CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Container-Orchestrierung

(1) Scheduling in Abgrenzung zur Orchestrierung

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CC0 1.0 Lizenz** bereitgestellt.

Sie dürfen die Inhalte also kopieren, verändern, verbreiten, mit eigenen Inhalten mixen, auch zu kommerziellen Zwecken, und ohne um weitere Erlaubnis bitten zu müssen.

Eine Nennung des Autors ist nicht erforderlich (aber natürlich gern gesehen, wenn problemlos möglich).

Diese Folien sind insb. für die Lehre an Hochschulen konzipiert und machen daher vom **§51 UrhG (Zitate)** Gebrauch.

Die CC0 Lizenz überträgt sich nicht auf zitierte Quellen. Hier sind bei der Nutzung natürlich die Bedingungen der entsprechenden Quellen zu beachten.

Die Quellenangaben finden sich auf den entsprechenden Folien.

KAPITEL 9

Container-Plattformen

9.1 Scheduling
- Heterogenität von Workloads
- Scheduling-Algorithmen
- Scheduling-Architekturen

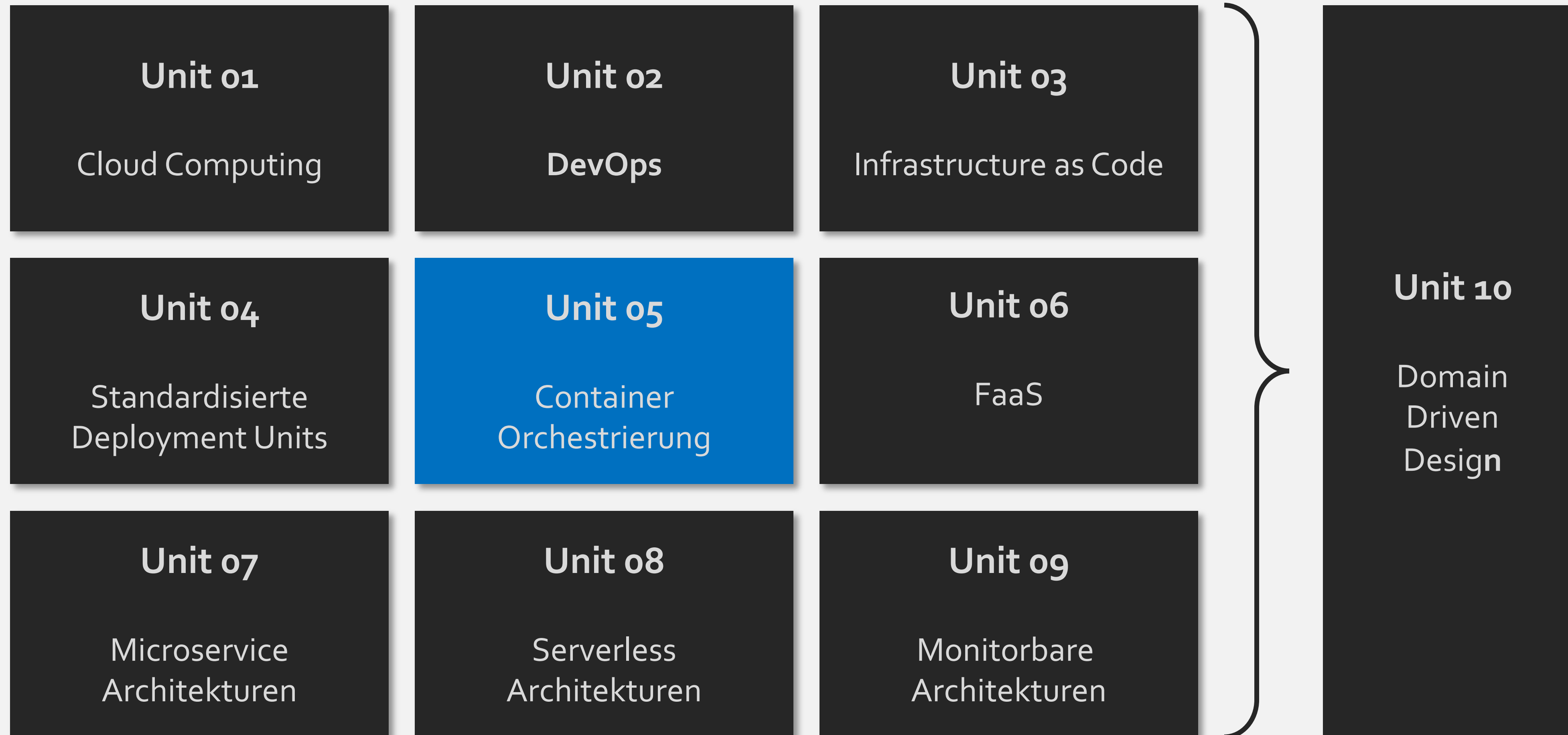
9.2 Orchestrierung
- Definition von Betriebszuständen
- Regelkreis: Desired vs Current State

9.3 Inside Kubernetes
- Kubernetes-Architektur und Ressourcen
- Workloadarten
- Scheduling Constraints
- Automatische Skalierung von Workloads
- Exponierung von Services
- Health Checking
- Persistenz
- Isolation von Workloads

9.4 Zusammenfassung

INHALTSVERZEICHNIS

Überblick über Units und Themen dieses Moduls



CLOUD NATIVE COMPUTING

INHALTE

Scheduling

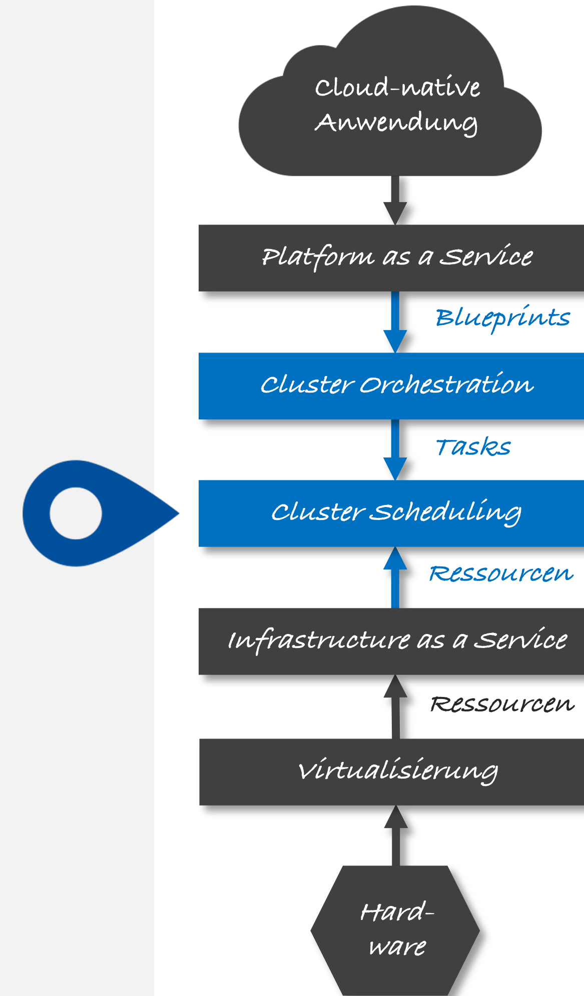
- Scheduling Problem Definition
- Scheduling Algorithmen
- Scheduler Architekturen
- Beispiele von Cluster Schedulern: Mesos, Swarm

Orchestrierung

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

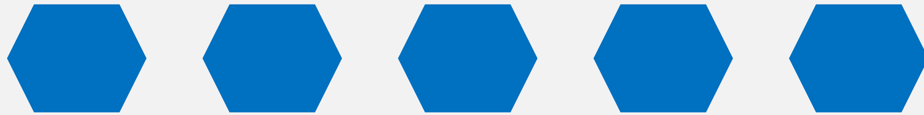
Inside Kubernetes (Typ-Vertreter)

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

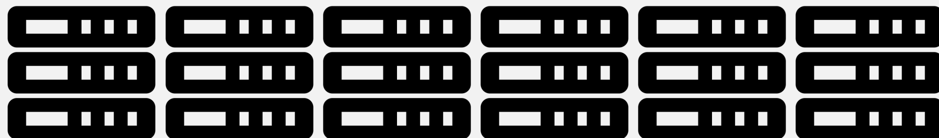


CLUSTER SCHEDULING

Das Problem



Rechenaufgaben
(Jobs/Workloads)



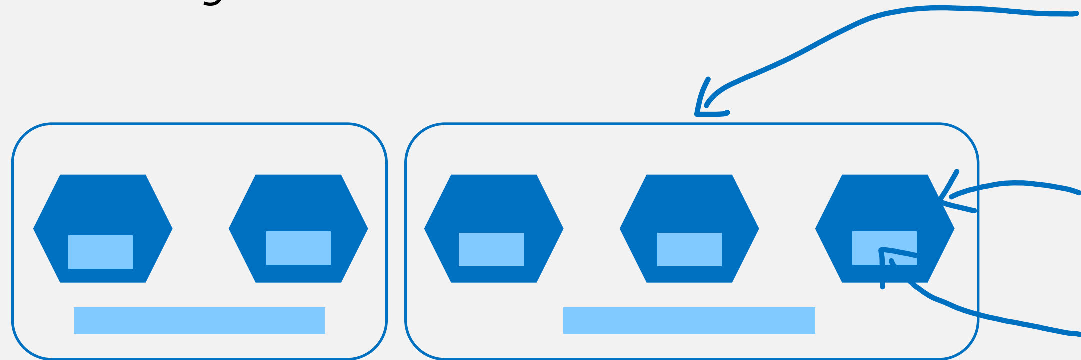
Rechenressourcen
(z.B. per IaaS oder GRID)

Wie möglichst Ressourcen-effizient und Aufgaben-angemessen zuteilen?

CLOUD NATIVE COMPUTING

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.

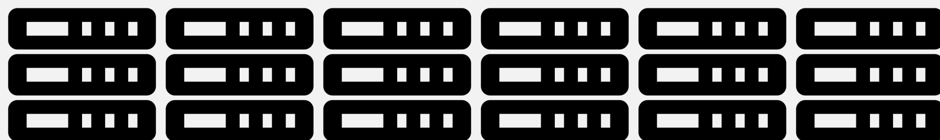
Task:
Atomare Rechenaufgabe inkl. Ausführungsvorschrift.

Properties:
Ausführungsrelevante Eigenschaften der Jobs und Tasks

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

?

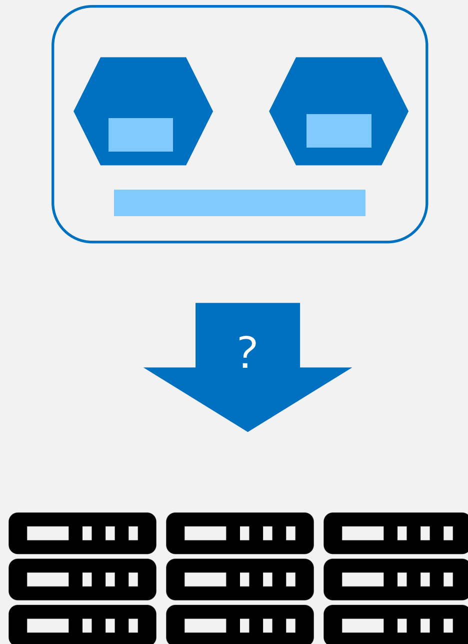
Scheduler:
Ausführung von Tasks auf den verfügbaren Ressourcen unter Berücksichtigung der Properties und zu optimierender Scheduling-Ziele (z.B. Fairness, Durchsatz, Ressourcenauslastung). Ein Scheduler kann präemptiv sein, also Tasks unterbrechen und neu aufsetzen.



Ressourcen:
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 allokieren.

Job Execution:
Einen Workload zuverlässig ausführen und dabei isolieren und überwachen.

SCHEDULING

Statische Partitionierung

Vorteil:

- Einfach zu realisieren

Nachteil:

- Nicht flexibel bei sich ändernden Bedürfnissen
- Geringe Auslastung
- Hohe Opportunitäts-kosten

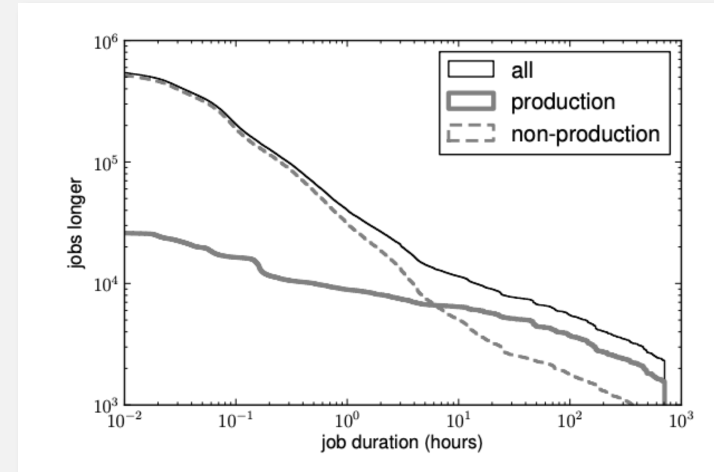
Benjamin Hindman: „Static partioning considered harmful!“

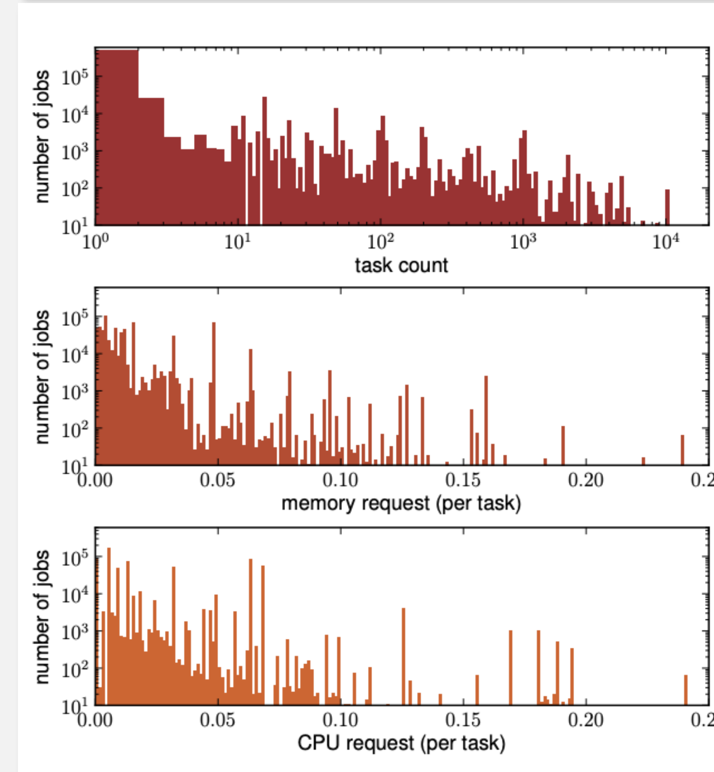
Quelle: Benjamin Hindman, Practical Considerations for Multi-Level Schedulers (Keynote, 19th Workshop on Job Scheduling Strategies for Parallel Processing, 2015) https://www.cs.huji.ac.il/~feit/parsched/jsspp15/p0-hindman.pdf

SCHEDULING

Hetereogenität von Workloads

- In typischen Clustern sind Jobs (und deren Workloads) 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.





CLOUD NATIVE COMPUTING

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 Dynamicity of Clouds at Scale: Google Trace Analysis, 2012
https://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf

SCHEDULING

Awareness + Request ⟶ Optimierung ⟶ Placement Decision

Awareness über Jobs/Tasks (Properties) und Ressourcen

- **Ressource:** 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:

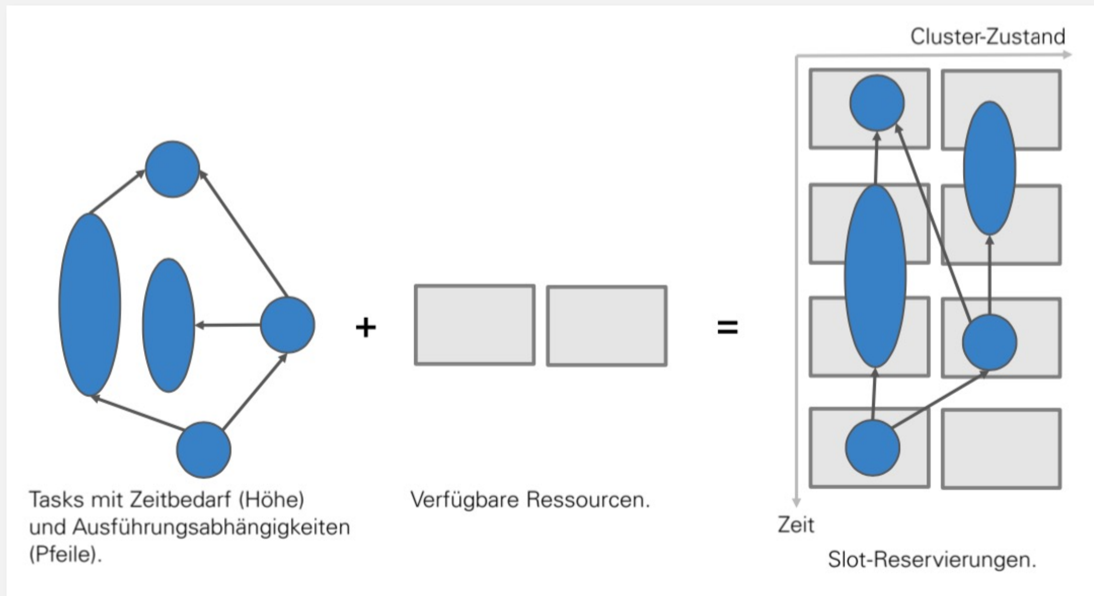
- **Fairness:** Kein Task soll unverhältnismäßig lange warten müssen, während ein anderer bevorzugt wird
- **Maximaler Durchsatz:** So viele Tasks pro Zeiteinheit wie möglich
- **Minimale Wartezeit:** Möglichst geringe Zeit von der Übermittlung bis zur Ausführung eines Tasks
- **Ressourcen-Auslastung:**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



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

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| x |Ressourcen|).

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

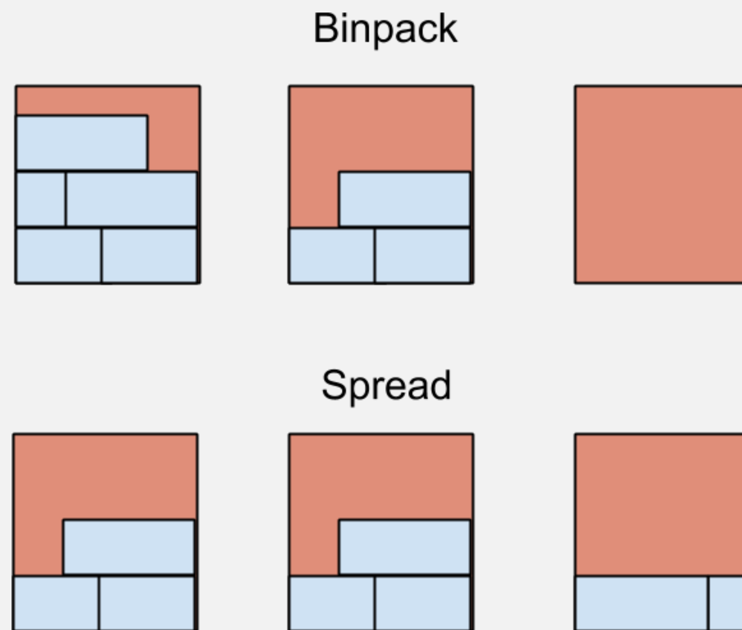
SCHEDULING

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:

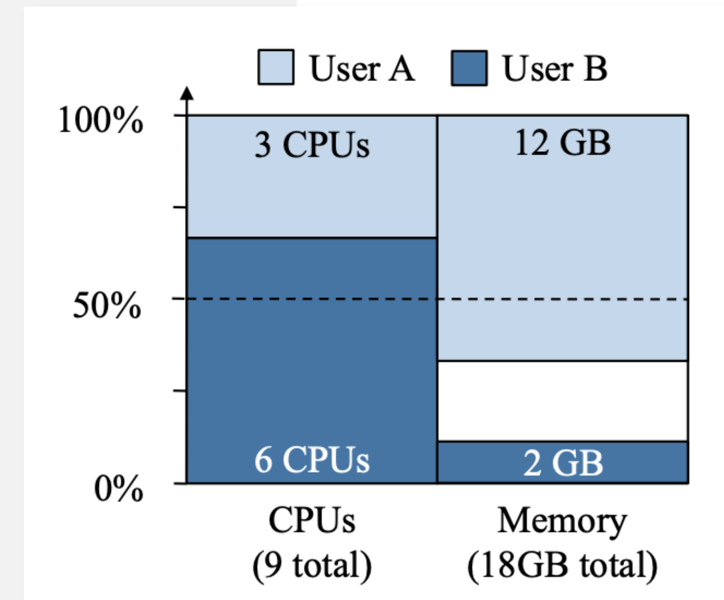
Cluster mit:	9 CPU	18 GB Mem
User A Task benötigt:	1 CPU	4 GB Mem
User B Task benötigt:	3 CPU	1 GB Mem
Rel. Bedarf A Task:	1/9 CPU	**2/9 Mem**
Rel. Bedarf B Task:	**1/3 CPU**	1/18 Mem

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

Also:

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

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

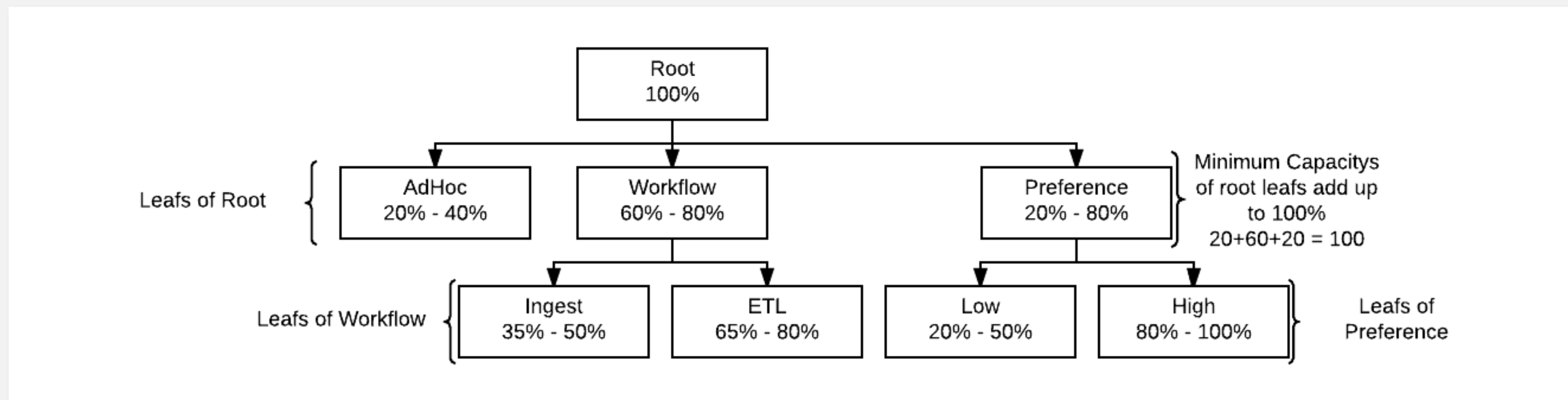


Quelle: Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: fair allocation of multiple resource types. `https://dl.acm.org/doi/10.5555/1972457.1972490`

SCHEDULING

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

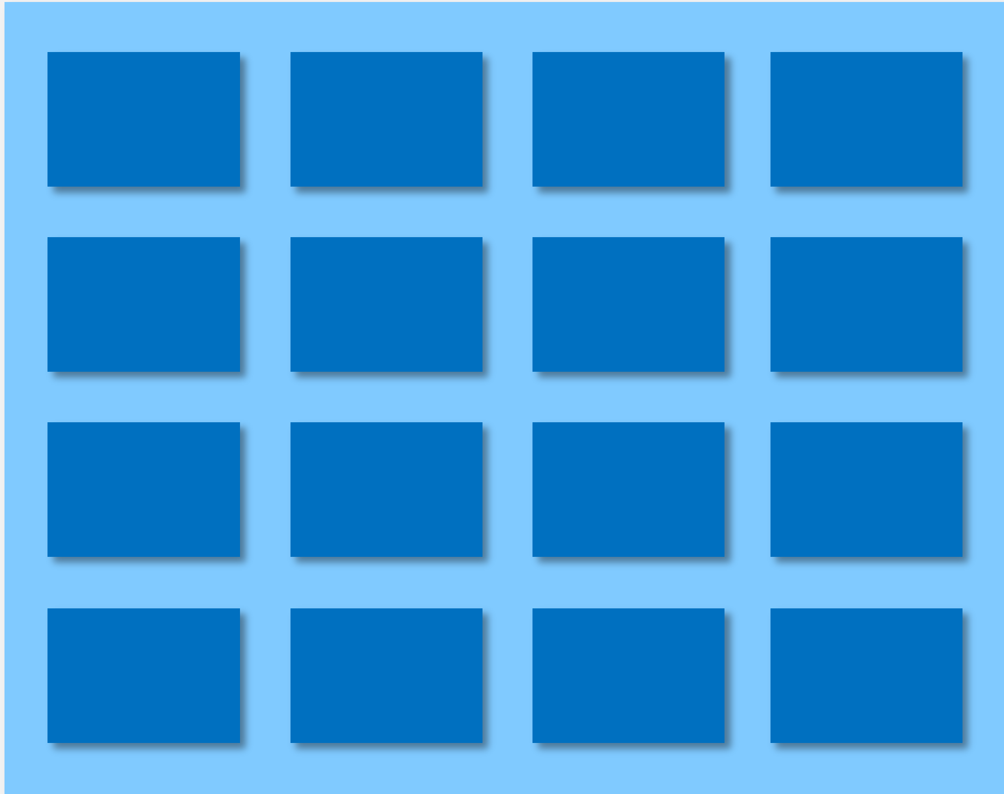
- Es werden Job Queues definiert und zu jeder Queue eine Kapazitätszusage in Ressourcenanteilen vom Cluster definiert.
- Fairness-Auffassung: Die minimale Kapazitätszusage wird stets eingehalten. Der Scheduling-Algorithmus stellt sicher, dass diese Fairness stets sichergestellt ist.
- Damit das Cluster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

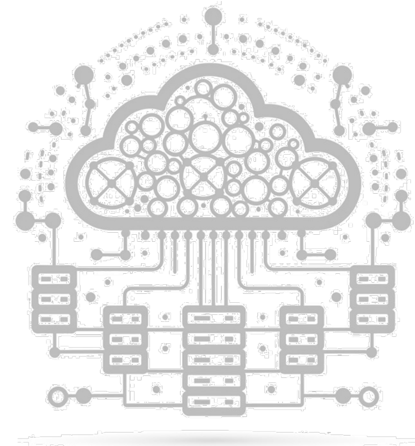
Ein Cluster sieht von außen wie ein großer einheitlicher Computer aus.

Konsequenz:

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

Das gilt insbesondere auch für das Scheduling.

CLOUD NATIVE COMPUTING



CLUSTER - SCHEDULER

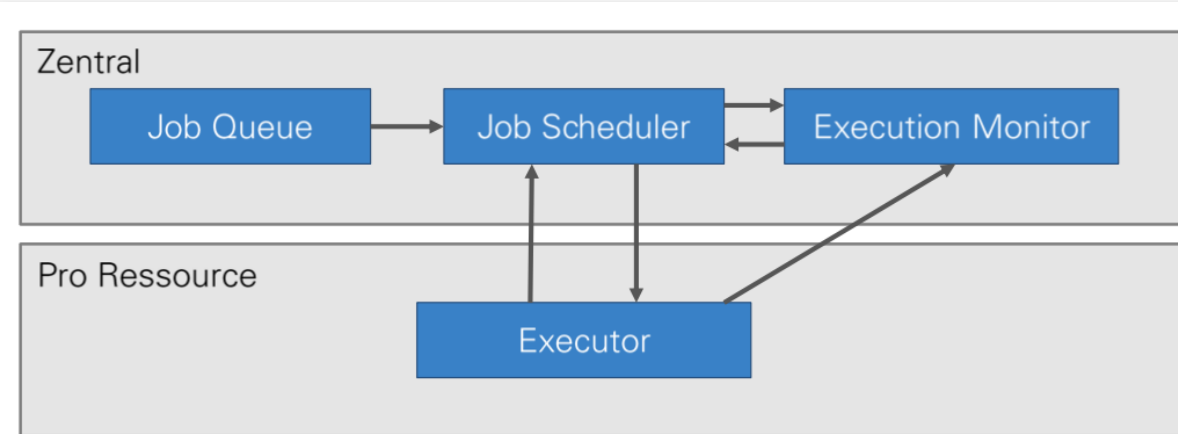
Eine konzeptionelle Architektur

Job Queue:
Eingehende Jobs zur Ausführung
Events zu eingegangen Jobs

Job Scheduler:
Jobs einplanen
Taskausführung steuern

Execution Monitor:
Taskausführung überwachen
Ressourcen überwachen

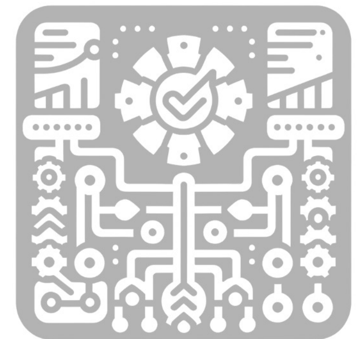
Executor:
Task ausführen
Informationen zur Ressource bereitstellen



Anforderungen:

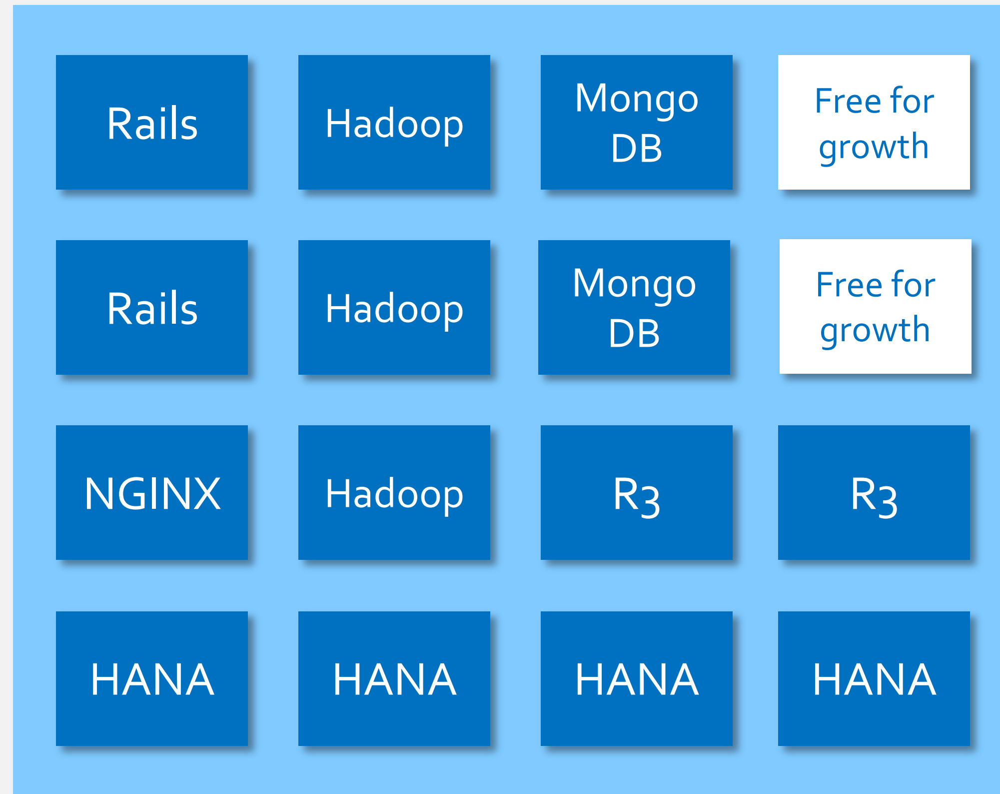
- Performance (Geringe Queeing-Time, Decision-Time, Ausführungslatenz)
- Hoch-Verfügbarkeit und Fehlertoleranz
- Skalierbarkeit bzgl. Anzahl an Jobs und verfügbaren Ressourcen

CLOUD NATIVE COMPUTING



SCHEDULER ARCHITEKTUREN

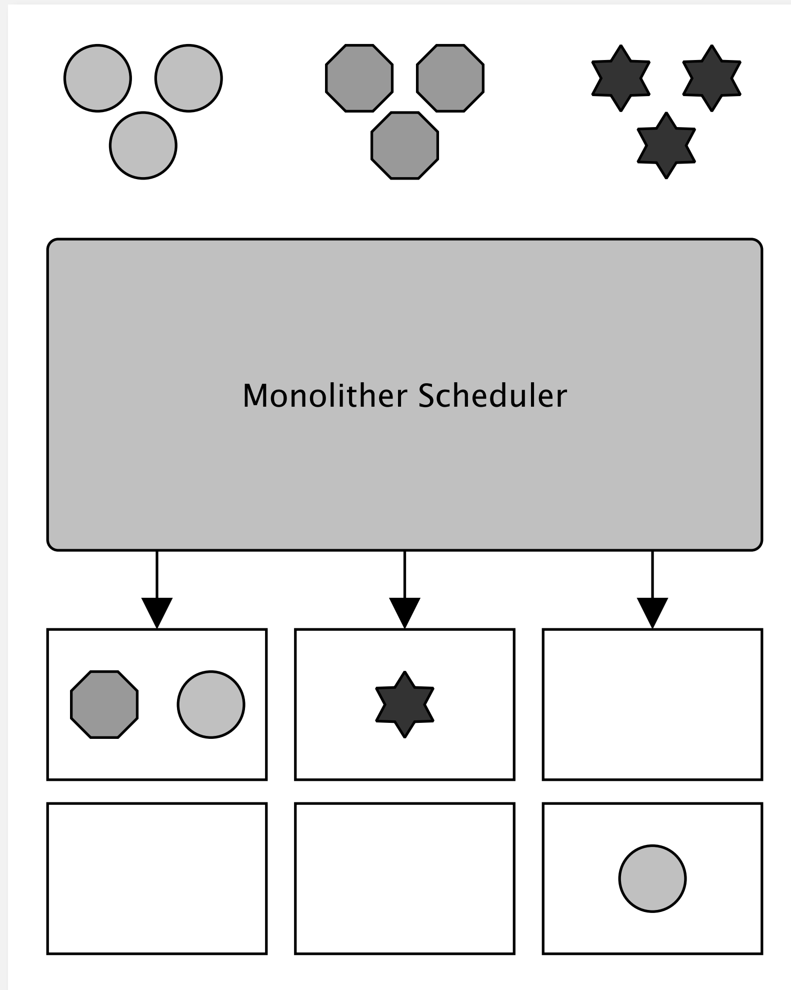
Variante 1: Statische Partitionierung



Kann man kaum Scheduler nennen.

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*

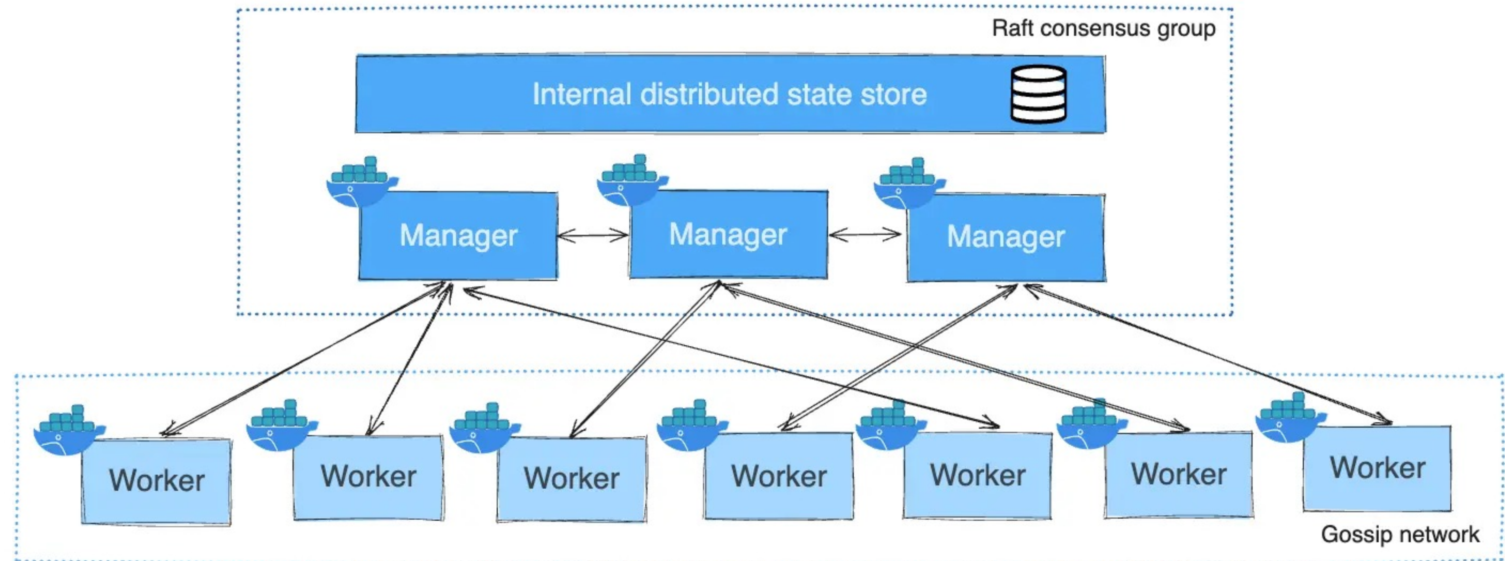
kubernetes

Quelle: 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

DOCKER SWARM

CLOUD NATIVE COMPUTING

- 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.



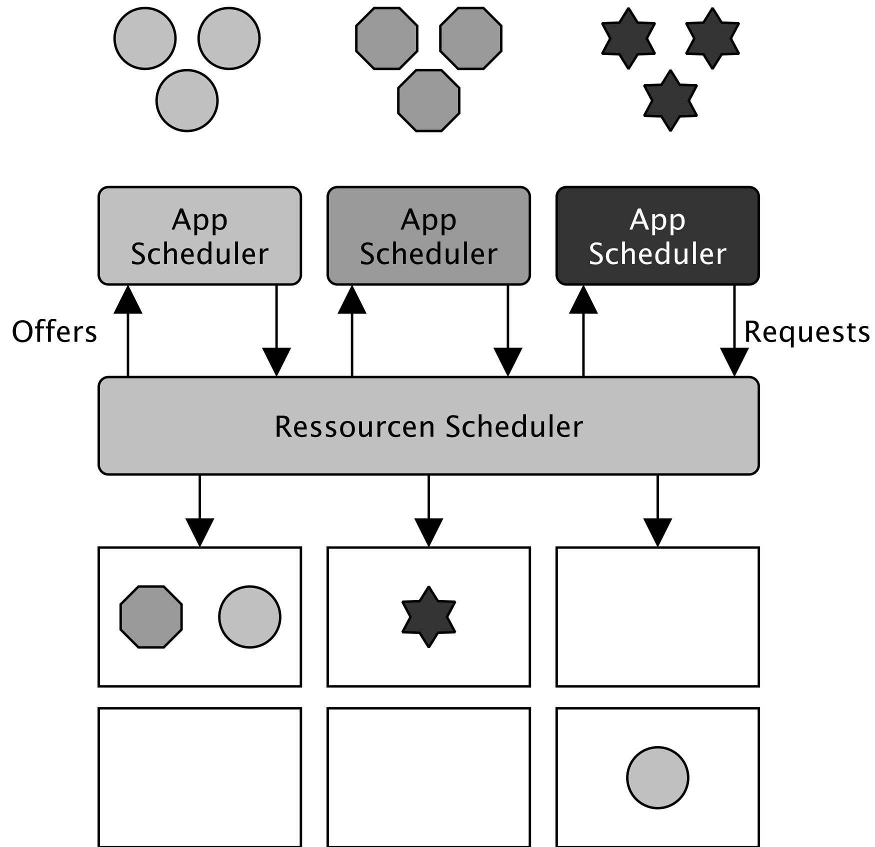
Scheduling gem. Spread Strategy

- 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 nächstmögliche 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/`

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 allokieren. 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

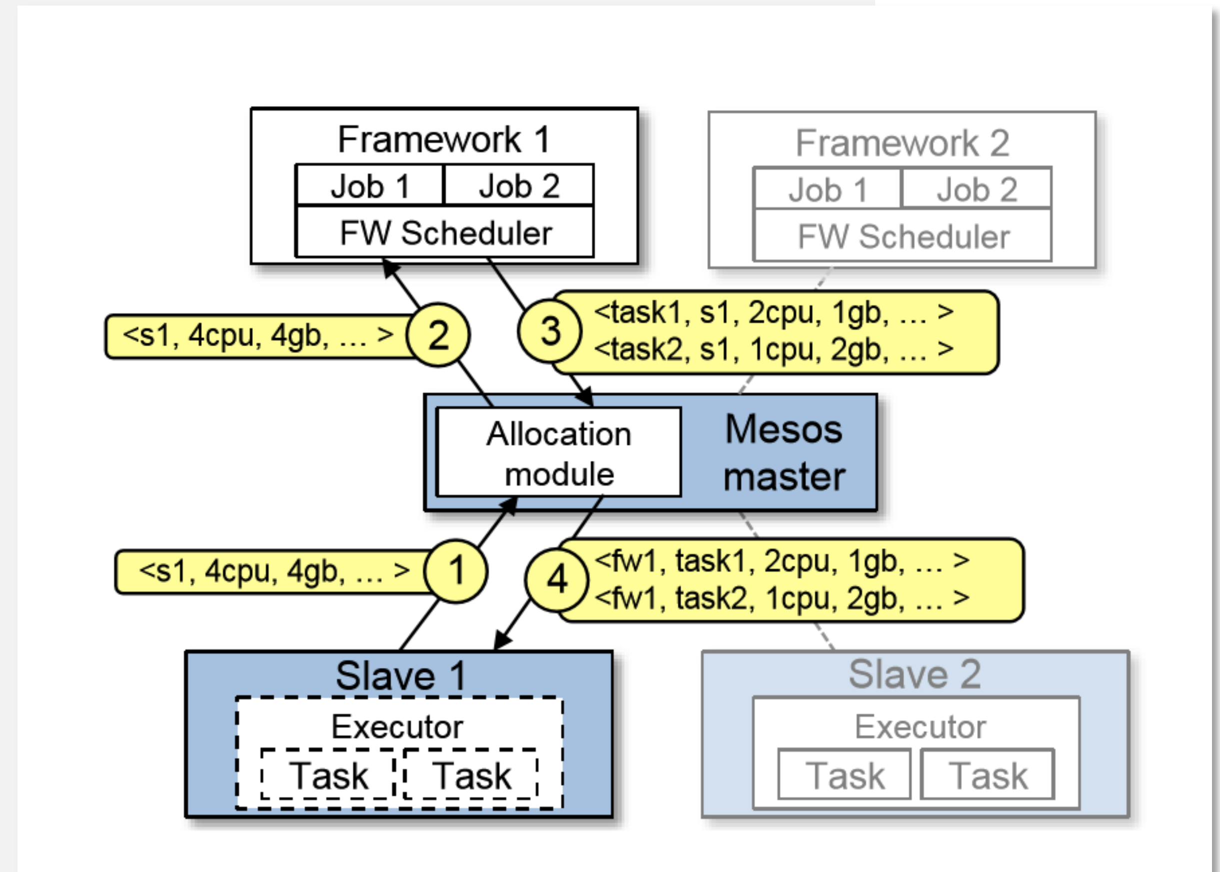
Nachteile:
App-Scheduler übergreifende Logiken nur schwer zu realisieren

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/~alig/papers/mesos.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 Ressource 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.

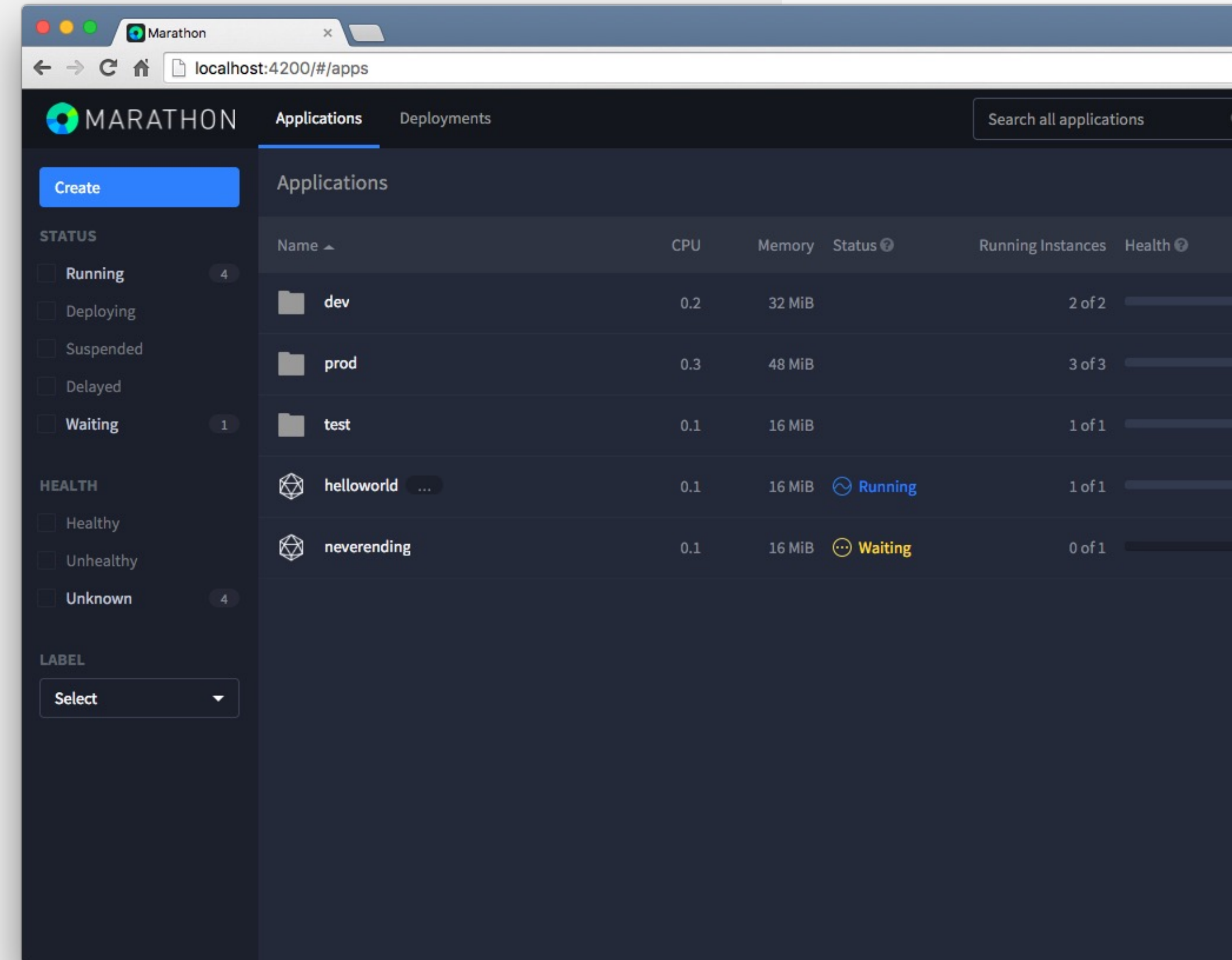


Quelle: Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: a platform for fine-grained resource sharing in the data center. `https://people.eecs.berkeley.edu/~alig/papers/mesos.pdf`

BEISPIELE

Apache Mesos + Marathon

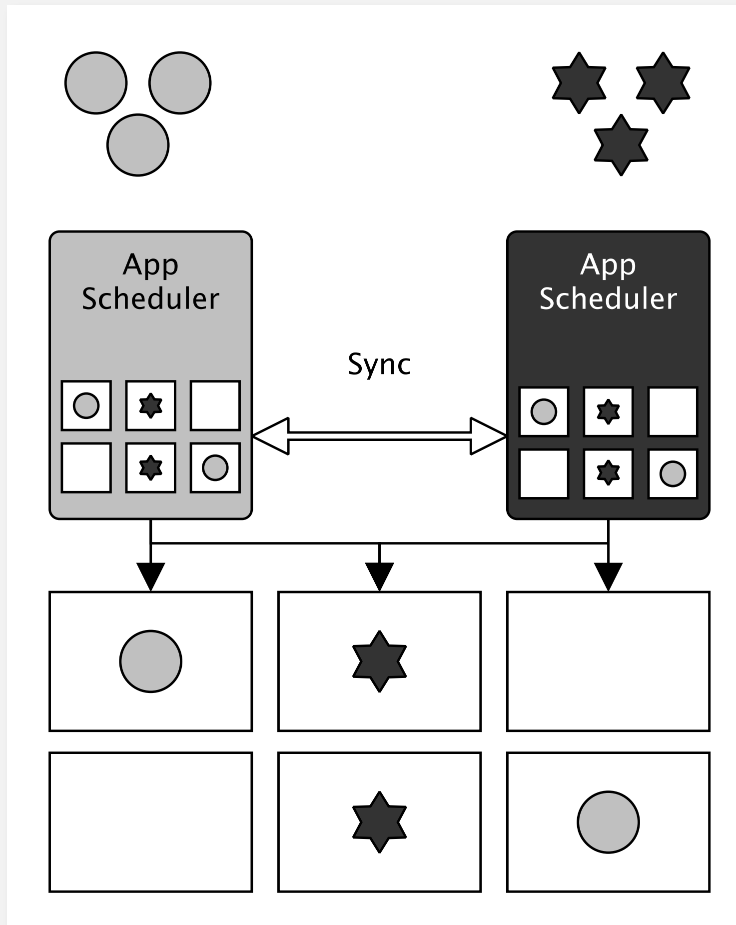
- Marathon ist ein 2nd-Level-Scheduler der auf die Ausführung von zustandslosen Services ausgelegt ist.
- Autor: Tobi Knaup (Ziel: Langlaufende zustandslose Services zuverlässig ausführen)
- Besitzt eigenständig Web-UI und REST-API
- Prozesse werden kontinuierlich am Leben gehalten. Terminiert ein Prozess, 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

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 entscheider 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 großen 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

AUSBLICK

Scheduling

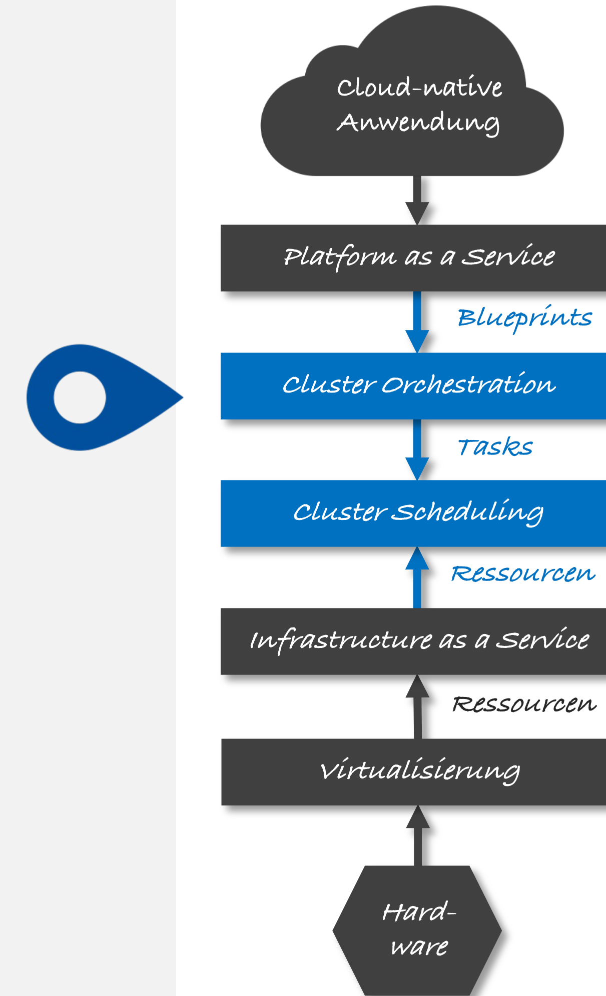
- Scheduling Problem Definition
- Scheduling Algorithmen
- Scheduler Architekturen
- Beispiele von Cluster Schedulern: Mesos, Swarm

Orchestrierung

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

Inside Kubernetes (Typ-Vertreter)

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



CLOUD NATIVE COMPUTING

KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de