



Cloud Computing 2023 / 2024 TH Rosenheim Zusammenfassung

Franz Wimmer | franz.wimmer@qaware.de

Lukas Buchner | lukas.buchner@qaware.de



QA|WARE

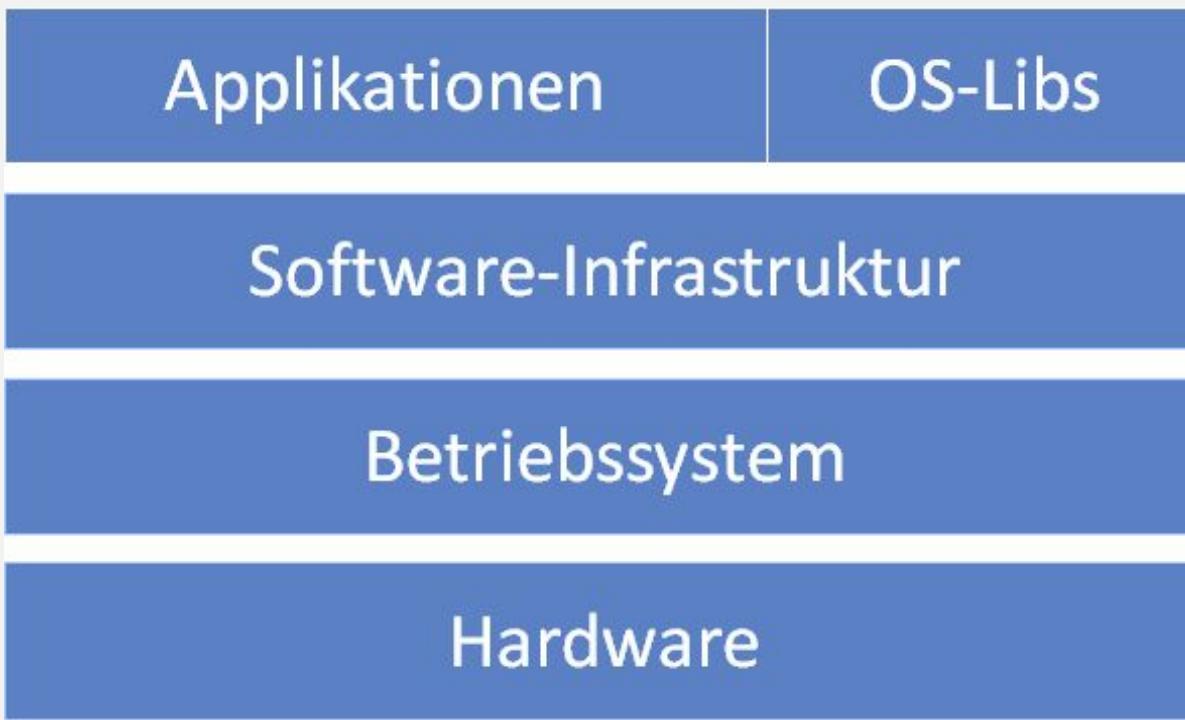
Evaluation



QA|WARE

Einführung

Beim Cloud Computing geht es im Kern um eine geringere Verbauungstiefe bei der Systementwicklung & dem Betrieb.

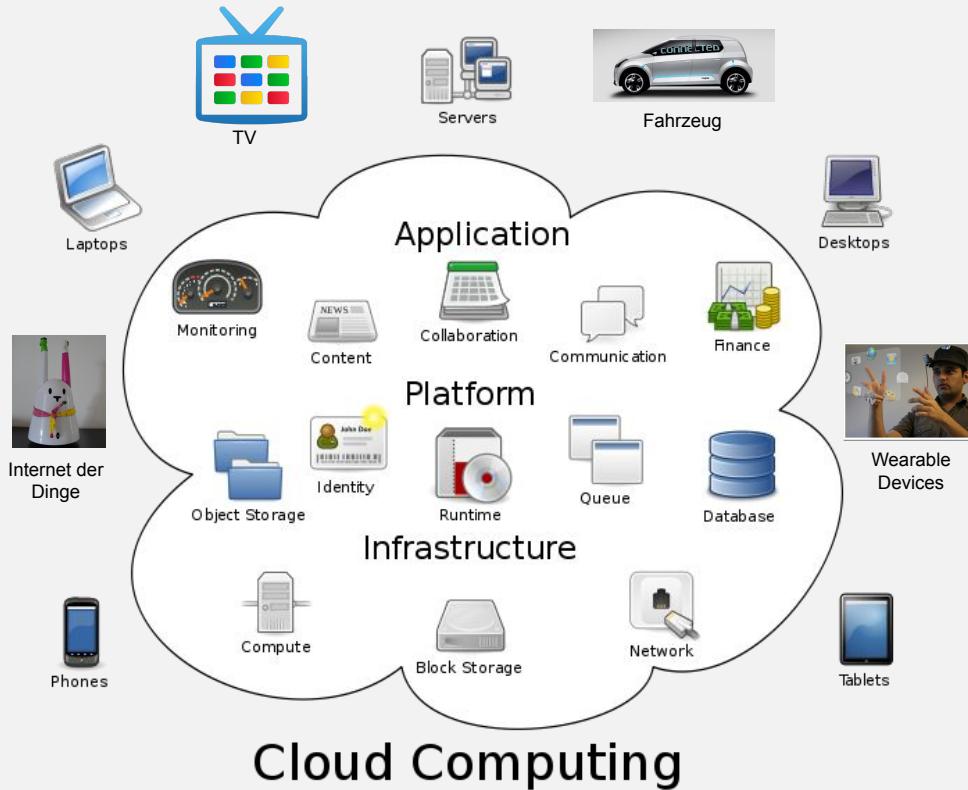


IT-Ressourcen aus der Cloud, die On-Demand konsumiert werden können.



“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

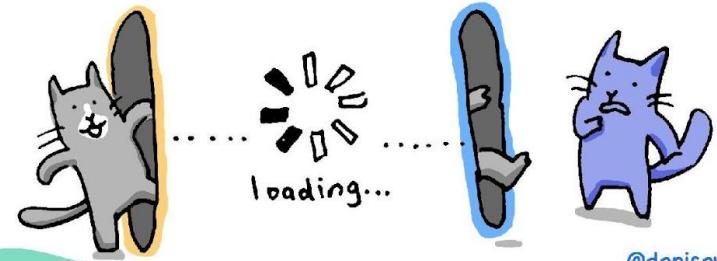
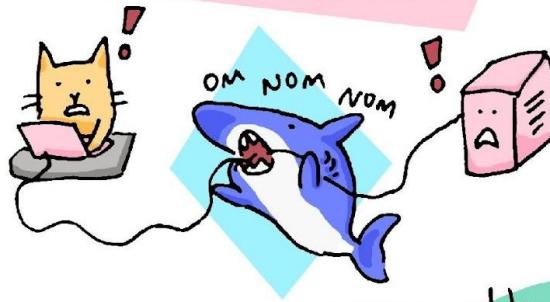


- Everything fails all the time
- Focus on MTTR, not on MTTF
- Respect the eight fallacies of distributed computing
- Scale out, not up
- Treat resources as cattles, not as pets



Quelle: https://de.wikipedia.org/wiki/Zehn_Gebote

- ① The network is reliable
- ② Latency is ZERO
- ③ Bandwidth is infinite



- ⑧ The network is homogeneous



- ⑦ Transport costs \$0



the 8 Fallacies of Distributed Computing

Originally formulated by L. Peter
Deutsch & Colleagues at Sun Microsystems
in 1994; #8 added in 1997 by James Gosling

- ④ The network is secure



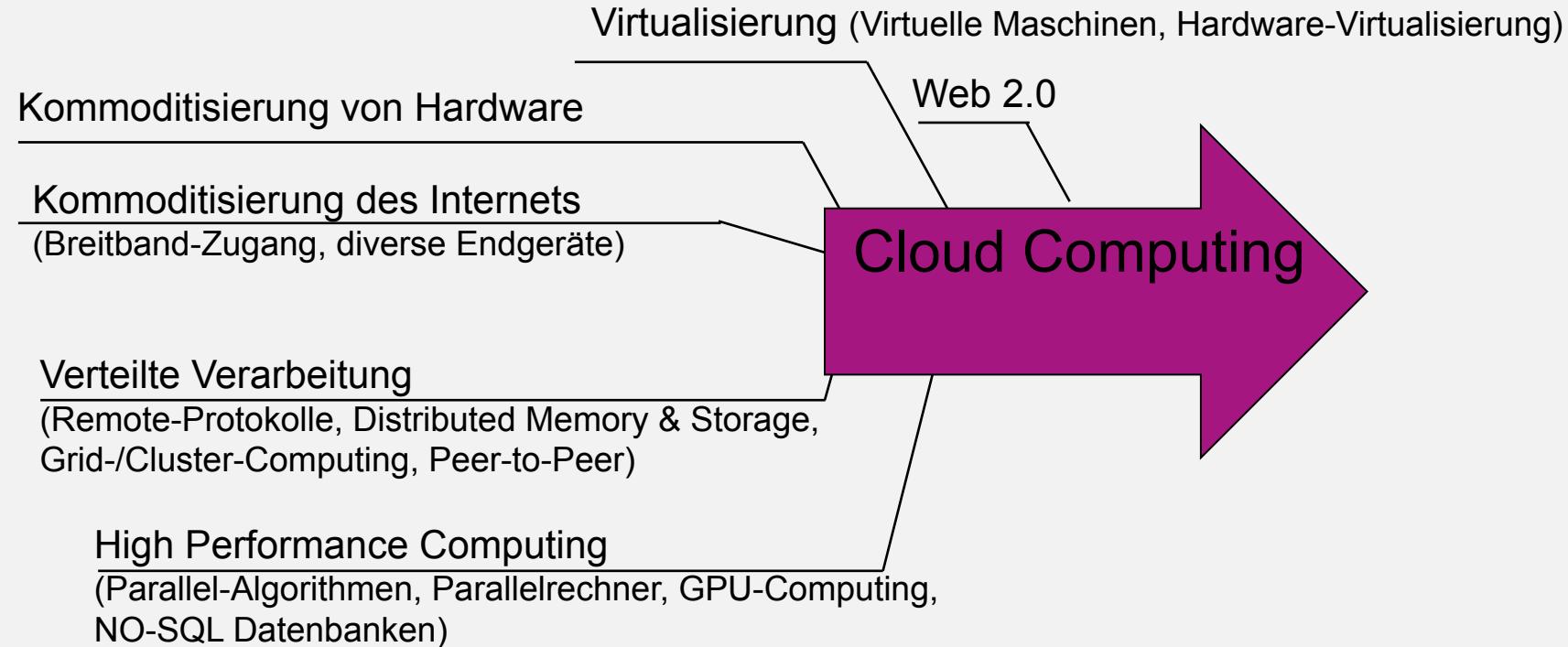
- ⑥ There is only
one administrator



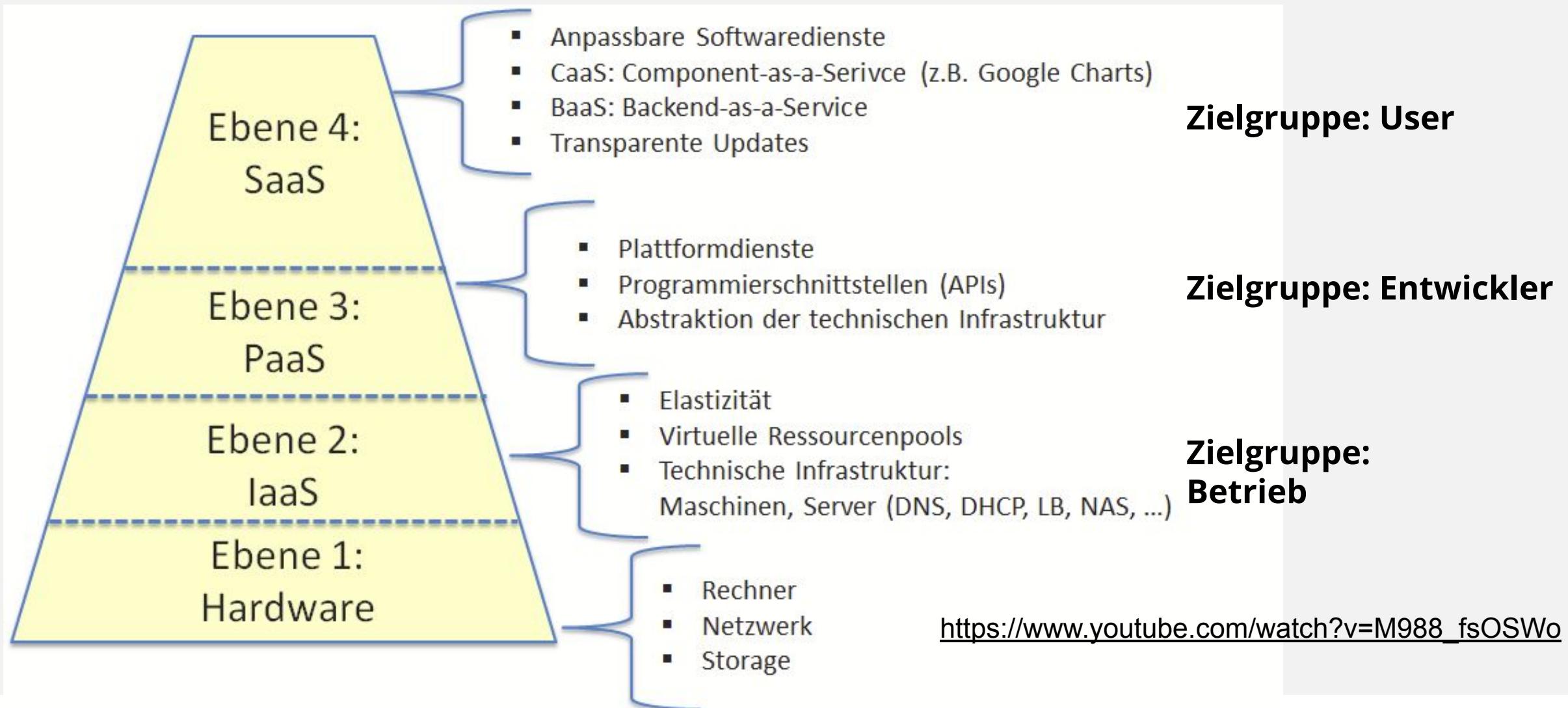
- ⑤ Topology doesn't
change



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



Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.



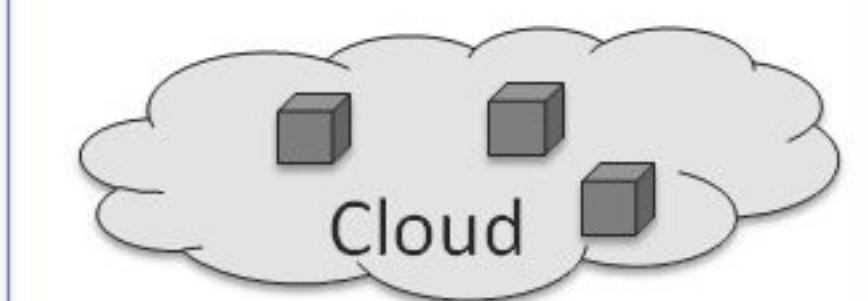
Öffentliche und private Wolken.



QA|WARE

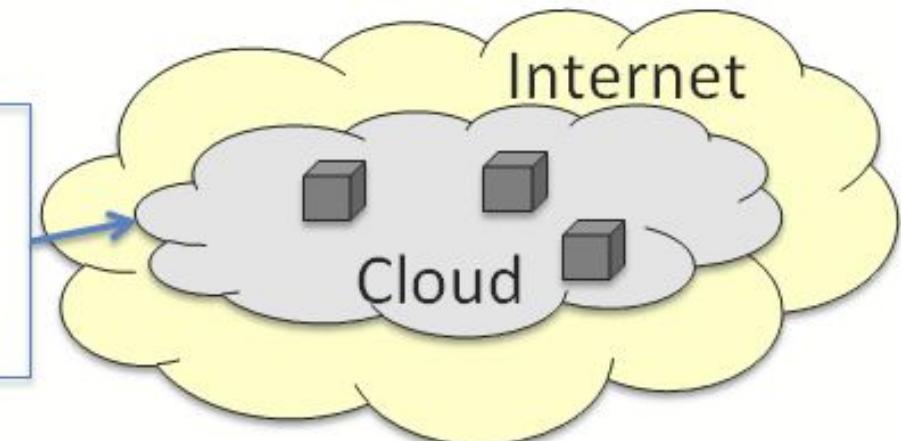
Private Cloud:

Mein Unternehmen



Public Cloud:

Mein
Unter-
nehmen

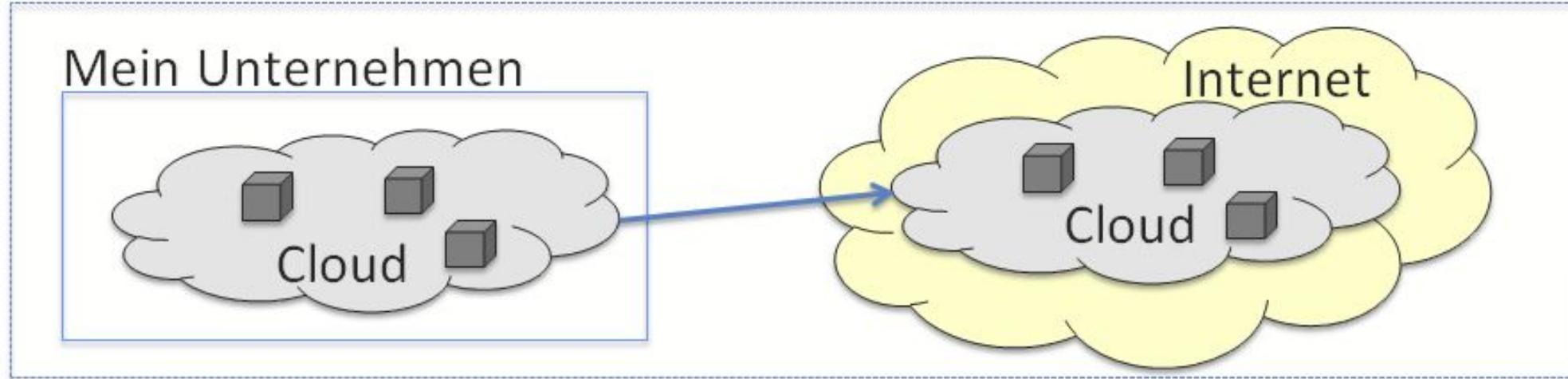


Hvbride und multiple Wolken.

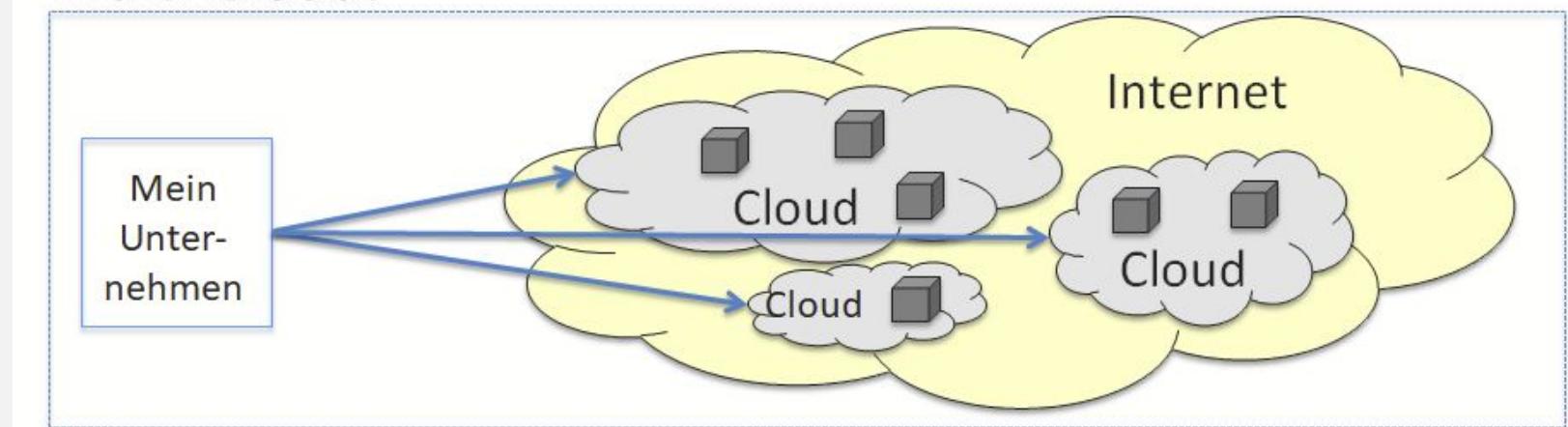


QA|WARE

Hybrid Cloud:



Multi-Cloud:





QA|WARE

Kommunikation

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

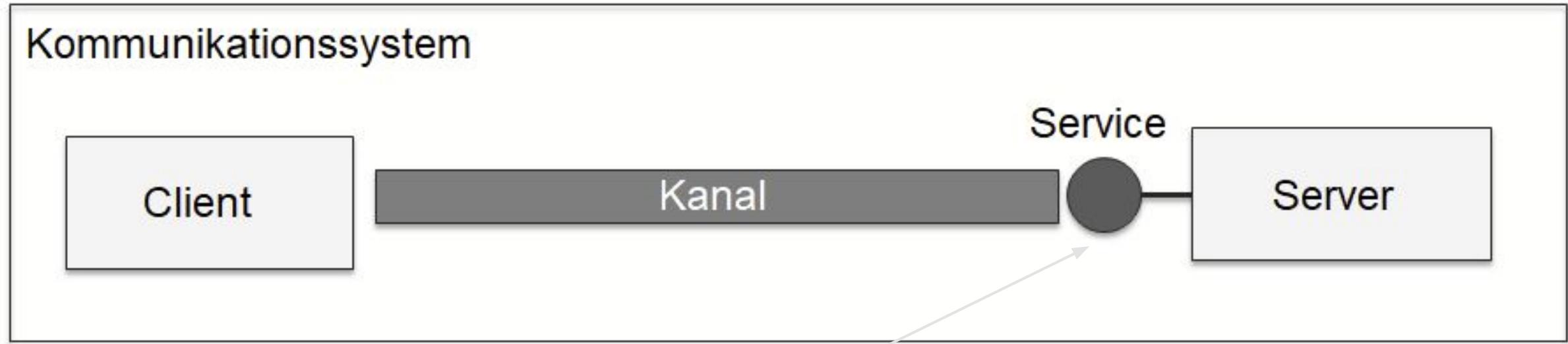
Kommunikationssystem = Infrastruktur für die Übermittlung von Informationen.



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



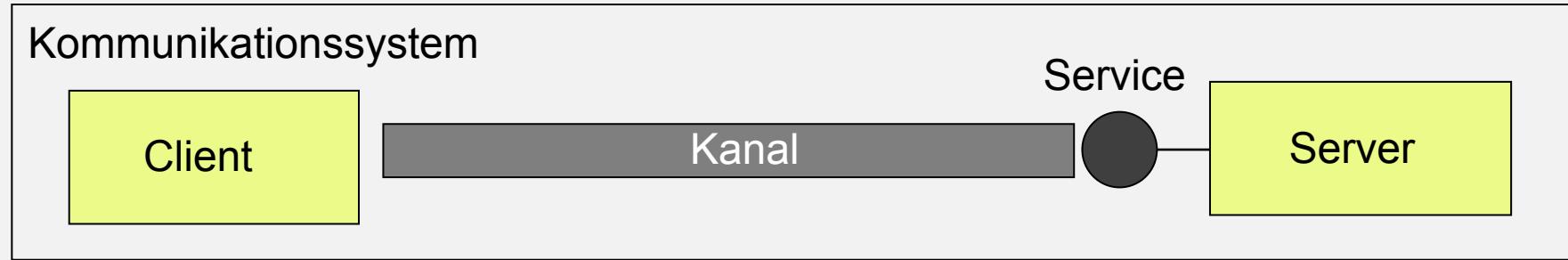
Ein **Service** ist eine Funktionalität, die über eine definierte Schnittstelle zur Verfügung gestellt wird. Jeder Service ist definiert durch eine **Serviceschnittstelle**.

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**.

Klassifikation von Kommunikationssystemen: Kardinalität der Empfänger einer Nachricht.

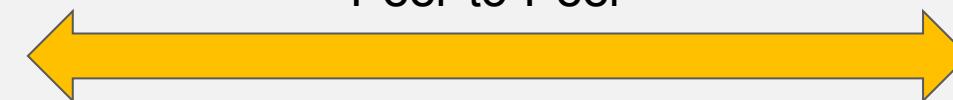


Klassifikation von Kommunikationssystemen: Wer beginnt mit der Kommunikation?



Client beginnt die Kommunikation. **Request/Response** Anfrage mit der Bitte um Antwort.

Benachrichtigung in eine Richtung. **Push** **Server beginnt die Kommunikation.**



Beide Kommunikationspartner können die Kommunikation beginnen (Ambivalenz der Rollen Client und Server).
Nachrichtenaustausch in beide Richtungen.

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 URIs 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.

REST-Webservices mit JAX-RS.

<http://www.example.com/hello/Josef?salutation=Servus>



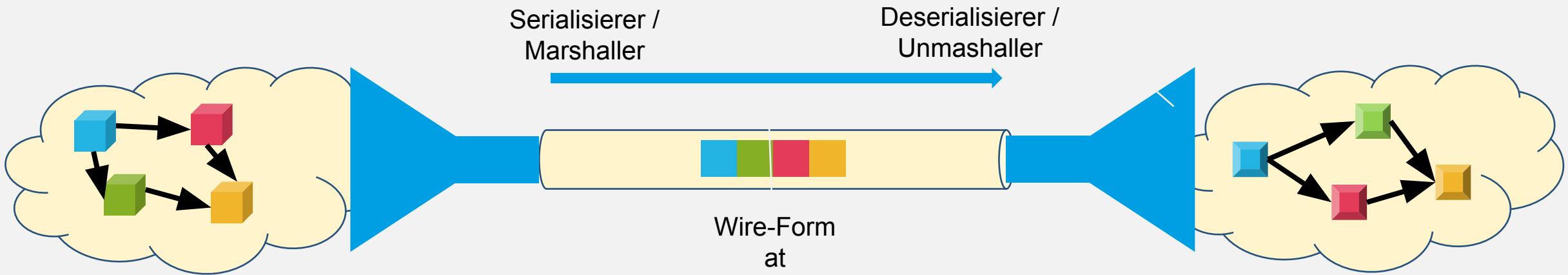
Request Path

```
@Path("/hello/{name}")
public class HelloWorldResource {
    @GET, @POST, @PUT, @DELETE
    @GET
    @Produces("application/json")
    public ResponseMessage getMessage(
        @DefaultValue("Hallo") @QueryParam("salutation") String salutation,
        @PathParam("name") String name) throws IOException {
        ResponseMessage response = new ResponseMessage(new Date().toString(), salutation + " " + name);
        return response;
    }
}
```

The diagram illustrates the mapping of a RESTful web service endpoint to its corresponding Java code. A blue arrow points from the URL <http://www.example.com/hello/Josef?salutation=Servus> down to the Java code. The code defines a `HelloWorldResource` class with a `getMessage` method. Annotations are used to map the URL path and query parameters to method parameters. Specifically, the `@Path("/hello/{name}")` annotation maps to the URL path, and the `@PathParam("name")` annotation maps to the `{name}` placeholder. The `@QueryParam("salutation")` annotation maps to the `?salutation=Servus` query parameter. The `@DefaultValue("Hallo")` annotation provides a default value for the `salutation` parameter. The `@GET`, `@Produces("application/json")`, and `ResponseMessage` annotations are also shown. Arrows with explanatory text point to specific annotations: one arrow points to the `@PathParam` annotation with the text "Analog @Consumes für 1. Parameter", another points to the `@QueryParam` annotation with "Analog @FormParam bei POST Requests".

Für XML und JSON Schnittstellen benötigen wir immer 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

- Wikipedia

Die effizienten Alternativen: Binärprotokolle

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

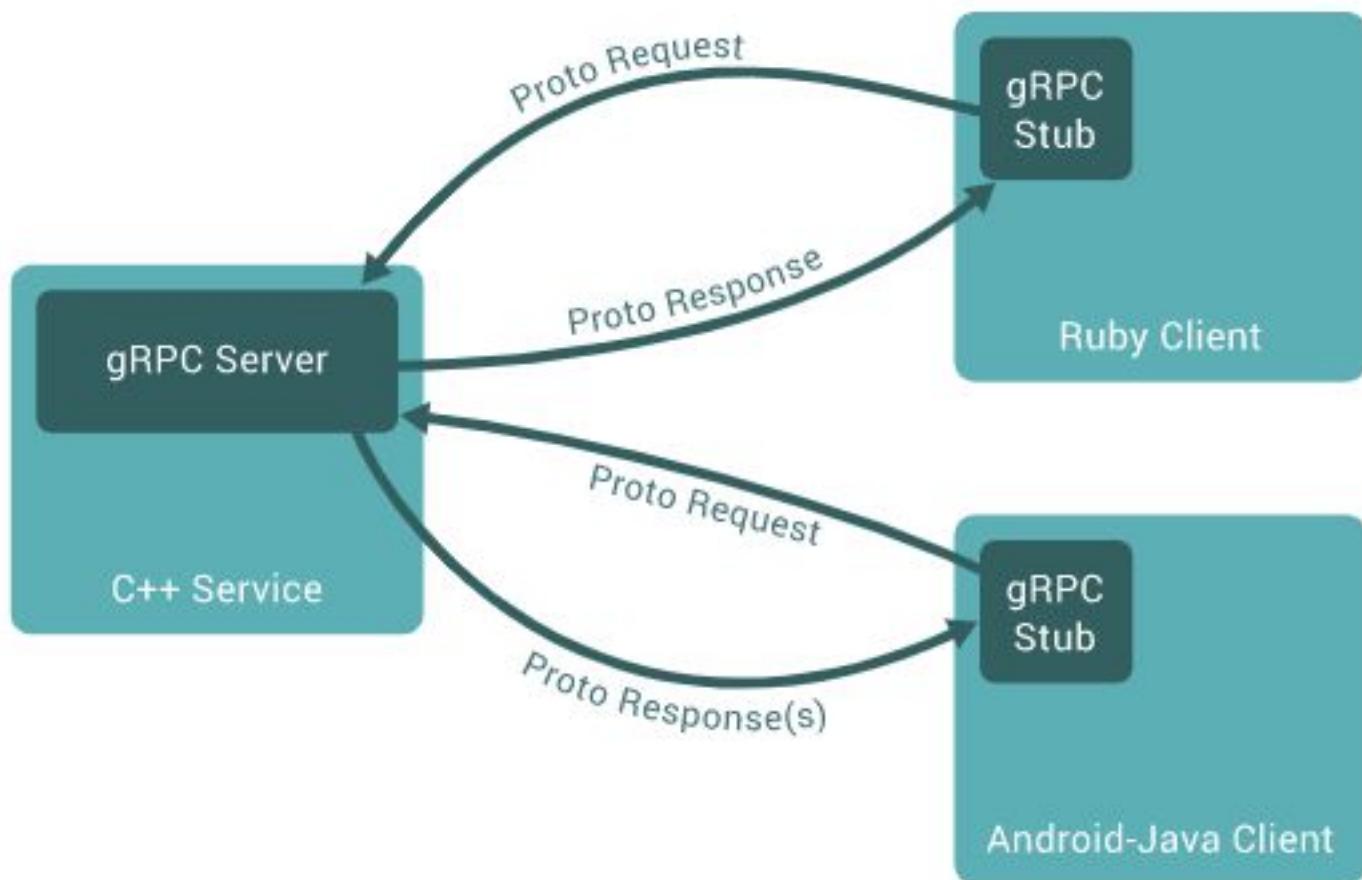
- Encoding der Payload als komprimiertes Binärformat
- Separate Schnittstellenbeschreibungen (IDLs, *Interface Definition Languages*) aus denen dann Client- und Server-Code in mehreren Programmiersprachen generiert werden können

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.

gRPC



<https://grpc.io/docs/what-is-grpc/introduction/>

Messaging ist zuverlässiger, asynchroner Nachrichtenaustausch.



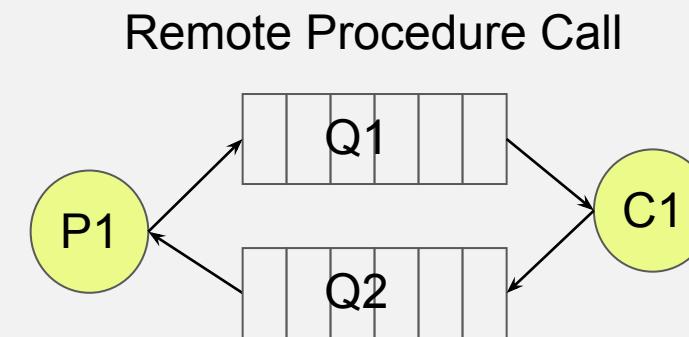
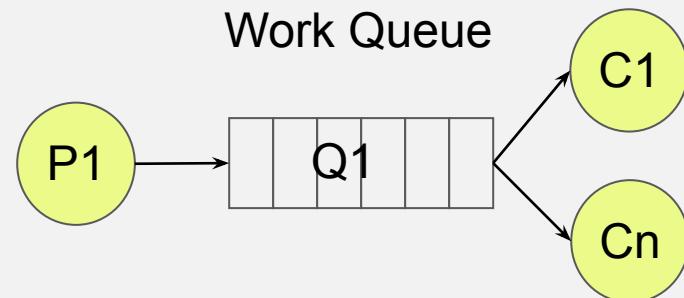
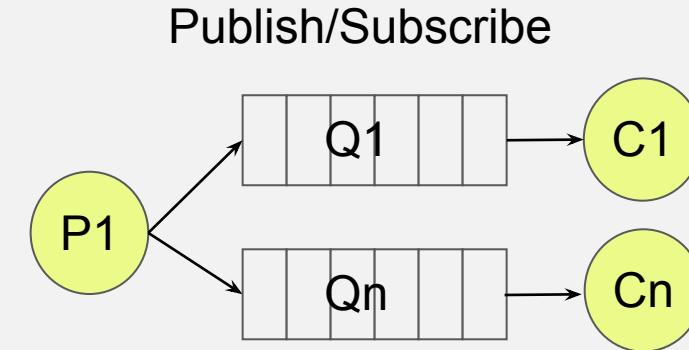
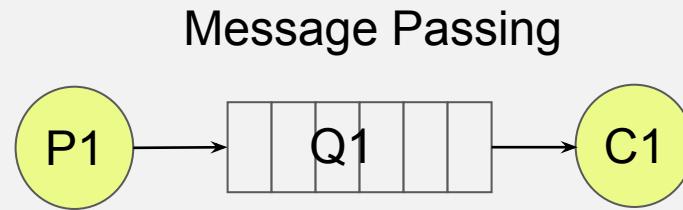
Entkopplung von Producer und Consumer.

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.





QA|WARE

Virtualisierung

Virtualisierung ist stellvertretend für mehrere grundsätzlich verschiedene Konzepte und Technologien:

Virtualisierung von Hardware-Infrastruktur

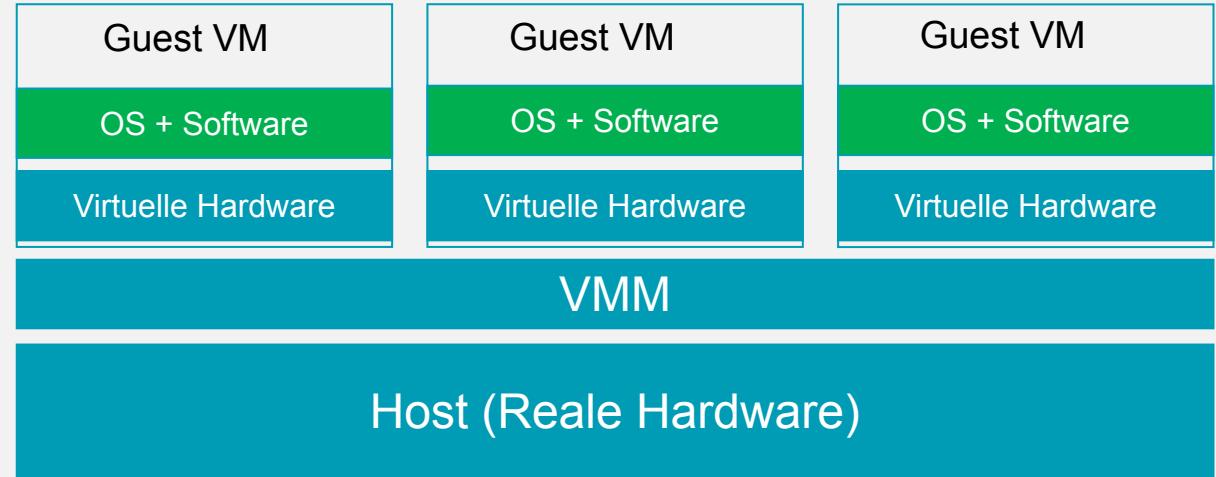
1. Emulation
2. Voll-Virtualisierung (Typ-2 Virtualisierung)
3. Para-Virtualisierung (Typ-1 Virtualisierung)

Virtualisierung von Software-Infrastruktur

4. Betriebssystem-Virtualisierung (*Containerization*)
5. Anwendungs-Virtualisierung (*Runtime*)

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 Virtualisierungs-software (Virtual Machine Monitor, VMM) abgefangen und auf die real vorhandene Hardware umgesetzt.



Host

- Der Rechner der eine oder mehrere virtuelle Maschinen ausführt und die dafür notwendigen Hardware-Ressourcen zur Verfügung stellt.

Guest

- Eine lauffähige / laufende virtuelle Maschine

VMM (Virtual Machine Monitor)

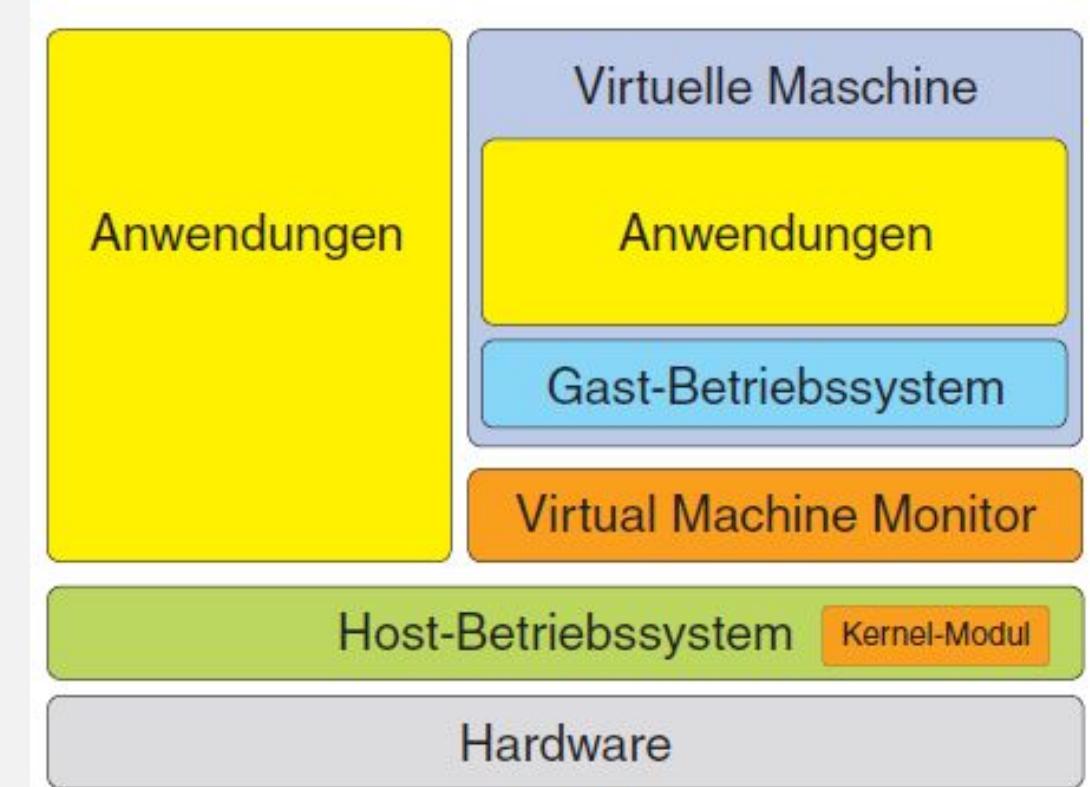
- Die Steuerungssoftware zur Verwaltung der Guests und der Host-Ressourcen

Hardware-Virtualisierung: Voll-Virtualisierung



QA|WARE

- Jedem Gastbetriebssystem steht ein eigener virtueller Rechner mit virtuellen Ressourcen wie CPU, Hauptspeicher, Laufwerken, Netzwerkkarten, usw. zur Verfügung
- Der VMM läuft hosted als Anwendung unter dem Host-Betriebssystem (Typ 2 Hypervisor)
- Der VMM verteilt die Hardwareressourcen des Rechners an die VMs
- Teilweise emuliert der VMM Hardware, die nicht für den gleichzeitigen Zugriff mehrerer Betriebssysteme ausgelegt ist (z.B. Netzwerkkarten, Grafikkarten)
- Leistungsverlust: 5-10%.



Hardware-Virtualisierung: Para-Virtualisierung

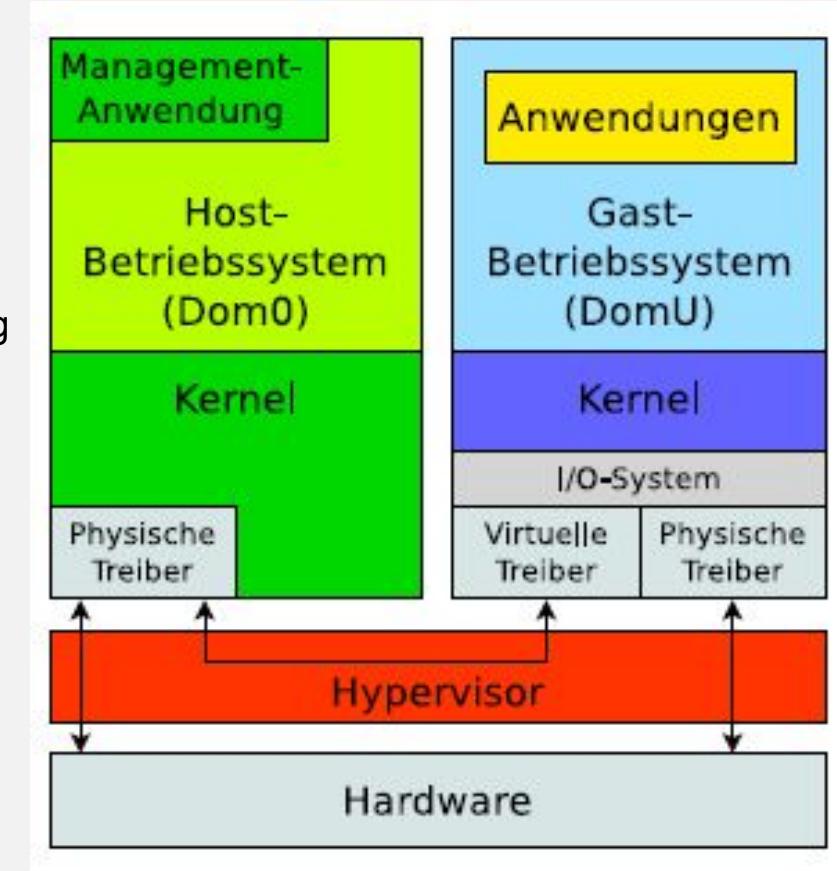
Der Hypervisor läuft direkt auf der verfügbaren Hardware. Er entspricht somit einem Betriebssystem, das ausschließlich auf Virtualisierung ausgerichtet ist.

Das Gast-Betriebssystem muss um virtuelle Treiber ergänzt werden, um mit dem Hypervisor interagieren zu können.

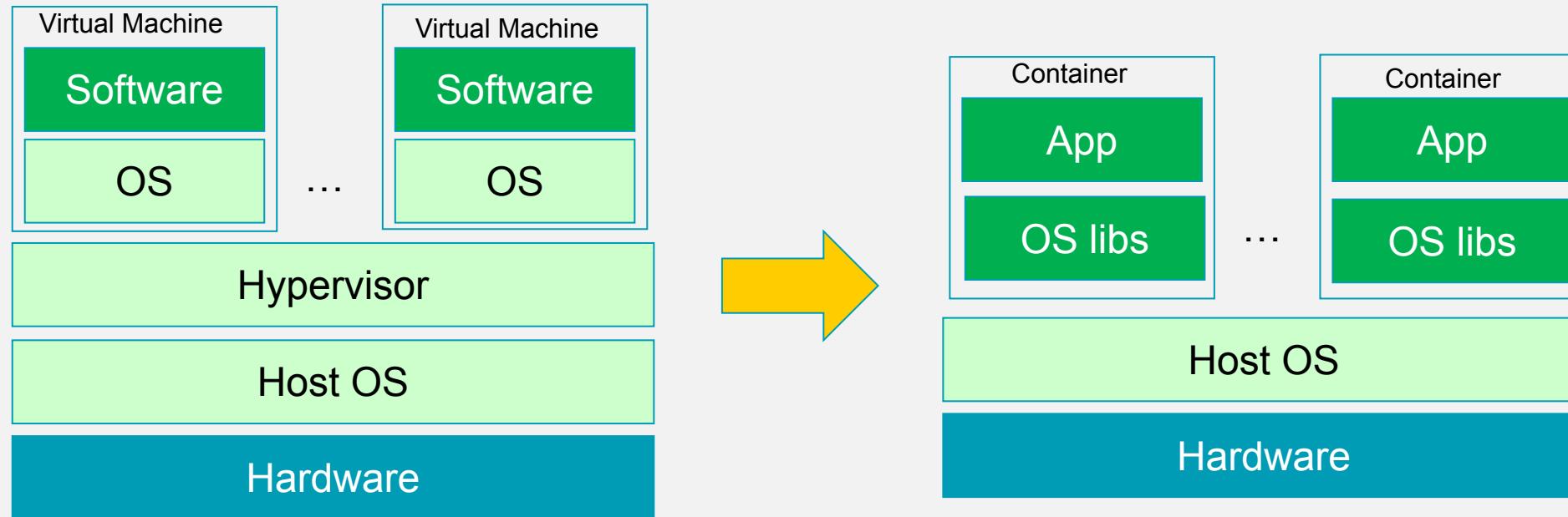
- Dem Gast-Betriebssystem stehen keine direkt low-level virtualisierten Hardware-Ressourcen (CPU, RAM, ...) zur Verfügung sondern eine API zur Nutzung durch die virtuellen Treiber.
- Unterstützte Betriebssysteme und Hardware-Varianten aus Sicht des Gastes eingeschränkt pro Hypervisor-Implementierung.

Der Hypervisor nutzt die Treiber eines Host-Betriebssystems, um auf die reale Hardware zuzugreifen. Damit brauchen im Hypervisor nicht aufwändig eigene Treiber implementiert werden.

Leistungsverlust: 2-3%



Betriebssystem-Virtualisierung



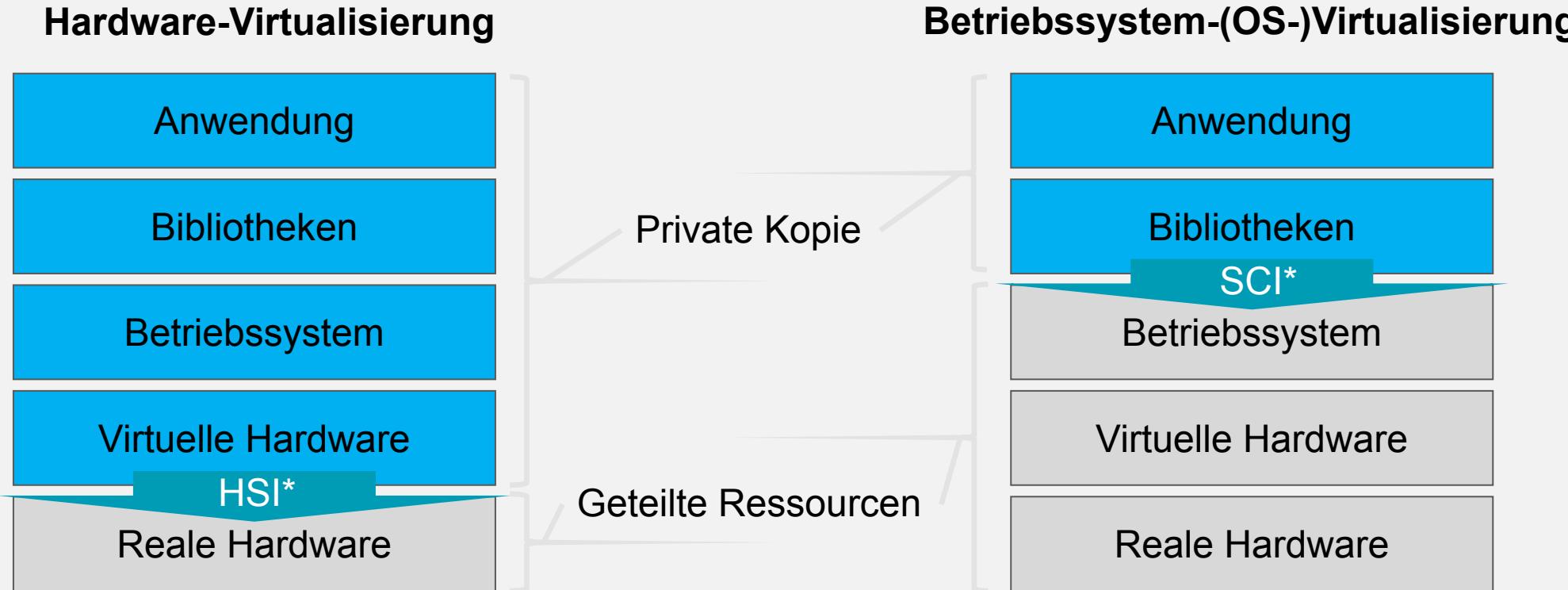
Leichtgewichtiger Virtualisierungsansatz: Es gibt keinen Hypervisor. Jede App läuft direkt als Prozess im Host-Betriebssystem. Dieser ist jedoch maximal durch entsprechende OS-Mechanismen isoliert (z.B. Linux LXC).

- Isolation des Prozesses durch Kernel Namespaces (bzgl. CPU, RAM und Disk I/O) und Containments
- Isoliertes Dateisystem
- Eigene Netzwerk-Schnittstelle

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

Startup-Zeit = Startdauer für den ersten Prozess

Hardware- vs. Betriebssystem-Virtualisierung



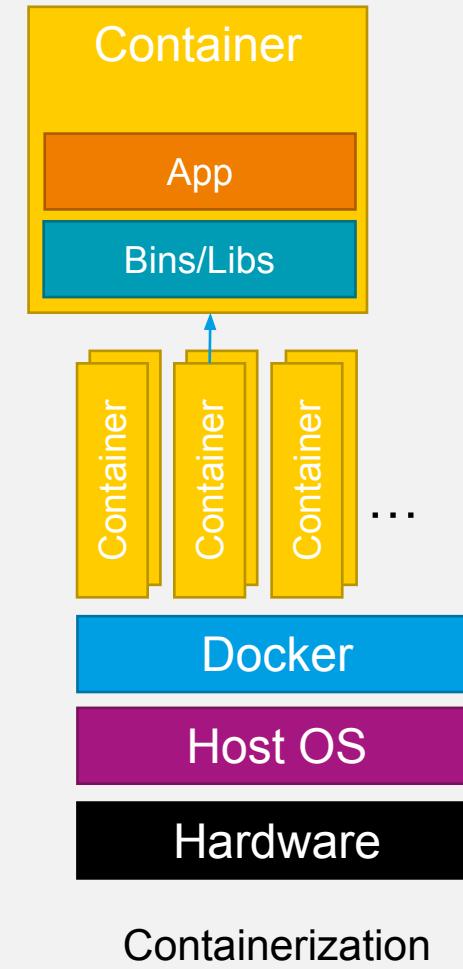
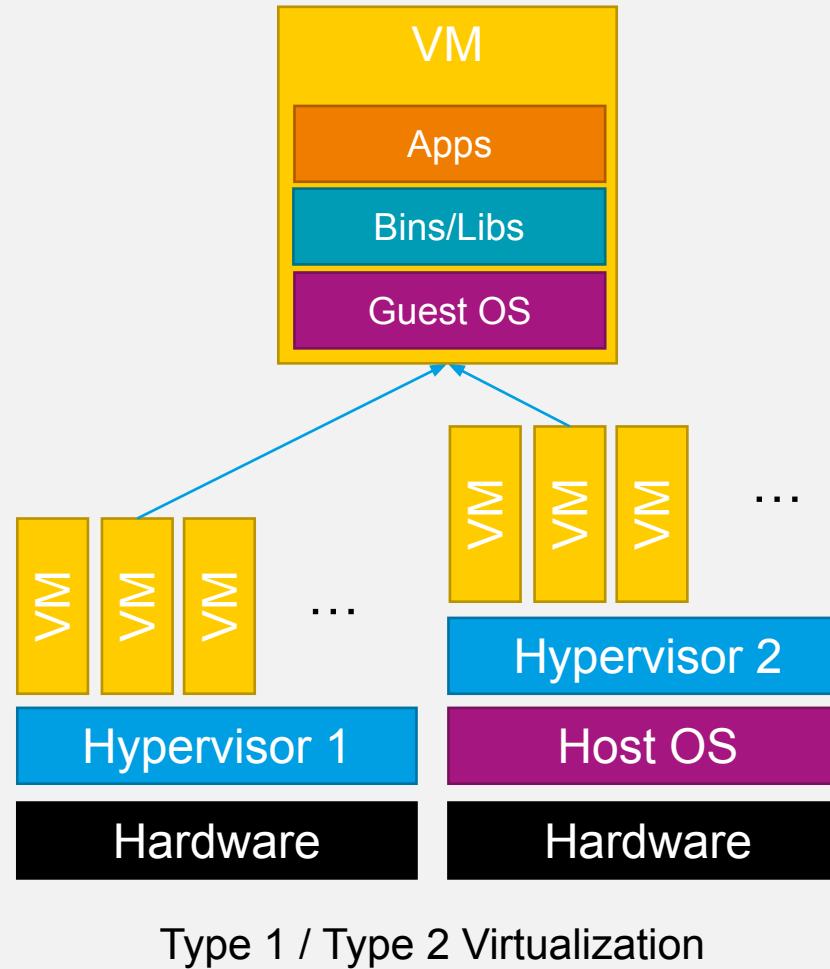
- Stärkere Isolation
- Höhere Sicherheit

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

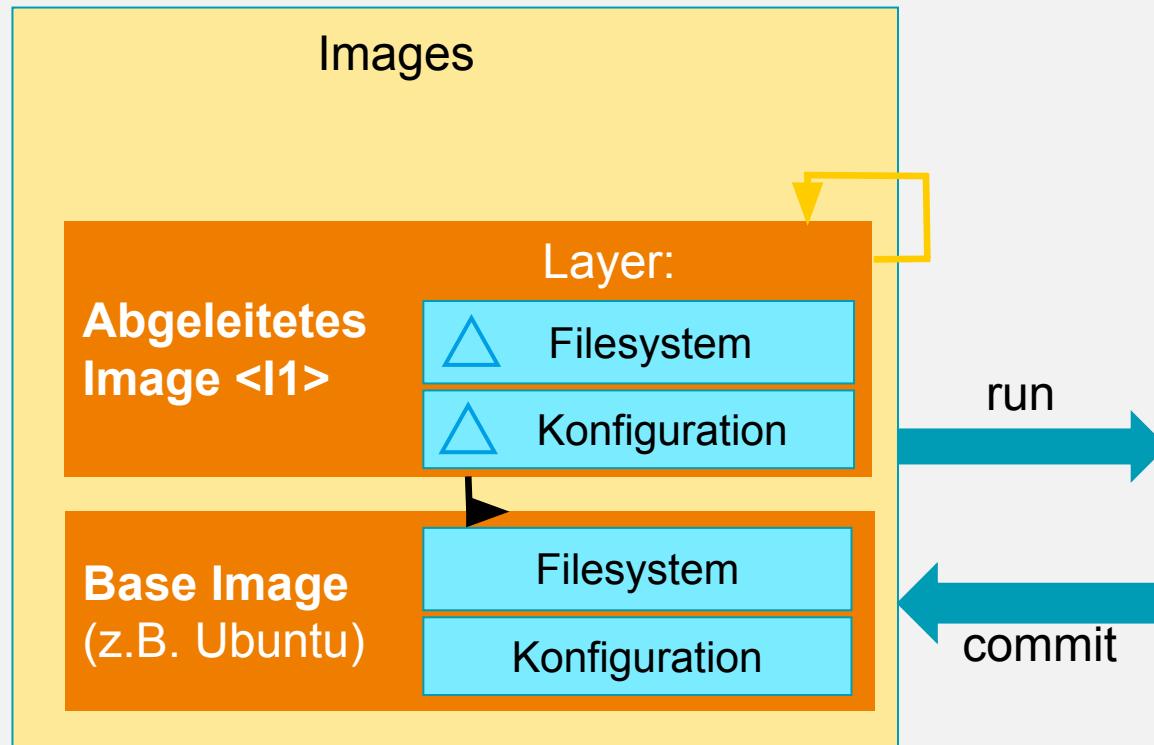
*) HSI = Hardware Software Interface

SCI = System Call Interface

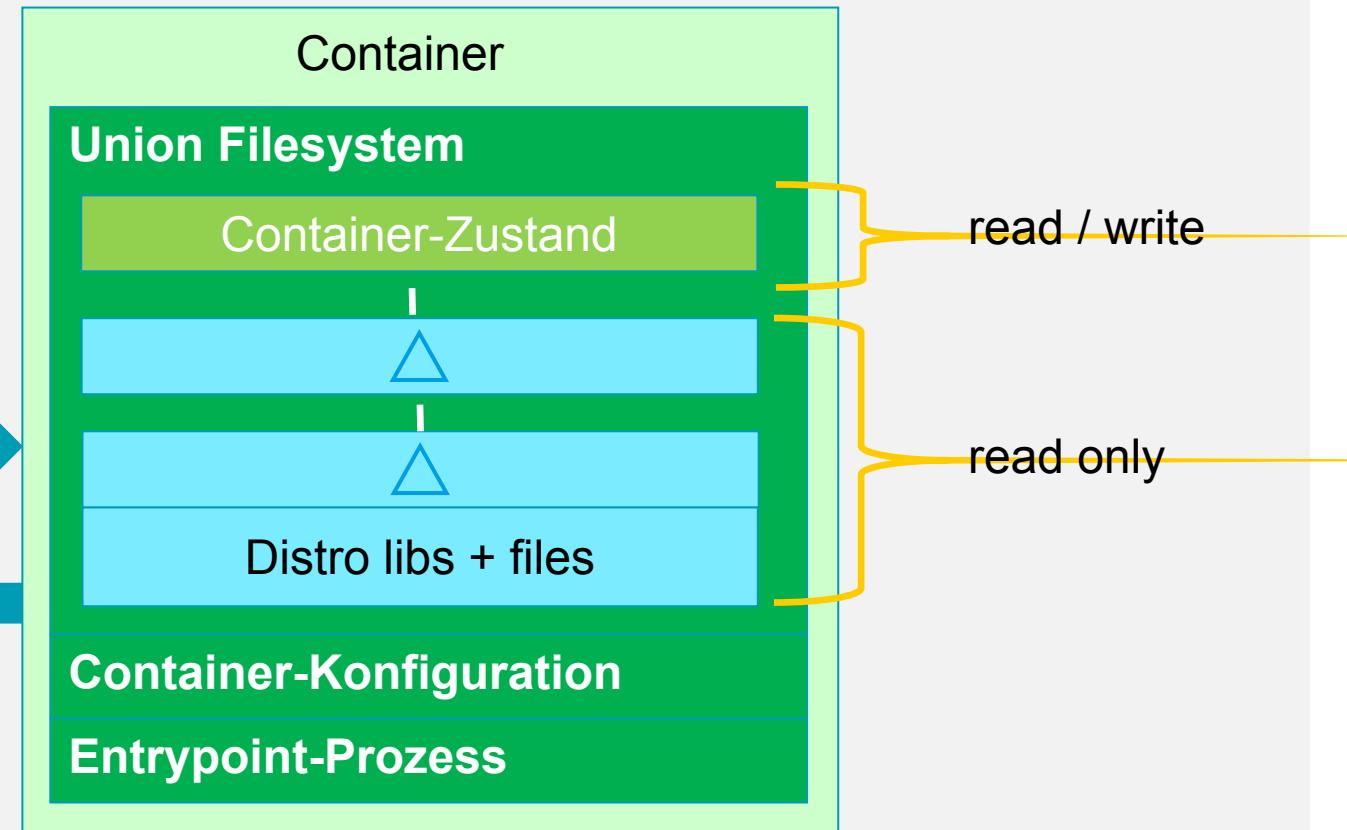
Containerization mit Docker



Im Zentrum von Docker stehen Images und Container.



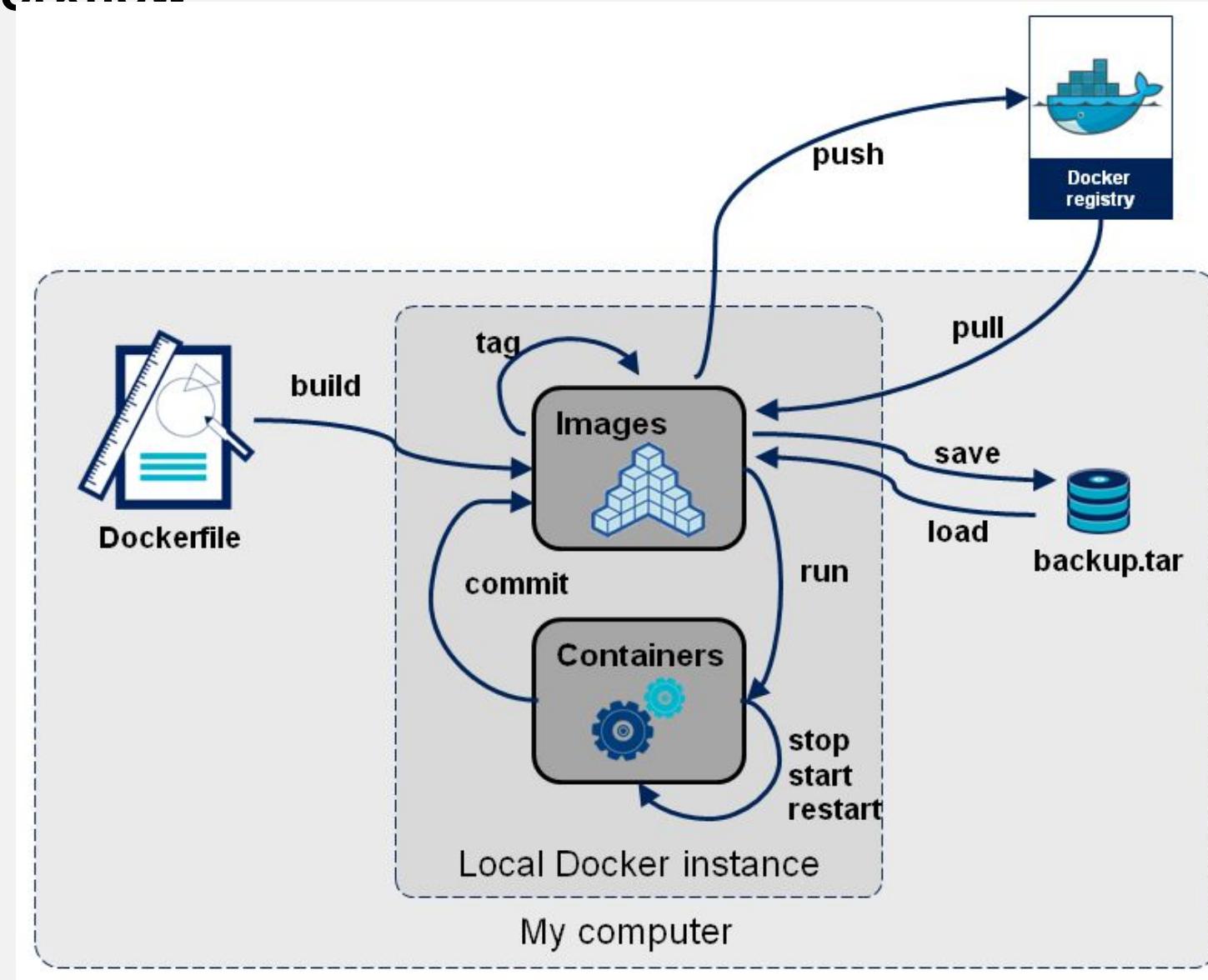
Ruhender und transportierbarer Zustand



Laufender Zustand

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

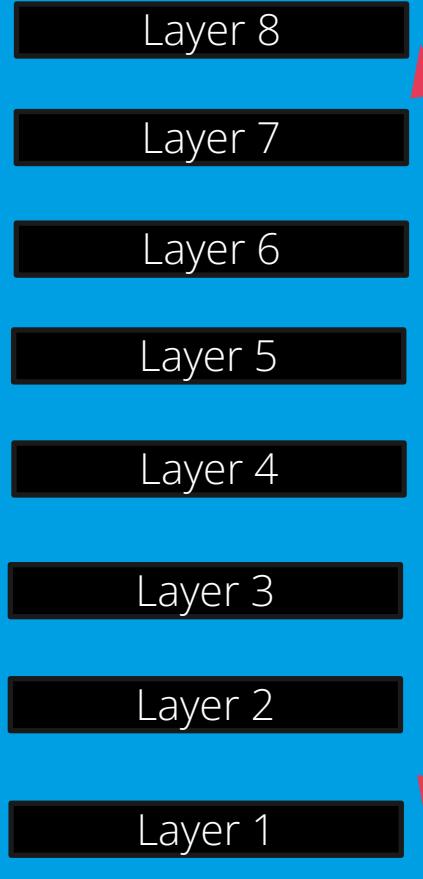
Der Docker Workflow



Das Dockerfile definiert Aufbau und Inhalt des Image.



My 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"]
```

Niemals „latest“ verwenden. Antipattern

Dockerfile-Kommandos



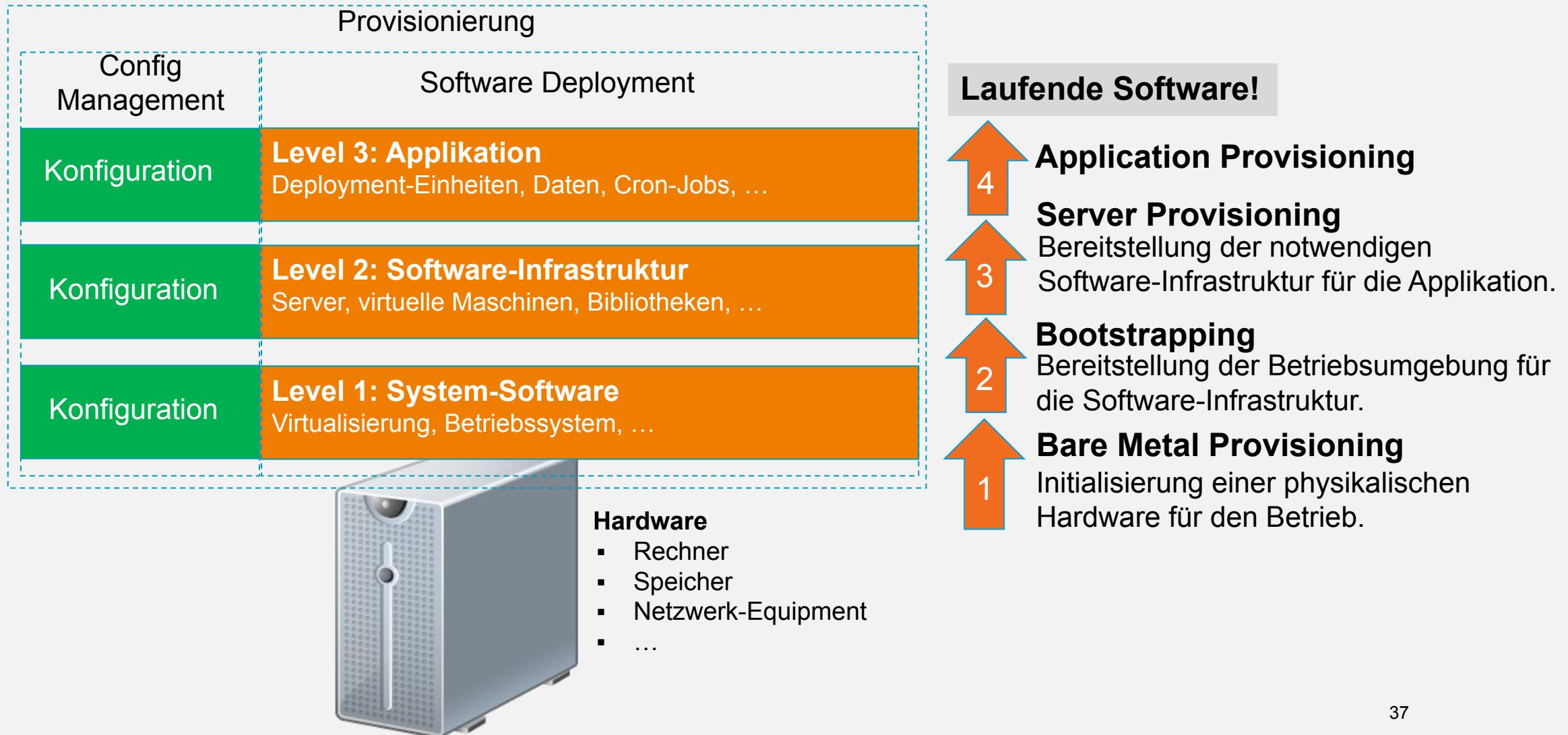
Element	Meaning
FROM <image-name>	Sets to base image (where the new image is derived from)
MAINTAINER <author>	Document author
RUN <command>	Execute a shell command and commit the result as a new image layer (!)
ADD <src> <dest>	Copy a file into the containers. <src> can also be an URL. If <src> refers to a TAR-file, then this file automatically gets un-tared.
VOLUME <container-dir> <host-dir>	Mounts a host directory into the container.
ENV <key> <value>	Sets an environment variable. This environment variable can be overwritten at container start with the -e command line parameter of docker run .
ENTRYPOINT <command>	The process to be started at container startup
CMD <command>	Parameters to the entrypoint process if no parameters are passed with docker run
WORKDIR <dir>	Sets the working dir for all following commands
EXPOSE <port>	Informs Docker that a container listens on a specific port and this port should be exposed to other containers
USER <name>	Sets the user for all container commands



QA|WARE

Provisionierung

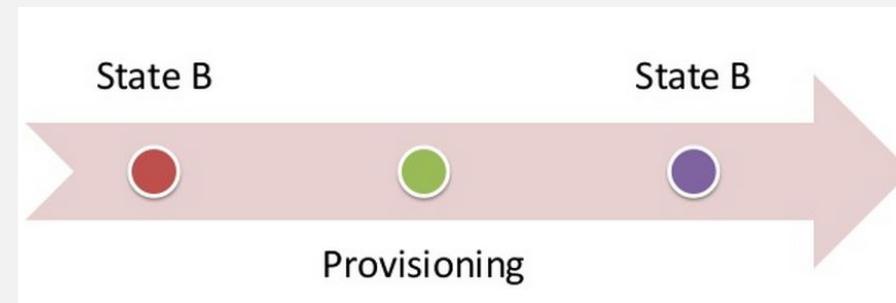
Provisierung erfolgt auf drei verschiedenen Ebenen und in vier Stufen.



Konzeptionelle Überlegungen zur Provisionierung.

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

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



Was ein Provisionierungsmechanismus leisten muss:

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

Zusicherungen

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

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

Die neue Leichtigkeit des Seins.

Old Style



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

New Style

„Immutable Infrastructure / Phoenix Systems“



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

Provisionierung mit DockerFile und Docker Compose

Deployment-Ebenen

Level 3: Applikation

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

Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

Level 1: System-Software

Virtualisierung, Betriebssystem, ...

Docker-Image-Kette

Applikations-Image

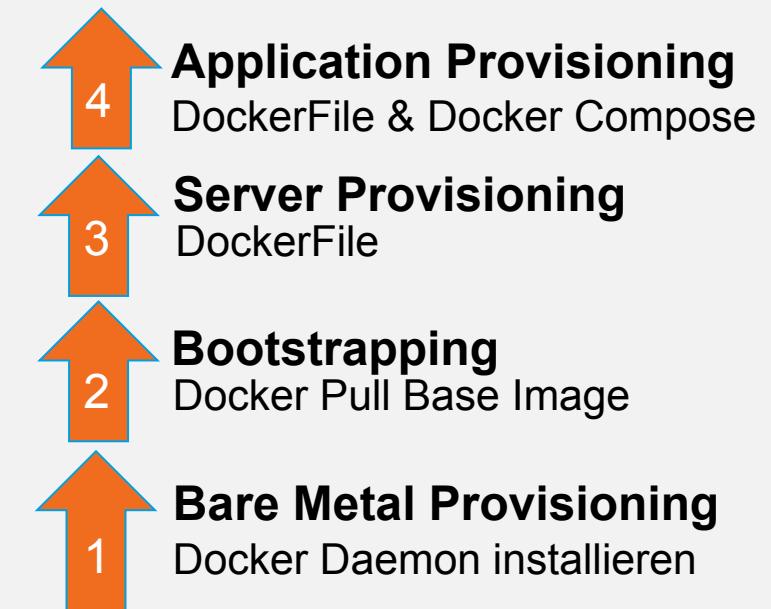
(z.B. www.qaware.de)

Server Image

(z.B. NGINX)

Base Image

(z.B. Ubuntu)



Provisionierung mit Ansible

Deployment-Ebenen

Level 3: Applikation

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

Level 2: Software-Infrastruktur

Server, virtuelle Maschinen, Bibliotheken, ...

Level 1: System-Software

Virtualisierung, Betriebssystem, ...

Docker-Image- oder VM-Kette

Applikations-Image

(z.B. www.qaware.de)

Server Image

(z.B. NGINX)

Base Image

(z.B. Ubuntu)



TODO: Ansible – Konzepte & Begriffe

Inventory

```
[webserver]  
my-web-server.example.com
```

```
my-other-web-server.example.com
```

Modules

```
[appserver-master]
```

```
app1-master ansible_ssh_host=myapp.example.net httpsports=9090
```

```
app2-master ansible_ssh_host=myapp2.example.net httpsports=9091
```

Tasks

Roles

```
[appserver-slaves]
```

```
app1-slave ansible_ssh_host=myapp3.example.net httpsports=9090
```

```
app2-slave ansible_ssh_host=myapp4.example.net httpsports=9091
```

Playbook

```
[dbserver]
```

```
postgresql ansible_ssh_host=10.0.0.5 maxconnections=1000
```

```
Postgresql-standby ansible_ssh_host=10.0.0.10 maxconnections=1000
```

Packer Terminologie (<https://www.packer.io/docs/terminology>)

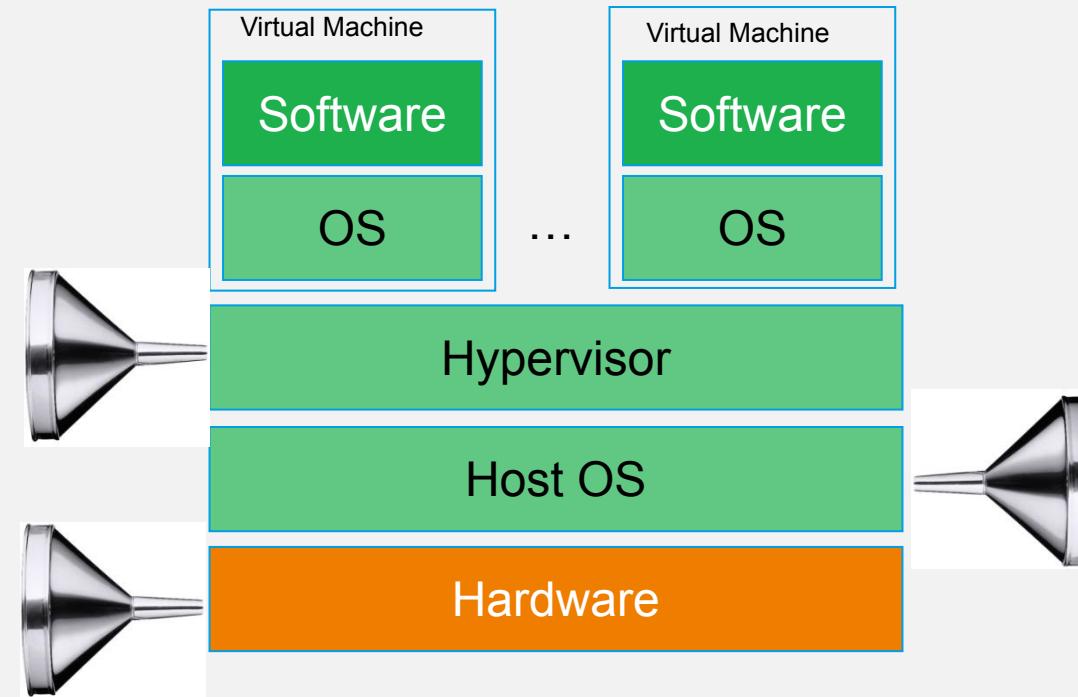
- Artifacts
 - Ergebnis eines Packer Builds, z.B. ein Dateiordner oder ein Set von AMI IDs
- Builds
 - Tasks, die ein Image für eine bestimmte Plattform erzeugen
- Builders
 - erzeugen einen bestimmten Image Typ
 - z.B. VirtualBox, Amazon EC2, Docker
- Commands
 - Unter-Commands, die man mit Packer ausführen kann, z.B. `packer build`
- Post-Processors
 - erzeugen aus Artifacts neue Artifacts (z.B. Komprimierung, Tagging, Publishing)
- Provisioners
 - installieren und konfigurieren Software in einer laufenden Instanz, bevor daraus ein statisches Artifact erzeugt wird
- Templates
 - JSON Files, die den Packer Build konfigurieren



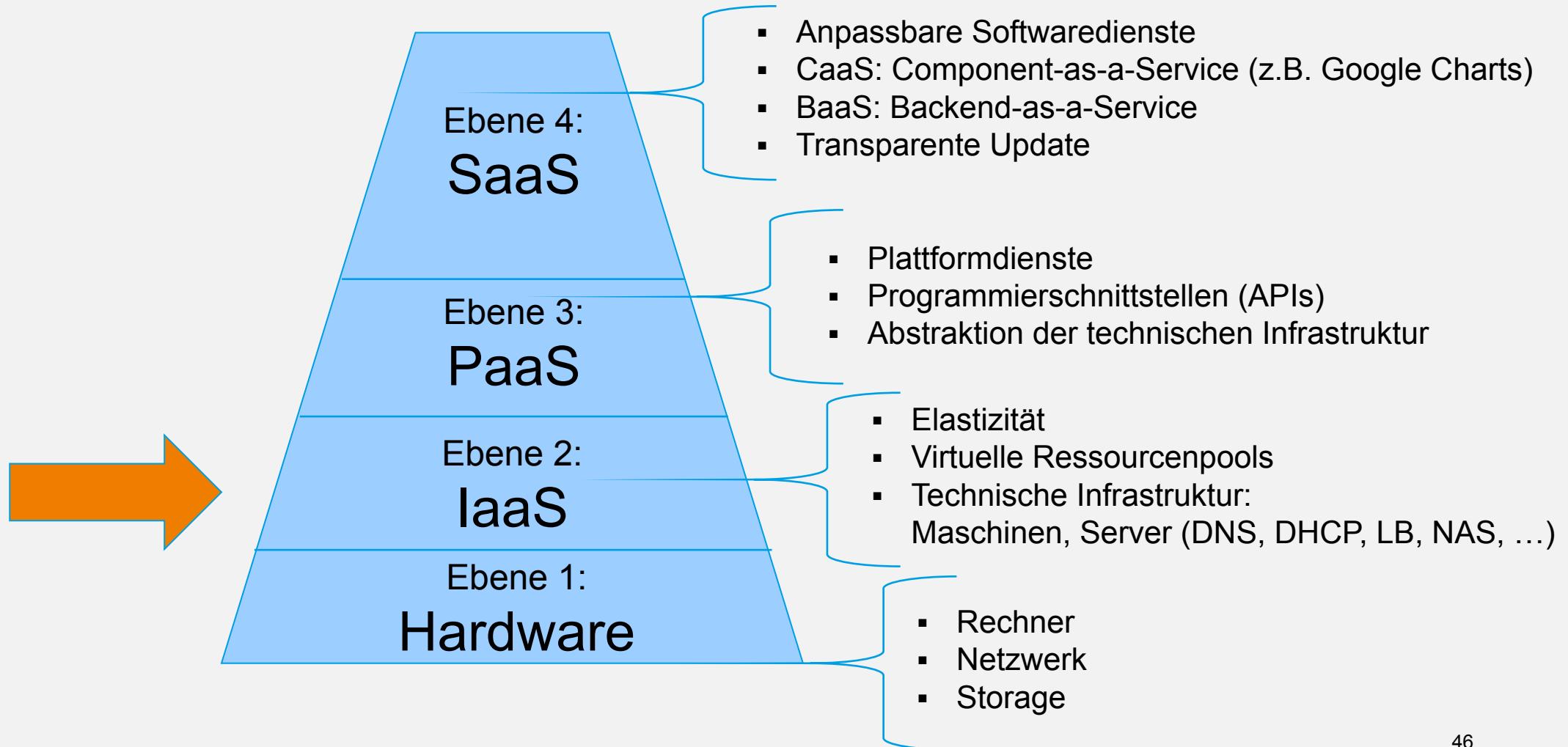
QA|WARE

IaaS

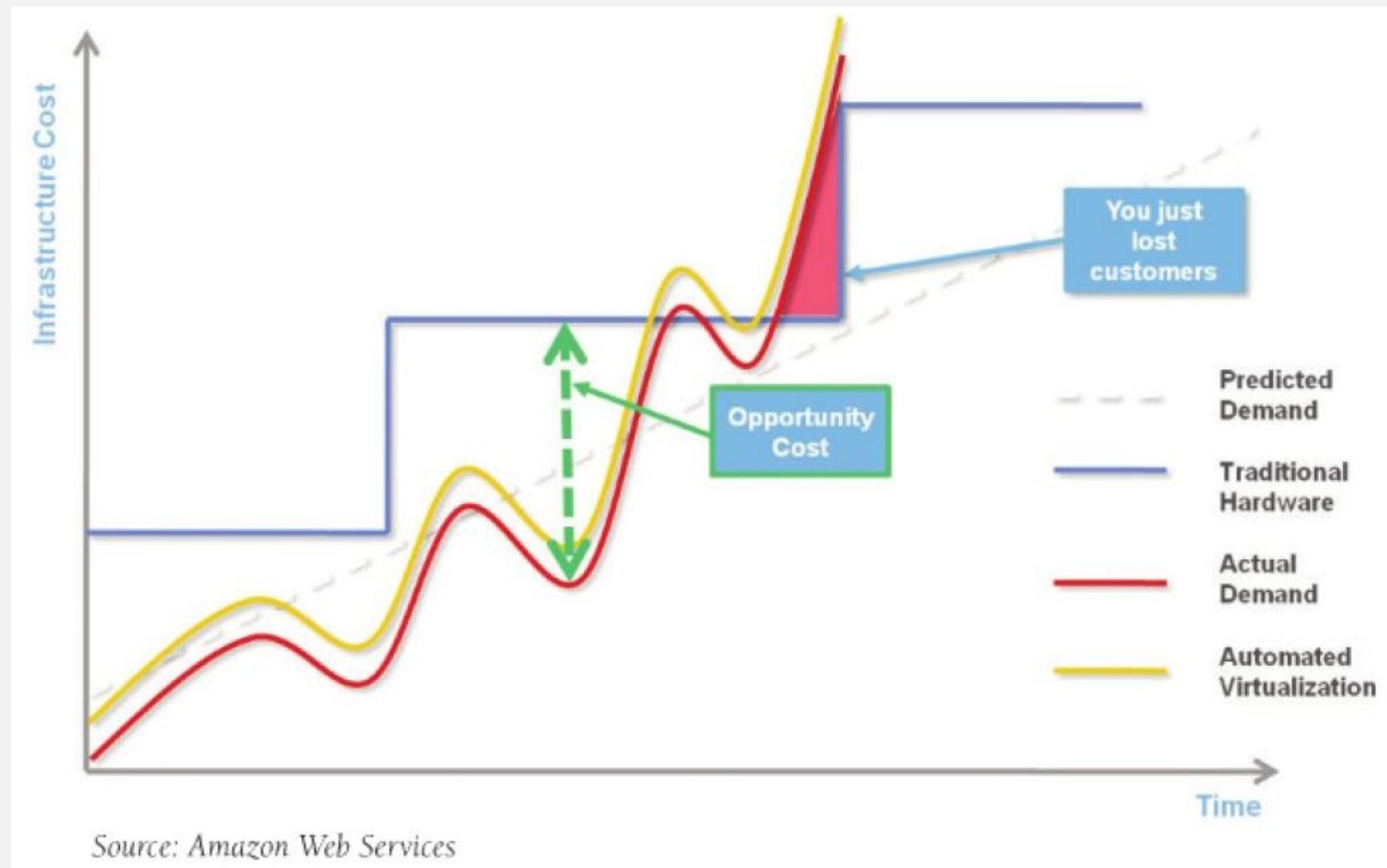
Wie kommt Software an das Blech?



Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.



Klassische Betriebsszenarien werden bei dynamischer Nachfrage teuer. Hohe Opportunitätskosten.



Definition IaaS

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

Eigenschaften einer IaaS-Cloud:

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

Ressourcen-Typen in einer IaaS-Cloud:

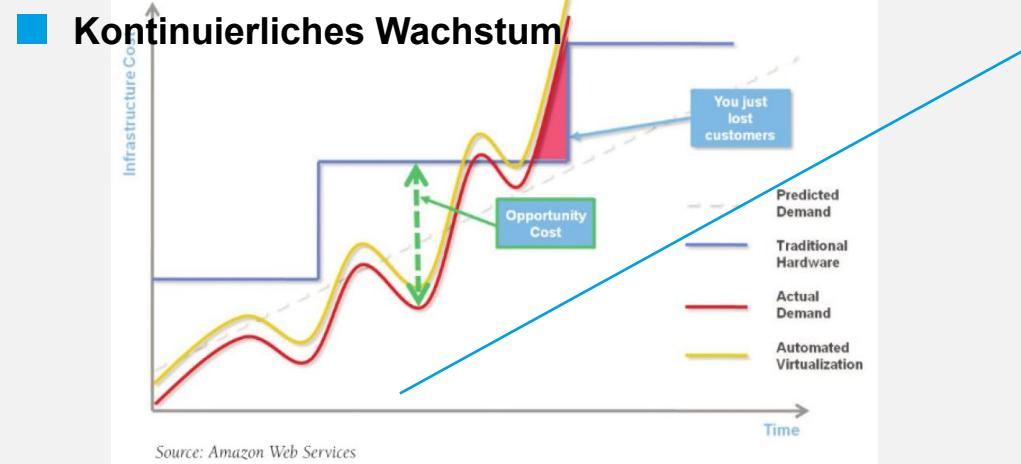
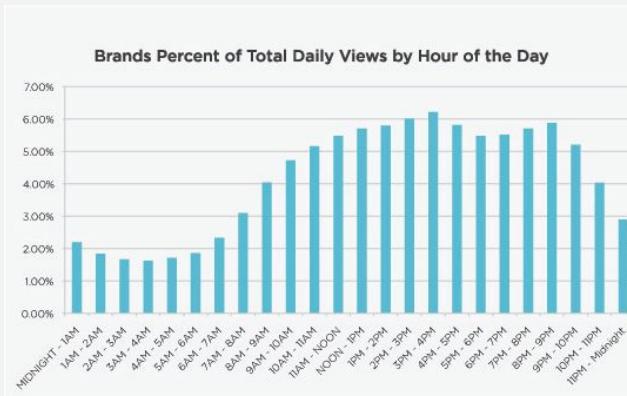
- **Rechenleistung:** Rechner-Knoten mit CPU, RAM und HD-Speicher.
- **Speicher:** Storage-Kapazitäten als Dateisystem-Mounts oder Datenbanken.
- **Netzwerk:** Netzwerk und Netzwerk-Dienste wie DNS, DHCP, VPN, CDN und Load Balancer.

Infrastruktur-Dienste einer IaaS-Cloud:

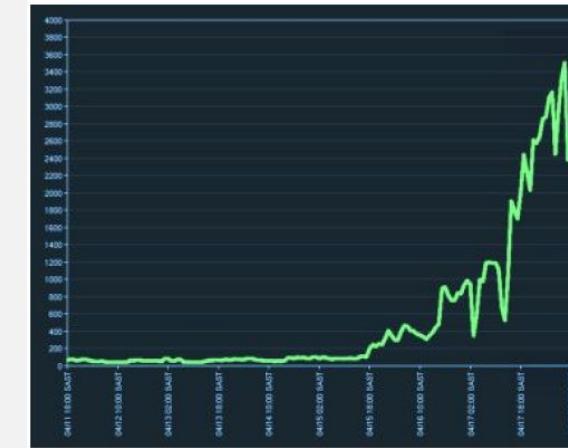
- **Monitoring**
- **Ressourcen-Management**

Skalierbarkeit: Effekte

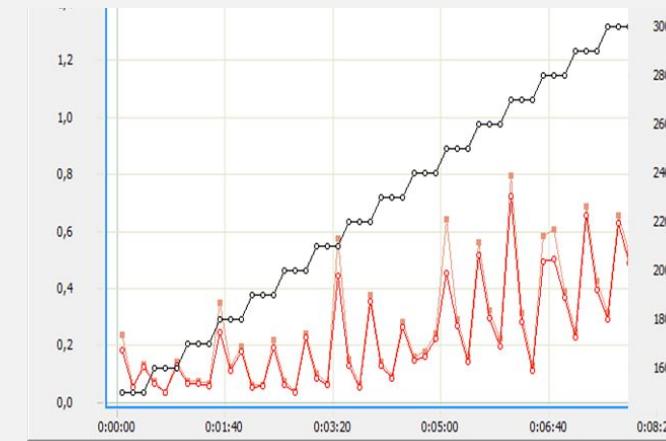
- **Tageszeitliche und saisonale Effekte:** Mittags-Peak, Prime-Time-Peak, Wochenend-Peak, Weihnachten, Valentinstag, Muttertag, ...
(vorhersehbare Belastungsspitzen)



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



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



Elastizitätsarten

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

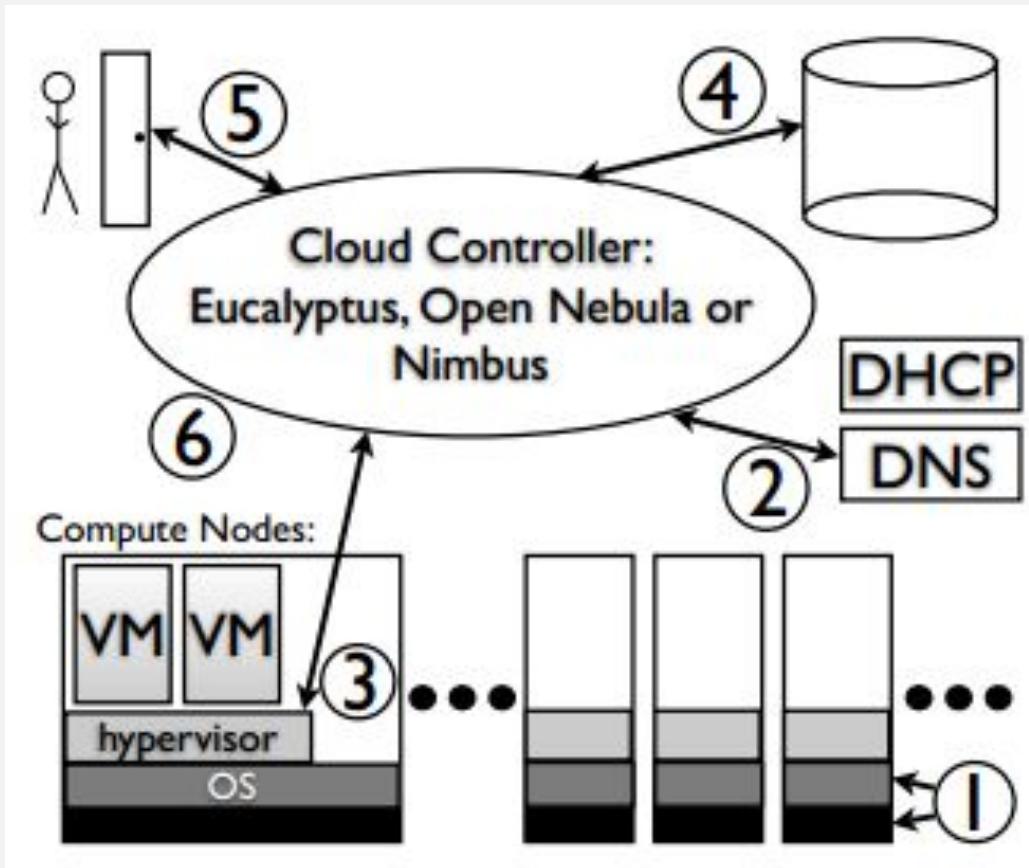
- Pseudo-Elastizität: Schneller Aufbau. Kurze Kündigungsfrist.
- Echtzeit-Elastizität: Allokation und Freigabe von Ressourcen innerhalb von Sekunden. Automatisierter Prozess mit manuellen Triggern oder nach Zeitplan.
- Selbstadaptive Elastizität: Automatische Allokation und Freigabe von Ressourcen in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

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

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

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

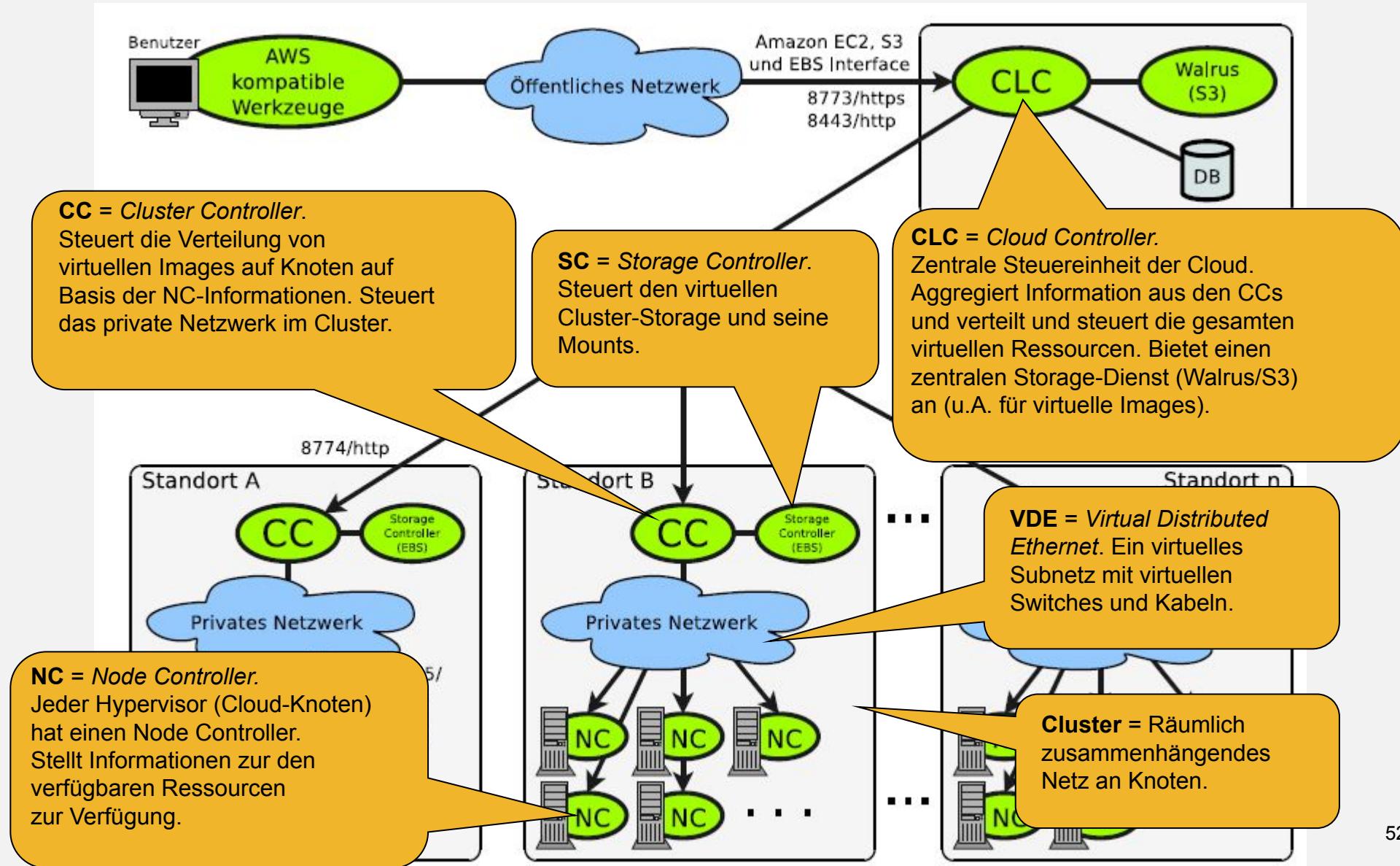
Eine IaaS-Referenzarchitektur.



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

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

Der interne Aufbau einer IaaS-Cloud am Beispiel Eucalyptus.

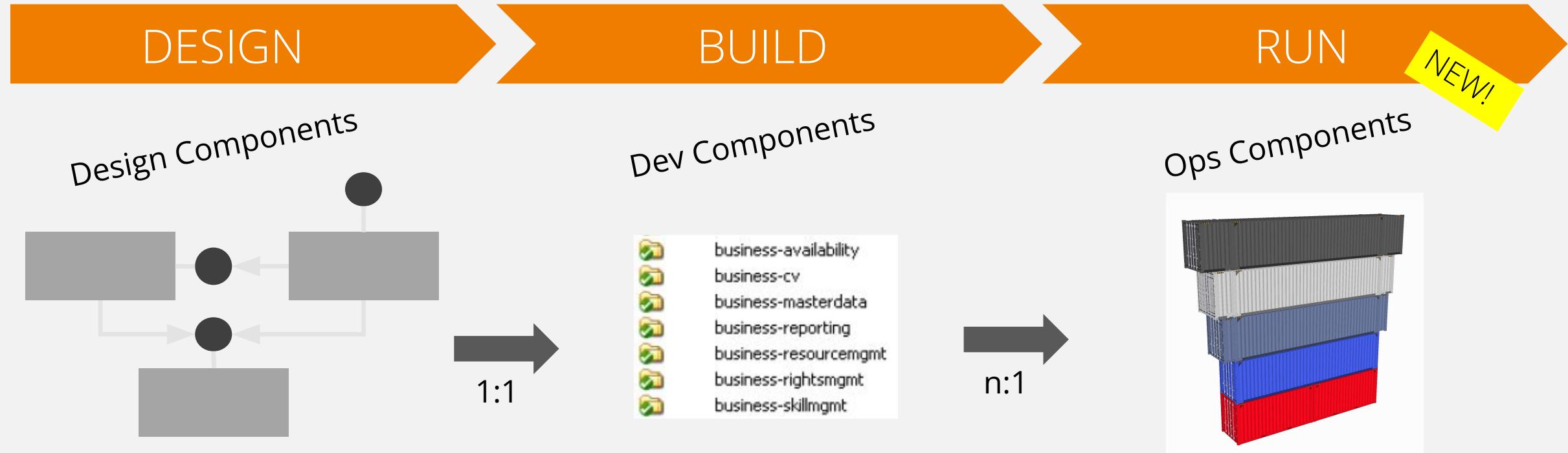




QA|WARE

Cloud-Architektur

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

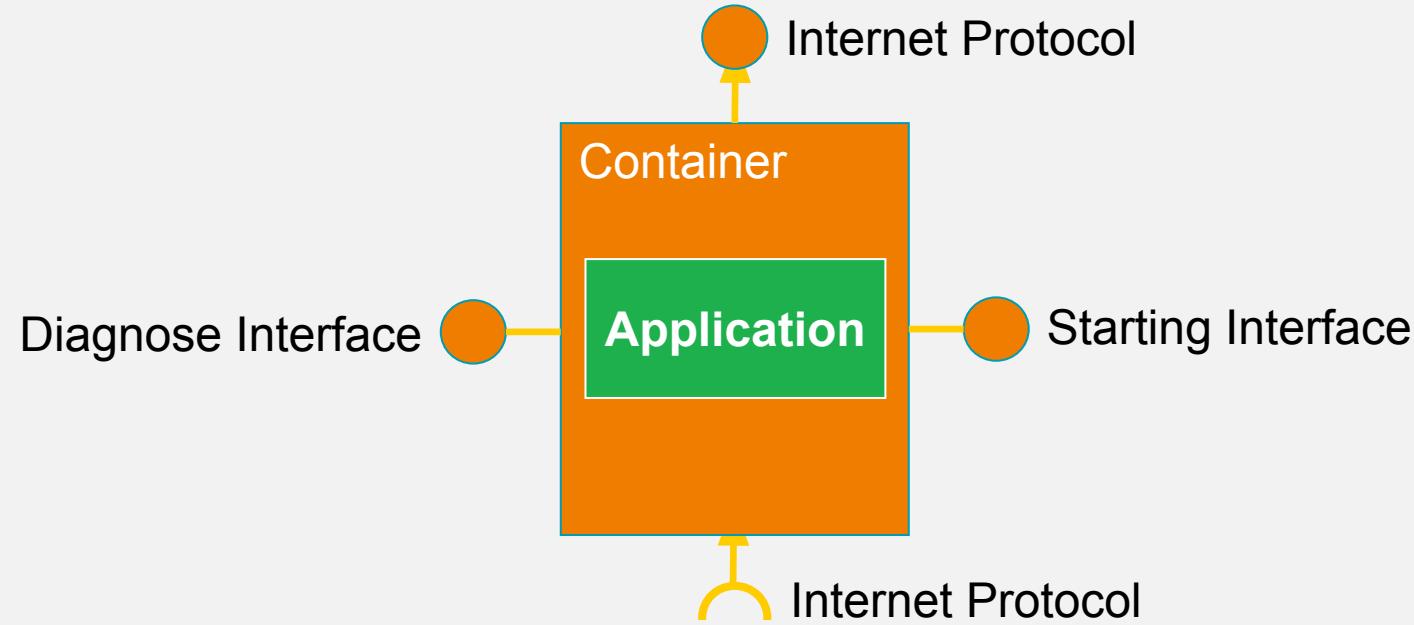


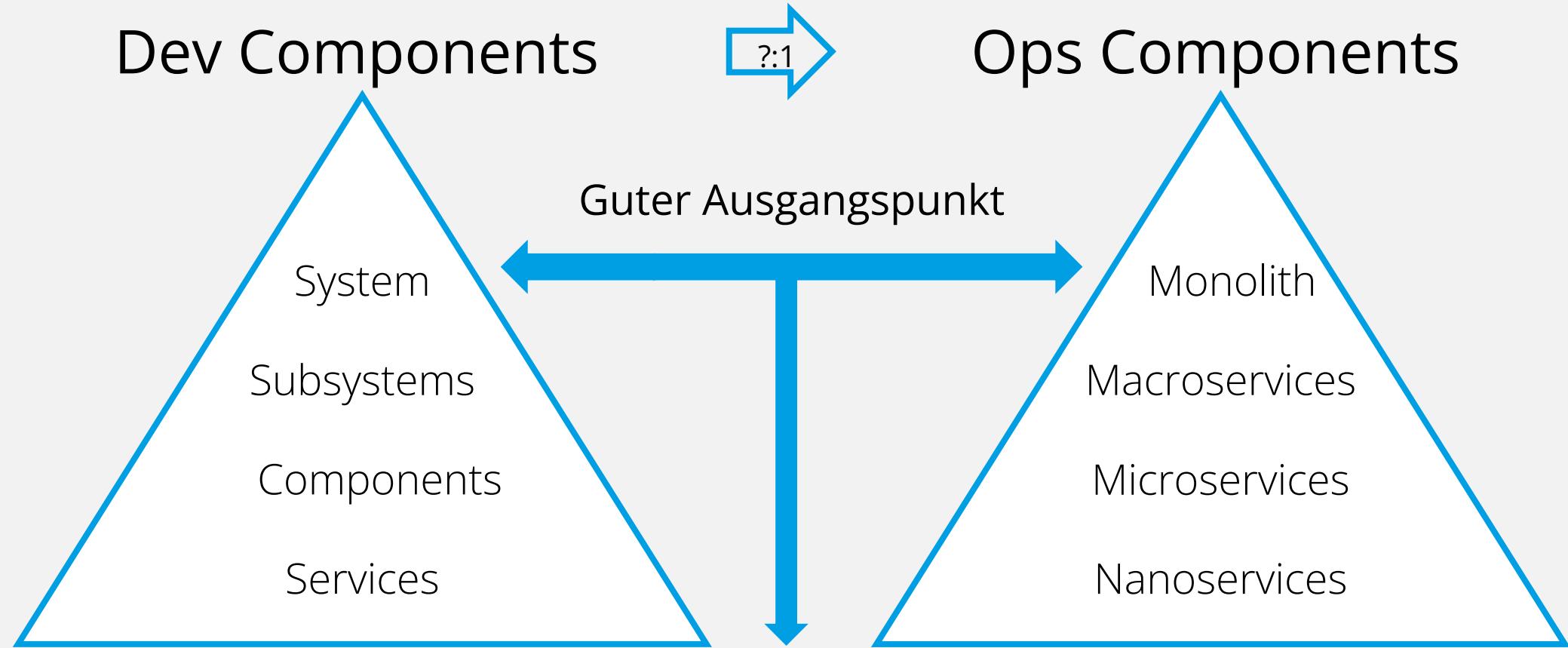
- Complexity unit
- Data integrity unit
- Coherent and cohesive features unit
- Decoupled unit

- 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.

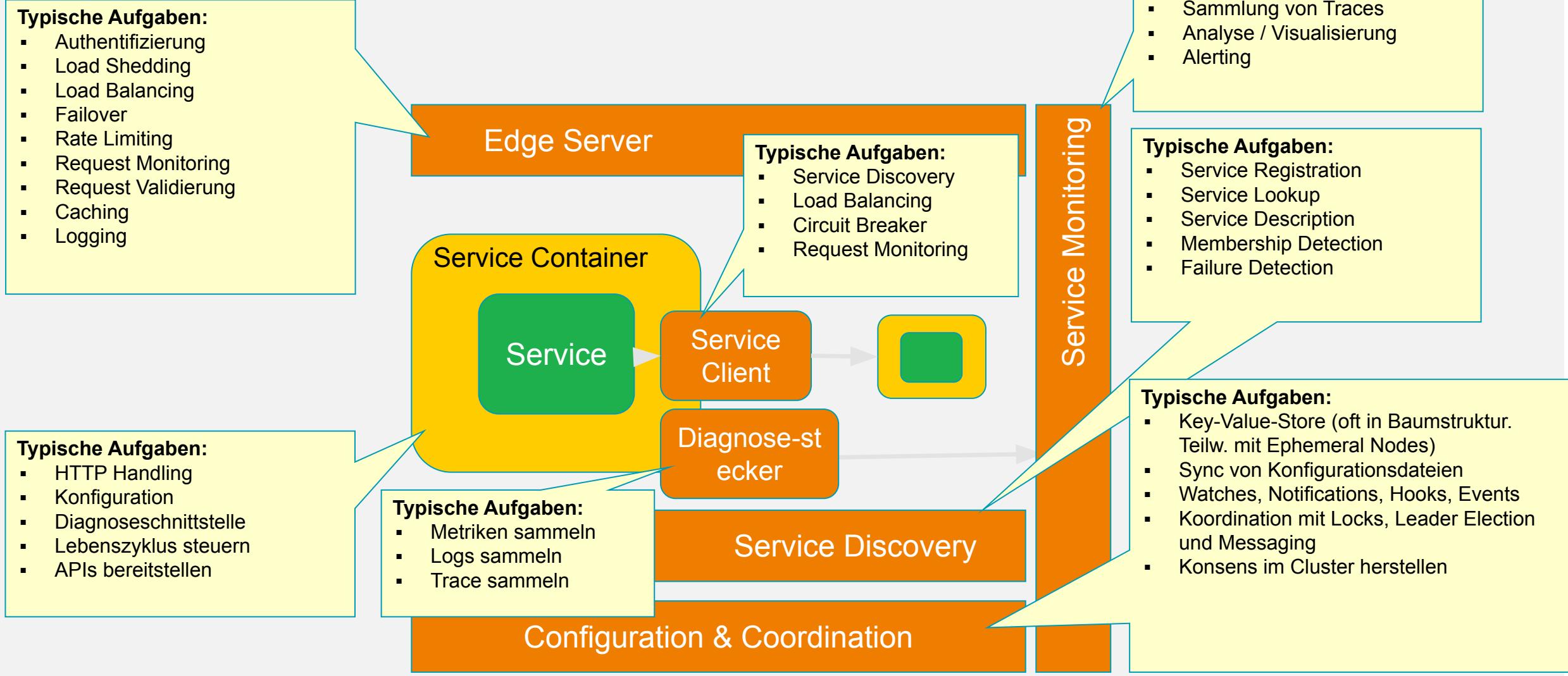




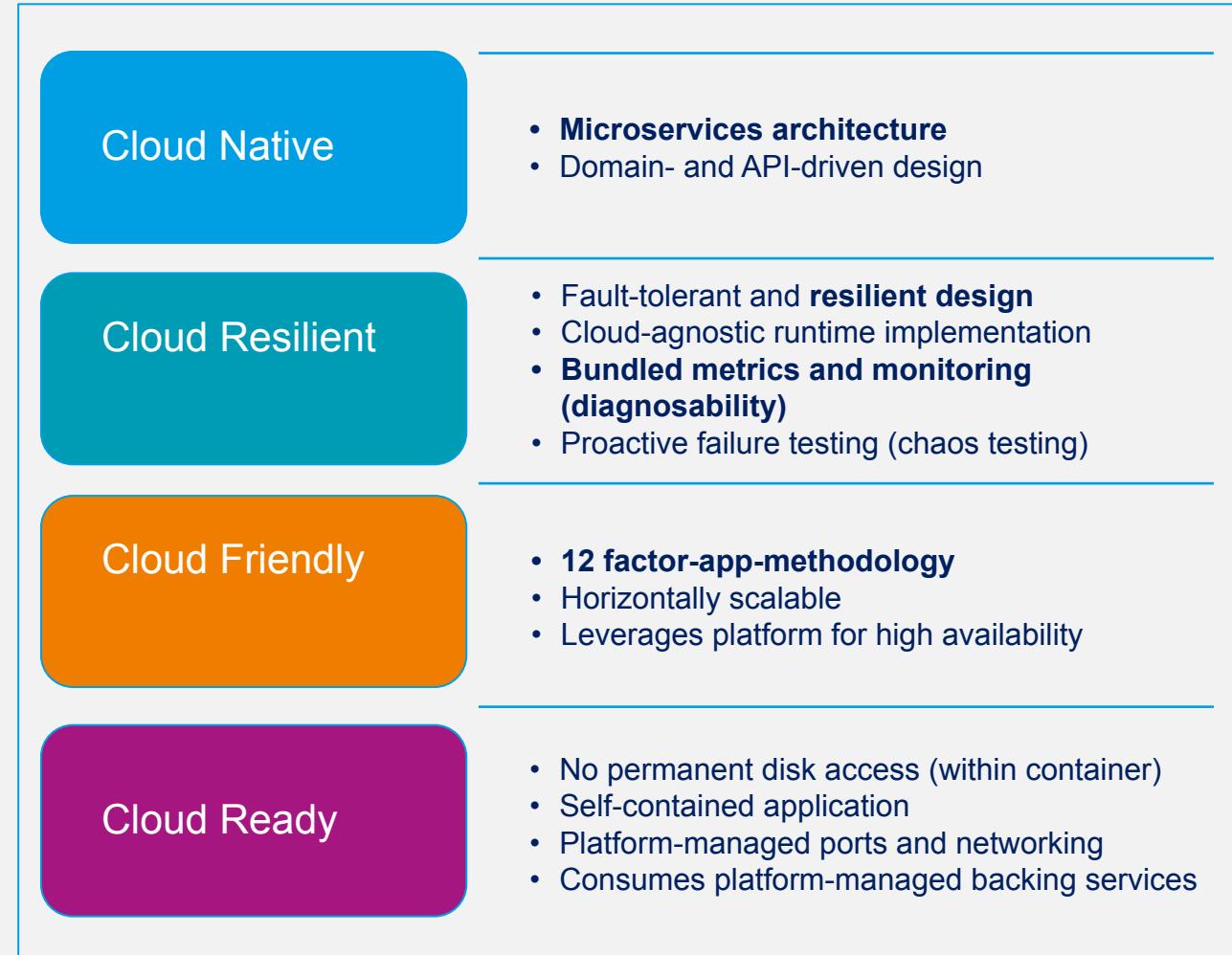
Decomposition Trade-Offs

- + 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

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



Das Cloud Native Application Reifegradmodell



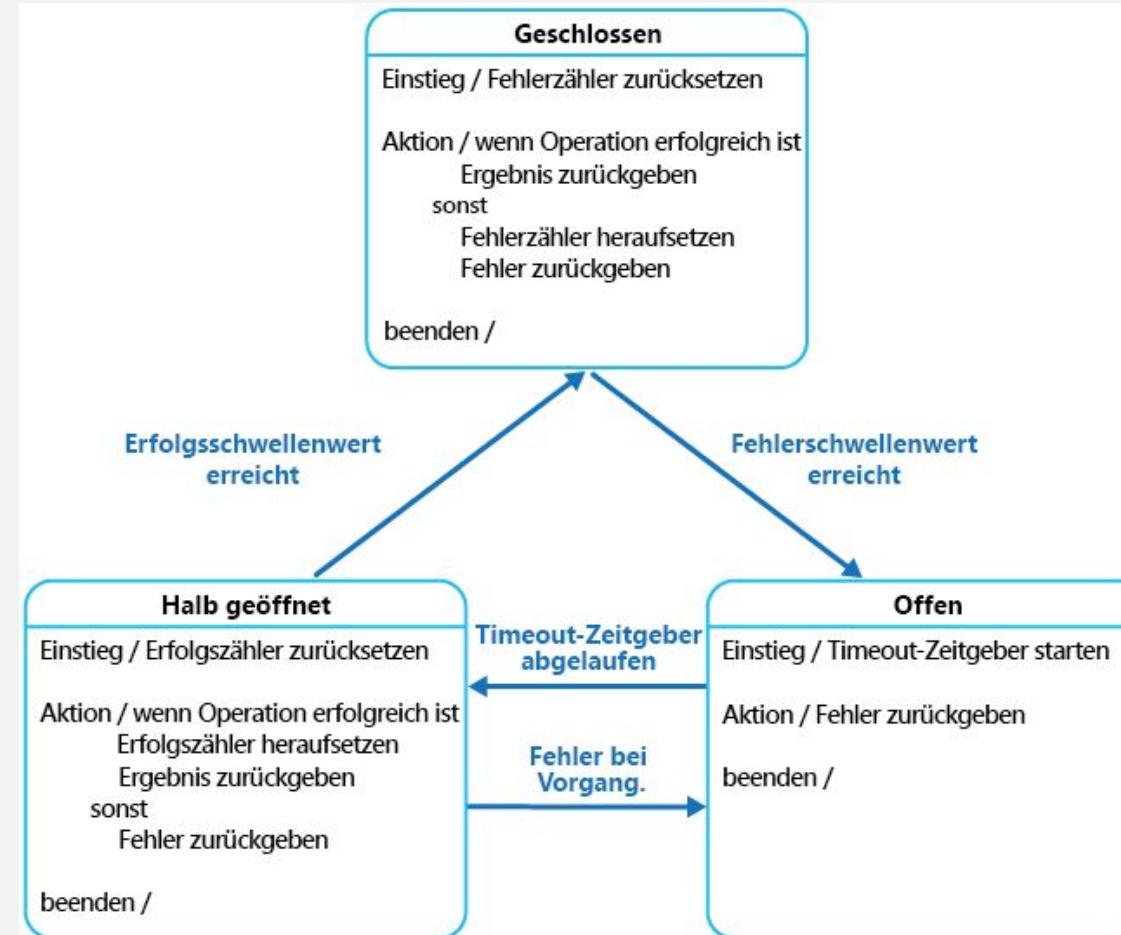
12 Factor App

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

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

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

Resilienz-Pattern: Circuit Breaker



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

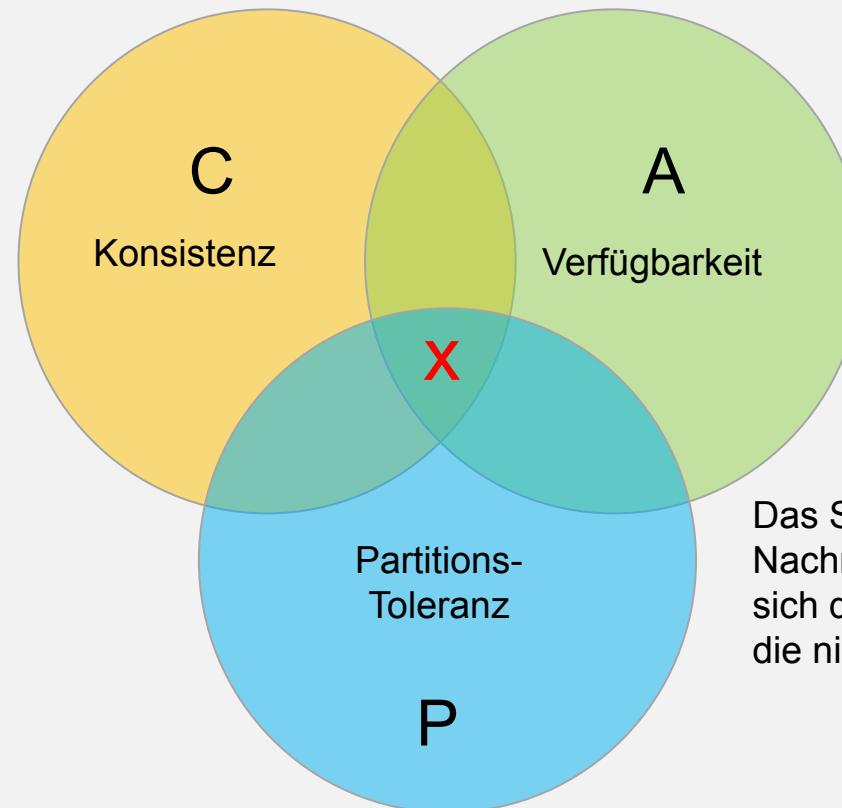
Das CAP Theorem.

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

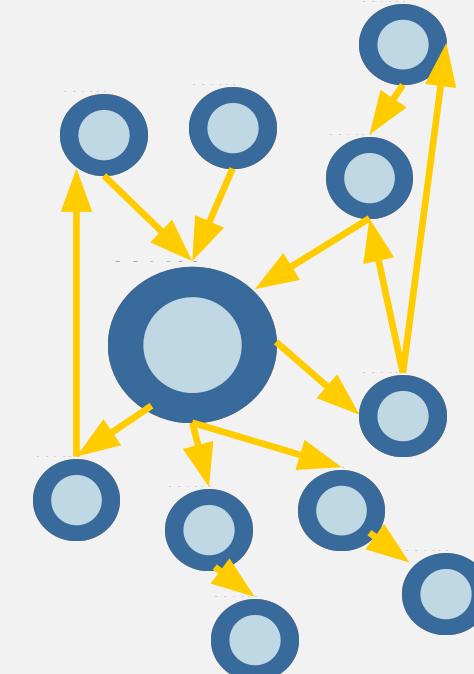
- Nachricht: Quelle, Inhalt / Zustand, Zeitstempel
- Nachrichten werden in einem festen Takt periodisch versendet an eine bestimmte Anzahl anderer Knoten (Fanout)

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.

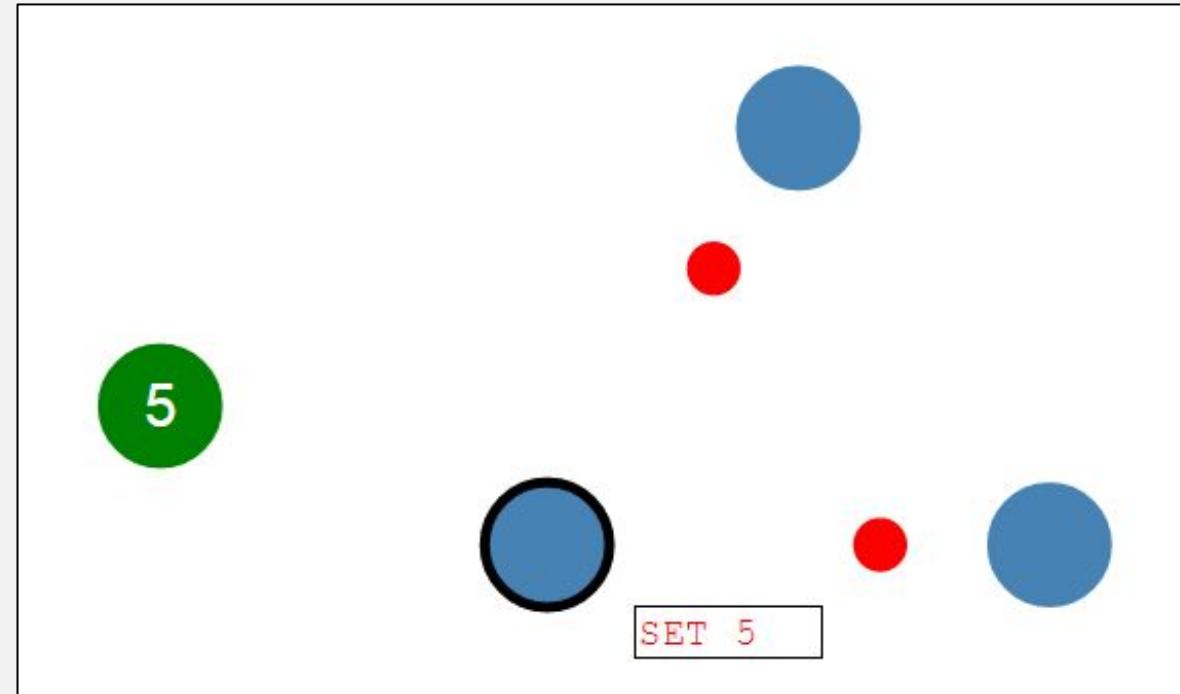
Nachteile:

- 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".

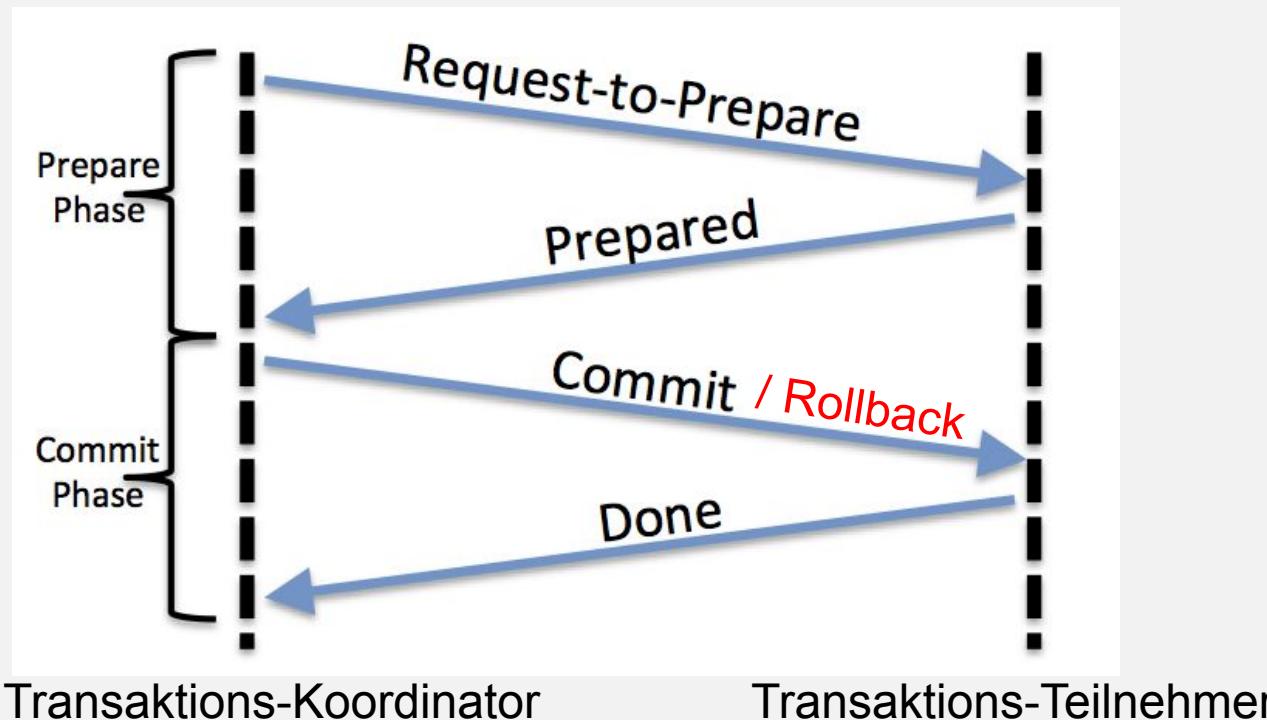


<http://thesecretlivesofdata.com/raft>

<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.



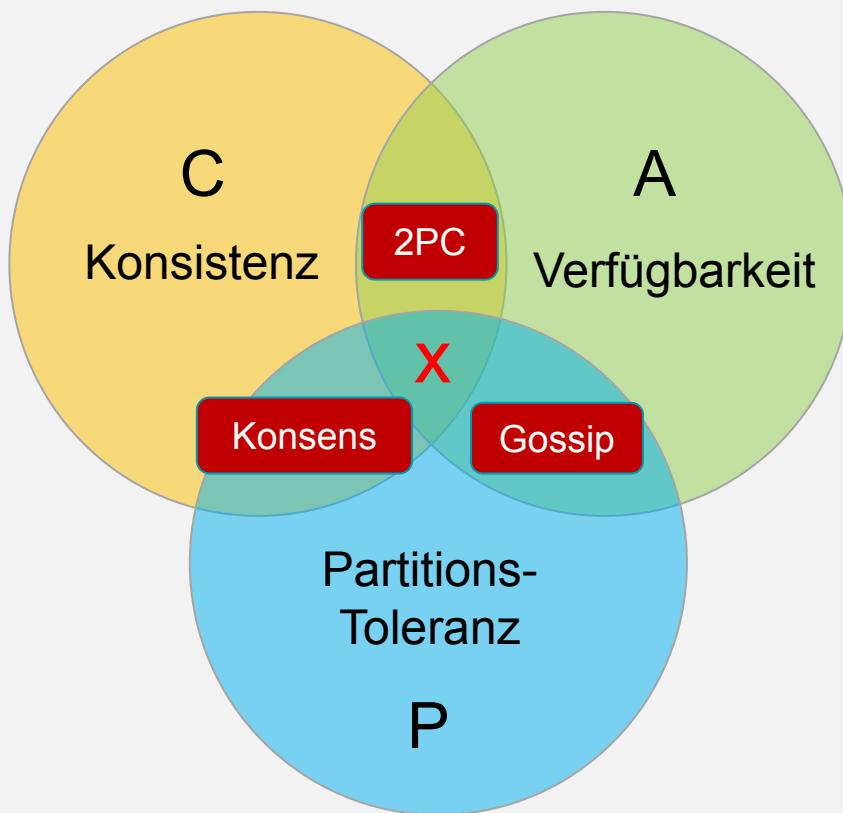
Vorteil:

- Alle Knoten sind konsistent zueinander.

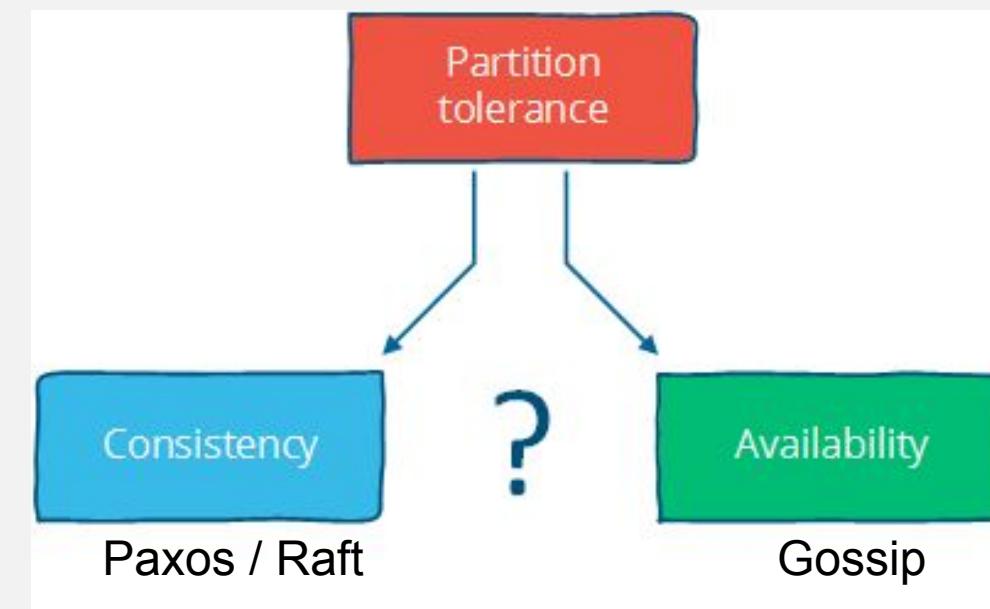
Nachteile:

- 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.





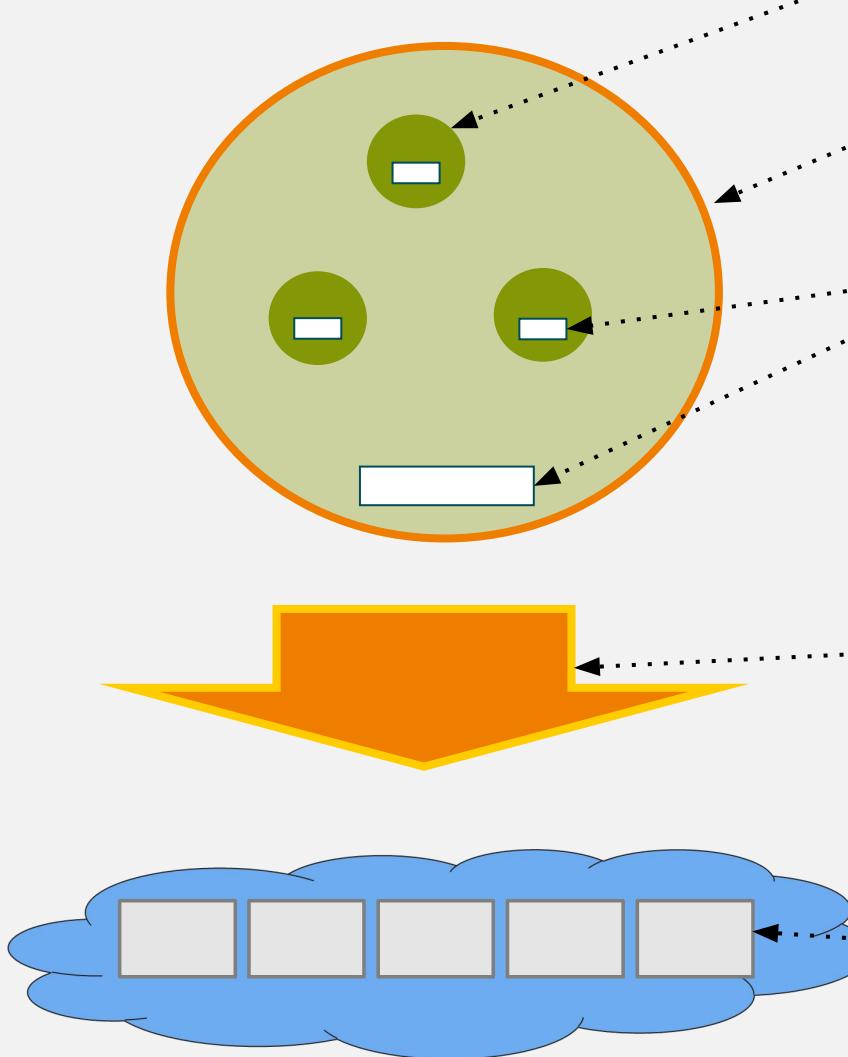
QA|WARE

Orchestrierung

Terminologie



QA|WARE



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

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

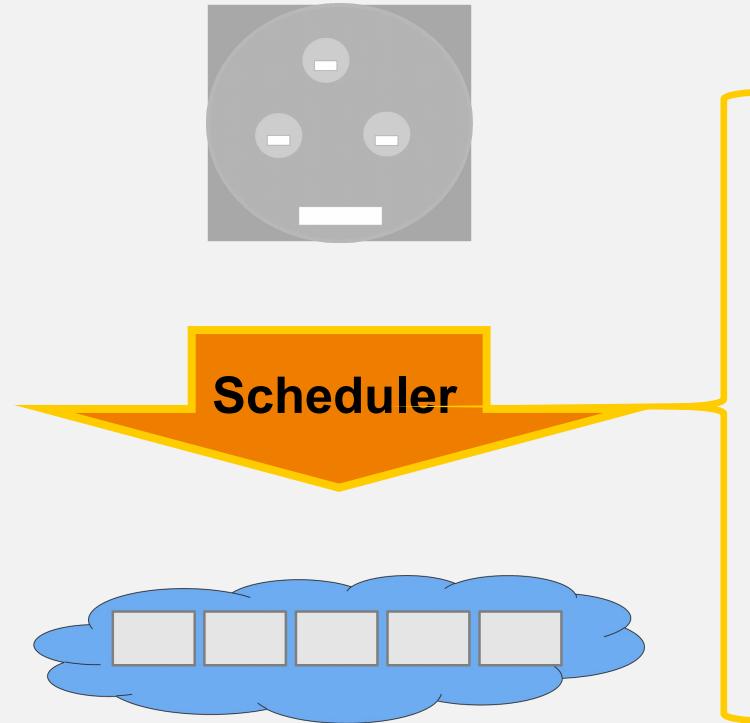
Scheduling-Ziele (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann **präemptiv** sein, also die Ausführung von Tasks unterbrechen und neu aufsetzen können.

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:



QA|WARE



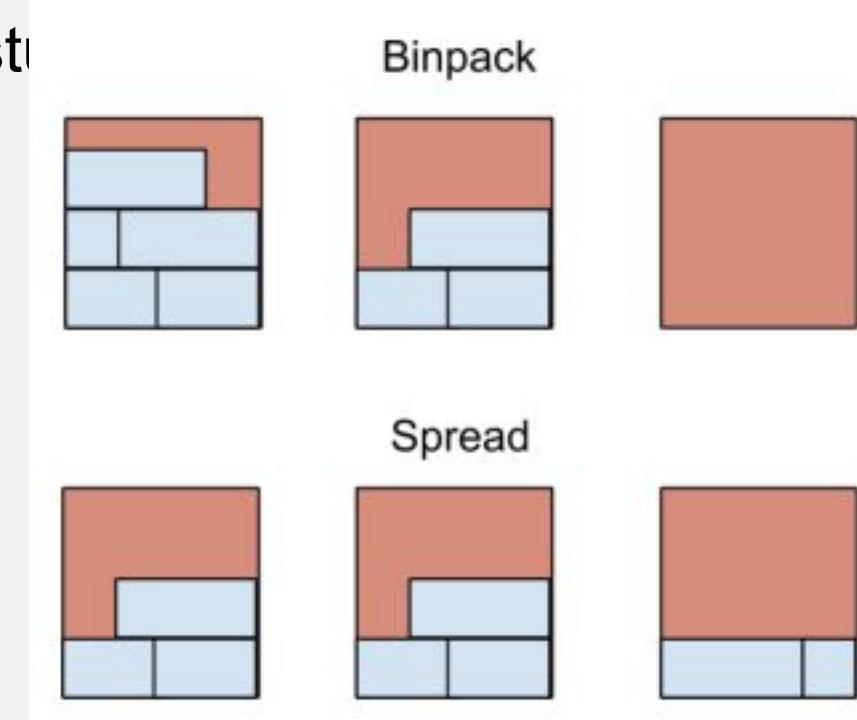
Cluster Awareness: Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, verfügbarer RAM und Festplattenspeicher sowie Netzwerkbandbreite). Dabei auch auf Elastizität reagieren.

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.

Einfache Scheduling-Algorithmen

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



Cluster-Orchestrierung

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.



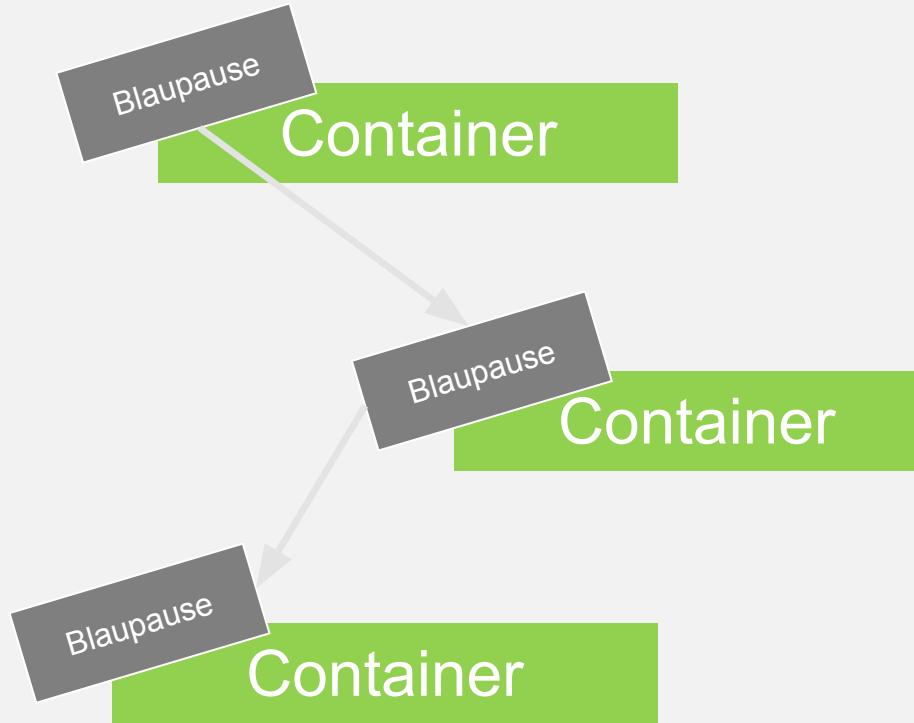
Steuerungsaktivitäten im Cluster:

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

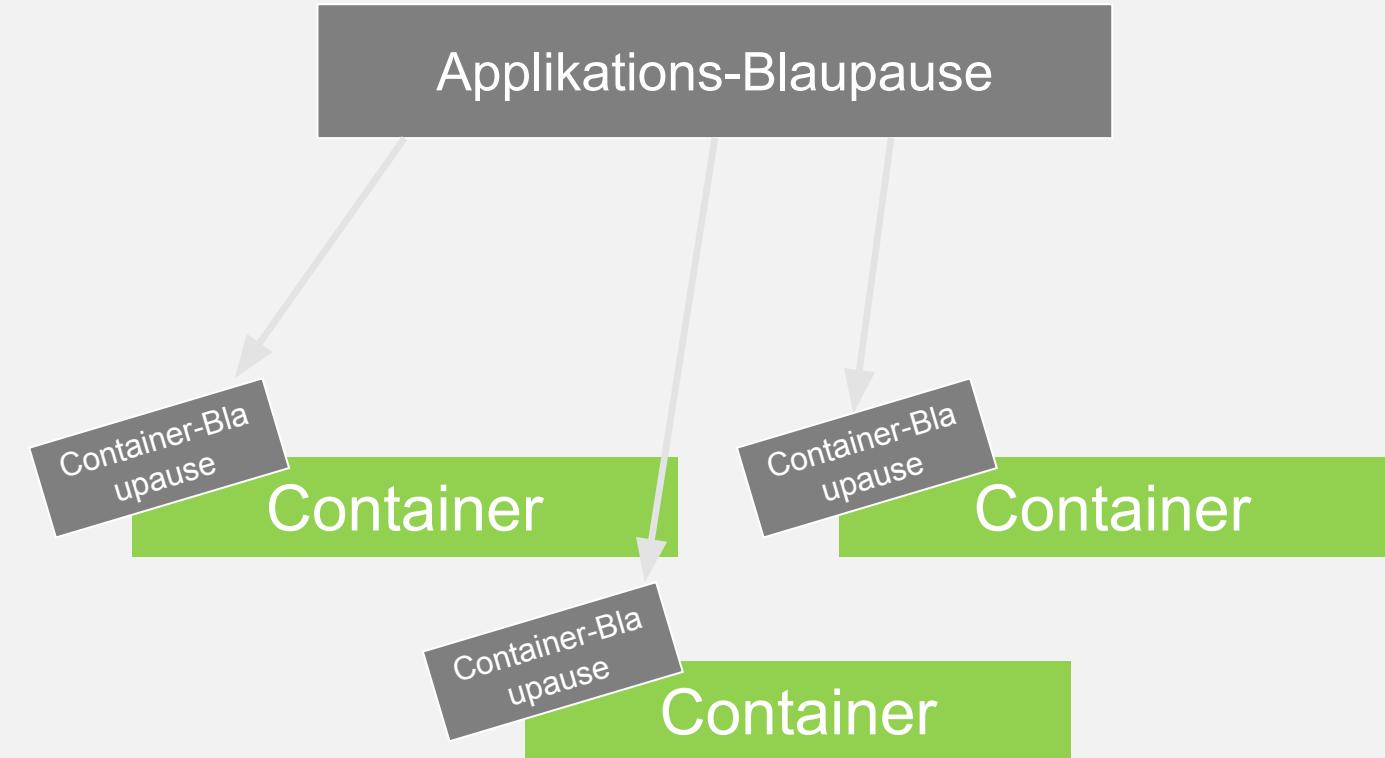
Ein Cluster-Orchestrator automatisiert vielerlei Betriebsaufgaben für Anwendung auf einem Cluster.

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

1-Level- vs. 2-Level-Orchestrierung



1-Level-Orchestrierung
(Container-Graph)



2-Level-Orchestrierung
(Container-Repository mit zentraler Bauanleitung)

Wichtige Kubernetes-Konzepte

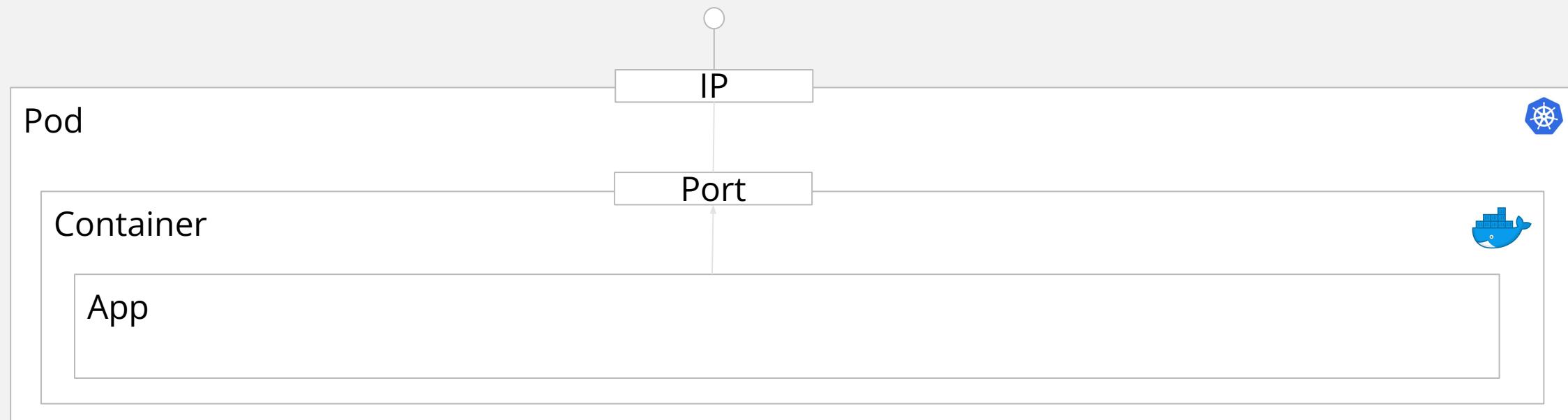


QA|WARE

Der Grundbaustein ist eure **Anwendung**.

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

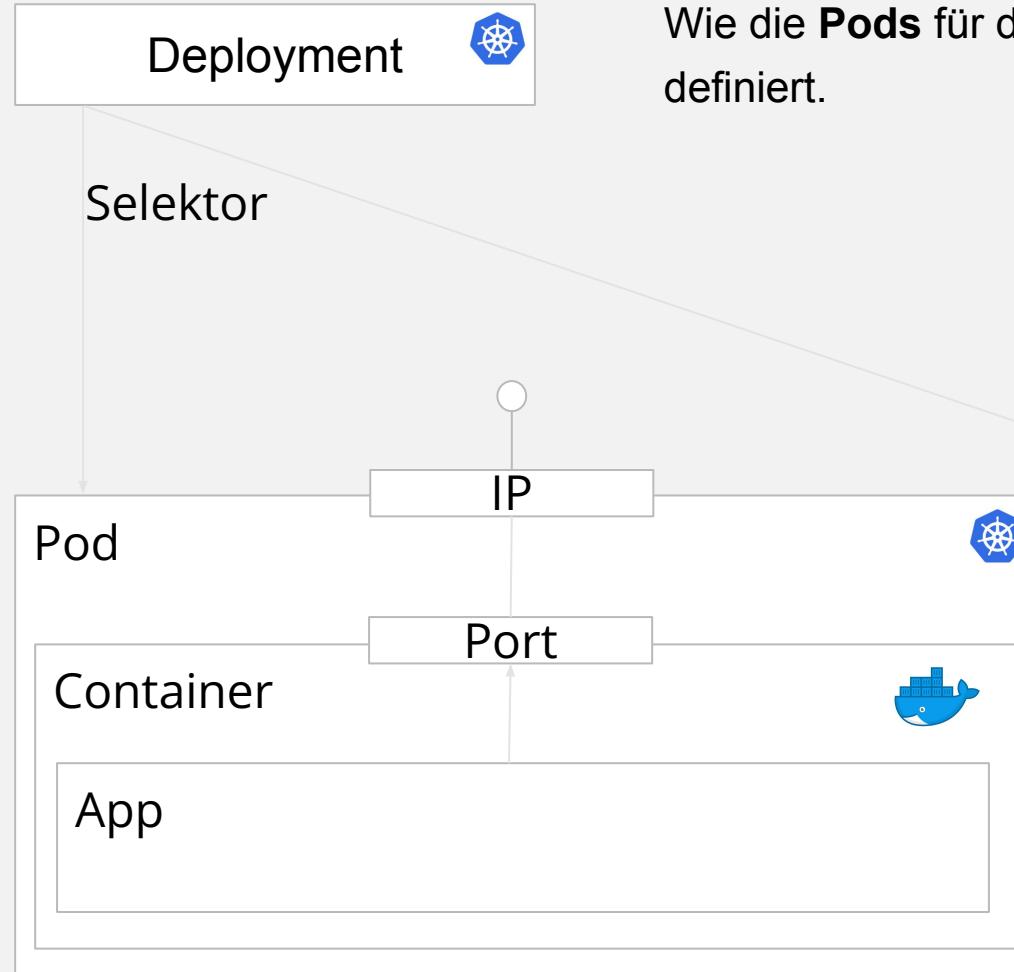
Container werden in Kubernetes zu **Pods** zusammengefasst. **Pods** haben nach außen hin eine **IP-Adresse**.



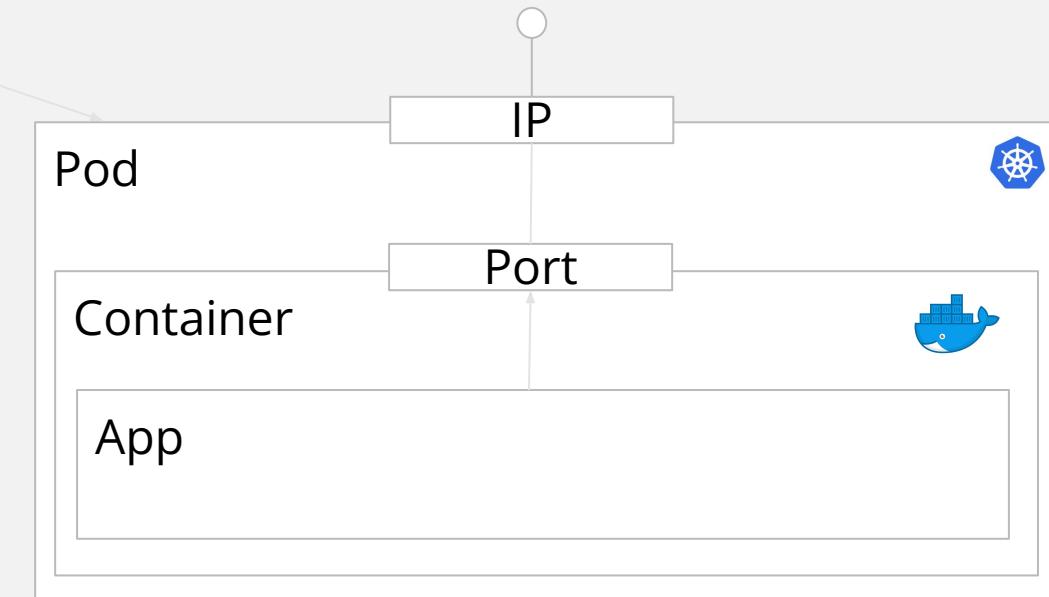
Wichtige Kubernetes-Konzepte



QA|WARE



Wie die **Pods** für die Anwendung aussehen sollen, wird in einem **Deployment** definiert.



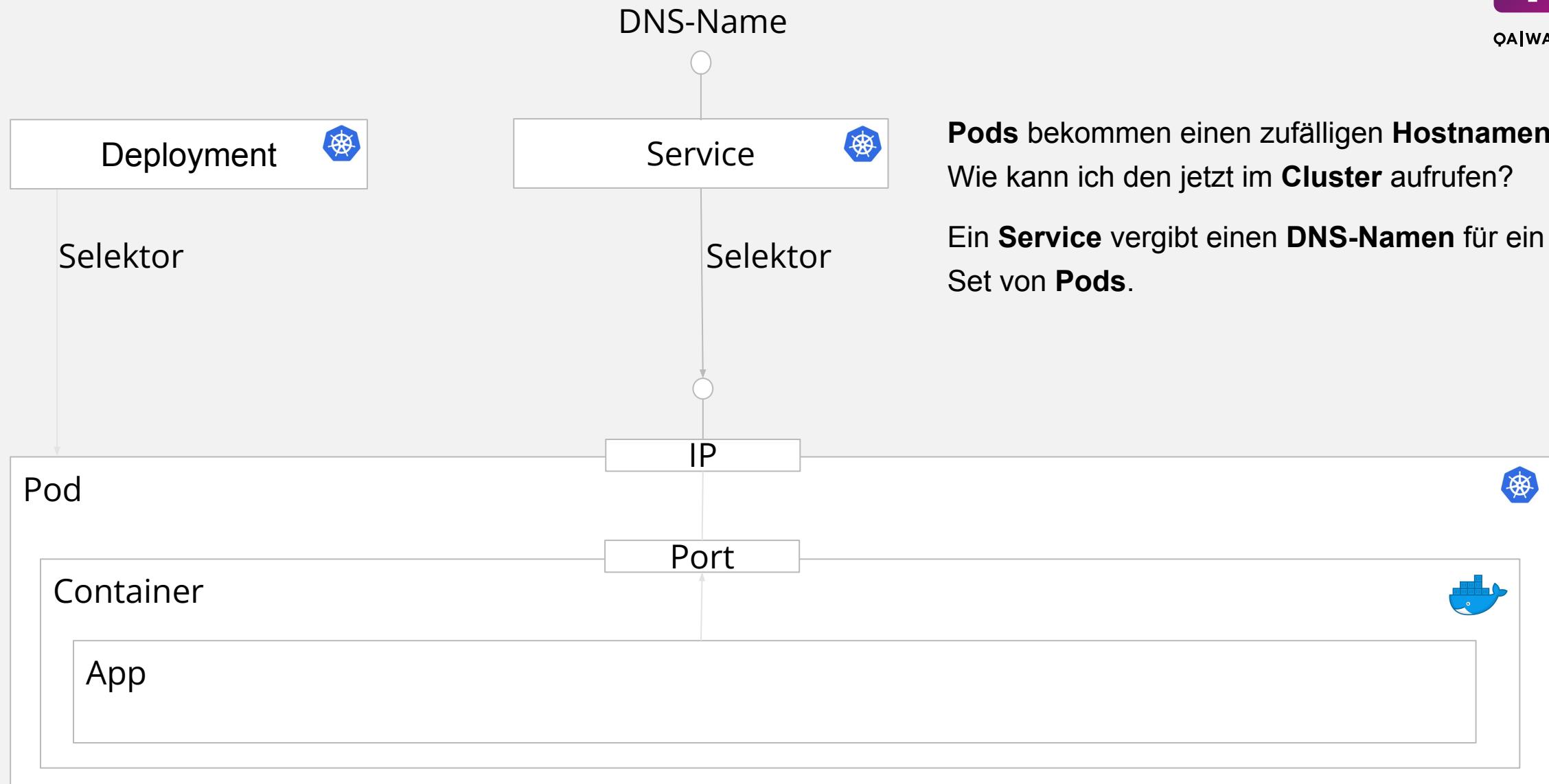
Deployment: Definition

```
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: 8080
      env:
        - name:
          value: zwitscher-consul
```

Wichtige Kubernetes Konzepte



QA|WARE



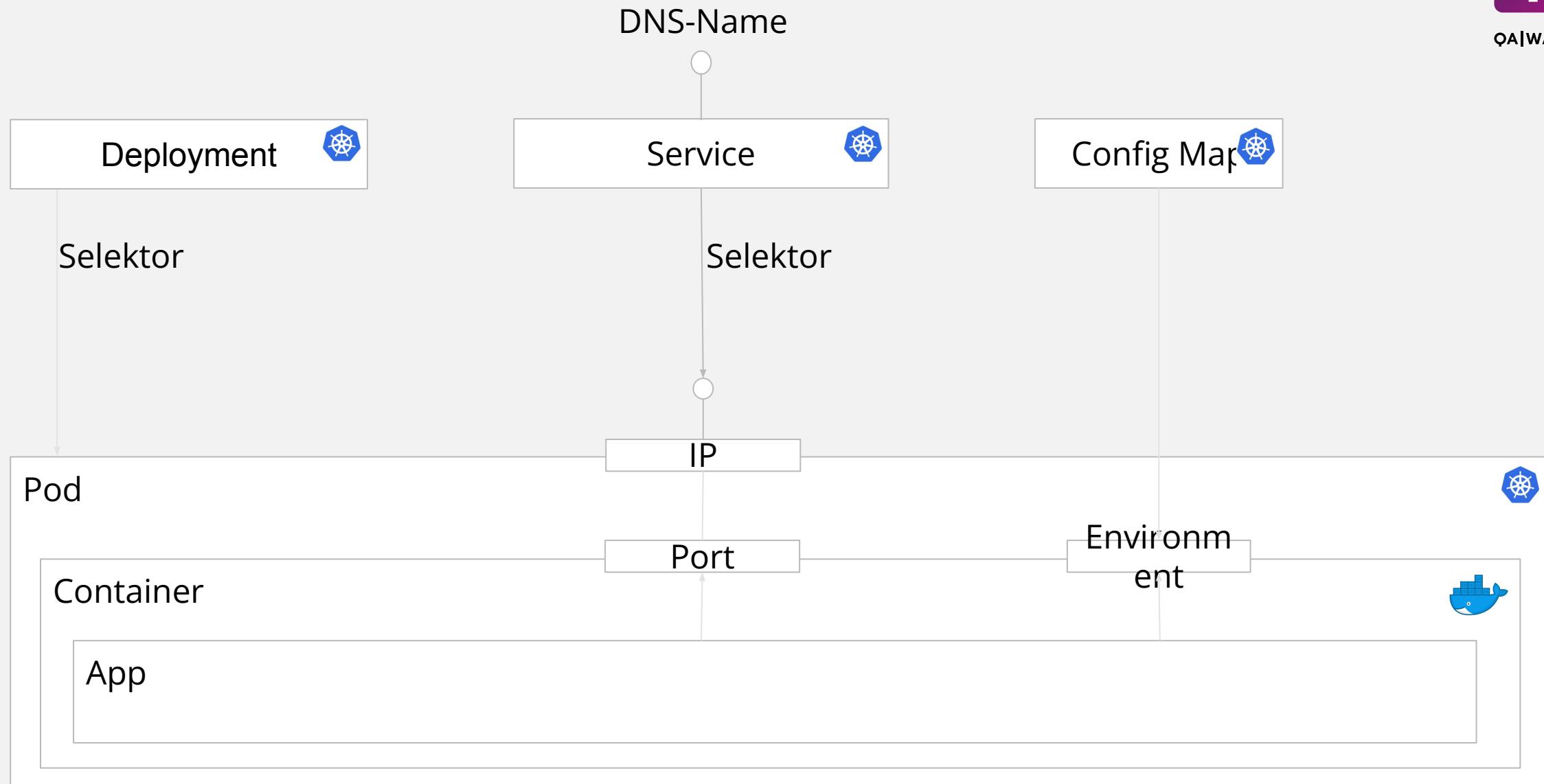
Service: Definition

```
apiVersion: v1
kind: Service
metadata:
  name: hello-service
  labels:
    app: helloservice
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:
    app: helloservice
```

Wichtige Kubernetes Konzepte



QA|WARE



Konfiguration: Config Maps (1)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: game-demo
data:
  # property-like keys; each key maps to a simple value
  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"]
      env:
        # Define the environment variable
        - name: PLAYER_INITIAL_LIVES # Notice that the case is different here
          # from the key name in the ConfigMap.
        valueFrom:
          configMapKeyRef:
            name: game-demo           # The ConfigMap this value comes from.
            key: player_initial_lives # The key to fetch.
    - name: UI_PROPERTIES_FILE_NAME
      valueFrom:
        configMapKeyRef:
          name: game-demo
          key: ui_properties_file_name
```

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"
```

```
# the Pod will be restarted if limits are exceeded
```

limits:

```
memory: "192M"
```

```
cpu: "0.5"
```

Liveness und Readiness Probes

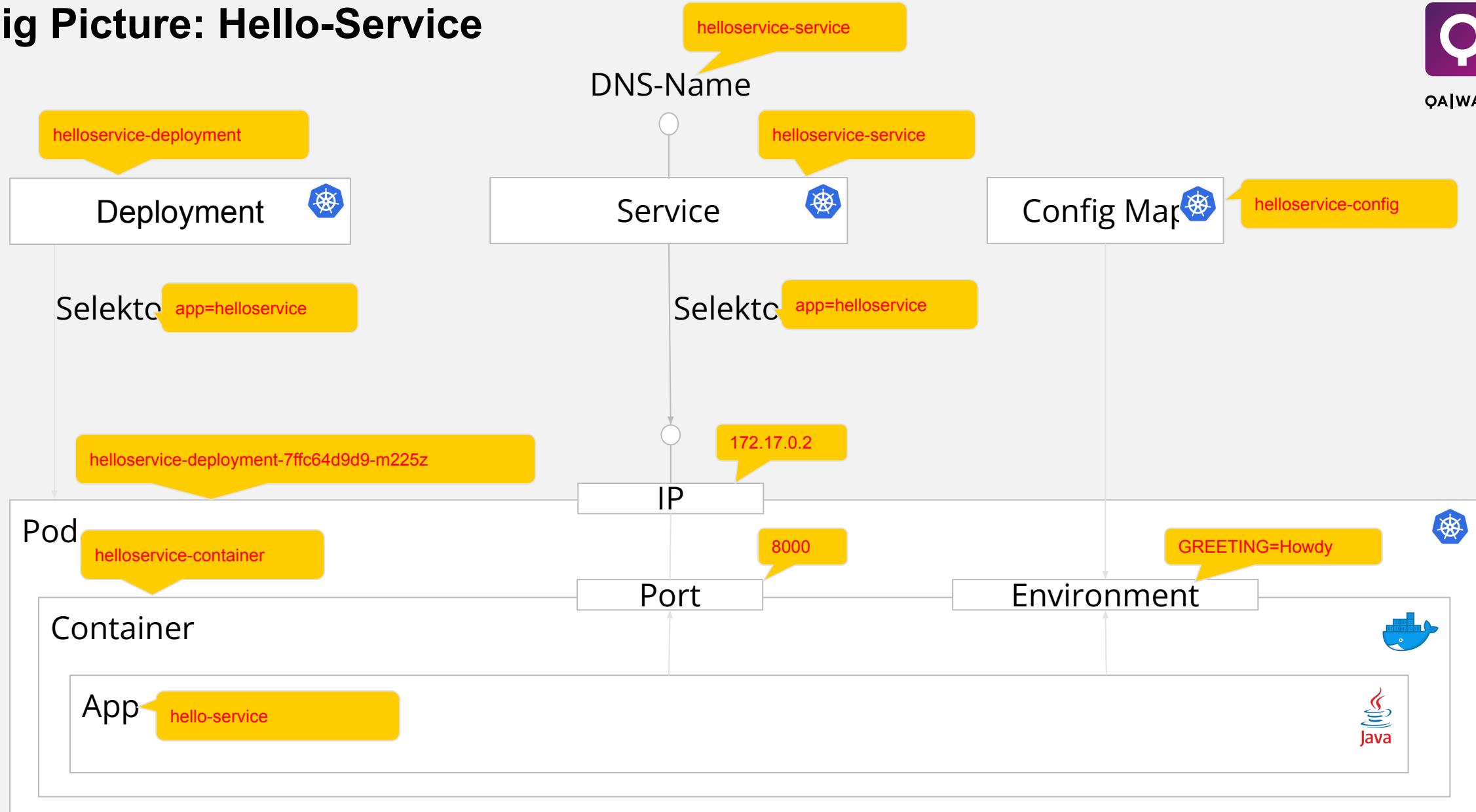


QA|WARE

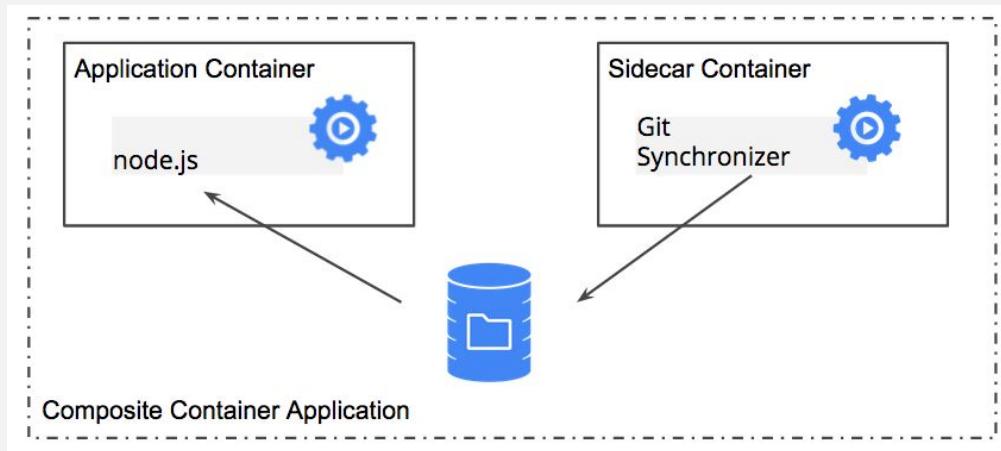
```
# container will receive requests if probe succeeds
readinessProbe:
  httpGet:
    path: /admin/info
    port: 8080
  initialDelaySeconds: 30
  timeoutSeconds: 5

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

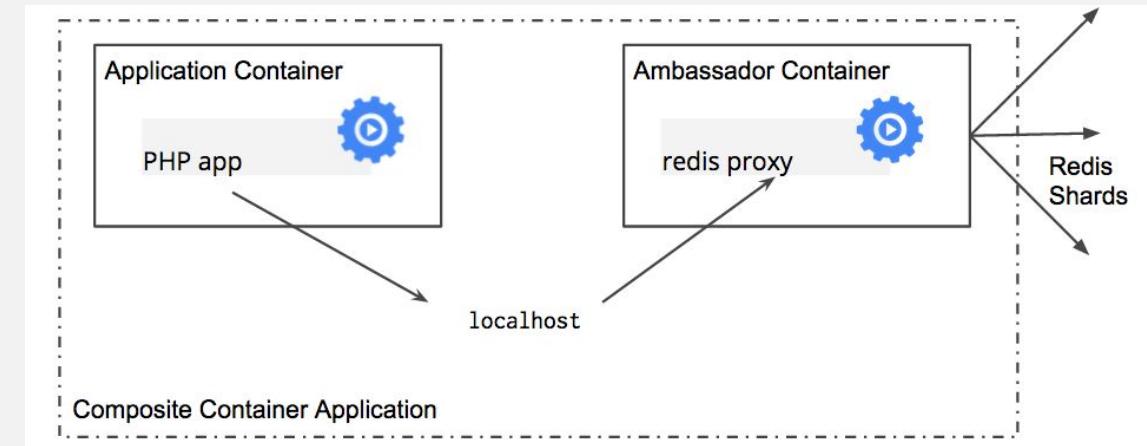
Big Picture: Hello-Service



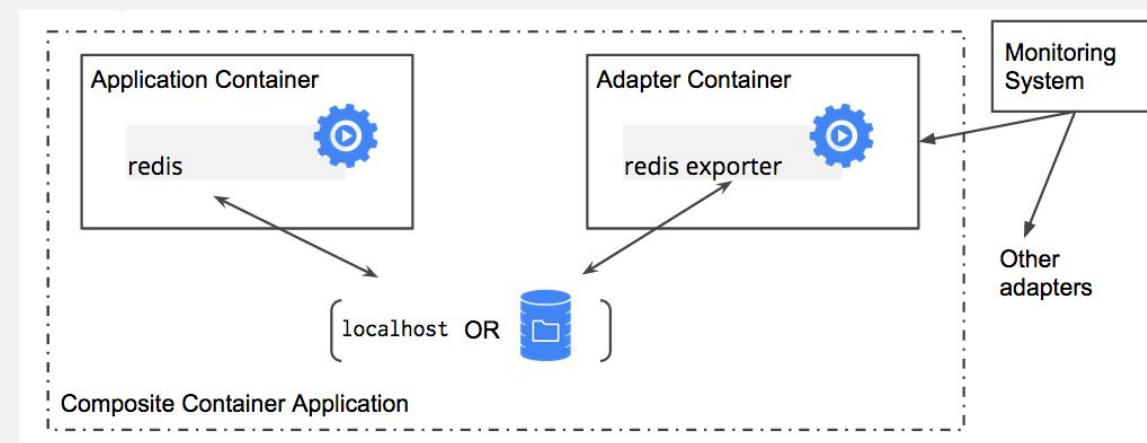
Orchestrierungsmuster – Separation of Concerns mit modularen Containern



Sidecar Container



Ambassador Container

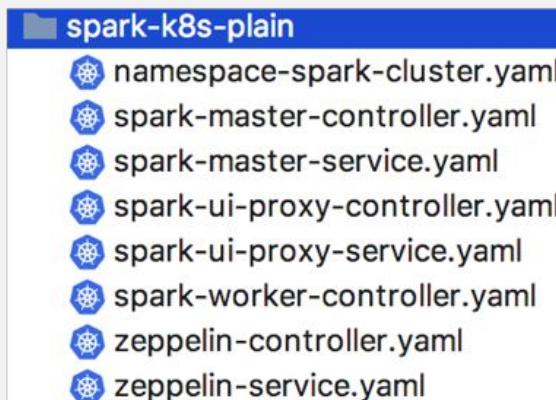


Adapter Container

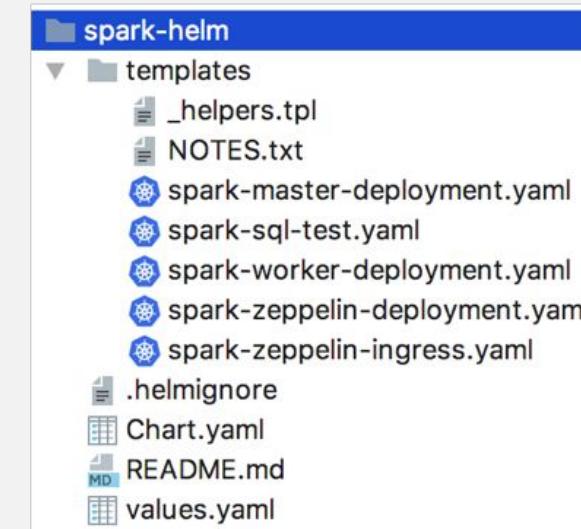
Helm: Verwaltung von Applikationspaketen für Kubernetes.



QA|WARE



- kubectl, kubectl, kubectl, ...
- Konfiguration?
- Endpunkte?



- Chart suchen auf <https://hub.kubeapps.com>
- Doku dort lesen (README.md)
- Konfigurationsparameter lesen:
`helm inspect stable/spark`
- Chart starten mit überschriebener Konfiguration:
`helm install --name my-release -f values.yaml stable/spark`



QA|WARE

Service Meshes

Was ist ein Service Mesh?



QA|WARE

*In software architecture, a **service mesh** is a dedicated infrastructure layer for facilitating service-to-service communications between services or microservices using a proxy.*

A dedicated communication layer can provide numerous benefits, such as providing observability into communications, providing secure connections or automating retries and backoff for failed requests.

Ein Service Mesh bietet uns...



QA|WARE

Traffic Management

- Load Balancing
- Service Discovery
- Routing
- Retries
- Timeouts
- Circuit Breaker

Security

- Encryption
- Authentication and Authorization
- Rate Limiting

Observability

- Metrics
- Tracing
- (Logging)

High Level Architektur von Service Meshes



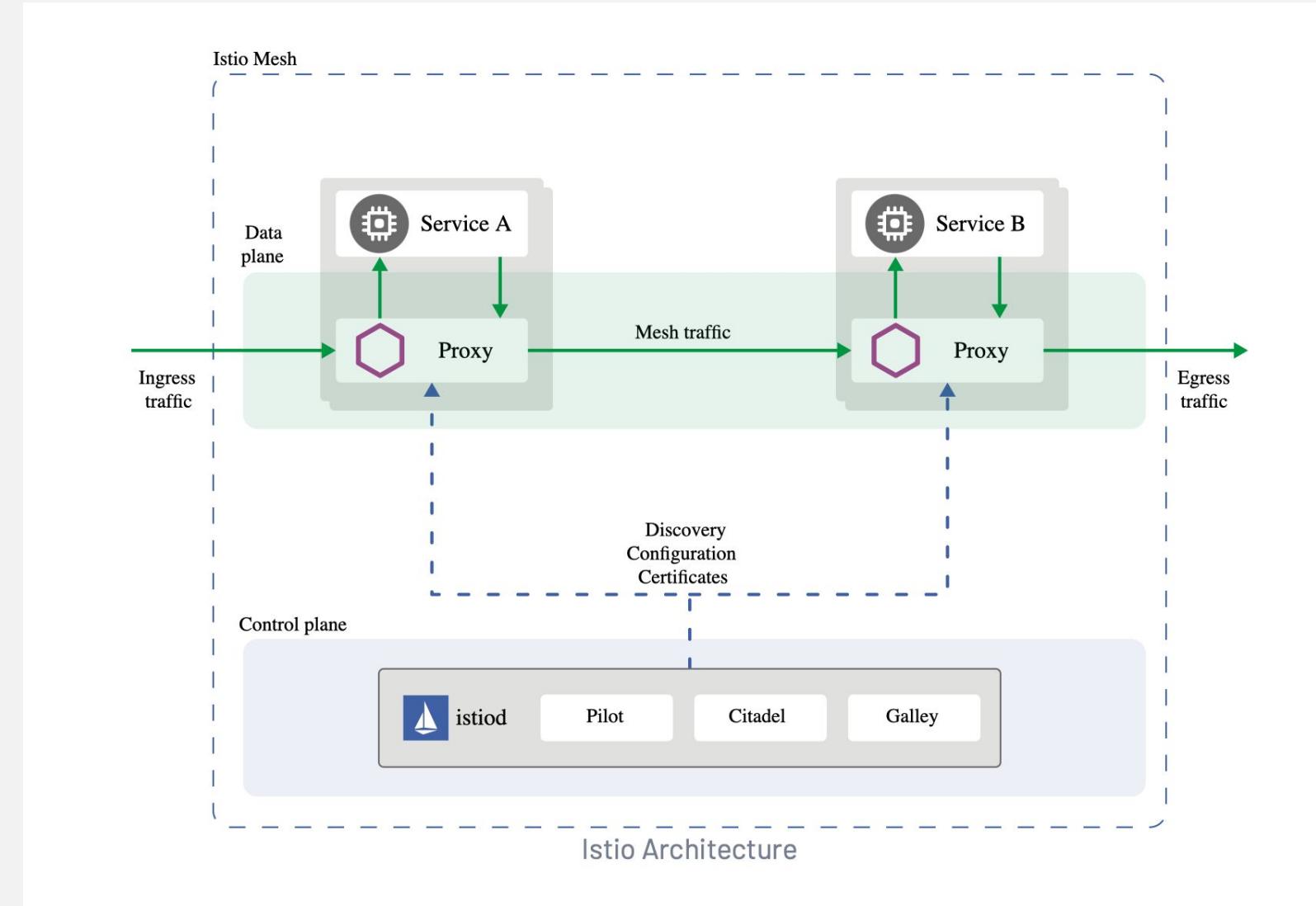
QA|WARE

Control Plane:

- umfasst Management Komponenten, sozusagen das "Hirn" des Service Meshes.
- Typische Komponenten umfassen Service Discovery & Traffic rules, Configuration Stores & APIs, Identity & Credential Management

Data Plane:

- umfasst workloads
- umfasst proxy Komponenten, die features des service Meshes implementieren



Data plane Architekturen - Sidecar



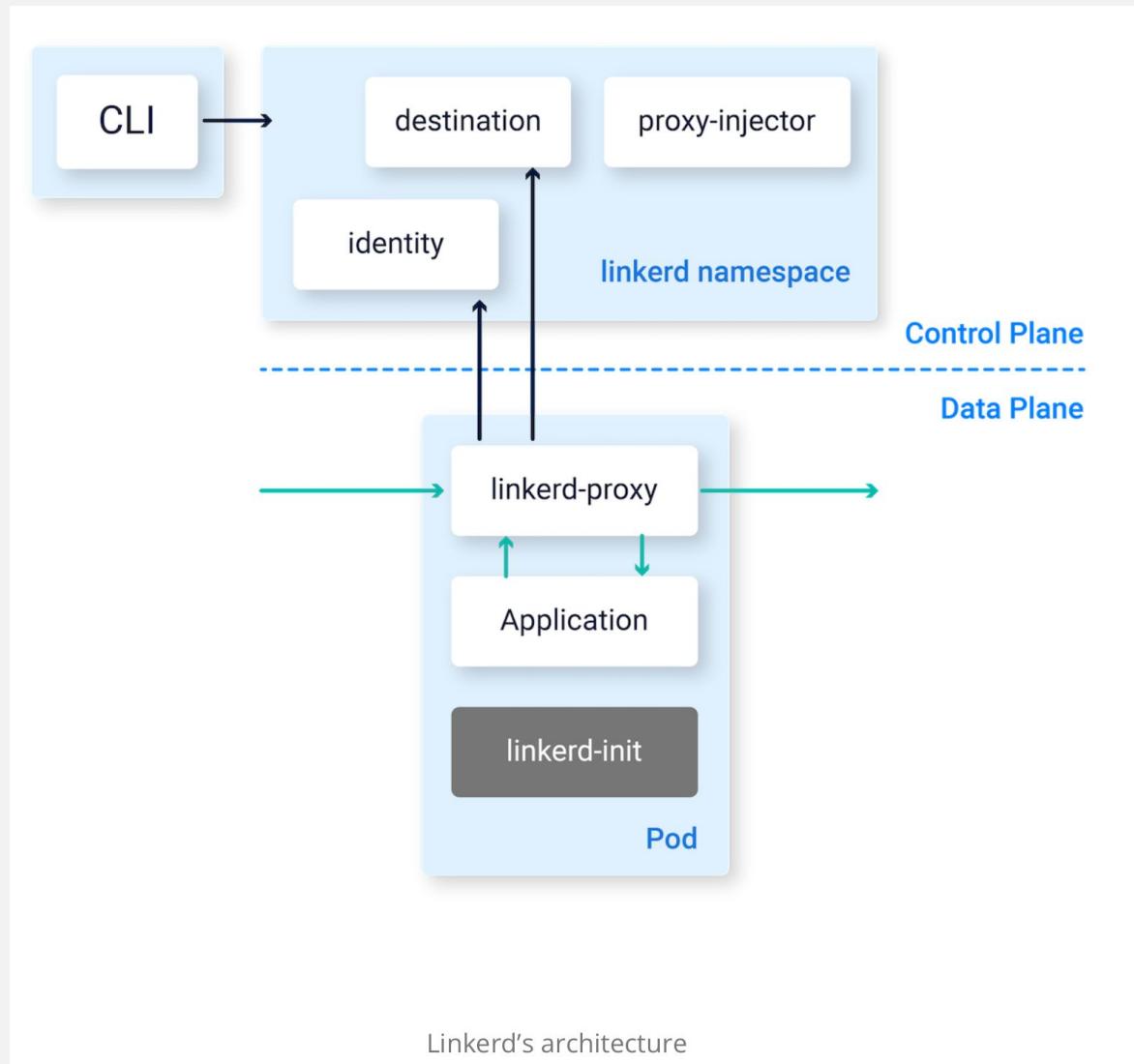
QA|WARE

Sidecar-Pattern:

Es wird ein extra Container in jedem Applikations Pod hochgefahren

Im Falle von Linkerd:

- Jeder Pod im Mesh bekommt automatisch einen Proxy injected
- Über IP-Tables Regeln wird eingehender sowie ausgehender Traffic über den Sidecar Proxy geleitet. Die Regeln werden entweder über den linkerd-init startup container gesetzt, oder über ein CNI-Plugin implementiert.
- Der Sidecar Proxy implementiert die Mesh Funktionalität



Data plane Architekturen - Sidecar



QA|WARE

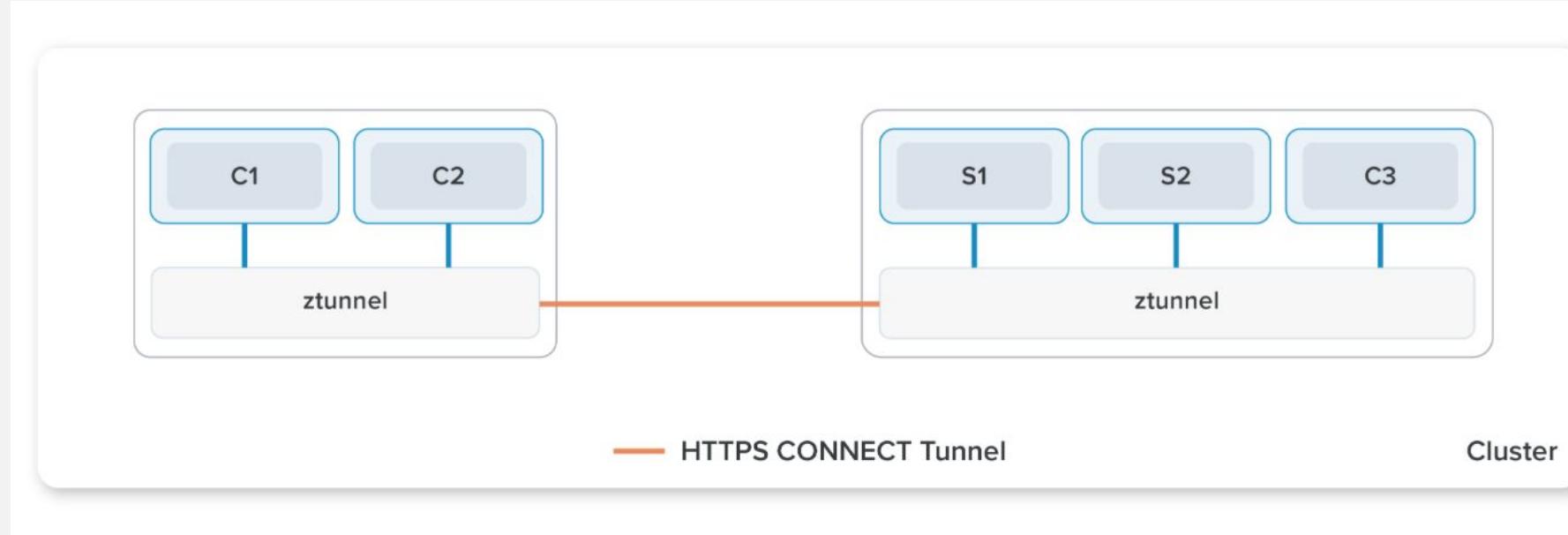
Vorteile:

- **Resilienz:** Kein single point of failure - da ein proxy pro Applikations Pod
- **Uniform:** Traffic wird uniform für alle Applikationen identisch durch den Proxy geschleift
- **Multi-Tenancy:** Es gibt in der Dataplane keine geteilte Infrastruktur zwischen verschiedenen Parteien
- **Skalierbarkeit:** Proxy skaliert horizontal mit Applikation
- **Zero code changes:** Die Anwendung kann ohne Anpassungen im Code gemeshed werden

Nachteile:

- **Resource Overhead:** Jeder Pod im Mesh bekommt automatisch einen Proxy injected, der wiederum Ressourcen wie CPU und RAM verbraucht.
- **Höhere Latenz:** Jeglicher Traffic wird durch Proxies geschleift, wodurch mehr Hops notwendig sind. Die Latenz verschlechtert sich. Abhängig von der Applikation kann das ein Problem darstellen.
- **Resource Management:** Der Sidecar Proxy läuft als Container in Kubernetes und unterliegt daher auch den typischen resource constraints. Diese müssen sinnvoll gewählt und gepflegt werden, um eine optimale Ressourcennutzung zu gewährleisten.
- **Security:** Es wird eine neue Infrastrukturkomponente deployed. Das erhöht potenziell die Angriffsfläche

Data plane Architekturen - Ambient Mesh

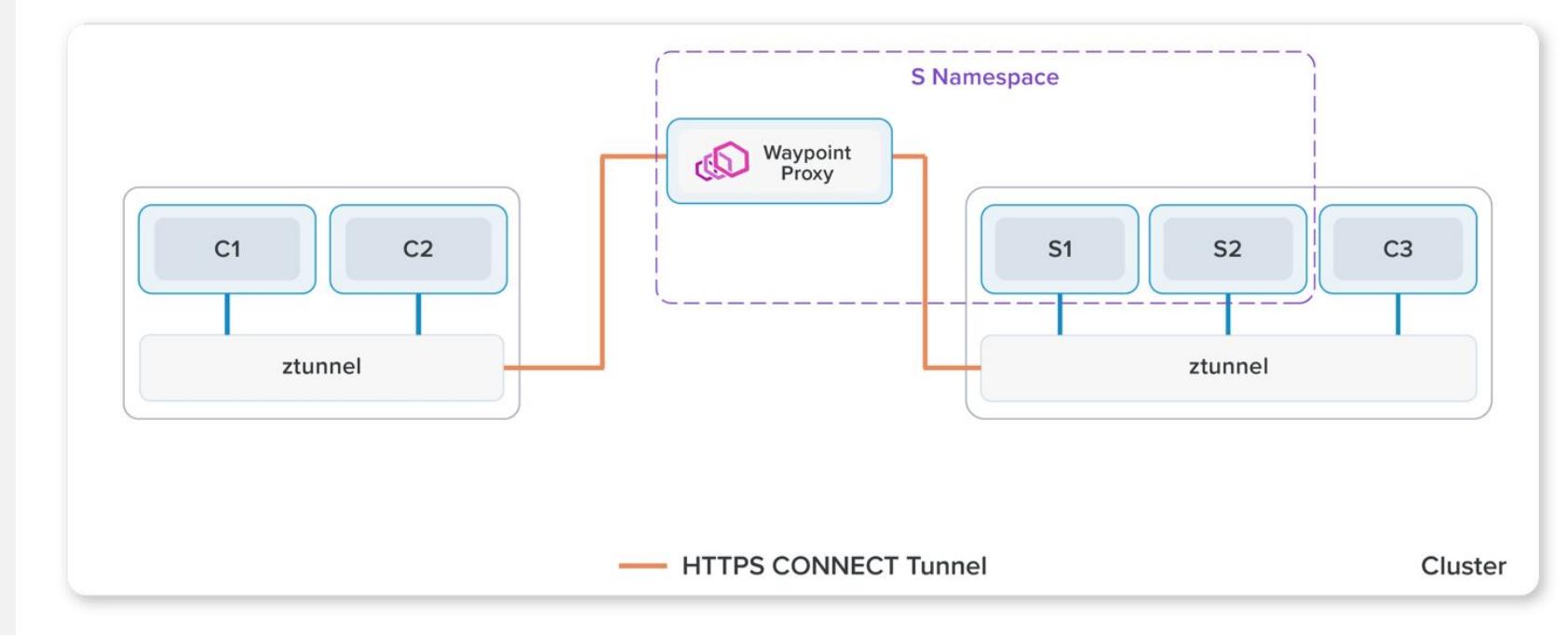


Unterstützt nur L4 features. Die Implementierung ist daher aber viel einfacher und performanter.

Data plane Architekturen - Ambient Mesh



QA|WARE



Wenn L7 features benötigt werden, werden zusätzlich envoy proxies deployed.
Aller Traffic wird dann von den zTunnels durch den Proxy geschleift.

Data plane Architekturen - Ambient Mesh



Vorteile:

- **Skalierbarkeit:** Waypoint Proxy skaliert horizontal unabhängig von Applikationen
- **Zero code changes:** Die Anwendung kann ohne Anpassungen im Code gemeshed werden
- **Resource efficiency:** Teure envoy proxies werden erst dann verwendet, wenn L7 features gebraucht werden. Es werden in Summe nun auch weniger Proxies in Gesamtheit deployed, da ein waypoint proxy pro Namespace verwendet werden kann.
- **Flexibilität:** Kann theoretisch neben dem Sidecar Model deployed werden

Nachteile:

- **Potenziell noch höhere Latenz:** Wenn L7 features gebraucht werden, wird der Traffic durch einen waypoint Proxy geschleift. Der Proxy ist jetzt aber möglicherweise nicht mehr auf demselben Knoten wie die Applikation.

Exkurs: extended berkeley packet filter (eBPF)

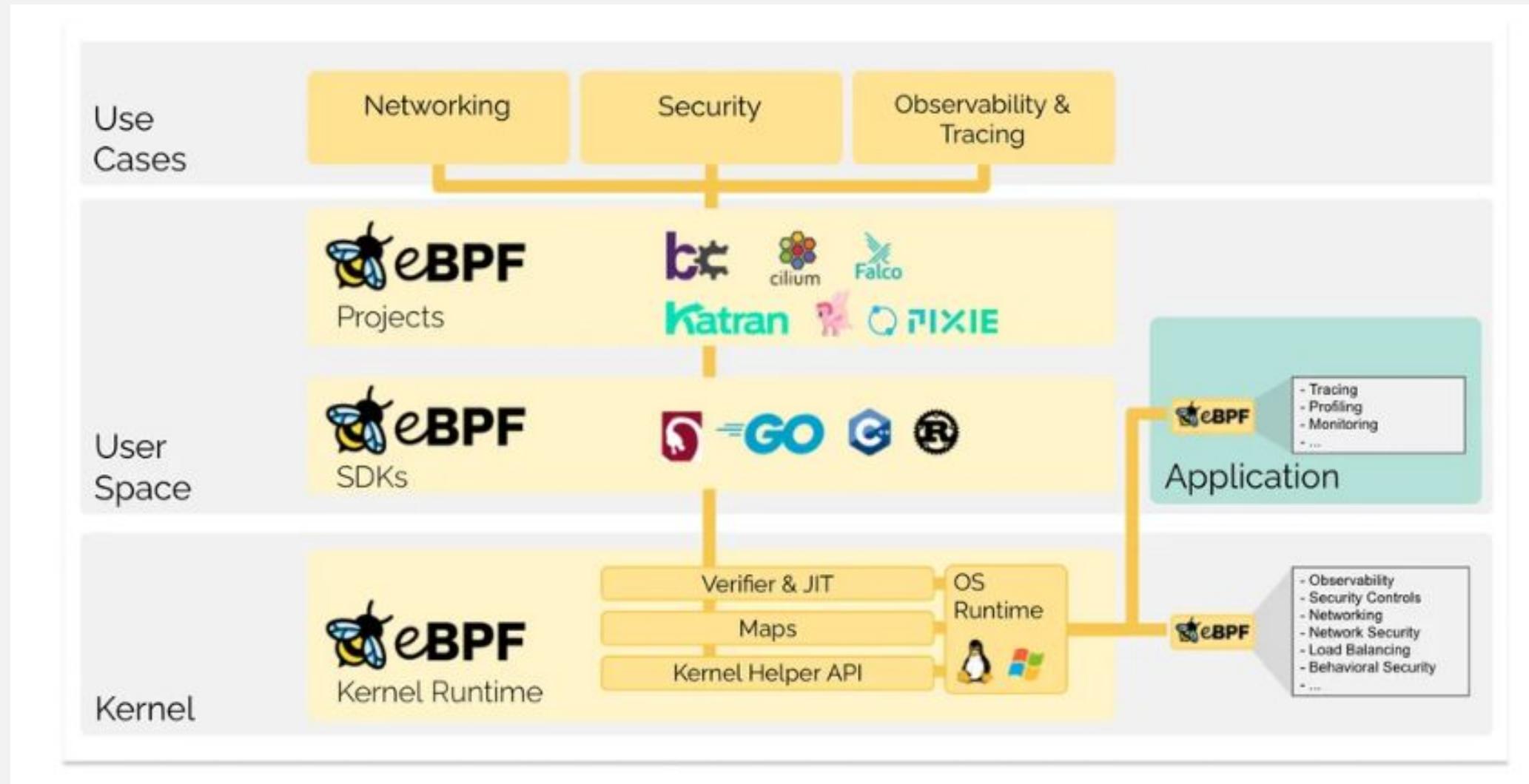


- Historisch aus dem Berkeley Packet Filter (BPF) entstanden.
- BPF war vor allem ein Netzwerk Tool
- eBPF hat den BPF umfassend erweitert, sodass das Akronym im Grunde keinen Sinn mehr macht
- eBPF erlaubt es dem Anwender den **Kernel dynamisch** mit Programmen zu erweitern
 - Diese Programme laufen in einer Sandbox im Kernel
 - Diese Programme werden JIT compiled & müssen einer Verification engine im Kernel genügen
 - Diese Programme laufen event driven und subscriben auf bestehende hooks im Kernel z.B. system calls
 - Falls es keinen bestehenden Hook gibt, kann mit einer kProbe/uProbe das Programm dennoch an relativ frei wählbare Punkte attached werden

Exkurs: extended berkeley packet filter (eBPF)



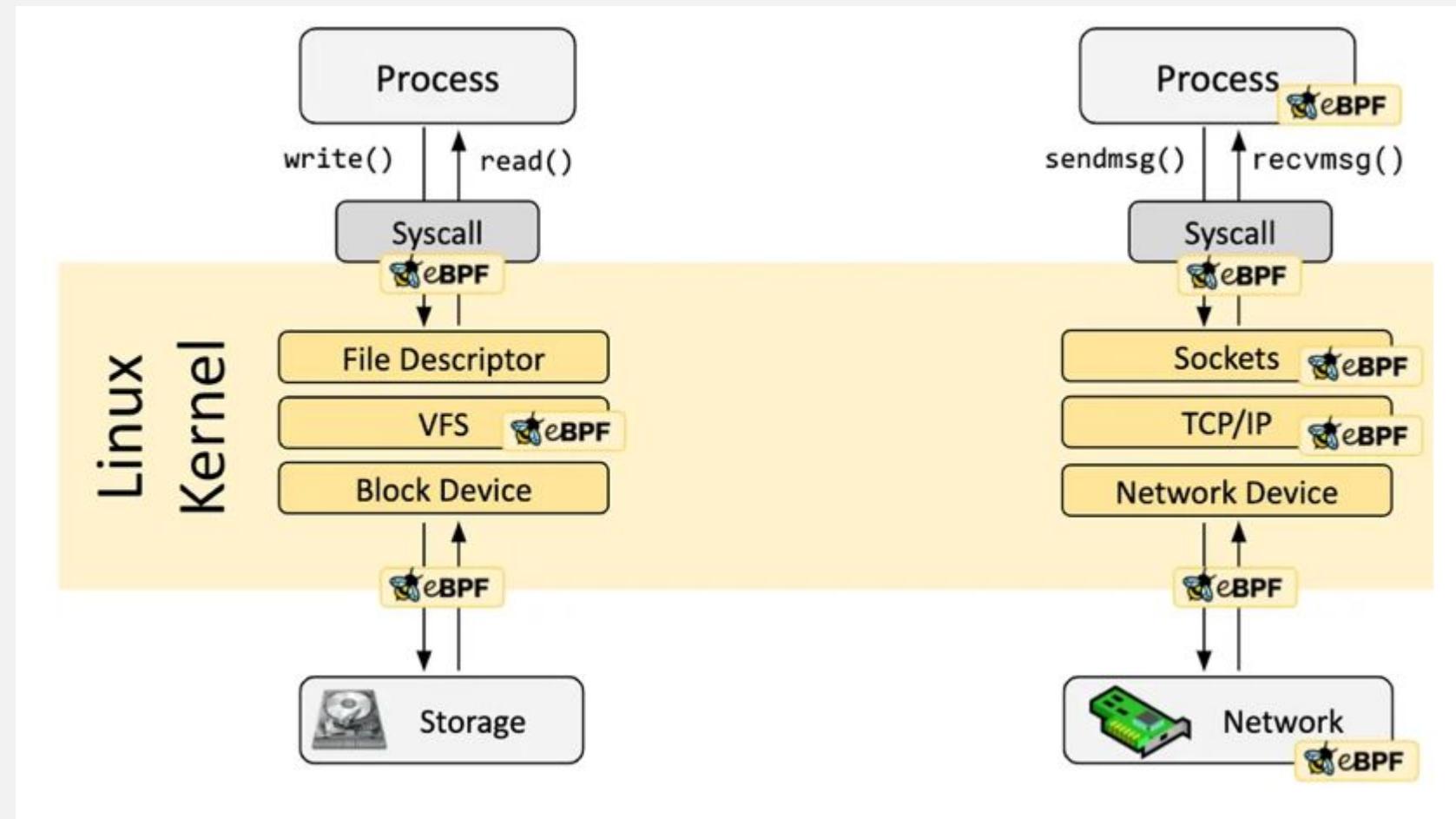
QA|WARE



Exkurs: extended berkeley packet filter (eBPF)



QA|WARE

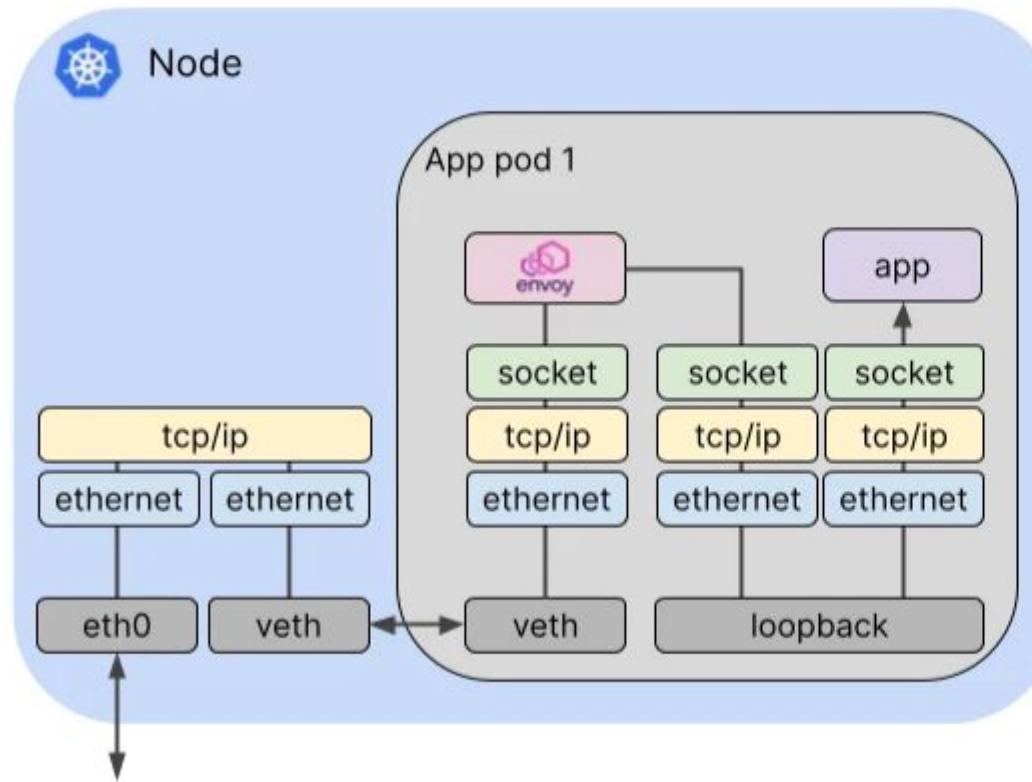


Data plane Architekturen - eBPF based

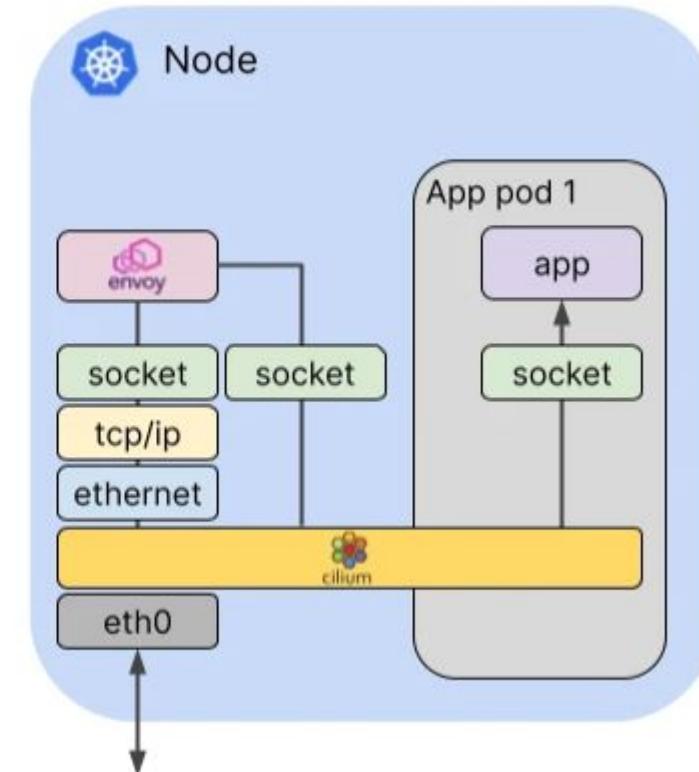


QA|WARE

Service mesh with traditional networking



Sidecarless model, eBPF acceleration



Data plane Architekturen - eBPF based



Vorteile:

- **Zero code changes:** Die Anwendung kann ohne Anpassungen im Code gemeshed werden
- **Resource efficiency:** Die Performance ist dem bisherigen Modell überlegen.
- **Flexibilität:** Es können viele Features in eBPF umgesetzt werden. Nur manche L7 Features erfordern nach wie vor einen L7 Proxy.

Nachteile:

- **Komplexität:** Neue, schwierige Technologie. Wie debugge ich das als Anwender?



QA|WARE

Observability

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



- 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



- Logs sind ein wichtiger Bestandteil jedes IT-Systems.
- In der Cloud müssen wir Logs von vielen verschiedenen Services gleichzeitig betrachten.
- Jedes System hat Logs. Aber immer in einem anderen Format.

- Beispiel: **Quarkus**-Webservice:

```
2023-01-10 20:56:42,122 DEBUG [io.qua.mic.run.bin.ver.VertxHttpServerMetrics]
(vert.x-eventloop-thread-11) requestRouted null HttpRequestMetric [initialPath=/q/metrics,
currentRoutePath=null, templatePath=null, request=io.vertx.core.http.Http1xServerRequest@512b1fd6]
```

- Beispiel: **Nginx**-Webserver:

```
- 2a02:c207:3005:5132::1 - - [10/Jan/2023:00:00:13 +0100] 0.000 repo.saturn.codefoundry.de "POST
/api/v4/jobs/request HTTP/1.1" 957 204 0 "-" "gitlab-runner 15.6.1 (15-6-stable; go1.18.8; linux/amd64)"
```

Logging. Bitte schön strukturiert. Bitte gut überlegt. /1



- Definiert eine 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, GELF etc.
 - 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 😎

Logging. Bitte schön strukturiert. Bitte gut überlegt. /2



- Loggt nicht zu viel, aber jede
 - jede Exception
 - jeden Fehler
 - jede sinnvolle Information
 - ... aber 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: Grundlagen



QA|WARE

Prometheus zieht alle
15s Metriken von allen
konfigurierten Zielen.

Der Transport erfolgt
meist über HTTP: **GET**
/metrics

Hier passiert noch keine
Aggregation. Dafür ist
der Konsument
zuständig.

```
# HELP process_open_fds Number of open file descriptors.
# TYPE process_open_fds gauge
process_open_fds 8
# HELP process_start_time_seconds Start time of the process since unix epoch in seconds.
# TYPE process_start_time_seconds gauge
process_start_time_seconds 1.65566242485e+09
# HELP process_virtual_memory_bytes Virtual memory size in bytes.
# TYPE process_virtual_memory_bytes gauge
process_virtual_memory_bytes 1.782685696e+09

# HELP http_server_requests_seconds
# TYPE http_server_requests_seconds summary
http_server_requests_seconds_count{method="GET",status="200",uri="/tle",} 1.0
http_server_requests_seconds_sum{method="GET",status="200",uri="/tle",} 8.183356641
# HELP http_server_requests_seconds_max
# TYPE http_server_requests_seconds_max gauge
http_server_requests_seconds_max{method="GET",status="200",uri="/tle",} 8.183356641
```

Metric Types



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.

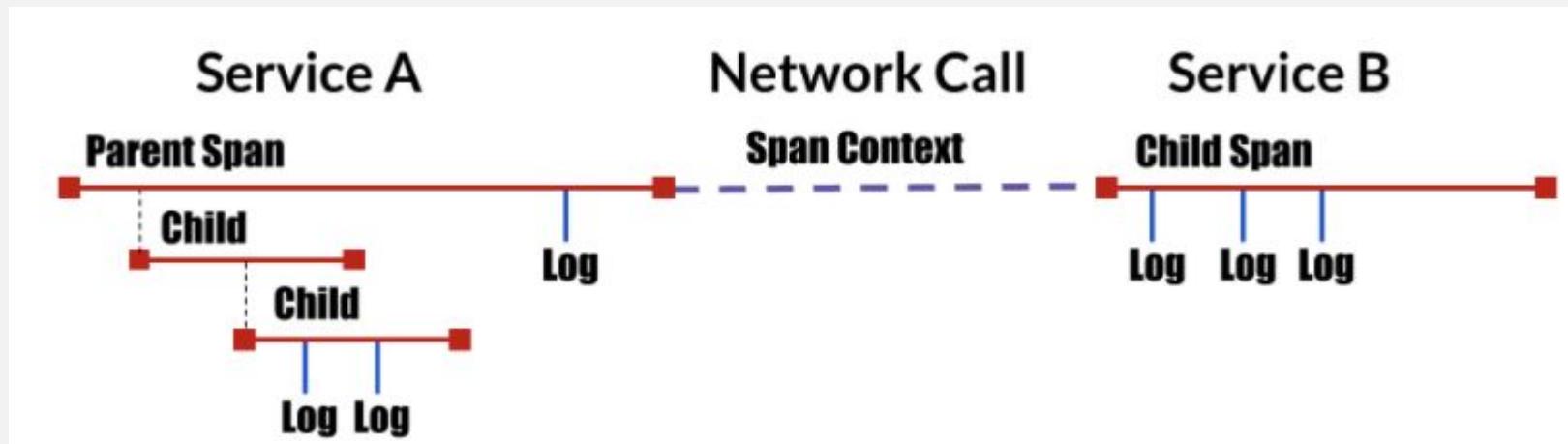


- Distributed Tracing: Technik zum Nachvollziehen von Aufrufen und Abläufen in verteilten Software Systemen.
- Heutige Überlegungen gehen zurück auf das Paper: *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*
- Idee: Jeder Service schickt Informationen vom Aufrufer zum nächsten Service 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/>
- Versuche der Community das zu vereinheitlichen: *OpenTelemetry*

Distributed Tracing: Im Wesentlichen basiert alles auf Google Dapper.



QA|WARE



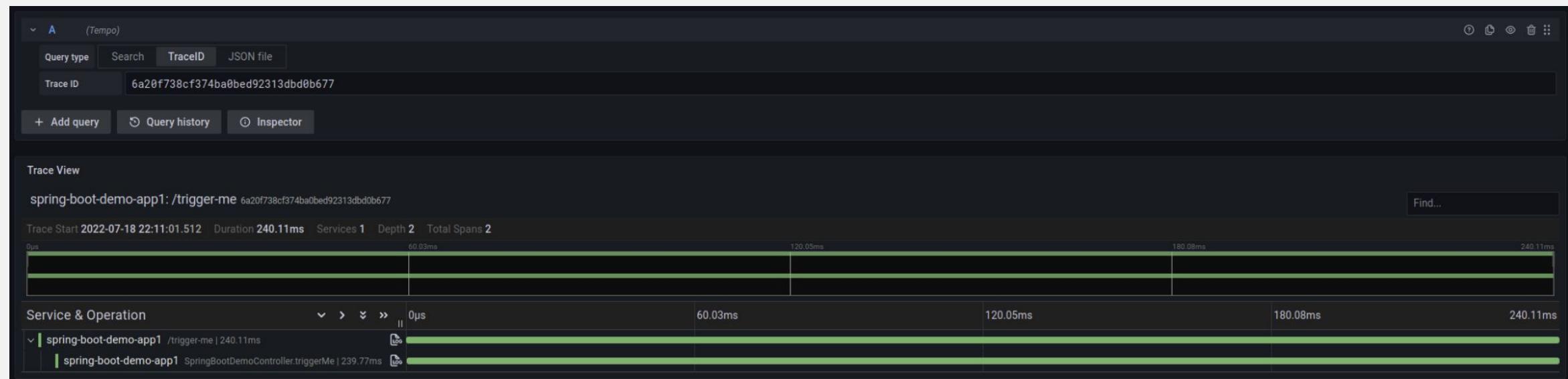
Traces: Grundlagen



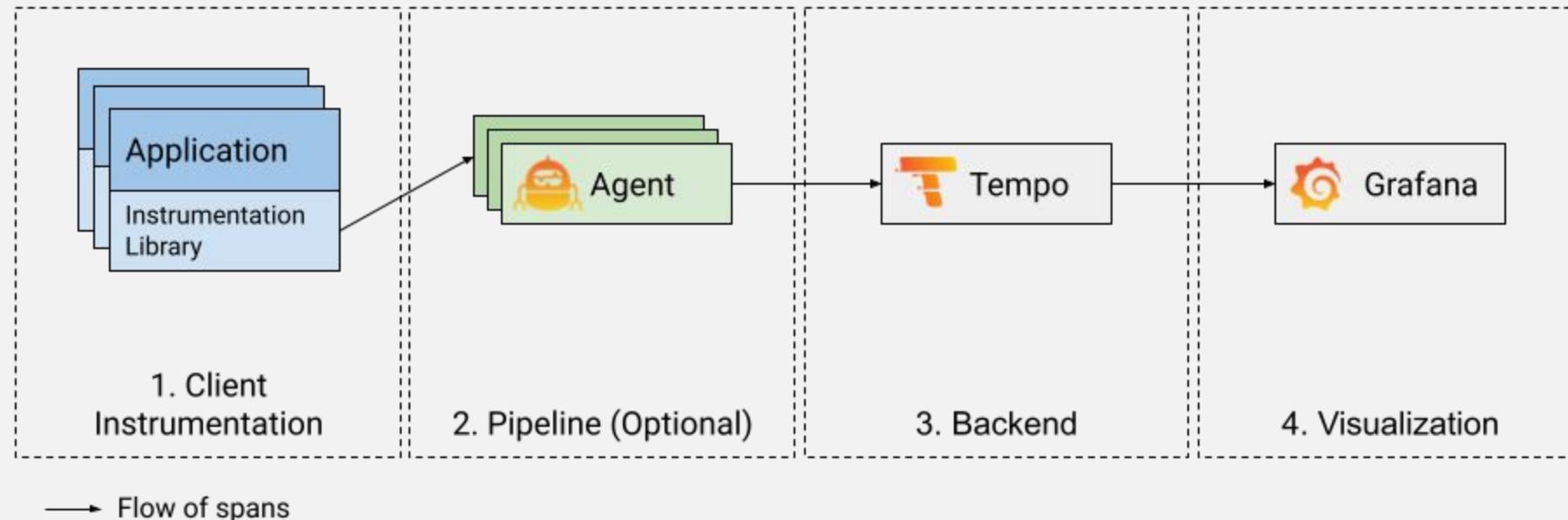
QA|WARE

Traces visualisieren, welche Wege ein Request durch die Microservices genommen hat...

... und wo vielleicht ein Fehler passiert ist!



Architektur

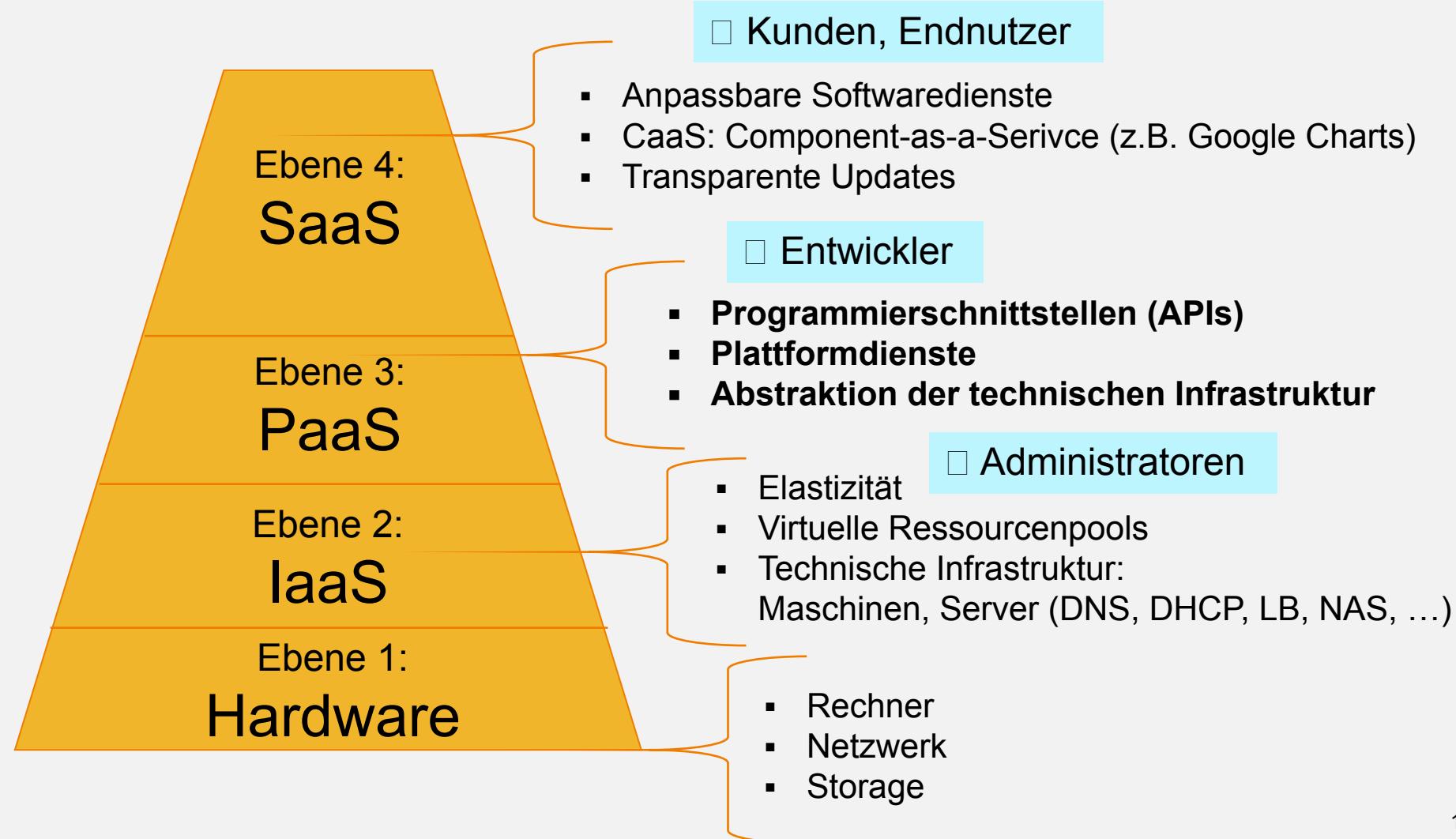




QA|WARE

PaaS

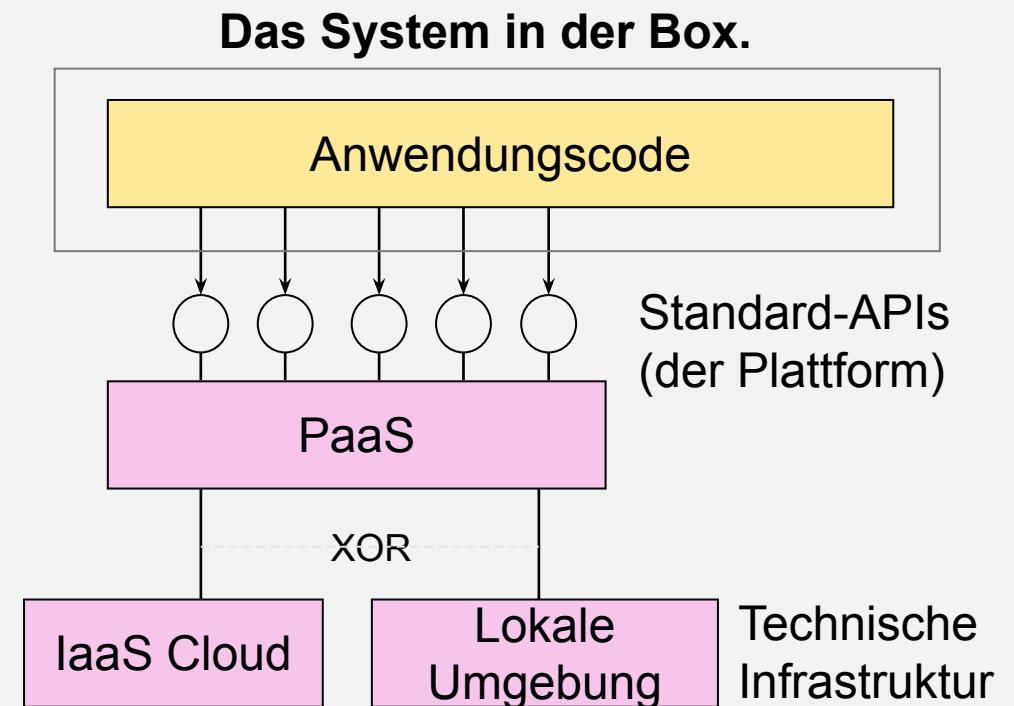
Das Schichtenmodell des Cloud Computing: Vom Blech zur Anwendung.



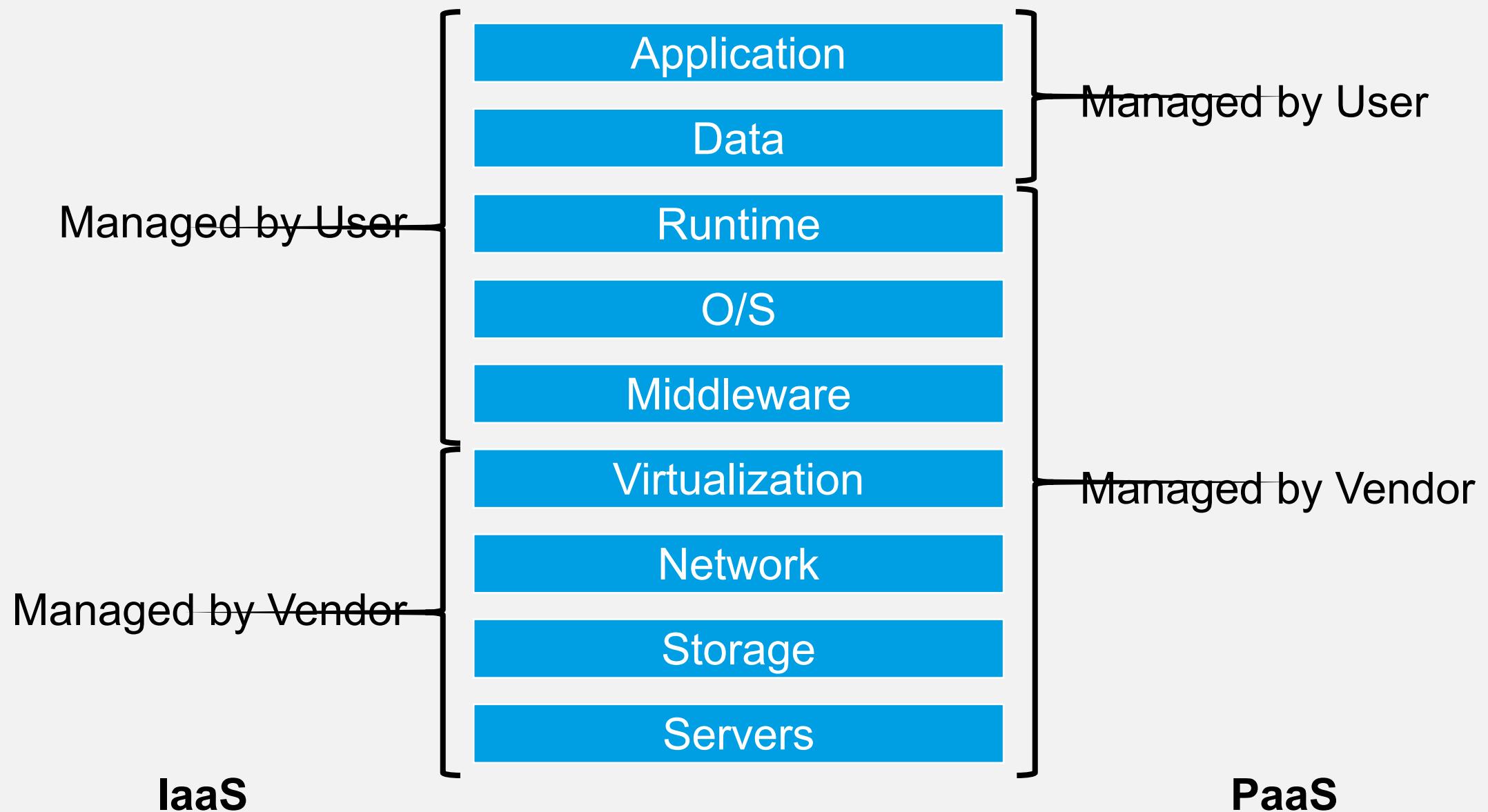
Die Lösung: Plattform-as-a-Service bietet eine ad-hoc Entwicklungs- und Betriebsplattform.

Entwicklungswerzeuge (insb. Plugins für IDEs und Buildsysteme sowie eine lokale Testumgebung) stehen zur Verfügung: „deploy to cloud“.

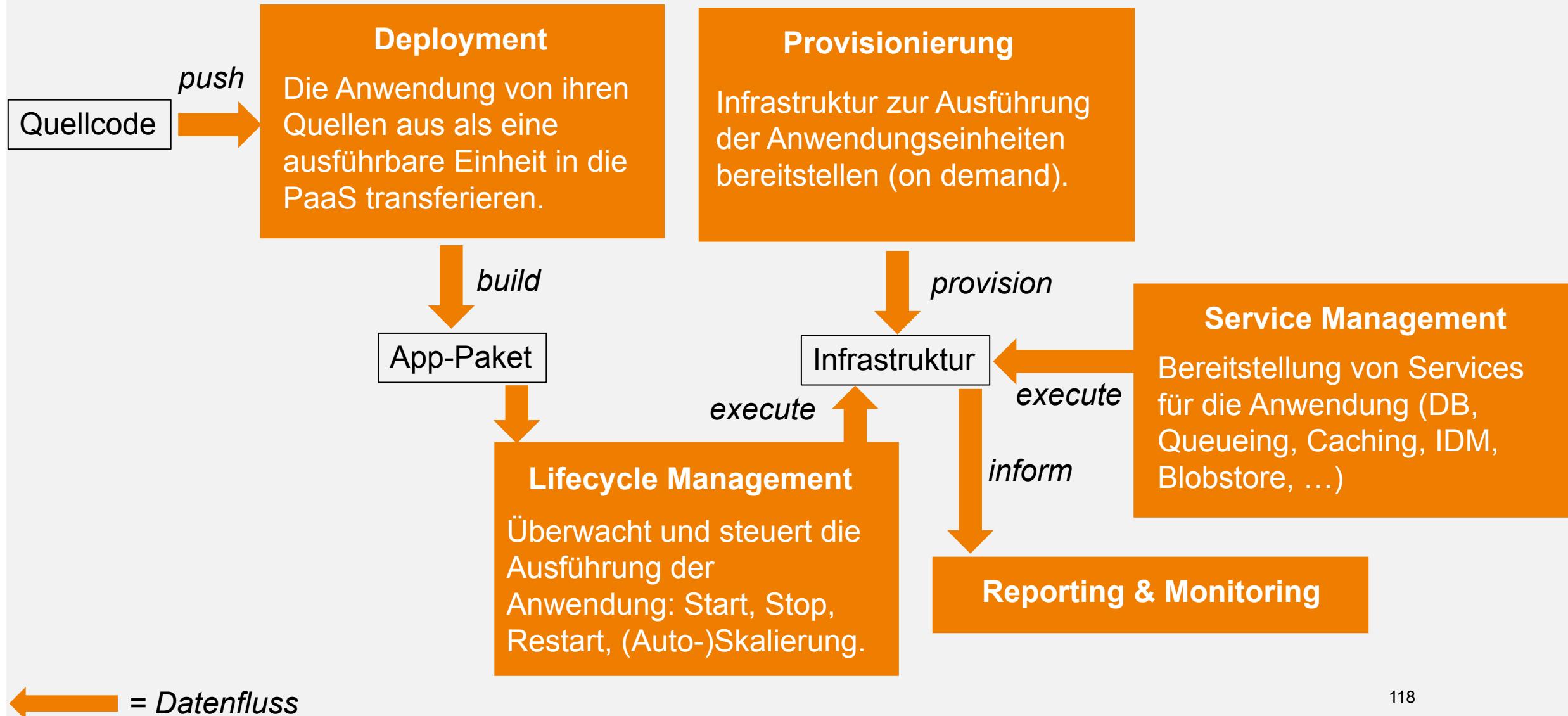
Die Plattform bietet eine Schnittstelle zur Administration und zum Monitoring der Anwendungen.



IaaS vs. PaaS



Die funktionalen Bausteine einer PaaS Cloud.





QA|WARE

Serverless Computing

Serverless Computing - Definition

Serverless computing is a [cloud computing execution model](#) in which the cloud provider dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.^[1] It is a form of [utility computing](#).

Serverless computing still requires servers, hence it's a [misnomer](#).^[1] The name "serverless computing" is used because the server management and capacity planning decisions are completely hidden from the developer or operator. Serverless code can be used in conjunction with code deployed in traditional styles, such as [microservices](#). Alternatively, applications can be written to be purely serverless and use no provisioned services at all.^[2]

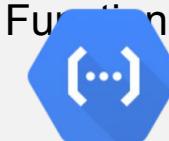
wikipedia.org

Serverless computing allows you to build and run applications and services without thinking about servers. Serverless applications don't require you to provision, scale, and manage any servers. You can build them for [nearly any type of application](#) or backend service, and everything required to run and scale your application with high availability is handled for you.

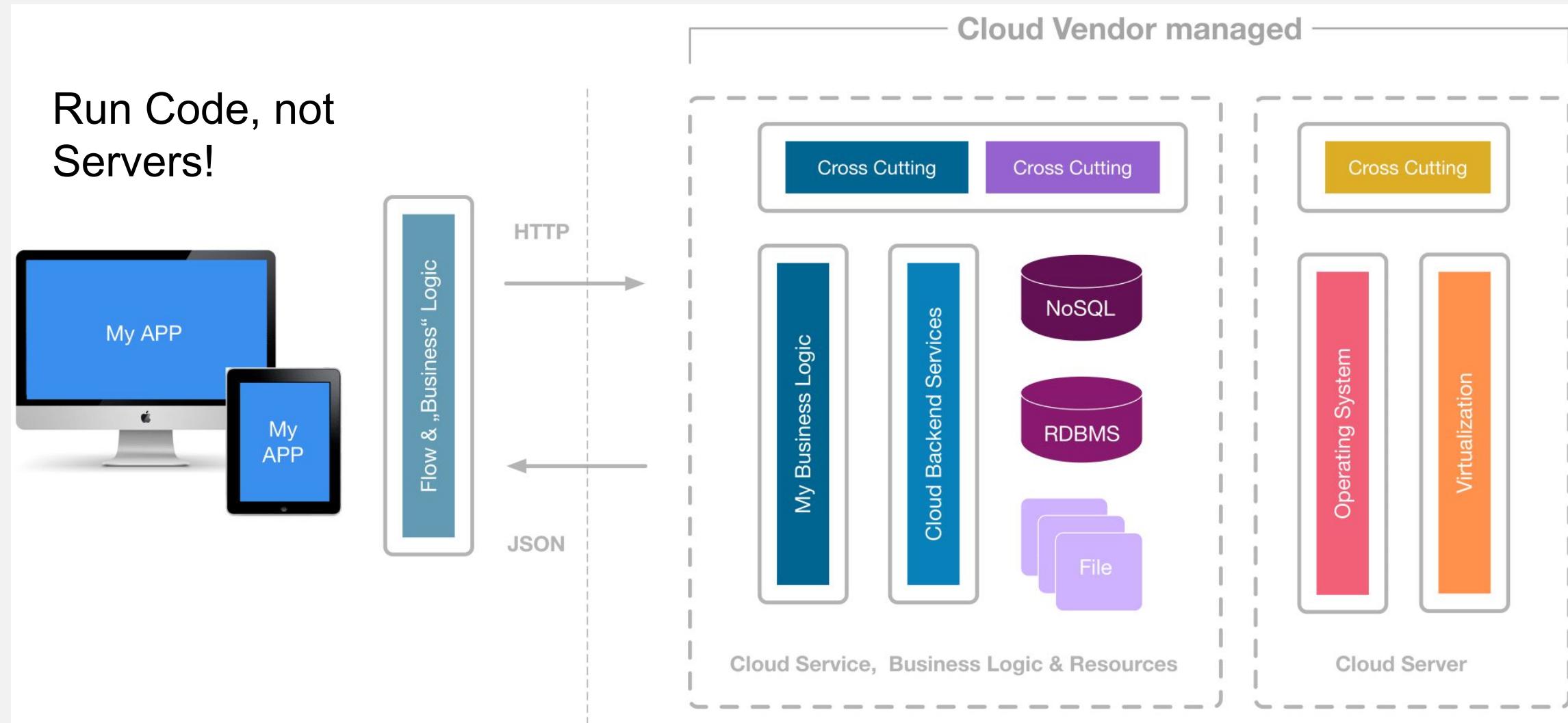
amazon.com

Serverless Computing – Überblick & Vergleich mit PaaS

- Serverless Computing wird häufig auch als Function as a Service (**FaaS**) bezeichnet
- FaaS ist eine Spezialform von PaaS
- Deployment und Betrieb wird vom Cloud Betreiber durchgeführt. Hier ähnelt eine FaaS Plattform PaaS
- Ein Unterschied zu ‚klassischen‘ PaaS Platformen:
 - Der Betreiber garantiert nicht, dass eine einzelne Funktion ständig deployed ist. Häufig wird diese bei Bedarf erst geladen / deployed.
 - Es entfällt die feingranulare Administration einer PaaS.
 - Entwickler müssen sich nicht um die Laufzeitumgebung (bau des Containers, o.ä.) kümmern
- Der primäre Architekturstil von FaaS ist Ereignisgetriebene Architektur (Event-driven Architecture / EDA)
- Die größten Anbieter sind Google mit App Engine, Amazon mit AWS Lambda, Microsoft mit Azure Functions und Google App Engine :



Serverless Anwendungsarchitektur



Serverless Computing – Vor- und Nachteile

Vorteile:

- Kosten: Da einzelne Funktionen nur bei Benutzung deployed werden ist dies oft kosteneffektiver, als Server ständig zu betreiben
- Produktivität: Einzelne Funktionen können sehr schnell geschrieben, deployed und aktualisiert werden.
- Performance: Einzelne Funktionen können sehr feingranular skaliert werden.

Nachteile:

- Performance: Da einzelne Funktionen evtl. erst bei Bedarf geladen werden, können starke Schwankungen bei der Ausführung auftreten.
- Debugging: Außer Fehlermeldungen und Log-Output, hat man weniger Möglichkeiten zur Diagnose. Dies erschwert das Debugging / Profiling der Anwendung.



QA|WARE

CI / CD

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



QA|WARE

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



“You’re doing continuous delivery when:

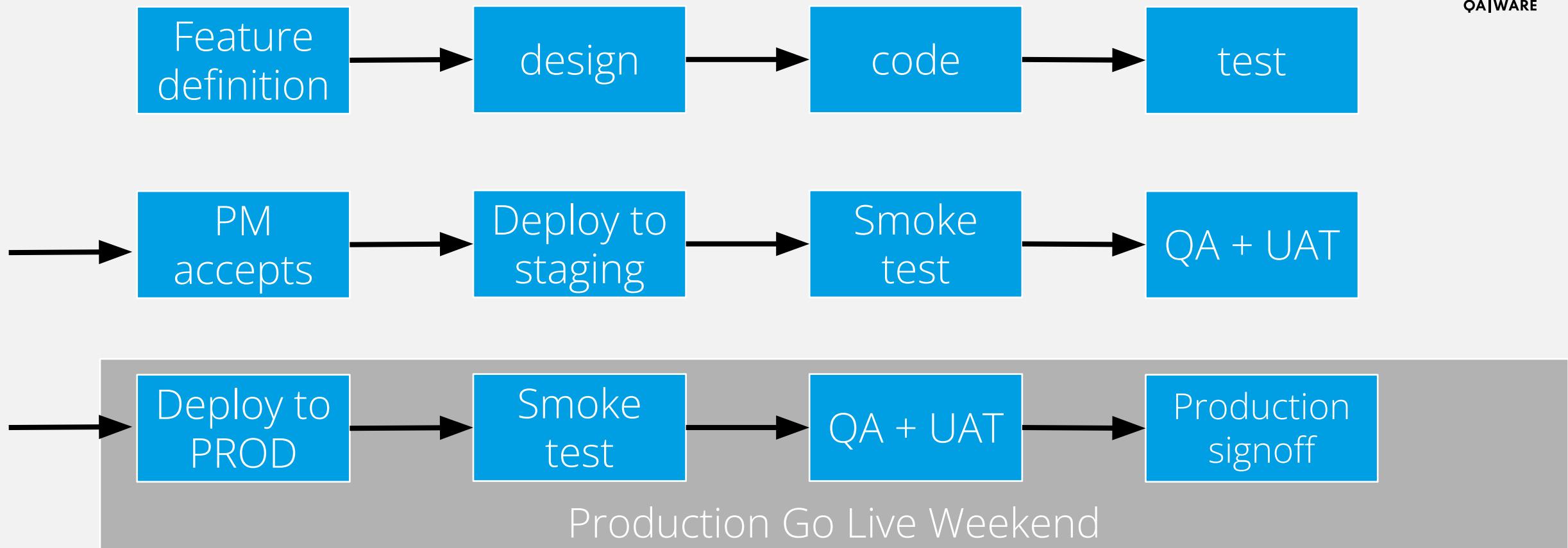
- Your software is deployable throughout its lifecycle
- Your team prioritizes keeping the software deployable over working on new features
- Anybody can get fast, automated feedback on the production readiness of their systems any time somebody makes a change to them
- You can perform push-button deployments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

nach M. Fowler / Continuous Delivery working group at ThoughtWorks

Beispiel: Pipeline ohne Continuous Delivery



QA|WARE



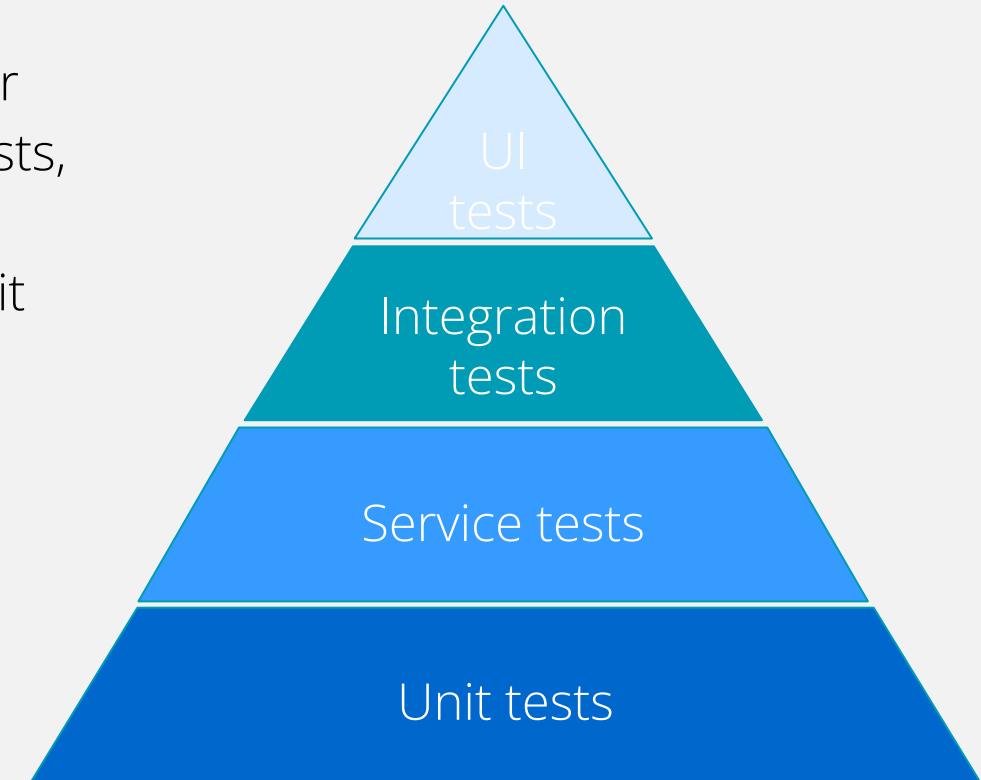
<https://www.scaledagileframework.com/continuous-delivery-pipeline/>

Beispiel-Aufbau einer Test-Pyramide für sofortiges Feedback bei Fehlern



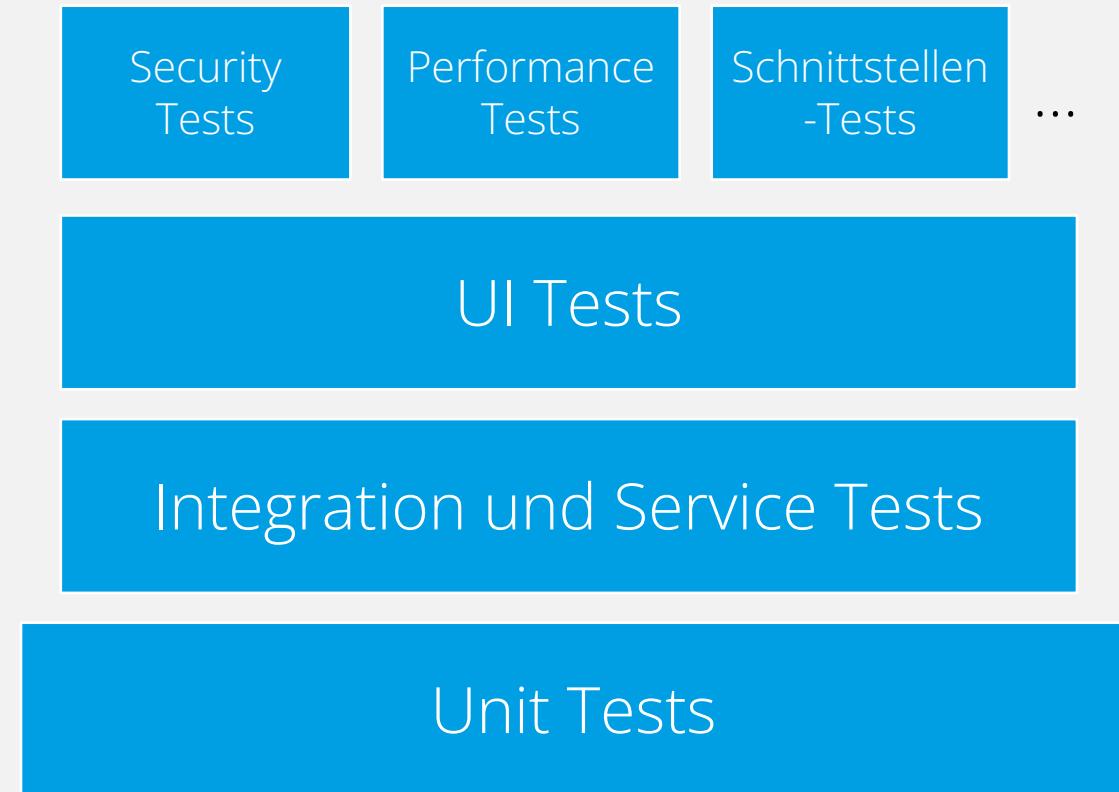
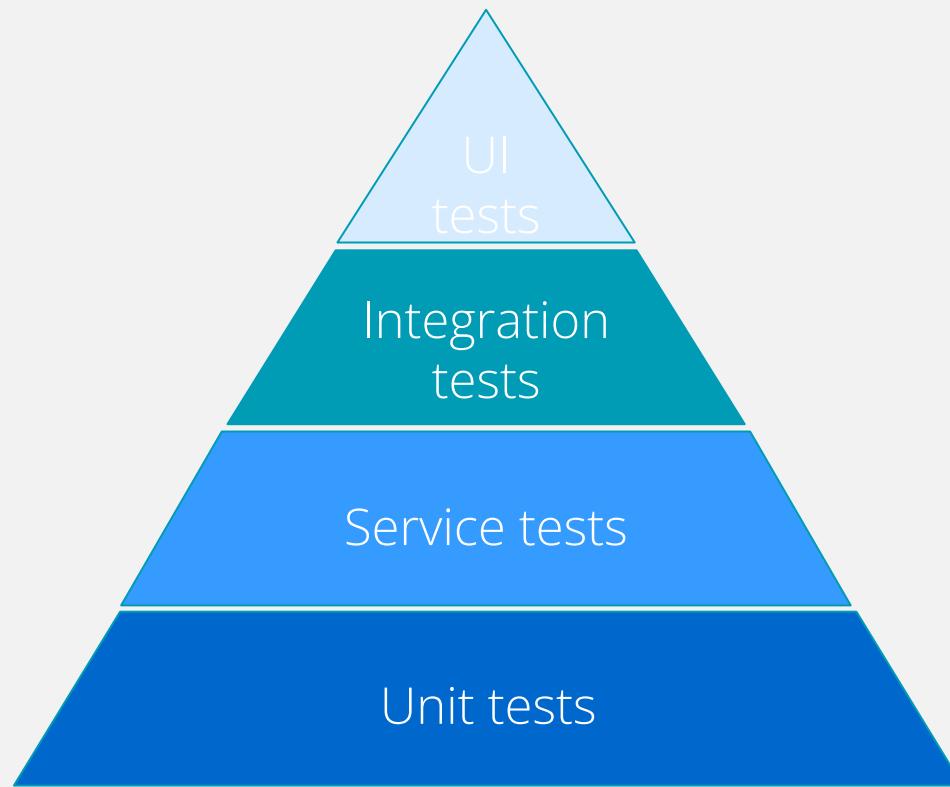
QA|WARE

- 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 heute gut automatisierbar. Anstelle der Testpyramide kann die Verteilung auch so aussehen:



Beispiel einer Continuous Delivery Pipeline

VCS

GitHub



Continuous Delivery - Cluster

Deploy & Test

Test - Cluster

Dev-1

Dev-2

Staging

Build

Push

Analyze

Docker Registry

JFrog Artifactory

Orchestrator

Scheduler

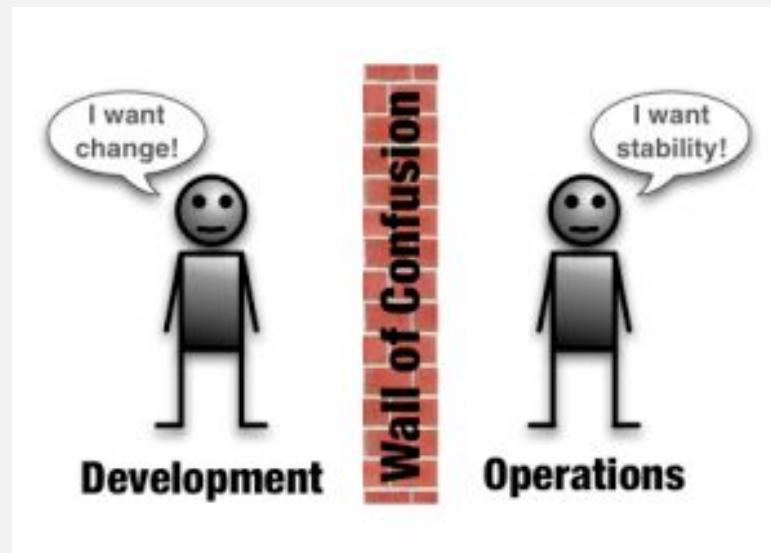
IaaS

Code push

Fast Feedback

Was ist eigentlich DevOps?

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



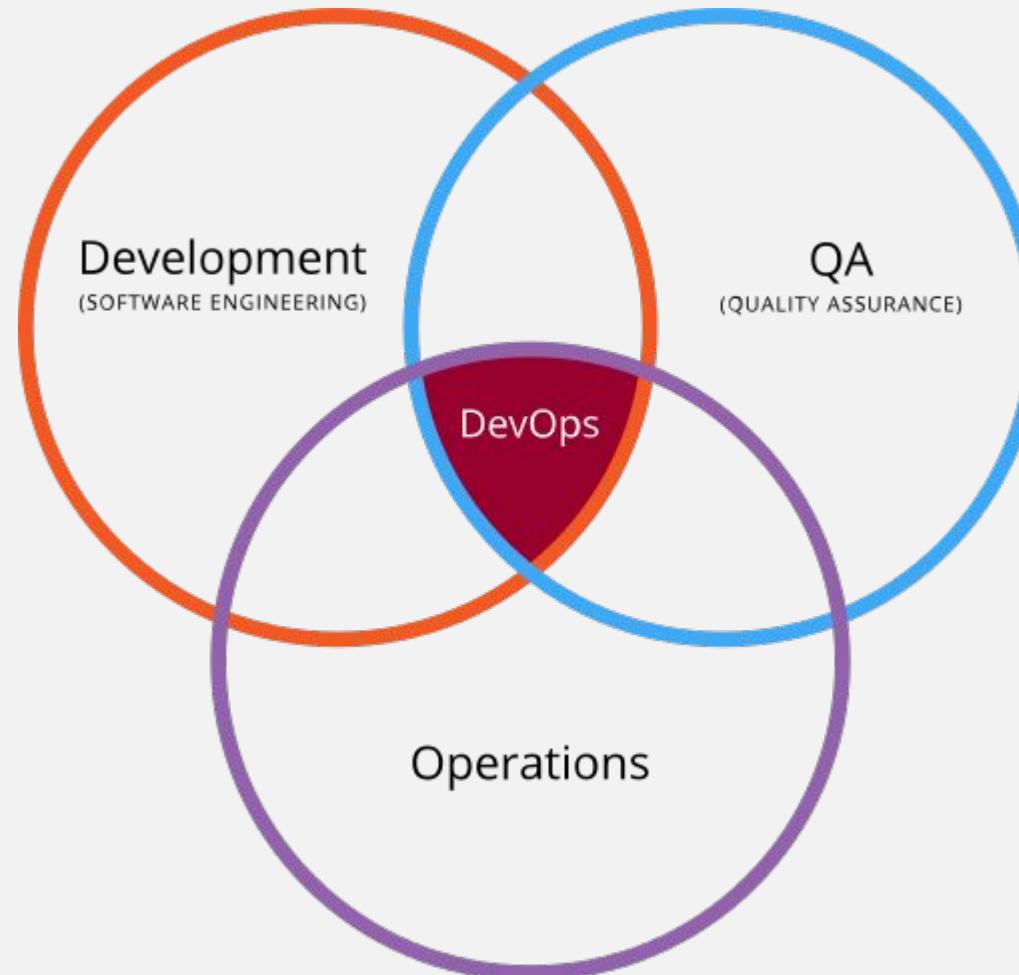
MVP + Feature-Strom
Pro Feature:

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

DevOps verbindet DEVelopment, OPerations und QA Assurance



QA|WARE



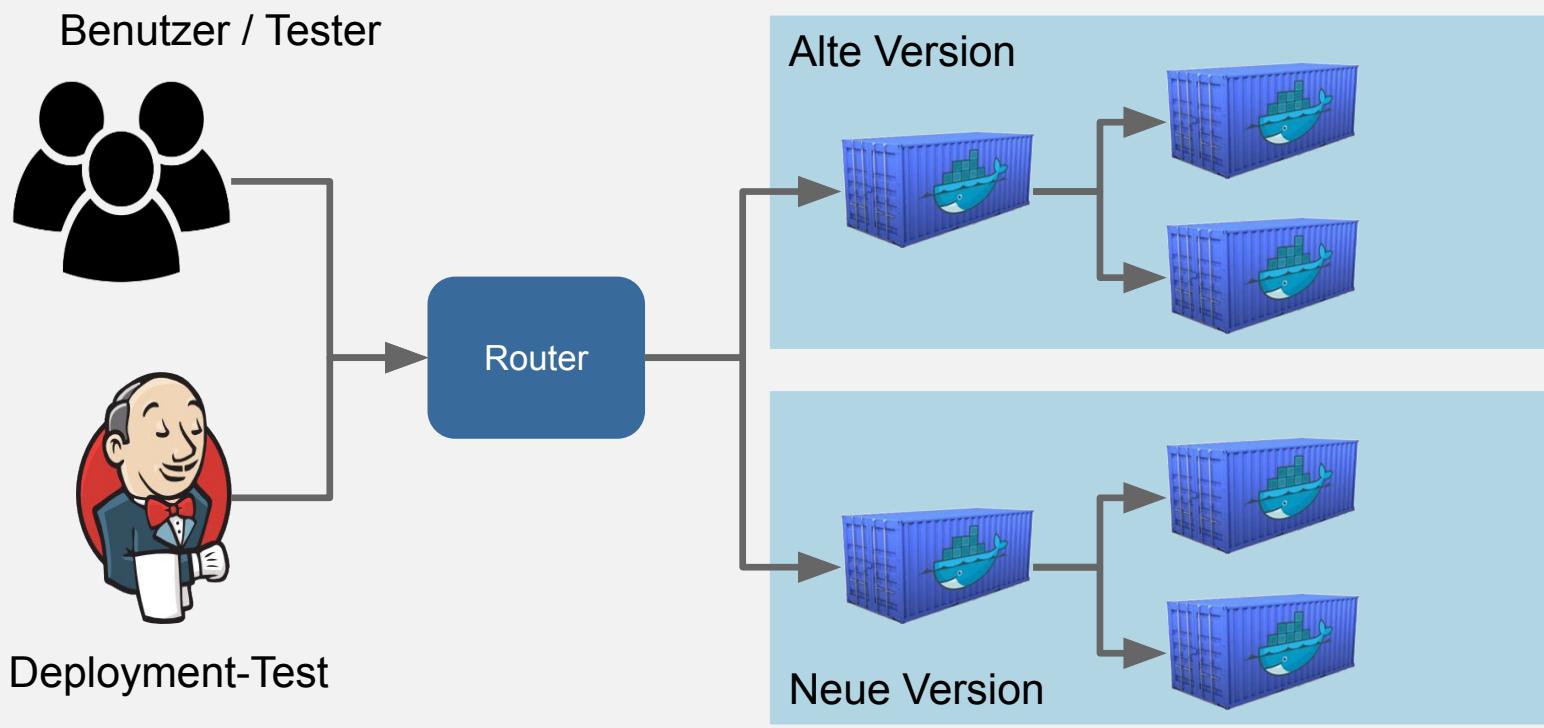
„Venn diagram showing DevOps“ by Rajiv.Pant/Wylve; Creative Commons 3.0

Alles, was die CD-Umgebung mit Leben befüllt, kommt VCS:

- Build-as-Code
 - Maven, Gradle, ...
 - Beschreibt wie die Anwendung gebaut wird
- Test-as-Code
 - Unit-, Component-, Integration-, API-, UI-, Performance-Tests
 - Beschreibt wie das Projekt getestet wird
- Infrastructure-as-Code
 - Docker, Terraform, Vagrant, Ansible, Marathon-Deployments
 - Beschreibt, wie die Laufzeitumgebungen aufgebaut werden
- Pipeline-as-Code
 - Build-Pipeline per Jenkinsfile
 - Buildklammer: Beschreibt alle Schritte bis zur lauffähigen Installation

Mögliche Strategie: Canary Releases

- 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



QA|WARE

Big Data

Big Data – was ist das überhaupt?

Charakteristische Eigenschaften:

- Die Größe des Datensatzes
- Die Komplexität des Datensatzes
- Die Technologien, die Verwendet werden, um den Datensatz zu verarbeiten

“Big data is a term describing the storage and analysis of large and or complex data sets using a series of techniques including, but not limited to: NoSQL, MapReduce and machine learning”

Quelle: . S. Ward und A. Barker. Undefined by data: a survey of big data definitions. arXiv preprint arXiv:1309.5821, 2013.

Große Datenmengen können effizient nur von parallelen Algorithmen verarbeitet werden.

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

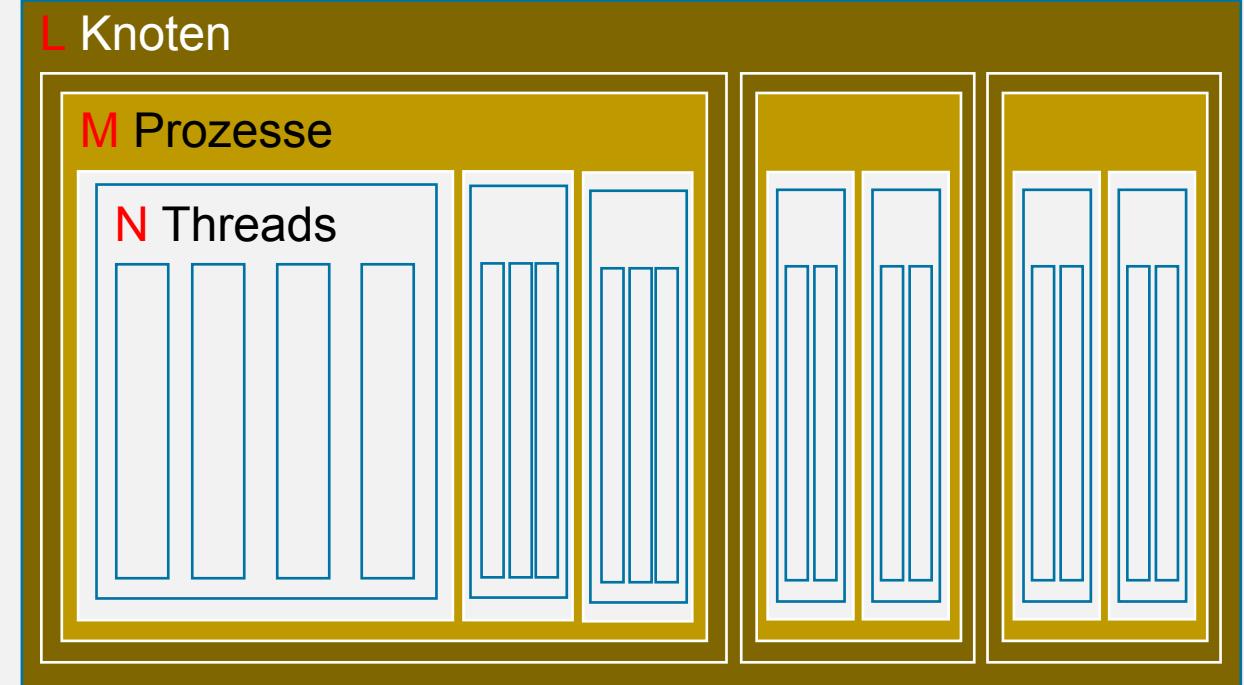
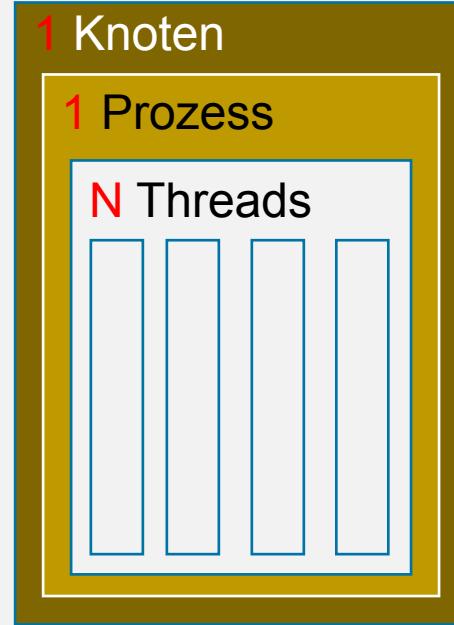
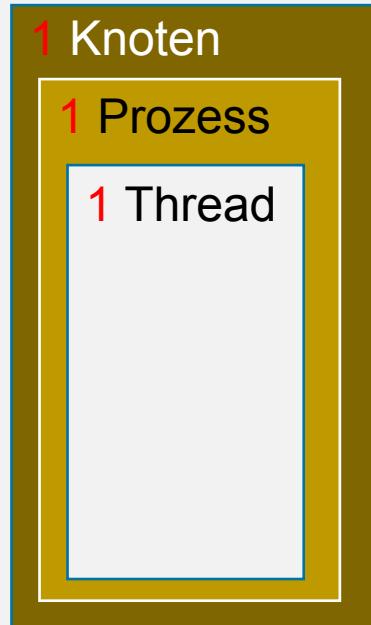
- Funktioniert gut: Quicksort. Aufwand: $O(n \log n) \square n \times O(\log n)$

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

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

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

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



Keine
Parallelität



Parallelität im Kleinen

Vorteile im Vergleich:

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



Parallelität im Großen

Vorteile im Vergleich:

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

Big Data erfordert Parallelität im Großen.

Die vier Paradigmen der Parallelität im Großen:



Folgt aus Datenmenge
im Vergleich zur
Programmgröße

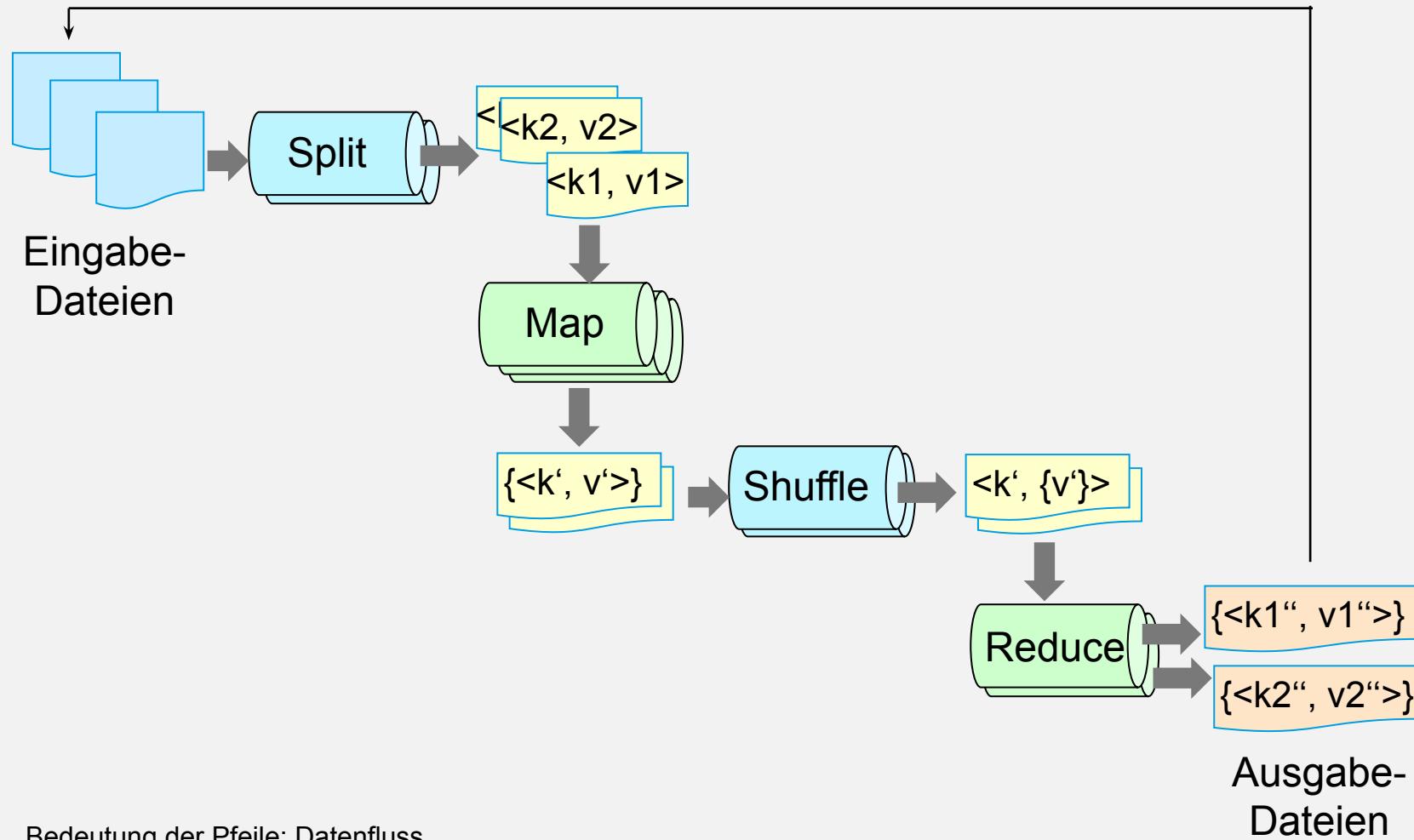
Das Grundprinzip von paralleler
Verarbeitung.

Folgt aus Praxisanforderung:
Viele Knoten bedeutet
viele Ausfallmöglichkeiten

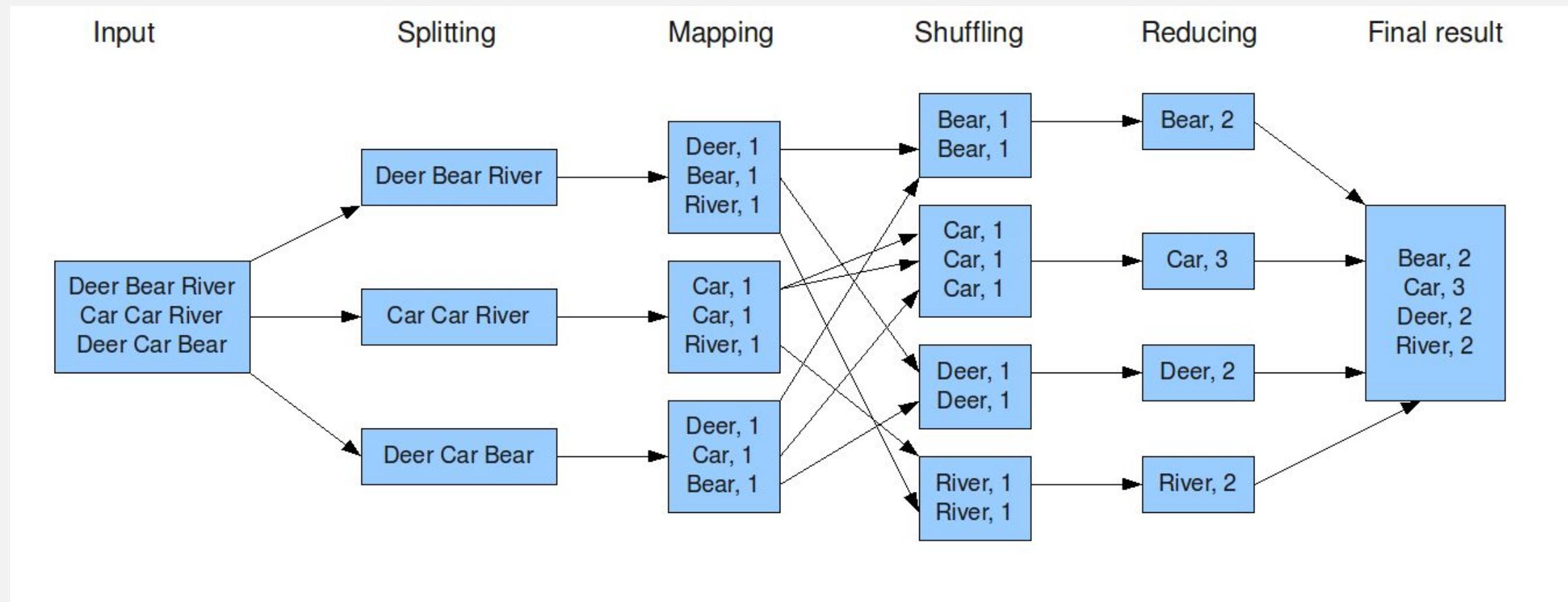
1. Die Logik folgt den Daten.
2. Falls Datentransfer notwendig, dann so schnell wie möglich:
In-Memory vor lokaler Festplatte vor Remote-Transfer.
3. Parallelisierung über *Tasks* (seiteneffektfreie Funktionen) und *Jobs* (Ausführungsvorschrift für Tasks) sowie entsprechend partitionierter Daten (*Shards*).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

Folgt aus potenziell
großer Datenmenge und
Verarbeitungs-geschwin
digkeit

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

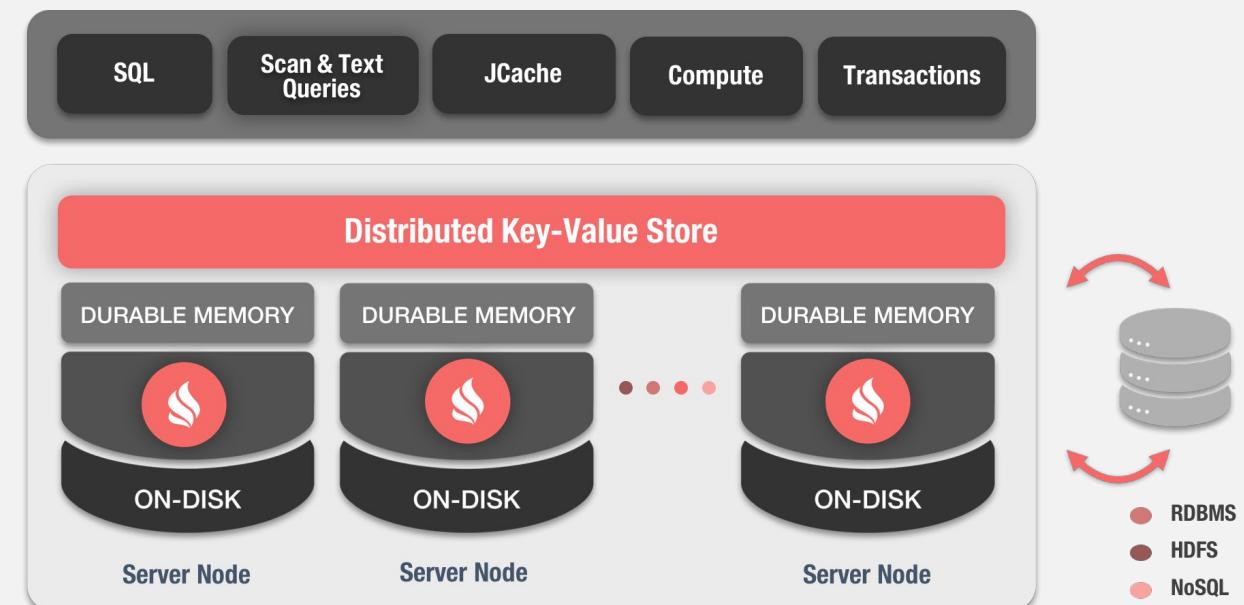


Übersicht über alle Phasen



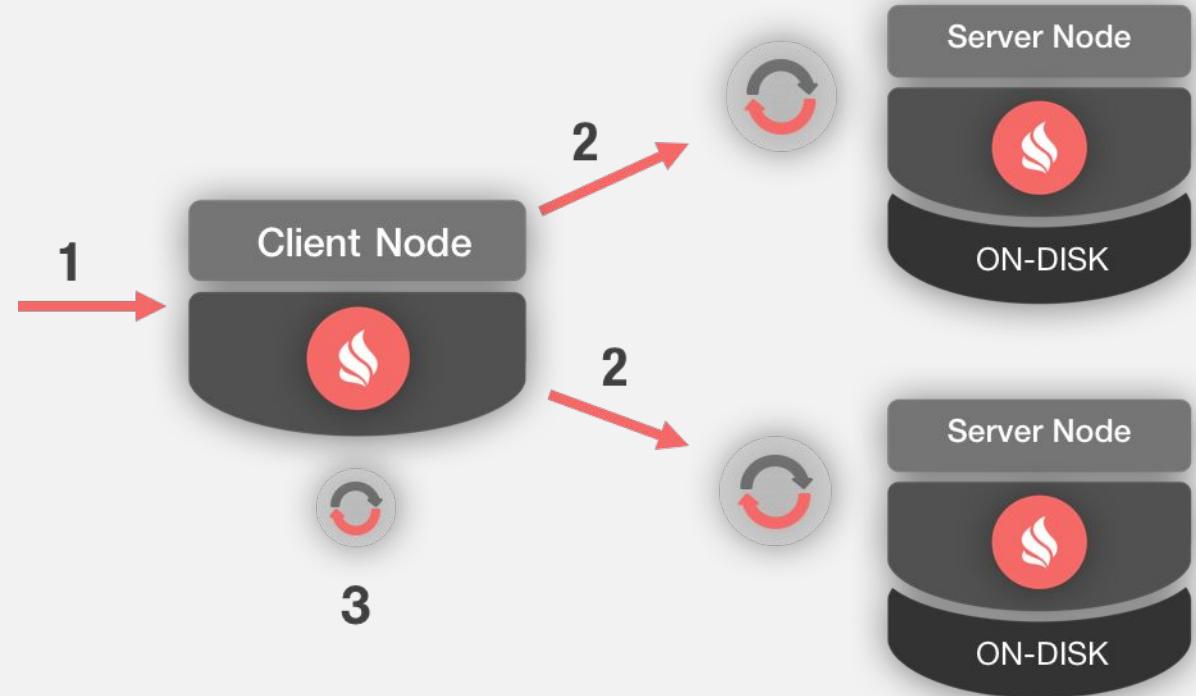
Ignite Data Grid

- In-Memory Key Value Store
- Implementiert die JCache-Spezifikation [**get()**, **put()**, **containsKey()**]
- Native Persistenz (=> Filesystem) vorhanden
- Eigene Storage-Provider möglich (z.B. SQL, MongoDB, ...)



Ignite Compute

- Verteilte Verarbeitung von Daten
- Code wird zu den Daten gebracht (Performance!)
- Ähnliche Projekte:
 - Hadoop MapReduce
 - Apache Spark



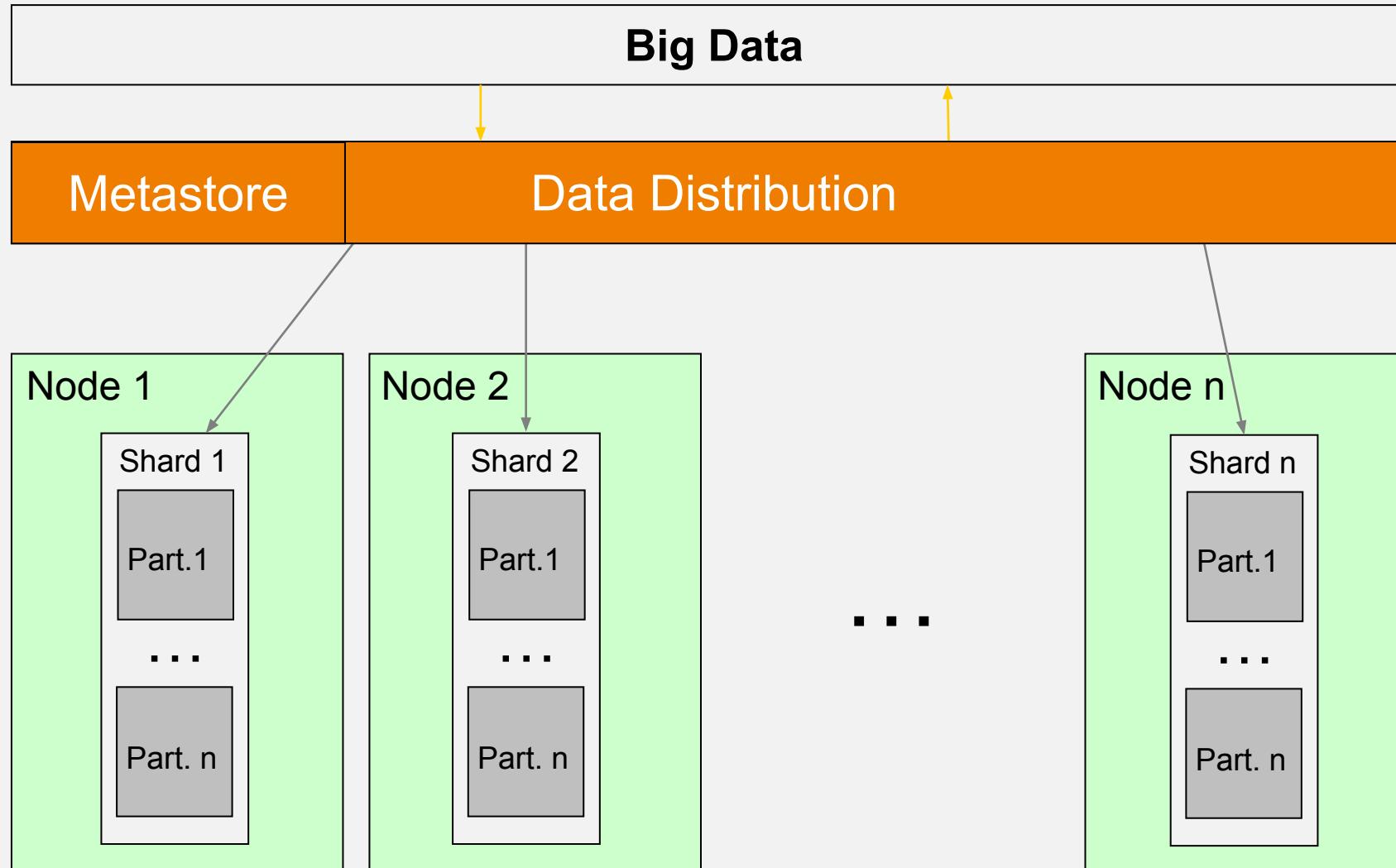
1. Initial Request
2. Co-located processing with data
3. Reduce multiple results in one

Apache Ignite Streaming

- Manchmal ist der Datensatz so groß, dass er nicht im Ignite-Cluster Platz hat.
- Die Lösung: Streaming und Verarbeitung on the Fly!
 - “With Apache Ignite you can load and stream large finite — or never-ending — volumes of data in a scalable and fault-tolerant way into the cluster.”
- Beispiele:
 - Data Loading
 - Real-Time Data Streaming

Quelle: <https://ignite.apache.org/features/streaming.html>

Sharding and Partitioning: Verteilung und Stückelung von großen Datenmengen.



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



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

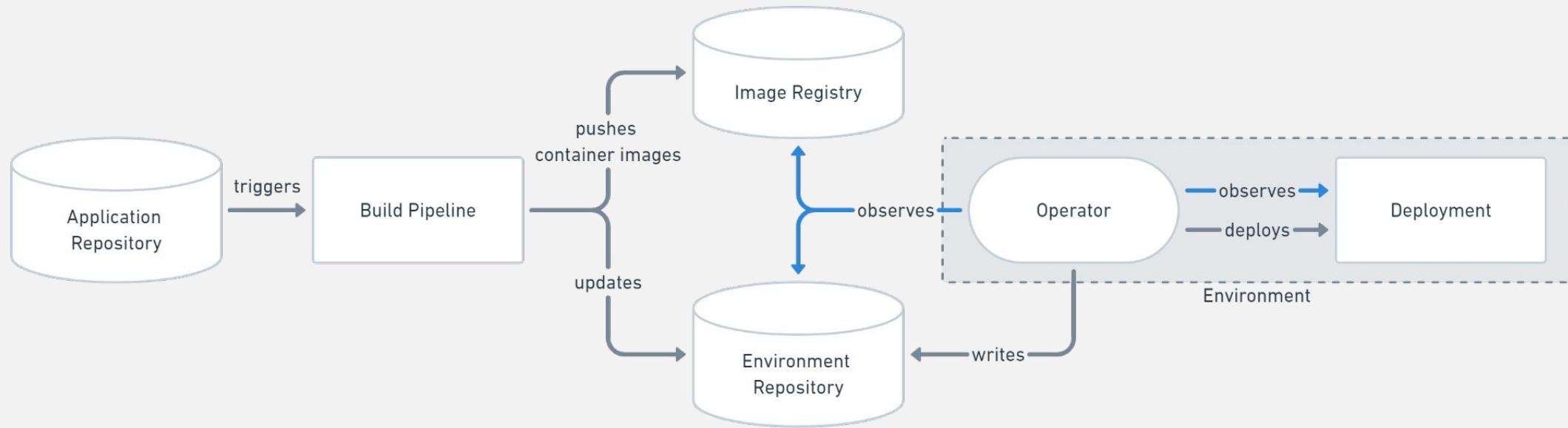


Flux

Pull-based deployments mit GitOps



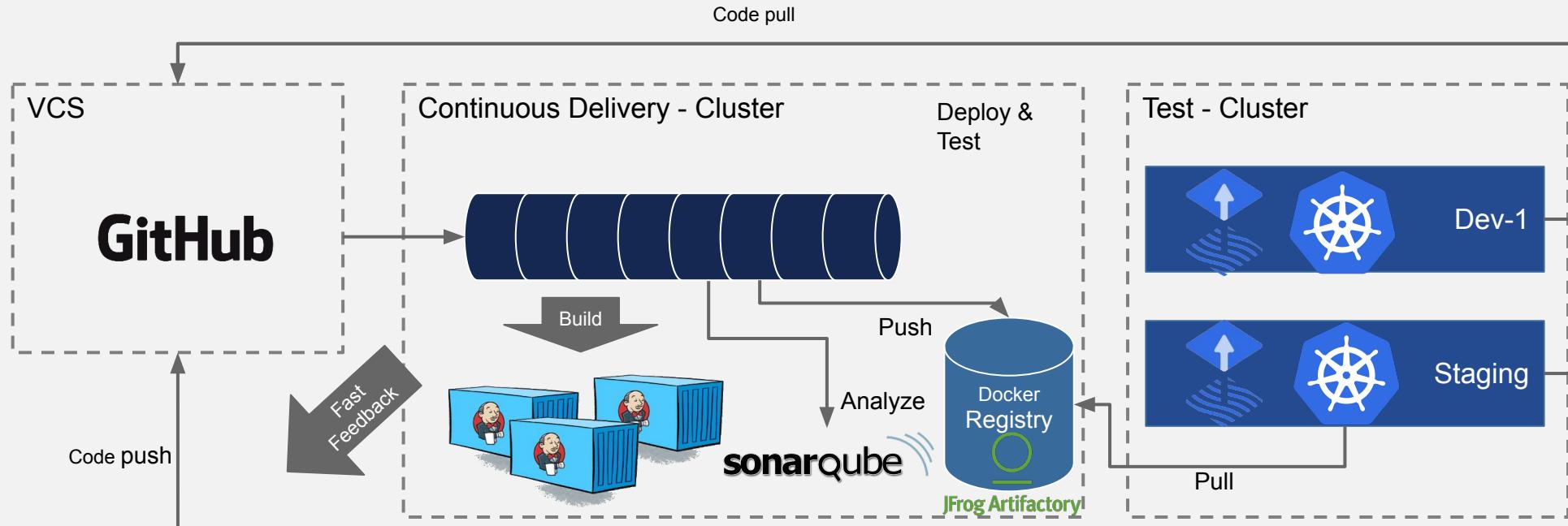
QA|WARE



GitOps mit Flux



QA|WARE



Unterschiede in der Architektur

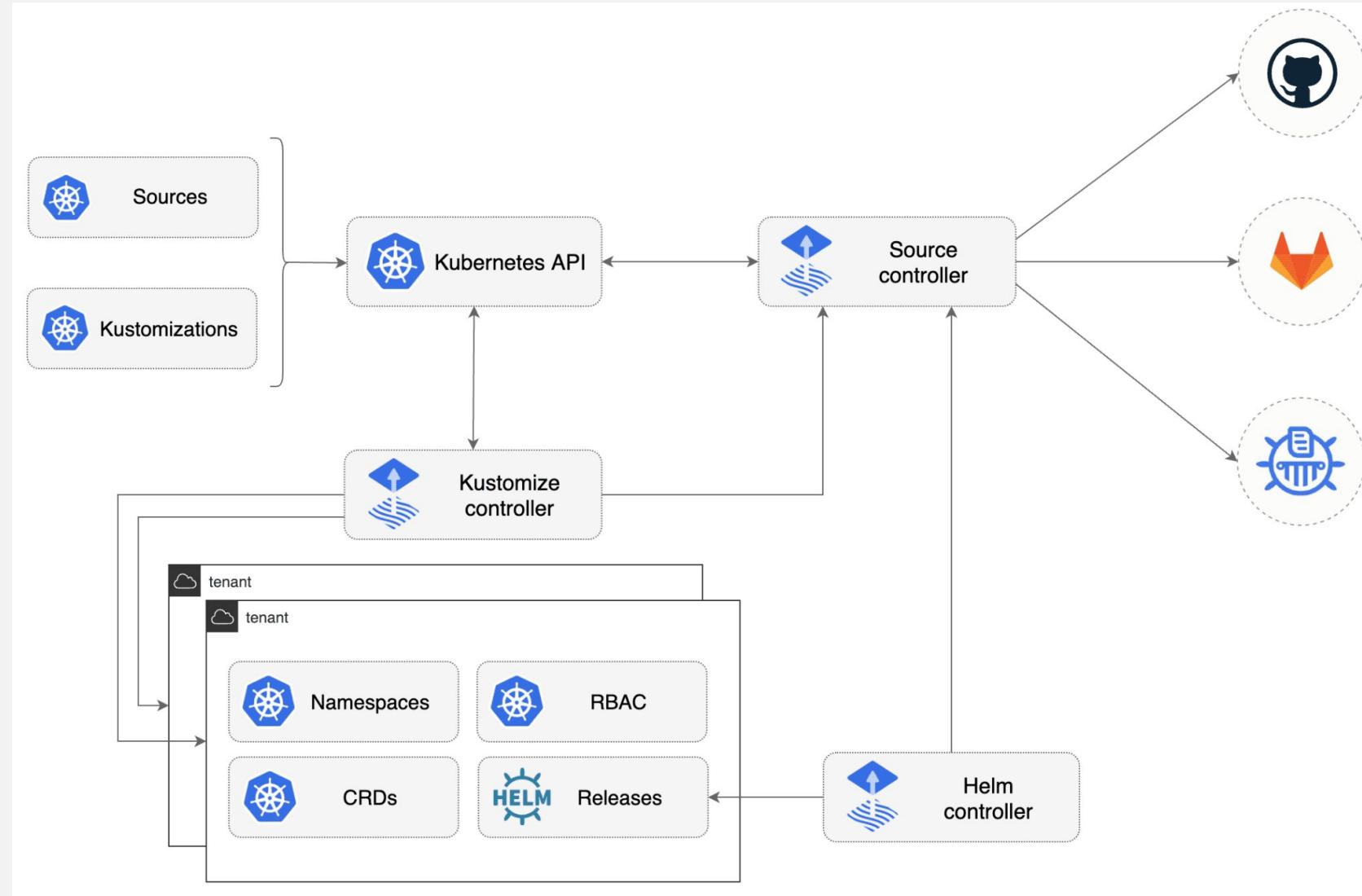


- CI / CD Pipeline braucht keinen cluster Zugriff mehr
 - => verbesserte Security, da für die Pipeline kein high privilege credential mehr notwendig ist
- Cluster pullen jetzt aktiv Source-Repos, welche den Zielzustand enthalten
- Cluster reconcilen ihrer Zustand, d.h. es wird der Ist-Zustand mit dem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

Flux im Detail



QA|WARE



Flux im Detail



- Source-Controller
 - bindet verschiedene Quellen an z.B. Git-Repos, OCI-Repos etc.
 - stellt die heruntergeladenen Pakete den anderen Controllern zur Verfügung
- Helm-Controller
 - bietet eine Helm Integration für Flux. In Verbindung mit dem Source-Controller lassen sich Helm-Charts automatisch herunterladen und installieren/updaten
- Kustomize-Controller
 - nutzt die Git-Repos des Source-Controllers um kustomize auszuführen und die konfigurierten Ressourcen im Cluster zu applien. Alle Ressourcen werden getracked und können auch automatisiert wieder garbage collected werden.
- Notification-Controller
 - bietet insights in flux events z.B. Erfolge oder aufgetretene Fehler.
 - bietet Integrationen in Chat Systeme

Sources konfigurieren



```
apiVersion: source.toolkit.fluxcd.io/v1
kind: GitRepository
metadata:
  name: podinfo
  namespace: default
spec:
  interval: 5m0s
  url: https://github.com/stefanprodan/podinfo
  ref:
    branch: master
```

- konfiguriert ein **GitRepository** als Quelle im Source-Controller
- klont den **master** branch von **url** und stellt ihn den anderen controllern als tar.gz zur Verfügung
- pollt die **url** alle 5 Minuten und prüft, ob es Änderungen gibt

Kustomizations konfigurieren



```
apiVersion: kustomize.toolkit.fluxcd.io/v1
kind: Kustomization
metadata:
  name: podinfo
  namespace: default
spec:
  interval: 10m
  targetNamespace: default
  sourceRef:
    kind: GitRepository
    name: podinfo
  path: "./kustomize"
  prune: true
  timeout: 1m
```

- erstellt eine Kustomization im Kustomize-Controller, die über **sourceRef** das GitRepository vom Source-Controller referenziert
- der kustomize-Controller holt sich den Source-Code vom Source-Controller und applied mit kustomize (k8s-Tool) alles unter **path**



QA|WARE

Q & A