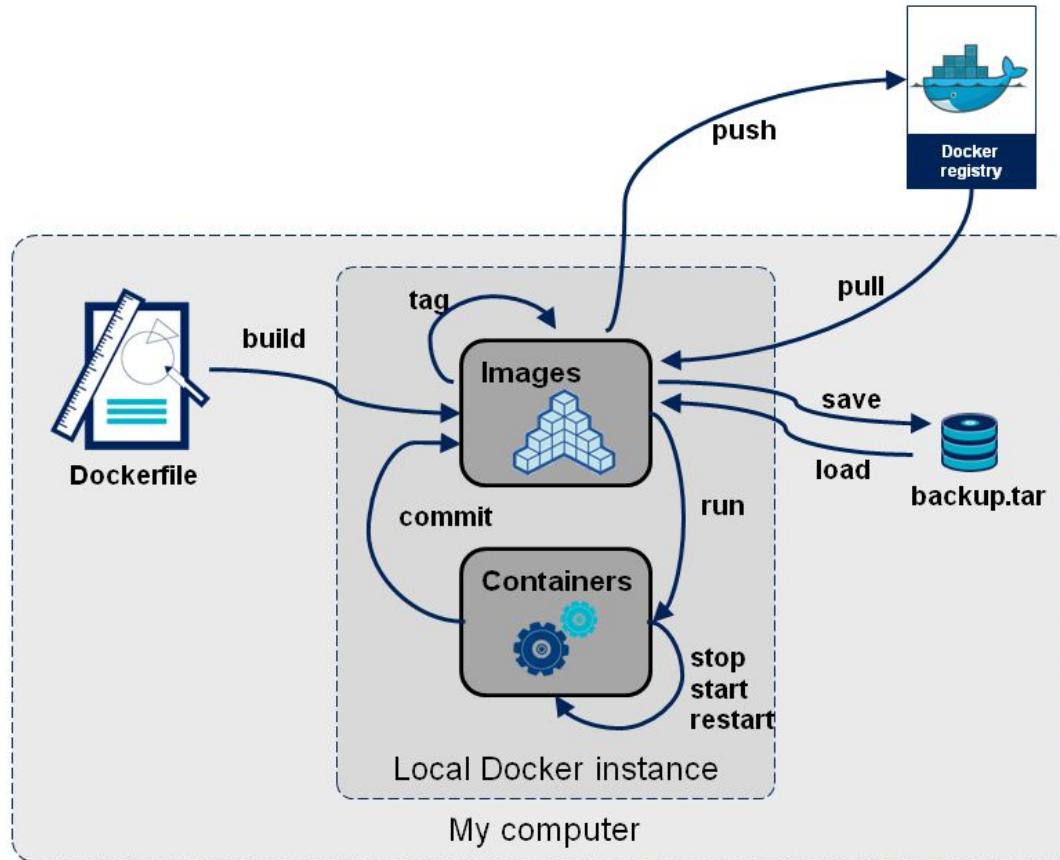
* HSI = Hardware Software Interface
SCI = System Call Interface



- 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"]
```

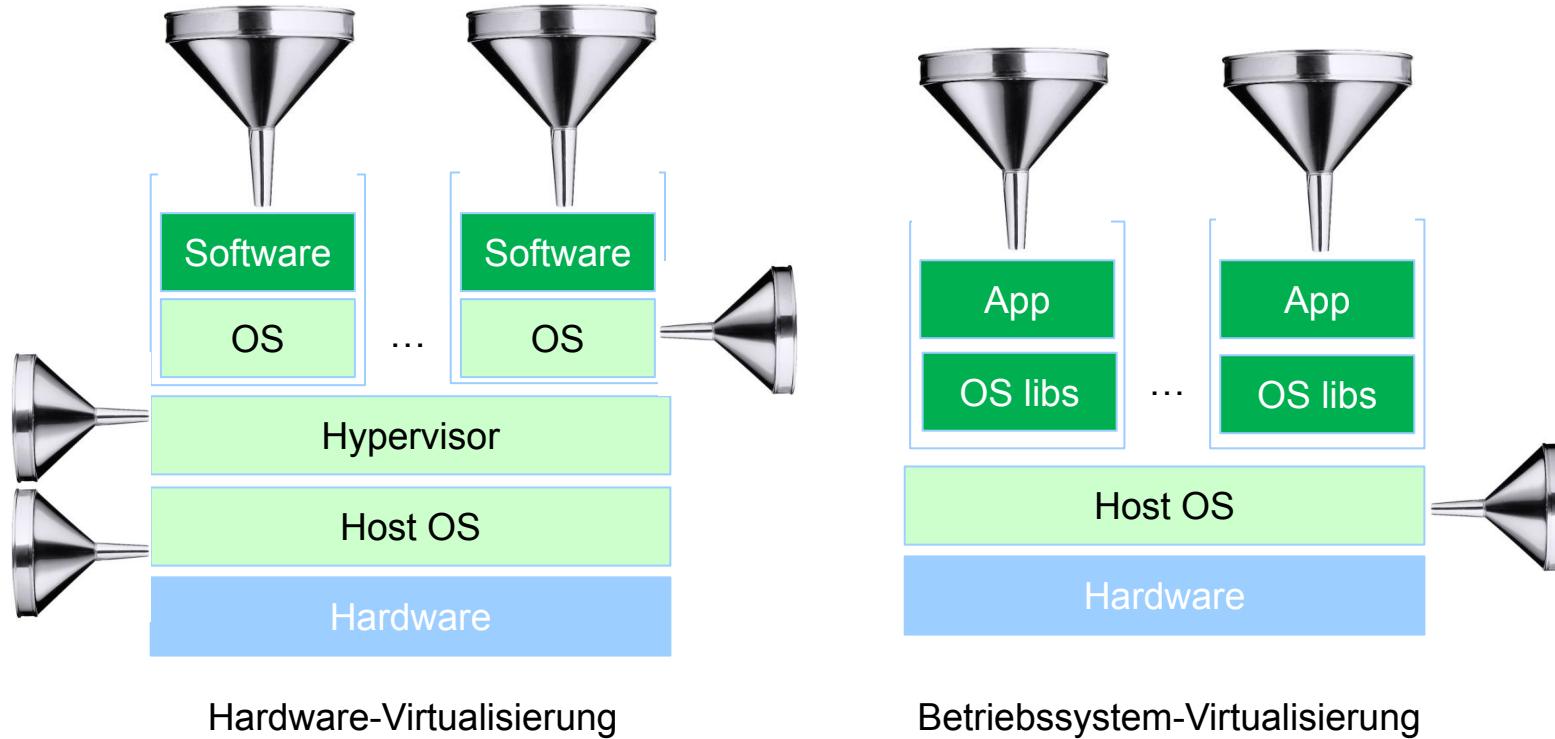
Typische Kommandos eines Docker Workflows

| Command | Action |
|--|---|
| <code>docker build -t <image> .</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 <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> <code>docker rm <container></code> | Terminate container (send SIGKILL to entrypoint process) Remove container |
| <code>docker rmi -f <image></code> | Remove local image |

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

Kapitel Provisionierung

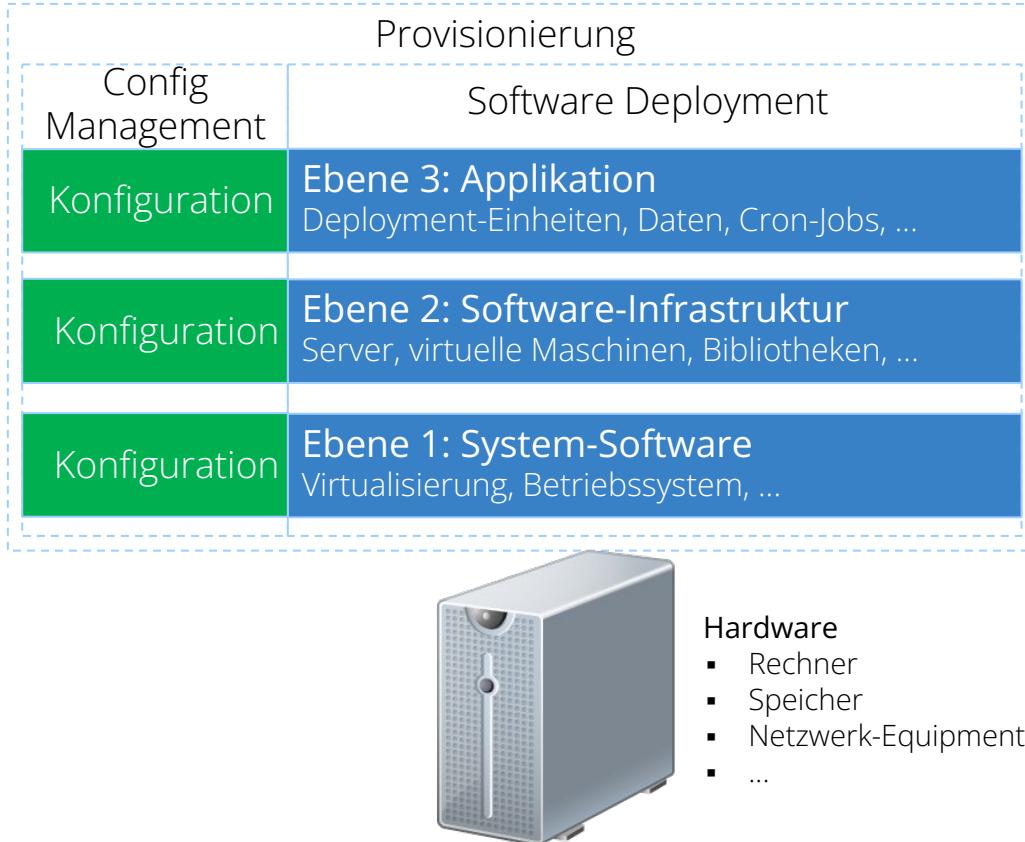
Provisionierung: Wie kommt Software in die Boxen?



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

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

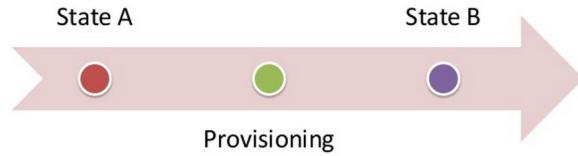
Provisierung erfolgt auf **3 Ebenen** und in **4 Stufen**



Konzeptionelle Überlegungen zur Provisionierung

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

Provisionierung := Überführung eines Systems von seinem aktuellen Zustand auf den 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 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.

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>

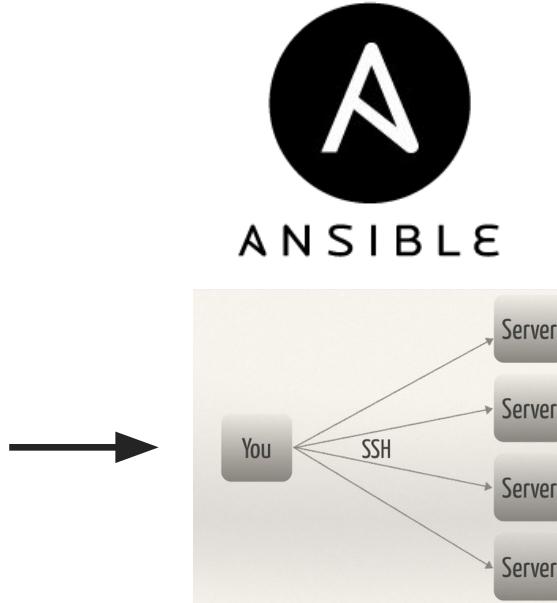
Docker Compose

Compose is a tool for defining and running multi-container Docker applications. With Compose, you use a YAML file to configure your application's services. Then, with a single command, you create and start all the services from your configuration.

<https://docs.docker.com/compose/>

Ansible

- Open-Source-Provisionierungswerkzeug von Red Hat
- Ausgelegt auf die Provisionierung großer heterogener IT-Landschaften
- Entwickelt in der Sprache Python
- Push-Prinzip: Benötigt im Vergleich zu anderen Lösung weder einen Agenten auf den Ziel-Rechnern (SSH & Python reicht) noch einen zentralen Provisionierungs-Server
- Ist einfach zu erlernen im Vergleich zu anderen Lösungen
- Deklarativer Stil
- Umfangreiche Bibliothek vorgefertigter Provisionierungs-Aktionen inkl. Community-Funktion
(<https://galaxy.ansible.com>)



Packer

Packer is an open source tool for creating identical machine images for multiple platforms from a single source configuration. Packer is lightweight, runs on every major operating system, and is highly performant, creating machine images for multiple platforms in parallel. Packer does not replace configuration management like Chef or Puppet. In fact, when building images, Packer is able to use tools like Chef or Puppet to install software onto the image.

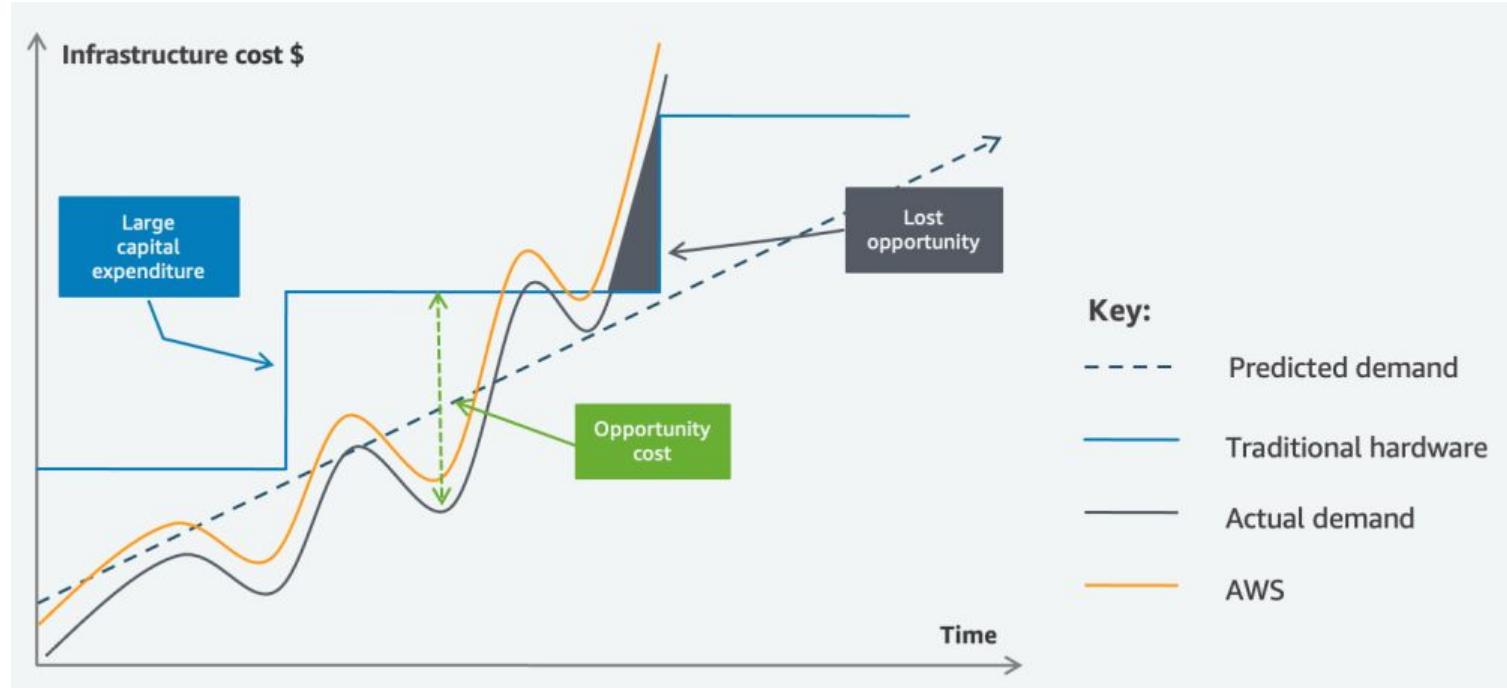
A machine image is a single static unit that contains a pre-configured operating system and installed software which is used to quickly create new running machines. Machine image formats change for each platform. Some examples include AMIs for EC2, VMDK/VMX files for VMware, OVF exports for VirtualBox, etc.

<https://developer.hashicorp.com/packer/docs/intro>

- Geschrieben in Go
- Templatisiert den Bau von Images
- Bestehende Provisionierungsskripte (z.B. Ansible) können wiederverwendet werden
- Ermöglicht den Bau von Images für mehrere Plattformen mit einer gemeinsamen Konfiguration

Kapitel Infrastructure-as-a-Service

Klassische Betriebsszenarien werden bei dynamischer Nachfrage teuer - Hohe Opportunitätskosten



Source: [Amazon Web Services](#)

Definition IaaS

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

Eigenschaften einer IaaS-Cloud:

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

Ressourcen-Typen in einer IaaS-Cloud:

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

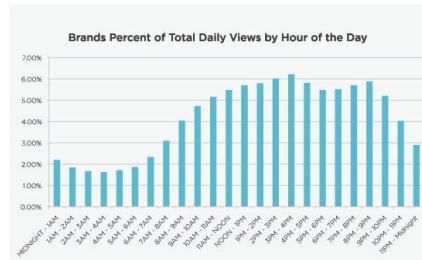
Infrastruktur-Dienste einer IaaS-Cloud:

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

Skalierbarkeit: Effekte

- Tageszeitliche und saisonale Effekte:

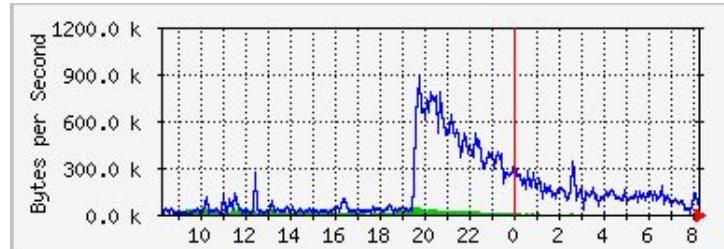
Mittags-Peak, Prime-Time-Peak, Wochenend-Peak, Weihnachten, Valentinstag, Muttertag, Black Friday (vorhersehbare Belastungsspitzen)



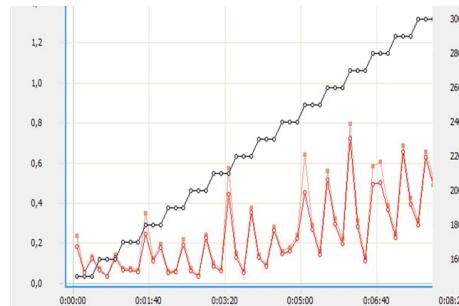
- Kontinuierliches Wachstum



- 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.

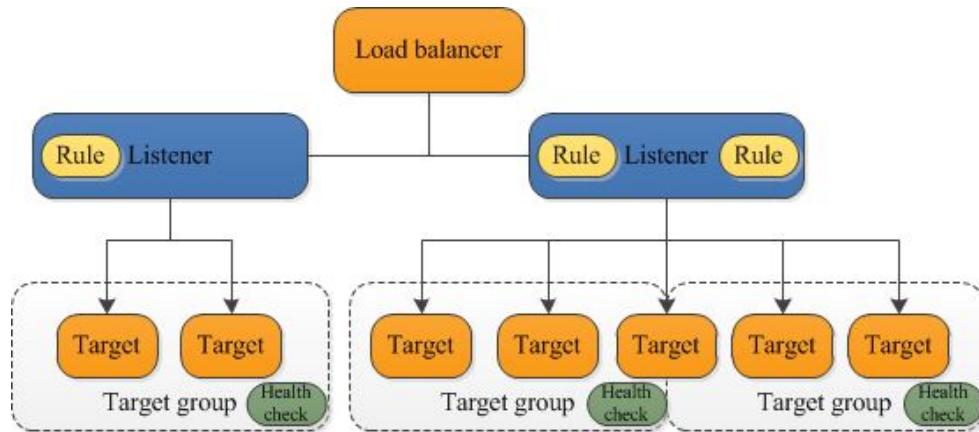
| Monthly Uptime Percentage | Service Credit Percentage |
|--|---------------------------|
| Equal to or greater than 99% but less than 99.9% | 10% |
| less than 99% | 25% |

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

Elastic Load Balancing

- Nimmt öffentlichen Verkehr entgegen und verteilt diesen auf Instanzen.
- Überwacht die Funktionalität der Instanzen / Applikation (Health Check) und verteilt die Anfragen / Verbindungen nur auf "gesunde" Ziele.
- Verschiedene Varianten:
 - Application Load Balancer - Layer 7
 - Network Load Balancer - Layer 4
 - Classic Load Balancer - Legacy
- Unterstützen TLS, Integration mit AutoScaling usw.



Beispiel: Application Load Balancer

Infrastructure as Code

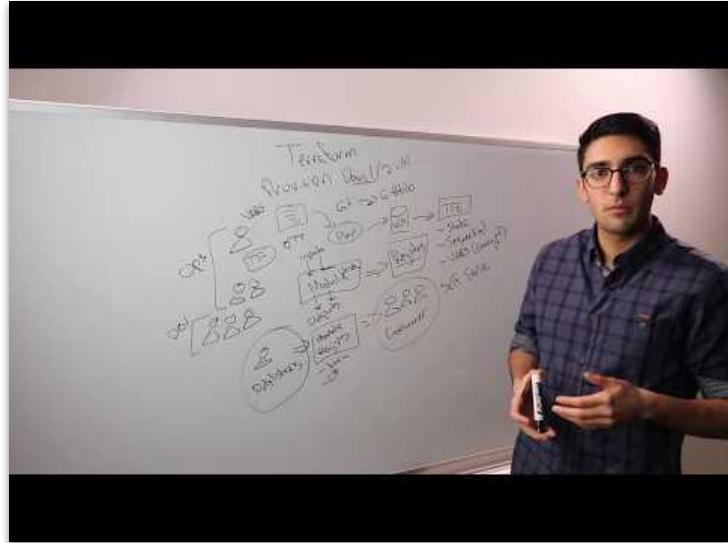
- Provisionieren und Managen ganzer Rechenzentren - nicht nur einzelne virtuelle Maschinen
- Abgrenzung zu Configuration Management (z.B. Ansible):
 - Explizite Erzeugung und Zerstörung der Infrastruktur eines (virtuellen) Rechenzentrums
 - Immutable Infrastructure, statt kontinuierlichem ändern existierender 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

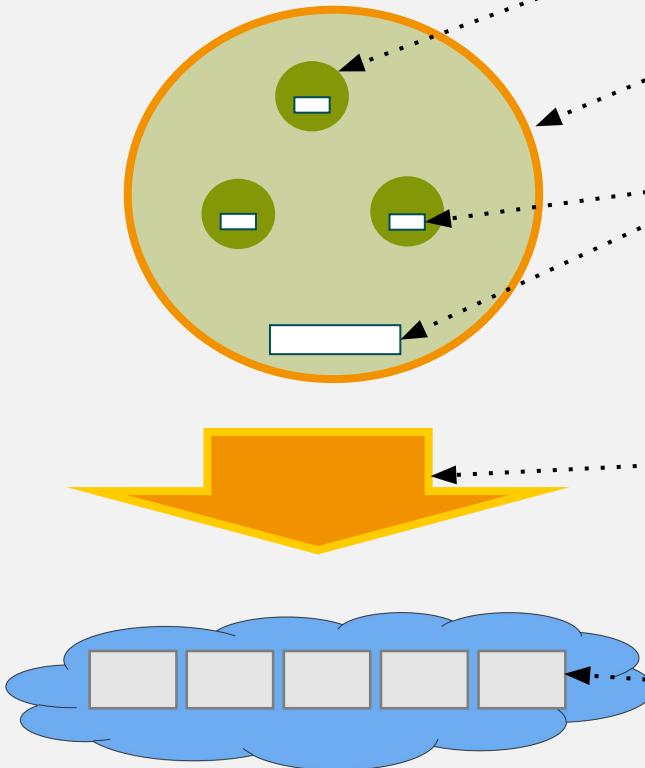
- **Write:** Beschreibung Zielzustand über eine domänenspezifische Sprache HCL (HashiCorp Configuration Language)
- **Plan (terraform plan):** Ist-Zustand ermitteln. Notwendige Änderungen planen (entsprechend Abhängigkeiten geordnet und parallelisiert, Unterbrechungen möglichst minimal)
- **Apply (terraform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



| 57
Video Pause bei 06m:30s, dann weiter bis 10m18s:
<https://www.youtube.com/watch?v=h970ZBgKING>

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.

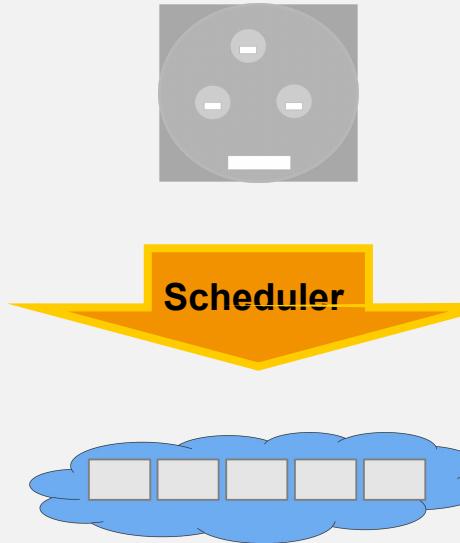
Properties: Ausführungsrelevante Eigenschaften der Tasks und Jobs, wie z.B.:

- Task: Ausführungszeitdauer, 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



QA|WARE



Vorteil:

- Einfach zu realisieren

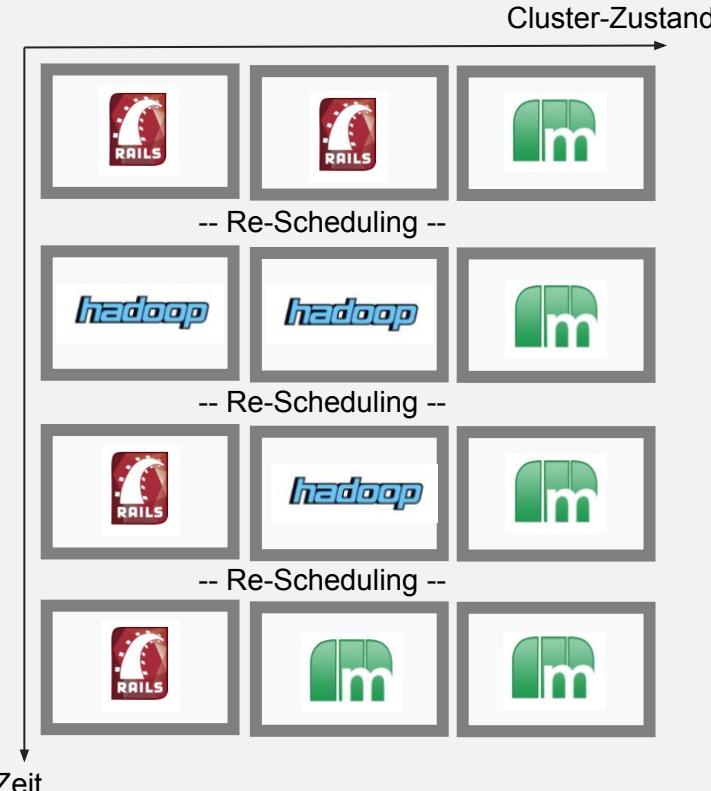
Nachteile:

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

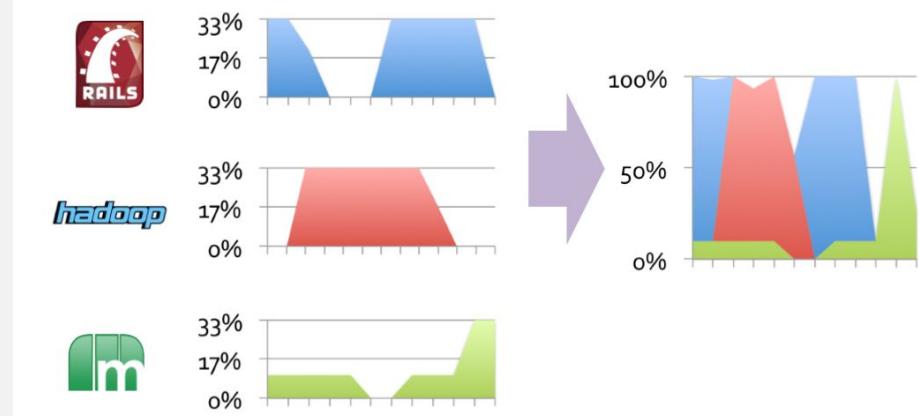


Bildquelle: Practical Considerations for Multi-Level Schedulers, Benjamin Hindman, 19th Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP) 2015

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



Statische Partitionierung Dynamische Partitionierung



Vorteile der dynamischen Partitionierung:

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

Cluster-Scheduler: Eingabe, Verarbeitung, Ausgabe

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

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



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 aller Ressourcen.
- **Zuverlässigkeit:** Ein Task wird garantiert ausgeführt.
- **Geringe End-to-End Ausführungszeit** (z.B. durch Daten-Lokalität und geringe Kommunikationskosten, syn. Makespan)

Ausgabe eines Cluster-Schedulers:

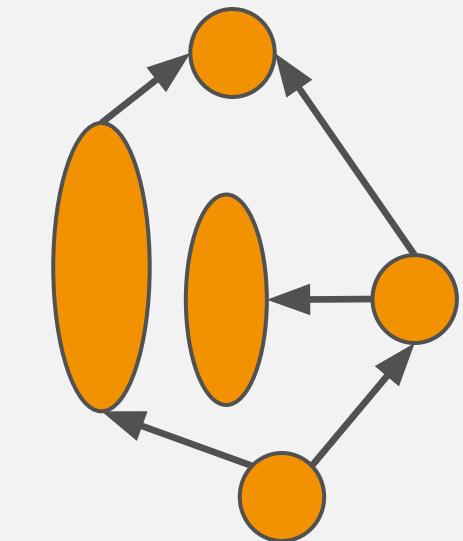
Placement Decision als

- **Slot-Reservierungen**

- **Slot-Stornierungen**

(im Fehlerfall, Optimierungsfall,
Constraint-Verletzung)

Cluster-Scheduling ist eine Optimierungsaufgabe

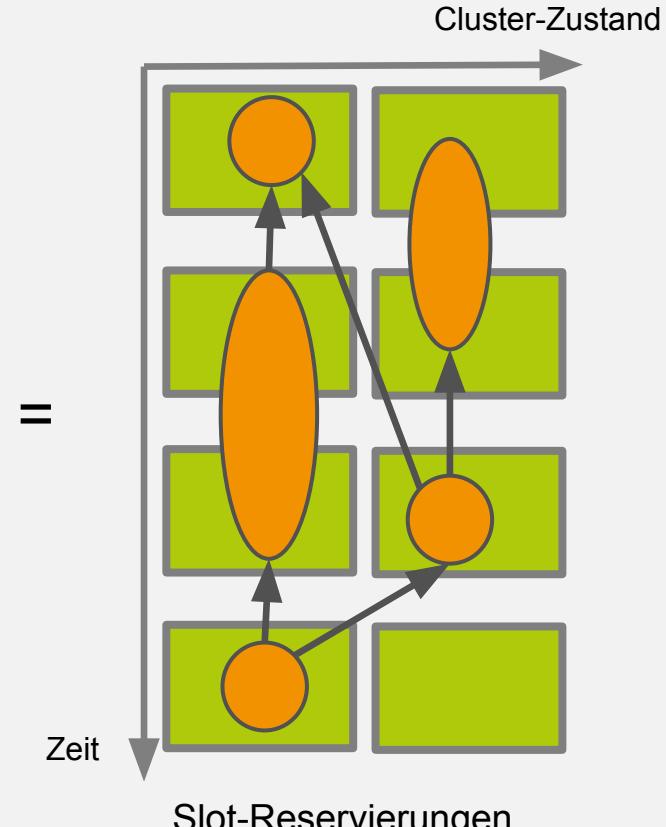


+



Tasks mit Zeitbedarf (Höhe)
und Ausführungsabhängigkeiten
(Pfeile)

Verfügbare Ressourcen



Slot-Reservierungen

Scheduling ist eine Optimierungsaufgabe...



... und ist **NP-vollständig**.

Die Optimierungsaufgabe lässt sich auf das Traveling Salesman Problem zurückführen.

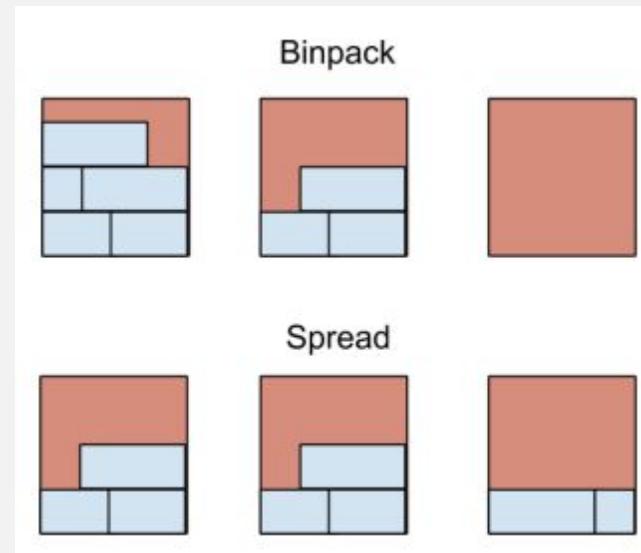
Das bedeutet:

- 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)

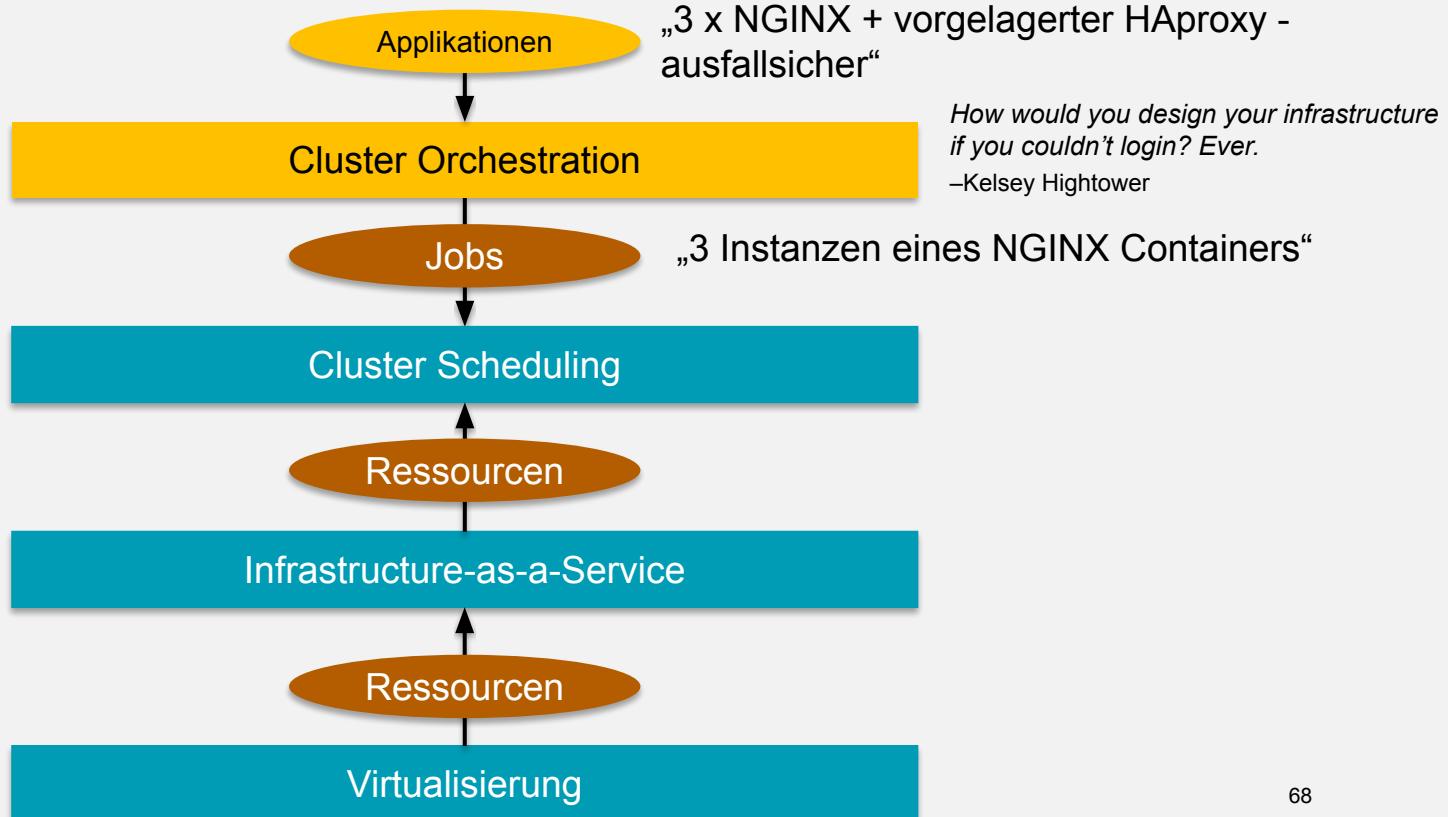


Kapitel Orchestrierung

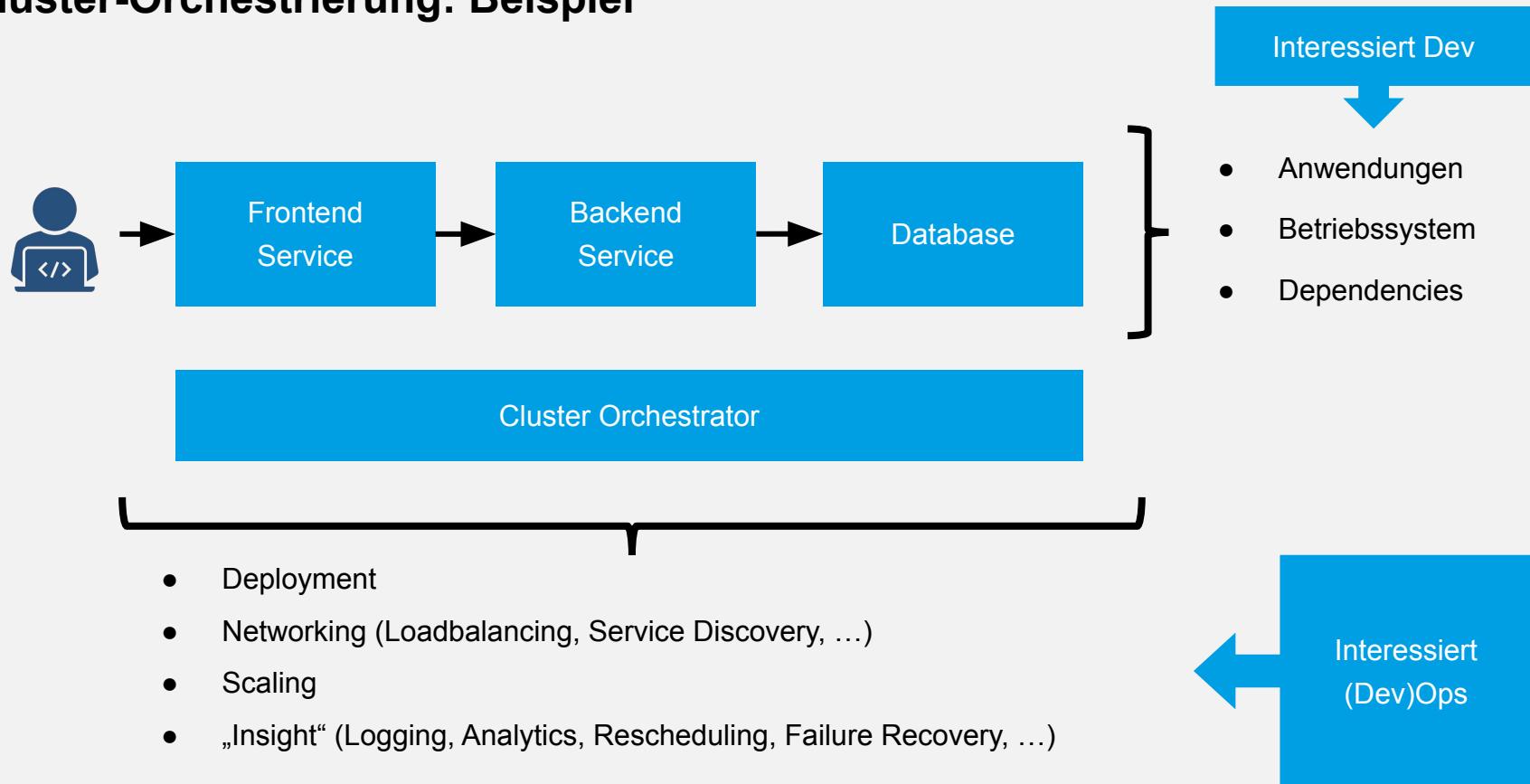
Das Big Picture: Wir sind nun auf Applikationsebene.



QA|WARE



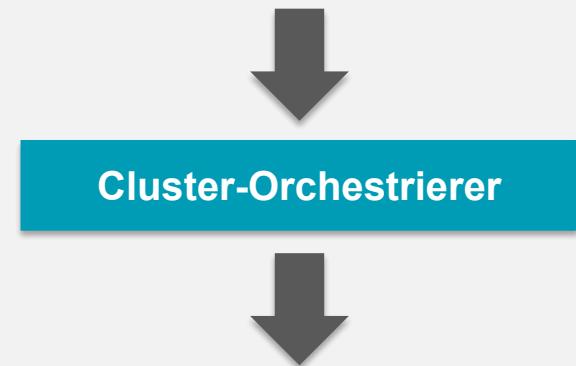
Cluster-Orchestrierung: Beispiel



Cluster-Orchestrierung

- Ziel: Eine Anwendung, die in mehrere Betriebskomponenten (Container) aufgeteilt ist, auf mehreren Knoten laufen lassen.
- 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.

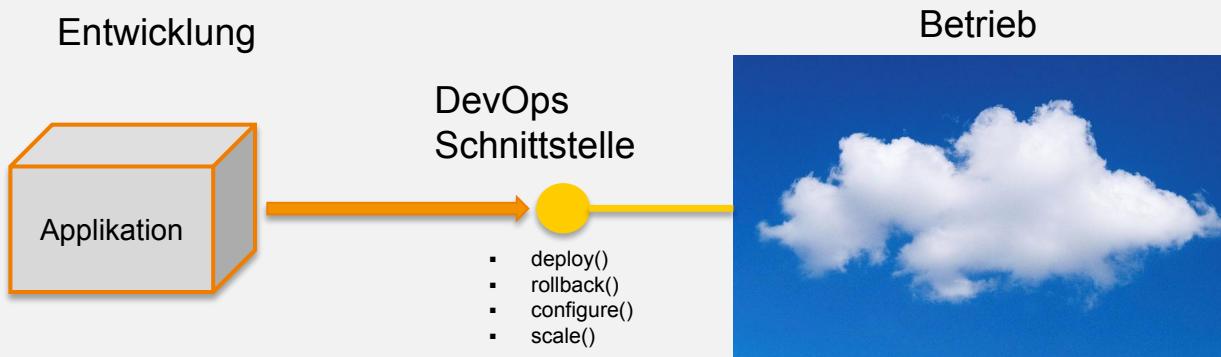


Cluster-Orchestrizer

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.



Ein Cluster-Orchestrator automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (1 / 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.
- Container-Logistik: Verwaltung und Bereitstellung von Containern.

Ein Cluster-Orchestrator automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster (2 / 2):



- 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.

Kubernetes



kubernetes by Google

Manage a cluster of Linux containers as a single system to accelerate Dev and simplify Ops.

Josef Adersberger @adersberger · Jul 21

Google spares no effort to launch
#kubernetes @ #OSCON

2015

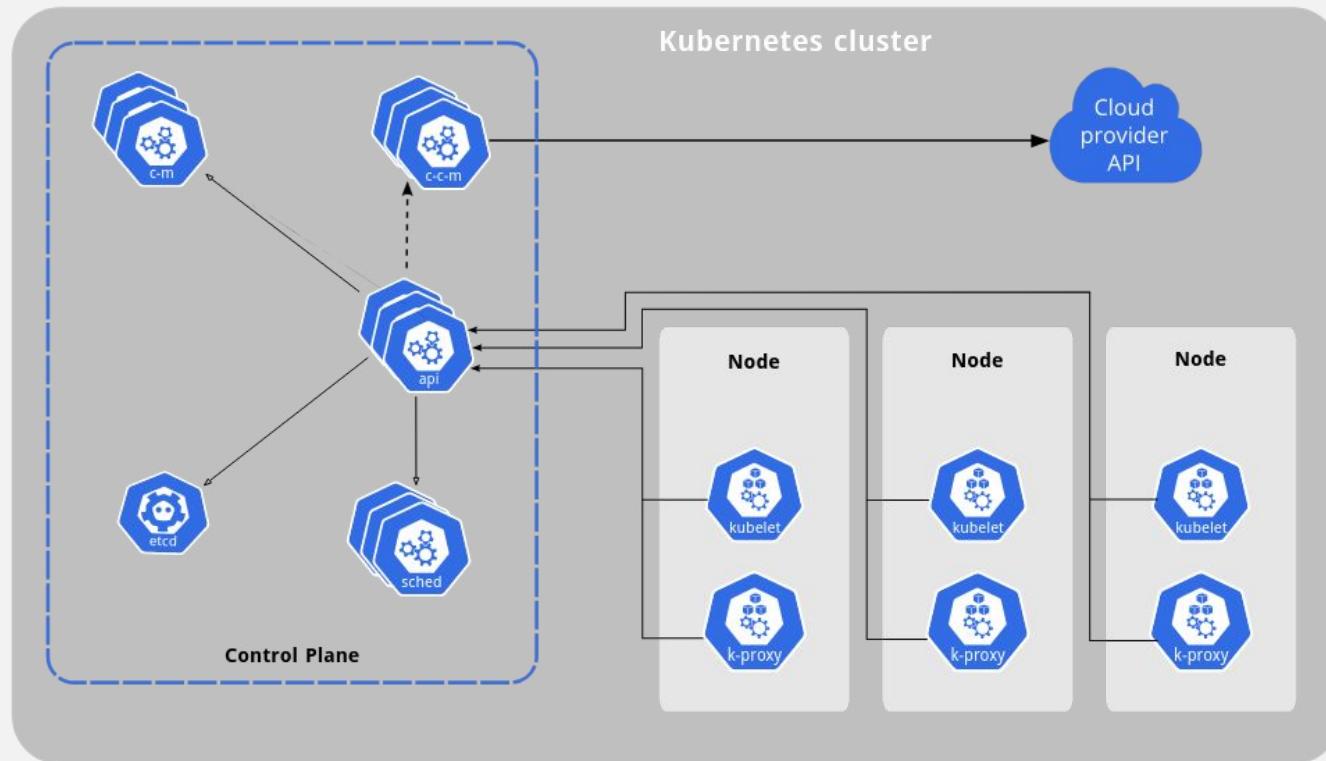


- **Cluster-Orchestrator** auf Basis von Containern, der eine Reihe an Kern-Abstraktionen für den Betrieb von Anwendungen in einem großen Cluster einführt. Die Blaupause wird über YAML-Dateien definiert.
- **Open-Source-Projekt**, das von Google initiiert wurde. Google will damit die jahrelange Erfahrung im Betrieb großer Cluster der Öffentlichkeit zugänglich machen und damit auch Synergien mit dem eigenen Cloud-Geschäft heben.
- Seit Juli 2015 in der Version 1.0 verfügbar und damit **produktionsreif**. Skaliert aktuell nachweislich auf 1000 Nodes großen Clustern.
- **Bei vielen Firmen im Einsatz** wie z.B. Google im Rahmen der Google Container Engine, Wikipedia, ebay. Beiträge an der Codebasis aus vielen Firmen neben Google – u.A. Mesosphere, Microsoft, Pivotal, RedHat.
- Setzt den **Standard im Bereich Cluster-Orchestrierung**. Dafür wurde auch eigens die Cloud Native Computing Foundation gegründet (<https://cncf.io>).

Architektur von Kubernetes



QA|WARE



| | |
|--|-------|
| API server | |
| Cloud controller manager (optional) | |
| Controller manager | |
| etcd (persistence store) | |
| kubelet | |
| kube-proxy | |
| Scheduler | |
| Control plane | ----- |
| Node | |



QA|WARE

Pods & Deployments

Wichtige Kubernetes-Konzepte



QA|WARE

Der Grundbaustein ist eure **Anwendung**.

App

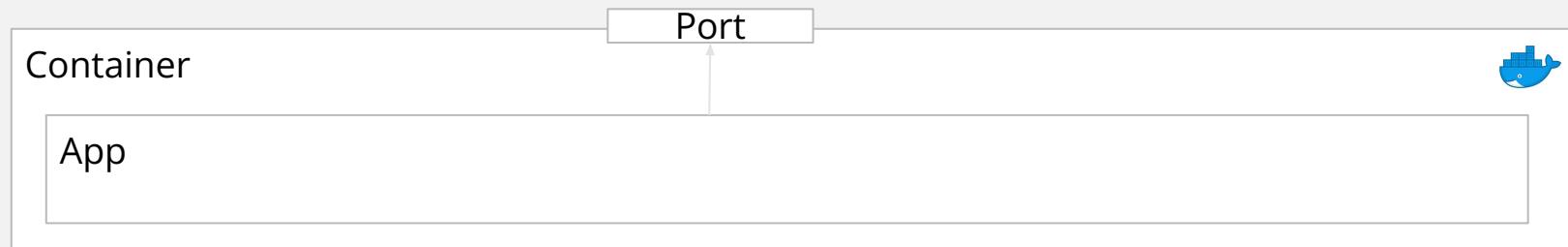
Wichtige Kubernetes-Konzepte



QA|WARE

Der Grundbaustein ist eure **Anwendung**.

Die **Anwendung** steckt in einem **Container** (siehe Vorlesung “Virtualisierung”). **Container** öffnen **Ports** nach außen.



Wichtige Kubernetes-Konzepte

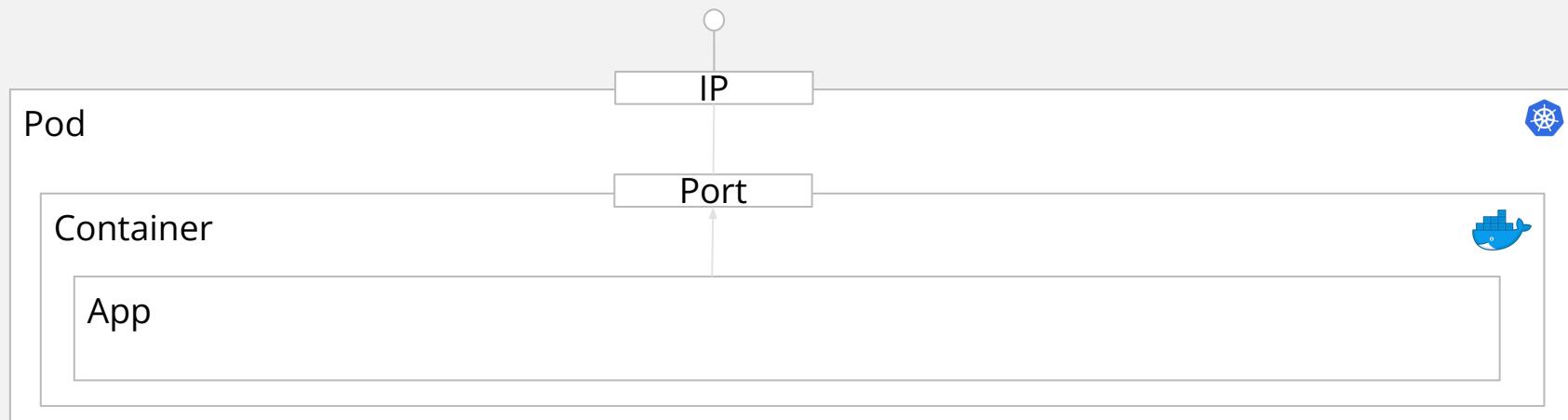


QA|WARE

Der Grundbaustein ist eure **Anwendung**.

Die **Anwendung** steckt in einem **Container** (siehe Vorlesung “Virtualisierung”). **Container** öffnen **Ports** nach außen.

Container werden in Kubernetes zu **Pods** zusammengefasst. **Pods** haben nach außen hin eine **IP-Adresse**.

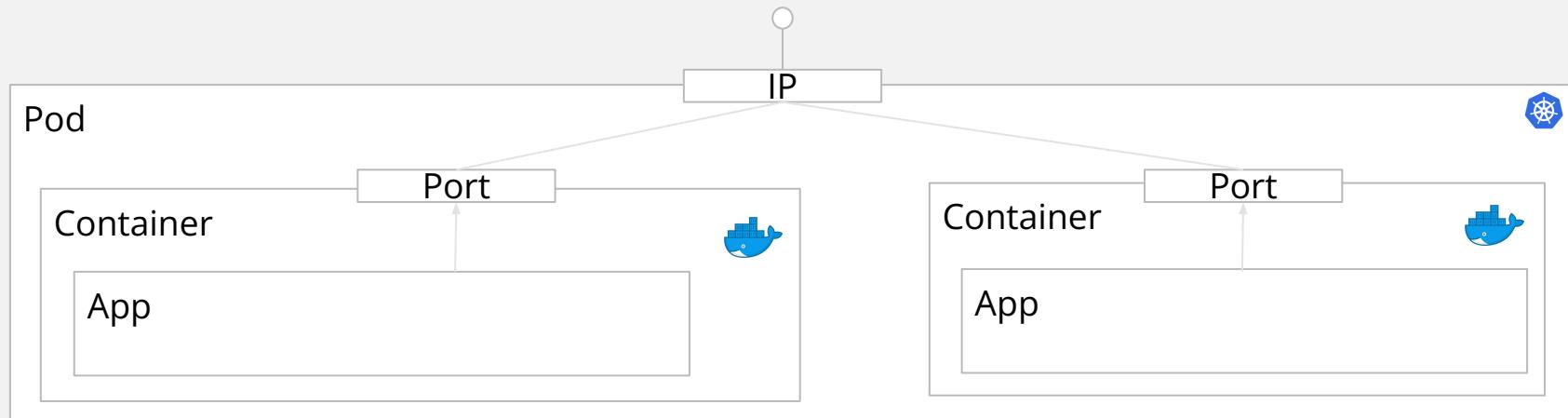


Wichtige Kubernetes-Konzepte



QA|WARE

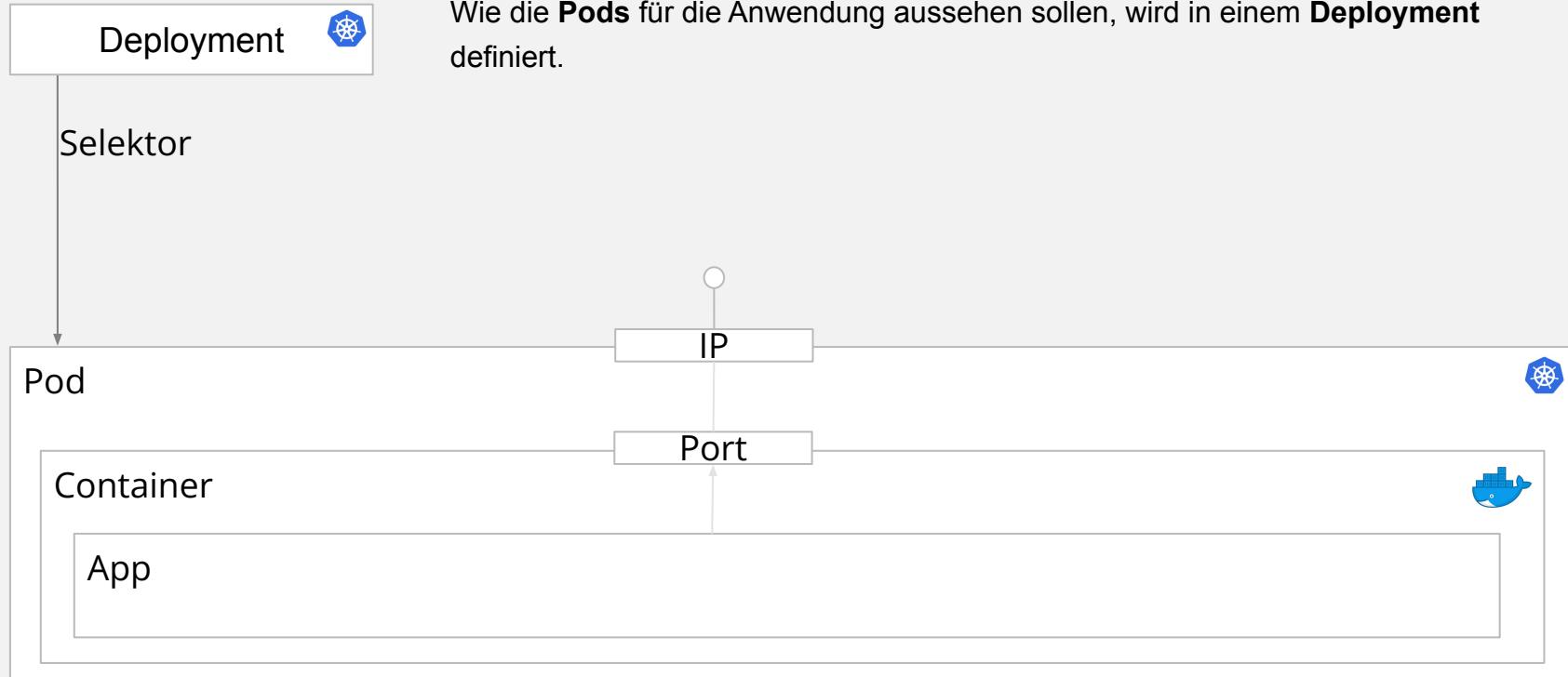
In einem **Pod** können auch mehrere **Container** laufen.



Wichtige Kubernetes-Konzepte



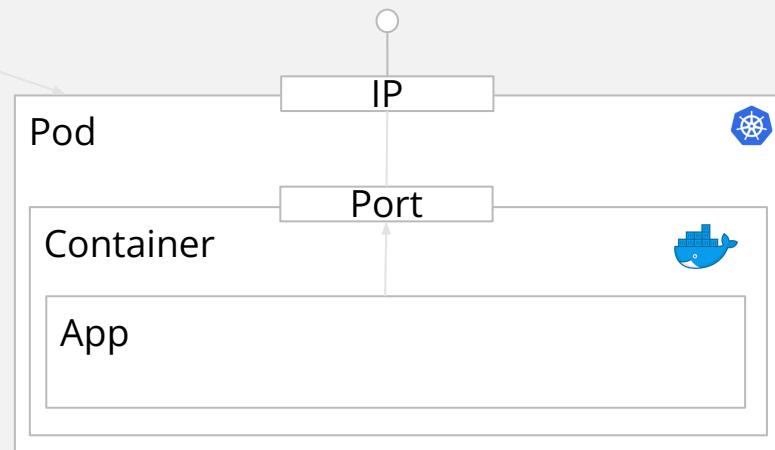
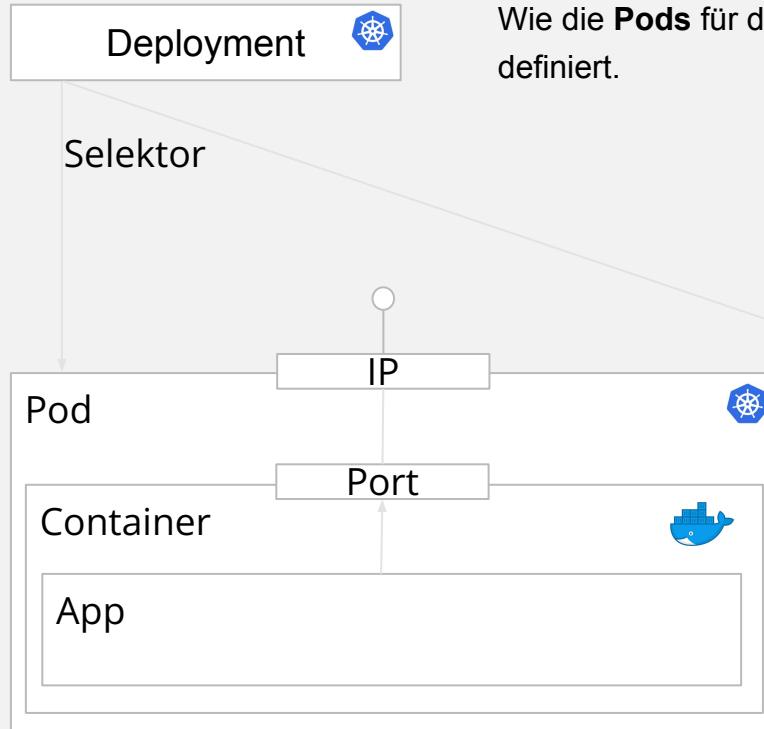
QA|WARE



Wichtige Kubernetes-Konzepte



QA|WARE



Deployment: Definition



QA|WARE

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-service
spec:
  replicas: 3
  template:
    metadata:
      labels:
        app: helloservice
    spec:
      containers:
        - name: hello-service
          image: "hitchhikersguide/zwitscher-service:1.0.1"
          ports:
            - containerPort: 8000
```



QA|WARE

Probes & Resources

Resource Constraints



QA|WARE

resources:

```
# Define resources to help K8S scheduler  
  
# CPU is specified in units of cores  
# Memory is specified in units of bytes  
  
# required resources for a Pod to be started
```

requests:

```
memory: "128M"  
cpu: "0.25"
```

limits:

```
memory: "192M"    # Pod will be restarted if mem limit is exceeded  
cpu: "0.5"        # Pod will be throttled to cpu limit if exceeded
```

Liveness und Readiness Probes



QA|WARE

```
# container will receive requests as long as probe succeeds
```

```
readinessProbe:
```

```
  httpGet:
```

```
    path: /actuator/readiness
```

```
    port: 8080
```

```
periodSeconds: 2
```

```
failureThreshold: 5
```

```
# container will be killed
```

```
if probe fails
```

```
livenessProbe:
```

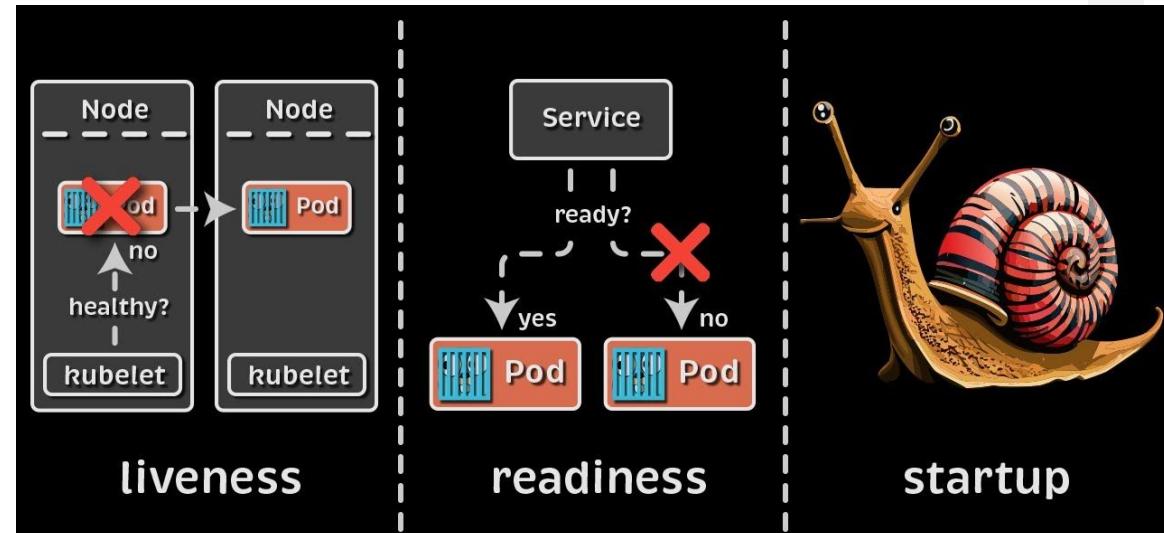
```
  httpGet:
```

```
    path: /actuator/liveness
```

```
    port: 8080
```

```
periodSeconds: 10
```

```
failureThreshold: 3
```

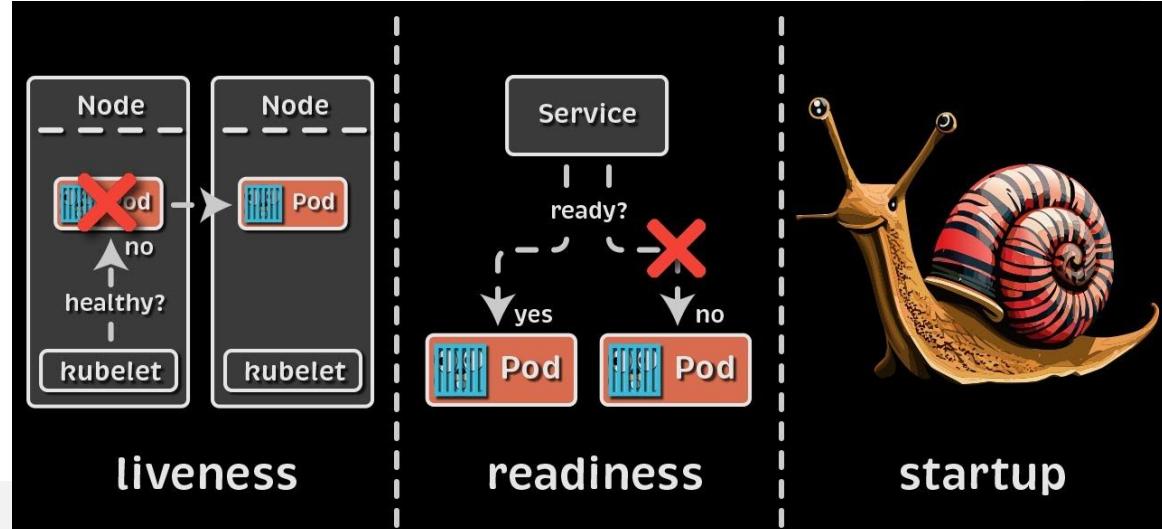


Startup Probe



QA|WARE

```
# readiness and liveness probes are paused until startup probe is successful
startupProbe:
  httpGet:
    path: /actuator/liveness
    port: 8080
periodSeconds: 1
failureThreshold: 300
```





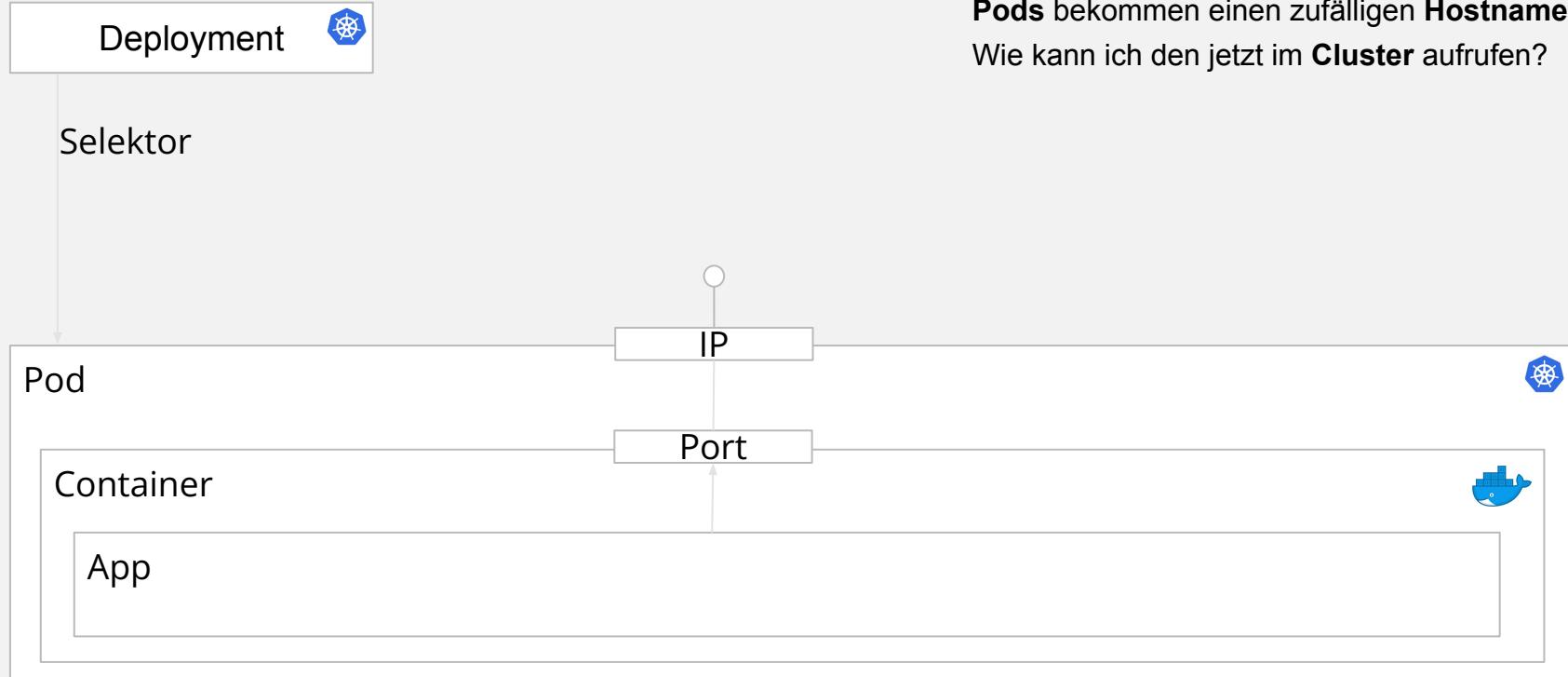
QA|WARE

Services

Wichtige Kubernetes Konzepte



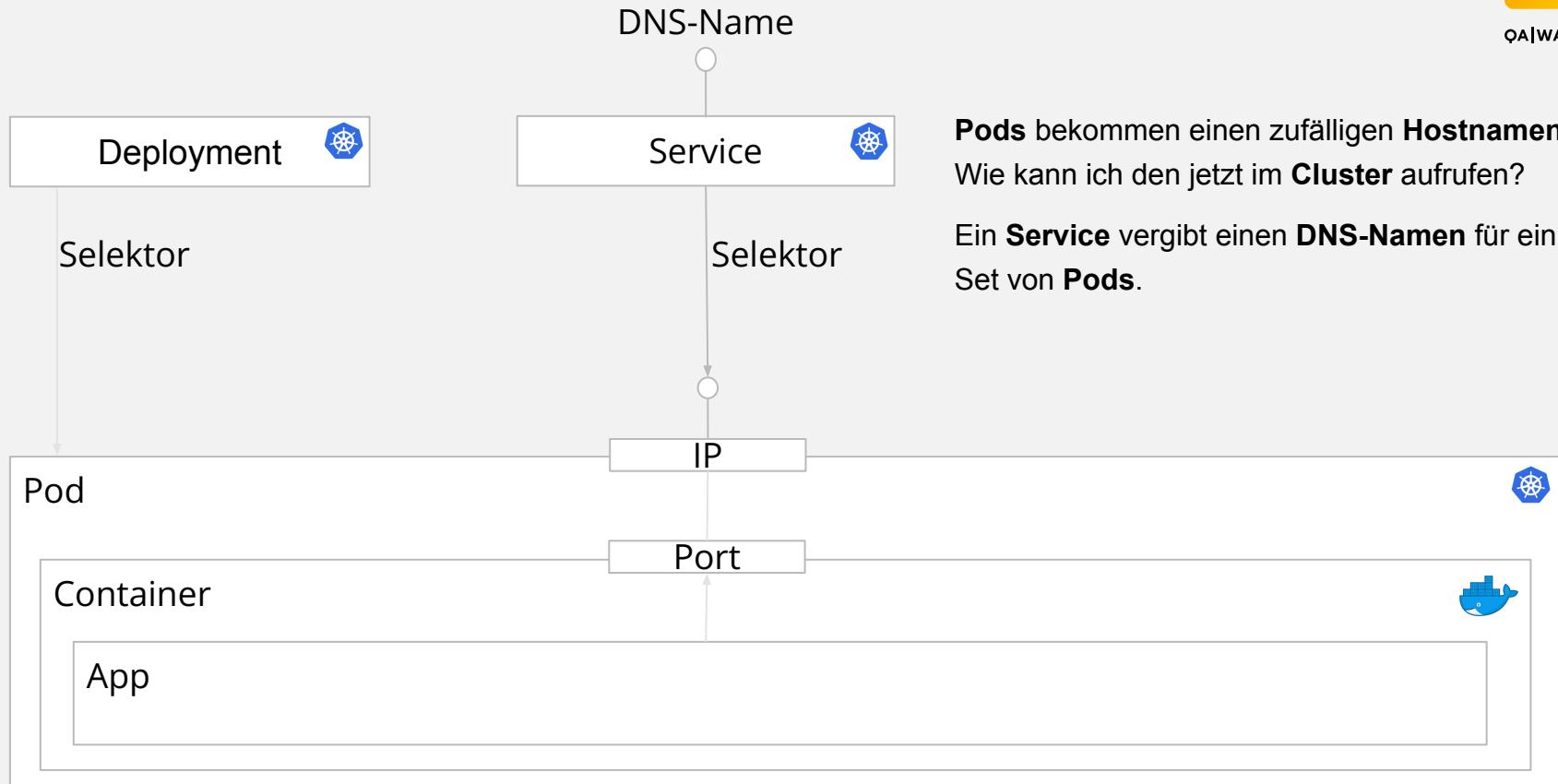
QA|WARE



Wichtige Kubernetes Konzepte



QA|WARE



Service: Definition



QA|WARE

```
apiVersion: v1
kind: Service
metadata:
  name: hello-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          # port on Service
      targetPort: 8080 # port on Pod
  # selector targets Pods with the given label
  selector:
    app: helloservice
```



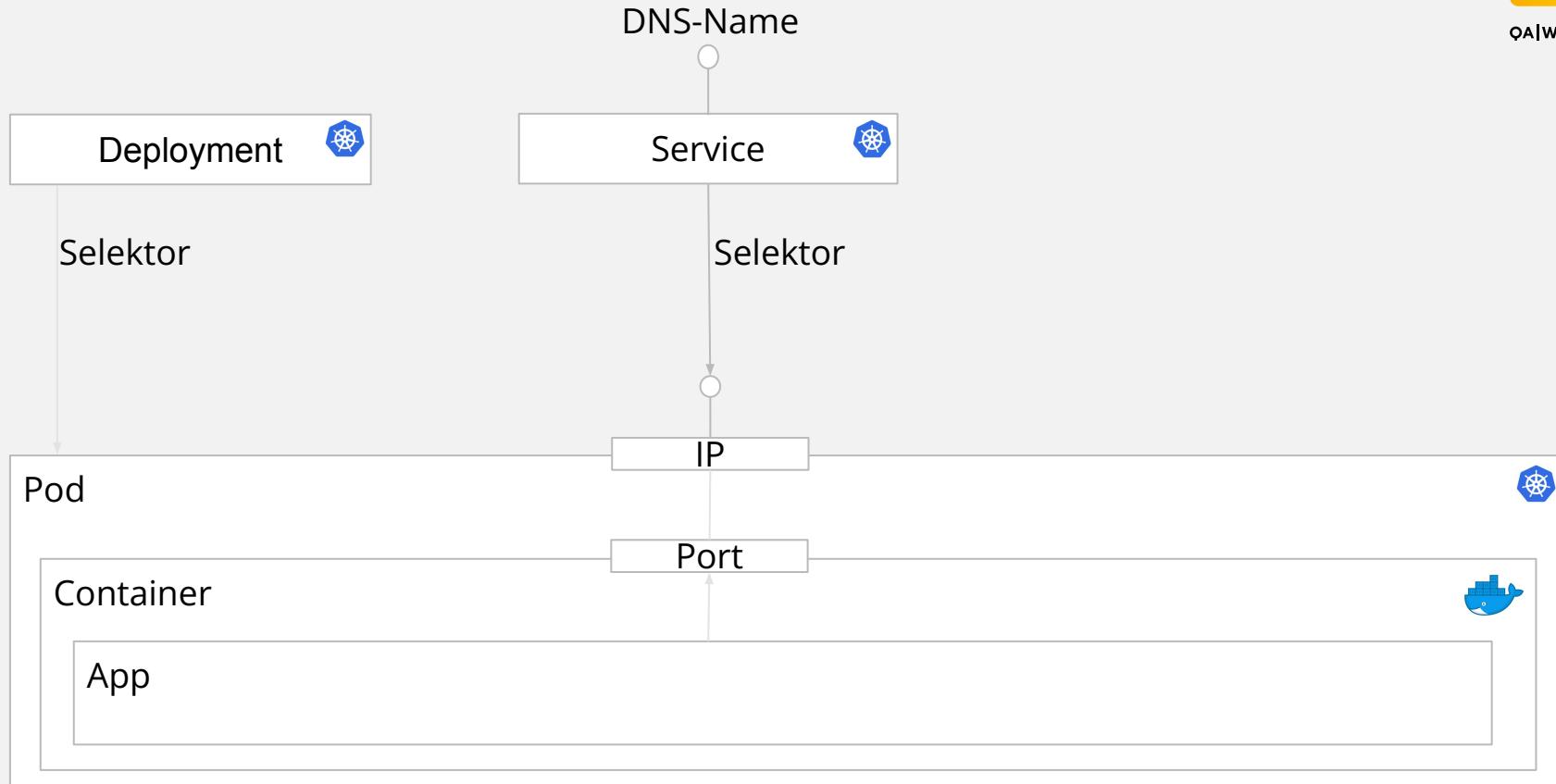
QA|WARE

Config Maps

Wichtige Kubernetes-Konzepte



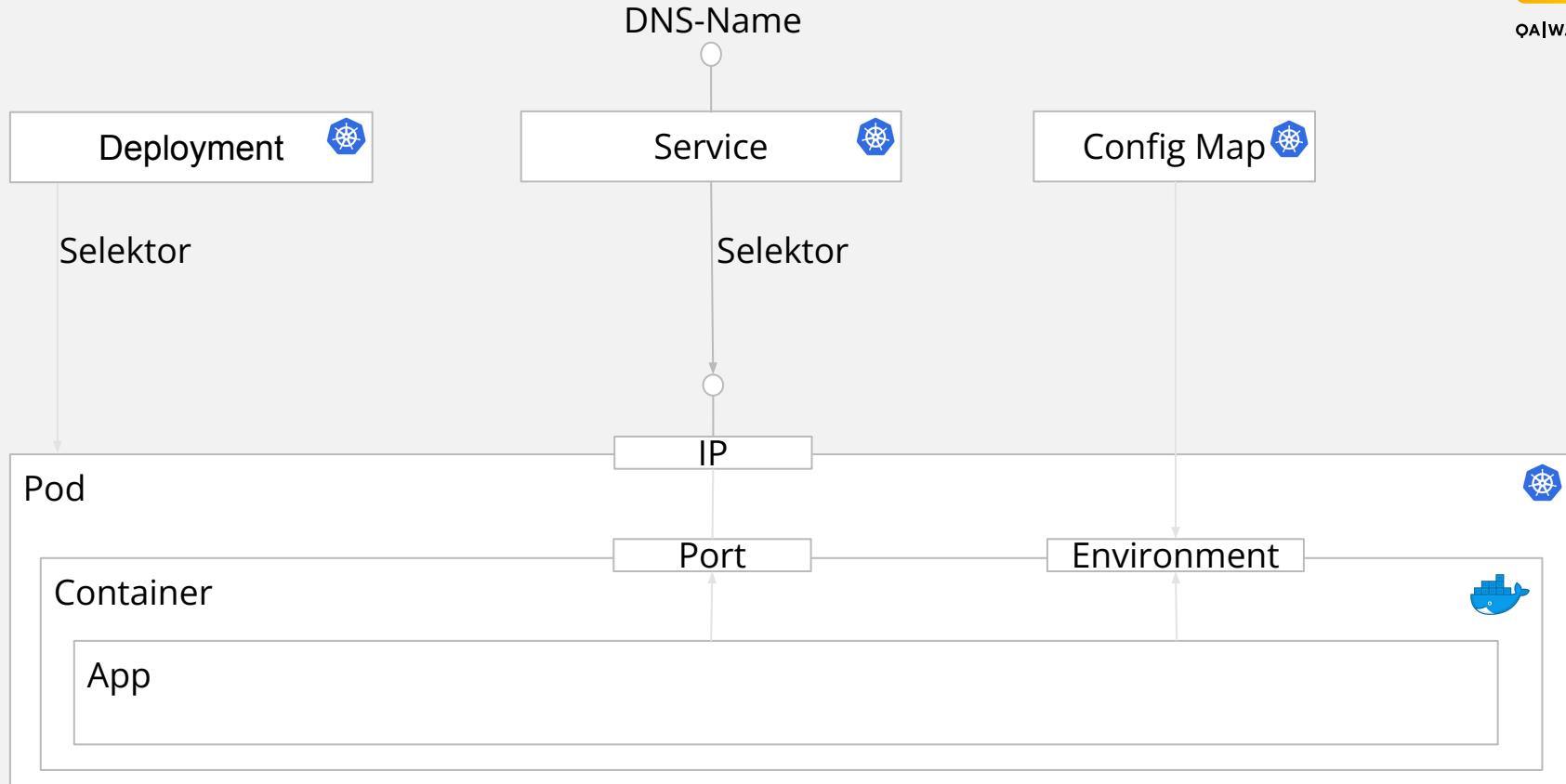
QA|WARE



Wichtige Kubernetes-Konzepte



QA|WARE



Konfiguration: Config Maps (1)



```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  PLAYER_INITIAL_LIVES: "3"
  UI_PROPERTIES_FILE_NAME: "user-interface.properties"
```

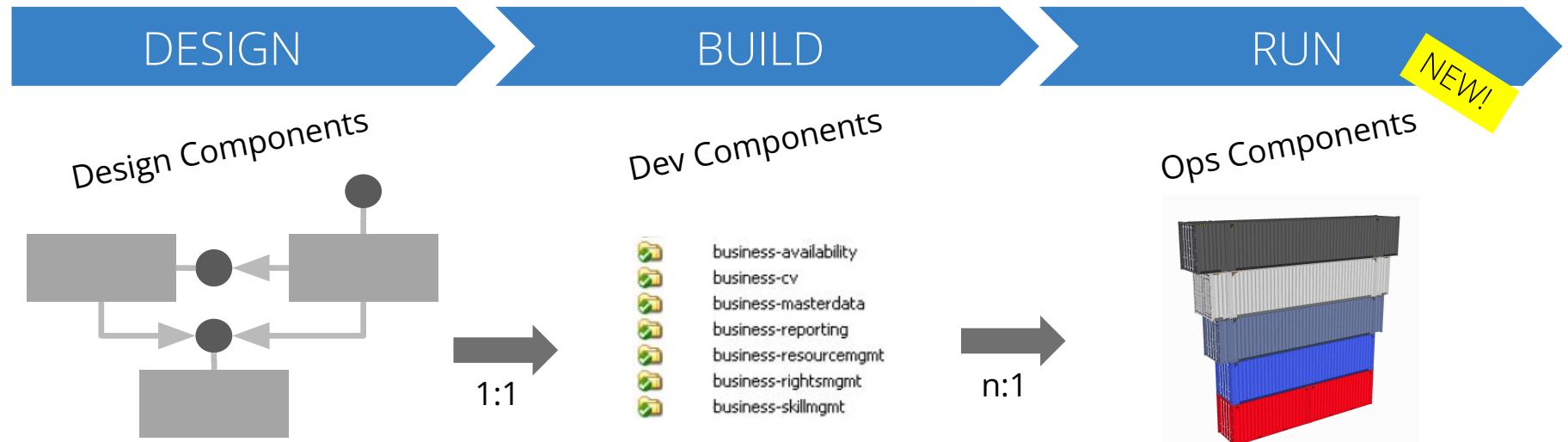
Konfiguration: Config Maps (2)



```
apiVersion: v1
kind: Pod
metadata:
  name: configmap-demo-pod
spec:
  containers:
    - name: demo
      image: alpine
      command: ["sleep", "3600"]
      # Import all environment variables from ConfigMap.
      # Keys should already be in screaming snake case.
      envFrom:
        - configMapRef:
            name: game-demo
```

Kapitel Cloud Architektur

Cloud Native Application Development: Components All Along the Software Lifecycle

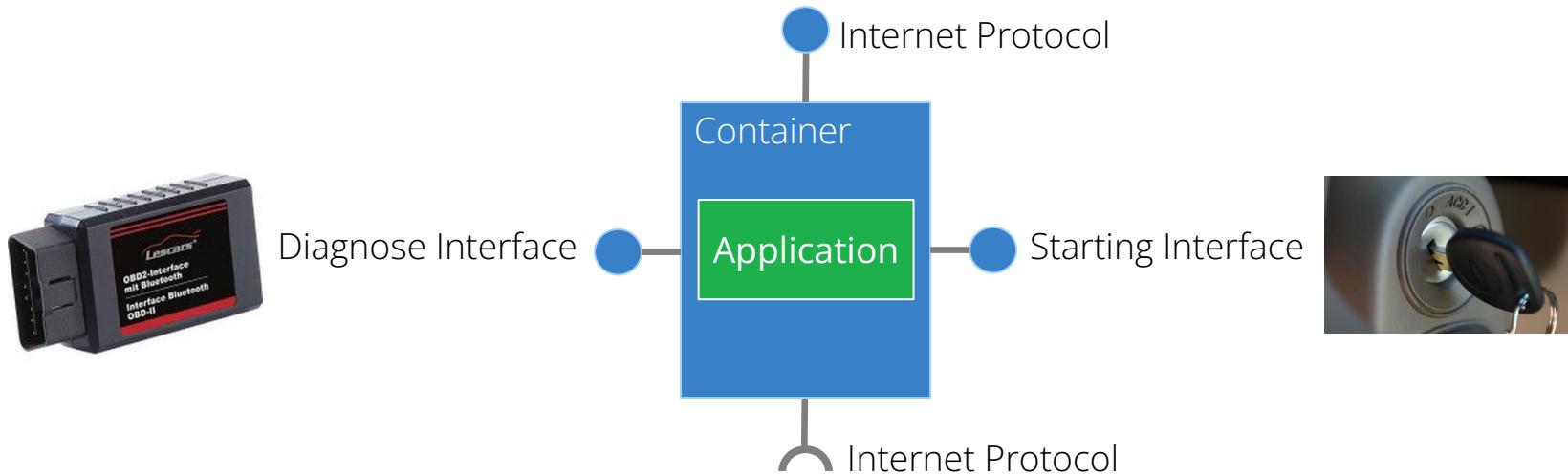


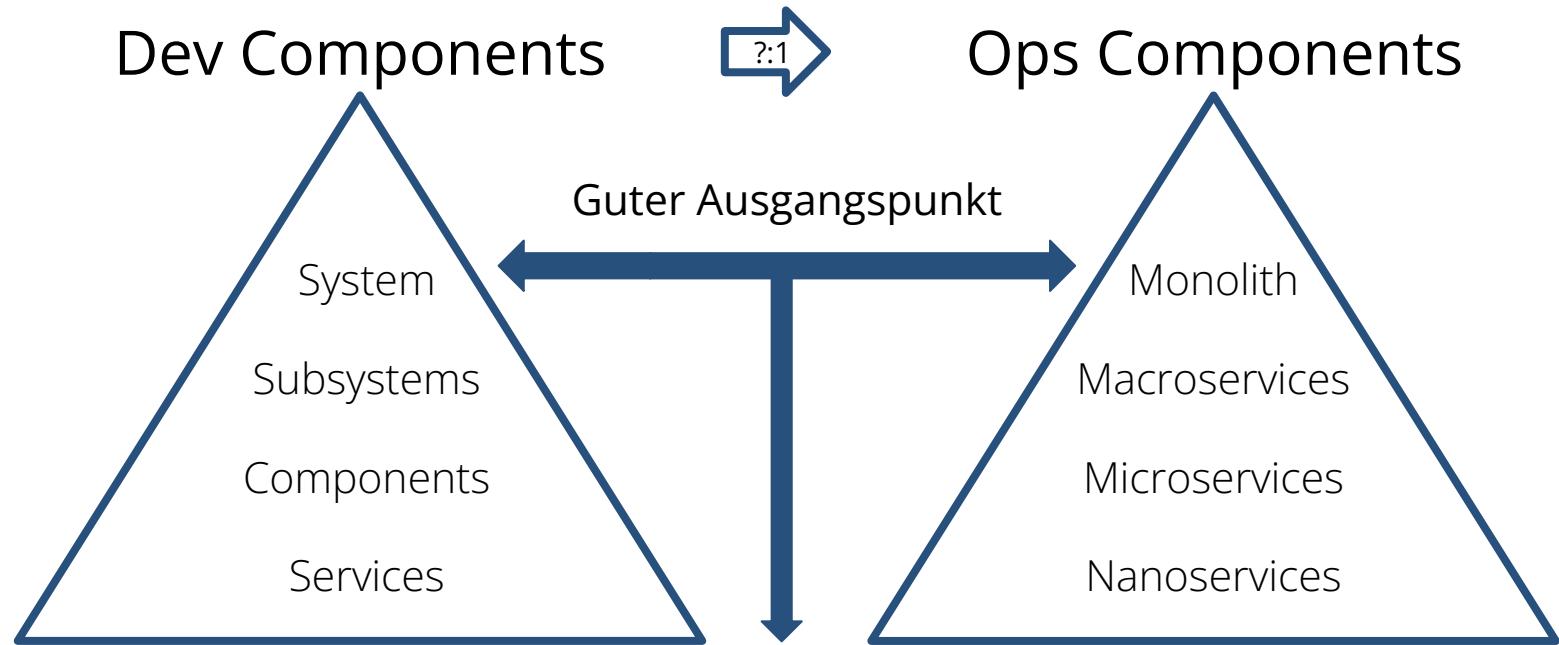
- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

- 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

The background of the slide features a complex, abstract network structure composed of numerous small, semi-transparent white dots connected by thin white lines, forming a web-like pattern across the dark blue background.

Regeln, Gebote, Trugschlüsse

12 Factor App

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

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

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

Resilienz



Resilienz

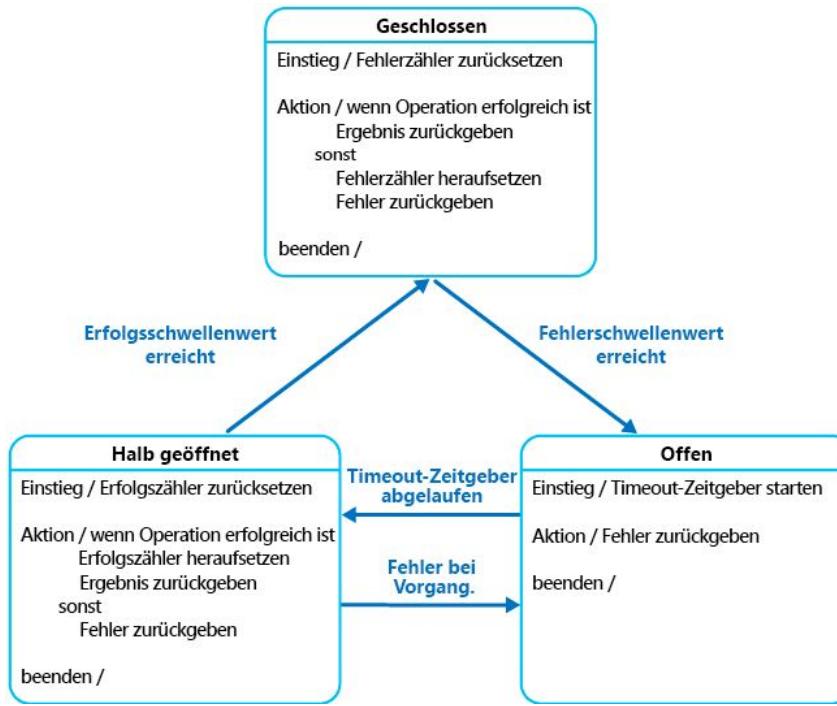
$$\text{Verfügbarkeit} = \text{MTTF} / (\text{MTTF} + \text{MTTR})$$



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

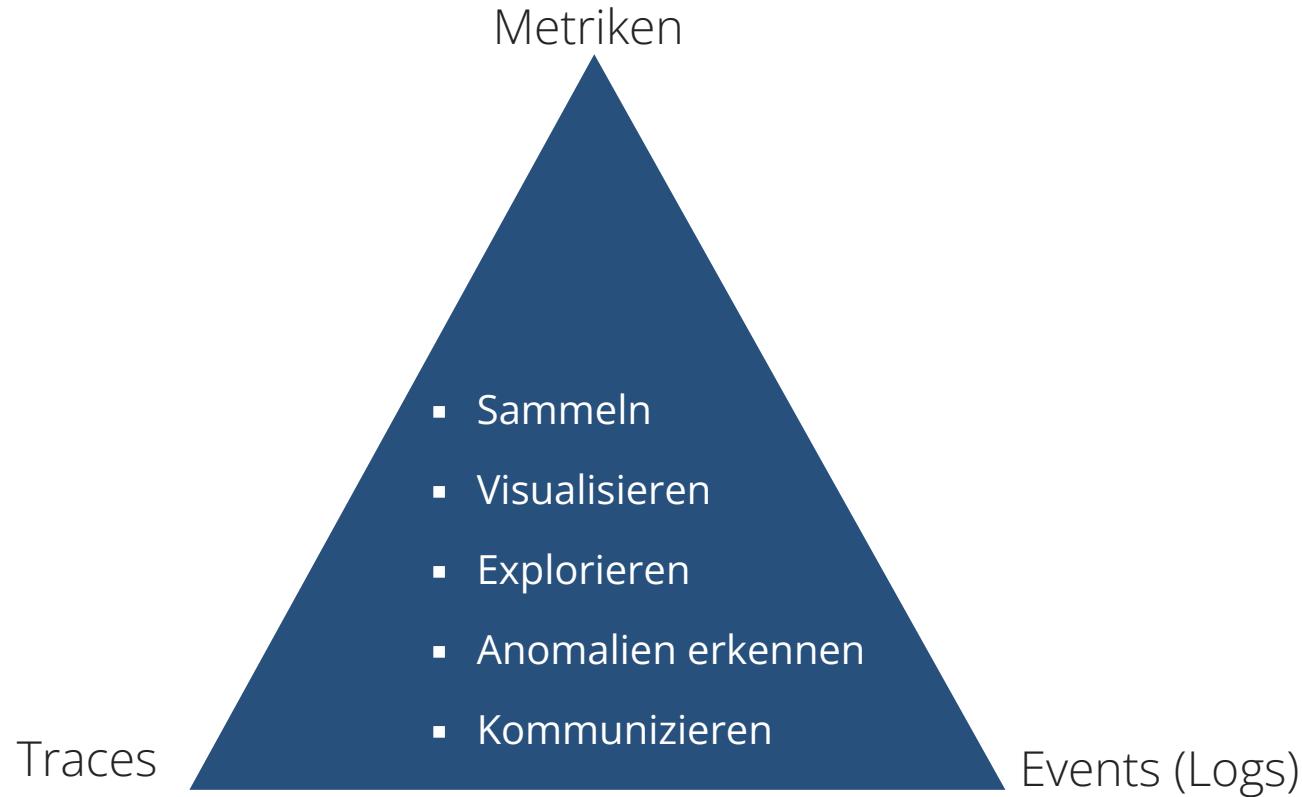
- Ohne dass es der Nutzer merkt (Bestfall)
- Mit einer „graceful degradation“ des Services (schlechtester Fall)

Resilienz-Pattern: Circuit Breaker



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

Diagnostizierbarkeit



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

Edge Server

Service Container

Service

Typische Aufgaben:

- Service Discovery
- Load Balancing
- Circuit Breaker
- Request Monitoring

Service Client

Diagnose-stecker

Typische Aufgaben:

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

Typische Aufgaben:

- Metriken sammeln
- Logs sammeln
- Trace sammeln

Configuration & Coordination

Service Discovery

Typische Aufgaben:

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

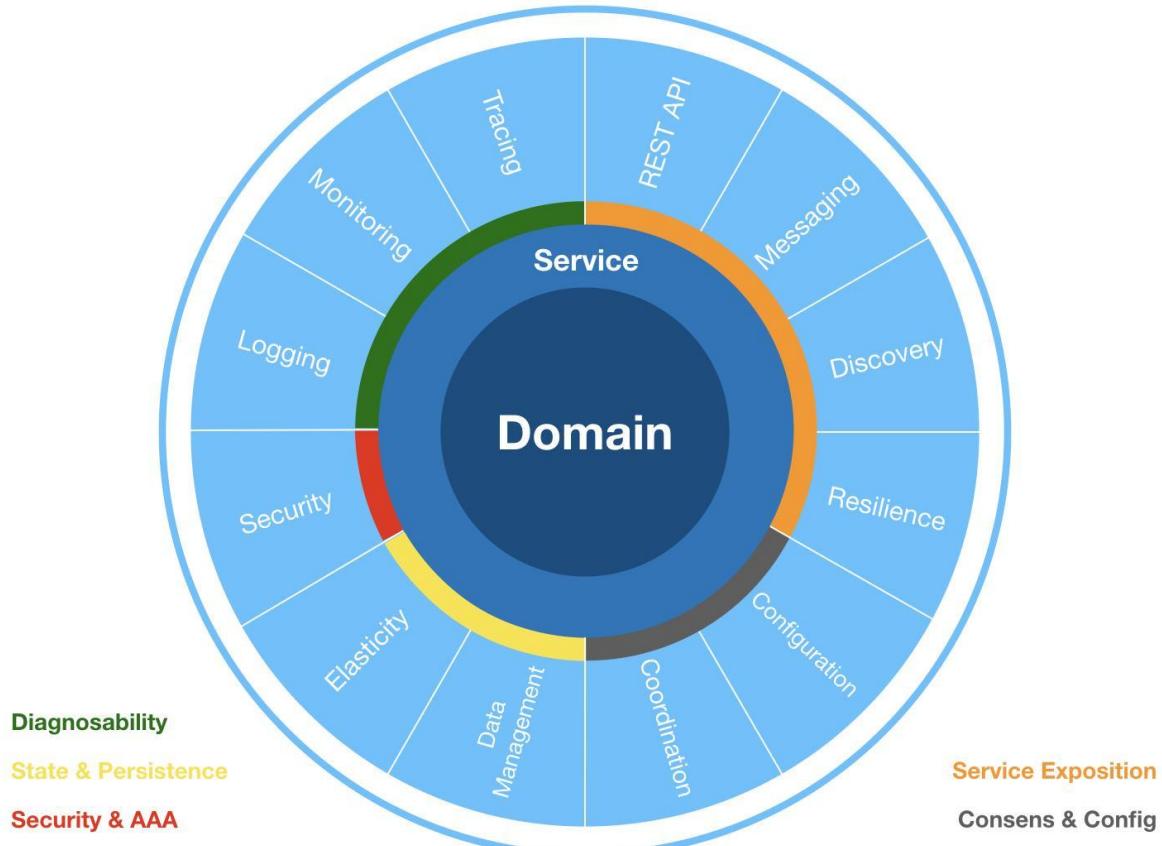
Typische Aufgaben:

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

Typische Aufgaben:

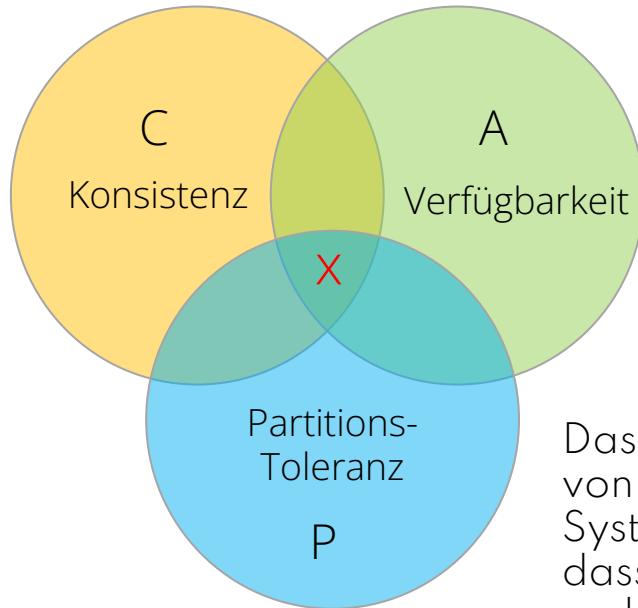
- Key-Value-Store (oft in Baumstruktur, teilw. mit Ephemeral Nodes)
- Sync von Konfigurationsdateien
- Watches, Notifications, Hooks, Events
- Koordination mit Locks, Leader Election und Messaging
- Konsens im Cluster herstellen

Technische Aspekte von Microservices



Das CAP-Theorem

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



Im Falle einer Netzwerk-Partition muss man sich zwischen Konsistenz und Verfügbarkeit entscheiden, beides geht nicht.

Jede Anfrage erhält eine Antwort (keinen Fehler) in akzeptabler Zeit. 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 für Hoch-Verfügbarkeit

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

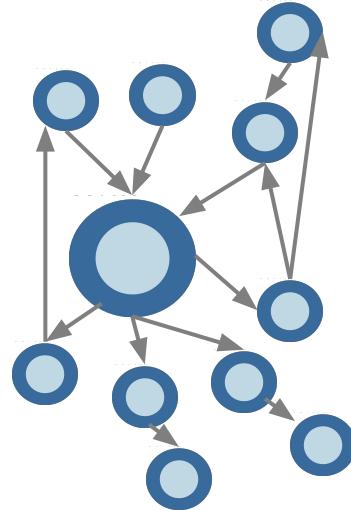
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

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



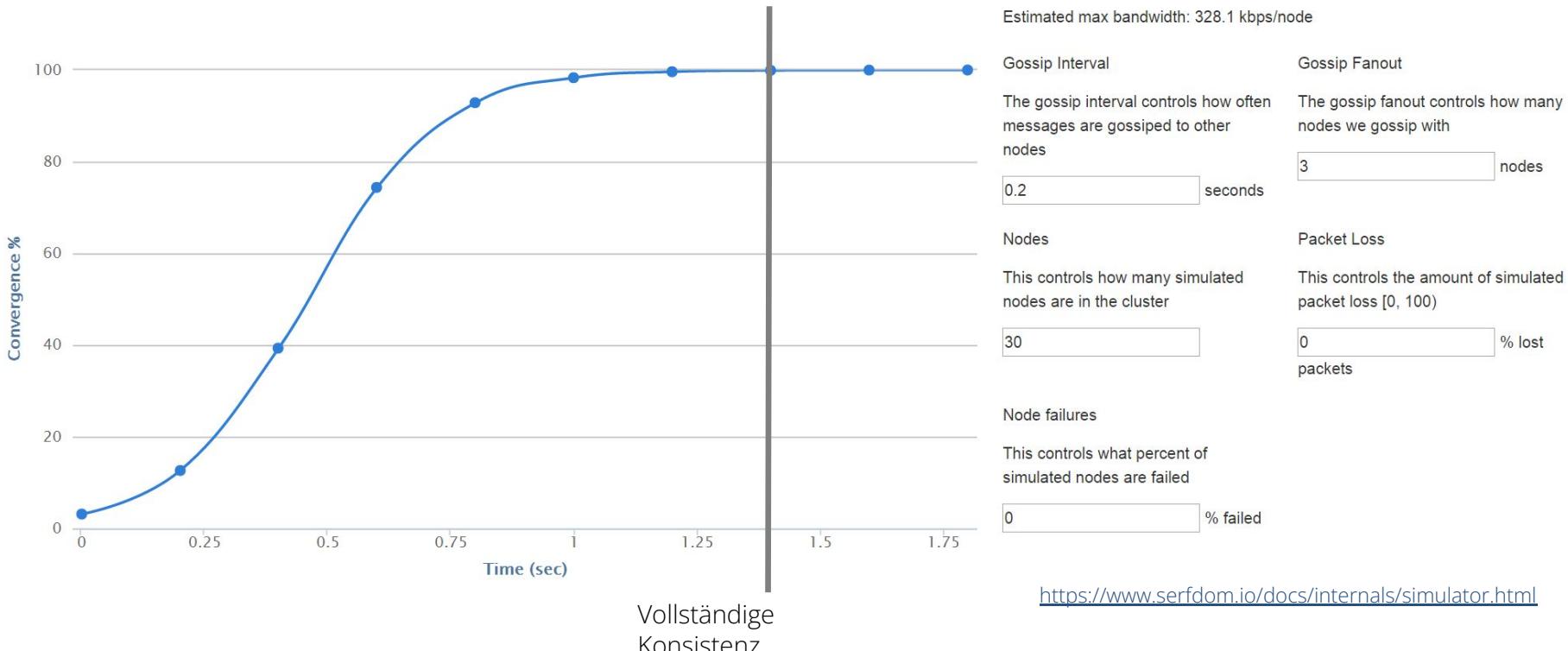
Vorteile:

- 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.

Die Konvergenz der Daten und damit der Zeitpunkt der vollständigen Konsistenz ist berechenbar



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.

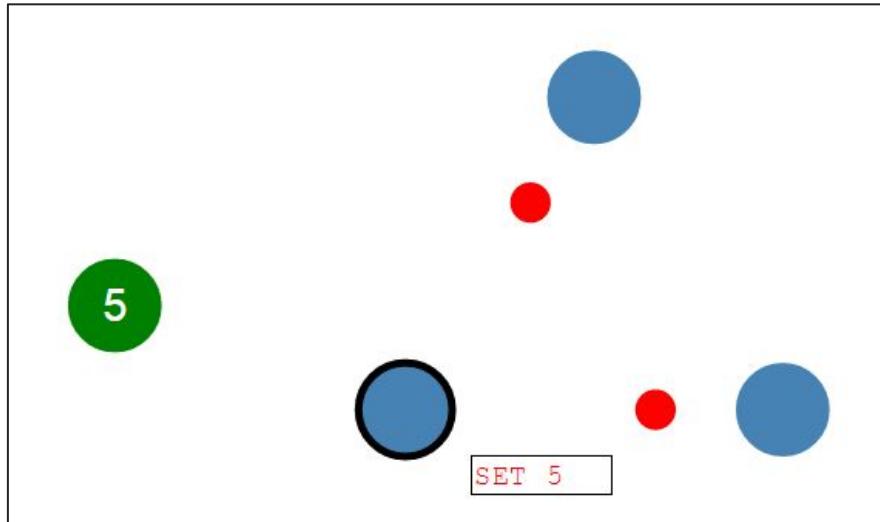
Nachteile:

- 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".

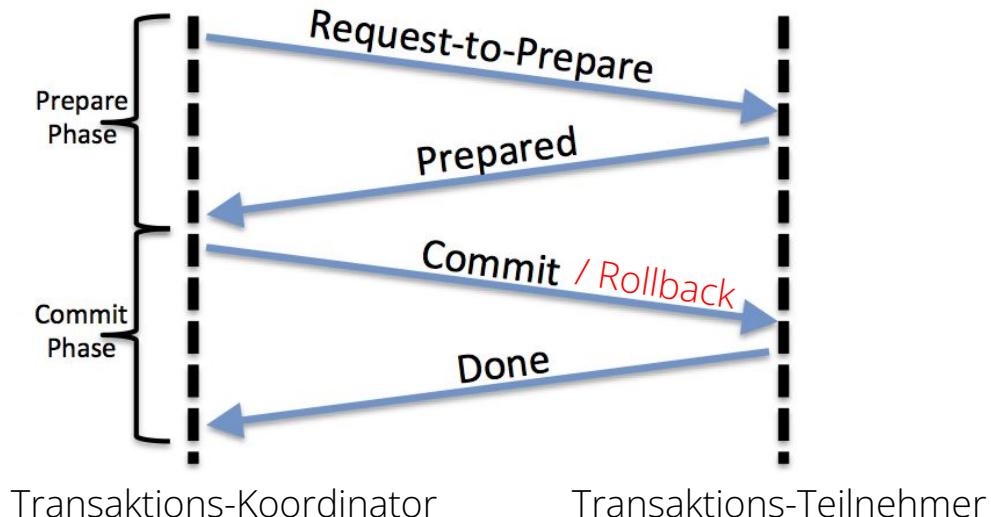


<http://thesecretlivesofdata.com/raft>

<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.



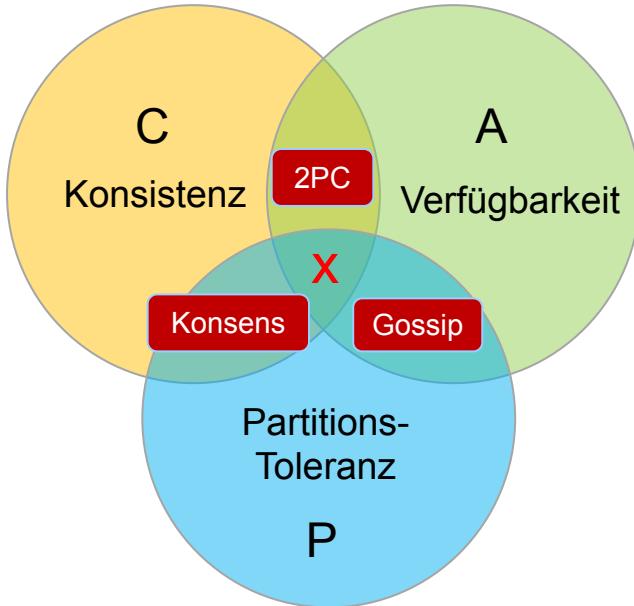
Vorteil:

- Alle Knoten sind konsistent zueinander.

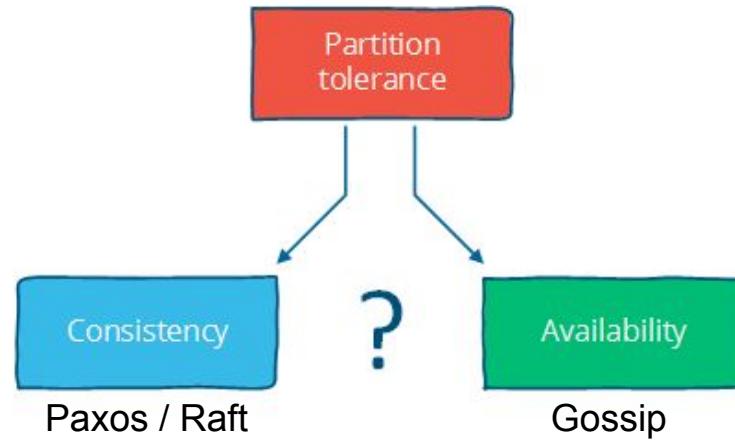
Nachteile:

- 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.



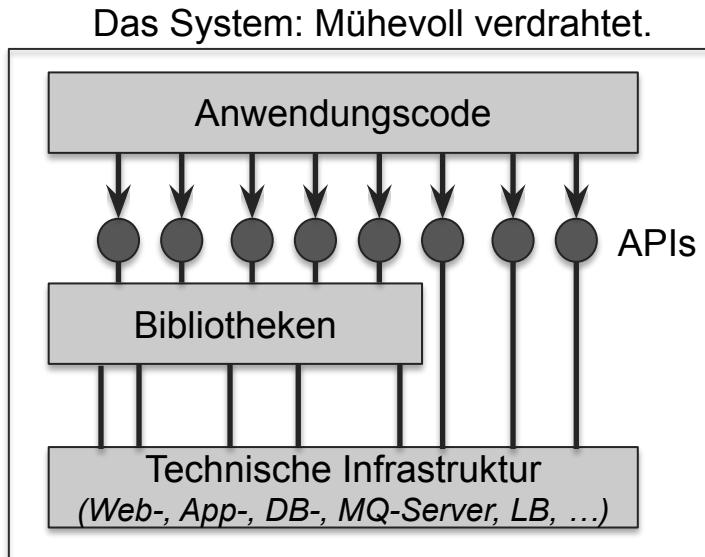
Kapitel Platform-as-a-Service



Ein Problem mit IaaS: Sie müssen Anwendungen aufwändig von Hand verdrahten

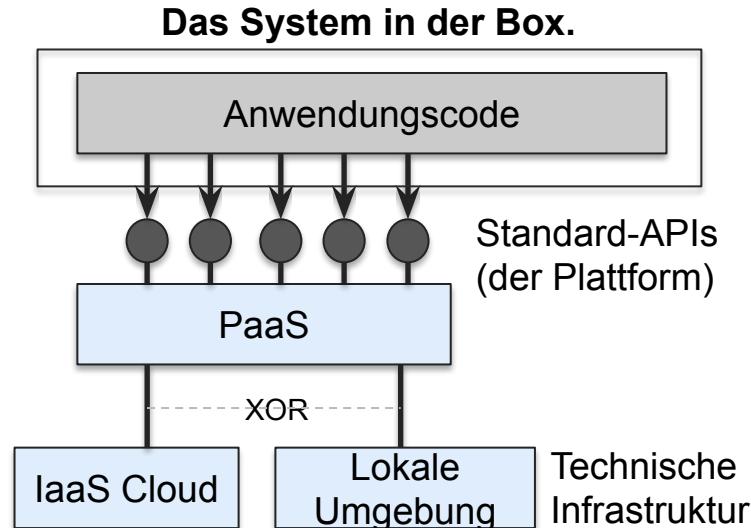


Stovepipe
Architecture

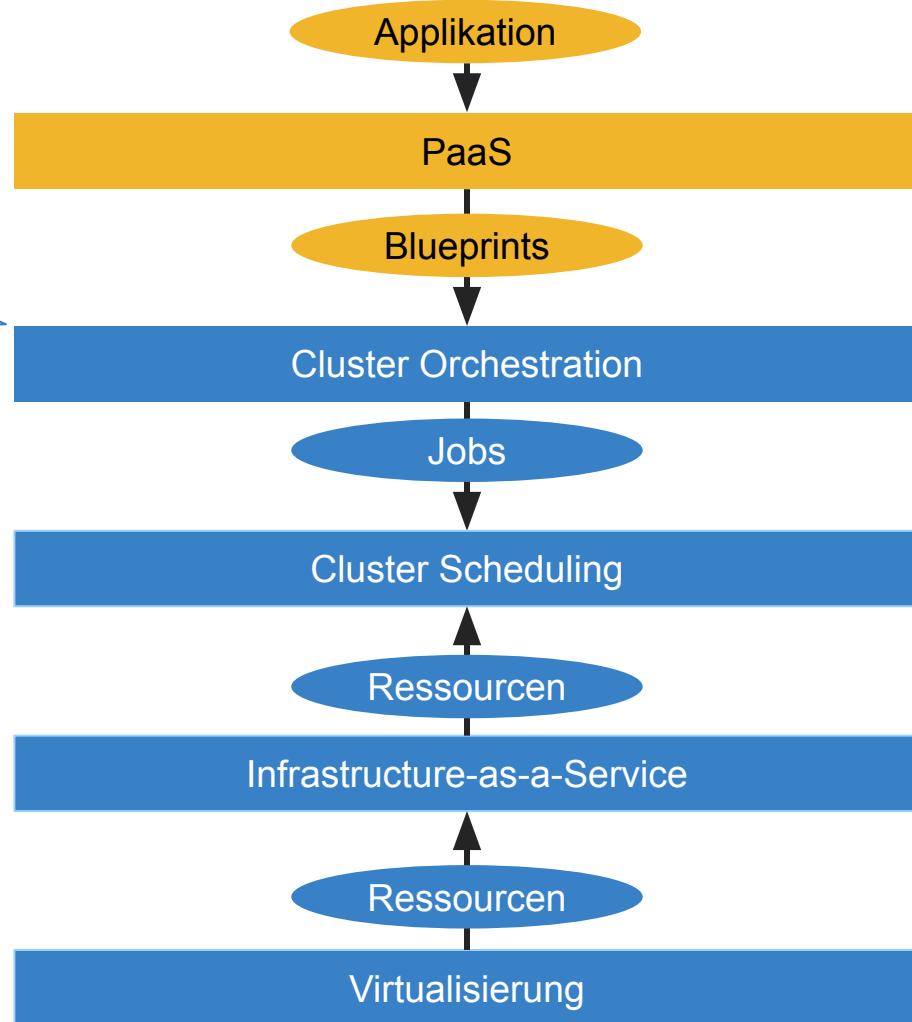


Lösung: Plattform-as-a-Service bietet eine Entwicklungs- und Betriebsplattform mit vorgegebenen APIs

- 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.



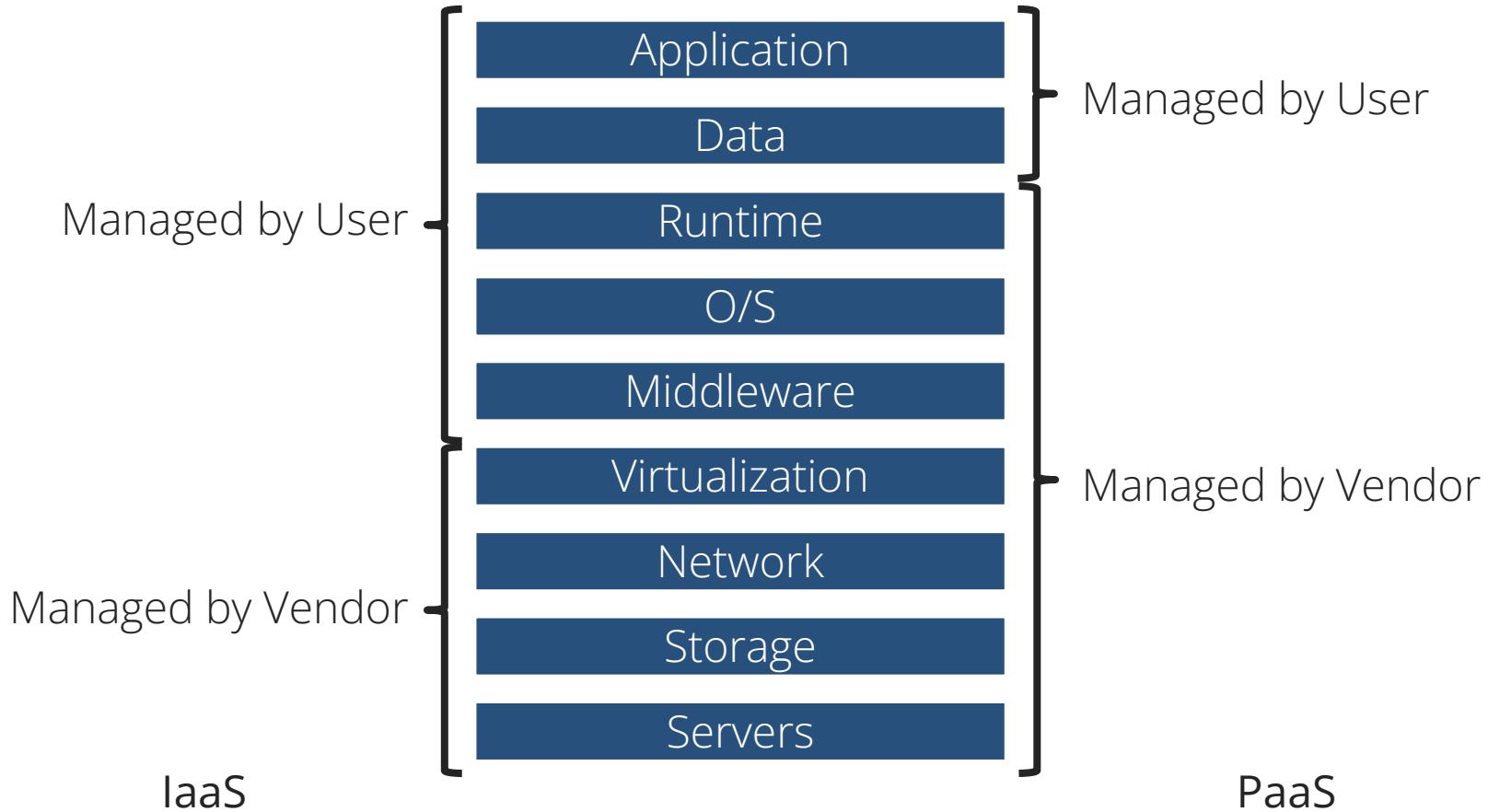
Das Big Picture



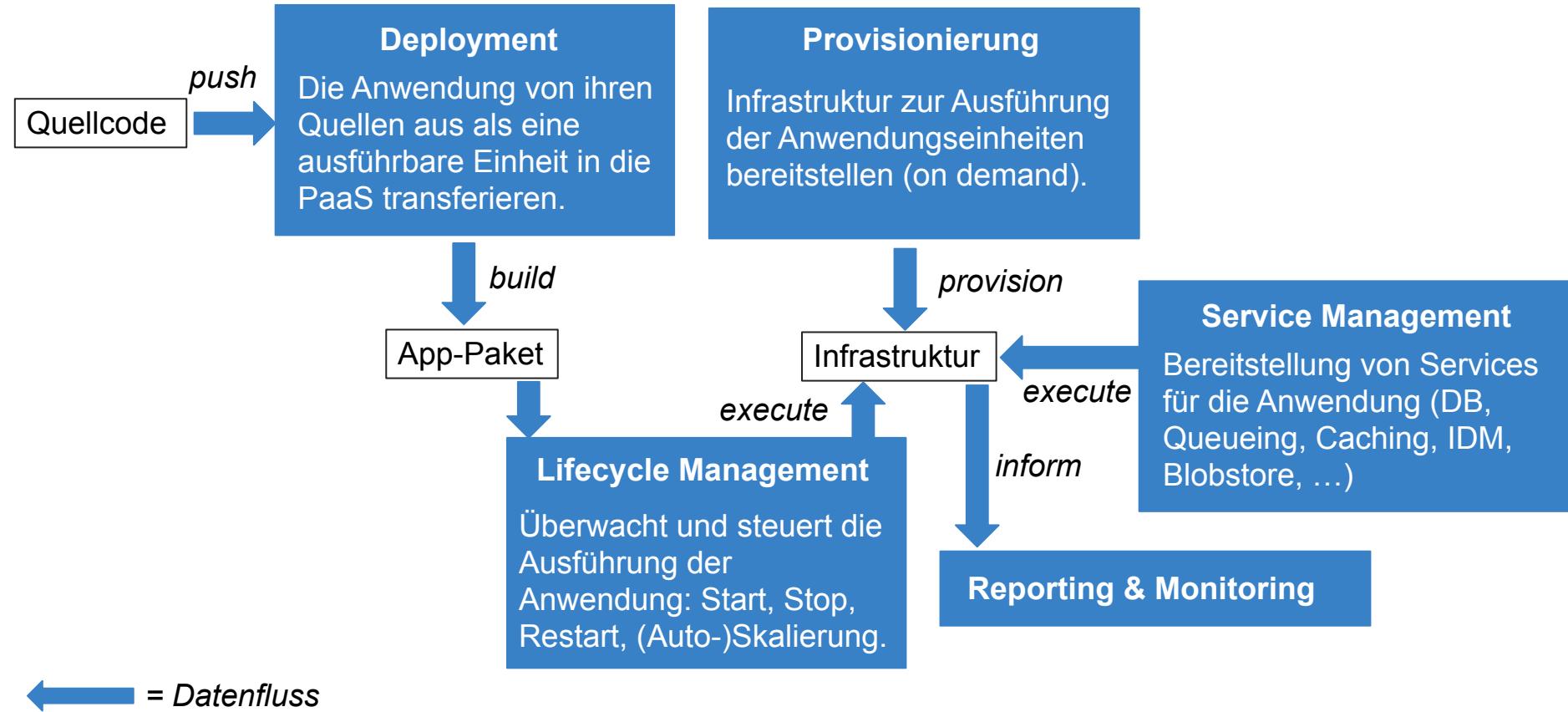
Hier ist man bereits bei 80% einer PaaS. Was noch fehlt:

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

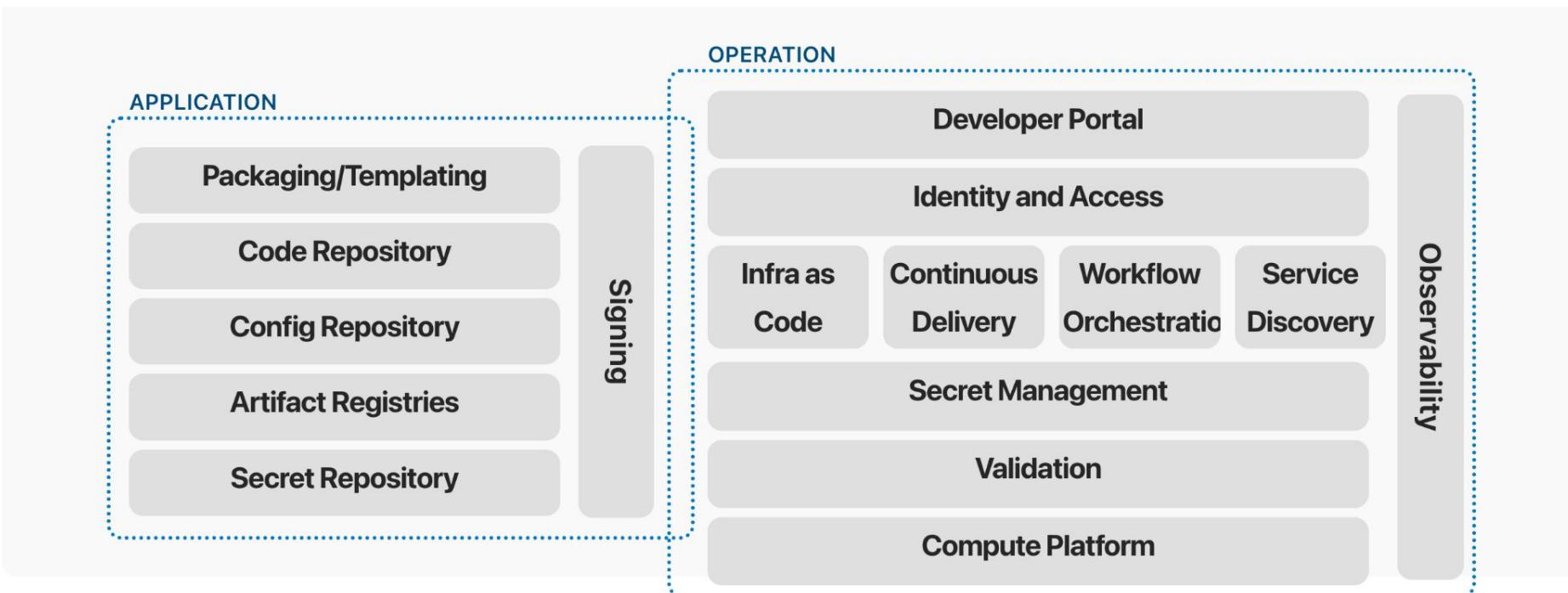
IaaS vs. PaaS



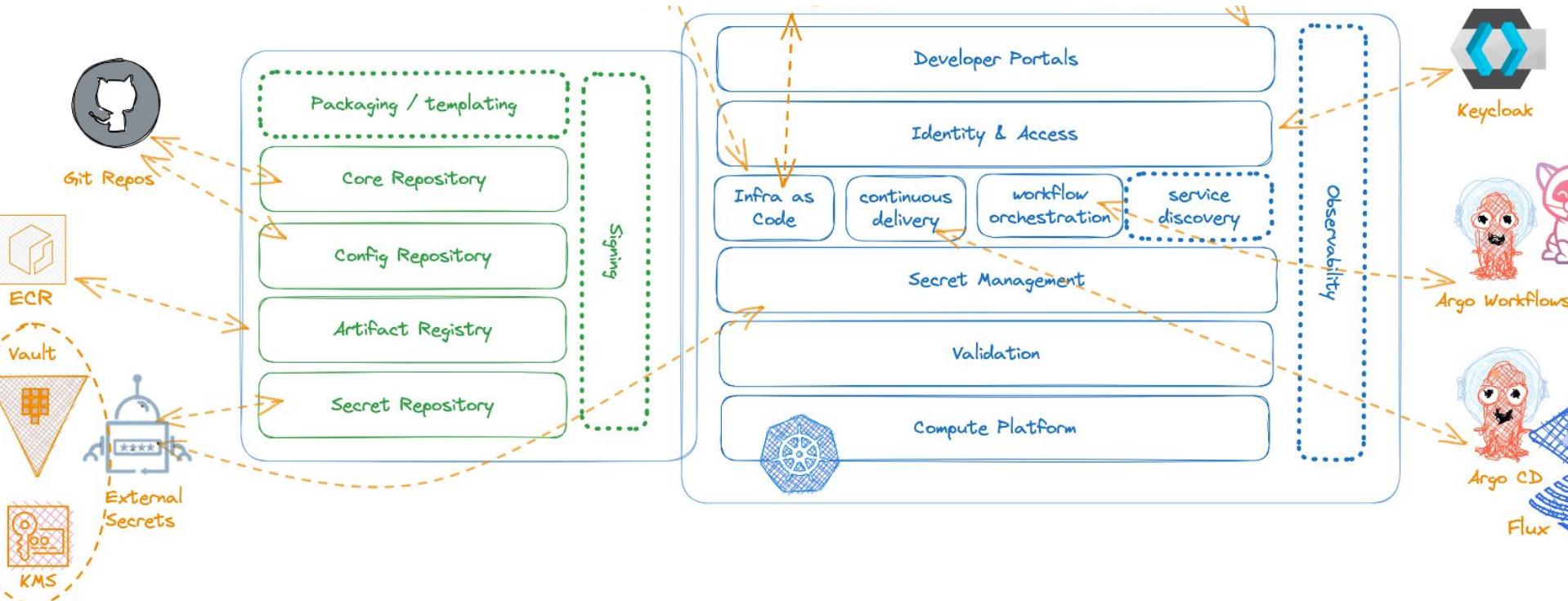
Die funktionalen Bausteine einer PaaS Cloud



IDP: Internal Developer Platform (Beispiel: CNOE)



IDP: Internal Developer Platform (Beispiel: CNOE)



Backstage als IDP-Enabler

Create a New Component | Temp x +

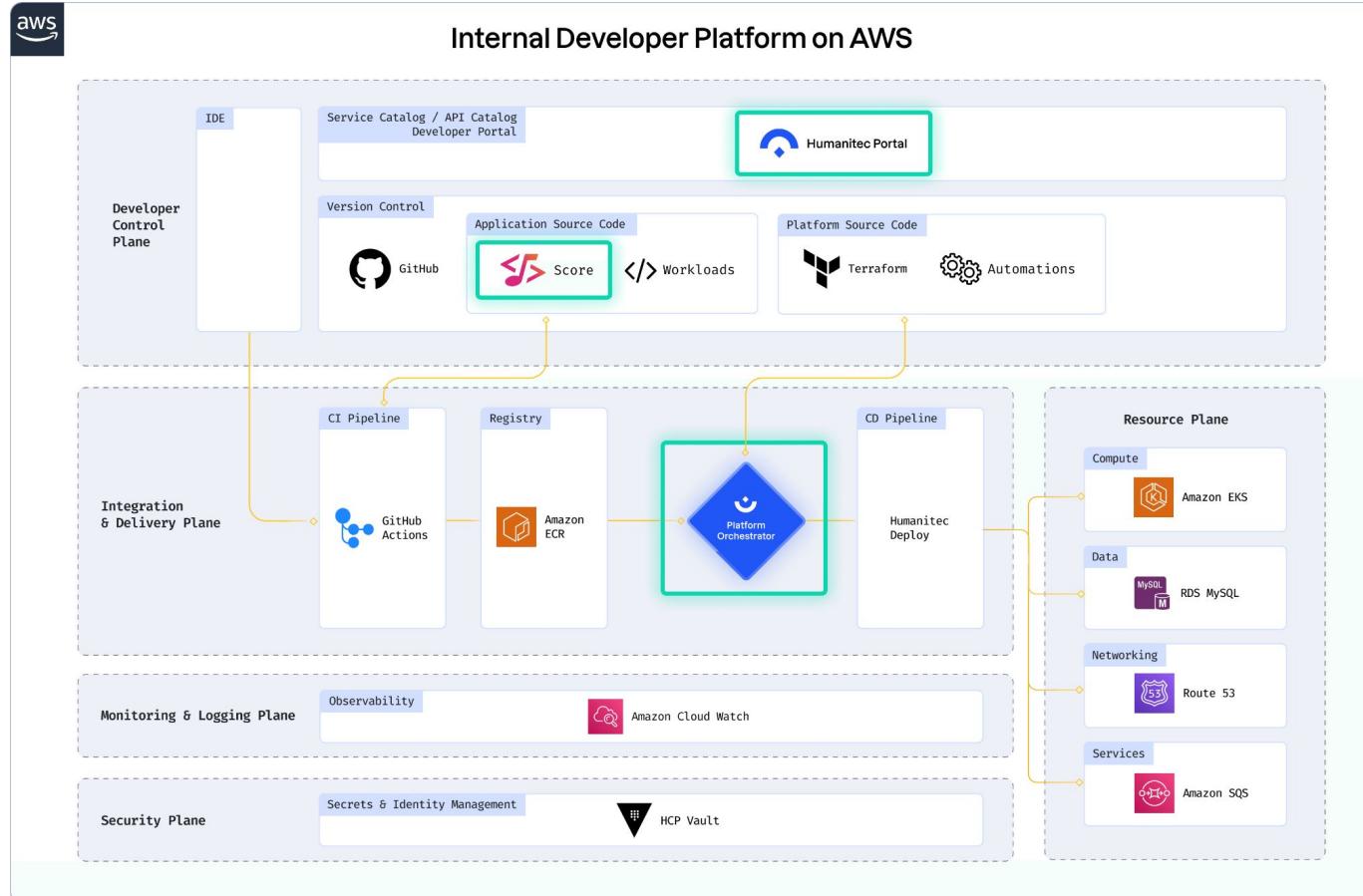
localhost:3000/create?filters%5Bkind%5D=template&filters%5Buser%5D=all

The screenshot shows the Backstage application interface. On the left, there's a sidebar with various navigation items: Backstage, Search, Home, APIs, Docs, Create..., and Tech Radar. The main area is titled "Create a New Component" and has a sub-section "Available Templates". A search bar is present at the top of the template list. Three templates are displayed in cards:

- Documentation Template** (pink card):
 - DESCRIPTION**: Create a new standalone documentation project.
 - OWNER**: backstage/techdocs-core
 - TAGS**: recommended, techdocs, mkdocs
- React SSR Template** (blue card):
 - DESCRIPTION**: Create a website powered with Next.js
 - OWNER**: web@example.com
 - TAGS**: recommended, react
- Spring Boot gRPC Service** (teal card):
 - DESCRIPTION**: Create a simple microservice using gRPC and Spring Boot Java
 - OWNER**: service@example.com
 - TAGS**: recommended, java, grpc

At the bottom right of the main area, there are buttons for "REGISTER EXISTING COMPONENT" and a question mark icon.

IDP-Komponenten (Beispiel: Humanitec)



Kapitel Continuous Delivery

Continuous Delivery - Definition



QA|WARE

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 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)

Kriterien für Continuous Delivery



QA|WARE

“You’re doing continuous delivery when:

- Your software is deployable throughout its lifecycle
- Your team prioritizes keeping the software deployable over working on new features
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them
- You can perform push-button deployments of any version of the software to any environment on demand”

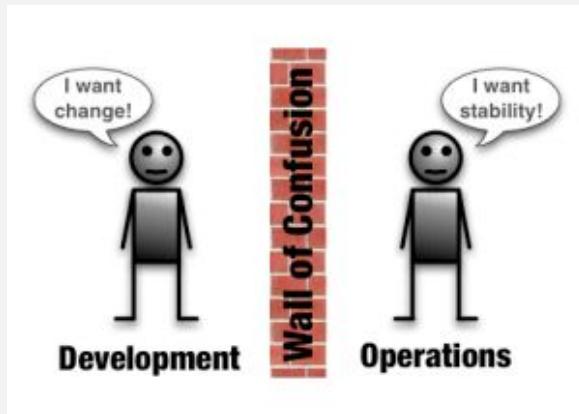
nach M. Fowler / Continuous Delivery working group at ThoughtWorks

Was ist eigentlich DevOps?



QA|WARE

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

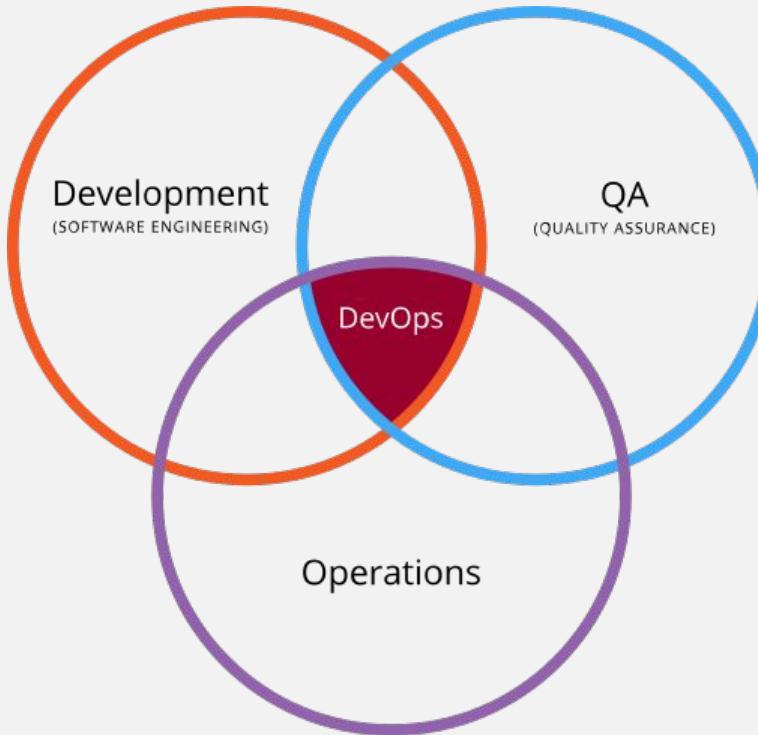
Pro Feature:

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

DevOps verbindet DEVelopment, OPerations und Quality Assurance



QA|WARE



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

DevOps Topologies



QA|WARE

Nicht überall wo DevOps draufsteht, ist auch DevOps drin.

Zahlreiche Anti-Types und Types finden sich auf:

<https://web.devopstopologies.com/>

DevOps Anti-Types

It's useful to look at some bad practices, what we might call 'anti-types' (after the ubiquitous 'anti-pattern').

A: Dev vs Ops B: DevOps Silo C: No Ops Needed D: Tools Team E: SysAdmin F: Embedded Ops G: Dev vs DBA H: Fake SRE

Anti-Type A: Dev and Ops Silos

This is the classic 'throw it over the wall' split between Dev and Ops. It means that story points can be claimed early (DONE means 'feature-complete', but not working in Production), and software operability suffers because Devs do not have enough context for operational features and Ops folks do not have time or inclination to engage Devs in order to fix the problems before the software goes live.

We likely all know this topology is bad, but I think there are actually worse topologies; at least with Anti-Type A (Dev and Ops Silos), we know there is a problem.

Anti-Type B: DevOps Team Silo

The DevOps Team Silo (Anti-Type B) typically results from a manager or exec deciding that they "need a bit of this DevOps thing" and starting a 'DevOps team' (probably full of people known as 'a DevOp'). The members of the DevOps team quickly form another silo, keeping Dev and Ops further apart than ever as they defend their corner, skills, and toolset from the 'clueless Devs' and 'dinosaur Ops' people.

The only situation where a separate DevOps silo really makes sense is when the team

Image based on work at [devopstopologies.com](https://web.devopstopologies.com/) - licensed under CC BY-SA.



QA|WARE

GitOps

1 Declarative

A system managed by GitOps must have its desired state expressed declaratively.

2 Versioned and Immutable

Desired state is stored in a way that enforces immutability, versioning and retains a complete version history.

3 Pulled automatically

Software agents automatically pull the desired state declarations from the source.

4 Continuously reconciled

Software agents continuously observe actual system state and attempt to apply the desired state.

Die Vorteile



- Ermöglicht im Idealfall einen beliebigen Systemzustand in der Historie wiederherzustellen, e.g. einfacher Rollback
- Forciert Pipelines
- Bietet Transparenz von Änderungen, im Falle von Git ist auch ersichtlich wer etwas geändert hat
- Stellt sicher, dass der Systemzustand nicht vom Zielzustand abweicht

GitOps im K8s Umfeld



QA|WARE



ArgoCD

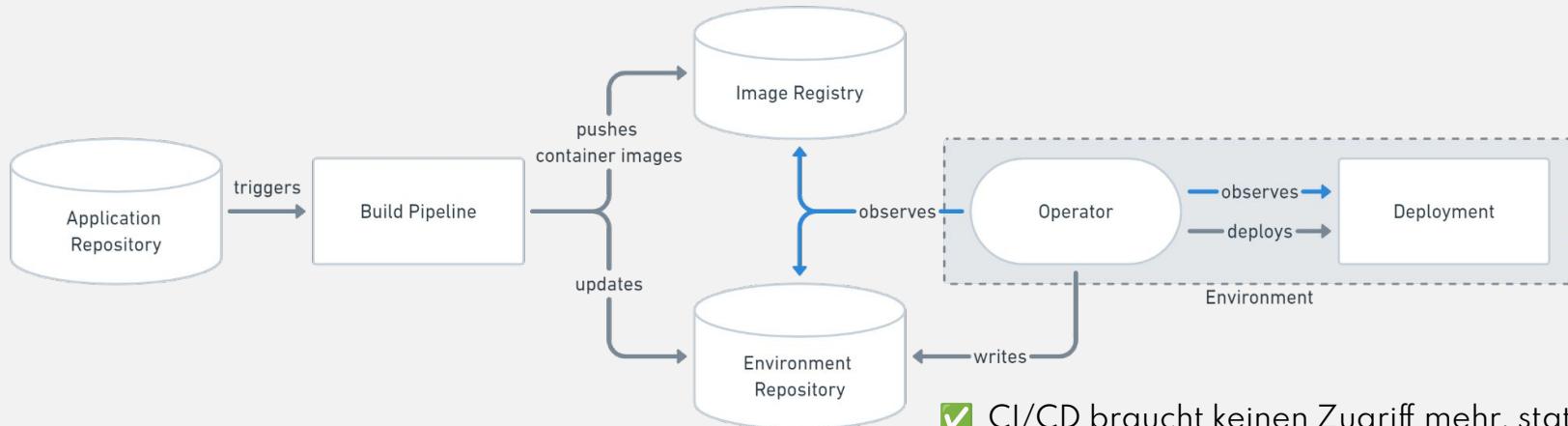


Flux

Pull-based deployments mit GitOps



QA|WARE



- ✓ CI/CD braucht keinen Zugriff mehr, stattdessen greift der Operator auf das Environment Repo und Registry zu
- ✓ Operator versucht den gewünschten Cluster-Zustand herzustellen, inklusive Löschen von Ressourcen
- ✗ Komplexer

Unterschiede in der Architektur

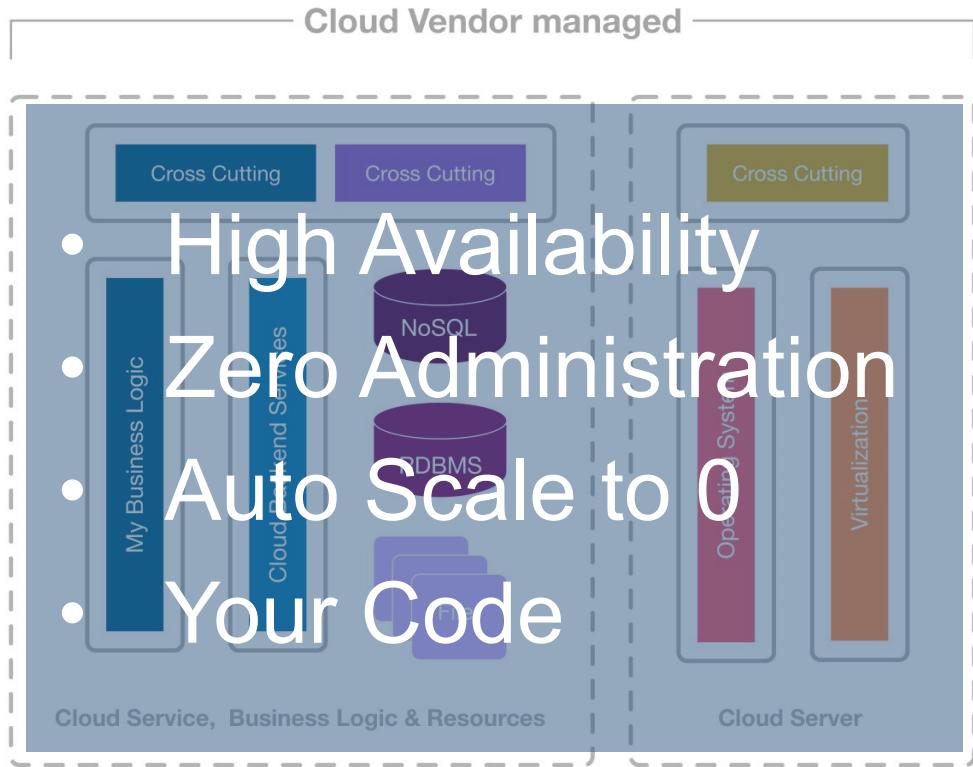
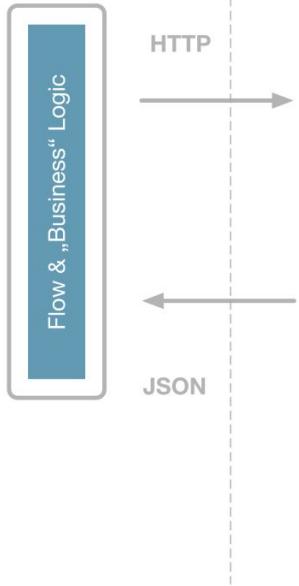


- CI / CD Pipeline braucht keinen cluster Zugriff mehr
 => verbesserte Security, da für die Pipeline kein high privilege credential mehr notwendig ist
- Cluster pullen jetzt aktiv Source-Repos, welche den Zielzustand enthalten
- Cluster gleichen ihren Zustand ab (reconcile), d.h. es wird der Ist-Zustand mit dem Soll-Zustand abgeglichen und versucht den Soll-Zustand zu erreichen.
 => Cluster konvergieren automatisch gegen den Sollzustand

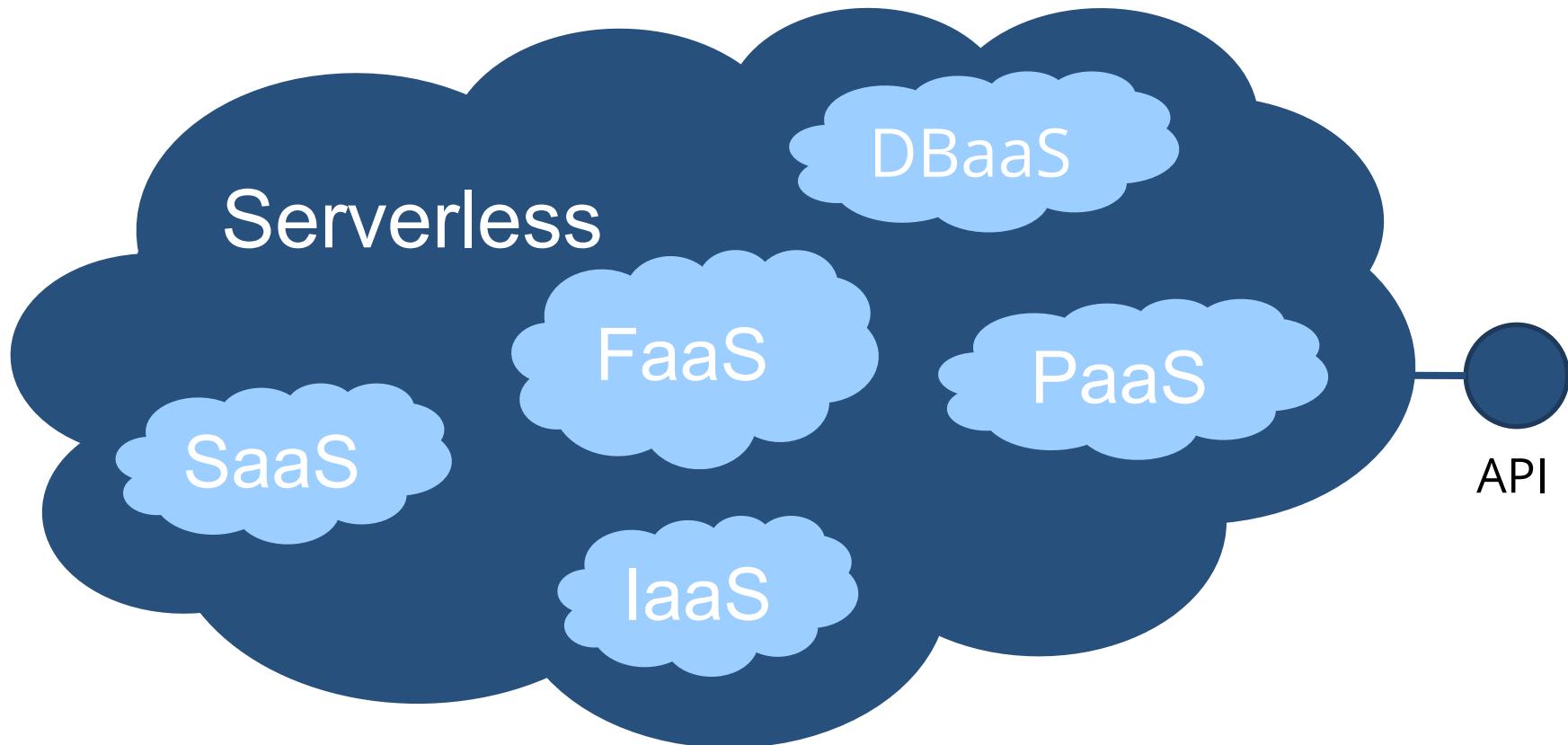
Kapitel Serverless

Serverless Anwendungsarchitektur

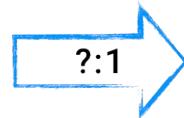
Run Code, not Servers!



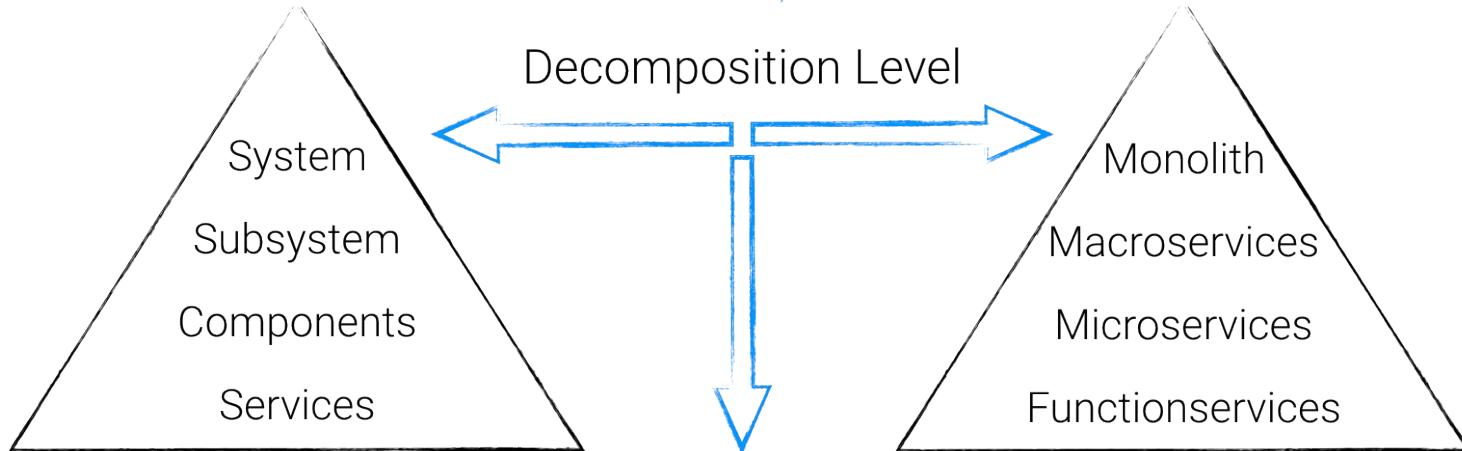
Out-of the Box Self-scaling Fully Managed Backend



Dev Components



Ops Components

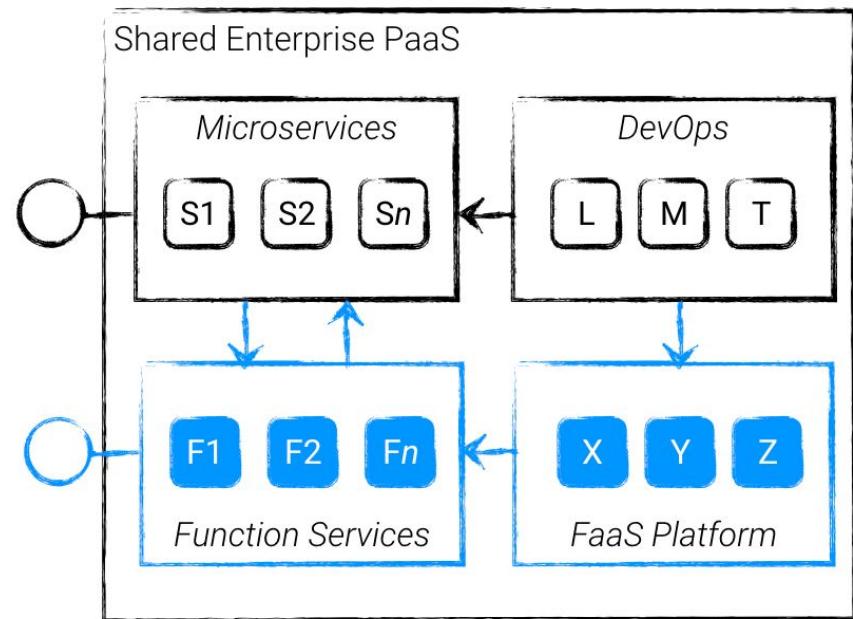


Decomposition Trade-Offs

- | | |
|--|---|
| <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 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 Observability

Was ist eigentlich Observability?



QA|WARE

“Observability” meint heute typischerweise vier Themen:

- Logs
- Metriken
- Traces

Alles zusammen ermöglicht einen Einblick auch in verteilte Microservices.



Ein System ist gut diagnostizierbar, wenn man gesunde und ungesunde Zustände leicht erkennen und beheben kann.

Ein diagnostizierbares System hat eine



- kurze Mean Time to Detect (MTTD)
- kurze Mean Time to Repair (MTTR)

und somit zu einer kurze Zeitspanne, in der Fehler unerkannt sind und überhaupt existieren.

Diagnosability: Strukturiertes Vorgehen ist notwendig, um das System nicht zu sehr zu beeinflussen



1. Überblick schaffen:

- a. Was sind zentrale Komponenten, z. B. Login, etc.
- b. Was sind unterstützende Komponenten, z. B. Batch-, Loader-Jobs, etc.

2. Fehlergrenzen identifizieren:

- a. Interne Fehlergrenzen: Schichten / Use Cases
- b. Externe Fehlergrenzen: Ein- und ausgehende Aufrufe

3. Fehlerklassen definieren:

- a. Schweregrade: Betrieb weiterhin möglich, Keine Auswirkungen vor Kunde, etc.
- b. Auswirkungen: Kunden stehen bestimmte Funktionen nicht zur Verfügung, etc.

4. Laufzeitdaten bestimmen, die zur Erkennung notwendigen sind:

- a. Einheitlichkeit: Daten haben dieselbe Bedeutung; Einheitliche Datenformate sind definiert, etc.
- b. Klar definiert: Es ist ersichtlich, was die Metrik bedeutet, z. B. CPU-Load

5. Handlungen definieren:

- a. Alerting: Wer wird wann benachrichtigt
- b. Playbooks für Fehler erstellen: Wie komme ich an die Daten etc.

Logs

Bitte schön strukturiert - Bitte gut überlegt



- Definiert ein Log-Format und stellt sicher, dass alle Services das gleiche Format nutzen.
- Definiert einen Diagnose-Kontext = Informationen die im Fehlerfall helfen, z. B.
 - Traceld
 - UserId
 - SessionId
- Nutzt Structured Logging (z. B. JSON)
 - Vereinfacht das Handling in der Analyse
- Nutzt asynchrones Logging (sofern das die Empfänger unterstützen), um Blockaden zu verhindern
 - Wenn das alles über TCP etc. verschickt wird und später gefiltert und sortiert wird, ist die Reihenfolge egal 😎

Logs

Bitte schön strukturiert - Bitte gut überlegt



- Loggt nicht zu viel, aber
 - jede Exception
 - jeden Fehler
 - jede sinnvolle Information
 - nicht mehrfach
- Nutzt **Log-Level**. Definiert dazu welche Kategorie welches Level haben soll, z.B.:
 - WARN: Fehler, die einen einzelnen Request betreffen, aber nicht die Stabilität des Services
 - ERROR: Fehler, die die Stabilität des Services betreffen

Metriken

Grundlagen



Metriken sind Messwerte, die den aktuellen Zustand des Systems abbilden.

Beispiele:

- **CPU-Auslastung**
- Größe des **JVM-Heaps**
- Anzahl der **Aufrufe** einer Schnittstelle

Die Metriken werden vom zu überwachenden System bereitgestellt.

Metriken können auch Metadaten haben.

Metriken

Typen



Metriken haben verschiedene Typen, um Daten abzubilden:

- **Counter**

Eine Zahl, die entweder inkrementiert oder zurückgesetzt werden kann.

- **Gauge**

Eine Metrik, die sich beliebig nach oben oder unten verändern kann.

- **Histogram**

Werte in “Buckets” - z.B. die Dauer von Requests

- **Summary**

Ähnlich zum Histogram, fasst Werte zusammen

Distributed Tracing

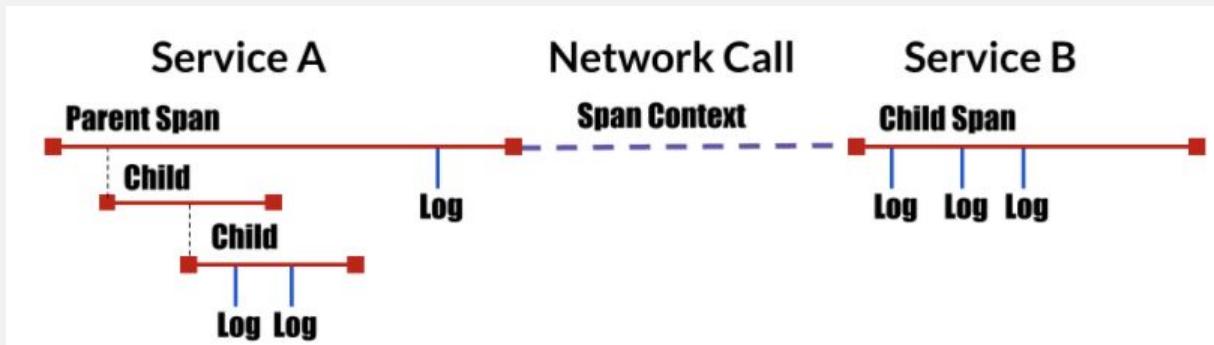
Im Wesentlichen basiert alles auf Google Dapper



- Distributed Tracing: Technik zum Nachvollziehen von Aufrufen und Abläufen in verteilten Software Systemen.
- Heutige Überlegungen gehen zurück auf das Paper: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure
- Idee: Jeder Service schickt Informationen vom Aufrufer zum nächsten Service weiter und auch wieder zurück. Dabei entsteht ein gerichteter Graph.
- Jeder Service muss instrumentiert sein. → Ansonsten entsteht eine Lücke.
- Die kleinste Einheit heißt Span. Ein Span hat eine Dauer und besitzt beschreibende Informationen. Ein Trace ist eine Menge aus 1..n Spans
- Viele Implementierungen existieren für unterschiedliche Programmiersprachen und Technologien.
 - <https://opentracing.io/docs/supported-languages/>
 - <https://opentracing.io/docs/supported-tracers/>
- Mittlerweile wird vor Allem **OpenTelemetry** genutzt

Distributed Tracing

Im Wesentlichen basiert alles auf Google Dapper



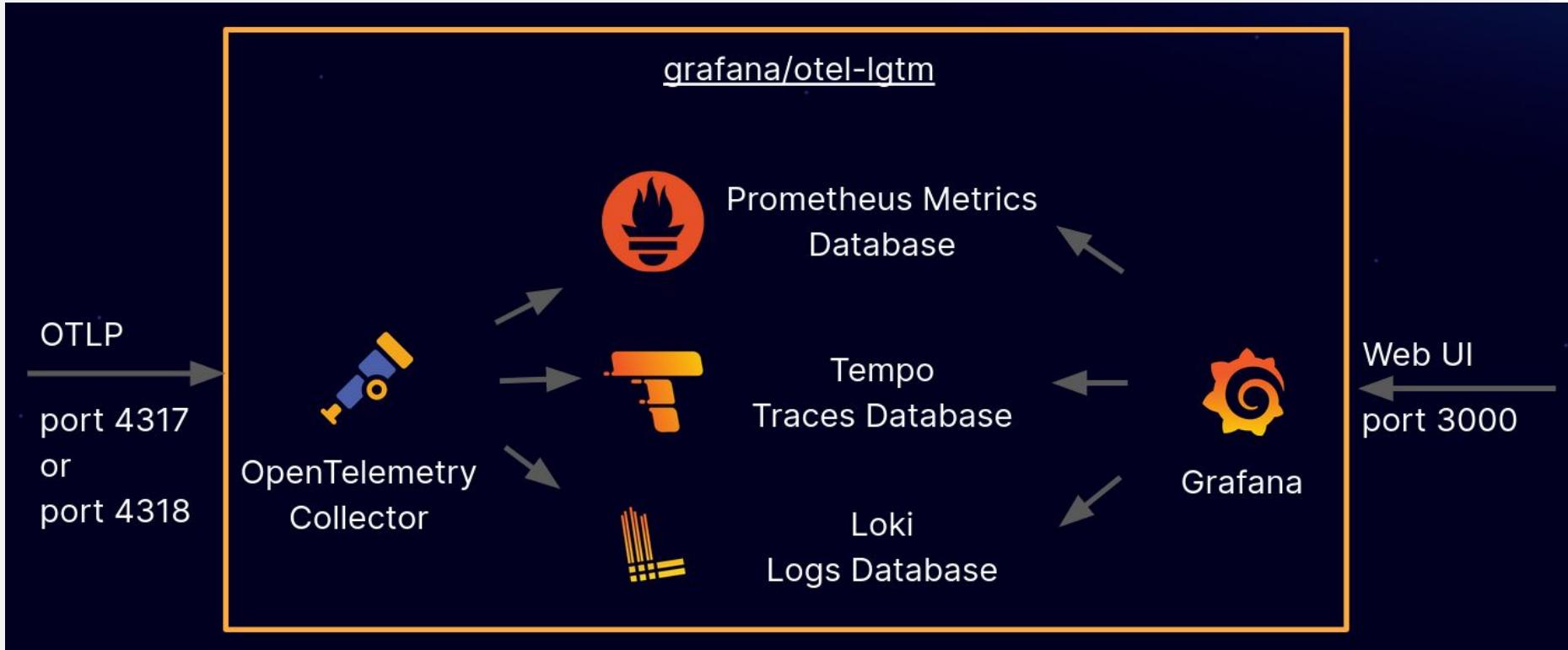
OpenTelemetry (OTeL) ist ein Open-Source-Framework für Observability, mit dem Entwicklungsteams Telemetriedaten in einem einzigen, einheitlichen Format generieren, verarbeiten und übertragen können. Dieses Format wurde von der Cloud Native Computing Foundation ([CNCF](#)) entwickelt, um standardisierte Protokolle und Tools zum Erfassen und Weiterleiten von Metriken, Logs und Traces an Überwachungsplattformen bereitzustellen.

OpenTelemetry stellt anbieterneutrale SDKs, APIs und Tools bereit, um Ihre Daten zur Analyse an beliebige Observability-Backends übermitteln zu können.

OpenTelemetry entwickelt sich schnell zum vorherrschenden Observability-Telemetriestandard in Cloud-native Anwendungen. Die Einführung von OpenTelemetry ist wichtig für alle Unternehmen, die sich auf zukünftige Datenanforderungen vorbereiten möchten, ohne sich an einzelne Anbieter oder die Einschränkungen vorhandener Technologien zu binden.

Der Grafana Stack ist komplett OTeL kompatibel!

Der (Open-Source-)Stack

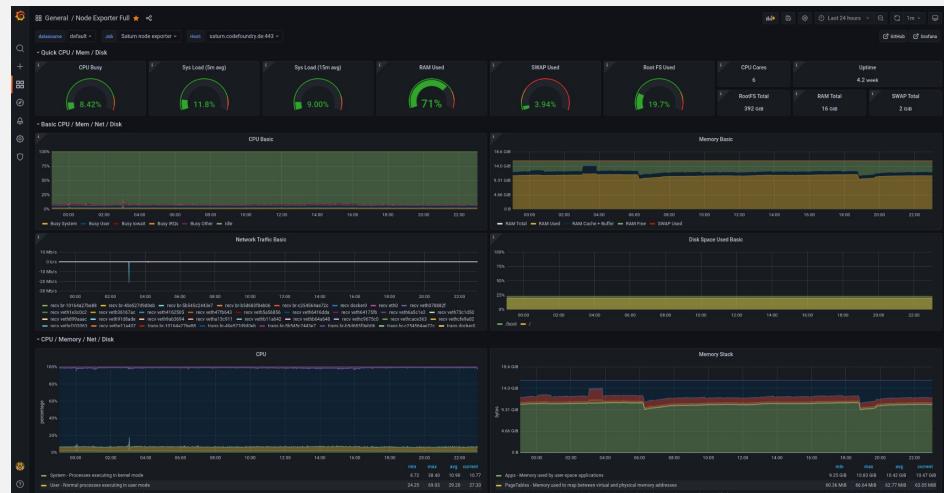


Visualisierung

Grafana Dashboards



Daten aller Art werden in Dashboards visualisiert.



The background of the image features a complex, abstract network structure composed of numerous small, semi-transparent white dots connected by thin white lines, forming a web-like pattern across the entire dark blue surface.

Viel Erfolg!