

# CLOUD-NATIVE PROGRAMMIERUNG

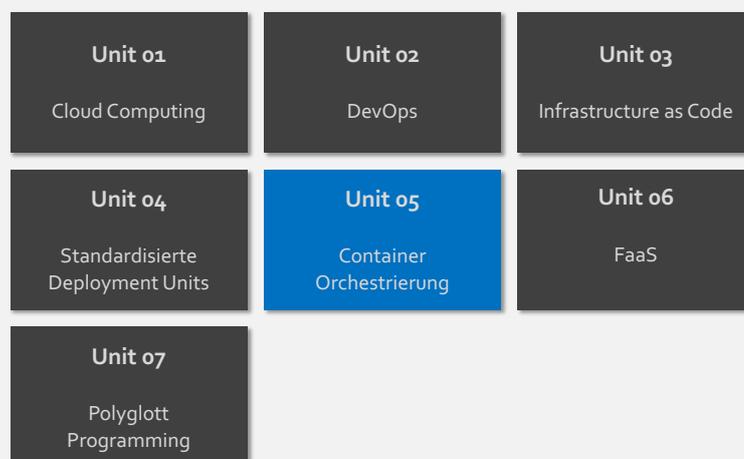
Unit 05:  
Container  
Orchestrierung

Stand: 09.10.2020

1

## INHALTSVERZEICHNIS

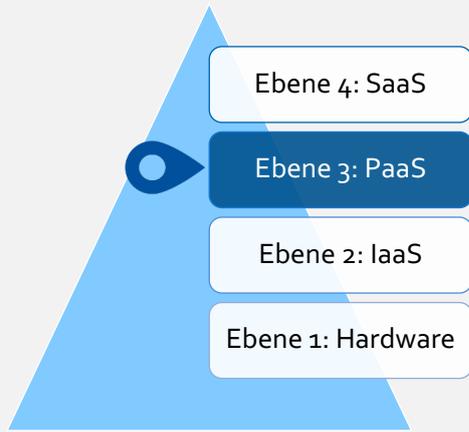
Überblick über Units und Themen dieses Moduls



2

# DAS SCHICHTENMODELL DES CLOUD COMPUTINGS

Wo sind wir?



## Kunden, Endnutzer

- Anpassbare Software-Dienste
- XaaS (Everything as a Service)
- Transparente Updates

## Entwickler

- Programmierschnittstellen (APIs)
- Plattformdienste
- Abstraktion der technischen Infrastruktur

## Administratoren

- Elastizität
- Virtuelle Ressourcenpools
- Technische Infrastruktur (VM, Storage, Network)

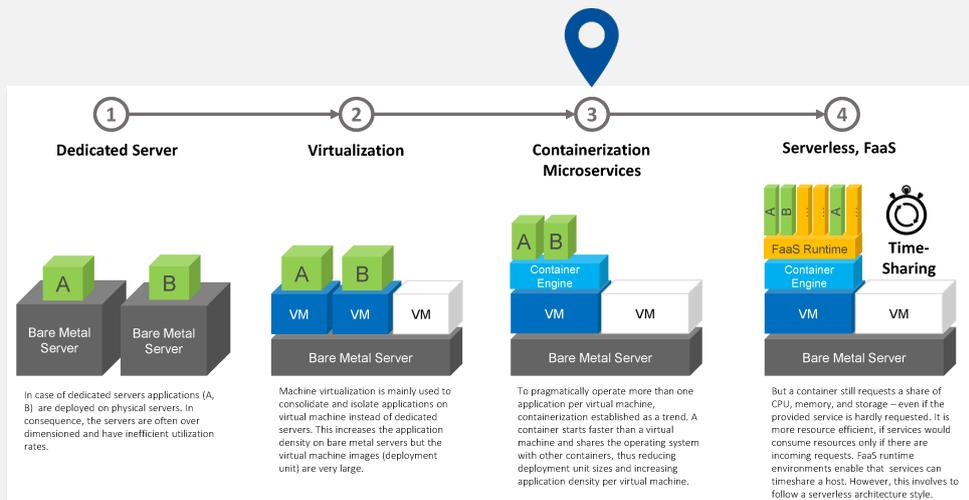
## Rechenzentrum

- Rechner
- Netzwerk
- Storage

3

# EINE KURZE GESCHICHTE DER CLOUD

Wo sind wir?



4

# INHALTE

## Scheduling

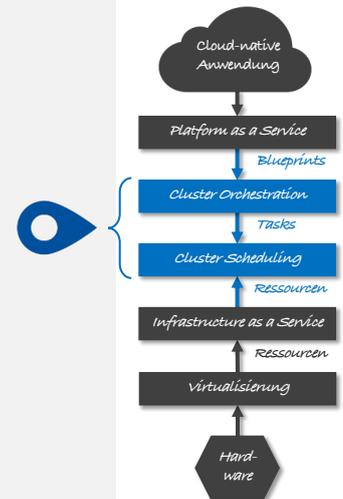
- Scheduling Problem Definition
- Scheduling Algorithmen
- Scheduler Architekturen
- Beispiele von Cluster Schemlern: Mesos, Swarm

## Orchestrierung

- Was ist Orchestrierung (in Abgrenzung zum Scheduling)?
- Was sind Blueprints?
- Container-Orchestrierungspatterns
- Überblick über bestehende Orchestrierungslösungen

## Inside Kubernetes (Typ-Vertreter)

- K8S-Architektur
- K8S-Ressourcen
- Workloads, Persistenz, Isolation und Exponieren von Services



# INHALTE

## Scheduling

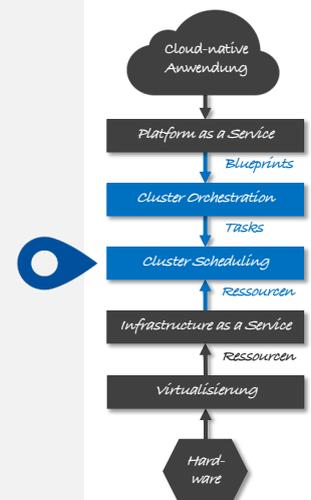
- Scheduling Problem Definition
- Scheduling Algorithmen
- Scheduler Architekturen
- Beispiele von Cluster Schemlern: Mesos, Swarm

## Orchestrierung

- Was ist Orchestrierung (in Abgrenzung zum Scheduling)?
- Was sind Blueprints?
- Container-Orchestrierungspatterns
- Überblick über bestehende Orchestrierungslösungen

## Inside Kubernetes (Typ-Vertreter)

- K8S-Architektur
- K8S-Ressourcen
- Workloads, Persistenz, Isolation und Exponieren von Services



# CLUSTER SCHEDULING

Das Problem



Rechenaufgaben  
(Jobs/Workloads)



Rechenressourcen  
(z.B. per IaaS oder GRID)

*Wie möglichst  
Ressourcen-effizient  
und Aufgaben-  
angemessen  
zuteilen?*

# CLUSTER SCHEDULING

Terminologie



**Job:**

Menge an Tasks mit gemeinsamen Ausführungsziel. Die Menge an Tasks ist in der Regel als DAG mit Tasks als Knoten und Ausführungsabhängigkeiten als Kanten darstellbar.

**Task:**

Atomare Rechenaufgabe inkl. Ausführungsvorschrift.

**Properties:**

Ausführungsrelevante Eigenschaften der Jobs und Tasks

- Job: z.B. Abhängigkeiten der Tasks, Ausführungszeitpunkt
- Task: Ausführungsdauer, Priorität, Ressourcenbedarf



**Scheduler:**

Ausführung von Tasks auf den verfügbaren Ressourcen unter Berücksichtigung der Properties und zu optimierender Scheduling-Ziele (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann präemptiv sein, also Tasks unterbrechen und neu aufsetzen.

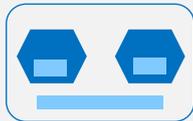


**Ressources:**

Cluster an Rechnern mit CPU-, RAM-, (H/S)DD- und Netzwerkressourcen. Ein Rechner stellt seine Ressourcen temporär zur Ausführung eines oder mehrerer Tasks zur Verfügung (Slot). Die parallele Ausführung von Tasks ist isoliert zu einander.

# CLUSTER SCHEDULER

Aufgaben



## Cluster-Awareness:

Die aktuell verfügbaren Ressourcen im Cluster kennen (Knoten inkl. verfügbare CPUs, RAM, Diskspace, Netzwerkbandbreite). Dabei auch auf Elastizität reagieren (hinzufügen, entfernen von Knoten).

## Job Allocation:

Zur Ausführung eines Workloads die passende Menge an Ressourcen für einen bestimmten Zeitraum bestimmen und allokatieren.

## Job Execution:

Einen Workload zuverlässig ausführen und dabei isolieren und überwachen.

9

# SCHEDULING

Statische Partitionierung



## Vorteil:

- Einfach zu realisieren

## Nachteil:

- Nicht flexibel bei sich ändernden Bedürfnissen
- Geringe Auslastung
- Hohe Opportunitätskosten

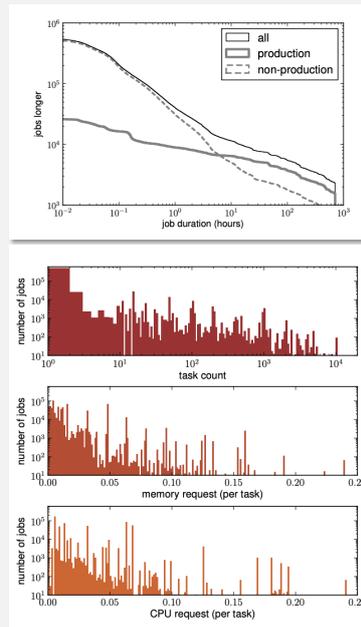
*Benjamin Hindman:*  
„Static partitioning considered harmful!“

10

# SCHEDULING

## Heterogenität

- In typischen Clustern sind Jobs (und deren Workload) üblicherweise sehr heterogen (Beispiel rechts).
- Charakteristische Unterschiede sind u.a.:
  - Dauer:** Sekunden, Minuten, Stunden, Tage, unendlich
  - Terminierung:** Sofort, später, zu einem def. Zeitpunkt
  - Zweck:** Datenverarbeitung, Request-Handling
  - Verbrauch:** CPU-, RAM-, HDD-, Netzwerk-dominant
  - State:** Zustandsbehaftet, zustandslos
- Zu unterscheiden sind mindestens:
  - Batch-Jobs:** Ausführungszeit üblicherweise im Minuten- bis Stundenbereich. Eher niedrige Priorität und gut unterbrechbar. Müssen häufig bis zu einem bestimmten Zeitpunkt abgeschlossen sein. Zustandsbehaftet.
  - Service-Jobs:** Sollen auf unbestimmte Zeit unterbrechungsfrei laufen. Haben eine hohe Priorität und sollten nicht unterbrochen werden. Teilweise zustandslos.



Beispiel einer Auswertung von Google Rechenzentren (Untersucht wurden 3 Plattformen mit insgesamt mehr als 10000 Knoten).

Es gibt zwar relativ viele kleine, Ressourcen-arme und kurze Jobs, aber eben nicht nur!

Cloud-Ressourcen können daher mittels dynamischer Partitionierung wesentlich effizienter genutzt werden.

„CPU and memory units are linearly scaled so that the maximum machine is 1.“ (siehe Quelle)

Quelle: Reiss et. al., Heterogeneity and Dynamism of Clouds at Scale: Google Trace Analysis, 2012  
<https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-soc2012.pdf>

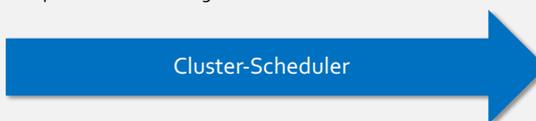
# SCHEDULING

## Awareness + Request → Optimierung → Placement Decision

### Awareness über Jobs/Tasks (Properties) und Ressourcen

- Resource:** Welche Ressourcen stehen bereit, welchen Bedarf hat der Task?
- Data:** Wo sind die Daten, die ein Task benötigt (Data locality)?
- QoS:** Welche Ausführungszeiten müssen garantiert werden?
- Economy:** Welche Betriebskosten sind einzuhalten?
- Priority:** Welche Priorität hat der Task?
- Failure:** Wahrscheinlichkeit eines Ausfalls? (sind bspw. Racks für Wartung markiert?)
- Experience:** Wie hat sich ein Task in der Vergangenheit verhalten?

Verarbeitung der Awareness-Daten im Cluster-Scheduler mittels Scheduling-Algorithmen entsprechend der jeweiligen beispielhaften Scheduling-Ziele:



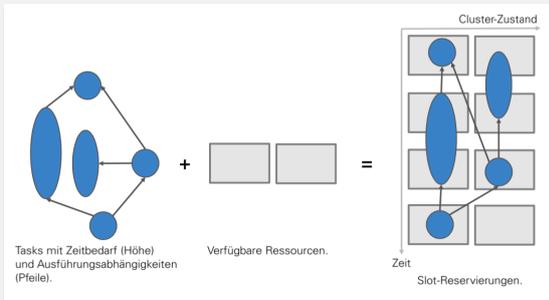
- Fairness:** Kein Task soll unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird
- Maximaler Durchsatz:** So viele Tasks pro Zeiteinheit wie möglich
- Minimale Wartezeit:** Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks
- Ressourcen-Auslastung:** Möglichst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

### Placement Decision

- Slot-Reservierung**
- Slot-Stornierungen:** Im Fehlerfall, Optimierungsfalls, bei Constraint-Verletzungen

# SCHEDULING

ist dummerweise eine NP-vollständige Optimierungsaufgabe



Es ist also kein Algorithmus bekannt, der eine optimale Lösung in polynomialer Laufzeit erzeugt.

Ein Algorithmus muss für tausende Jobs und tausende Ressourcen skalieren.

Optimale Algorithmen, die den Lösungsraum komplett durchsuchen sind nicht praktikabel, da deren Entscheidungszeit zu lange dauert für große Eingabemengen ( $|Jobs| \times |Ressourcen|$ ).

*Das Gute daran ist: Sie finden hier ein großes Feld für Forschung ;-)*

Darüber hinaus kommen Job-Anfragen üblicherweise kontinuierlich an, so dass selbst bei optimaler Allokation der Re-Organisationsaufwand pro Job unverhältnismäßig hoch werden kann.

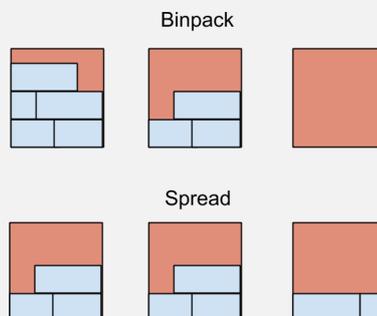
# SCHEDULING

Einfache Algorithmen

... optimieren das Scheduling von Tasks oft in genau einer Dimension (z.B. CPU-Auslastung) bzw. wenigen Dimensionen (CPU + RAM):

Populäre Algorithmen:

- **Binpack**  
Fit First => Hohe Auslastung von Knoten
- **Spread**  
Round Robin => Gleichmäßige Auslastung von Knoten



*So arbeitet bspw. Docker Swarm*



# SCHEDULING

So arbeitet bspw. Mesos.



## Multidimensionale Algorithmen am Bsp. Dominant Resource Fairness (DRF)

Aufteilung der Ressourcen an verschiedene User (Kunden, Projekte, etc.)

**Ausgangslage:** Jeder User hat eine dominante Ressource, die besonders intensiv genutzt wird. Diese Ressource kann durch Beobachtung ermittelt werden.

**Fairness Auffassung:** Jeder User bekommt mind.  $1/N$  aller Ressourcen der dominanten Ressourcen (equalize dominant share).

Der Scheduling Algorithmus ist also darauf ausgelegt, die dominanten Ressourcen pro User zu maximieren.

**Beispiel:**

Cluster mit: 9 CPU 18 GB Mem  
 User A Task benötigt: 1 CPU 4 GB Mem  
 User B Task benötigt: 3 CPU 1 GB Mem

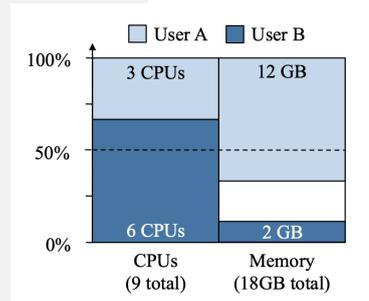
Rel. Bedarf A Task:  $1/9$  CPU  $2/9$  Mem  
 Rel. Bedarf B Task:  $1/3$  CPU  $1/18$  Mem

$$\begin{aligned}
 a + 3b &\leq 9 && \text{CPU constraint} \\
 4a + b &\leq 18 && \text{Memory constraint} \\
 \frac{2}{9}a &= \frac{b}{3} && \text{Equalize dominant shares}
 \end{aligned}$$

Also:

$$\begin{aligned}
 \frac{2}{3}a = b &\Rightarrow a \leq 3 \wedge a \leq 18 \frac{3}{14} \approx 3.86 \\
 &\Rightarrow a = 3 \wedge b = 2
 \end{aligned}$$

Die Fairness kann auch noch gewichtet werden. Wenn ein Team doppelt so wichtig wäre, würde es in der Ausgangslage doppelt so viel Ressourcen bekommen.



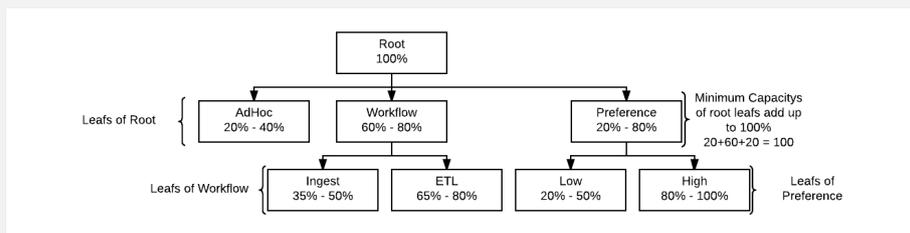
Quelle: Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. <https://dl.acm.org/doi/10.5555/1972457.1972490>

# SCHEDULING

## Kapazitäts-basierte Algorithmen am Bsp. Capacity Scheduling (CS)

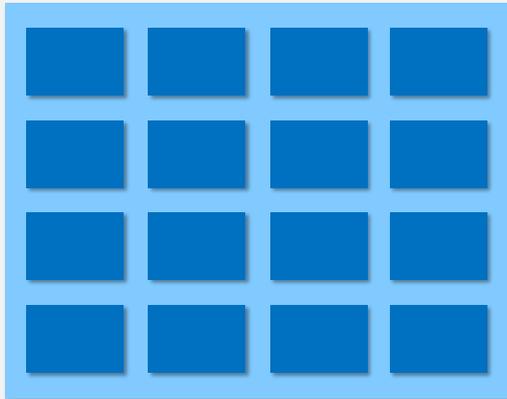
- Es werden Job Queues definiert und zu jeder Queue eine Kapazitätssusage in Ressourcenanteilen vom Cluster definiert.
- Fairness-Auffassung: Die minimale Kapazitätssusage wird stets eingehalten. Der Scheduling-Algorithmus stellt sicher, dass diese Fairness stets sichergestellt ist.
- Damit das Cluster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

So arbeitet bspw. YARN (Yet Another Resource Negotiator) des Hadoop-Systems.



# CLUSTER - SCHEDULER

*The Datacenter as a Computer*



## Idee:

Ein Cluster sieht von Außen aus wie ein großer Computer.

## Konsequenz:

Es müssen als Fundament viele Konzepte klassischer Betriebssysteme übertragen werden (ein Cluster-Betriebssystem).

Das gilt insbesondere auch für das Scheduling.

17

# CLUSTER - SCHEDULER

*Eine konzeptionelle Architektur*

## Job Queue:

Eingehende Jobs zur Ausführung  
Events zu eingegangenen Jobs

## Job Scheduler:

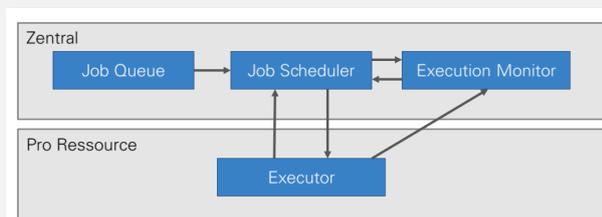
Jobs einplanen  
Taskausführung steuern

## Execution Monitor:

Taskausführung überwachen  
Ressourcen überwachen

## Executor:

Task ausführen  
Informationen zur Ressource bereitstellen



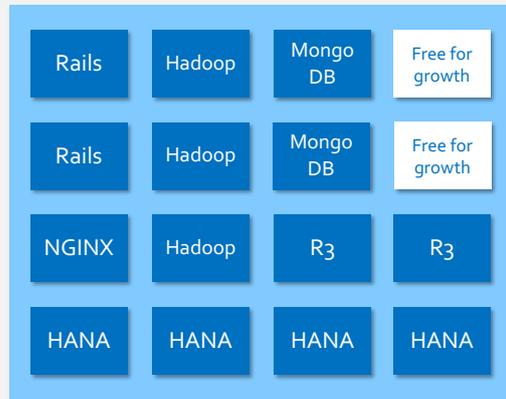
## Anforderungen:

- Performance (Geringe Queueing-Time, Decision-Time, Ausführungslatenz)
- Hoch-Verfügbarkeit und Fehlertoleranz
- Skalierbarkeit bzgl. Anzahl an Jobs und verfügbaren Ressourcen

18

# SCHEDULER ARCHITEKTUREN

Variante 1: Statische Partitionierung

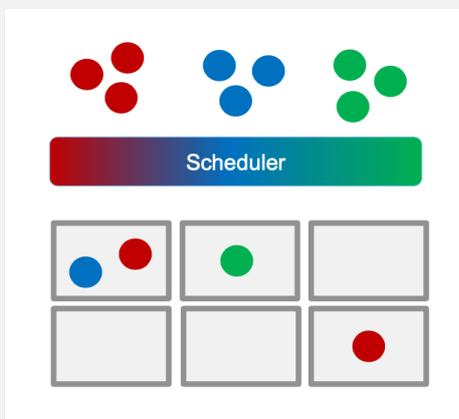


*Kann man kaum Scheduler nennen.*

19

# SCHEDULER ARCHITEKTUREN

Variante 2: Monolithischer Scheduler



## Vorteile:

Globale Optimierungsstrategien einfach möglich.

## Nachteile:

Heterogenes Scheduling für heterogene Jobs schwierig.

- Komplexe und umfangreiche Implementierung notwendig
- ... oder homogenes Scheduling geringer Effizienz.

Potenzielles Skalierbarkeits-Bottleneck.

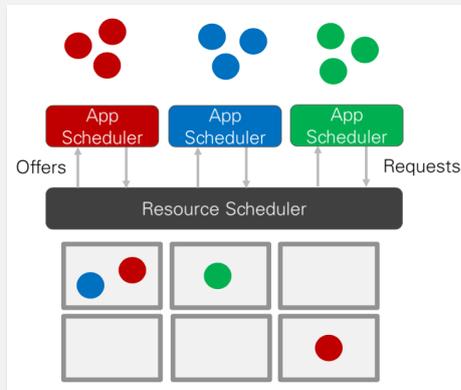
## Beispiele:

- Google Borg
- Hadoop YARN
- Kubernetes
- Docker Swarm

20

# SCHEDULER ARCHITEKTUREN

## Variante 3: 2-Level Scheduler



Auftrennung der Scheduling-Logik in einen Resource Scheduler und einen App Scheduler.

- Der **Resource Scheduler** kennt alle verfügbaren Ressourcen und darf diese allozieren. Er nimmt Ressourcen-Anfragen (Requests) entgegen und unterbreitet entsprechend einer Scheduling Policy Ressourcen-Angebote (Offers)
- Der **App Scheduler** nimmt Jobs entgegen und „übersetzt“ diese in Ressourcen-Anfragen und wählt applikationsspezifisch die passenden Ressourcen-Angebote aus.

Offers sind eine zeitlich beschränkte Allokation von Ressourcen, die explizit angenommen werden muss.

Im Sinne der Fairness kann ein prozentualer Anteil der Ressourcen pro App Scheduler garantiert werden.

### Beispiele:

- *Apache Mesos*



**Vorteile:**  
Mit Mesos nachgewiesene Skalierbarkeit auf tausenden von Knoten (z.B. Twitter, Airbnb, Apple Siri)

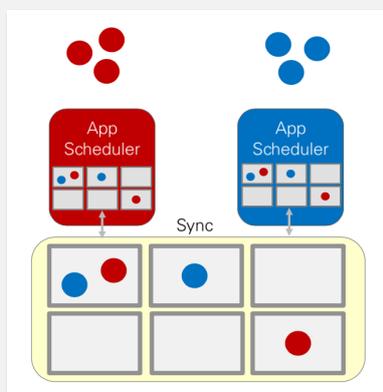
Flexible Architektur für heterogene Scheduling-Logiken

**Nachteile:**  
App-Scheduler übergreifende Logiken nur schwer zu realisieren

Quelle: Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. <https://people.eecs.berkeley.edu/~aig/papers/mesos.pdf>

# SCHEDULER ARCHITEKTUREN

## Variante 4: Shared-State-Scheduler



Es gibt ausschließlich applikationsspezifische Scheduler.

Die App-Scheduler synchronisieren kontinuierlich den aktuellen Zustand des Clusters (Job-Allokationen und verfügbare Ressourcen).

Jeder App-Scheduler entscheidet die Platzierung von Tasks auf Basis des ihm bekannten aktuellen Cluster-Zustands.

Optimistische Strategie: Ein zentraler Koordinierungsdienst erkennt Konflikte im Scheduling und löst diese auf, in dem er Zustands-Änderungen nur für einen der beteiligten App-Scheduler erlaubt und für die anderen App-Scheduler einen Fehler meldet.

### Beispiele:

- *Google Omega*

**Vorteile:**  
Tendenziell geringerer Kommunikations-Overhead.

**Nachteile:**  
Komplettes Scheduling muss pro App-Scheduler entwickelt werden.

Keine globalen Scheduling-Ziele (z.B. Fairness) möglich.

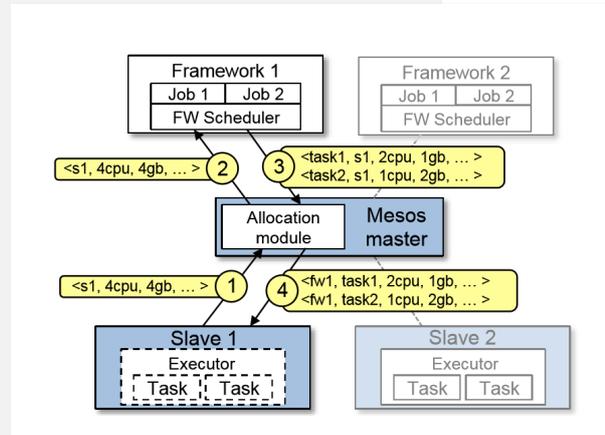
Skalierbarkeit in groen Clustern unklar, da noch nicht in der Praxis erprobt und insbesondere Auswirkung bei hoher Anzahl an Konflikte ungeklärt.

Quelle: Schwarzkopf et al. Omega: flexible, scalable schedulers for large compute clusters, SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic (2013), <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/41684.pdf>

## BEISPIELE

### Apache Mesos

- Entstanden an der UC Berkely im Rahmen der Arbeiten von Benjamin Hindman (1. Release 2009)
- Open Source Project unter Apache Lizenz 2.0
- Im Kern ein Cluster-Scheduler.
- Mesos ist als Cluster-Scheduler in DC/OS (Open Source Cluster-Betriebssystem) enthalten.
- 2-Level Scheduler (Dominant Resource Fairness)
- Alle Bestandteile von Mesos können ausfallsicher ausgelegt werden.
- Wird im großen Stil bei Twitter, Apple, Microsoft, Verizon, CERN, Airbnb, ... eingesetzt.
- Alle Teile sind per REST-API zugänglich.
- Task-Isolation per Docker oder eigenem Mechanismus.



Quelle: Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. <https://people.eecs.berkeley.edu/~aig/papers/mesos.pdf>

PROF.DR.  
NANEKRATZKE

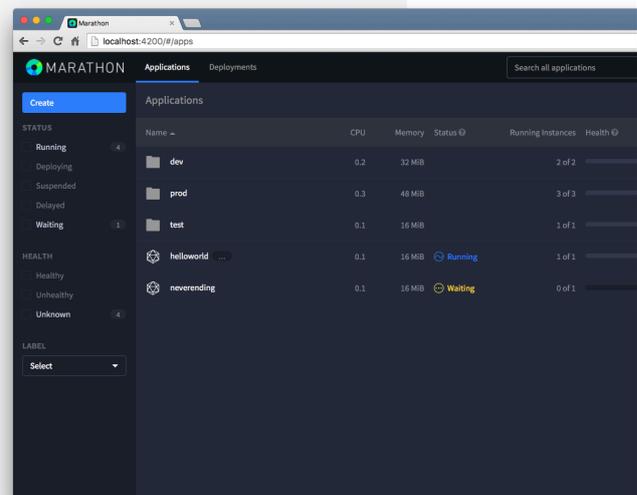
23

23

## BEISPIELE

### Apache Mesos + Marathon

- Marathon ist ein 2nd-Level-Scheduler der auf die Ausführung von zustandslosen Services ausgelegt ist.
- Autor: Tobi Knap (Ziel: Langlaufende zustandslose Services zuverlässig ausführen)
- Besitzt eigenständig Web-UI und REST-API
- Prozesse werden kontinuierlich am Leben gehalten. Terminiert ein Prozess, so wird er automatisch wieder gestartet.
- Mechanismen für Health-Checking von Services.
- Eingebauter Mechanismus für Service-Discovery und Load-Balancing.



<https://github.com/mesosphere/marathon>

PROF.DR.  
NANEKRATZKE

24

24

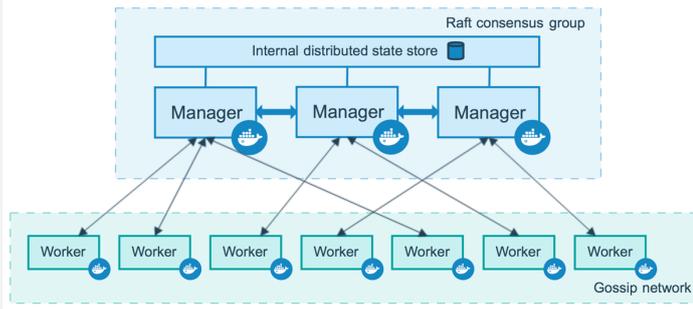
# BEISPIELE

## Docker Swarm

- Ein Docker Swarm besteht aus mehreren Docker Nodes, die im sogenannten Swarm Mode laufen.
- Die Nodes sind Manager, Worker, oder auch beides.
- Manager verteilen die Tasks auf die Worker.
- Die Aufgabe der Worker ist die Ausführung der Container.
- Ein Task ist ein laufender Container, der Teil eines Swarm Services ist.
- Für Services gibt man den Container, die auszuführenden Commands und bei Bedarf weitere Konfigurationen an.

## Scheduling gem. Spread Strategy

- Ein Task (Container) werden gleichmäßig auf die Nodes verteilt.
- Ein Task wird einem Node zugewiesen der noch keinen Task für den Service ausführt.
- Wenn schon alle Nodes einen Task für dessen Service ausführen, wird der Node gewählt, der die wenigsten Tasks für diesen Service ausführt.
- Zusätzlich lassen sich Ressourcenbedarf und Limits angeben. Können diese auf dem eigentlich zugewiesenen Node nicht erfüllt werden, wird der nächst mögliche Node gewählt.
- Mittels Constraints und Preferences lassen sich weitere Randbedingungen für das Scheduling formulieren.



Quelle: <https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/>

# INHALTE

## Scheduling

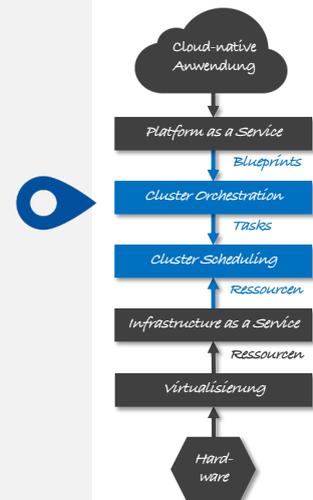
- Scheduling Problem Definition
- Scheduling Algorithmen
- Scheduler Architekturen
- Beispiele von Cluster Schemlern: Mesos, Swarm

## Orchestrierung

- Was ist Orchestrierung (in Abgrenzung zum Scheduling)?
- Was sind Blueprints?
- Container-Orchestrierungspatterns
- Überblick über bestehende Orchestrierungslösungen

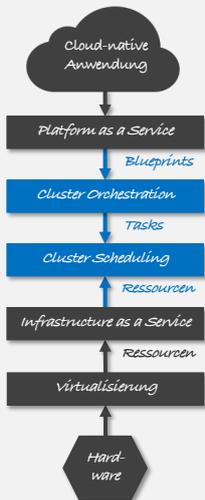
## Inside Kubernetes (Typ-Vertreter)

- K8S-Architektur
- K8S-Ressourcen
- Workloads, Persistenz, Isolation und Exponieren von Services



# BACK TO BIG PICTURE

## Orchestrierung



3 x NGINX Webserver + vorgelagerter HAproxy (ausfallsicher)

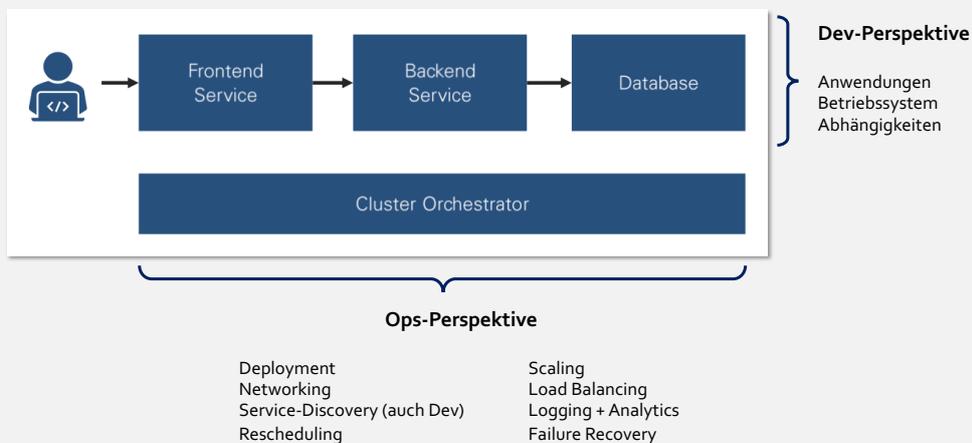
3 Instanzen NGINX, mind. 2 Instanzen HAproxy mit Liveness Probes

*„How would you design your infrastructure if you couldn't login. Never!“*

Kelsey Hightower

# ORCHESTRIERUNG

## Unterschiedliche DevOps-Perspektiven



# ORCHESTRIERUNG

Was ist Cluster-Orchestrierung?

Eine Anwendung, die in mehrere Betriebskomponenten 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 (Blueprint) der Anwendung

Gewünschter Betriebszustand der Anwendung. Diese beschreibt die Betriebskomponenten (Container), deren Betriebsanforderungen sowie die angebotenen und benötigten Schnittstellen.

## Cluster-Orchestrierer

## Steuerungsaktivitäten im Cluster

- Start von Containern auf Knoten (→ Scheduler)
- Verknüpfung von Container
- Replizierung/Skalierung von Containern
- ...

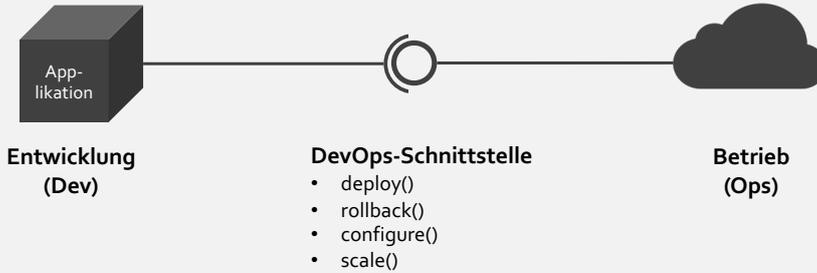
# CLUSTER - ORCHESTRIERER

Automatisierung von Betriebsaufgaben auf einem Cluster

- Scheduling von Containern mit Applikations-spezifischen Constraints (z.B. Affinities und Anti-Affinities)
- 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 Ressourcen-Optimierung (→ Scheduling)
- Container-Logistik: Verwaltung, Bereitstellung und Cacheing von Container-Images.
- Package-Management: Verwaltung und Bereitstellung von Applikationen.
- Bereitstellung von Administrationsschnittstellen (Remote-API, Kommandozeile).
- Management von Services: Service Discovery, Naming, Load Balancing.
- Automatismen für Rollout-Workflows wie z.B. Canary Rollout.
- Monitoring und Diagnose von Containern und Services.
- ...

# CLUSTER - ORCHESTRIERER

Schnittstelle zwischen Betrieb und Entwicklung

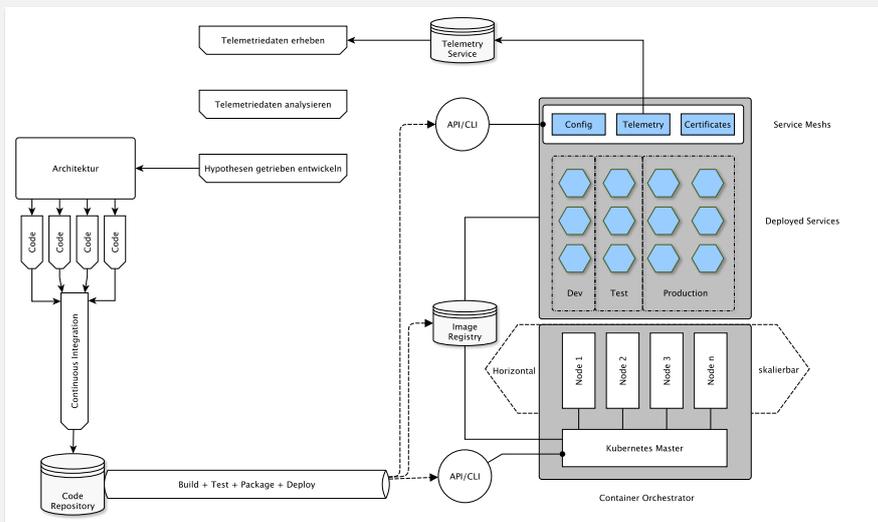


Quelle: QAware GmbH  
<https://github.com/qaware/cloudcomputing>

31

# CLUSTER - ORCHESTRIERER

Schnittstelle zwischen Betrieb und Entwicklung



*Oder etwas komplizierter.*

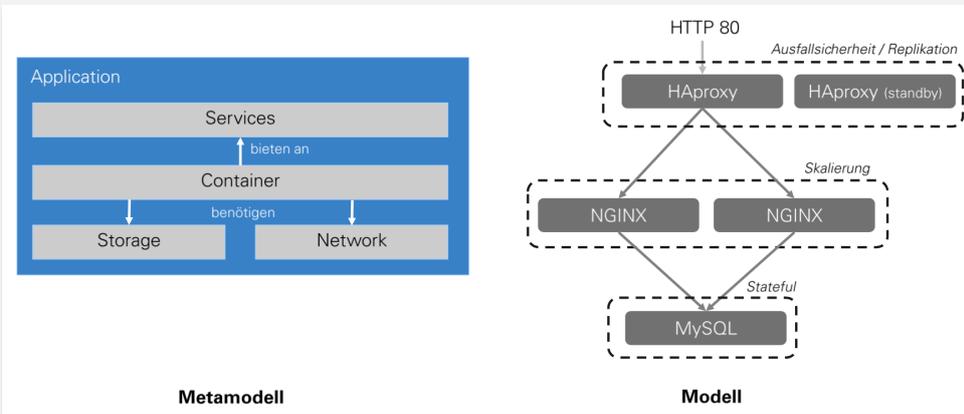
*Erinnern Sie sich an den DevOps-Cycle aus Unit 02.*

32

# BLUEPRINTS

Blaupausen einer Anwendung

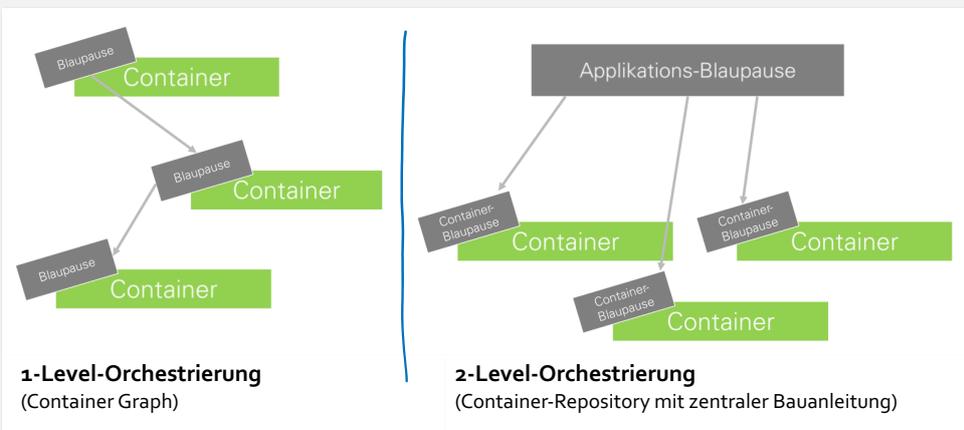
vereinfachte Darstellung



Quelle: QAware GmbH  
<https://github.com/qaware/cloudcomputing>

# CONTAINER - ORCHESTRIERUNG

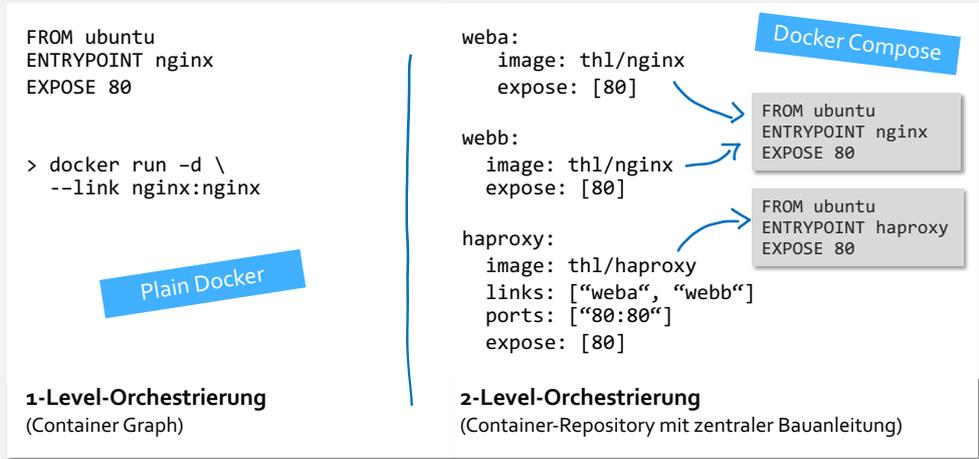
1-Level vs. 2-Level Orchestrierung



Quelle: QAware GmbH  
<https://github.com/qaware/cloudcomputing>

# CONTAINER - ORCHESTRIERUNG

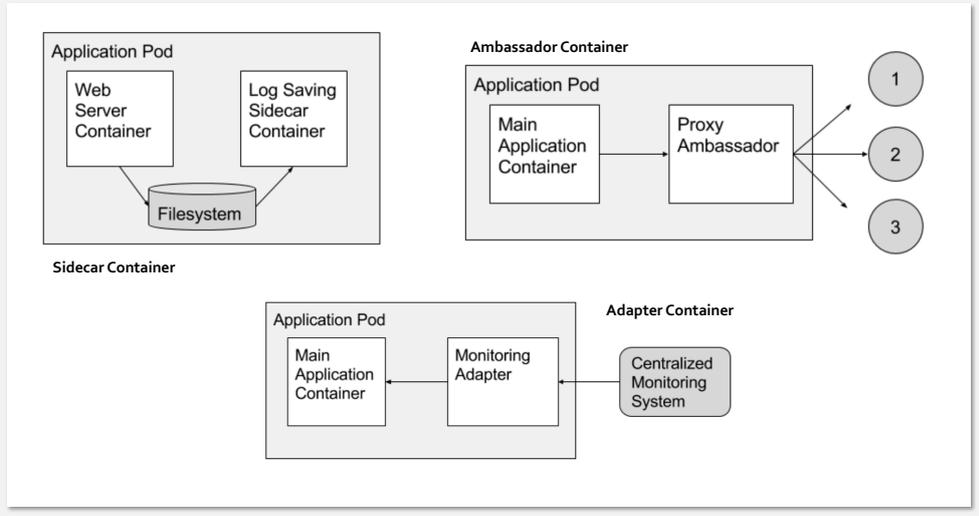
1-Level vs. 2-Level Orchestrierung



Quelle: QAware GmbH  
<https://github.com/qaware/cloudcomputing>

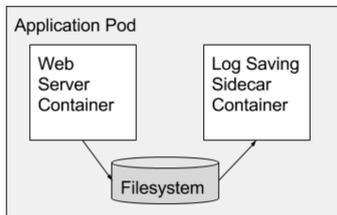
# ORCHESTRIERUNGS - PATTERNS

Separations of Concerns mittels modularer Container



# ORCHESTRIERUNGS - PATTERNS

## Sidecar Container



Sidecar Container

**„Sidecar containers extend and enhance the “main” container, they take existing containers and make them better.**

*As an example, consider a container that runs the Nginx web server. Add a different container that syncs the file system with a git repository, share the file system between the containers and you have built Git push-to-deploy.*

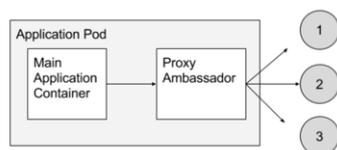
*But you’ve done it in a modular manner where the git synchronizer can be built by a different team, and can be reused across many different web servers (Apache, Python, Tomcat, etc). Because of this modularity, you only have to write and test your git synchronizer once and reuse it across numerous apps. And if someone else writes it, you don’t even need to do that.“*

**Brendan Burns**

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

# ORCHESTRIERUNGS - PATTERNS

## Ambassador Container



Ambassador Container

**„Ambassador containers proxy a local connection to the world.**

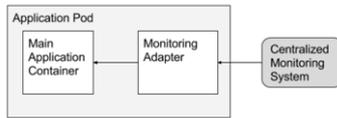
*As an example, consider a Redis cluster with read-replicas and a single write master. You can create a Pod that groups your main application with a Redis ambassador container. The ambassador as a proxy is responsible for splitting reads and writes and sending them on to the appropriate servers. Because these two containers share a network namespace, they share an IP address and your application can open a connection on “localhost” and find the proxy without any service discovery. As far as your main application is concerned, it is simply connecting to a Redis server on localhost.*

*This is powerful, not just because of separation of concerns and the fact that different teams can easily own the components, but also because in the development environment, you can simply skip the proxy and connect directly to a Redis server that is running on localhost.“*

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

# ORCHESTRIERUNGS - PATTERNS

## Adaptor Container



Adaptor Container

**„Adapter containers standardize and normalize output.“**

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

Again because these Pods share namespaces and file systems, the coordination of these two containers is simple and straightforward.“

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

# CLUSTER - ORCHESTRIERER

## Überblick

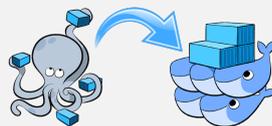
**Apache Mesos**  
Marathon + Chronos



Für große Rechenzentren

Eher Ops-fokussiert (DC, Data Center)

**Docker**  
Compose + Swarm



Lightweight

Mehr Dev-fokussiert

**Kubernetes**



De-facto Standard

**Anmerkung:**  
Die Systeme sind vom Prinzip her vergleichbar. Die Zielgruppen und damit die Feature-Schwerpunkte sind nur unterschiedlich.

Aus Gründen der Praktikabilität fokussieren wir im folgenden den De-facto Standard Typvertreter Kubernetes.

# INHALTE

## Scheduling

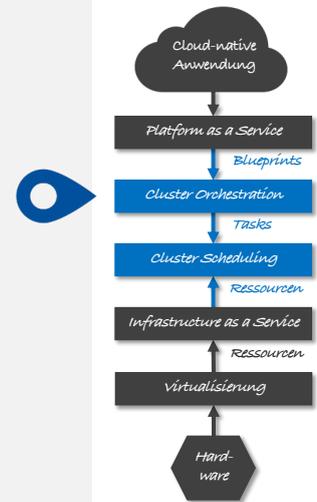
- o Scheduling Problem Definition
- o Scheduling Algorithmen
- o Scheduler Architekturen
- o Beispiele von Cluster Schemlern: Mesos, Swarm

## Orchestrierung

- o Was ist Orchestrierung (in Abgrenzung zum Scheduling)?
- o Was sind Blueprints?
- o Container-Orchestrierungspatterns
- o Überblick über bestehende Orchestrierungslösungen

## Inside Kubernetes (Typ-Vertreter)

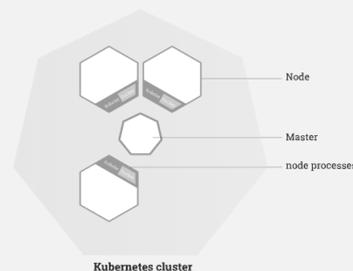
- o K8S-Architektur
- o K8S-Ressourcen
- o Workloads, Persistenz, Isolation und Exponieren von Services



# KUBERNETES

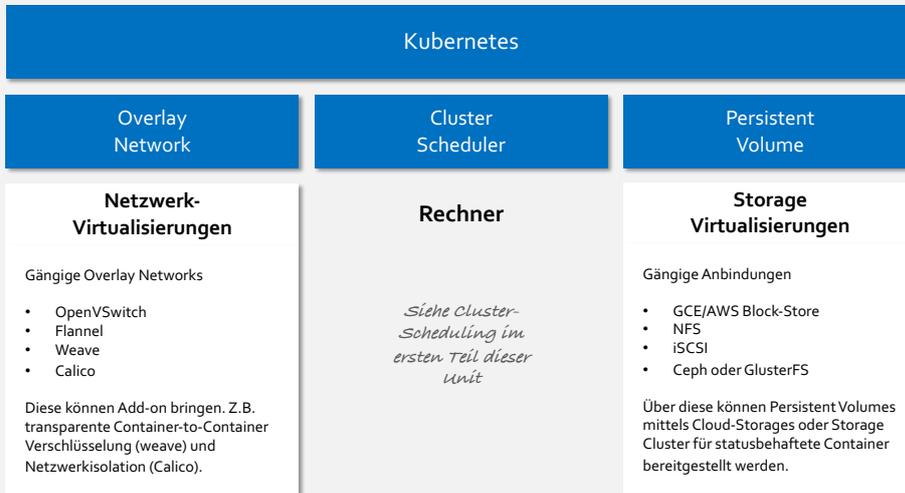
De-factor Standard für Container-Orchestrierung

- Cluster-Orchestrierer für Docker-Container, der eine Reihe an Kern-Abstraktionen für den Betrieb von Anwendungen in einem großen Cluster einführt.
- Applikations-Blueprints werden überYAML- oder JSON-Dateien (sogenannte Manifests) 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 Version 1.0. Skaliert auf bis zu 1000 Nodes großen Clustern.
- Beiträge zur Codebasis von vielen Firmen neben Google – u.a. Mesosphere, Microsoft, Pivotal, RedHat, u.v.m.
- Zur Standardisierung der Cluster-Orchestrierung wurde die Cloud Native Computing Foundation (<https://cncf.io>) gegründet.
- Kubernetes ist das Lead-Projekt der CNCF.



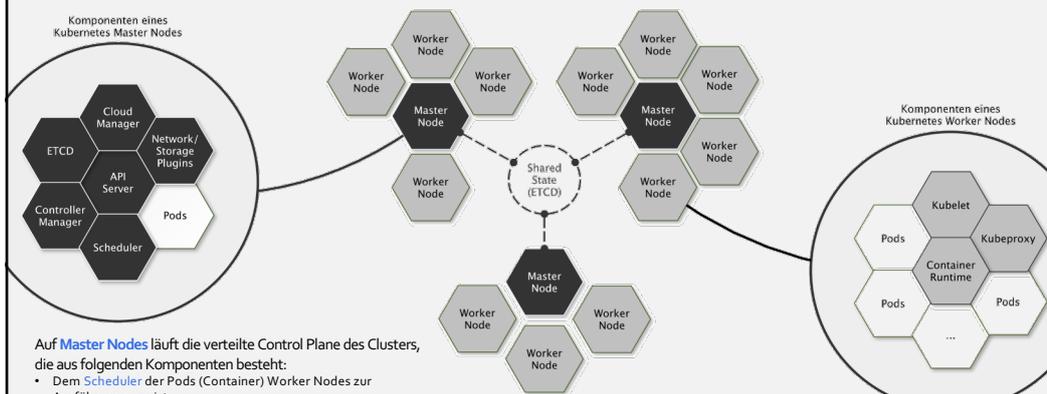
# KUBERNETES

## Netzwerk- und Storage-Virtualisierung



Neben einem Cluster-Scheduler setzt Kubernetes auch noch auf Netzwerk- und Storage-Virtualisierungen auf.

# KUBERNETES INFRASTRUKTUR



Auf **Master Nodes** läuft die verteilte Control Plane des Clusters, die aus folgenden Komponenten besteht:

- Dem **Scheduler** der Pods (Container) Worker Nodes zur Ausführung zuweist
- **Controller Managern**, die Ressourcen von Anwendungen überwachen und steuern.
- **Cloud Managern**, die Anbindungen an Cloud Infrastrukturen kapseln (z.B. Load Balancer Anforderungen).
- Einem verteilten und konsistenten **Key-Value-Store** (ETCD) um Clusterzustände hochverfügbar und konsistent zu speichern.
- Ein **API-Server** zur Steuerung und Verwaltung des Clusters
- Sowie weiteren Plugins, wie bspw. **Network** und **Storage Plugins**.
- Grundsätzlich können Pods auf Master Nodes ausgeführt werden (meist werden diese allerdings nur Worker Nodes zur Ausführung zugewiesen).

**Worker Nodes** dienen zur Ausführung von Applikationen (Pods). Die Interaktion mit der Control Plane erfolgt dabei über folgende Komponenten:

- Das **Kubelet** ist der primäre auf jedem Knoten laufende Node Agent, der u.a. den Knoten beim API-Server registriert.
- **Kube-proxy** wird als Netzwerk Proxy auf jedem Knoten ausgeführt und stellt die Kubernetes-API auf jedem Knoten bereit und ist u.a. für Weiterleitung von TCP-, UDP- und SCTP-Traffic sowie Load Balancing für Services verantwortlich.
- Um Container in Pods auszuführen, verwendet Kubernetes eine **Container Runtime** Environment, wie bspw. Docker, CRI-O oder containerd.

Kubernetes (K8s) ist ein Open-Source Cluster-System zur Automatisierung der Bereitstellung, Skalierung und Verwaltung von containerisierten Anwendungen.

Ein Kubernetes Cluster besteht aus **Master Nodes** (Control Plane) und **Worker Nodes** zur Ausführung von containerisierten Anwendungen.

Als Nutzer dieses Clusters interagiert man normalerweise mit dem Kommandozeilen-Programm **kubectl**. Zur Ausführung von Anwendungen erforderliche **Ressourcen** werden in sogenannten **Manifestdateien** (YAML oder JSON) definiert.

# KUBERNETES CONTROL PLANE

Master(s)

API server	Scheduler	etcd	Controller manager	Cloud manager
Exposes the Kubernetes REST-API.  Front end for the control plane.  Horizontally scalable (see etcd).	Selects a node for pods to run them on.  Factors for scheduling decisions include: <ul style="list-style-type: none"> <li>resource requirements</li> <li>hardware/ software/ policy constraints</li> <li>affinity and anti-affinity specifications</li> <li>data locality</li> <li>inter-workload interference</li> <li>deadlines</li> </ul>	Consistent and highly-available key value store for all cluster data.	Runs the following controller processes. <ul style="list-style-type: none"> <li><b>Node controller</b> notices and responds when nodes go down.</li> <li><b>Replication controller</b> maintains the correct number of pods for every replication controller object in the system.</li> <li><b>Endpoints controller</b> populates the Endpoints object (that is, joins Services &amp; Pods).</li> <li><b>Service Account &amp; Token controllers</b> create default accounts and access tokens for namespaces.</li> </ul>	Embeds cloud-specific control logic.  Kind of a "Cloud" driver.  The following controllers can have cloud provider dependencies: <ul style="list-style-type: none"> <li><b>Node controller:</b> For checking if a node has been deleted.</li> <li><b>Route controller:</b> For setting up routes in the underlying cloud infrastructure</li> <li><b>Service controller:</b> managing cloud provider load balancers</li> </ul>

*The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, restarting crashed pods).*

*Control plane components can be run on any machine in the cluster. However, mostly all control plane components are placed on dedicated master machines, that do not execute user containers.*

# KUBERNETES NODES

Nodes

kubelet	Kube-proxy	Container runtime
An agent that runs on each node in the cluster.  It makes sure that containers are running in a Pod.  The kubelet ensures that the containers described in those PodSpecs are running and healthy.  The kubelet doesn't manage containers which were not created by Kubernetes.	kube-proxy is a network proxy that runs on each node in your cluster.  kube-proxy maintains network rules on nodes. These network rules allow network communication to Pods from network sessions inside or outside of the cluster.  kube-proxy uses the operating system packet filtering layer if there is one and it's available.  Otherwise, kube-proxy forwards the traffic itself.	The container runtime is the software that is responsible for running containers.  Kubernetes supports several container runtimes: <ul style="list-style-type: none"> <li>Docker</li> <li>Containerd</li> <li>CRI-O</li> <li>and any implementation of the Kubernetes CRI (Container Runtime Interface).</li> </ul>

*Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.*

# KUBERNETES ADDONS

Features that are provided as cluster resources



### DNS

- A mandatory DNS server must serve DNS records for Kubernetes services.
- Containers started by Kubernetes automatically include this DNS server in their DNS searches.

### Web UI (Dashboard)

- Dashboard is a general purpose, web-based UI for Kubernetes clusters.
- It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

### Container Resource Monitoring

- Container Resource Monitoring records generic time-series metrics in a central database.
- It provides a UI for browsing that data.

### Cluster-level Logging

- A cluster-level logging mechanism is responsible for saving container logs to a central log store.
- It provides a search/browsing interface.

Addons use Kubernetes resources (DaemonSet, Deployment, etc) to implement cluster features.

Because these are providing cluster-level features, namespaced resources for addons belong within the kubernetes namespace.

Only very common addons are described listed.

# KUBERNETES

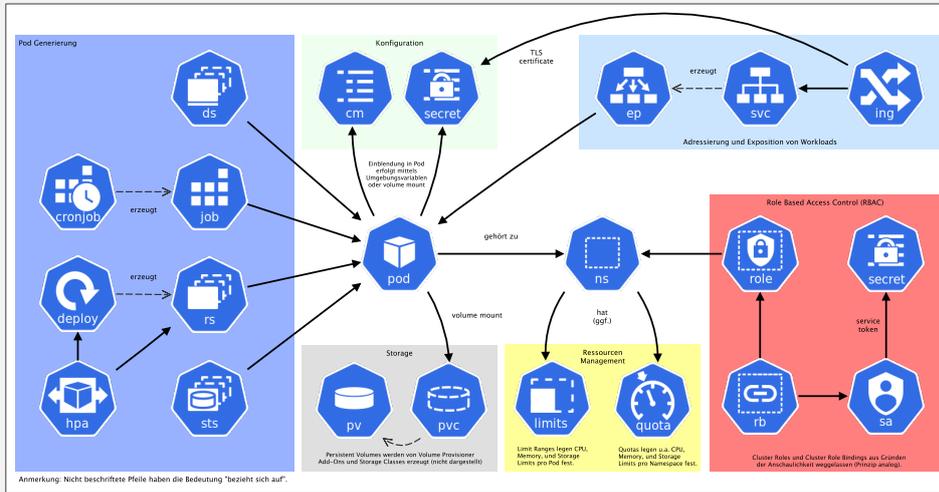
Überblick über die wichtigsten Kubernetes-Ressourcen



Ausführbare Jobs/Workloads		Persistenz	Isolieren von Anwendungen mittels Namespaces	Exponieren von Anwendungen und Diensten
Deployments (Cron-)Jobs	Persistent Volume Claim	Quotas und Limit Ranges 	Services <small>Primär Intra-Cluster</small>	
Daemon Sets Stateful Sets	Persistent Volume	Role Based Access Model	Ingress <small>Primär Extra-Cluster</small>	

# KUBERNETES

Die meist genutzten Ressourcen und ihre Zusammenhänge

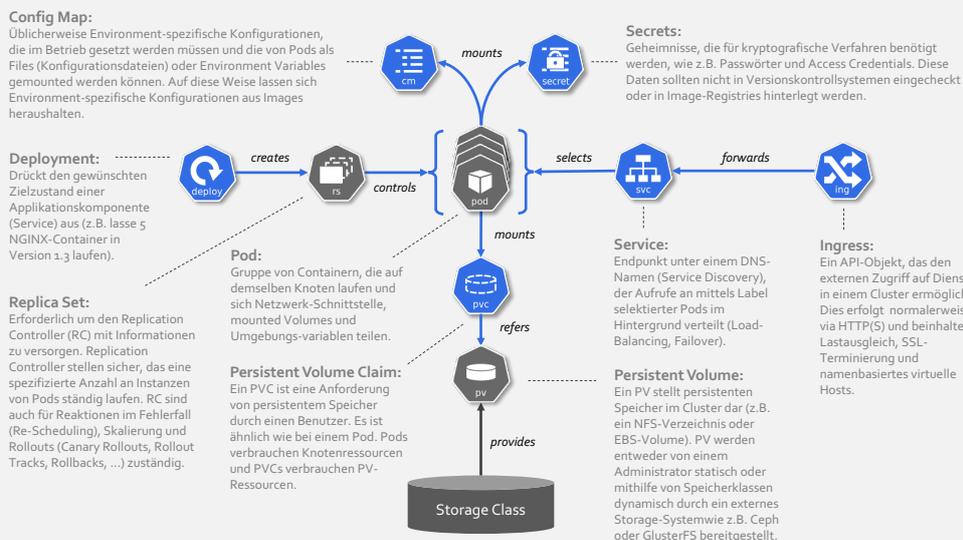


Von Kubernetes verwaltete Ressourcen

- Pod
- Namespace
- Deployment
- Horizontal Pod Autoscaler
- Replica Set
- Stateful Set
- Daemon Set
- (Cron-) Job
- ConfigMap
- Secret
- Persistent Volume
- Persistent Volume Claim
- ServiceAccount
- (Cluster-) Role
- Cluster Role Binding
- Service
- Endpoint
- Ingress
- Limit Ranges
- Quota

# KUBERNETES

Kern-Abstraktionen und Basis-Blueprint eines in K8S deployten Services

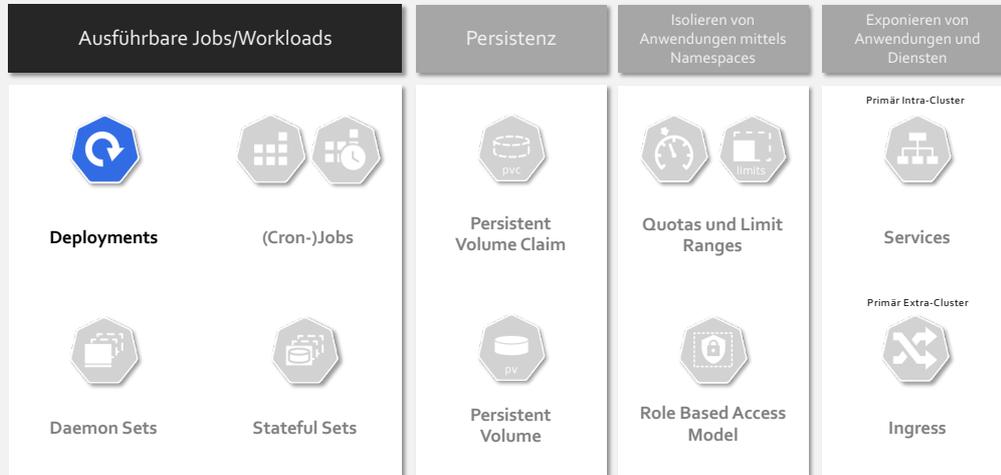


Legende:

- Ressource muss durch User angelegt werden.
- Ressource wird üblicherweise durch K8S-interne Prozesse (meist Controller) angelegt (kann aber auch durch User oder durch Administrator angelegt werden).

# KUBERNETES

Überblick über die wichtigsten Kubernetes-Ressourcen

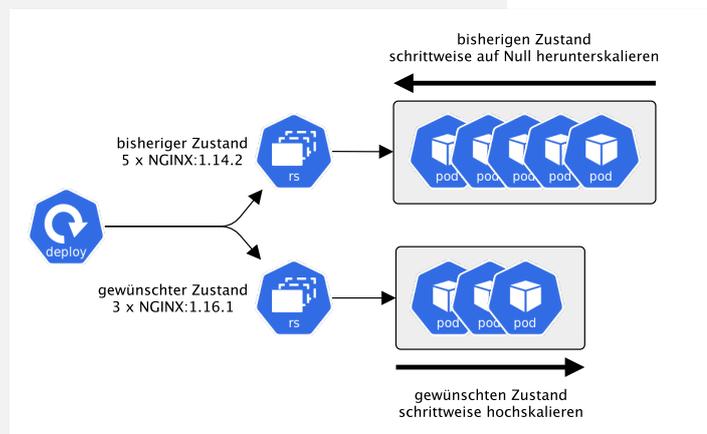


**Hinweis:**  
 Diverse Konzepte wie Scheduling Constraints (Requests + Limits, Node Selektoren, Affinities) sowie Health-Probes (Liveness, Readiness, Startup) werden an Deployments erläutert, lassen sich aber für die anderen K8S-Workloads analog anwenden!

# DEPLOYMENTS

Ausführbare Workloads

- Ein Deployment ermöglicht deklaratives Ausbringen und Aktualisieren von Applikationen.
- Sobald die Anwendungsinstanzen erstellt wurden, überwacht ein Deployment Controller diese Instanzen kontinuierlich. Wenn der Node, der eine Instanz hostet (oder ein Pod), ausfällt oder gelöscht wird, ersetzt der Deployment Controller die Instanz durch eine Instanz auf einem anderen Node im Cluster.
- Deployment Controller überführen hierzu einen Ist-Stand in einen Soll-Stand.
- Erforderliche Maßnahmen werden automatisch abgeleitet.
- Z.B. anlegen eines neuen Replica Sets, welches schrittweise auf die Wunschanzahl an Replicas hochskaliert wird, während das bestehende Replica Set schrittweise auf Null herunterskaliert und anschließend gelöscht wird).
- Sollte ein Update fehlerhaft sein, kann es auch zurückgenommen werden.



Quelle: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

# DEPLOYMENTS

Ausführbare Workloads

```
# Ausbringen eines Deployments (5 x NGINX Webserver)
kubectl apply -f xample-nginx-deployment.yaml

# Update eines Images (NGINX:1.14.2 => NGINX:1.16.1) und
# Downscaling von 5 auf 3 Replicas
kubectl set image deployment.v1.apps/nginx-deployment \
  nginx=nginx:1.16.1 --record=true
kubectl scale --replicas=3 -f xample-nginx-deployment.yaml \
  --record=true

# Alle Revisionen eines Deployments auflisten
kubectl rollout history deployment.v1.apps/nginx-deployment

# Letztes Update zurücknehmen
# (bzw. auf eine spezifische Revision zurückspringen)
kubectl rollout undo deployment.v1.apps/nginx-deployment \
  [--to-revision=2]
```

```
Manifest: xample-nginx-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels: {
    app: nginx
  }
spec:
  replicas: 5
  selector: { matchLabels: { app: nginx }}
  template:
    metadata: { labels: { app: nginx }}
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports: [{ containerPort: 80 }]
```

53

# SCHEDULING CONSTRAINTS

Requests und Limits

- Requests und Limits sind die Mechanismen, mit denen Kubernetes Ressourcen wie CPU und Speicher steuert.
- Requests sind das, was ein Container garantiert bekommt. Wenn ein Container eine Ressource anfordert, plant Kubernetes diese nur auf einem Knoten, der ihm diese Ressource geben kann.
- Limits stellen sicher, dass ein Container niemals einen bestimmten Wert überschreitet.
- Limits können niemals niedriger sein kann als der Request.
- **Requests und Limits werden pro Container in den Container-Specs angegeben.**
- Da Pods jedoch immer als Gruppe geplant werden, müssen Requests und Limits für alle Container eines Pods addiert werden, um einen Gesamtwert für den Pod zu erhalten.
- Um zu steuern, wie viele Ressourcen insgesamt vergeben werden können, können pro Namespace Quotas festgelegt werden.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels: {
    app: nginx
  }
spec:
  replicas: 5
  selector: { matchLabels: { app: nginx }}
  template:
    metadata: { labels: { app: nginx }}
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports: [{ containerPort: 80 }]
        resources: {
          requests: { cpu: "100m", memory: "100M" }
          limits: { cpu: "1000m", memory: "1G" },
        }
      }
```

*Auch für die weiteren Workloads (Jobs, DaemonSets, StatefulSets) kann man Requests und Limits an den entsprechenden Container-Specs vornehmen.*

54

# SCHEDULING CONSTRAINTS

Einheiten für CPU- und Speicher-Requests und -Limits

## CPU (Millicores)

- Requests und Limits für CPU-Ressourcen werden in CPU-Einheiten (Millicores) gemessen.
- Eine CPU in Kubernetes entspricht 1 vCPU / Core für Cloud-Anbieter und 1 Hyperthread auf Bare-Metal-Intel-Prozessoren.
- Bruchteile von Anforderungen sind zulässig, d.h. ein Container mit einem Request von 0.5 erhält garantiert halb so viel CPU wie einer, der 1 CPU anfordert.
- Der Ausdruck **0.1 entspricht dem Ausdruck 100m**, der üblicherweise als "einhundert Millicpu" (oder Millicores) gelesen wird.
- Eine Anforderung mit einem Dezimalpunkt wie 0,1 wird von der API in 100 m konvertiert.
- Eine Genauigkeit von weniger als 1 m ist nicht zulässig.
- Die CPU wird immer als absolute Menge angefordert, niemals als relative Menge. D.h. 0,1 ist die gleiche CPU-Menge auf einem Single-Core-, Dual-Core- oder 48-Core-Computer (die Taktrate der Prozessoren spielt dabei keine Rolle).

## Memory (und Storage)

- Speicher-Requests und Speicher-Limits werden in Bytes gemessen.
- Eine Speichermenge kann als einfache Ganzzahl oder als Festkommazahl mit einem der folgenden Suffixe ausgedrückt werden:
  - E: (Exa-Byte),
  - P: (Peta-Byte),
  - T: (Terra-Byte),
  - G: (Giga-Byte),
  - M: (Mega-Byte),
  - K: (Kilo-Byte).
- Es können auch die Zweierpotenzäquivalente verwendet werden: Ei, Pi, Ti, Gi, Mi, Ki.
- Folgende Angaben bezeichnen als dieselbe Menge an Speicher!

**0.25GB = 250MB = 250000KB = 250000000**

### Achtung:

- 1KB = 1000 Bytes
- 1KiB = 1024 Bytes

Beides wird im Sprachgebrauch häufig etwas ungenau als „Kilobyte“ bezeichnet.

Für alle weiteren Einheiten analog!

Je größer die Einheiten, desto größer der absolute Unterschied!

$1GiB = 1024MiB = 1024 * 1024KiB = 1024 * 1024 * 1024 Bytes = 1.073.741.824$  (immerhin schon fast 1GB Unterschied zu 1GB)

# SCHEDULING CONSTRAINTS

## Node-Selektoren

Mittels Node-Labels kann man Nodes kennzeichnen, um z.B. den Scheduler bestimmte Workloads nur bestimmten Nodes zuweisen zu lassen.

Z.B. könnte einen Tensorflow-Workload nur auf Nodes Sinn machen, die das Label „gpu=nvidia“ haben.

### Beispiel:

Es könnten Nodes des Clusters nicht vollkommen homogen aufgebaut sein, sondern es könnten Nodes mit schnellen SSD-Platten und (aus Kostengründen) Nodes mit langsameren HDD-Platten konfiguriert sein.

Aus Geschwindigkeitsgründen könnte gewollt sein, dass ein Web-Server nur auf Nodes mit SSD-Platten gesetzt wird, um geringe Auslieferungszeiten von Platte bis Browser sicherstellen zu können.

Solche Scheduling-Randbedingung lassen sich mittels eines auf Pod-Ebene angegebenen Node-Selektors realisieren.

```
# Nodes lassen sich mittels Key-Value Paaren labeln,  
# z.B. so:  
kubectl label nodes node-1 disktype=ssd  
kubectl label nodes node-2 disktype=ssd  
kubectl label nodes node-3 disktype=hdd  
kubectl label nodes node-4 disktype=hdd
```

```
apiVersion: v1  
kind: Pod  
metadata:  
  name: nginx  
  labels:  
    env: test  
spec:  
  containers:  
  - name: nginx  
    image: nginx  
    imagePullPolicy: IfNotPresent  
  nodeSelector:  
    disktype: ssd
```

Man kann in K8S fast alle Ressourcen auf diese Weise labeln und selektieren!

Auch für die weiteren Workloads (Jobs, DaemonSets, StatefulSets) kann man Node-Selektoren in den entsprechenden Pod-Specs angeben.

# SCHEDULING CONSTRAINTS

## Affinities

- Node-Selektoren ermöglichen Pods auf Knoten mit bestimmten UND-verknüpften Labels hart zu beschränken.
- Das Affinitätskonzept erweitert die ausdrückbaren Randbedingungen erheblich:
  - Affinitäten sind ausdrucksvoller, da sie neben exakten (UND-verknüpften) Übereinstimmungen mehr Übereinstimmungsregeln bieten.
  - Es lässt sich bspw. angeben, dass Affinitäten zwar zu präferieren, aber keine harte Anforderung sind. Wenn der Scheduler Affinitäten nicht erfüllen kann, wird der Pod in diesen „weichen“ Fällen dennoch geplant.
  - Es können Affinitäten formuliert werden, die sich auf andere Pods (und deren Platzierung im Cluster) beziehen. So lassen sich bspw. Regeln festlegen, die vermeiden, dass Pods auf identische Nodes platziert werden (z.B. um Ausfallwahrscheinlichkeiten zu minimieren).
- Es gibt zwei Arten von Affinitäten:
  - **Knotenaffinität** ähnelt Node-Selektoren (jedoch mit den ersten beiden oben aufgeführten Vorteilen)
  - **Inter-Pod-Affinität / Anti-Affinität** wertet Pod-Labels und Pod-Platzierungen aus.



# SCHEDULING CONSTRAINTS

## Node Affinity

- Es gibt zwei Arten von Knotenaffinitäten
  - **requiredDuringSchedulingIgnoredDuringExecution** (harte Randbedingung, ähnlich Node-Selektor)
  - **preferredDuringSchedulingIgnoredDuringExecution** (weiche Randbedingung)
- Der Teil IgnoredDuringExecution bedeutet, dass der Pod nicht umgeplant wird, wenn sich zur Laufzeit Node-Labels ändern.
- RequiredDuringExecution würde bedeuten, dass auch Labeländerungen zur Laufzeit berücksichtigt würden. Dies wird aber aktuell vom K8S Scheduler nicht unterstützt.
- Ein Beispiel wäre Pods präferiert auf der eigenen On-Premise (provider=on-premise) Hardware auszuführen und nur in Overflow-Szenarien auf virtuelle Public IaaS-Maschinen (provider=aws oder provider=gce oder provider=azure) auszuweichen.

```
apiVersion: v1
kind: Pod
metadata:
  name: pod-with-constant-workload
spec:
  affinity:
    nodeAffinity:
      preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
              - key: provider
                operator: In
                values:
                  - on-premise
  containers:
    - name: constant-workload
      image: crunch.2.0
```

*Auch für die weiteren Workloads (Jobs, DaemonSets, StatefulSets) kann man Node-Affinities in den entsprechenden Pod-Specs vornehmen.*

*Dieser Workload würde präferiert im eigenen RZ platziert werden, es sei denn, dort findet der Scheduler keinen Platz mehr!*

*Dann würden ggf. teurere Public Cloud Maschinen genutzt werden.*

# SCHEDULING CONSTRAINTS

## Inter-Pod (Anti-)Affinities

- Mit Inter-Pod-(Anti-)Affinitäten lassen sich mittels Pod-Labels (und nicht anhand von Knoten-Labels) festlegen, welche Knoten für einen Pod geplant (oder vermieden) werden sollen.
- Die (Anti-)Affinitäts-Regeln haben die Form:  
*Dieser Pod sollte (nicht) auf Knoten X ausgeführt werden, wenn auf Knoten X bereits ein Pod ausgeführt wird, der Regel Y erfüllt.*
- Konzeptionell werden die Knoten X über eine Topologiedomäne wie Knoten, Rack, Cloud-Provider-Zone, Cloud-Provider-Region bestimmt.

```
apiVersion: apps/v1
kind: Deployment
metadata: name: redis-cache
spec:
  selector:
    matchLabels:
      app: store
  replicas: 3
  template:
    metadata:
      labels:
        app: store
    spec:
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            - labelSelector:
                matchExpressions:
                  - key: app
                    operator: In
                    values: ["store"]
              topologyKey: "kubernetes.io/hostname"
      containers:
        - name: redis-server
          image: redis:3.2-alpine
```



Dieses Beispiel platziert alle Redis Pods (In-Memory Cache) so, dass nie zwei Pods auf demselben Node landen.

Dies kann aus Gründen der gleichmäßigen Verteilung von Netzwerklasten und Ausfallsicherheit durchaus Sinn machen.

Auch für die weiteren Workloads (Jobs, DaemonSets, StatefulSets) kann man Inter-Pod-Affinities in den entsprechenden Pod-Specs vornehmen.

PROF.DR. NANEKRATZKE

59

59

# POD HEALTH

## Liveness Probes

- Viele Anwendungen, die über einen längeren Zeitraum ausgeführt werden, enden ggf. in einen fehlerhaften Zustand (z.B. in einem seltenen Deadlock) und können nur durch einen Neustart wiederhergestellt werden.
- Kubernetes bietet sogenannte Liveness Probes, um solche Situationen zu erkennen und zu beheben.
- Mittels Liveness Probes lässt sich ein sogenannter Heart Beat realisieren (I am still alive), den die K8S Plattform kontinuierlich prüft.
- Liveness Probes können pro Container in der **Container Spec** definiert werden.
- Definiert werden können
  - Command-basierte
  - HTTP-basierte
  - oder TCP-basierte Probes
- Diese Probes werden in definierten Zeitintervallen geprüft werden. Ist die Prüfung mehrmals (definierbar über einen Schwellwert) erfolglos (d.h. Fehlerfall) wird der Container neu gestartet.

```
livenessProbe:
  httpGet:
    path: /healthz
    port: 80
  failureThreshold: 3
  periodSeconds: 10
```

```
livenessProbe:
  tcpSocket:
    port: 3306
  failureThreshold: 3
  periodSeconds: 10
```

```
livenessProbe:
  exec:
    command:
      - cat
      - /tmp/healthy
  failureThreshold: 1
  periodSeconds: 10
```



HTTP-basierter Probe, Return Codes zwischen 200 und 399 werden als „Alive“ interpretiert. Alles andere als „Failure“.

TCP-basierter Probe auf Port 3306 (MySQL-Server). Ein erfolgreicher Aufbau wird dies als „Alive“ interpretiert. Alles andere als „Failure“.

Command-basierter Probe. Hierzu wird das Command im Container ausgeführt. Exit Code 0 wird als „Alive“ interpretiert. Alles andere als „Failure“.

PROF.DR. NANEKRATZKE

60

Quelle: <https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/>

60

# POD HEALTH

## Readiness und StartUp Probes

### Readiness Probe

- Manchmal können Anwendungen auch Requests nur vorübergehend nicht bedienen.
- Beispielsweise könnte eine Anwendung möglicherweise beim Start große Datenmengen oder Konfigurationsdateien laden oder nach dem Start von externen Diensten abhängig sein.
- In solchen Fällen sollte der Prozess nicht beendet werden, allerdings auch keine neuen Requests bekommen.
- Kubernetes bietet hierfür sogenannte Readiness Probes an, um solche Situationen zu erkennen und zu mildern.
- Ein Pod mit Containern, die mittels Readiness Probes melden, dass sie nicht bereit sind, empfängt keinen Datenverkehr über Kubernetes Services.
- Readiness Probes werden grundsätzlich so definiert, wie auch Liveness Probes.

```
readinessProbe:  
  exec:  
    command:  
    - cat  
    - /tmp/ready  
  failureThreshold: 1  
  periodSeconds: 10
```

*auch HTTP und TCP-  
Probes möglich*

### StartUp Probe

- Insbesondere bei sogenannten Legacy Applikationen können für erste Initialisierungen möglicherweise zusätzliche Startzeit erforderlich sein.
- In solchen Fällen kann es schwierig sein, Parameter für die Liveness Probes einzurichten.
- Hierzu kann ein StartUp Probe eingerichtet werden. Dieser wird erst geprüft. Dieser StartUp Probe hat üblicherweise einen höheren Fehlerschwellenwert und längere Prüfperiode um die Startzeit im schlimmsten Fall abzudecken.
- Der unten stehende Startup-Probe deckt Startup-Phasen von bis zu 5 Minuten ab, und würde erst dann die Readiness- und Liveness Prüfung vornehmen.

```
startupProbe:  
  tcpSocket:  
    port: 3306  
  failureThreshold: 10  
  periodSeconds: 30
```

*auch HTTP und  
Command-Probes  
möglich*

*Auch für die weiteren  
Workloads (Jobs,  
DaemonSets, StatefulSets)  
kann man Probes in den  
entsprechenden Container-  
Specs vornehmen.*

# POD HEALTH

## Konfiguration von Probes

### Allgemeine Konfiguration von Probes:

- **initialDelaySeconds:**  
Anzahl der Sekunden nach dem Start des Containers, bevor Liveness- oder Readiness-Tests gestartet werden.
- **periodSeconds:**  
Wie oft (in Sekunden) der Probe gecheckt wird.
- **timeoutSeconds:**  
Anzahl der Sekunden, nach denen ein Probe antworten muss.
- **successThreshold:**  
Minimale aufeinanderfolgende Erfolge, damit ein Probe nach einem Fehler als erfolgreich angesehen wird.
- **failThreshold:**  
Anzahl an Wiederholungsversuchen, bevor ein Probe als nicht bereit interpretiert wird.

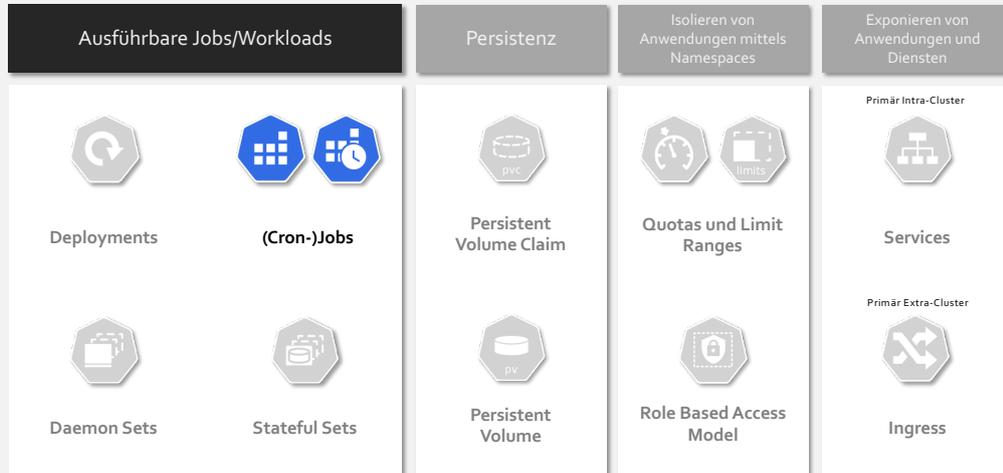
### Weitere Optionen für HTTP-Probes:

- **host:**  
Hostname, zu dem eine Verbindung hergestellt werden soll, standardmäßig die Pod-IP.
- **schema:**  
Schema für die Verbindung zum Host (HTTP oder HTTPS). Der Standardwert ist HTTP.
- **pfad:**  
Pfad für den Zugriff auf dem HTTP-Server.
- **httpHeaders:**  
Benutzerdefinierte Header, die in der Anforderung festgelegt werden sollen.
- **port:**  
Name oder Nummer des Ports, auf den im Container zugegriffen werden soll. Die Nummer muss im Bereich von 1 bis 65535 liegen.

*Auch für die weiteren  
Workloads (Jobs,  
DaemonSets, StatefulSets)  
kann man die  
Konfiguration von Probes  
in den entsprechenden  
Container-Specs  
vornehmen.*

# KUBERNETES

Überblick über die wichtigsten Kubernetes-Ressourcen



63

## JOBS

Ausführbare Workloads

Jobs sind meist länger laufende Aufgaben (z.B. das Trainieren eines neuronalen Netzes), die einmalig ausgeführt werden.

Ein Job erzeugt hierzu ein oder mehrere Pods und trägt Sorge dafür, dass eine vorgegebene Anzahl von diesen erfolgreich terminiert. Es können mehrere Pods parallel ausgeführt werden.

Wird ein Job gelöscht, werden auch alle von ihm erzeugten Pods gelöscht.

```
# Ausbringen eines Deployments
# (z.B. zur Berechnung der ersten 2000 Nachkommastellen von PI)
kubectl apply -f xample-job.yaml

# Holen des Berechnungsergebnisses
kubectl logs -l job-name=pi

# Löschen des Jobs
kubectl delete jobs/pi
```

```
apiVersion: batch/v1
kind: Job
metadata:
  name: pi
spec:
  template:
    spec:
      containers:
      - name: pi
        image: perl
        command: ["perl", "-Mbignum=bpi", "-wle", "print bpi(2000)"]
        restartPolicy: Never # Container nie neu starten!
```

Manifest: xample-job.yaml

Quelle: <https://kubernetes.io/docs/concepts/workloads/controllers/job/>

64

# CRON-JOBS

## Ausführbare Workloads

Cron-Jobs sind periodisch auszuführende Aufgaben (z.B. Erstellung einer Monatsabrechnung).

Die Periodizität von Jobs wird dabei nach dem gleichen Schema angegeben, dass auch von Linux/UNIX-Cron-Jobs bekannt ist (\* \* \* \* \*).

Ein Cron-Job erzeugt dazu zu den so festgelegten Zeitpunkten Jobs, die wie beschrieben ausgeführt werden.

```
apiVersion: batch/v1beta1
kind: CronJob
metadata:
  name: hello
spec:
  schedule: "* * * * *"
  jobTemplate:
    spec:
      template:
        spec:
          containers:
            - name: hello
              image: busybox
              args:
                - /bin/sh
                - -c
                - date; echo Hello from the Kubernetes cluster
              restartPolicy: OnFailure
```

Manifest: xample-cron-job.yaml

*Dieser Beispiel-Cronjob gibt alle fünf Minuten die aktuelle Uhrzeit und eine Hallo-Nachricht aus.*

Quelle: <https://kubernetes.io/docs/concepts/workloads/controllers/cron-jobs/>

# KUBERNETES

## Überblick über die wichtigsten Kubernetes-Ressourcen

Ausführbare Jobs/Workloads		Persistenz	Isolieren von Anwendungen mittels Namespaces	Exponieren von Anwendungen und Diensten
 Deployments	 (Cron-)Jobs	 Persistent Volume Claim	 Quotas und Limit Ranges	 Services
 Daemon Sets	 Stateful Sets	 Persistent Volume	 Role Based Access Model	 Ingress

# DAEMON SETS

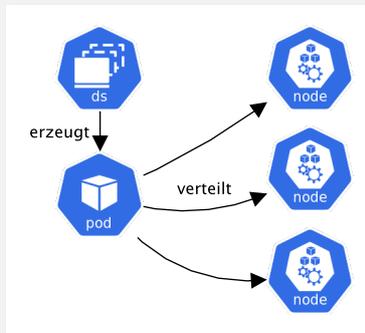
## Ausführbare Workloads

Ein Daemon Set stellt sicher, dass auf allen (oder ausgewählten) Nodes der Kubernetes Infrastruktur eine Kopie eines Pods läuft.

Werden Nodes dem Cluster hinzugefügt, werden so auch Pods auf diesen automatisch ausgeführt. Werden Nodes entfernt, werden entsprechende Pods durch den Garbage Collector entfernt.

Typische Anwendungsfälle sind häufig zentrale Dienste des Clusters (Kubernetes Add-Ons) wie bspw.:

- Clustered Storage Daemons pro Cluster Node
- Log Collection Daemon pro Cluster Node
- Node Monitoring Daemon pro Cluster Node
- Ingress Controller pro Cluster Node



Quelle: <https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/>

# DAEMON SETS

## Ausführbare Workloads

Beispiel eines DaemonSets, das einen fluentd Log-Konsolidator Pod auf allen Nodes ausführt, um Pod Logs in einem zentralen Logging-Store zu konsolidieren und dort mittels Elasticsearch durchsuchbar zu hinterlegen (nicht Gegenstand dieses Manifests).

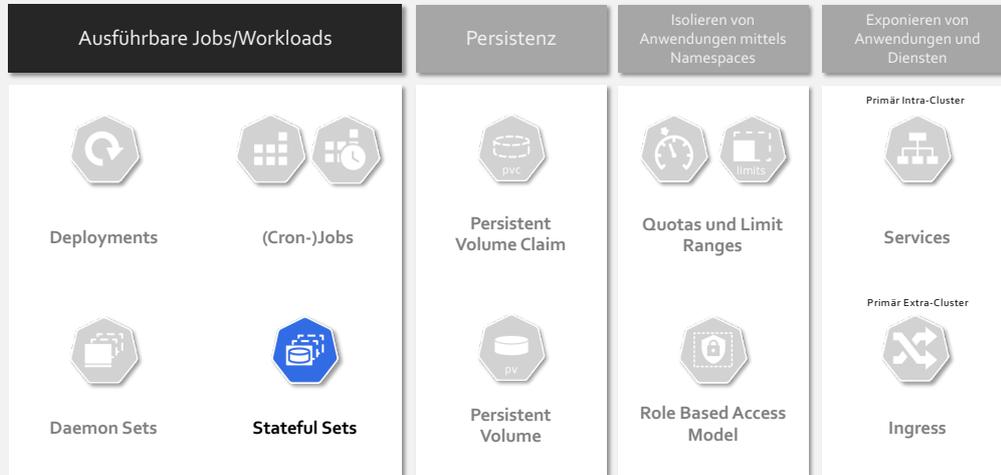
```
# Ausbringen einer Cluster-weiten  
# Log-Aggregation auf Basis  
# von fluentd  
kubectl apply -f fluentd-daemon.yaml  
  
# Löschen der Log-Aggregation  
kubectl delete -f fluentd-daemon.yaml
```

```
apiVersion: apps/v1  
kind: DaemonSet  
metadata:  
  name: fluentd-elasticsearch  
  namespace: kube-system  
spec:  
  selector:  
    matchLabels:  
      name: fluentd-elasticsearch  
  template:  
    metadata:  
      labels:  
        name: fluentd-elasticsearch  
    spec:  
      containers:  
        - name: fluentd-elasticsearch  
          image: fluentd:v2.5.2  
          volumeMounts:  
            - name: varlog  
              mountPath: /var/log  
            - name: varlibdockercontainers  
              mountPath: /var/lib/docker/containers  
              readOnly: true  
      volumes:  
        - name: varlog  
          hostPath:  
            path: /var/log  
        - name: varlibdockercontainers  
          hostPath:  
            path: /var/lib/docker/containers
```

Manifest: fluentd-daemon.yaml

# KUBERNETES

Überblick über die wichtigsten Kubernetes-Ressourcen



69

## STATEFUL SETS

Ausführbare Workloads

Stateful Sets dienen dem Ausbringen, Aktualisieren und Skalieren verteilter „Stateful Applications“.

Solche Applikationen sind häufig Datenbanken; z.B. etcd (ein verteilter konsistenter Key-Value Store).

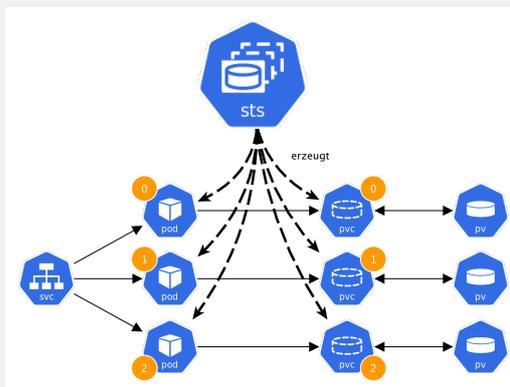
Dabei wird die Reihenfolge der Pod-Erzeugung und Eindeutigkeit der erzeugten Pods garantiert (Sticky Identity).

Obwohl Pods aus derselben Spec generiert werden, haben diese eine persistente Identität, die auch Reschedulings (z.B. im Rahmen von Aktualisierungen) überdauert.

Eine Besonderheit besteht bei Volume Mounts.

Stateful Sets können mittels einem **Volume Claim Template** pro Pod einen jeweils eigenen Volume Claim mit derselben Ordinalnummer wie der Pod erzeugen. Auf diese Weise, können Volumes bei einem Update des Pods wieder einem Pod mit derselben Ordinalnummer zugewiesen werden.

Auf diese Weise können auch Stateful Applikationen wie bspw. Datenbanken aktualisiert und skaliert werden.



Quelle: <https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/>

70

# STATEFUL SETS

## Ausführbare Workloads

Beispiel einer auf einem StatefulSet basierenden ETCD-Datenbank.

ETCD ist ein verteilter auf dem RAFT-Algorithmus beruhender Key-Value Store, der bspw. von Kubernetes Master-Nodes genutzt wird, um einen ausfallsicheren und konsistenten Zustand über alle Master-Nodes herzustellen und zu erhalten.

```
# Ausbringen
kubectl apply -f etcd.yaml

# Hochskalieren (wird zwei weitere Pods
# inkl. Volumes 3 + 4 erzeugen)
kubectl scale --replicas=5 -f etcd.yaml

# Runterskalieren (wird nur die
# Pods 3 + 4 löschen, aber nicht die
# Volumes 3, 4!)
kubectl scale --replicas=3 -f etcd.yaml

# Erneutes Hochskalieren (Noch bestehende
# Volumes 3 und 4 werden wieder Pod 3 und 4
# zugeordnet, Pod 5 und Vol 5 neu erzeugt!)
kubectl scale --replicas=6 -f etcd.yaml
```

```
apiVersion: apps/v1
kind: StatefulSet
metadata:
  name: etcd
  labels: { app: etcd }
spec:
  serviceName: etcd
  replicas: 3
  selector:
    matchLabels: { app: etcd }
  template:
    metadata:
      name: etcd
      labels: { app: etcd }
    spec:
      containers:
        - name: etcd
          image: etcd:3.2.24
          ports:
            - { containerPort: 2380, name: peer }
            - { containerPort: 2379, name: client }
          env:
            - { name: INITIAL_CLUSTER_SIZE, value: 3 }
            - { name: SET_NAME, value: etcd }
          volumeMounts:
            - name: datadir
              mountPath: /var/run/etcd
  volumeClaimTemplates:
    - metadata: { name: datadir }
      spec:
        accessModes: ["ReadWriteOnce"]
        resources:
          requests: { storage: 1Gi }
```

# KUBERNETES

## Überblick über die wichtigsten Kubernetes-Ressourcen

Ausführbare Jobs/Workloads		Persistenz	Isolieren von Anwendungen mittels Namespaces	Exponieren von Anwendungen und Diensten
 Deployments	 (Cron-)Jobs	 Persistent Volume Claim	 Quotas und Limit Ranges	 Services
 Daemon Sets	 Stateful Sets	 Persistent Volume	 Role Based Access Model	 Ingress

# PERSISTENZ

## Organisation persistenten Speichers in Kubernetes

Das Verwalten von persistentem Speicher ist ein anderes Problem als das Verwalten von flüchtigen Recheninstanzen.

Das Persistenz-Subsystem von Kubernetes stellt hierzu eine API für Benutzer und Administratoren bereit, die Details zur Bereitstellung von Speicher von der Art seiner Verwendung abstrahiert.

Dies erfolgt mittels zweier API-Ressourcen namens **PersistentVolume** und **PersistentVolumeClaim**.

### Persistent Volume (PV)

- Ein PV repräsentiert persistenten Speicher (Festplatte) im Cluster, der von einem Administrator bereitgestellt oder mithilfe von Speicherklassen dynamisch erzeugt wurde.
- Es ist wie ein Knoten eine normale statische Ressource des Clusters. PVs haben einen Lebenszyklus, der von einem Pod unabhängig ist.
- Dieses API-Objekt erfasst die Details der Implementierung des Speichers, sei es NFS, iSCSI oder ein Cloud-Provider-spezifisches Speichersystem.

### Persistent Volume Claim (PVC)

- Ein PVC ist eine Anforderung von persistentem Speicher (Festplatte) durch einen Benutzer.
- Es ist analog einem Pod, der CPU- und Memory-Ressourcen eines Knotens benötigt. PVCs verbrauchen PV-Ressourcen.
- Mittels PVCs können bestimmte QoS-, Größen- und Zugriffsmodi von persistentem Speicher angefordert werden (z. B. können sie ReadWriteOnce, ReadOnlyMany oder ReadWriteMany bereitgestellt werden).

Für K8S gibt es u.a. die folgenden Block-Storage und Filesystem Provisioner:

- AWS Elastic Block Store
- Azure File
- Azure Disk
- Ceph (FS + RBD)
- Cinder (OpenStack)
- FC (Fibre Channel)
- Flex Volume
- Flocker
- GCE Persistent Disk
- GlusterFS
- iSCSI
- Quobyte
- NFS
- vsphere volume
- Portworx volume
- Scale IO
- Storage OS
- Local
- ...

Quelle: <https://kubernetes.io/docs/concepts/storage/persistent-volumes/>

# PERSISTENT VOLUME

## Bekanntmachung von persistentem Speicher durch den Administrator

Beispiel eines mittels NFS bereitgestellten Persistent Volumes.

Werden PVs auf diese Weise statisch dem Cluster bekannt gemacht, erfolgt dies dann normalerweise durch den Administrator. Dieser muss auch Details der Infrastruktur kennen, wie bspw. IP-Adresse des NFS-Servers, Kapazität der Platte, usw.

PVs werden daher meist nicht durch User im System angelegt.

Für User werden solche PVs meist mittels PVCs durch Storage Provisioner (Storage Classes) bereitgestellt.

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0003
spec:
  capacity:
    storage: 5Gi
  volumeMode: Filesystem
  accessModes:
  - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: slow
  mountOptions:
  - hard
  - nfsvers=4.1
  nfs:
    path: /tmp
    server: 172.17.0.2
```

# PERSISTENT VOLUME CLAIM

Anforderung und mounten von persistentem Speicher durch den User

## Anforderung von persistentem Speicher

User können persistenten Speicher mittels PVCs anfordern. Hierzu muss eine Storage-Klasse angegeben werden, aus dem der Speicher bedient werden soll (auf die Angabe einer Storage Klasse kann verzichtet werden, wenn es eine Default Storage Klasse im Cluster gibt).

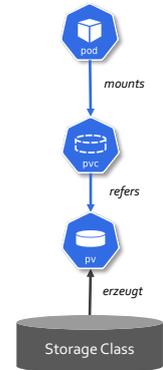
```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: myclaim
spec:
  accessModes: ["ReadWriteOnce"]
  volumeMode: Filesystem
  resources:
    requests: { storage: 8Gi }
  storageClassName: slow
```

## Claims as Volumes

Pods greifen auf den so angeforderten persistenten Speicher zu, indem sie den PVC als Volume im Pod mounten. PVCs müssen sich hierzu im selben Namespace wie der Pod befinden.

Der Cluster nutzt das PVC im Namespace des Pods und verwendet es, um ein Persistent Volume zu erzeugen, welches den Anforderungen des PVC genügt. Das Persistent Volume wird dann auf dem Host und im Pod bereitgestellt.

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: nginx
      volumeMounts:
        - mountPath: "/var/www/html"
          name: mypd
  volumes:
    - name: mypd
      persistentVolumeClaim: { claimName: myclaim }
```



Storage Classes und Storage Provisioner werden nicht behandelt.

# KUBERNETES

Überblick über die wichtigsten Kubernetes-Ressourcen

Ausführbare Jobs/Workloads	Persistenz	Isolieren von Anwendungen mittels Namespaces	Exponieren von Anwendungen und Diensten
Deployments (Cron-)Jobs	Persistent Volume Claim	Quotas und Limit Ranges  Role Based Access Model	Primär Intra-Cluster Services  Primär Extra-Cluster Ingress
Daemon Sets Stateful Sets	Persistent Volume		

# QUOTAS UND LIMIT RANGES

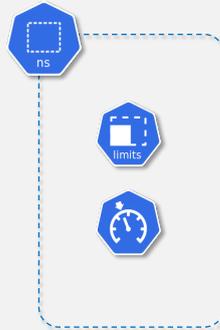


Isolieren von Anwendungen mittels Namespaces

Mit Ressourcen-Kontingenten (**ResourceQuota**) kann der Ressourcenverbrauch pro Namespace beschränkt werden.

Um zu vermeiden, dass einzelne Pods oder Container alle verfügbaren Ressourcen monopolisieren, kann zusätzlich die Ressourcenzuweisung pro Container limitiert bzw. mit default Werten zugewiesen werden (**LimitRange**).

Hierzu müssen einfach nur die entsprechenden Manifeste in einem Namespace angelegt werden.



```
Manifest: compute-quota.yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: compute-resources
spec:
  hard:
    limits.cpu: "2"
    limits.memory: 2Gi
    requests.nvidia.com/gpu: 4
```

kubectl apply -f compute-quota.yaml -n xample

```
Manifest: cpu-limits.yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: cpu-limits
spec:
  limits:
  - type: Container
    max: { cpu: "1000m" }
    min: { cpu: "200m" }
    default: { cpu: "200m" }
```

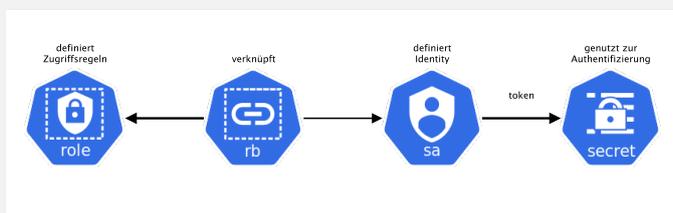
kubectl apply -f cpu-limits.yaml -n xample

Quotas: <https://kubernetes.io/docs/concepts/policy/resource-quotas/>  
LimitRanges: <https://kubernetes.io/docs/concepts/policy/limit-range/>

# ROLE BASED ACCESS MODEL



Isolieren von Anwendungen mittels Namespaces



In Kubernetes können Namespaces zur Isolation von Workloads angelegt werden.

Mittels Serviceaccounts und Zugriffsregeln können innerhalb eines Namespaces komplexe Rechtesysteme realisiert werden.

Jeder per kubectl angelegte Namespace erhält automatisch einen default Serviceaccount mit einem generierten Access Token zur Authentifizierung. Weitere Serviceaccounts können angelegt werden.

**Roles** weisen Rechte in einem Namespace zu (für den diese Rolle erzeugt wurde).

**ClusterRoles** weisen Rechte Namespace-übergreifend auf Cluster-Ebene zu.

Das Anlegen von ClusterRoles erfolgt analog zu Roles.

Quelle: <https://kubernetes.io/docs/reference/access-authn-authz/rbac/>

# ROLE BASED ACCESS MODEL



## Isolieren von Anwendungen mittels Namespaces

Dieses Beispiel zeigt, wie sich Leserechte in Kubernetes dem default Serviceaccount und volle Schreibrechte dem Admin Serviceaccount zuweisen lassen.

```
# Das Erzeugen eines Namespaces (mit kubectcl) erzeugt auch
# immer automatisch einen default Serviceaccount (sa)
kubectcl create namespace xample
kubectcl get sa -n xample

NAME      SECRETS  AGE
default   1         6m2s

# Jedem Serviceaccount ist ein Token zur Authentifizierung
# zugeordnet (bspw. in der kubeconfig von kubectcl nutzbar)
TOKEN=kubectcl get sa default -o=jsonpath="{.secrets[0].name}"
kubectcl get secret $TOKEN -n xample -o jsonpath="{.data.token}"

# Es können beliebig weitere Serviceaccounts in einem
# Namespace angelegt werden, z.B. ein Admin Account
kubectcl create sa admin -n xample

# Anlegen von Rollen
kubectcl apply -f read-role.yaml
kubectcl apply -f rw-role.yaml

# Zuordnen von Rollen zu Accounts
kubectcl create rolebinding read-binding --role=read-role \
--serviceaccount=xample:default -n=xample
kubectcl create rolebinding rw-binding --role=rw-role \
--serviceaccount=xample:admin -n=xample
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: read-role
rules:
- apiGroups: ["" ] # "" => core API group
  resources: [ "*" ]
  verbs: [ "get", "watch", "list" ]
```

Manifest: read-role.yaml

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: rw-role
rules:
- apiGroups: [ "*" ]
  resources: [ "*" ]
  verbs: [ "*" ]
```

Manifest: rw-role.yaml

# KUBERNETES

## Überblick über die wichtigsten Kubernetes-Ressourcen



# SERVICES

## Exponieren von Anwendungen und Diensten

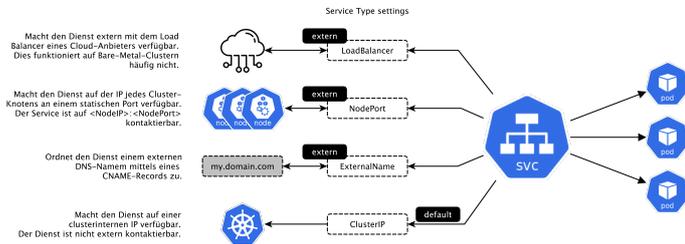
### Definieren von Diensten mittels Pod-Labels und Selektoren

Dienste ermöglichen es eine Anwendung, die auf einer Reihe von Pods ausgeführt wird, als Netzwerkdienst unter einem **DNS-Namen** verfügbar zu machen.

Hierzu werden die Pods eines Services mittels **Selektoren** zu machen.

Kubernetes gibt Pods ihre eigenen **IP-Adressen** und Services einen eindeutigen **DNS-Namen**.

Pods, die über einen Service angesprochen werden unterliegen automatisch einem **Load Balancing** durch kube-proxy.



### Exponieren von Diensten mittels Service Types

Dienste werden per default nur Cluster-intern bereitgestellt. Sie können aber mittels des **serviceType** Feldes Cluster-extern exponiert werden:

- **Load Balancer** bindet einen Dienst an eine Public IP eines Cloud Providers
- **NodePort** macht einen Service-Port auf allen IP-Adressen der Cluster Nodes bekannt.
- Mittels **ExternalName** können externe Dienste in den Cluster eingebunden und wie Cluster-interne Dienste genutzt werden.

Die Exponierung mittels funktioniert meist nur in Public IaaS-provisionierten Clustern.

Im bereitgestellten MicroK8S-basierten K8S-Cluster können Sie Dienste nur mittels HTTP-basierten Ingress exponieren (dazu gleich mehr).

**Hintergrund:** LoadBalancer benötigen eine Public-IP-Adresse für jeden exponierten Dienst (diese sind mittlerweile recht rar für IP-V4).

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

# SERVICES

## Exponieren von Anwendungen und Diensten

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
            - containerPort: 8080
            - containerPort: 8443
```

Manifest: nginx-deployment.yaml

```
apiVersion: v1
kind: Service
metadata:
  name: http-service
spec:
  selector:
    app: nginx
  ports:
    - name: http
      protocol: TCP
      port: 80
      targetPort: 8080
    - name: https
      protocol: TCP
      port: 443
      targetPort: 8443
```

Manifest: nginx-service.yaml

```
# Ausbringen des Deployments
kubectl apply -f nginx-deployment.yaml

# Ausbringen eines Services
kubectl apply -f nginx-service.yaml

# Auflisten aller Services
kubectl get svc

# Details zu einem Service
kubectl describe http-service

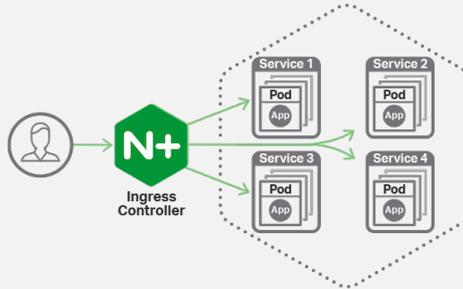
# Löschen eines Services
kubectl delete -f nginx-service.yaml

# Services mit der Kommandozeile erzeugen
# (funktioniert auch für andere
# Ressourcen)
kubectl create svc --help
```

# INGRESS

## Exponieren von Service APIs via HTTP(S)

- Ein Ingress macht HTTP- und HTTPS-Routen von außerhalb des Clusters für Dienste innerhalb des Clusters verfügbar.
- Ein Ingress kann so konfiguriert werden, dass für Dienste extern erreichbare URLs bereitgestellt (inkl. Load Balancing, SSL/TLS Termination und namenbasiertes virtuelles Hosting) angeboten werden.
- Das Routing wird hierzu durch Regeln gesteuert, die in der Ingress-Ressource definiert sind.
- Wenn Dienste mittels anderer Protokolle als HTTP(S) exponiert werden sollen, muss normalerweise ein Service vom Typ NodePort oder LoadBalancer verwendet werden.



Wieso nur HTTP(S) und nicht beliebige Protokolle?

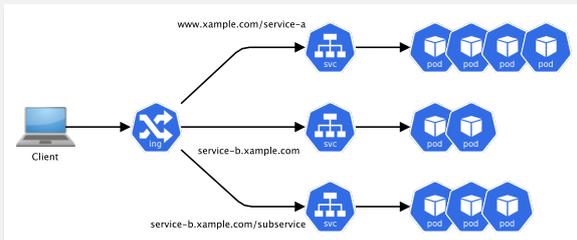
REST-basierte APIs sind einfach eine verbreitete Art und Weise Schnittstellen zu Cloud-nativen Dienste bereitzustellen.

Beliebige andere TCP/UDP basierte Protokolle können mittels **Load Balancern** über Services Cluster-extern bekannt gemacht werden.

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

# INGRESS

## Exponieren von Anwendungen und Diensten



Insbesondere HTTP-basierte Dienste (z.B. REST-basierte APIs) werden meist mittels Ingress Ressourcen für Zugriffe von außerhalb des Clusters bereitgestellt.

Anders als Service-basierte NodePort oder LoadBalancer Exponierungen (die pro Exponierung jeweils eine eindeutige IP-Adresse+Portnummer Kombination benötigen) braucht ein Ingress nur eine einzige IP-Adresse und eine Portnummer, um eine Vielzahl an Diensten zu exponieren. Insbesondere Public-IP4 Adressen sind mittlerweile ein knappes Gut geworden.

Ingresses laufen ferner auf Anwendungsschicht des Netzwerkstapels (HTTP) und bieten daher Funktionen wie bspw. Cookie-gestützte Sitzungsaffinitäten und können ferner mittels TLS-Zertifikaten abgesichert werden (HTTPS).

```
kind: Ingress
apiVersion: extensions/v1beta1
metadata:
  name: xample-ingress
spec:
  rules:
  - host: www.xample.com
    http:
      paths:
      - path: /service-A
        backend:
          serviceName: service-a
          servicePort: 8080
  - host: service-b.xample.com
    http:
      paths:
      - path: /
        backend:
          serviceName: service-b
          servicePort: 80
      - path: /subservice
        backend:
          serviceName: service-b-subservice
          servicePort: 8888
```

Manifest: xample-ingress.yaml

## ZUM NACHLESEN



### Teil I: Überblick

1. Einführung in Kubernetes
2. Erste Schritte mit Docker und Kubernetes

### Teil II: Grundlagen

3. Pods: Container in Kubernetes ausführen
4. Replikationscontroller
5. Dienste: Pods finden und Ihnen kommunizieren
6. Volumes: Festplattenspeicher zur Containern hinzufügen
7. Konfigurationszuordnungen und Secrets: Anwendungen konfigurieren
9. Deployments: Anwendungen deklarativ aktualisieren
10. StatefulSets: Replizierte statusbehaftete Anwendungen bereitstellen

### Teil III: Fortgeschrittene Themen

14. Die Computerressourcen eines Pods verwalten

85

## ZUM NACHLESEN

### Paper

- o Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. **Dominant resource fairness: fair allocation of multiple resource types.** <https://dl.acm.org/doi/10.5555/1972457.1972490>
- o Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. **Mesos: a platform for fine-grained resource sharing in the data center.** <https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf>
- o Reiss et. al., **Heterogeneity and Dynamicity of Clouds at Scale: Google Trace Analysis**, 2012 <https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf>
- o Schwarzkopf et al. **Omega: flexible, scalable schedulers for large compute clusters**, SIGOPS European Conference on Computer Systems (EuroSys), ACM, Prague, Czech Republic (2013) <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/41684.pdf>
- o Benjamin Hindman, **Practical Considerations for Multi-Level Schedulers** (Keynote, 19th Workshop on Job Scheduling Strategies for Parallel Processing), 2015 <https://www.cs.huji.ac.il/~feit/parsched/jssp15/p0-hindman.pdf>
- o Verma et al., **Large-scale cluster management at Google with Borg**, Proceedings of the European Conference on Computer Systems (EuroSys), ACM, Bordeaux, France (2015) <https://storage.googleapis.com/pub-tools-public-publication-data/pdf/43438.pdf>
- o Burns, Brendan, et al. **"Borg, omega, and kubernetes."** Queue 14.1 (2016): 70-93. <https://research.google/pubs/pub44843.pdf>



86

## KONTAKT

*Disclaimer*

**Nane Kratzke**  +49 451 300-5549  
 nane.kratzke@th-luebeck.de  
 kratzke.mylab.th-luebeck.de



PROF. DR.  
NANE KRATZKE

87