CLOUD NATIVE COMPUTING

CLOUD-NATIVE

Unit:

Container-Orchestrierung

(4) Kubernetes (Teil II: Scheduling und Gesundheit von Workloads)

PROF. DR.
NANE KRATZKE

creative commons

CLOUD NATIVE COMPUTING

Urheberrechtshinweise

Diese Folien werden zum Zwecke einer praktikablen und pragmatischen Nutzbarkeit im Rahmen der **CCo 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 CCo 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

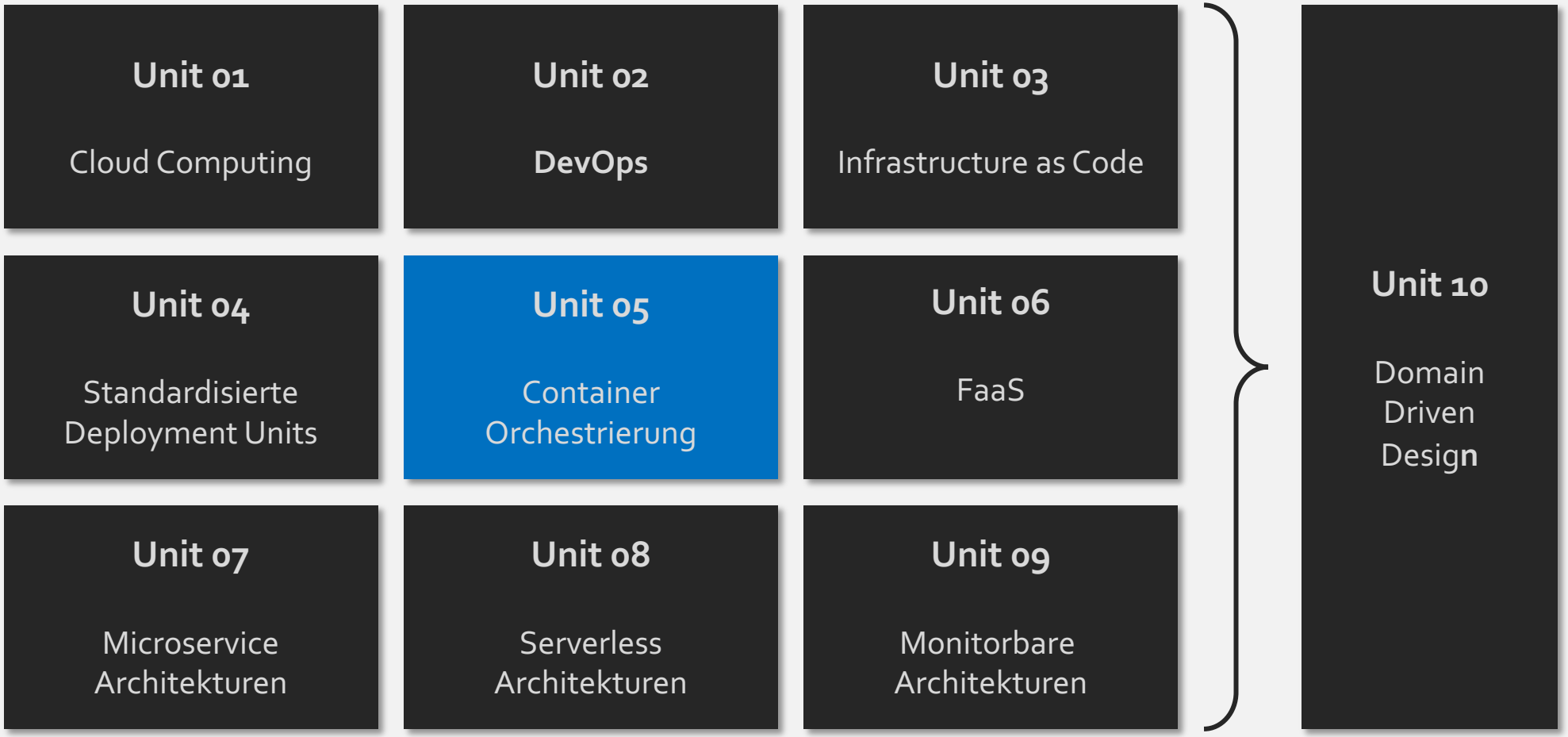
- Kubernetes-Architektur und Ressourcen
- Workloadarten
- Scheduling Constraints
- Automatische Skalierung von Workloads
- Exponierung von Services
- Health Checking
- Persistenz
- Isolation von Workloads

9.4 Zusammenfassung

CLOUD NATIVE COMPUTING

INHALTSVERZEICHNIS

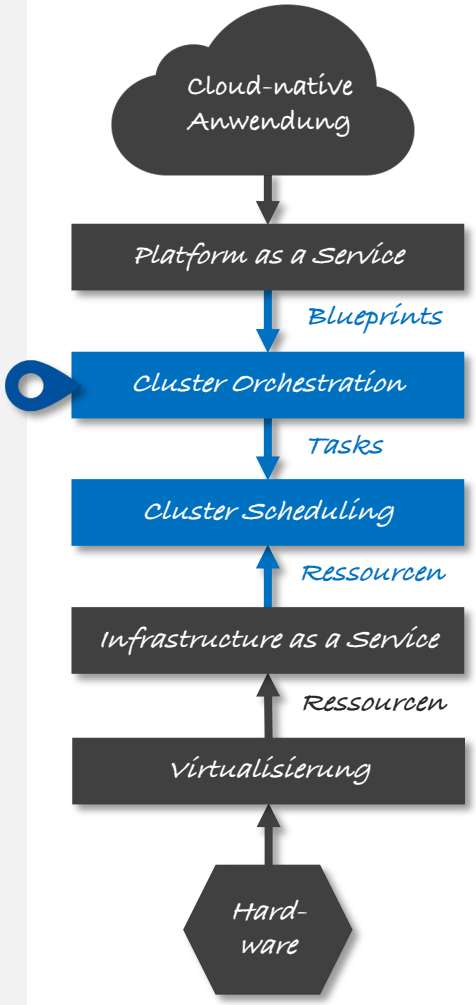
Überblick über Units und Themen dieses Moduls



KUBERNETES

Überblick über die wichtigsten von Controllern überwachten Kubernetes-Ressourcen

Ausführbare Workloads	Exponieren von Anwendungen und Diensten	Persistenz	Isolieren von Anwendungen mittels Namespaces
Deployments	Primär Intra-Cluster: Services	Persistent Volume Claim	Quotas und Limit Ranges
(Cron-)Jobs			
Daemon Sets	Primär Extra-Cluster: Ingress	Persistent Volume	RBAC
Stateful Sets			Network Policies



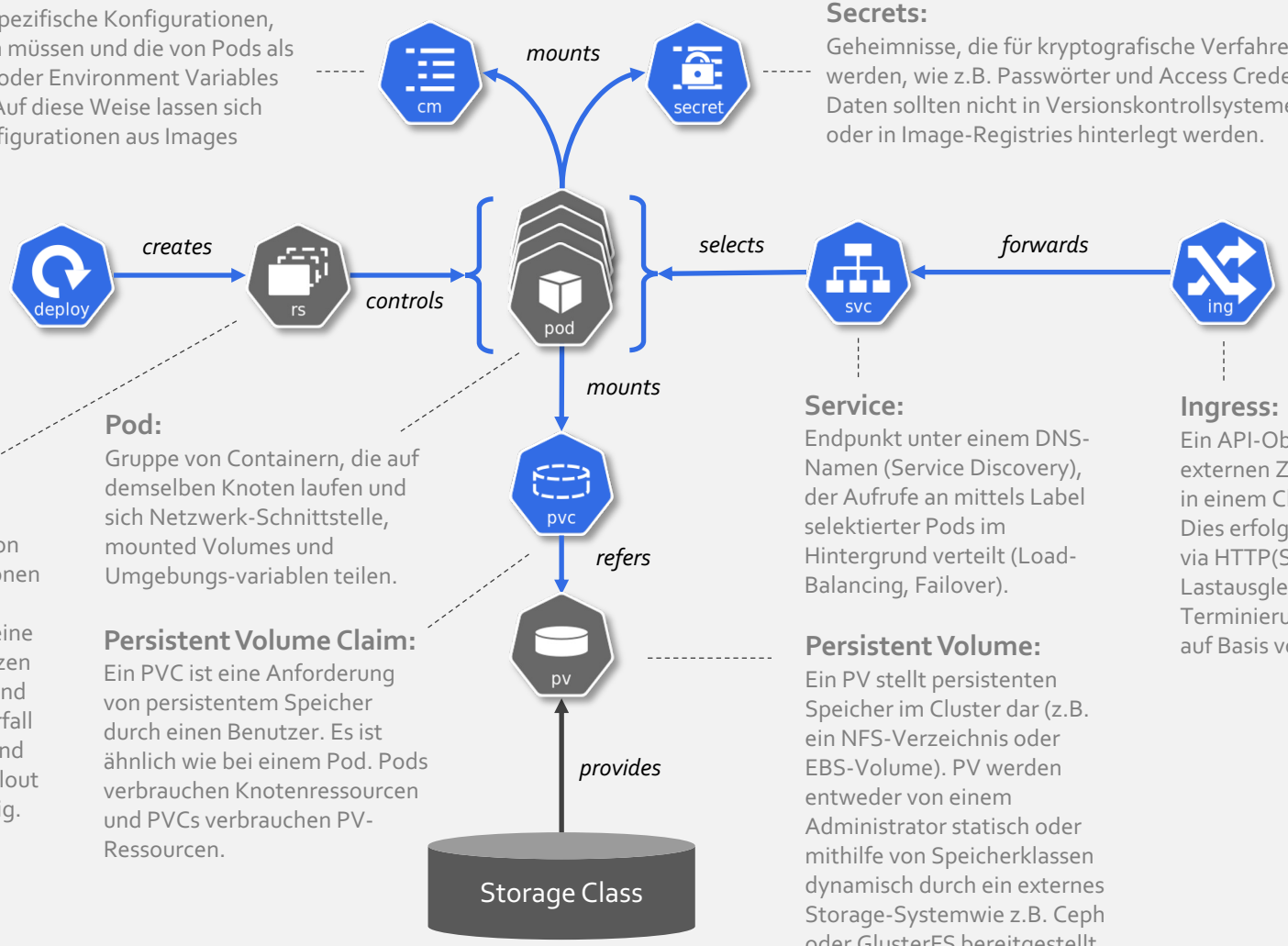
KUBERNETES

Basis-Blueprint eines in K8S deployten Webservices

Config Map:
Üblicherweise Environment-spezifische Konfigurationen, die im Betrieb gesetzt werden müssen und die von Pods als Files (Konfigurationsdateien) oder Environment Variables gemounted werden können. Auf diese Weise lassen sich Environment-spezifische Konfigurationen aus Images heraushalten.

Deployment:
Drückt den gewünschten Zielzustand einer Applikationskomponente (Service) aus (z.B. lasse 5 NGINX-Container in Version 1.3 laufen).

Replica Set:
Erforderlich um den Replication Controller (RC) mit Informationen zu versorgen. Replication Controller stellen sicher, das eine spezifizierte Anzahl an Instanzen von Pods ständig laufen. RC sind auch für Reaktionen im Fehlerfall (Re-Scheduling), Skalierung und Rollouts (Canary Rollouts, Rollout Tracks, Rollbacks, ...) zuständig.



Pod:
Gruppe von Containern, die auf demselben Knoten laufen und sich Netzwerk-Schnittstelle, mounted Volumes und Umgebungs-variablen teilen.

Persistent Volume Claim:
Ein PVC ist eine Anforderung von persistentem Speicher durch einen Benutzer. Es ist ähnlich wie bei einem Pod. Pods verbrauchen Knotenressourcen und PVCs verbrauchen PV-Ressourcen.

Secrets:
Geheimnisse, die für kryptografische Verfahren benötigt werden, wie z.B. Passwörter und Access Credentials. Diese Daten sollten nicht in Versionskontrollsystemen eingecheckt oder in Image-Registries hinterlegt werden.

Service:
Endpunkt unter einem DNS-Namen (Service Discovery), der Aufrufe an mittels Label selektierter Pods im Hintergrund verteilt (Load-Balancing, Failover).

Persistent Volume:
Ein PV stellt persistenten Speicher im Cluster dar (z.B. ein NFS-Verzeichnis oder EBS-Volume). PV werden entweder von einem Administrator statisch oder mithilfe von Speicherklassen dynamisch durch ein externes Storage-Systemwie z.B. Ceph oder GlusterFS bereitgestellt.

Ingress:
Ein API-Objekt, das den externen Zugriff auf Dienste in einem Cluster ermöglicht. Dies erfolgt normalerweise via HTTP(S) und beinhaltet Lastausgleich, SSL-Terminierung virtuelle Hosts auf Basis von DNS-Namen.

CLOUD NATIVE COMPUTING

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 von Controllern überwachten Kubernetes-Ressourcen

CLOUD NATIVE COMPUTING

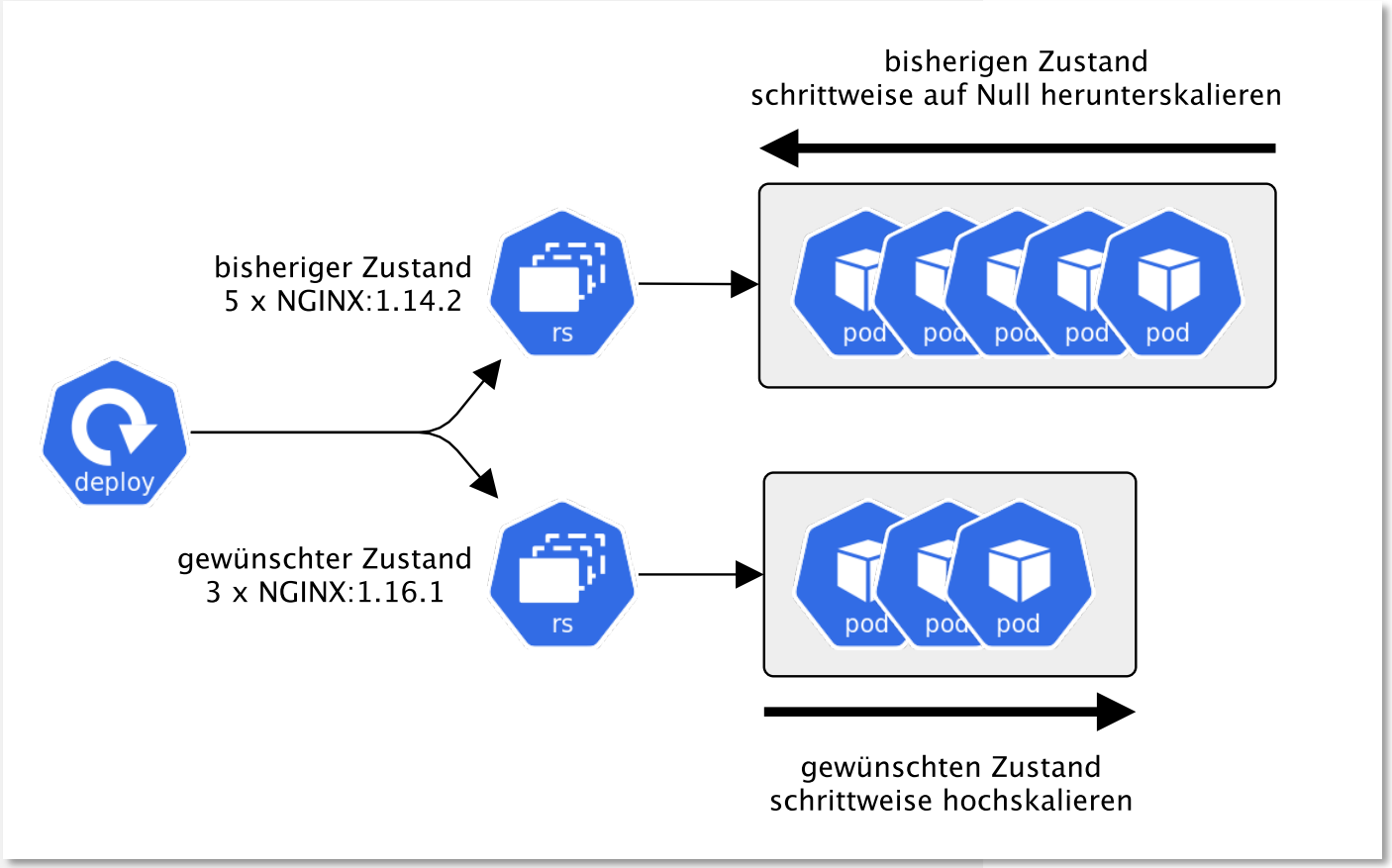
Ausführbare Workloads	Exponieren von Anwendungen und Diensten	Persistenz	Isolieren von Anwendungen mittels Namespaces
 **Deployments**	Primär Intra-Cluster  **Services**	 **Persistent Volume Claim**	  **Quotas und Limit Ranges**
 **(Cron-)Jobs**			
 **Daemon Sets**	Primär Extra-Cluster  **Ingress**	 **Persistent Volume**	 **RBAC**  **Network Policies**
 **Stateful Sets**			

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.
- 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

Workloads mit kubectl ausbringen und verwalten

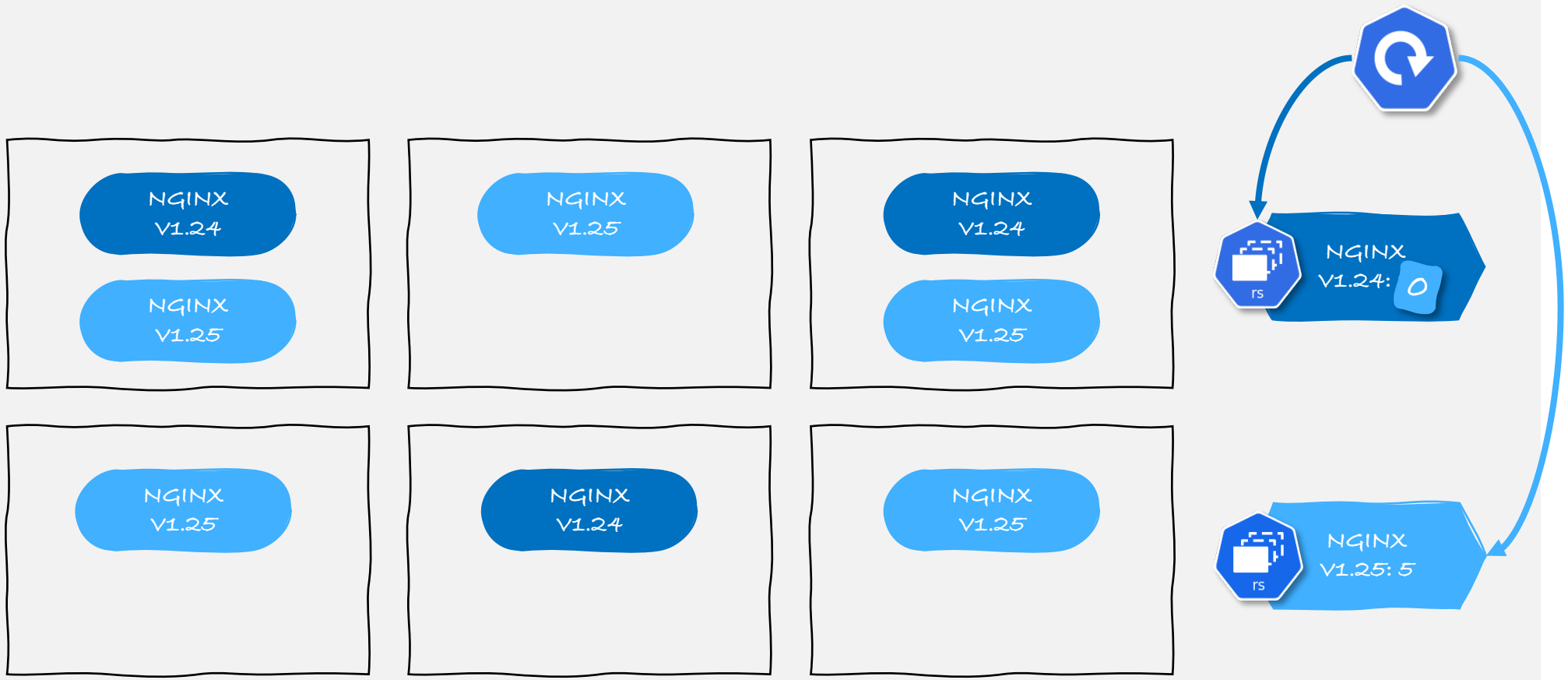
```
# Ausbringen eines Deployments (5 x NGINX Webserver)
kubectl apply –f nginx-dep.yaml

# 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 nginx-dep.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
  namespace: nane-kratzke
spec:
  replicas: 5
  selector: { matchLabels: { app: nginx }}
  template:
    metadata: { labels: { app: nginx }}
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports: [{ containerPort: 80 }]
```

REGELKREIS DES DEPLOYMENT CONTROLLERS

Wirkungsweise eines Rolling Updates



Current State:
Es laufen 3 Instanzen von NGINX (V1.24)

Desired State:
Betreibe 5 Instanzen von NGINX (V1.25)

Maßnahmen:

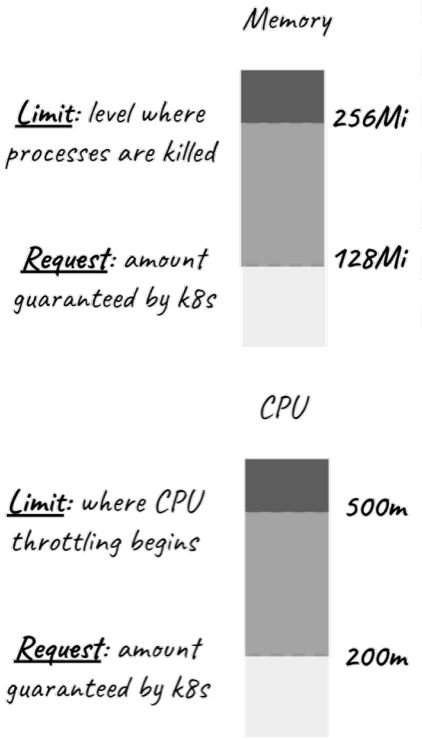
- Lösche 3 Instanzen (V1.24)
- Starte 5 Instanzen (V1.25)

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.
- **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 }]
        ressources: {
           requests: { cpu: “200m“, memory: “128Mi“ }
           limits: { cpu: “500m“, memory: “256Mi“ },
        }
```



Quelle: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

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

CLOUD NATIVE COMPUTING

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 74MB Unterschied zu 1GB)*

SCHEDULING CONSTRAINTS

Node-Selektoren

Mit Labeln (Key-Value Paare) kann man Nodes kennzeichnen, um z.B. den Scheduler bestimmte Workloads nur bestimmten Nodes zuweisen zu lassen.

```
# 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
```

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

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 in einem Content Delivery Network nur auf Nodes mit SSD-Platten platziert wird, um geringe Auslieferungszeiten von Platte bis Browser sicherstellen zu können.

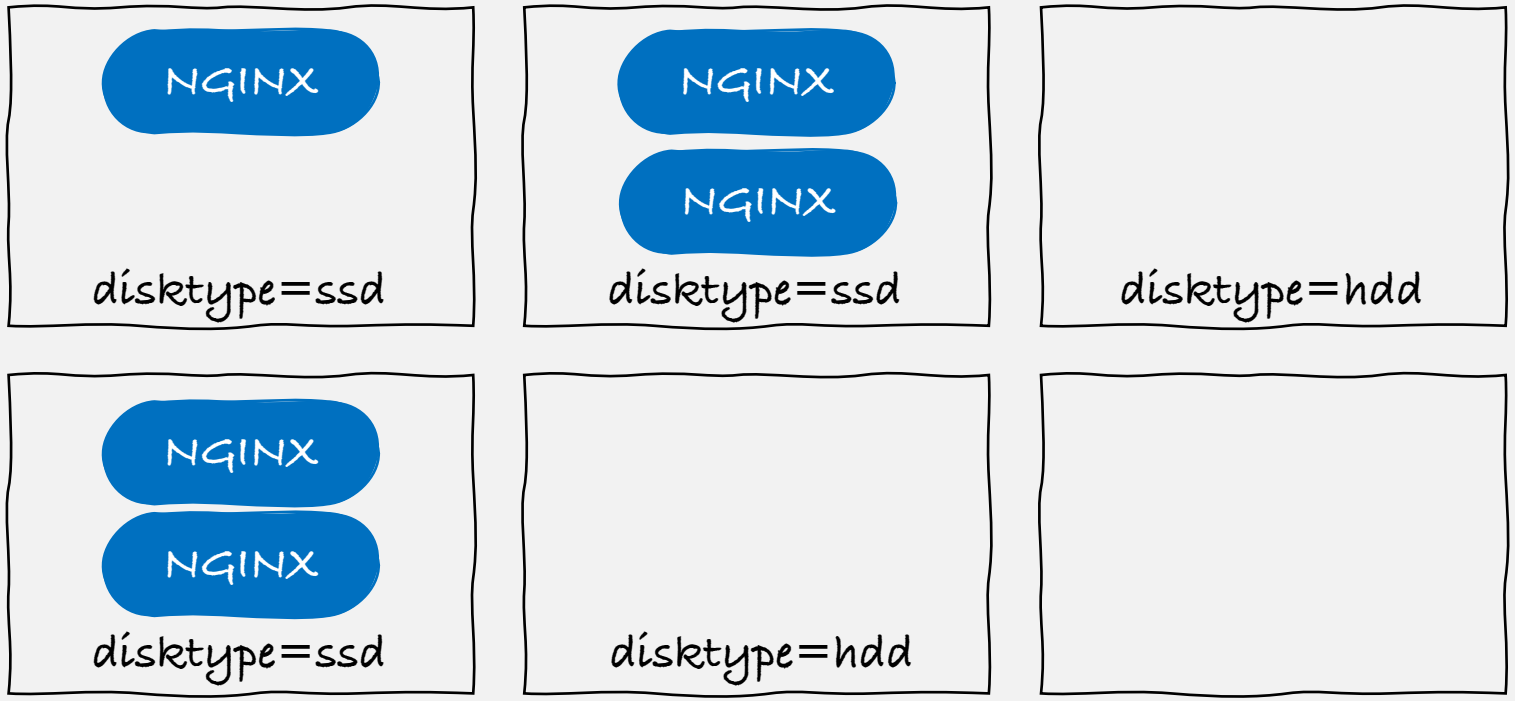
```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
  labels:
    env: test
spec:
  containers:
  - name: nginx
    image: nginx
    imagePullPolicy: IfNotPresent
  nodeSelector:
    disktype: ssd
```

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

Quelle: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/

SCHEDULING

Mit Node Labels




```
> kubectl apply -f nginx-dep.yaml
```

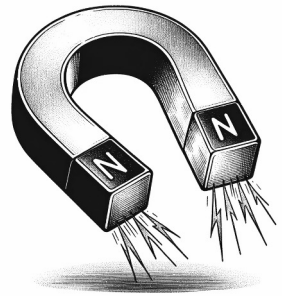
```
> kubectl scale --replicas=5 -f nginx-dep.yaml
```

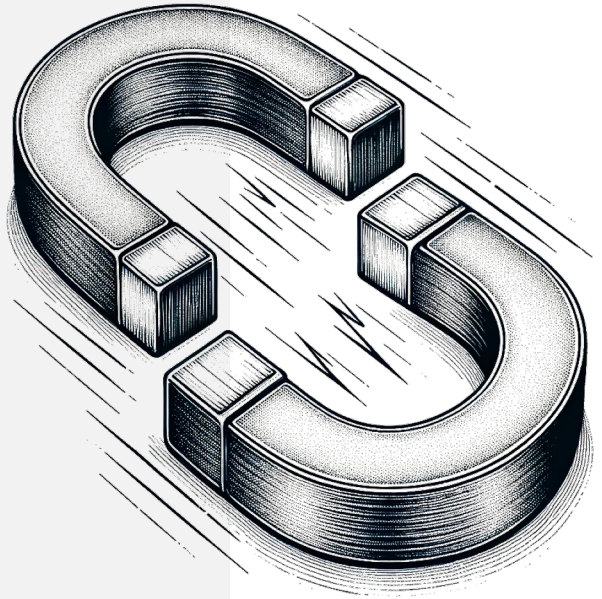
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
      nodeSelector:
        disktype: ssd
```

SCHEDULING CONSTRAINTS

Affinities

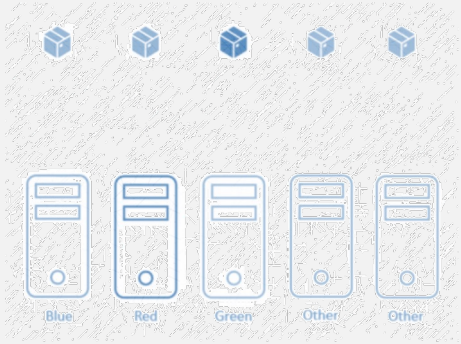
- Node-Selektoren ermöglichen Pods auf Knoten mit bestimmten UND-verknüpften Labeln hart zu beschränken.
- Das Affinitätskonzept erweitert die ausdrückbaren Randbedingungen erheblich:
  - Affinitäten sind ausdrucksvoller, da sie neben exakten (UND-verknüpften) Übereinstimmungen mehr Übereinstimmungsregeln 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 platziert.
  - Es können Affinitäten formuliert werden, die sich auf 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.



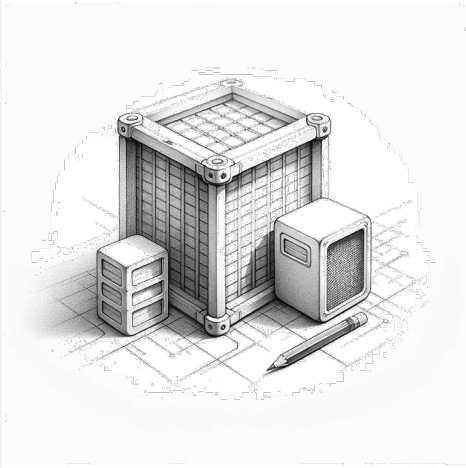


SCHEDULING CONSTRAINTS

Node Affinity



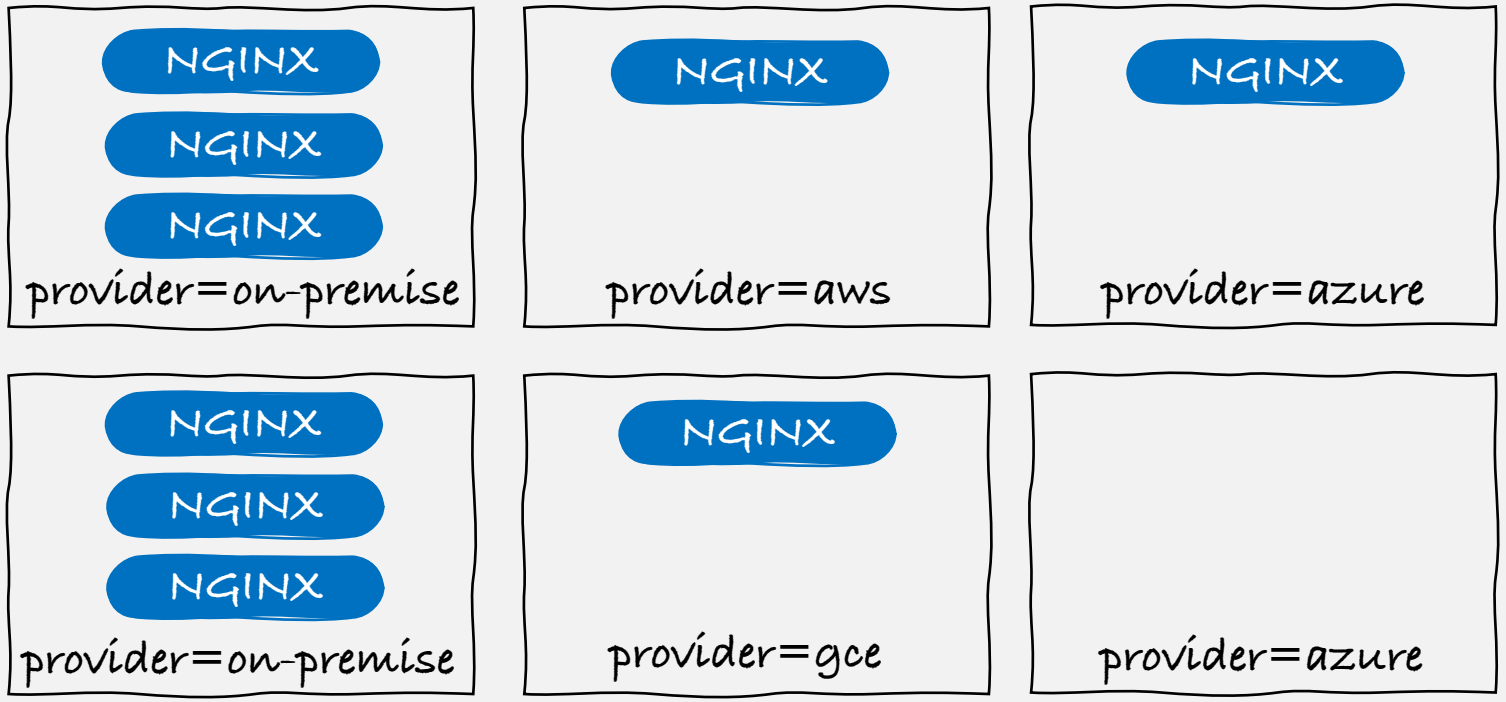
- Es gibt zwei Arten von Knotenaffinitäten
  - **`requiredDuringSchedulingIgnoredDuringExecution`**
    (harte Randbedingung, ähnlich Node-Selektor)
  - **`preferredDuringSchedulingIgnoredDuringExecution`**
    (weiche Randbedingung)
- Der Teil `IgnoredDuringExecution` bedeutet, dass der Pod nicht umgeplant 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.



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

SCHEDULING

Mit Node Affinities




```
> kubectl apply -f nginx-dep.yaml
```

```
> kubectl scale --replicas=6 -f nginx-dep.yaml
```

```
> kubectl scale --replicas=9 -f nginx-dep.yaml
```

```yaml
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: 80
      nodeAffinity:
        preferredDuringSchedulingIgnoredDuringExecution:
        - weight: 1
          preference:
            matchExpressions:
            - key: provider
              operator: In
              values:
              - on-premise

```

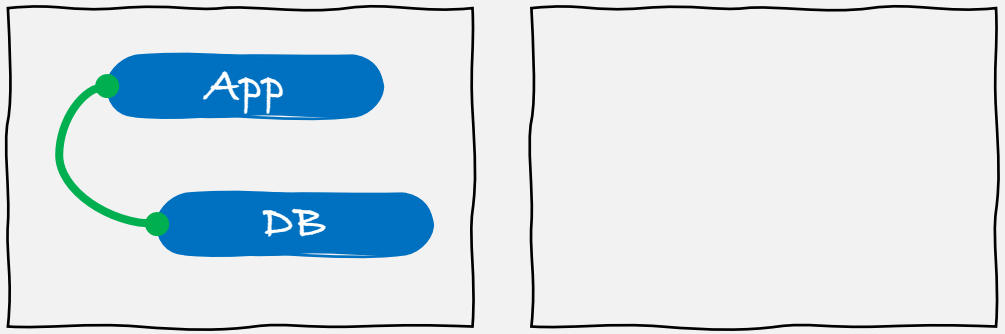
SCHEDULING CONSTRAINTS

Inter-Pod (Anti-)Affinities

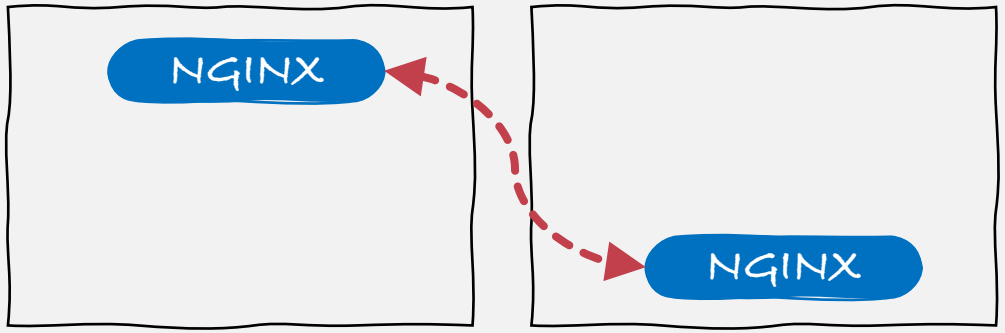
- Mit Inter-Pod-(Anti-)Affinitäten lassen sich mittels Pod-Labels (und nicht anhand von Knoten-Labels) festlegen, welche Knoten für einen Pod geplant (oder vermieden) werden sollen.
- Die (Anti-)Affinitäts-Regeln haben die Form:

  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.

Pod-Affinity



Pod-Anti-Affinity



Oft nutzt man, um Pods so zu platzieren, dass sie bewusst auf demselben Node landen.

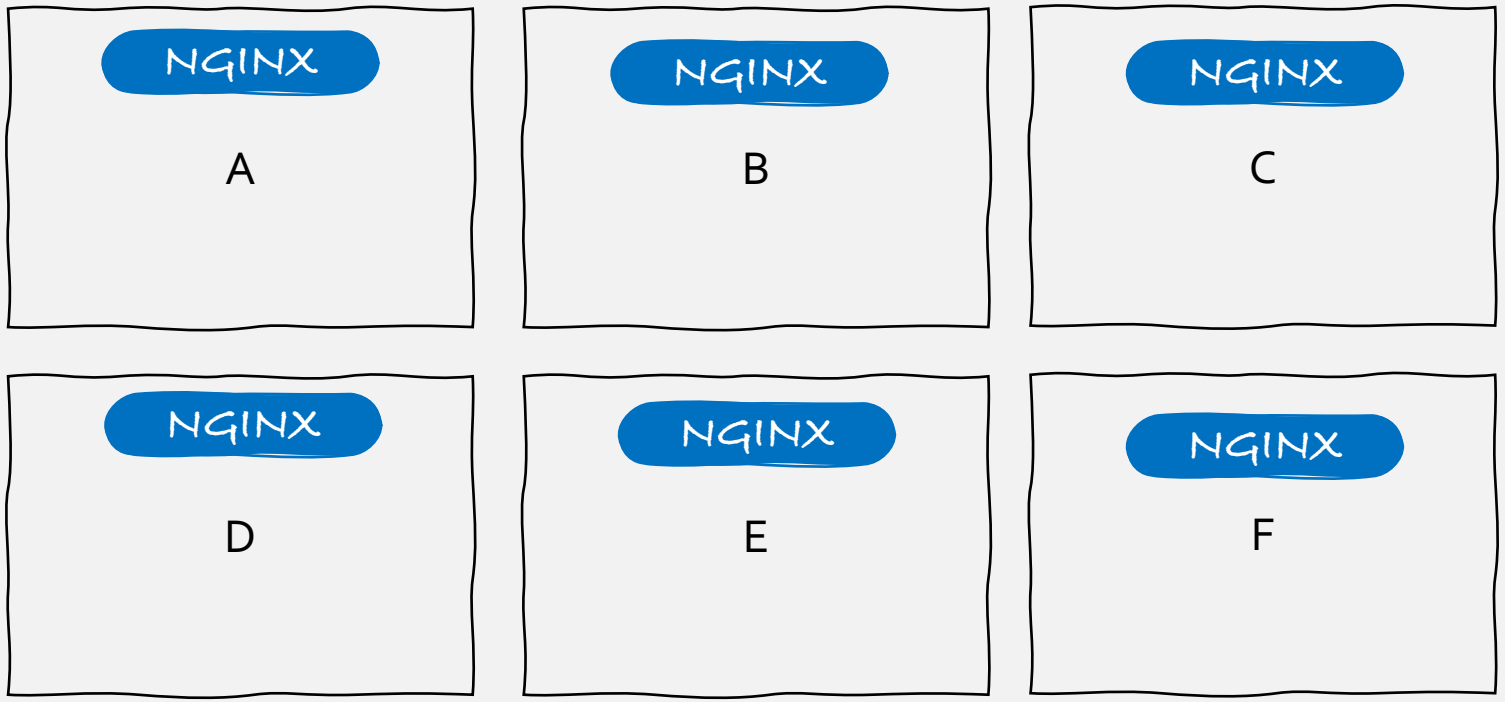
Dies kann aus Gründen der Minimierung von Latenzen Sinn machen. Z.B. um Datenbanken nah an Applikationsservern zu platzieren.

Oft nutzt man Anti-Affinities um Pods so zu platzieren, dass nie zwei Pods auf demselben Node landen.

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

SCHEDULING

Mit Pod Anti-Affinities




```
> kubectl apply –f nginx-dep.yaml
```

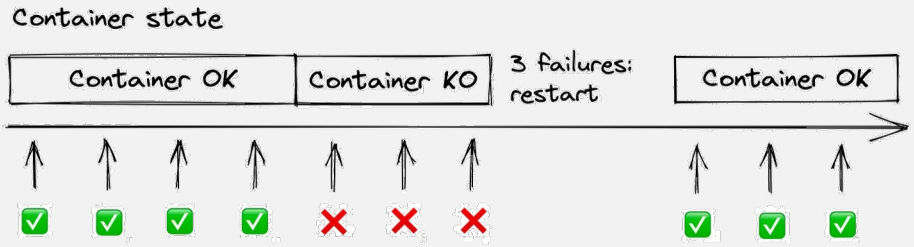
```
> kubectl scale --replicas=6 -f nginx-dep.yaml
```

```
> kubectl scale --replicas=9 -f nginx-dep.yaml
```

```yaml
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: 80
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - podAffinityTerm:
              labelSelector:
                matchExpressions:
                - key: app
                  operator: In
                  values: ["nginx"]
              topologyKey: "kubernetes.io/hostname"
```

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
```

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

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

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

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

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

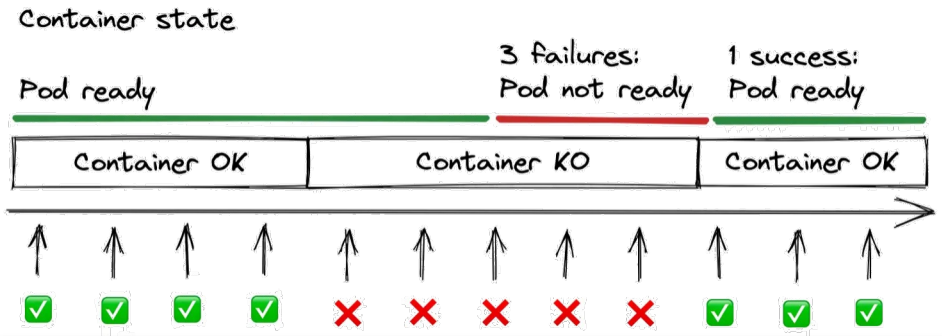
Quelle: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

POD HEALTH

Readiness und StartUp Probes

Readiness Probe

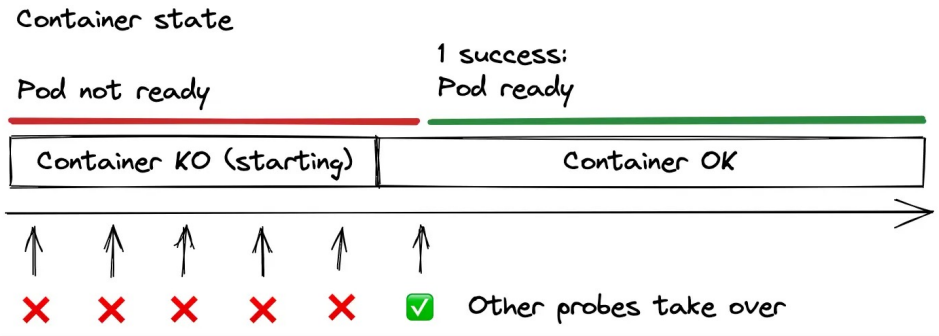
- Manchmal können Anwendungen auch Requests nur vorübergehend nicht bedienen.
- Beispielsweise könnte eine Anwendung beim Start große Datenmengen oder Konfigurationsdateien zeitaufwändig laden.
- In solchen Fällen sollte der Prozess nicht beendet werden, allerdings auch keine neuen Requests bekommen.
- Ein Pod dessen Readiness Probes melden, dass Container nicht bereit sind, empfangen keine Daten über Services.




```
readinessProbe:
   exec:
     command:
     - cat
     - /tmp/ready
   failureThreshold: 3
   periodSeconds: 10
```

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.




```
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.

*Dieser Startup Probe deckt 10 * 30 Sekunden = 300 Sekunden = 5 Minuten Startup-Time ab.*

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.
- **`path:`**
  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.

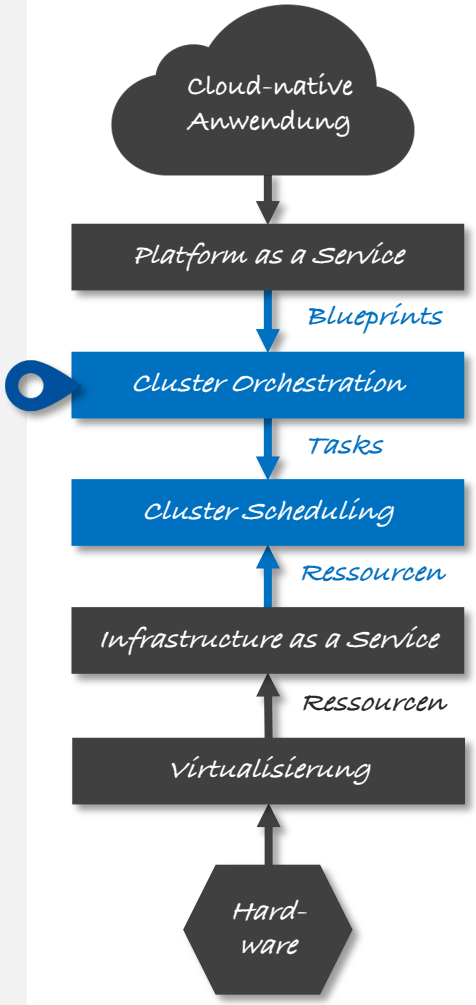
CLOUD NATIVE COMPUTING

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

AUSBLICK

Überblick über die wichtigsten von Controllern überwachten Kubernetes-Ressourcen

Ausführbare Workloads		Exponieren von Anwendungen und Diensten	Persistenz	Isolieren von Anwendungen mittels Namespaces
Deployments	(Cron-)Jobs	Primär Intra-Cluster: Services	Persistent Volume Claim	Quotas und Limit Ranges
Daemon Sets	Stateful Sets	Primär Extra-Cluster: Ingress	Persistent Volume	RBAC, Network Policies



KONTAKT

Disclaimer

Nane Kratzke

+49 451 300-5549

nane.kratzke@th-luebeck.de

kratzke.mylab.th-luebeck.de