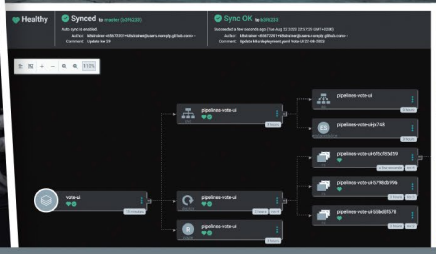
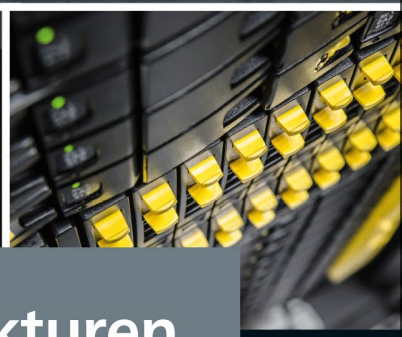
Oliver Liebel





Skalierbare Container-Infrastrukturen

Das Handbuch für Planung und Administration

- Architektur, Setup, strategischer Background, Best Practices
- Professionelle Orchestrierung mit Kubernetes und OpenShift
- IaC, Operatoren, GitOps, Security, Federated Cluster, GPU-Cluster, API-Gateways, Software-Defined Storage, IDM, Autoscaler

4., aktualisierte Auflage

Alle Beispielkonfigurationen zum Download

Rheinwerk Computing

Kapitel 1
Catch-22

»How hard can it be?«
– Jeremy Clarkson, Top Gear

Ja, wie schwer kann das alles schon sein …

Nun, wenn ich an die mittlerweile weit mehr als 6.000 Seiten denke, die ich in den letzten 6 Jahren im Rahmen von nunmehr 5 deutsch- und englischsprachigen Container- und KI/ML-Infrastruktur- Publikationen zu Papier bzw. in E-Books gebracht habe, stellt sich die Frage von Mister Clarkson bei ernsthafter Betrachtung schon lange nicht mehr.

Denn mittlerweile sind Vanilla-Kubernetes-Cluster und die von ihnen für ein halbwegs produktivtaugliches Umfeld benötigten 3rd-Party Tool Stacks dank ihrer extremen Komplexität und hohen Volatilität oft nur noch eines: eine Arbeitsbeschaffungsmaßnahme erster Güte. Ein Selbstzweck, eine meist kaum noch seriös auflösbare Sisyphus-Schleife, die mittlerweile ein kleines bisschen an den legendären, koordinierten Wahnsinn aus dem im Titel benannten Antikriegsdrama *Catch-22* erinnert.

Der Catch-22 besagt vereinfacht: Nur ein Pilot, der verrückt ist, kann beim Arzt einen Antrag auf Fluguntauglichkeit stellen. Ist der Pilot jedoch in der Lage, seinen Antrag auf Fluguntauglichkeit zu stellen, kann er anscheinend doch rational denken und ist daher nicht verrückt, ergo ist er diensttauglich.

Portieren wir das Ganze:

Kunde: »Wir wollen Vanilla Kubernetes einsetzen, damit wir effizienter arbeiten können.«

Neutraler Berater: »Aber mit Vanilla Kubernetes und allen benötigten 3rd-Party Stacks wird sich dank der hohen Volatilität und kurzen Lifecycles Ihr HR- und Kosten-Aufwand eher erhöhen.«

Kunde: »Aber dank des hohen Automationsgrades lässt sich das doch kompensieren.«

Berater: »Prinzipiell ja, wenn sich unter der Kubernetes-Haube nicht alles ständig verändern würde und selbst jede Drittkomponente bei jedem Update immer wieder an etliche andere neu angepasst werden müsste. Wie oft sollen Ihre DevOps-Teams die Automation denn neu aufsetzen? Zudem gibt es keinen Long Term Support. Stattdessen *permanent change*. Ist sogar der offizielle, stolze CNCF Tenor.«

Kunde: »Aber alle sind doch längst auf den Zug aufgesprungen und es scheint gut zu funktionieren, vor allem in der Cloud, also muss die Sache doch Hand und Fuß haben.«

Berater, in Gedanken: *Sicher. Und im Fernsehen gibt es sprechende Knuffelhäschen*. Berater, in Worten: »Wenn Sie statt Vanilla Kubernetes beispielsweise OpenShift einsetzen, dann wahrscheinlich.«

Kunde: »Aber das ist doch auch nur ein Kubernetes von Red Hat und kostet zudem Lizenzgebühren. Wir nehmen Vanilla Kubernetes. Punkt.«

Berater, in Gedanken: *Ich hab's versucht. Dann eben das dreifache Beratungskontingent. Ka-Tsching.* Berater, in Worten: »Gern, selbstverständlich.«

Als ich mich 2017 entschloss, die erste Publikation über Container-Cluster auf den Weg zu bringen, dachte ich bereits damals, dass Kubernetes als legitimer Nachfolger der guten, alten Pacemaker-HA-Cluster aus der Legacy-Welt ganz sicher einiges umkrempeln würde.

So weit, so gut. Und die Veränderungen kamen, und sie hörten nicht auf. Und es wurden mehr und mehr. Ich begleitete Kubernetes und Co über die Jahre, versuchte mit meinen Büchern halbwegs am Innovationspuls zu bleiben, aber letztlich blieben meine Publikationen nur Snapshots eines bestimmten Evolutionsstandes.

Von Version zu Version beglückte und beglückt uns die CNCF-gesteuerte Innovationsmaschine dank Tausenden von Kontributoren und Millionen von Entwicklern weltweit mit unzähligen neuen Konzepten, Features und Changes, ausgespuckt vom High-Speed-CI/CD-Fließband, und oft genug ohne Backward-Compatibility. Neue Funktionen wurden und werden im gefühlten Sekundentakt eingeführt, alte verworfen, und bestehende, wichtige, aber unausgereifte Funktionalitäten krebsen oft jahrelang mit eklatanten Mankos herum, weil dank ADHS-Mentalität lieber fleißig an neuen Nischen-Features entwickelt wird, die eventuell irgendwann einmal irgendjemand gebrauchen kann, anstatt für den Unternehmensbereich wichtige Features rock-solid zu Ende zu denken.

Um das wortwörtliche Ausmaß etwas besser zu verdeutlichen: Der Punkt, an dem Ihnen *eine einzelne Person* einen Kubernetes-Cluster mit allen für Produktivsysteme benötigten Tool Stacks von der ersten bis zur letzten, einhundertmillionsten Schraube wirklich *vollumfänglich*, bis ins letzte Bit, erklären könnte, ist längst passé. Es sei denn, auf diesem Planeten existiert ein Superhirn, das mir persönlich nicht bekannt ist. Auch in Beratungen oder Workshops entdecke ich immer wieder den gleichen Verlauf bei Teilnehmern: die anfängliche Euphorie, sich ›mal eben‹ etwas Know-how zu Kubernetes anzuschaffen, die nach und nach der etwas bedrückenden Erkenntnis weicht, dass man selbst nach vier mit vielen Informationen angefüllten Tagen lediglich einen kleinen, oberflächlichen Blick in die gewaltige Tiefe geworfen hat, die Kubernetes-Cluster längst darstellen.

Die Problematik wurde noch deutlicher, als ich in den Jahren 2020 bis 2022 zusammen mit NVIDIA an meiner Publikation über Skalierbare KI/ML-Infrastrukturen arbeitete, die Ende 2022 erschien.

Obwohl das KI-Infra-Buch mit einem Volumen von 468 Seiten gerade einmal einem Drittel des Umfangs meiner bisherigen Container-Publikation entsprach, dauerte die Arbeit daran erstmals ganze drei Jahre, und damit fast dreimal so lange wie die an einer meiner Container-Publikationen. Ursache: das hochvolatile und komplexe Kubernetes im Unterbau, und dazu der hochvolatile und hochkomplexe KI/ML-Stack mit GPU-, NFD- und AI-Workspace-Operatoren on top, dazu die Virtualisierungsschicht mit IaC-Automation, deren GPU-Hardware mithilfe Dutzender Einstellschrauben, die sich ebenfalls gern und schnell verändern, für den jeweiligen Anwendungsfall vernünftig und vollautomatisiert integriert werden will. Und alles in permanenter Veränderung.

Trauriger Fakt war, dass ich nach Beendigung der Arbeit an einem Themenblock oft genug den vorherigen, bereits fertiggestellten wieder überarbeiten oder gar komplett neu erstellen musste. Sisyphus lässt grüßen, aber wie üblich sucht sich jeder sein zu tragendes Päckchen meist selber aus.

Und das gilt auch für die Unternehmen, die sich entscheiden, Vanilla Kubernetes einzusetzen – und darunter fallen auch alle Managed-Cloud-Derivate wie GKE, AKS, EKS und Co. Die Gründe für die Ineffizienz im Vergleich zu echten Enterprise-Kubernetes-Lösungen sind vielfältig und werden später im Buch ausführlich thematisiert.

Aber die Vanilla-Kubernetes-basierten Lösungen sind zumindest eines: eine unlimitierte Gelddruckmaschine für Beratungsunternehmen, die mit oftmals brandgefährlichem Halbwissen ihren Kunden Kubernetes-Cluster nach wie vor als *die* simple Universal-Lösung für alle IT-Probleme schmerzfrei anpreisen. Zügige Migration? *Kein Problem*. Know-how-Transfer? *Alles schnell erledigt, vertrauen Sie uns!*

Na sicher.

Und genau diese Praktiken sind wiederum erfahrungsgemäß auch der Hauptgrund, warum ebenfalls zahllose Unternehmen – nach ebendiesen fragwürdigen Versprechungen – in den fatalen Irrglauben versetzt werden, dass die Migration ihrer hochkomplexen Legacy-Systeme in containerisierte Umgebungen in maximal zwei Wochen komplett abgefrühstückt ist. Und ihre Mitarbeiter nebenbei »mal eben«, am besten in maximal zwei Tagen, vollumfänglich zu hochkompetenten Container-Spezialisten ausgebildet werden können. Und wieder: *Na sicher*.

Willkommen im Schwurbel-Bingo-Einhorn-Land.

Eine höchst bedauerliche Problematik, auf die ich bereits in meinen letzten vier Container-Publikationen mehr als ausdrücklich hingewiesen habe und an der sich bis heute herzlich wenig geändert hat. Im Gegenteil. Aber was soll's – denn den Sales-Fraktionen von AWS, GCP, Azure, Red Hat, VMware, NVIDIA und Co. bereitet das meist keine mächtigen Kopfschmerzen. Allen übrigen Beteiligten sehr wohl.

Denn im Grunde gilt nur noch eine Regel: dass nichts mehr gilt. Oder in Langform: Das meiste, was gestern noch superhip und State of the Art war, ist heute gern nur noch alter Krempel, der keinen mehr wirklich interessiert. LTS, Long Term Support … was war das gleich noch?

Und das trifft auf fast alle Beteiligten zu – mit Ausnahme des Endkunden im Unternehmensbereich. Denn der würde sich gern einfach mal eine etwas langsamere Pace wünschen, deutlich weniger Komplexität und dass die mittlerweile unzähligen Schrauben einfach mal länger als gefühlte zwei Tage supported werden – Innovation hin oder her. Was schätzen Sie, wie viele Unternehmenskunden aktuell noch auf einer Vanilla-Kubernetes-Version herumjonglieren, deren EOL bereits vor einem Jahr oder deutlich längerer Zeit abgelaufen ist – ganz einfach, damit die Systeme beim längst überfälligen Upgrade nicht komplett explodieren und dann ohnehin *from scratch* neu aufgebaut werden müssen?

Seien Sie versichert, es sind nicht wenige. Und es sind einige durchaus bekannte Namen dabei.

Selbstverständlich ist das keine Info, die die CNCF oder die betroffenen Unternehmen gern an die große Glocke hängen. Fatalerweise müssen sich die Unternehmen aber ebendieser hochvolatilen und auf Dauer explosiven Gemengelage permanent neu stellen und anpassen, und dies oft zu einem – im wahrsten Sinne des Wortes – hohen Preis. Kosteneffizienz ist langfristig nun einmal nur mit maximaler Vollautomation zu erreichen. Und die kann mittel- und langfristig niemals erreicht werden, wenn sich das Kernsystem im Unterbau und alle wichtigen Zusatzkomponenten (Letztere natürlich in asynchronen Zyklen zu den ebenfalls kurzlebigen Kubernetes-Release-Cycles) alle paar Wochen bzw. Monate ändern.

Das Resultat: Kaum ein anderes IT-Geschäftsfeld ist im betrachteten Stand mit derartig hohen Kosten hinterlegt wie Vanilla-Kubernetes-basierte Systeme im Unternehmensumfeld, und mit zusätzlichem Nachbrenner, wenn noch KI/ML-Stacks *on top* werkeln. Dank maximaler Volatilität und dem Willen zum Wahnsinn, dem sich anscheinend in diesem IT-Bereich niemand mehr wirklich entziehen kann.

Mich selbst eingeschlossen.

Denn sonst würde ich dieses Buch nicht erneut schreiben.

Catch-22.

Wie ich bereits sagte.

Aber ich glaube nach wie vor daran, dass selbst Käpt'n Vanilla Kube irgendwann den Sprung in die echte Effizienz schaffen wird, so wie sein einziger wirklich erwachsener Bruder mit dem roten Hut, und vielleicht mit einer ordentlichen Prise LTS. Vielleicht dann, wenn die Entwickler irgendwann müde werden, alle paar Wochen die x-millionste API-Change einzubringen. Oder wenn alle Unternehmen sprichwörtlich den Kaffee auf haben und Kubernetes und der Containerisierung den Rücken kehren, weil HR- und Zeitaufwand, und damit der ROI, alle

bisherigen Negativrekorde sprengen. Letzteres wird aber sehr wahrscheinlich nicht passieren – dafür ist Kubernetes bereits viel zu groß, viel zu gewaltig und nahezu überall präsent. Aber wer weiß.

Um diesem omnipräsenten Permanent-Change zumindest grob folgen zu können, erfährt dieses Buch in seiner vierten Auflage daher auch eine komplette, umfassende Neuausrichtung.

Denn jede meiner bisherigen Publikation zum Thema Kubernetes-basierter Container-Cluster konnte dank der extremen und weiterhin ungebremsten Volatilität bisher immer nur eines sein – eine Momentaufnahme.

Und genau aus diesem Grunde habe ich dieses Mal, wie auch bereits in meiner letzten KI/ML-Infrastruktur-Publikation, einen anderen Ansatz gewählt: eine Ausrichtung, die etwas weniger mit praktischen Beispielen bis ins kleinste Bit hinterlegt ist, sondern stattdessen mehr auf Theorie und vor allem auf Strategie im Unternehmensumfeld setzt und fokussiert, um die Halbwertzeit der Informationen zumindest etwas zu optimieren.

In eigener Sache

Und damit die Leserinnen und Leser, die mich noch nicht aus meinen acht bisherigen Publikationen kennen, wissen, von wem die Informationen in diesem Buch stammen: Ich bin jemand, den andere vielleicht als Spezialisten im RZ-/Cloud-/Großkunden-Bereich für High-Availability-Cluster, Software-Defined Storage, Verzeichnisdienste sowie für hochskalierbare, vollautomatisierte Container-Cluster und GPU-beschleunigte Microservice-Infrastrukturen von großen Unternehmen und international operierenden Konzernen bezeichnen würden. Als Systemarchitekten und oft genug auch als Problem-Fixer für bestimmte Bereiche der IT von Unternehmen und Konzernen.

Aber im Grunde bin ich unter dem Strich nichts anderes als ein IT-Veteran, der unzählige Hypes hat kommen und gehen sehen. Und genau deswegen geht es mir, wie in all meinen bisherigen Publikationen, vor allem darum, eine möglichst realistische Einschätzung abzuliefern. Darüber, wo wir im Bereich skalierbarer, vollautomatisierter Container-Infrastrukturen auf Kubernetes-Basis aktuell stehen und für wen sich der Einsatz welches Kubernetes-Derivats auf welcher Plattform lohnen kann.

Gehen wir ans Werk.

1.1 Vorbemerkungen

Im Buch finden sich vermehrt Anglizismen (welche in der Regel auch erklärt werden), die sich im Rahmen aktueller und fachspezifisch fortgeschrittener Applikationen/Publikationen ohne kilometerlange und oft unpassende deutsche Um- und Beschreibungen oft nicht umgehen lassen.

1.1.1 Verwendete Formatierungen

Die in diesem Buch verwendeten Formatierungen schlüsseln sich auszugsweise wie folgt auf:

Kommandozeilenbefehl

```
Ausgabe (STDOUT/STDERR)
```

```
Listings von Konfigurationsdateien
Änderungen in Konfigurationsdateien
```

URLs

Dateien und Pfade

Tasteneingabe: ↵

Wichtige Hinweise
Hinweistext

1.1.2 Weiterführende Hinweise

Ungeachtet der Tatsache, dass dieses Buch, wie seine Vorgänger, erneut relativ umfangreich geworden ist, bieten viele der behandelten Themen noch mehr Stoff und Details.

Um dieser Tatsache – wie auch der schnellen Evolution des Themas – gerecht zu werden, habe ich an den relevanten Stellen der jeweiligen Abschnitte wie üblich meist weiterführende Links zu den betreffenden Themen eingefügt.

Aufgrund der extrem schnellen Evolution kann leider nicht verbindlich sichergestellt werden, dass nach der Veröffentlichung des Buchs noch jeder Link passt. Entsprechende korrespondierende Referenzen sollten dennoch jederzeit zu finden sein.

1.1.3 Klartext

Wie in all meinen Publikationen werden nicht nur Vorteile, sondern explizit auch die Nachteile der vorgestellten Produkte/Stacks aufgezeigt, und dies in der Regel im »Klartext«. Das bedeutet, es wird nicht um den heißen Brei herumgeredet, sondern klar, unmissverständlich, ausdrücklich und zu einem kleinen Teil auch wiederholt aufgezeigt, wo und warum Probleme existieren.

Damit konnten sich in der Vergangenheit nicht alle Leser anfreunden, aber es war und ist nach wie vor nicht meine primäre Intention, es jedem recht zu machen. Es geht daher ausdrücklich nicht um eine möglichst großflächige Bedienung einer jedweden persönlichen Befindlichkeit, sondern es muss vielmehr wie immer nur darum gehen, ein realistisches Bild *einprägsam* aufzuzeigen – umso mehr für Leser, die mit Themen dieses Buches bisher kaum Berührungspunkte hatten. Denn insbesondere fehlerhaft geplante und implementierte ska-

lierbare Container-Infrastrukturen besitzen, wie im Vorwort bereits angerissen, erfahrungsgemäß ein hohes Potenzial zur Budget-Vernichtung eines Unternehmens.

Und wie bisherige Erfahrungen ebenfalls gezeigt haben, ist es wichtig, die möglichen, teuren Stolpersteine auch ausdrücklich und gegebenenfalls wiederholt und damit unmissverständlich aufzuzeigen, da andernfalls gern »falsch (und damit oft teuer) abgebogen« wird.

Einige Leser bemängelten in früheren Ausgaben zudem, dass einige Kubernetes-3rd-Party-Tools bzw. -Themengebiete wie z. B. Keycloak oder Dynatrace nicht ausführlich genug behandelt wurden. Jeder mit einem noch halbwegs intakten Realitätsbewusstsein sollte verstehen, dass selbst in einem über 1.000-seitigen Kompendium über ein extrem komplexes Thema bestimmte Subthemen, die selbst wiederum eigene Publikationen füllen würden und können, nicht in epischer Breite behandelt werden können.

Wem das Vorgenannte nicht zusagt, dem wird ausdrücklich empfohlen, zu anderer Literatur zu greifen.

1.2 Kernziele und rote Fäden

Das Buch orientiert sich an 2 generellen »roten Fäden« bzw. Kernzielen.

- Von Kleinsten ins Größte/Komplexeste: Das Buch führt von einigen (extrem kompakt gehaltenen) Container-Grundlagen über Pods, weitere Metahüllen für Container, Service-Kommunikation etc. zu immer komplexeren, darauf aufbauenden Themen, die bis hin zu vollautomatisierten, resilienten, Operator-gestützten Core- und Applikationsstacks, CI/CD-GitOps-Pipelines, intelligenten und extrem wichtigen Security-Lösungen und autoskalierbaren Clustern führen.
- Die drei Säulen der Vollautomation für maximale Kosteneffizienz: Auch, wenn diese Botschaft noch nicht überall angekommen ist: Abgesehen von Testsystemen sind Container-Cluster kein Spielplatz für Entwickler, sondern Mittel zum Zweck, damit das Unternehmen maximalen Umsatz generieren und sich im Wettbewerb behaupten kann. Damit das gelingt, müssen die 3 Säulen der Vollautomation – in Verbindung mit LTS-(Long Term Support-)tauglichen Kubernetes-Derivaten – beachtet werden. Im Klartext: Der/die Cluster müssen mithilfe der drei folgenden Punkte maximal optimiert und vollautomatisiert werden:
  a) *Infrastruktur-Vollautomation*, die im Optimalfall von der Container-Plattform selbst gesteuert werden kann. Das ist unerlässlich für autoskalierbare Container-Infrastrukturen und Disaster Recovery.
  b) *In-Cluster-Vollautomation der Core-, 3rd-Party- und eigenen Applikations-Stacks mit Operatoren*, für maximale Resilienz und Effizienz
  c) *Operator-gestützte, vollautomatisierte CI/CD-GitOps-Systeme*

1.3 Was dieses Buch sein soll und was nicht

- **Was es sein soll**

  Ein praxisorientierter Leitfaden für Admins, Container-Teams und Entscheider, der neben praktischen Beispielen aus dem Bereich der Container-Cluster/Orchestrierung im Unternehmensumfeld auch die Hintergründe zu den behandelten Konzepten, Verfahren, Strategien und Tools durchleuchtet. Betrachtungen aus der »8 Miles Above«- bzw. Vogelperspektive sollen, ebenso wie einige Deep-Dives in bestimmte Techniken, ein besseres Verständnis ermöglichen.

- **Was es nicht sein soll und nicht ist**

  Dieses Buch ist ausdrücklich kein Handbuch für Container-Neulinge oder ein Nachschlagewerk für Entwickler, das ausschließlich auf Dev-spezifische Belange fokussiert.

Achtung

Diese Auflage des Buches ist ausdrücklich nicht mehr für komplette Container-Neulinge geeignet.

1.4 Wie dieses Buch zu lesen ist

Bedingt durch die Neuausrichtung und den ausdrücklichen Scope auf Kubernetes im Unternehmenseinsatz entfallen die in früheren Auflagen noch vorhandenen Kapitel zu Container-Basics sowie zu den Docker-Replacement-Tools wie Podman, Skopeo und Buildah komplett. Gleiches gilt auch für viele andere Basic-Bereiche (Was ist DevOps, Funktionsweise von Registries, Grundlagen zu Key/Value-Stores etc.). Das Setup eines Kubernetes-Clusters per `kubeadm` fällt in den Heimanwender-Bereich und ist daher ebenfalls nicht mehr Bestandteil dieser Auflage.

Seit Jahren etablierte Standard-/Core-Workloads wie z. B. Pods (und deren Standard-Metahüllen), Services etc. bzw. deren Basis-Funktionalitäten werden nicht mehr – wie in früheren Auflagen – bis in das letzte Bit in epischer Breite aufgedröselt, denn diese Standard-Infos finden sich vollumfänglich in der Kubernetes-Dokumentation. Dort, wo fortgeschrittene Konzepte zu den jeweiligen Workloads oder angrenzenden Themengebieten ins Spiel bzw. zum Einsatz kommen, geht es natürlich tiefer ins Detail. Stattdessen liegt der Fokus auf Verfahren und fortgeschrittenen Konzepten, Tools und Strategien, die für einen (kosten-)effizienten und vor allem sicheren Betrieb von Kubernetes-basierten Container-Clustern im Unternehmensumfeld unabdingbar sind.

1.4.1 Neue Gliederung und Day 0-1-2-3 Operations

Im Zuge der 5. (inklusive der englischsprachigen Ausgabe) Überarbeitung erhält dieses Buch erstmals eine komplett neue Gliederung der Kapitel. Diese ist nun nicht mehr auf den rein technischen Aufbau fokussiert, sondern orientiert sich an der realen Arbeitsweise und zeitlichen Logistik der DevOps/Operations-Abteilungen.

Die sehr grobe Gliederung stellt sich (hier rein exemplarisch) in etwa wie folgt dar:

- *Day 0 Operations* – Planung und Strategie-Konzepte: Cloud vs. On-Prem, Hybrid, Multi-Cloud, Security- und Kostenfaktoren, Setup-Varianten, LTS-Aspekte, Kubernetes-Foundations, Cluster-Komponenten und Arbeitsweise
- *Day 1 Operations* – Rollout der Cluster, CLI-Tools, apiVersions/-Resources, Core Workloads und Konzepte: Pods, Deployments, Services, Ingress, ConfigMaps, Secrets, RBAC-Objekte etc.
- *Day 2 Operations* – tiefergehende Konzepte und Tools: Healthchecks, Compute Ressources, [De-]Scheduling, Monitoring/Metrics, Logging, APM, Operatoren etc.
- *Day 3 Operations* – fortgeschrittene Konzepte und Tools: Cluster Federation, Backup/Disaster Recovery, Operator-Build, Security-Appliances, vollautomatisierte CI/CD-GitOps-Systeme, autoskalierbare GPU-Cluster und etliches andere mehr

1.4.2 Weniger Detailschritte, Listings und Outputs, mehr Automation

Im Vergleich zu den Vorgängerversionen ist in dieser Auflage, wie bereits im Vorwort angerissen, nicht mehr jedes kleine CLI-Setup-Kommando, jeder zugehörige Output und jedes Listing in jedem Abschnitt haarklein bis in die letzte Schraube aufgeführt:

- zum einen, weil die Halbwertzeiten in der Kubernetes-Welt einfach zu gering sind, viele der gezeigten Outputs bereits beim Erscheinen des Buches ein Stück veraltet sind und daher das Verständnis der Konzepte, ihrer Vor- und Nachteile in der Regel einen deutlich höheren Stellenwert hat.
- zum anderen, weil ich mich aus mehreren Gründen entschlossen habe, die meisten komplexeren Setups zu scripten bzw. zu automatisieren: Dies spart Ihnen Zeit beim Setup und ermöglicht zudem eine deutlich bessere Reproduzierbarkeit und auch einfachere Portierbarkeit auf neue Versionen und andere Umgebungen.

1.4.3 Kapitel und Zielgruppen im groben Überblick

Grundsätzlich sind alle Einleitungen zu den Hauptabschnitten wie auch das jeweilige Fazit ebenfalls für Entscheider geeignet, da es in ihnen auch um Zukunftsperspektiven, konzeptionelle Ansätze und andere Punkte in unternehmensspezifischen Bereichen geht. Ebenso können die Einführungen und Schlussbetrachtungen zu fast allen Kapiteln ebenfalls relevant sein.

Aufgrund der generischen Ausrichtung des Buchs insbesondere auf Admins/Container-Teams (mit Schwerpunkt auf *Ops*) sollten alle Kapitel für sie relevant sein. Bei entsprechenden Vorkenntnissen können die einführenden Kapitel gegebenenfalls übersprungen werden.

1.5 Docker-Replacement-Tools

In den ersten Auflagen dieses Buches (2017, 2018) hatte ich zunächst *Docker* und in den Auflagen 2019 [englische Ausgabe] und 2020 die im Kubernetes-Bereich an seine Stelle getretenen Docker-Replacement-Tools wie *Podman*, *Skopeo* und *Buildah* ausführlich vorgestellt.

Durch den geänderten Scope des Buches und seine damit einhergehende Neuausrichtung und komplette Restrukturierung fokussieren die Inhalte primär nur noch auf Kubernetes im (fortgeschrittenen) Unternehmenseinsatz und nicht mehr auf Container-Basics und die zugehörigen Tools. Daher sind so gut wie alle Inhalte über Container-Basics und umfassende Erläuterungen zu den vorbenannten Tools nicht mehr Bestandteil dieser Auflage.

Nachstehend finden Sie eine kompakte Übersicht zu den Tools sowie weiterführende Links zu den jeweiligen Themen.

1.5.1 Podman

Das Tool der Wahl für das Standalone-Image-, Container- und Pod-Management ist Podman. Es beherrscht neben den üblichen Pull- und Run-Fähigkeiten unter anderem Build-, Push- und Signing-Funktionalitäten. Podman ist kompatibel mit fast allen Docker-CLI-Befehlen, beherrscht aber zusätzlich das unter Docker unbekannte, aber für Kubernetes elementar wichtige *Pod*-Konzept (siehe dazu Abschnitt 7.3).

Zur Vertiefung der Podman-Thematik sei das Red-Hat-Developer-Buch *Podman in Action* (2023, ca. 300 Seiten) von einem der fähigsten Red-Hat-Ingenieure im Container-Umfeld empfohlen: Dan Walsh. Das Buch ist – sofern ein Red-Hat-Account vorhanden ist – kostenlos und kann z. B. hier heruntergeladen werden:

- *https://developers.redhat.com/e-books/podman-action*

1.5.2 Buildah

Buildah ist ein Red-Hat-Tool mit Fokus auf Image-Builds, es kommt auch gern containerisiert in CI-Pipelines als Task zum Einsatz. Grundlegende Funktionalitäten für den Container-Run sind enthalten. Dokus und Tutorials finden sich unter anderem hier:

- *https://github.com/containers/buildah/tree/main/docs/tutorials*
- *https://developers.redhat.com/blog/2021/01/11/getting-started-with-buildah*

1.5.3 Skopeo

Skopeo ist der ideale, weil weitaus intelligentere *Push*-Ersatz. Es kann unter anderem Images von und in verschiedenste Sourcen und Targets *kopieren* (ohne vorher notwendiges Image-Tagging) und dabei auch signieren. Es kann ebenfalls als eine Art »Image-rsync« für das Kopieren vieler Images (optional unter Beibehaltung der Hierarchie) von A nach B verwendet werden. Skopeo wird in späteren Abschnitten gegebenenfalls fallweise kurz behandelt.

- *https://github.com/containers/skopeo*
- *https://www.redhat.com/de/topics/containers/what-is-skopeo*

Kapitel 5
Kubernetes-Cluster-Setups (Cloud)

»Automatisierung, die auf einen ineffizienten Vorgang angewendet wird, vergrößert die Ineffizienz.«
– Bill Gates

Wer schnell zum Ziel, d. h. zu einem funktionierenden Kubernetes-Cluster kommen will, sollte sich mit einer Bereitstellung von Kubernetes-Clustern in der Cloud anfreunden. Systeme wie *EKS* (Elastic Kubernetes Service, AWS), *AKS* (Azure Kubernetes Service, Microsoft) oder *GKE* (Google Kubernetes Engine) lassen sich relativ einfach aufsetzen und eignen sich je nach gewählten Instanztypen und Anzahl der Nodes auch für ein eher schlankes Budget.

Die üblichen Mankos der »Managed-Kubernetes«-Angebote in der Cloud sind folgende: Die Controlplane-Komponenten liegen üblicherweise nicht im Zugriff des Kunden, und jeder kocht bei nicht-Kubernetes-nativen Addons, Cluster-Autoscalern und Integrationen mit Diensten der jeweiligen Hyperscaler-Plattform sein ganz eigenes, nicht auf andere Plattformen portierbares Süppchen.

Auch wenn unzählige Kontributoren am Kubernetes-Projekt mitarbeiten, steuert Google, neben Red Hat, die größten Anteile zur Kubernetes-(Weiter-)Entwicklung bei. Verglichen mit anderen Managed-Kubernetes-Cloud-Angeboten wie Amazons EKS oder Microsofts AKS ist GKE daher bezogen auf die Features, Usability, Maturität und Setup-Automation die mit großem Abstand am weitesten entwickelte Plattform. Das ist nicht wirklich verwunderlich, da – wie gerade schon angerissen – ein nicht unbeträchtlicher Teil der Kubernetes-DNA ihren Ursprung eben bei Google hat.

Aus einer objektiven Perspektive ist es daher relativ unverständlich, warum sich GKE mit AKS um die Plätze hinter Amazons EKS prügeln muss – denn technisch betrachtet liegt das Ranking komplett andersherum. Erfahrungsgemäß spielt das jedoch auf den Entscheider-Ebenen der meisten Unternehmen nur eine geringe bzw. keine Rolle. Dort geht es wie üblich in der Regel um bestehende Verträge und damit um Provideraffinität und das billigste Angebot, das Ganze garniert mit einer hart erarbeiteten Portion Beratungsresistenz.

Im Folgenden sind exemplarisch und kompakt Setup-Verfahren für Managed-Kubernetes-Systeme in der Google Cloud sowie unter Azure und AWS beschrieben.

Achtung: Kontingente

Beachten Sie, dass Sie bei Ihrem Cloud-Provider üblicherweise die Default-Ressourcen-Quotas bzw. -Kontingente für z. B. vCPUs, GPUs, Storage, LB-IPs und etliches andere mehr überprüfen und erhöhen müssen, da diese in der Regel in der Standardeinstellung zu gering für halbwegs produktivtaugliche Cluster ausgelegt sind.

Achtung: KubeConfig-Verhalten bei AKS und EKS

Im Gegensatz zu GKE verwenden AKS und EKS bei der Erzeugung und Ablage der Zugangs-Credentials in der KubeConfig-Datei etwaig vorhandene KUBECONFIG-Settings und mergen ihre Settings mit denen einer bereits vorhandenen KubeConfig-Datei. In konkreten, mehrfach reproduzierbar getesteten Szenarien wurden dadurch bereits vorhandene KubeConfig-Dateien von anderen Kubernetes/OpenShift-Clustern unbrauchbar.

5.1 GKE

In diesem Abschnitt werden wir Cloud-Hosted/Managed-Kubernetes am Beispiel von *GKE* (*Google Kubernetes Engine*) betrachten.

Da GKE-Cluster auch in späteren Abschnitten für Beispiel-Setups zum Einsatz kommen, ist dieser Abschnitt der ausführlichste der Managed-Kubernetes-Angebote.

Das Management von Kubernetes wird in späteren Kapiteln bis in die tiefsten Tiefen durchexerziert. Daher beschränken sich die folgenden Abschnitte an dieser Stelle lediglich auf die Verfahren und Konzepte, die die Installation eines GKE-Clusters betreffen, sowie auf entsprechende zonale und regionale Konzepte des GKE-Angebotes. Ähnliches gilt für die kompakt gefassten Betrachtungen zu AKS und EKS.

5.1.1 GKE – Google Kubernetes Engine

Für den Endverwerter des GKE-Angebotes stellt sich der Kubernetes-Cluster weitestgehend wie eine Standard-Kubernetes-Implementierung dar. Ausgenommen davon sind natürlich u. a. die üblichen, cloudbedingten Abweichungen, beispielsweise die – zumindest auf nicht-Anthos-basierten Systemen – in der Regel versteckten Controlplane-Komponenten bzw. Master-Instanzen (siehe weiter unten). Der GKE-Cluster lässt sich per UI über die Google Console oder per `kubectl` über die ab Abschnitt 6.5 vorgestellten Verfahren administrieren, entweder über einen lokalen Bastionshost im privaten oder Unternehmensnetz oder beispielsweise über eine VM innerhalb des GCP-Projekts, in dem der GKE-Cluster läuft.

Als übergeordnetes Tool, keinesfalls nur zum Ausrollen von GKE-Clustern, steht die funktional extrem umfangreiche **gcloud**-CLI zur Verfügung, deren Setup und Handling im Folgenden auszugsweise beschrieben wird. Ein guter Einstieg zu `gcloud` findet sich hier:

https://cloud.google.com/sdk/gcloud

Ein wichtiger Unterschied bei der Betrachtung eines GKE-Clusters sind die aus der Sicht des Admins scheinbar nicht vorhandenen Master-/Controlplane-Nodes. Diese sind, genau wie bei den Konkurrenzangeboten von Microsoft und Amazon, für den Kunden nicht direkt zugänglich, sondern integraler Bestandteil der GKE und werden ausschließlich von Google gemanagt.

Was der Kunde im GKE-Angebot ursprünglich bezahlen musste, waren in der Vergangenheit nur seine Worker-Nodes. Seit Juli 2020 fallen ebenfalls Kosten für die Master an, wie schon länger gang und gäbe unter AWS. Im betrachteten Stand beliefen sich die Kosten auf 0,10 USD/Stunde. Sofern Sie das Setup über die Google UI vornehmen, wird Ihnen zu den jeweils gewählten Setup-Optionen eine Kostenübersicht angezeigt, die Ihnen die geschätzten Kosten für den gewählten Cluster pro Monat anzeigt.

5.1.2 Regionen, Zonen und Verfügbarkeiten

Google unterteilt seine Cluster in regionale und zonale Cluster. Unter einer *Region* versteht man eine geografische Lokation, z. B. die Region us-west2, womit die Google-RZ-Lokationen in der Nähe von Los Angeles an der Westküste der Vereinigten Staaten gemeint sind. Jede Region ist üblicherweise in (Verfügbarkeits-)*Zonen* unterteilt, bezogen auf die o. a. Region us-west2 also beispielsweise us-west2-a, us-west2-b, us-west2-c, was heißt: verschiedene Rechenzentren im Großraum von L.A. Die Region europe-west3 bezeichnet beispielsweise die Google-Region im Großraum Frankfurt am Main, mit den dort verfügbaren Availability-Zonen a, b und c. Beachten Sie, dass nicht jede Instanz-Art bzw. Hardware-/CPU-/GPU-Plattform in jeder Region verfügbar ist.

Unter *zonalen Ressourcen* würde man im Google-Sprech einen Cluster verorten, dessen Nodes nur innerhalb einer einzigen Zone (bezogen auf Frankfurt z. B. europe-west3-c) arbeiten würden. Diese Ressourcen besitzen logischerweise die geringste Verfügbarkeit bzw. höchste Ausfallwahrscheinlichkeit. *Regionale Ressourcen* definieren Cluster, die in mehreren Zonen (in der Regel die benannten drei: a, b, c) einer einzelnen Region arbeiten, folglich mit einer besseren Verfügbarkeit als zonale Cluster. Die höchste Ausfallsicherheit bieten *multiregionale Ressourcen*, die sich über mehrere Regionen und die darin enthaltenen Verfügbarkeitszonen erstrecken. Eine Auflistung der Regionen, Zonen und ihrer Lokationen findet sich unter:

https://cloud.google.com/compute/docs/regions-zones

Google garantierte in seinen SLAs im betrachteten Stand für zonale Cluster eine generelle, monatliche Uptime von 99,5 %, entsprechend 215 Minuten »erlaubter« Downtime, die der

Kunde pro Monat akzeptieren muss. Regionale Cluster und Autopilot-Cluster können zumindest mit einer Uptime von 99,95 % aufwarten, was nur noch etwas mehr als 5 Minuten pro Monat entspricht. Details zu den jeweils aktuellen GKE-SLAs finden sich unter anderem hier:

https://cloud.google.com/kubernetes-engine/sla

5.1.3 GKE-Setup-Varianten

Nach der Anmeldung auf der *Google Cloud Platform* (GCP) kann der Container-Service bzw. Container-Cluster auf verschiedene Arten erstellt werden – entweder über manuelle Konfiguration oder per Autopilot, siehe auch:

https://cloud.google.com/kubernetes-engine/docs/concepts/types-of-clusters.

Im Folgenden wird aus Gründen der einfachen Reproduzierbarkeit nur noch die CLI-Variante per `gcloud` behandelt, da diese zudem gescriptet automatisiert werden kann.

5.1.4 GKE Shielded Nodes

Bereits beginnend mit GKE-Kubernetes-Version 1.13.6 konnte man den über den `gcloud`-Schalter `--enable-shielded-nodes` den sogenannten *Shielded Mode* aktivieren. Sobald der Shielded Mode aktiviert ist, verifiziert das GKE-Controlplane kryptografisch, dass jeder Node des Clusters eine VM ist, die als Bestandteil einer *Managed Instance Group* in einem Datacenter von Google läuft, und dass die Kubelets mit passenden, zum Cluster gehörenden Zertifikaten ausgestattet sind.

Das klingt zunächst einmal sehr generisch und angriffstechnisch wenig besorgniserregend, wird aber deutlicher, wenn man hinter die Kulissen schaut. Denn ohne den Shielded Mode besteht – bedingt durch den funktionalen Aufbau von GKE-Clustern – potenziell die Möglichkeit, dass ein Angreifer über einen normalen Pod in den Besitz der *Bootstrapping Credentials* kommen kann (Stichwort: das `kube-env`-Attribut, das diese beinhaltet) – und damit in Folge an die Secrets des Clusters, was letztlich in einer kompletten Übernahme des Clusters resultieren kann: *kube-env-Stealer*.

Kurzum: Der Zugriff auf diese Meta-Informationen der GKE-Instance ist ohne Shielded Mode jederzeit möglich. Zudem validiert ein GKE-Controlplane im Non-Shielded Mode nicht, ob der Hostname in einem CSR mit dem übereinstimmt, der ursprünglich das Bootstrapping-Keypair angefordert hat. Siehe zu diesem Thema auch:

https://cloud.google.com/kubernetes-engine/docs/how-to/shielded-gke-nodes

Seit der GKE-Kubernetes-Version 1.18 ist der Shielded Mode die Standardeinstellung, sofern man diese nicht explizit bei der Cluster-Erzeugung abwählt (`--disable-shielded-nodes`) oder im laufenden Betrieb abschaltet (`gcloud container clusters update [...] --no-enable-shielded-nodes`). Das mit dem Shielded Mode korrespondierende *Integritätsmonitoring* ist ebenfalls

per Default aktiviert (`--shielded-integrity-monitoring`) und kann beim Erstellen eines Clusters oder beim Erstellen eines Node-Pools per gcloud über den Switch `--no-shielded-integrity-monitoring` deaktiviert werden.

Achtung: Shielded Mode

Sobald der Shielded Mode aktiviert ist, können Nodes, die in einem Node-Pool ohne Shielding bzw. außerhalb eines Node-Pools erstellt wurden, dem Cluster nicht hinzugefügt werden.

5.1.5 Verfügbare Maschinen bzw. Instanz-Typen in der GCP

Unter der URL *https://cloud.google.com/compute/docs/machine-resource?hl=de* finden Sie die für GKE-Cluster auf der Google Cloud Platform verfügbaren Instanztypen. Diese waren im betrachteten Stand auf einer großen Flughöhe in vier Kategorien unterteilt :

- **Allgemeine Zwecke** – Multi-Purpose-Instanzen für verschiedene Workloads, ohne dedizierten Spezial-Einsatzzweck (Instanz-Typen z. B. C3, E2, N*). Siehe dazu auch: *https://cloud.google.com/compute/docs/general-purpose-machines*
- **Computing-optimiert** – die höchste Leistung pro Kern in Googles Compute Engine, für rechenintensive Arbeitslasten optimiert (Instanz-Typ C2*). Siehe dazu auch: *https://cloud.google.com/compute/docs/compute-optimized-machines*
- **Speicheroptimiert** – für arbeitsspeicherintensive Arbeitslasten, mit mehr Arbeitsspeicher pro Kern als bei anderen Maschinenfamilien und bis zu 12 TB Arbeitsspeicher (Instanz-Typen z. B. M1, M2, M3). Siehe dazu auch: *https://cloud.google.com/compute/docs/memory-optimized-machines*
- **Beschleunigungsoptimiert/GPU-Nodes** – für parallelisierte CUDA-(*Compute Unified Device Architecture*-)Computing-Arbeitslasten, z. B. maschinelles Lernen (ML) und Hochleistungs-Computing (HPC). Instanz-Typen wären z. B. die A2- und G2-Instanzen. Siehe dazu auch: *https://cloud.google.com/compute/docs/accelerator-optimized-machines*

5.1.6 Mindestangaben für das Setup

Für die Erzeugung eines Clusters müssen minimal die folgenden Angaben mit auf den Weg gegeben werden:

- der Name des Clusters
- der Standort-Typ, d. h. welchen Grad die Verfügbarkeit haben soll (regionale bzw. überregionale [zonale] Verteilung der Master)
- in welcher Region der Cluster ausgerollt werden soll (Asien, Europa, USA usw.)
- welche Kubernetes-Version verwendet werden soll. Im zuletzt betrachteten Stand (08/2023) konnte z. B. die Kubernetes-Version 1.27.x als Stable-Version zum Einsatz kommen.

Weitere optionale Parameter, die aber dennoch gesetzt werden sollten, wären z. B. wie viele Nodes der Cluster pro Zone umfassen soll, welcher Instanztyp verwendet werden soll, Anzahl und Typ der GPUs, Größe und Art (z. B. SSD) des Bootlaufwerks etc. Ebenso kann eine automatische Cluster-Skalierung aktiviert werden (Cluster-Autoscaler, erzeugt neue Worker-Nodes on-demand). Die laufenden Kosten richten sich natürlich nach den gewählten Instanztypen und diversen anderen Faktoren. Infos zu den Preisen sowie einen Pricing-Kalkulator liefert Google z. B. hier:

- *https://cloud.google.com/kubernetes-engine/pricing*
- *https://cloud.google.com/products/calculator/*

5.1.7 Googles Node-OS-Varianten

Im betrachteten Stand waren als Betriebssysteme für die Nodes entweder Googles *Container-Optimized OS* (CoS, typischerweise mit cri-containerd) und Ubuntu (ebenfalls üblicherweise mit cri-containerd) verfügbar. Für die Unverbesserlichen existiert auch noch die Möglichkeit, Windows-Server-Nodes als Worker einzubinden. Hinter CoS (oder auch cos geschrieben) versteckt sich nichts Spektakuläres, sondern nur ein weiteres abgespecktes Linux-OS mit angepasstem Kernel, einem Cloud-Init zur Provisionierung, einer Container-Engine, Kubernetes und weiteren Tools. Sollen die Cluster-Nodes Storage z. B. per Ceph unterstützen oder andere Aufgaben bewältigen, ist Ubuntu – je nach betrachtetem Stand – als (einzige) Alternative im GKE-Angebot die Präferenz.

Details zu den Node-Images finden sich unter anderem hier:

https://cloud.google.com/kubernetes-engine/docs/concepts/node-images

5.1.8 Auszüge sonstiger einstellbarer Features

Des Weiteren kann eingestellt werden, welche Instanz-Typen zum Einsatz kommen, welche Upgrade-Strategien (maxSurge, Blue/Green) verwendet werden sollen und ob automatische Node-Upgrades aktiviert sind – per Default können sie zumindest im Standard-Channel ausgeschaltet werden.

Als weiteres Feature steht eine automatische Node-Reparatur zur Verfügung. Dabei wird z. B. ein Node, der fortgesetzt den Status *NotReady* oder gar keinen Status meldet oder keinen freien Speicherplatz mehr auf dem Bootlaufwerk vorweisen kann, »repariert«. Bei dieser Reparatur wird im Prinzip nichts anderes getan, als den Node zu räumen, ihn anschließend neu zu erstellen und dem Cluster wieder hinzuzufügen. Da auch parallele »Reparaturen« unterstützt werden, wird das Ganze in kleineren Clustern spannend, wenn mehrere Nodes fehlerhaft sind und alle Pods evicted werden. Außerdem finden sich – je nach Version – in der Regel noch Optionen zur Aktivierung der *Cloud-Operation-Stacks* (Monitoring/Logging) für GKE, siehe dazu auch: *https://cloud.google.com/stackdriver/docs/solutions/gke*

Weitere Customizing-Optionen sind noch die Größe und Art des Boot-Drives, die Freischaltung von Alpha-Features, generelle Netzwerkeinstellungen (virtuelle IPs/VPCs und IP-Ranges für Pods), die Loadbalancer-Aktivierung (Default: true), die Sperrung nicht vertrauenswürdiger Quell-IPs, die Aktivierung von NetworkPolicies, zertifikatsrelevante Einstellungen und anderes.

Im Bereich der Cluster-Automatisierung können Update-Fenster angegeben werden, Cluster-Autoscaling (siehe Abschnitt 11.12) und/oder vertikales Pod-Autoscaling (siehe Abschnitt 11.10) aktiviert werden. Als Security-Features können die bereits angesprochenen Shielded Nodes aktiviert werden (Default: `true`), Confidential-GKE-Nodes eingerichtet (verschlüsselte VMs) oder Secrets verschlüsselt (siehe Abschnitt 20) sowie etliche andere Security-relevante Features aktiviert werden.

Einige der vorbenannten Setup-Optionen behandeln wir im nun folgenden Abschnitt.

5.1.9 gcloud – CLI-basierte Cluster-Installation

Der Schlüssel für alle folgenden Aktionen ist das Tool **`gcloud`**. Es verfügt über umfangreiche Suboptionen, wie die für uns in diesem Kontext zunächst relevanten **`compute`** und **`container`**.

Zunächst muss auf dem Installer-/Bastions-Node das *gcloud-SDK* installiert werden. Dies kann unter RHEL 8 z. B. über die Einbindung des entsprechenden Repos und eine anschließende normale Installation per `yum` oder `dnf` vonstattengehen:

```
# vi /etc/yum.repos.d/google-cloud.repo
```

```
[google-compute-engine]
name=Google Compute Engine
baseurl=https://packages.cloud.google.com/yum/repos/google-compute-engine-el8-x86_64-stable
enabled=1
gpgcheck=1
repo_gpgcheck=0
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
[google-cloud-sdk]
name=Google Cloud SDK
baseurl=https://packages.cloud.google.com/yum/repos/cloud-sdk-el8-x86_64
enabled=1
gpgcheck=1
repo_gpgcheck=0
gpgkey=https://packages.cloud.google.com/yum/doc/yum-key.gpg
       https://packages.cloud.google.com/yum/doc/rpm-package-key.gpg
```

Anschließend können Sie das gcloud-SDK wie folgt installieren:

```
# dnf -y install google-cloud-sdk
```

Natürlich wird auch noch das `kubectl`-Paket für den späteren Zugriff auf den GKE-Cluster benötigt, idealerweise in einer – zur später ausgewählten Cluster-Version – passenden Variante.

gcloud-Bash-Completion

Die gcloud-Bash-Completion-Datei liegt nach der Installation des *google-cloud-sdk*-Paketes unter:

- */usr/lib64/google-cloud-sdk/completion.bash.inc*

  und/oder:

- */usr/share/google-cloud-sdk/completion.bash.inc*

In der Regel reicht es aus, eine neue Bash-Instanz zu starten, um die Completion zu aktivieren. Falls dies nicht klappt, müssen Sie eine der Include-Dateien unter */etc/bash_completion.d/* ablegen und dann eine neue Bash-Instanz starten. Danach sollten sich die zur Verfügung stehenden Sub-Optionen wie üblich per ⇆ komplettieren lassen.

gcloud-Befehlsreferenz

Die komplette **gcloud**-Referenz findet sich unter *https://cloud.google.com/sdk/gcloud/reference*.

5.1.10 gcloud init auf dem Verwaltungs-Node

Direkt nach der Installation ist **gcloud** allerdings noch nicht einsatzbereit. Um es verwenden zu können, muss auf dem Node, von dem aus der GKE-Cluster erzeugt und verwaltet werden soll, zunächst ein **gcloud init**-Lauf gestartet werden.

Achtung

Ein *Google-Account* sollte vor dem **gcloud init**-Lauf angelegt werden bzw. bereits vorhanden sein!

Im Verlauf des interaktiven **gcloud init** wird bei einem neuen Setup eine URL eingeblendet, die per Browser aufgerufen werden muss und einen Verifikations-Code für den verwendeten Google-Account enthält:

```
Go to the following link in your browser:
    https://accounts.google.com/o/oauth2/auth?code_challenge=-xx(stark gekürzt)x-accounts.reauth
Enter verification code:
```

Der Link leitet auf eine Website, auf der – sofern vorhanden – der Google-Account des Benutzers angezeigt wird. Alternativ kann eine Anmeldung per E-Mail/Telefon erfolgen. Am Ende des Vorgangs wird ein Verifizierungs-String ausgegeben, der in den **gcloud**-Prompt kopiert werden muss. Danach wählen Sie das gewünschte Projekt aus oder erzeugen ein neues und wählen danach die gewünschte Compute-Region/Zone aus.

Hinweis: Auth per CLI, Credential-Ablage

Ein Authentifizierungsvorgang kann auch separat per **gcloud auth login** getriggert werden.

Die **gcloud**-Credentials werden unter *~/.config/gcloud/credentials* abgelegt.

5.1.11 Installierte/verfügbare gcloud-Komponenten auflisten bzw. nachinstallieren

gcloud versteht sich, abstrahiert ausgedrückt, als modulares System mit Plugins, die nachträglich installiert werden können. Eine Liste der verfügbaren und bereits installierten Plugins liefert:

```
# gcloud components list
```

Leider lassen sich etliche der Komponenten nicht per **gcloud components install <Component>** installieren, wenn das Google Cloud SDK über das Paketmanagement der jeweiligen Linux-Distribution installiert wurde, sondern nur per `yum`/`dnf` etc. über das eingebundene Repo:

```
# dnf search google-cloud
```

```
google-cloud-sdk.noarch : Google Cloud SDK
google-cloud-sdk-app-engine-go.x86_64 : Google Cloud SDK
google-cloud-sdk-app-engine-grpc.x86_64 : Google Cloud SDK
google-cloud-sdk-app-engine-java.noarch : Google Cloud SDK
…
google-cloud-sdk-firestore-emulator.noarch : Google Cloud SDK
google-cloud-sdk-kind.x86_64 : Google Cloud SDK
google-cloud-sdk-kpt.x86_64 : Google Cloud SDK
google-cloud-sdk-minikube.x86_64 : Google Cloud SDK
google-cloud-sdk-pubsub-emulator.noarch : Google Cloud SDK
google-cloud-sdk-skaffold.x86_64 : Google Cloud SDK
google-cloud-sdk-tests.noarch : Google Cloud SDK
```

5.1.12 Container-API und Billing im Projekt aktivieren

Wird ein neues Projekt angelegt, sollte dafür die Compute-Engine-API aktiviert werden, andernfalls kann per **gcloud** und/oder GUI gar nicht bzw. nur eingeschränkt darauf zugegriffen werden. Die Aktivierung ist per GUI oder alternativ per **gcloud** über eine Verlinkung des *Billing-Accounts* mit dem Projekt möglich:

```
# gcloud beta billing accounts list
```

```
ACCOUNT_ID            NAME                OPEN  MASTER_ACCOUNT_ID
XXXX-YYYYY-ZZZZZ      Billing Account 1    True
AAAA-BBBBB-CCCCC      Billing Account 2    True
```

```
# gcloud projects list

PROJECT_ID          NAME         PROJECT_NUMBER
cluster-01-999999   cluster-01   xxxxxxxxxxx
cluster-02-999999   cluster-02   xxxxxxxxxxx
…

# gcloud beta billing projects link --billing-account=XXXX-[…] cluster-04-999999

billingAccountName: billingAccounts/ XXXX-YYYYY-ZZZZZ
billingEnabled: true
name: projects/cluster-04-999999/billingInfo
projectId: cluster-04-999999

# gcloud services enable container.googleapis.com

Operation "operations/acf.xxxxxxx-xxxx-xxxx-xxxxx-xxxxxxx" finished successfully.
```

5.1.13 GKE-Channels (Stable, Regular, Rapid)

Überlegen Sie vor dem Setup, welche GKE-Version zum Einsatz kommen soll. Google bietet GKE-Versionen über verschiedene *Channels* an, die unterschiedliche Kubernetes-Versionen und Stabilitätsgrade reflektieren.

Google nimmt dabei eine grobe Unterteilung in die Channels *Rapid*, *Stable* und *Regular* vor, die in Tabelle 5.1 kurz erläutert werden. Beachten Sie auch die *Upgrade-Policies* von Google für die jeweiligen Channels. Diese sind unter *https://cloud.google.com/kubernetes-engine/docs/concepts/release-channels?hl=de* zu finden.

GKE-Channel	Release-Verfügbarkeit	Eigenschaften
Rapid	Mehrere Wochen nach der allgemeinen Open-Source-Verfügbarkeit der Kubernetes-Version	Für alle mit zu viel Langeweile. Die brandneueste Kubernetes-Version, mit allen neuen Features – und Bugs. Die Cluster werden – per Default – regelmäßig aktualisiert, um die neueste verfügbare Patchversion zu reflektieren. Obwohl laut Google als GA gekennzeichnet, wird ausdrücklich auf den Einsatz in reinen Test-Umgebungen hingewiesen. Zudem sind diese Versionen von der GKE-SLA ausgenommen, da sie in der Regel Probleme enthalten, für die es noch keine Lösungen gibt.
Regular (Standard)	Zwei bis drei Monate nach der Veröffentlichung im Rapid-Channel	Die Kubernetes-Version wurde über einen längeren Zeitraum von Google qualifiziert. Der Channel bietet – laut Google – ein ausgewogenes Verhältnis zwischen neuen Features und Release-Stabilität.

Tabelle 5.1 Übersicht der GKE-Channels

GKE-Channel	Release-Verfügbarkeit	Eigenschaften
Stable Version	Zwei bis drei Monate nach der Veröffentlichung im Regular-Channel	Stabilität hat Vorrang vor neuen Features. Der korrekte Channel für Produktivumgebungen. Alternativ kann auch ein statischer Channel mit einer fixen Version beim Rollout angegeben werden, die von Google nicht aktualisiert werden darf. Neue Versionen in diesem Kanal werden zuletzt publiziert, und zwar erst, nachdem sie auf den Kanälen *Rapid* und anschließend *Regular* validiert wurden.

Tabelle 5.1 Übersicht der GKE-Channels (Forts.)

Um die GKE-Versionen vor dem Setup optimal mit den eigenen Erfordernissen abgleichen zu können, bietet sich sowohl ein Blick in die Kubernetes-Release-Notes als auch in die GKE-spezifischen an:

https://cloud.google.com/kubernetes-engine/docs/release-notes

Um die verfügbaren Versionen per **gcloud** anzugfragen, gehen Sie wie folgt vor (die folgenden Befehle wurden 07/2023 abgesetzt):

Actual Stable Version:

```
# gcloud container get-server-config --flatten="channels" \
  --filter="channels.channel=STABLE" \
  --format="yaml(channels.channel,channels.defaultVersion)"
```

```
[…]
channels:
  channel: STABLE
    defaultVersion: 1.27.3-gke.100
```

Valid Stable Versions:

```
# gcloud container get-server-config --flatten="channels" \
  --filter="channels.channel=STABLE" \
  --format="yaml(channels.channel,channels.validVersions)"
```

```
[…]
  channel: STABLE
  validVersions:
  - 1.27.3-gke.1700
  - 1.27.3-gke.100
  - 1.26.5-gke.2700
  - 1.26.5-gke.2100
```

```
  - 1.25.10-gke.2700
  - 1.25.10-gke.2100
[...]
```

Regular (between Rapid and Stable):

```
# gcloud container get-server-config --flatten="channels" \
  --filter="channels.channel=REGULAR" \
  --format="yaml(channels.channel,channels.validVersions)"
```

```
[…]
  channel: REGULAR
  validVersions:
  - 1.27.3-gke.1700
  - 1.27.3-gke.100
  - 1.26.6-gke.1700
  - 1.26.5-gke.2700
  - 1.25.11-gke.1700
  - 1.25.10-gke.2700
  - 1.24.15-gke.1700
[…]
```

Rapid:

```
# gcloud container get-server-config --flatten="channels" \
  --filter="channels.channel=RAPID" \
  --format="yaml(channels.channel,channels.validVersions)"
```

```
 […]
  channel: RAPID
  validVersions:
  - 1.27.4-gke.900
  - 1.27.3-gke.1700
  - 1.26.7-gke.500
  - 1.26.6-gke.1700
  - 1.25.12-gke.500
  - 1.25.11-gke.1700
  - 1.24.16-gke.500
[…]
```

5.1.14 Cluster-Installation mit Anpassungen

Im Folgenden wird per **gcloud** ein Cluster (Name: `cluster-01`) mit je 3 Master- und (autoskalierbaren) Worker-Nodes mit aktiviertem, NetworkPolicy-fähigem CNI-Plugin (Calico, siehe auch der nächste Abschnitt) in der Region europe-west4 (NL) generiert.

Als Instanz-Typ kommt `e2-standard-4` mit 4 vCPUs, 16 GB RAM, 150 GB OS-Disk, Ubuntu als OS und Kubernetes-Version 1.27.3 zum Einsatz.

Hinweis: kubeconfig

Bei der Erstellung des Clusters wird die Datei *~/.kube/config* für den passenden Cluster-Kontext automatisch im aktuellen Arbeitsordner erzeugt.

Da alle verfügbaren Konfigurationsoptionen zur Erzeugung eines Clusters per `gcloud` den Rahmen dieses Abschnittes sprengen würden, sei an dieser Stelle auch ergänzend auf die entsprechende `gcloud container clusters create`-Referenz verwiesen:

https://cloud.google.com/sdk/gcloud/reference/container/clusters/create

```
# gcloud container clusters create cluster-01 --cluster-version=1.27.3-gke.100 \
  --image-type=UBUNTU_CONTAINERD --disk-size=150GB --num-nodes=1 \
  --machine-type=e2-standard-4 --no-enable-autorepair --no-enable-autoupgrade \
  --enable-network-policy --enable-ip-alias --enable-autoscaling --min-nodes=3 \
  --max-nodes=7 --region=europe-west4 --project=cluster-01-999999
```

Über `--cluster-version` geben Sie die gewünschte Kubernetes-Version auf den Master-Nodes an. Per Default wird bei zonalen Clustern (`--zone=<>` statt `--region=<>`) nur 1 Master-Node mit *n* Workern in 1 Zone erstellt, bei multizonalen Clustern 1 Master mit *n* Workern in *n* Zonen. Bei regionalen Clustern, wie im Beispiel, werden 3 Master-Nodes pro Region, d. h. 1 Master je untergeordneter Zone, erstellt.

Achtung: Späteres Scaling und zonale/regionale Cluster

Die Art und Weise, wie der Cluster erstellt wird (zonal, regional), schreibt auch für spätere Scaling-Aktionen fest, wie der Cluster skaliert werden kann. Provisionieren Sie den Cluster z. B. *regional* (je 1 Master und je 1 Worker für jede der 3 Availability-Zonen), bedeutet dies, dass Sie später als kleinste Scaling-Einheit nur um jeweils um 3, 6, 9 [...] Nodes in dem bestehenden Node-Pool hochskalieren können. Kleinere Granulierungen sind dann nicht mehr möglich.

Für die auszurollenden Worker-Nodes wird per Default die gleiche Kubernetes-Version gesetzt wie auf den Mastern. Wer abweichende Kubernetes-Versionen auf den Worker-Nodes wünscht (in der Regel nicht anzuraten), kann dies per `--node-version` einstellen. Die Anzahl der Worker-Nodes (in diesem Fall Worker-Node pro Zone: a, b, c) wird per `--num-nodes` gesetzt. Wenn Sie Ihre Worker-Nodes mit bestimmten Zone-Affinities setzen wollen, können Sie dies per `--node-locations` einstellen. Hinzu kommen unter anderem die üblichen Verdächtigen, um z. B. IP-Ranges für das Master-, Pod- und Service-IP-Netz zu setzen (`--master-ipv4-cidr`, `--cluster-ipv4-cidr`, `--services-ipv4-cidr`).

Über den Schalter `--machine-type` kann der zu verwendende VM-Typ eingestellt werden – extrem wichtig im Bezug auf Leistungsfähigkeit des Clusters und, natürlich damit korrelierend, die zu erwartenden Kosten. Per `gcloud compute machine-types list` können Sie sich eine (sehr umfangreiche) Liste der VM-Templates bezogen auf die jeweilige Region anzeigen las-

sen und per **gcloud compute machine-types describe <Instance>** etwas detailliertere Infos dazu abrufen.

Post-Rollout stellen sich die Worker-Nodes des Clusters wie folgt dar:

```
# gcloud compute instances list

NAME                                          ZONE            MACHINE_TYPE   PREEMPTIBLE  INTERNAL_IP
    EXTERNAL_IP    STATUS
gke-cluster-01-default-pool-86085e9f-h661  europe-west4-c  e2-standard-4               10.164.0.21
   34.32.210.215  RUNNING
gke-cluster-01-default-pool-8d2102f1-kmv1  europe-west4-b  e2-standard-4               10.164.0.12
   34.32.255.245  RUNNING
gke-cluster-01-default-pool-9e9cd1d8-lhws  europe-west4-a  e2-standard-4               10.164.0.18
   34.32.215.31   RUNNING
```

5.1.15 GKE-Cluster und NetworkPolicies

Soll ein neu auszurollender GKE-Cluster mit NetworkPolicies arbeiten können, war im betrachteten Stand das Flag **--enable-network-policy** beim Setup per **gcloud container cluster create** notwendig.

Soll die Interpretation von NetworkPolicies in einem bestehenden GKE-Cluster nachträglich aktiviert werden, sind die beiden im Folgenden beschriebenen Schritte nötig, welche leider sehr zeitaufwendig sind. Dabei ist zu beachten, dass mit Ausführung des nachstehend aufgeführten 2. Befehls (der Aktivierung des Addons) ein komplettes Rolling Update des Clusters getriggert wird, während dessen der ganze Cluster nicht mehr erreichbar ist.

Die nachträgliche Aktivierung der NetworkPolicies in einem bestehenden Cluster erfolgt über diese Befehle:

```
# gcloud container clusters update \
  cluster-01 --update-addons=NetworkPolicy=ENABLED

# gcloud container clusters update \
  cluster-01 --enable-network-policy --zone=europe-west3
```

Siehe hierzu auch: *https://cloud.google.com/kubernetes-engine/docs/how-to/network-policy*

5.1.16 Manuelles Cluster-Sizing/-Scaling

Soll der GKE-Cluster über seinen Default-Pool nachträglich skaliert werden, hängen die zur Verfügung stehenden Skalierungsarten und Mengen, wie bereits in Abschnitt 5.1.14 beschrieben, von der ursprünglich gewählten Art der Provisionierung ab.

Im Folgenden wird ein Cluster-Scaling auf Basis eines neuen Node-Pools in einem regional provisionierten Cluster kurz und kompakt beschrieben. Die Anzahl der Nodes bezieht sich auf jeweils 1 Zone. Geben wir bei der Erzeugung eines neuen Node-Pools keine weiteren Parameter an, wird per Default auf den neuen Nodes die gleiche Kubernetes-Version wie auf den

bestehenden installiert, es kommt Googles CoS zum Einsatz und es werden wie im Default Instanzen vom Typ `e2-medium` genutzt: 2 vCPU, 4 GB RAM, 100 GB Disk.

```
# gcloud container node-pools create node-pool-1 --num-nodes=1 \
  --cluster=cluster-01 --region=europe-west4
```

```
# gcloud compute instances list
```

```
NAME                              ZONE            MACHINE_TYPE  INTERNAL_IP    EXTERNAL_IP     STATUS
gke-cluster-01-default-pool-[…] europe-west4-c  e2-standard-4  10.164.0.21    34.32.210.215   RUNNING
gke-cluster-01-node-pool-1-[…]  europe-west4-c  e2-medium      10.164.15.236  34.141.218.148  RUNNING
gke-cluster-01-default-pool-[…] europe-west4-b  e2-standard-4  10.164.0.12    34.32.255.245   RUNNING
gke-cluster-01-node-pool-1-[…]  europe-west4-b  e2-medium      10.164.15.237  34.141.159.65   RUNNING
gke-cluster-01-default-pool-[…] europe-west4-a  e2-standard-4  10.164.0.18    34.32.215.31    RUNNING
gke-cluster-01-node-pool-1-[…]  europe-west4-a  e2-medium      10.164.15.222  34.91.212.240   RUNNING
```

Wollen Sie einen bestehenden Pool skalieren, ist zu beachten, dass die Skalierung nur auf Basis der gleichen GCP-Instanzgrößen erfolgen kann, die bereits für die bestehenden Nodes verwendet werden. Zudem muss, wie bereits mehrfach für regionale Cluster erwähnt, darauf geachtet werden, dass der Integer-Zahlenwert in **`--num-nodes=<Integer>`** die gesamte Anzahl der Nodes *pro Zone* darstellt:

```
# gcloud container clusters resize cluster-01 --num-nodes=2 \
  --region=europe-west4 --node-pool=default-pool
```

5.1.17 Einen Node-Pool oder Cluster löschen

Um einen Node-Pool zu löschen, gehen Sie so vor, wie hier am Beispiel des erzeugten Node-Pools `node-pool-1` gezeigt:

```
# gcloud container node-pools delete node-pool-1 \
  --cluster=cluster-01 --region=europe-west4
```

```
The following node pool will be deleted.
[node-pool-1] in cluster [cluster-01] in [europe-west4]
Do you want to continue (Y/n)?  y
Deleting node pool node-pool-1...⠹
```

Sollen bestehende Cluster gelöscht werden, können Sie wie folgt vorgehen:

```
# gcloud container clusters list
```

```
NAME       LOCATION     MASTER_VERSION MASTER_IP     MACHINE_TYPE  NODE_VERSION   NUM_NODES STATUS
cluster-01 europe-west4 1.27.3-gke.100 34.91.208.192 e2-standard-4 1.27.3-gke.100 7         RUNNING
```

```
# gcloud container clusters delete cluster-01 --zone=europe-west4
```

```
The following clusters will be deleted.
 - [cluster-01] in [europe-west3]
Do you want to continue (Y/n)?  Y
Deleting cluster cluster-01...
```

5.1.18 Auth-Entries, Kontexte fetchen/switchen, Projekt setzen

Möchten wir zwischen mehreren GKE-Clustern umschalten, um direkt per `kubectl` an verschiedenen Clustern ohne umständliche Änderungen an Kontext- und Cluster-Settings arbeiten zu können, lässt sich dies ebenfalls per `gcloud` und dessen Sub-Option `get-credentials` erledigen:

```
# gcloud container clusters get-credentials cluster-01 \
  --project cluster-01-999999 --zone europe-west4
# gcloud container clusters get-credentials cluster-02 \
  --project cluster-02-999999 --zone europe-west4
```

Liegen alle Authentifizierungsdaten vor, reicht anschließend nur noch ein `gcloud config set project`, um zwischen den Clustern zu switchen:

```
# gcloud config set project cluster-02-999999
```

5.1.19 X509-Zertifikatsfehler bei der Ausführung von kubectl

Kommt es trotz korrektem `gcloud init` und syntaktisch intakter *.kube/config*-Datei zu folgendem Fehler bei einer `kubectl`-Aktion:

```
error: couldn't read version from server: Get
https://<API-Server-IP>/api: x509: certificate signed by unknown authority
```

... dann liegen in der Regel noch alte (nicht mehr passende) gecachte Keys unter *~/.config/gcloud*, die leider – aus welchen Gründen auch immer – trotz eines korrekten `gcloud init` nicht gelöscht wurden. Abhilfe erfolgt über das bereits vorgestellte Kommando zum Switchen zwischen Kontexten bzw. zum Fetchen der Credentials für den gewünschten Cluster, z. B.:

```
# gcloud container clusters get-credentials cluster-01 \
  --project cluster-01-999999 --zone europe-west4
```

Falls das nicht hilft, brauchen Sie einen größeren Hammer: Löschen Sie zuvor die *~/.kube/config*. Siehe auch:

https://cloud.google.com/sdk/gcloud/reference/container/clusters/get-credentials

5.1.20 Die Google-Registry nutzen

Siehe dazu auch:

- *https://cloud.google.com/container-registry/docs/quickstart*
- *https://cloud.google.com/container-registry/docs*

Zur Nutzung von Googles Container-/Artifact-Registry muss sichergestellt sein, dass die API für das entsprechende GCP-Projekt aktiviert ist. Ein Link findet sich unter der ersten URL. Alternativ kann die Aktivierung per **gcloud** erfolgen, z. B.:

```
# gcloud services enable containerregistry.googleapis.com --project <project>
```

```
# gcloud services enable artifactregistry.googleapis.com --project <project>
```

Beachten Sie, dass seit 2023 offiziell nur noch die Artifact-Registry, der designierte Nachfolger der Container-Registry, unterstützt wird. Die Container-Registry wird ab dem 15. Mai 2024 endgültig abgeschaltet.

Siehe dazu auch:

- *https://cloud.google.com/artifact-registry/docs/transition/transition-from-gcr*
- *https://cloud.google.com/container-registry/docs/deprecations/container-registry-deprecation*

5.1.21 Zugriff auf GKE-Worker-Nodes

Wenn Sie etwas direkt auf GKE-Nodes debuggen müssen, stehen für den Zugriff mehrere Verfahren zur Verfügung.

Per GUI

In der grafischen Oberfläche wählen Sie im passenden Projekt die Clusteransicht oder alternativ direkt die Ansicht der Compute-Engine-Instanzen, dann klicken Sie auf VERBINDEN/CONNECT bzw. SSH. Es wird eine Mini-VM-Instanz generiert, die den Zugriff auf den Cluster per Cloud-Shell ermöglicht. Alle Kommandos in dem GKE-Worker-Node müssen per **sudo** ausgeführt werden. Alternativ können Sie auf Testsystemen nach erfolgreichem Login per **sudo su -** in den root-Account wechseln.

Direkter ssh vom Bastions-/Client-Host

Wollen Sie vom Bastionshost aus direkt auf einen der GKE-Nodes zugreifen, können Sie wie folgt vorgehen:

```
# gcloud compute config-ssh
```

```
Updating project ssh metadata..Updated [https://www.googleapis.com/compute/v1/projects/cluster-01-
999999].
Updating project ssh metadata...done.
You should now be able to use ssh/scp with your instances.
For example, try running:
  $ ssh gke-cluster-01-default-pool-f576be9e-5d9g.europe-west3-b.cluster-01-999999
```

```
# ssh gke-cluster-01-default-pool-86085e9f-hxbt.europe-west4-c.cluster-01-999999

The authenticity of host 'compute.5083670228833498396 (35.234.91.72)' can't be established.
ECDSA key fingerprint is SHA256:SCMD96yuoIOuzNOgB1TaEO7iV9QLWWoWvtezGzjKncM.
ECDSA key fingerprint is MD5:ca:7f:2f:13:1b:ae:64:fc:54:ce:ea:6a:08:2c:7a:15.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'compute.5083670228833498396' (ECDSA) to the list of known hosts.
…
Welcome to Kubernetes v1.27.3-gke.100!

root@gke-cluster-01-default-pool-86085e9f-hxbt:~# crictl pods

POD ID        CREATED         STATE  NAME                NAMESPACE     ATTEMPT    RUNTIME
72e7554fca095 11 minutes ago  Ready  gmp-operator-[…]    gmp-system    0          (default)
f45a944f8fd0e 28 minutes ago  Ready  metrics-server-[…]  kube-system   0          (default)
f8cc7a288b5f1 30 minutes ago  Ready  rule-evaluator-6[…] gmp-system    0          (default)
1d95be26f8791 30 minutes ago  Ready  konnectivity-[…]    kube-system   0          (default)
c7de5c1d3c5e7 30 minutes ago  Ready  calico-typha-d[…]   kube-system   0          (default)
[…]
```

Zugriff per Debug-Pod

Alternativ kann, wie in Abschnitt 6.7.2 beschrieben, auch per Debug-Pod auf die Nodes zugegriffen werden. Dieses Verfahren ist (neben dem GUI-Verfahren) das unkomplizierteste und kann auf beliebige Kubernetes-Cluster angewendet werden.

5.2 EKS

Die Diskrepanz zwischen der nativen Managed-Kubernetes-Lösung von AWS ist zwar nicht mehr so gigantisch wie noch vor wenigen Jahren, aber zwischen EKS- und GKE-Clustern liegen bezogen auf die Usability und Konfigurationsmöglichkeiten immer noch Welten.

EKS-Cluster können wie ihre Pendants von Google und Microsoft auf verschiedene Arten provisioniert werden: per UI oder CLI, wobei die **`eksctl`**-Variante die für Einsteiger am besten geeignete darstellt, da sie wie **`gcloud container`** relativ intuitiv gestrickt und vor allem reproduzierbar und scriptbar ist.

5.2.1 Region, Zonen und Verfügbarkeiten

Auch bei Amazon stehen die üblichen Verdächtigen zur Verfügung, hier nur mit anderen Bezeichnern, wie z. B. eu-central-1 (Region Frankfurt) oder us-west-1 (Northern California).

Siehe dazu auch:

https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.RegionsAndAvailabilityZones.html

Die SLAs/Verfügbarkeiten finden sich unter: *https://aws.amazon.com/de/eks/sla/*

Kapitel 7
Kubernetes-Cluster: Day 1 Operations – Core-Workloads

»Das Schöne am Holzhacken ist, dass man sofort einen Erfolg sieht.«
– Albert Einstein

In diesem Abschnitt geht es darum, die Kubernetes-Standard-/Core-Workloads (wie Pods, Deployments, Daemon- und StatefulSets, Services, Ingress etc.) kennenzulernen, zu verstehen und mit ihnen zu arbeiten.

Wie bereits in den einleitenden Kapiteln des Buches ausdrücklich angemerkt, werden die Standard-Objekte aufgrund des fortgeschrittenen Scopes des Buches nicht mehr in einer epischen Tiefe behandelt. Tiefergehende Betrachtungen zu den jeweiligen Workloads kommen dort zum Einsatz, wo spezielle, für den Unternehmenseinsatz relevante Kriterien gefragt sind.

Beachten Sie bitte, das in den folgenden Abschnitt aus Effizienzgründen nur noch der in Abschnitt 6.5.1 vorgestellte `kubectl`-Alias `k` zum Einsatz kommt.

Hinweis

Es liegt auf der Hand, das es im Hinblick auf die zahlreichen Konzepte, Objekte, Ressourcen und Verfahren, welche in einem Kubernetes-basierten Cluster ineinandergreifen, niemals vollumfänglich möglich ist, alle Workloads in einer konsistenten, chronologischen Reihenfolge so vorzustellen, dass keine Vorgriffe auf andere Ressourcen oder Verfahren notwendig sind. Ich habe dennoch versucht, im Zuge der kompletten Neustrukturierung und Überarbeitung dieser Auflage die vorgestellten Themen so zu ordnen, dass sie bestmöglich aufeinander aufbauen.

7.1 Namespaces: Foundations

Siehe zu den folgenden Punkten auch:

- *https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/*

7.1.1 Vorbetrachtungen

Wir beginnen bei den Kubernetes-Foundations mit der *Namespace*-Ressource, da letztlich die meisten Ressourcen eines Kubernetes-Clusters bestimmten Namespaces zugeordnet sind bzw. in ihnen betrieben werden. Die *Namespaces* eines Kubernetes-Clusters definieren, vereinfacht erklärt, *logische Partitionen* für Ressourcen. Sie unterteilen den Cluster in logische Bereiche, die zum einen für eine bessere Übersicht sorgen sollen: separate Namespaces, z. B. für Logging-, Monitoring- und Pipeline-Stacks/Ressourcen, lassen sich wesentlich einfacher überblicken, durchsuchen und managen.

Beachten Sie bitte: Auch wenn es in diesem Abschnitt zunächst nur um ein generelles Verständnis von Namespaces geht, wird bereits an dieser Stelle ausdrücklich darauf hingewiesen, dass Namespaces *keine* Isolation der Ressourcen bedeuten: Ohne zusätzliche Sicherungsmaßnahmen wie *NetworkPolicies* (siehe ab Abschnitt 7.2 und Abschnitt 9.11) stellen Namespaces *keine* netzwerktechnisch unüberwindbare Barriere zwischen Ressourcen aus unterschiedlichen Namespaces dar. Das heißt: Ohne NetworkPolicies kann Pod A aus Namespace A ohne Probleme mit Service B oder Pod B in Namespace B kommunizieren.

Abbildung 7.1 zeigt abstrahiert ein (funktional valides) Applikations-Konstrukt mit multiplen Namespaces, das in dieser Form keinesfalls zwingend exakt so konstruiert sein soll/muss.

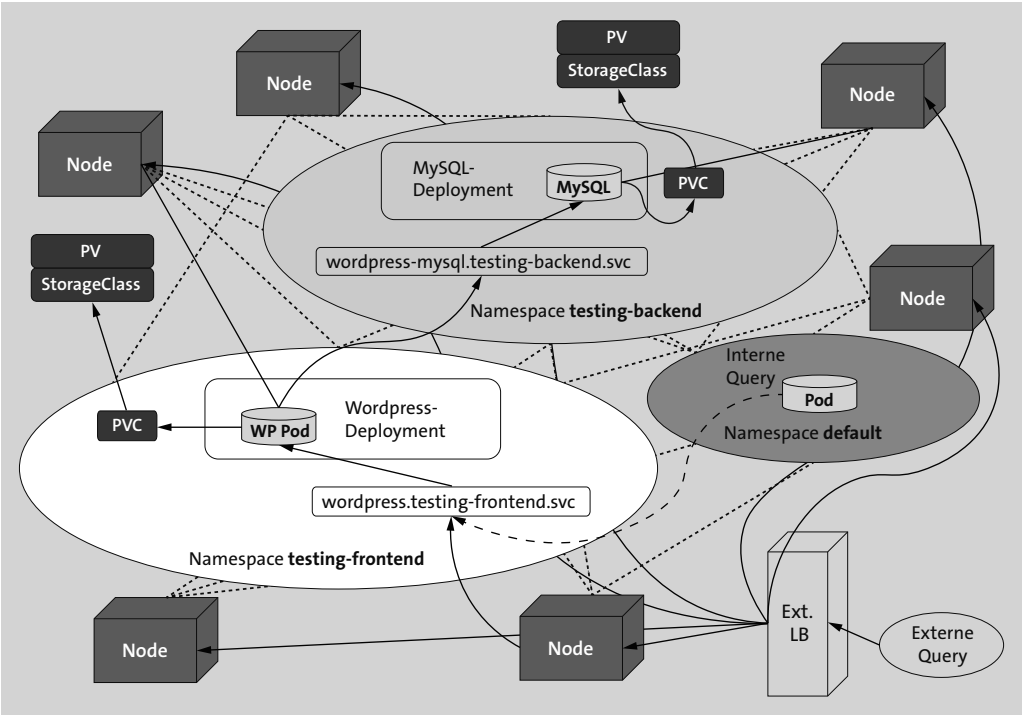


Abbildung 7.1 Multi-Namespace-WP-MySQL-Konstrukt ohne NetworkPolicies

Die Splittung der Ressourcen auf verschiedene Namespaces dient lediglich der Veranschaulichung in Bezug auf die Trennung von Applikations-Komponenten, um so via NetworkPolicies (siehe Abschnitt 9.11) expliziten, Namespace-übergreifenden Zugriff zu gewähren oder zu verbieten.

7.1.2 Funktionalitäten und Regeln bzw. Vorschriften für Namespaces

Nachfolgend ein paar kleine Auszüge von Funktionalitäten und Regeln bzw. Vorschriften für Namespaces:

- Ein Namespace wird über *alle* Nodes eines Clusters aufgespannt.
- Pro Namespace können z. B. individuelle Quotas und Ressourcen-Limitierungen gelten.
- *Abschottung*? Teil 1: Sofern keine zusätzlichen Restriktionen vorhanden sind, können Pods aus Namespace A (unter Angabe der Notation `<Service>.<Namespace>`) z. B. jederzeit mit einem *Service* (siehe Abschnitt 7.22) aus Namespace B kommunizieren und umgekehrt.
- *Abschottung*? Teil 2: Sofern keine zusätzlichen Restriktionen vorhanden sind, kann Pod A aus Namespace A jederzeit Pod B aus Namespace B direkt über seine IP erreichen und umgekehrt.
- Sind netzwerktechnische Abschottungen aus Sicherheitsgründen zwingend erforderlich, können `NetworkPolicies` (siehe Abschnitt 9.11) eine Möglichkeit sein oder – wie unter OpenShift – spezielle Netzwerk-Plugins (Stichwort: *ovs-multitenancy*).
- Pods aus verschiedenen Namespaces können auf den gleichen Worker-Nodes landen.
- Namespace-Namen sollten DNS-kompatibel sein (siehe Abschnitt 7.22, »Services«), da sie Teil der im Cluster bzw. Namespace verwendeten Service-Namen sind bzw. werden.
- Bevor Objekte in einem Namespace erzeugt werden können, muss dieser Namespace vorhanden sein.
- Ein Namespace muss durch entsprechende *Zugriffskontrollen* (*RBAC*, siehe ab Abschnitt 7.19) und/oder prozedurale Abläufe vor versehentlicher Löschung geschützt sein, da mit ihm zusammen auch alle Objekte automatisch gelöscht werden, die in diesem Namespace enthalten sind.
- Ressourcenverbrauch: Werden auf einem Cluster multiple Namespaces betrieben, muss sichergestellt sein, das alle Namespaces sorgfältig limitiert sind (Stichworte: *Limits*, *LimitRanges*, *RessourceQuota*, *ResourceQuotaScopeSelector* u. a., siehe ab Abschnitt 9.10), damit die Ressourcen eines oder mehrerer Namespaces durch einen unlimitierten Namespace und dessen Ressourcen nicht »an die Wand gedrückt« werden.

7.1.3 Standard-Namespaces in einem Vanilla-Kubernetes-Cluster

Standardmäßig existieren je nach Kubernetes-Version bzw. Anbieter in der Regel 4 Partitionen bzw. Namespaces:

```
# k get ns
NAME              STATUS   AGE
default           Active   4h
kube-node-lease   Active   4h
kube-public       Active   4h
kube-system       Active   4h
```

- **`default`** – Dieser Namespace dient üblicherweise als Startpunkt in einem Kubernetes-Cluster, um Workloads deployen zu können. Sobald man sich mit dem System vertraut gemacht hat, sollten für long-running Applikationsstacks eigene Namespaces angelegt werden. Wie die Aufteilung zu erfolgen hat, hängt von Tenancy-Aspekten, Security-Anforderungen und der gewünschten Übersichtlichkeit ab.
- **`kube-public`** – Objekte in diesem Namespace können von allen Benutzern des Clusters gelesen werden, auch von nicht authentifizierten. Wird insbesondere für per **`kubeadm init`** erzeugte Cluster benötigt. Diese legen beim Provisionieren des Masters dort z. B. eine ConfigMap ab, die öffentlich zugänglich ist: **`k describe -n kube-public cm cluster-info`**. Dieser Namespace vereinfacht zwar das *Sharing* von *öffentlichen* Cluster-Ressourcen, kann aus sicherheitstechnischer Sicht aber bedenklich sein.
- **`kube-system`** – der Maschinenraum in Vanilla-Kubernetes-Clustern und Derivaten. Er sorgt für eine Separierung der System-Pods vom Rest des Kubernetes-Clusters, und in Verbindung mit den *Admission-Controls* (siehe Abschnitt 7.19.8) sorgt er dafür, dass der Namespace der Kubernetes-Kerndienste zumindest rudimentär geschützt ist und nicht versehentlich gelöscht werden kann. Gleiches gilt für alle der hier benannten systemrelevanten Namespaces.
- **`kube-node-lease`** – Enthält *Lease*-Objekte für jeden Node des Clusters. Stellt einen Ersatz für das bis Kubernetes-Version 1.13 verwendete und aus der HA-Welt übernommene Heartbeat-Verfahren (»Lebt der Node noch?«) dar. Jeder Node muss seinen Lease intervallbasiert permanent erneuern, andernfalls gilt er als tot. Parallel dazu wird auch das Attribut `NodeStatus` vom Cluster ausgewertet, welches jedoch weniger häufig aktualisiert wird. So können die Node-Leases ausgelesen werden, hier für einen regionalen GKE-Cluster mit Ceph-Storage-Nodes:

```
# k get -n kube-node-lease leases.coordination.k8s.io
NAME                                          HOLDER                                        AGE
gke-cluster-01-ceph-pool-1-72d547d6-54kp      gke-cluster-01-ceph-pool-1-72d547d6-54kp      7h2m
gke-cluster-01-ceph-pool-1-c9a39bd1-54kv      gke-cluster-01-ceph-pool-1-c9a39bd1-54kv      7h2m
gke-cluster-01-ceph-pool-1-defd3618-nvqn      gke-cluster-01-ceph-pool-1-defd3618-nvqn      7h2m
gke-cluster-01-default-pool-633ed446-n9sv     gke-cluster-01-default-pool-633ed446-n9sv     2d8h
```

```
gke-cluster-01-default-pool-8d4a4e59-fk3s   gke-cluster-01-default-pool-8d4a4e59-fk3s   2d8h
gke-cluster-01-default-pool-ec1f714a-qh7r   gke-cluster-01-default-pool-ec1f714a-qh7r   2d8h
```

7.1.4 Uniqueness pro Namespace

Innerhalb eines Namespaces müssen die Namen der verwendeten Ressourcen bzw. Objekte logischerweise einzigartig (engl. *unique*) sein; über Namespace-Grenzen hinweg können die Ressourcen bzw. Objekte durchaus identisch bezeichnet sein: Ein Deployment *web-frontend1* kann im *testing*-Namespace existieren und ein gleichnamiges *web-frontend1* im Namespace *qa*, aber es darf keine zwei Deployments mit dem Namen *web-frontend1* in ein und demselben Namespace geben.

7.1.5 Objekte mit und ohne Namespace-Zuordnung

Auch in einem Kubernetes-Cluster müssen – und dürfen – nicht alle Objekte zwingend einem Namespace zugeordnet sein. Einfachstes Beispiel: *Nodes*. Sie sind Objekte, die für *alle* Namespaces im Cluster verfügbar sein müssen. Genauso wenig können *Namespace*-Objekte selber einem Namespace zugeordnet sein, sondern müssen »global« auf Cluster-Scope-Ebene existieren:

```
# k api-resources --namespaced=false
```

```
NAME                        SHORTNAMES   APIGROUP                       NAMESPACED   KIND
componentstatuses           cs                                          false        ComponentStatus
namespaces                  ns                                          false        Namespace
nodes                       no                                          false        Node
persistentvolumes           pv                                          false        PersistentVolume
[...]
```

Das Gegenstück ist logischerweise der Boolean-Schalter `--namespaced=true`.

7.1.6 Namespaces erzeugen

Damit Kubernetes-Ressourcen in bestimmten Namespaces platziert werden können, müssen diese zunächst angelegt werden. Es empfiehlt sich in Produktivumgebungen immer, diese Unterteilungen vorzunehmen und z. B. entsprechende Produkt- oder aufgabenspezifische Namespaces anzulegen, wie z. B. im einfachsten Fall für Monitoring- oder Logging-Stacks. Ein Namespace ist einfach zu erzeugen, entweder per `kubectl create ns <name>` oder, feiner granulierbar, über ein passendes Manifest:

```
# ns.yaml
apiVersion: v1
kind: Namespace
metadata:
  name: k8-test
```

```
  labels:
    name: testing
    tier: backend
```

```
# k create -f ns.yml
```

7.1.7 Namespace löschen

Achtung: Wird ein Namespace gelöscht, werden damit auch *alle* in ihm gehosteten Objekte und Ressourcen gelöscht. Standardmäßig sind seit Kubernetes-Version 1.11 alle Default-Namespaces (*default*, *kube-public*, *kube-system, kube-node-lease*) gegen versehentliche Löschung geschützt.

Um die versehentliche Löschung eigendefinierter Namespaces zu verhindern, können passende *Admission-Controls* (siehe Abschnitt 7.19.8) und entsprechende RBACs (siehe Abschnitt 7.19 und folgende) gesetzt werden. Normale Benutzer ohne spezielle administrative Berechtigungen oder ACLs können einen Namespace nicht löschen.

7.1.8 Wenn die Namespace-Löschung hängt

Die Löschung eines Namespaces kann gerne einmal hängen (Status bleibt ewig auf `Terminating`, auch ein **`--force --grace-period 0`** hilft in der Regel nicht), wenn eine bestimmte Komponente innerhalb des relevanten Namespaces klemmt, d. h. nicht korrekt arbeitet. Die Kontrolle erfolgt z. B. per **`kubectl api-resources`**. Taucht dort folgende Meldung auf, liegt das Problem in der ausgespuckten Komponente, hier für einen metrics-server:

```
error: unable to retrieve the complete list of server APIs: custom.metrics.k8s.io/
v1beta1: the server is currently unable to handle the request
```

Zur forcierten Löschung kann das in den Beispieldateien zu findende Script *ns-force-del.sh* aufgerufen werden, das hängende Finalizer auch wirklich finalisiert. Zuvor muss man den entsprechenden Namespace in das Script eintragen:

```
(
NAMESPACE=<your-rogue-namespace>
kubectl proxy &
kubectl get namespace $NAMESPACE -o json |jq '.spec = {"finalizers":[]}' >temp.json
curl -k -H "Content-Type: application/json" -X PUT --data-
binary @temp.json 127.0.0.1:8001/api/v1/namespaces/$NAMESPACE/finalize
)
```

Ein sehr guter Blog-Artikel des Red Hat OpenShift Teams zum Thema *The Hidden Dangers of (forcefully) Terminating Namespaces* findet sich hier:

https://cloud.redhat.com/blog/the-hidden-dangers-of-terminating-namespaces

7.2 Namespaces: Multi-Tenancy- und Security-Aspekte

Wie bereits eingangs des letzten Abschnitts festgestellt, stellen Kubernetes-Cluster, welche – aus logistischen/kostentechnischen/sonstigen Gründen – Namespaces für verschiedene Kunden (im Sinne von Tenants) im gleichen Cluster hosten sollen/müssen, sicherheitstechnisch eine ernst zu nehmende Herausforderung dar. Aufgrund der permanent und weiterhin zunehmenden Komplexität von Kubernetes-basierten Clustern müssen bei dieser Betriebsart alle Security-relevanten Maßnahmen ergriffen werden, um unerwünschte Übergriffe von Objekten im Namespace A des Kunden A auf Objekte im Namespace B des Kunden B zu verhindern. Neben den in Abschnitt 7.19 vorgestellten RBAC-Methoden, die die Autorisierung im Cluster regeln, müssen zwingend NetworkPolicies (Abschnitt 9.11), die zentrale Authentifizierung via IDM-System und viele andere Verfahren zum Einsatz kommen.

Die wichtigste Regel lautet eigentlich: **Verwenden Sie keine Multi-Tenant-Cluster. Gönnen Sie jedem Kunden/Mandanten seinen eigenen Cluster.**

Warum diese Variante immer die sicherheits-, aufwands- und ressourcentechnisch bessere Alternative ist, wird gleich ersichtlich. Falls die *1-Cluster-pro-Mandant*-Variante kostentechnisch nicht umsetzbar ist, versuchen Sie, die folgenden Regelwerke zu befolgen – welche sich selbstverständlich auch nur als lose Zusammenfassung von Auszügen einiger wichtiger Punkte verstehen und zudem immer abhängig von der jeweiligen Situation, der eingesetzten Kubernetes-Version, deren Features und etlicher anderer Faktoren sind. Sie werden merken, dass der Aufwand schnell immens groß werden kann. Betrachten wir die Probleme nun auszugsweise.

7.2.1 Namespaces/Networking – kundenspezifische und System-Namespaces

Verschiedene Kunden erhalten ihre eigenen Namespaces in dem gleichen Cluster. Dies setzt zwingend voraus, dass ein CNI-Plugin zum Einsatz kommt, welches NetworkPolicies unterstützt. Ebenso ist der Einsatz von restriktiven RBAC-Objekten zwingend erforderlich. Aber dies löst nicht alle Probleme, denn alle Tenants arbeiten mit dem gleichen CNI-Plugin: Nehmen wir an, dass Kunde A bestimmte Features eines CNI-Plugins nutzen will/muss. Die Kunden B und C wollen/dürfen dieses Plugin jedoch keinesfalls nutzen. Zudem bringen in einem Multi-Tenant-Cluster IP-Range-spezifische Allow/Deny-Regeln nichts, da diese nicht kundenspezifisch bestimmten Pods und deren IPs zugeordnet werden können.

Die Zugriffe aller Tenants auf den Namespace `kube-system` (den es nur einmal pro Cluster gibt) können problematisch werden. Arbeitet der Kunde mit Produkten, die »unbedingt« darin laufen müssen oder bestimmte Zugriffsberechtigungen benötigen, die bezogen auf die Sicherheit des Controlplanes bedenklich sind, sind wiederum exakte Privilegien-Modelle gefragt, die den schreibenden Zugriff auf die globalen Core-Komponenten regeln und limitieren.

Zwischen unterschiedlichen Kunden-Namespaces ist ein `Deny All`-Ingress Pflicht. Egress darf nur aus dem Cluster heraus erlaubt sein, nicht in andere Namespaces. Zwischen 2 oder mehr Namespaces, die dem gleichen Kunden zugeordnet sind, können explizite Grantings/Allows eingebracht werden. Zudem müssen alle Verfahren exakt und strikt durch entsprechende RBAC-Regelwerke ergänzt werden.

7.2.2 Geteilte Core-Komponenten

In Kubernetes-Clustern, die von verschiedenen Kunden genutzt werden, existieren trotzdem Bereiche (Control-Plane-Komponenten, Proxies u. a.), die sich alle Mandanten/Tenants/Kunden teilen müssen. Dies bringt ebenfalls massive Probleme mit sich, hier nur einige Beispiele:

- Selbst ein granulierter Zugriff kann problematisch werden, wenn Kunde A das (nur global einstellbare) AdmissionControl X zwingend benötigt, Kunde B jedoch diese globale Richtlinie auf gar keinen Fall aktiv haben will/darf.
- Und es geht weiter: Kunde A implementiert einen Operator (inklusive CRDs, die in älteren Kubernetes-Versionen bzw. je nach Build eines Operators auf typischerweise Cluster-Scope-Ebene residieren) und dessen Controller-Loop im zentralen API-Server und Controller-Manager. Kunde B will einen gleichnamigen Operator (und damit Controller-Loop) implementieren, jedoch in einer anderen Version oder mit anderen CRDs bzw. Funktionalitäten.
- Und es geht mit den Proxies weiter: Soll/muss es kube-proxy sein? Oder doch lieber HAProxy, den Kunde A präferiert, aber Kunde B nicht. Und welche Balancer Modes sollen per Default eingestellt sein?

Die Liste lässt sich beliebig fortsetzen.

7.2.3 Problematiken mit Scheduling, Eviction, Preemption

Betrachten Sie die zahlreichen Kräfte (siehe ab Abschnitt 9.2), die an einem Pod zerren können (beim Scheduling: hinsichtlich seiner Platzierung im Cluster; bei der Eviction/Preemption: In welcher Reihenfolge wird der Pod evicted?).

Nun portieren Sie dies in einen Multi-Tenancy-Cluster, in dem die Nodes (zumindest potenziell) von allen Kunden/Tenants gleichermaßen beansprucht werden können

Erfahrungsgemäß läuft bereits nach kurzer Zeit alles auf einen Wettlauf der Kunden A, B, C, D hinaus, die sich in QoS-Klassen, Prioritäten, PDBs und den anderen benannten Faktoren überbieten, um auf ihren Nodes bleiben zu dürfen bzw. dort die Präferenz zu haben.

7.2.4 Node-Fixing als Lösung?

Nein, denn die Probleme gehen tiefer:

In einer Multi-Tenancy-Variante teilen sich verschiedene Kunden die gleichen Cluster-Nodes und deren Compute-Ressourcen. Gleiches gilt für (Shared) Storage. Natürlich können und müssen in einem solchen Multi-Tenancy-Szenario LimitRanges/ResourceQuotas zum Einsatz kommen, um die Nutzung von Ressourcen pro Namespace zu limitieren.

Aber auch dann, wenn Tenant-spezifische Node-Selektoren zum Einsatz kommen, die z. B. alle Pods des Kunden A auf die Nodes 1, 3 und 5 verbannen, die des Kunden B auf die Nodes 2, 4 und 6, ergeben sich weitere Probleme – unter anderem:

- Eine valide, kundenspezifische und möglichst allzeit anwendbare Berücksichtigung von Failure-Domains und Verfügbarkeitszonen muss sichergestellt sein.
- Autoscaler-Komponenten müssen diese Konzepte ebenfalls aufgreifen/verwenden.
- Wie ist vorzugehen, wenn Nodes mit unterschiedlichen Kapazitäten dem Cluster hinzugefügt werden? Welcher Kunde darf was beanspruchen?
- Und was ist bei dem Ausfall von Nodes zu tun, deren Leistungs-Patterns primär nur einen einzelnen Kunden betreffen? Das Ganze natürlich weiterhin unter kundenspezifischer Berücksichtigung von Failure-Domains/Verfügbarkeitszonen.

7.2.5 Node Security

Und es geht weiter: Kunde A will das neueste EKS/AKS/GKE-Whatever-Node-Image und die neueste Kubernetes-Version. Kunde B fährt einen konservativen Kurs und will / braucht z. B. eine bestimmte/andere Node- und Kubernetes-Version.

7.2.6 Logging/Monitoring

Und wieder geht es um gesharte bzw. Core-Komponenten. Die Log-Agents der meisten Stacks arbeiten üblicherweise als DaemonSet auf den Nodes, mit einer spezifischen Konfiguration, die für das Scraping aller (Container-)Logs auf dem jeweiligen Node bzw. im Cluster gilt.

Und wie üblich hat jeder Kunde seine eigenen Vorstellungen davon, wie das Pre-Processing/Ingesting auszusehen hat, damit er möglichst wenig Aufbereitungsarbeit hat. Ähnliches gilt für die Metrikdaten im Prometheus-Kontext (Abschnitt 11.4).

7.2.7 Wechselwirkungen mit Cluster-Autoscalern

(Automatisches) Up- oder Down-Scaling des Clusters? Gern, aber natürlich. Allerdings ist dies ohne umfangreiche Vorkehrungen immer mit dem Umsortieren von Pods eines Kunden auf andere Nodes verbunden – und dieses Prozedere gegebenenfalls wiederum mit Downtimes.

7.2.8 Security-Lösungen

Auch ganzheitliche Security-Suiten wie NeuVector, StackRox oder Aquasec können in der Regel pro Cluster nur einmal installiert werden, da sie sich sonst erfahrungsgemäß funktional ins Gehege kommen. Informationen zu diesen Security Lösungen finden Sie in Kapitel 17.

7.2.9 Haftung

Kunde A betreibt (unbeabsichtigt) ein kompromittiertes Image mit Crypto-Miner und/oder Trojanern, das sich an der Security-Lösung vorbeimogelt und den Kernel des Nodes und damit alle Container auf diesem (und somit auch die des Kunden B) kompromittiert. Kunde F betreibt ein Sicherheits-/Privilegien-/Capabilities-technisch nicht ausreichend limitiertes Image. Ein Angreifer nutzt die Schwachstelle und kompromittiert den ganzen Cluster, und damit alle Tenants. Wer haftet wie?

... und, und, und.

7.2.10 Fazit

Unter dem Strich sind Multi-Tenancy-Cluster immer mit einem hohen Risiko und hohem Zeitaufwand bei der Absicherung verbunden. Und egal wie sorgfältig Sie Ihre Hausaufgaben machen, irgendetwas kann – und wird, wie die Erfahrungen zeigen – an irgendeinem Punkt trotzdem passieren, das ist sicher. Eine 1-Cluster-pro-Tenant-Variante bietet die höchste Sicherheit und kann entsprechend klein dimensioniert werden. Zudem ist sie mit IaC schnell auszurollen, zu zerstören und wiederherstellbar – bei vollständiger Mandanten-Trennung.

Wägen Sie daher den Einsatz einer 1-Cluster-with-multiple-Tenants-Architektur sehr sorgfältig ab. Die gewünschte Kosten-/Zeit-Einsparung kann sehr schnell nach hinten losgehen: sowohl vom Zeit- und damit Kostenaufwand als auch von der – gegenüber einer 1-Tenant/1-Cluster Lösung – in der Multi-Tenant-Variante bedenklicheren Security.

7.3 Pods und Container

Siehe zu allem Folgenden auch:

https://kubernetes.io/docs/concepts/workloads/pods/pod/

7.3.1 Foundations

Die wichtigste Core-Ressource in jedem Kubernetes-basierten Container-Cluster sind Pods. Pods beinhalten Container. Container nutzen – da sie keinen eigenen Kernel besitzen – einige Funktionen/Syscalls des Host-Kernels (Stichwort: *Kernel Namespace Sharing*). Was den Applikations-Containern im Pod erlaubt ist, wird durch die bereits vorgestellten Seccomp-

Profile sowie (optional) durch in der Pod-Spezifikation gesetzte SecurityContexts (und darin gesetzte Kernel-Capabilities, siehe Abschnitt 7.6 und `man 7 capabilities`) definiert.

Aus der Sicht eines Kubernetes-basierten Clusters stellt ein Pod immer die kleinste ausführbare Einheit dar, die aus 1 bis *n* Containern bestehen kann. Der Pod, im Sinne von »Gehäuse« oder »Behälter«, kann dabei im weitesten Sinne als eine logische Zusammenfassung für eine oder multiple Container-Instanzen angesehen werden. Technisch betrachtet ist der Pod selbst auch nur ein leerer Container (im Vanilla Kubernetes der sogenannte *pause*-Container), der Kernel-Namespaces (`NET`, `MNT`, `IPC` usw., siehe auch `man 7 namepaces`) auf dem Host reserviert und diese den eigentlichen Applikations-Containern, die er (der Pod) hostet, zur Verfügung stellt.

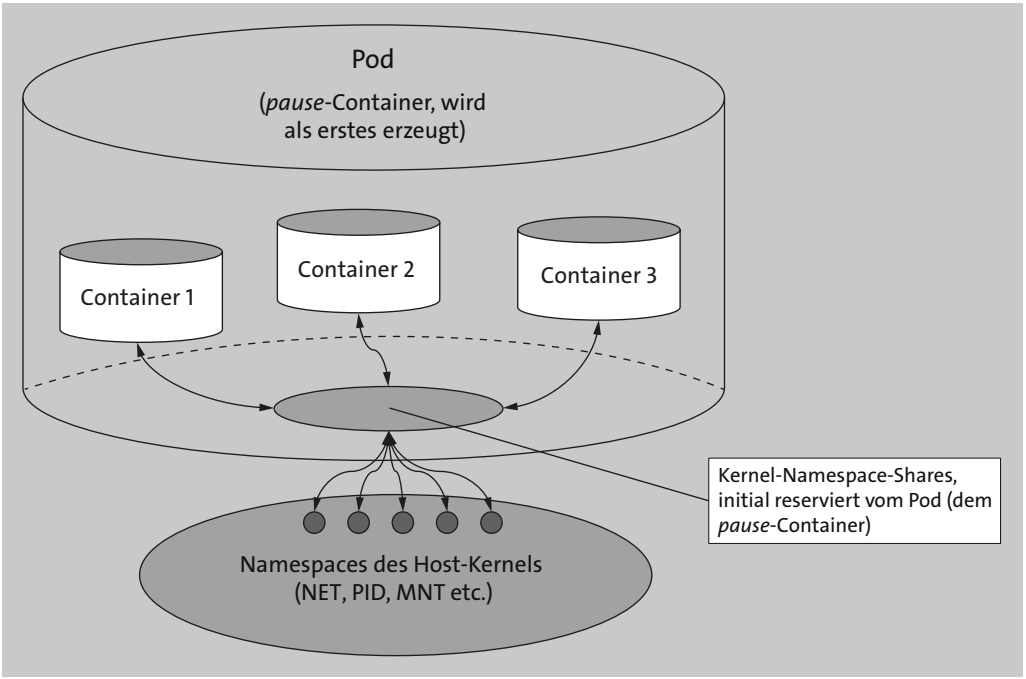


Abbildung 7.2 Kubernetes-Pod mit drei Containern

Nach dem Start einer Pod-Ressource wird Mikrosekunden vor dem/den eigentlichen Applikations-Container(n) der korrespondierende *pause*-Container gestartet, der den eigentlichen Pod (im Sinne des übergeordneten Behälters) bereitstellt. Der *pause*-Container ist immer der erste Container, den das Kubelet instanziiert, sobald ein Pod mit einem oder mehreren Applikations-Containern darin erzeugt wird. Der Pod erhält anschließend vom Overlay-Netz seine IP, richtet den Netzwerk-Namespace ein und reserviert ihn. Sein Kernprozess ist im Grunde nichts anderes als eine Art *pause*/*sleep*-Modus. Alle weiteren, »echten« Applikations-Container, die anschließend im Pod gestartet (oder nach einem Crash neu gestartet) werden, verbinden sich, sobald sie verfügbar sind, mit dem Netzwerk- und IPC-Namespace des *pause*-

Infrastruktur-Containers bzw. joinen diese. Ian Lewis (Google) hat zu den *pause*-Containern einen netten Blog-Post verfasst:

https://www.ianlewis.org/en/almighty-pause-container

Durch die Nutzung der Kernel-Ressourcen des Hosts ist ein Pod (bzw. dessen Container) auch physikalisch immer an den Node gebunden, auf dem er ausgeführt wird. Der Vorteil des Pod-Konzepts im Vergleich zu Standalone-Containern liegt darin, dass logische Applikationsgruppen (in der Regel die containerisierte Hauptapplikation sowie Helper-Applikationen) in einem Pod zusammengefasst arbeiten können: Die Kommunikation der Container untereinander, d. h. innerhalb des Pods, muss nicht über das (Overlay-)Netzwerk erfolgen, sondern kann direkt via Loopback/localhost abgewickelt werden. Jeder Container innerhalb eines Pods kann bei Bedarf über IPC-Sockets mit einem anderen kommunizieren, und alle Container innerhalb eines Pods können auf die gleichen Mountpunkte (des Pods) zugreifen. Zudem erhält lediglich der Pod vom Overlay-Netzwerk-Plugin eine IP, über die er erreichbar ist, nicht die einzelnen Container, die er enthält.

Insofern kann man den Pod durchaus und extrem stark vereinfacht als eine Art »Mini-VM« für den oder die darin inkludierten Container betrachten:

- Nicht der/die Container im Pod erhalten IPs, sondern – wie eine VM – der Pod selbst.
- Jede Applikation (konkret: Container) im Pod ist – wie bei einer Mini-VM – über die (dem Pod zugewiesene) IP und die Applikations-Port erreichbar.
- Jede Applikation (jeder Container) innerhalb des Pods kann – wie auf einem Host oder einer VM – netzwerktechnisch direkt via Loopback (localhost) mit jeder anderen Applikation im Pod kommunizieren.
- Jede Applikation (jeder Container) kann – wie auf einem Host oder in einer VM – den/die vom Pod zur Verfügung gestellten Mountpunkt(e)/Pfad(e) nutzen.
- Die Container innerhalb eines Pods werden immer zusammen »umgezogen« bzw. auf einem anderen Node (oder dem gleichen) in einem neuen Pod (mit einer neuen IP) zusammen neu gestartet.

Des Weiteren können die Container innerhalb eines Pods über *cgroups* limitiert werden, z. B. CPU/Memory-Limits, und je nach Kubernetes-Version kann auch ein statisches CPU-Pinning von Container-Prozessen erfolgen.

Pods, als kleinste Entität eines Kubernetes-Clusters, werden jedoch *standalone* so gut wie nie ausgerollt. Ihr Rollout und Betrieb erfolgt üblicherweise mithilfe von umschließenden Metahüllen wie DaemonSets, StatefulSets oder Deployments (Letztere mit den darin enthaltenen ReplicaSets), welche erweiterte Funktionalitäten wie HA, Skalierbarkeit, Steuerung von Rolling Updates etc. mitbringen. Diese Standard-Metahüllen und ihre Features werden in den nun folgenden Kapiteln noch kurz und kompakt erläutert.

7.3.2 Überblick: Pods, Startup-Orderings, Init-Container und die Nonsense-Altlasten von Docker, Inc.

Docker, Inc. und ihre zum Teil kruden Paradigmen wie »Eine App pro Container« und »Kein systemd-init im Container« haben Probleme wiederbelebt, die in immer komplexer werdenden Microservice-Ökosystemen weder zeitgemäß noch haltbar sind. Dies führte zu einem speziellen Designansatz in Vanilla-Kubernetes-Clustern, den *Init-Containern*, was die Komplexität wiederum erhöhte.

In der Linux-Host- und Cluster-Welt können Dependencies und Orderings einfach definiert werden, z. B. »Applikation A braucht zwingend Applikation B, daher muss B vor A gestartet sein«. Diese Regelwerke sind in systemd und Pacemaker-HA-Clustern seit Langem fest verankert. Docker, Inc. machte diese Konzepte in Container-Clustern zunächst wieder zunichte: Kleine, eng mit der Hauptapplikation verdrahtete Hilfsprozesse mussten umständlich separiert und in externe Units ausgelagert werden, die dann mehr oder weniger gut mit der Hauptapplikation zusammenarbeiteten. Damit entstanden Probleme, die eigentlich längst gelöst waren. Als Antwort im Kubernetes-Land kamen dann Init-Container auf, die die Koordination von 1 bis *n* containerisierten Helper-Apps in einem Pod ermöglichen sollten. Klingt in der Theorie gut, aber in der Praxis führt es zu einer Erhöhung der Komplexität und macht das Debugging schwieriger (siehe dazu auch die Abschnitt 7.4 und Abschnitt 7.5).

Natürlich haben Init-Container ihre Berechtigungen und sind in bestimmten Szenarien die richtige Wahl. Aber das Dogma »Always use Init-Containers« hat sich leider penetrant festgesetzt, obwohl systemd-init-Container – als Paradebeispiel Red Hats *ubi-init* – an manchen Stellen die bessere, weil effizientere, einfachere, simplere und robustere Wahl wären.

7.3.3 Einfache Pod-Manifeste

Hier ein erstes, sehr einfaches Pod-Manifest:

```
# simple_pod.yml
apiVersion: v1
kind: Pod
metadata:
  name: myhttpd
  labels:
    app: web
spec:
  containers:
  - name: httpd
    image: registry.access.redhat.com/ubi9/httpd-24:latest
    imagePullPolicy: IfNotPresent
    ports:
    - containerPort: 8080
```

Die wichtigsten Attribute in Kurzform: `metadata.name` definiert den Namen des Pods, optionale Label werden über `metadata.labels` gesetzt. Der Containername wir über `spec.containers.name` definiert, der Container basiert auf dem unter `spec.containers.image` angegebenen Attributwert und kann potenziell Traffic auf Port 80 (`spec.containers.ports.containerport`) entgegennehmen. Das - (ein Minuszeichen bzw. ein Bindestrich) im Rahmen der Definition von Subattributen – wie im oberen Beispiel (`spec.containers.(-)name`) – leitet in der Regel ein Array von mehreren Folge-Werten bzw. eine *Liste* ein. Das Attribut `imagePullPolicy` wird im übernächsten Abschnitt erläutert.

Nachfolgend ein weiteres, kleines Beispiel, das zwei Container in einem Pod bereitstellt. Aufgrund der umfangreichen Vorbetrachtungen sollten die meisten der im Folgenden verwendeten Attribute größtenteils selbsterklärend bzw. bereits bekannt sein:

```
# simple_pod-2.yml
apiVersion: v1
kind: Pod
metadata:
  name: webpod
  labels:
    app: webpod
spec:
  containers:
  - name: httpd
    image: registry.access.redhat.com/ubi9/httpd-24:latest
    imagePullPolicy: IfNotPresent
    ports:
      - containerPort: 80
  - name: client
    image: registry.access.redhat.com/ubi8
    imagePullPolicy: IfNotPresent
    command: ["/bin/bash","-c","while true; do curl http://
localhost:80; sleep 5; done"]
```

Entscheidend für die Nutzung der Pod-Funktionalität (hier: NET-Namespace-Sharing) ist der Teilstring `http://localhost:80` in der `command`-Zeile. Die Direktive `command:` in einer Pod-Definition überschreibt jede etwaig vorhandene `CMD` oder `ENTRYPOINT`-Direktive im Container bzw. im unterliegenden Image. Die vom Container innerhalb des Pods auszuführenden Kommandos können entweder kommasepariert, wie im oberen Beispiel, oder als Kombination aus `command` und `args` angegeben werden. Letzteres (`args`) setzt dann weitere Parameter für den auszuführenden Befehl:

```
command: ["/bin/echo"]
args: ["hello"," ","world"]
```

7.3.4 ImagePullPolicies für Container

https://kubernetes.io/docs/concepts/containers/images/

Über das Attribut `imagePullPolicy` wird das Verhalten der Kubelets beim Image-Pull bzw. der von ihnen gesteuerten Container-Engines festgelegt. Mögliche Einstellungen des Attributes `imagePullPolicy` sind:

- `IfNotPresent` – Das Image wird nur dann gepullt, wenn es nicht bereits lokal auf dem Node vorhanden ist.
- `Always` (default) – Jedes Mal, wenn das Kubelet einen Container startet, fragt das Kubelet die Registry ab, um den Namen in den assoziierten Image-Digest aufzulösen. Wenn der Node, auf dem das Kubelet läuft, bereits über ein Image verfügt, bei dem genau dieser Digest bereits lokal zwischengespeichert ist, verwendet das Kubelet sein lokal zwischengespeichertes Image. Andernfalls pullt das Kubelet über die Container-Engine das Image mit dem entsprechenden (neuen) Digest und verwendet dieses Image zum Starten des Containers.
- `Never` – Das Kubelet weist die Container-Engine erst gar nicht an, das Image zu pullen. Wenn das Image schon lokal vorhanden ist, versucht das Kubelet über die Container-Engine, den Container zu starten; andernfalls schlägt der Start fehl. Alle Images, die trotz `Never` gestartet werden sollen, müssen auf den Nodes bereits vorhanden sein, z. B. über Vorab-Pulls während der Provisionierung des Nodes.

Wenn Sie (oder einer der Controller) die Anforderung zur Erzeugung eines neuen Pods an den API-Server senden, legt Kubernetes den Wert für das `imagePullPolicy`-Attribut automatisch fest, falls bestimmte Bedingungen erfüllt sind:

- Ist `imagePullPolicy` nicht gesetzt und das Image-Tag `latest`, wird die `imagePullPolicy` automatisch auf `Always` gesetzt.
- Ist `imagePullPolicy` nicht gesetzt, und das Tag für das Container-Image ebenfalls nicht, wird das Tag `latest` angenommen und die `imagePullPolicy` wieder automatisch auf `Always` gesetzt.
- Ist `imagePullPolicy` nicht gesetzt und ein Tag für das Image angegeben, das nicht `latest` ist, wird die `ImagePullPolicy` automatisch auf `IfNotPresent` gesetzt.

Achtung

Der Wert von `imagePullPolicy` wird nur bei der initialen Erzeugung einer Ressource festgelegt, er wird nicht entsprechend der o. a. Regeln automatisch aktualisiert, wenn sich das Tag des Images später ändert.

ImagePullPolicy: Always, NAT-IPs und Cloud-Installationen

Ein weiterer Punkt: Kommt es durch Fehler im Image und der verwendeten `ImagePullPolicy: Always` zu einem permanenten Re-Pull der Images, kann dies, je nach Cloud-Provider und Netzwerk-Settings, zu einem erhöhten Verbrauch der verfügbaren NAT-IPs führen, was wiederum dazu führen kann, dass der Cluster in NAT-IP-Quota-Limitierungen läuft, welche dann gegebenenfalls erst aufwendig wieder gefixt bzw. bereinigt werden müssen. Zudem kann dies eine Kaskade auslösen: Die geblockten bzw. nicht verfügbaren NAT-IPs verhindern gegebenenfalls ihrerseits, dass sich externe Applikationen mit dem Kubernetes-Cluster konnektieren können bzw. vice versa und, und, und.

ImagePullPolicy-Override von höchster Stelle

Über die *Admission-Controls* (siehe Abschnitt 7.19.8) des *kube-apiserver* kann Cluster-weit das Flag `AlwaysPullImages` gesetzt werden. In diesem Fall stellt der Admission-Controller jede neue Pod-basierte Ressource auf die `ImagePullPolicy: Always` um. Darüber kann z. B. erreicht werden, dass Images in einem Multi-Tenant-Cluster explizit *vor* jedem Container-Start neu gepullt werden. So ist gewährleistet, dass – selbst dann, wenn das Image auf dem Node bereits vorhanden wäre – das Image immer nur von dem Eigentümer gepullt/verwendet werden kann, der auch die entsprechenden Pull-Credentials besitzt.

Achtung

Das Flag bzw. sein Wert tritt selbst dann in Kraft, wenn in einem Manifest oder einer **`kubectl run`**-Anweisung eine andere Policy (wie z. B. `Never`) explizit gesetzt wurde. Bereits bestehende Ressourcen werden nicht beeinflusst.

7.3.5 Serielle und parallele Image-Pulls

Standardmäßig ruft das Kubelet auf einem Node Images seriell bzw. sequenziell ab. Das bedeutet: Jeder Pull-Request in der Queue muss so lange warten, bis der aktuell laufende Pull abgeschlossen ist. Wenig effizient. Um eine Parallelisierung zu aktivieren, muss in den Kubelet-Settings der Parameter `serializeImagePulls` auf `false` gesetzt werden.

Diese Einstellung gilt allerdings nur für Images unterschiedlicher Pods: Das Kubelet ruft (über die Container-Engine) keine Images parallel ab, die für ein und denselben Pod gedacht sind. Geht es z. B. um einen Pod, der einen Init-Container und einen Anwendungscontainer besitzt, werden die Image-Pulls für die beiden Container nicht parallelisiert. Ein ergänzender (Kubelet-)Parameter, der seit Kubernetes 1.27 als Alpha-Feature verfügbar ist, ist der Schalter `maxParallelImagePulls`, über den die maximale Anzahl paralleler Pulls definiert werden kann, um zu verhindern, dass einzelne Nodes den kompletten Netz-Traffic okkupieren.

Siehe dazu auch: *https://kubernetes.io/docs/concepts/containers/images/#serial-and-parallel-image-pulls*

7.3.6 Pod/Container-Phasen

Jeder Pod und Container durchläuft, egal ob standalone oder als verwalteter Teil einer übergeordneten Metahülle (Deployment, DaemonSet etc.), in seinem Lifecycle etliche Phasen bzw. Zustände. Natürlich gilt wie immer: Je nach Kubernetes-Version sind gegebenenfalls nicht alle der hier gelisteten Phasen/Zustände verfügbar bzw. besitzen andere, hier nicht erfasste Werte:

- `Running` – Der Pod ist an einen Worker-Node gebunden, und alle Container in ihm laufen im Moment der Abfrage. Dies muss jedoch noch kein Indiz dafür sein, dass es dem Pod/Container gut geht bzw. dass er nicht kurz zuvor aufgrund einer fehlgeschlagenen Liveness/Readiness-Probe (siehe ab Abschnitt 9.1) wiederholt neu gestartet werden musste.
- `Pending` – Der Pod wurde vom System akzeptiert, kann aber nicht auf einen Node scheduled werden. Gängige Gründe sind typischerweise, dass der Cluster keine verfügbaren Ressourcen mehr hat, ein benötigtes Volumen nicht verfügbar ist oder kein Node mit einem passenden Label gefunden wird.
- `ContainerCreating` – genau das, was es beschreibt: Dies ist der Status, während der Container nach dem (erfolgreichen) ImagePull erzeugt wird.
- `Succeeded`: Alle Container im Pod sind erfolgreich beendet und werden nicht neu gestartet.
- `Terminating` – Der Pod wird beendet.
- `Suceeded` – Die Terminierung war erfolgreich.
- `Failed` – Alle Container im Pod wurden terminiert, davon jedoch mindestens einer mit einem Fehler-Status.
- `Unknown` – Der Zustand des Pods kann nicht ermittelt werden. Die Ursache liegt in der Regel in der Kommunikation des Controlplanes mit dem Kubelet auf dem Node.
- `Waiting` – Wenn sich ein Container weder im Status `Running` noch im Status `Terminated` befindet, befindet er sich typischerweise im Status `Waiting`. Er führt in dieser Phase Vorgänge aus, die zum Abschließen des Startvorgangs benötigt werden, z. B. Warten auf den ImagePull oder die Einbindung eines Secrets.
- `OOMKilled` – Dieser Zustand zeigt an, dass ein Container im Pod aufgrund von Speichermangel (*Out of Memory*, OOM) vom System gekillt wurde.
- `CrashLoopBackOff` – Dieser Status ist normalerweise ein Indikator dafür, dass die Anwendung innerhalb des Containers nicht ordnungsgemäß gestartet werden kann bzw. dass die Liveness/Readiness-Probe fehlgeschlagen ist und der/die Container im Pod als Folge davon wiederholt neu gestartet werden muss/müssen. Durch den Zustand `CrashLoopBackOff` tritt Kubernetes wortwörtlich »einen Schritt zurück« und startet den Container nach Ablauf eines Intervalls neu.
- `ErrImagePull` – Unschwer falsch zu deuten: Das in der **`run`**-Anweisung oder im YAML-Manifest angegebene Image konnte nicht heruntergeladen werden. Gründe sind in der Regel: Syntaxfehler bei der Angabe des Images, das Image ist in der gewünschten Registry nicht vorhanden, oder es fehlen die Berechtigungen, um das Image zu pullen.

- `ImagePullBackOff` – Der Zustand vor dem »finalen« `ErrImagePull`. Im Prinzip wie `CrashLoopBackOff`. Es werden *n* Versuche gestartet, das Image zu pullen. Der Versuch schlägt fehl, der Pull-Prozess tritt einen Schritt zurück, bevor er es nach einer Weile wieder versucht.

Hinweis

Ein Pod, der mit einer Liveness-Probe ausgestattet ist, wird zudem die Zustände `Healthy`/`Unhealthy` berichten. Pods mit Readiness-Probe berichten die Zustände `True`, `False` oder `Unknown`.

7.3.7 Pod-RestartPolicies und Startverzögerung

Per Default setzt Kubernetes beim Erzeugen von Pods, z. B. im Rahmen eines Deployments, die RestartPolicy auf `Always`, sofern wir nichts anderes angeben. Mögliche Werte für `restartPolicy` sind `Always` (Default), `OnFailure` oder `Never`.

Die gewählte RestartPolicy gilt dabei immer für *alle Container eines Pods*. Die RestartPolicy bezieht sich nur auf durch das Kubelet befohlene Neustarts der Container auf demselben Knoten. Fehlgeschlagene und vom Kubelet durchgestartete Container werden mit einer exponentiell wachsenden Verzögerung neu gestartet. Die (Re-)Startverzögerung wird mit steigender Anzahl verlangsamt: Erster Versuch nach einem Back-Off-Delay von 10 Sekunden, zweiter Versuch nach 20 Sekunden usw. Sie endet nach Ablauf von 5 Minuten. Kann der Container in dieser Zeitspanne neu gestartet werden, wird der Restart-Counter nach 10 Minuten erfolgreicher Ausführung zurückgesetzt.

Hinweis

Die Default-RestartPolicy der Container eines Pods kann ab Kubernetes 1.27 erstmals selektiv durch das Attribut `spec.containers.resizePolicy.restartPolicy` überschrieben werden, das beim Online-Resizing von Compute-Ressourcen (CPU/Memory-Requests und Limits, siehe Abschnitt 9.5.1 und 9.5.11) zum Tragen kommt.

7.3.8 Auszüge einiger Beispiele für mögliche Zustände von Pods

Diese Auszüge gelten selbstverständlich immer in Abhängigkeit der eingesetzten Kubernetes-Version und deren Fehlerbehandlungsalgorithmen.

Ein Pod mit einem Container läuft und wird mit »Success« beendet

- Ein Log-Completion-Event wird geschrieben.

- Wenn *RestartPolicy*:
  - `Always`: Starte Container neu, Pod läuft weiter.
  - `OnFailure`: Pod geht in den Zustand `Succeeded`.
  - `Never`: Pod geht in den Zustand `Succeeded`.

Ein Pod mit einem Container läuft, der Container wird mit »Failure« beendet

- Ein Log-Failure-Event wird geschrieben.
- Wenn *RestartPolicy*:
  - `Always`: Starte Container neu, Pod läuft weiter.
  - `OnFailure`: Starte Container neu, Pod läuft weiter.
  - `Never`: Pod geht in den Zustand `Failed`.

Ein Pod läuft mit zwei Containern, Container #1 steigt mit »Failure« aus

- Ein Log-Failure-Event wird geschrieben.
- Wenn *RestartPolicy*:
  - `Always`: Starte Container neu, Pod läuft weiter.
  - `OnFailure`: Starte Container neu, Pod läuft weiter.
  - `Never`: Pod läuft weiter.

Wenn nun auch noch Container #2 in diesem Pod mit »Failure« aussteigt

- Ein Log-Failure-Event wird geschrieben.
- Wenn *RestartPolicy*:
  - `Always`: Starte Container neu, Pod läuft weiter.
  - `OnFailure`: Starte Container neu, Pod läuft weiter.
  - `Never`: Pod geht in den Zustand `Failed`.

Ein Pod mit einem Container ist im Zustand »Running«, der Container darin läuft »out of memory«

- Der Container geht in einen Fehlerzustand.
- Ein OOM-Log-Event wird geschrieben.
- Wenn *RestartPolicy*:
  - `Always`: Starte Container neu, Pod läuft weiter.
  - `OnFailure`: Starte Container neu, Pod läuft weiter.
  - `Never`: Log-Failure-Event, der Pod geht in den Zustand `Failed`.

Ein Pod ist im Zustand »Running«. Ein Volume, auf das er zugreift, ist nicht mehr verfügbar

- Alle Container werden gekillt.
- Es wird versucht, ein passendes Log-Event zu schreiben.
- Der Pod geht in den Zustand `Failed`.
- Läuft der Pod in einem ReplicaSet oder Deployment, wird der Pod auf einem anderen Node neu erzeugt.

Ein Pod läuft, der Worker-Node geht offline

- Der Node-Controller wartet den Timeout ab.
- Der Node-Controller markiert den/die Pod(s) auf dem Node als `Failed`.
- Die Pods werden auf einem intakten Worker-Node neu gestartet.

7.4 Pod-Sidecar-Patterns und das Applikations-Design

Offiziell lautet das Statement zu Sidecar-/Helper-Containern innerhalb eines Pods in etwa wie folgt: Sidecars können die Funktionalität eines in dem Pod gehosteten Applikations-Containers erweitern/verbessern, *ohne den Applikations-Container modifizieren zu müssen*. Das zielt natürlich in Richtung des bereits erläuterten Konzepts der System-Init-Container, Stichwort: UBI-init.

Was bei dem generischen Sidecar-Ansatz gerne vergessen wird: Oft ist er eben nicht generisch und damit generell wiederverwertbar, sondern muss aufwendig an den jeweiligen Anwendungsfall angepasst werden. Und spätestens dann stellen sich weitere Fragen, die normale Sidecar-Container nicht bzw. nur eingeschränkt lösen können: echte Dependency-Modelle, Hochverfügbarkeit auf Prozessebene (und nicht nur ein simpler Sidecar-Container-Restart), kurze Wege zwischen den Helpern und der Primärapplikation und etliches andere mehr.

Aber natürlich gilt auch bei dieser Konzept A vs. Konzept B-Debatte wieder – es gibt nicht *das* eindeutige Richtig oder Falsch, sondern wie üblich: »it depends«.

Sidecar-Container sind mittlerweile ein normales und weitverbreitetes Konzept in Kubernetes-basierten Clustern. Davon macht z. B. Istio (Service-Meshes, siehe ab Abschnitt 12.1) mit seinen zusätzlich in den Pods untergebrachten Envoy-Reverse-Proxies fleißig Gebrauch. Diese Proxies werden dem eigentlichen Applikations-Container im Pod vorgeschaltet und prozessieren als Primärelement den Traffic, während der eigentliche Applikations-Container nur noch per *localhost* mit dem Proxy sprechen muss und daher so generisch wie möglich gehalten werden kann. Ein anderes Beispiel für einen wesentlich simpler gestrickten Sidecar-Container wäre der gute alte `kube-dns`, der aus Performancegründen ab Kubernetes 1.12 durch CoreDNS ersetzt wurde. (Dies interessiert Google aber nicht: In GKE-Nodes bis 1.27.x ein-

schließlich wird dort `kube-dns` verwendet.) Im `kube-dns`-Pod verrichtet beispielsweise ein Sidecar-Container als externer Health-Checker seinen Dienst.

In Kubernetes können verschiedene Sidecar-Muster (Patterns) genutzt werden, um den Pods spezifische Funktionalitäten hinzuzufügen. Nachstehend folgt eine kurze Übersicht der gängigsten Patterns.

7.4.1 Klassischer Sidecar

Im klassischen Sidecar-Pattern hat der Sidecar-Container denselben Lifecycle wie der Applikations-Container. Der Sidecar führt Aufgaben aus, die speziell für den Hauptcontainer bestimmt sind, wie z. B. das Loggen von Daten oder das Überwachen von Metriken. Dabei ergänzt der Sidecar- den Hauptcontainer, indem er Aufgaben erfüllt, die nicht unmittelbar mit der Hauptfunktion des Containers zu tun haben.

7.4.2 Ambassador

Der eben bereits erwähnte (Mesh-)Anwendungsfall mit vorgeschalteten Reverse-Proxies wäre ein typisches Beispiel. Das Funktionsmodell eines Ambassador-Containers ist generell mit dem Ziel konzipiert, die Kommunikation zwischen externen/anderen Diensten/Applikationen für die eigentliche Kern-Applikation im Pod zu vereinfachen bzw. zu generalisieren, für die der Ambassador-Container auch als Service-Discovery-Layer fungiert. Die gesamte Konfiguration/Logik für die Kommunikation mit dem/n externen Dienst(en) befindet sich im Ambassador-Container. Die im Workflow dahinter liegende Kern-Applikation spricht lediglich via localhost mit ihrem vorgeschalteten Proxy. Der Ambassador-Container kümmert sich um die Verbindung zu den externen Diensten, hält die Verbindung offen, stellt sie im Fehlerfall wieder her und kann in der Regel seine Konfiguration dynamisch aktualisieren (Stichwort *ConfigMaps*, siehe Abschnitt 7.8).

7.4.3 Adapter

Das Adapter-Pattern wird häufig verwendet, um die Schnittstellen oder Daten eines Applikations-Containers zu standardisieren oder zu modifizieren. Dieses Pattern kann hilfreich sein, wenn verschiedene Applikationen unterschiedliche Formate für Protokolle, Metriken oder Logs verwenden. Das *Adapter*-Pattern kann in dem Fall dazu verwendet werden, die zu überwachenden Telemetrie-Daten des eigentlichen Applikations-Containers in das Format zu konvertieren, das den Metrik-Erfassungsstandards des gesamten Clusters entspricht.

7.4.4 Initializer

(aka Init-Container, siehe den nächsten Abschnitt)

Das Initializer-Pattern wird genutzt, um eine oder mehrere Aktionen sequenziell vor dem Start des Hauptcontainers durchzuführen. Diese Initialisierungsaufgaben könnten die Konfiguration von Einstellungen, das Einmounten von Zertifikaten, das Laden von Daten oder die Vorbereitung von Ressourcen im Applikations-Container mit Tools sein, die nicht im Hauptcontainer vorhanden sein sollen. Sobald alle Init-Container erfolgreich beendet wurden, wird der Hauptcontainer gestartet. Dieses Pattern wird im Folgenden noch etwas genauer betrachtet.

7.5 Pods und Init-Container

Siehe hierzu auch:

- *https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-initialization/*
- *https://kubernetes.io/docs/tasks/debug-application-cluster/debug-init-containers/*

Init-Container, die bereits seit Kubernetes 1.6 *stable* sind, können, wie im letzten Abschnitt bereits angerissen, als vorbereitende Helfer angesehen werden. Sie werden üblicherweise dazu verwendet, vorbereitende Tasks für den Start der Primärapplikation durchzuführen bzw. ihren Start zu übernehmen. Dabei können der oder die Init-Container beispielsweise Utilities oder Setup-Scripts enthalten, die vom Applikations-Image/-Container entkoppelt laufen sollen bzw. es Security-technisch müssen.

7.5.1 Funktionsweise

Init-Container funktionieren ähnlich wie ein (Batch-)Job und durchlaufen bestimmte vorbereitende Arbeitsschritte, bevor die eigentlichen Applikations-Container starten. Jeder Init-Container muss bis zu seiner erfolgreichen Beendigung (*Completion*) durchlaufen, bevor der nächste Init-Container bzw. die eigentlichen Applikations-Container starten können bzw. dürfen – die bereits benannte sequenzielle Abarbeitung. Schlägt die Ausführung eines Init-Containers fehl, startet das Kubelet (sofern die `restartPolicy` des Pods nicht `Never` ist) den Init-Container so lange durch, bis er sich erfolgreich beendet hat. Mögliche Werte für die `restartPolicy` des Init-Containers sind die üblichen: `Always`, `Never` und `OnFailure`. `Never` macht in der Regel bei Init-Containern meist keinen Sinn, da diese konzeptbedingt durchaus oft fehlschlagen können.

Der Init-Container verfügt als regulärer (Job-)Container bei Bedarf über alle Features eines regulären Containers in einem Pod, wie z. B. Ressource-Limits und Requests, Volume-Mappings und Security-relevante Einstellungen. Die Ressource-Requests und -Limits arbeiten jedoch etwas anders als in regulären Containern/Pods. Stark vereinfacht: Da der Init-Container nur zeitlich limitiert und *vor* dem Lifecycle des eigentlichen Applikations-Containers arbeitet, kann er (zusätzliche) Ressourcen reservieren. Sind mehrere Init-Container im Einsatz, gilt

die Angabe des Init-Containers mit dem höchsten Limit (siehe dazu auch den gleich folgenden Abschnitt 7.5.5).

7.5.2 Readiness

Ein wichtiger Unterschied zu regulären Containern innerhalb eines Pods besteht z. B. darin, dass in einem Init-Container keine *Readiness-Probes* unterstützt werden (siehe Abschnitt 9.1). Ein Pod, der Init-Container enthält, wird zudem erst dann den Status *Ready* erhalten, wenn alle Init-Container erfolgreich beendet wurden und seine eigene Readiness-Probe (sofern vorhanden) Betriebsbereitschaft an das Kubelet meldet.

7.5.3 Anwendungsmöglichkeiten für Init-Container

Init-Container können beispielsweise Utilities enthalten, die für die Ausführung vorbereitender Tasks Tools beinhalten, die aus Sicherheitsgründen nicht im Applikations-Container enthalten sein sollten. So können Tools wie `sed`, `awk`, `python` oder `dig` während der Vorbereitungen des Setups durch den Init-Container bereitgestellt oder über ihn heruntergeladen werden, ohne dass die Tools später im eigentlichen Applikations-Image bzw. -Container vorhanden sein müssen. Ebenso können die Init-Container beispielsweise Per-Flight-Conditions für die Applikations-Container prüfen (*»Volume XYZ gemountet und schreibbar?«*, *»MySQL-DB schon gestartet und erreichbar?«*) und die darauf zugreifenden Applikations-Container so lange blocken, bis alle notwendigen Vorbedingungen erfüllt sind. Alternativ können darüber beispielsweise auch Inhalte in Volume-Mounts der Pods kopiert werden.

7.5.4 Phasen des Init-Containers, mehrstufiges Init-Beispiel

Tabelle 7.1 listet mögliche Initialisierungsphasen und ihre Bedeutung auf.

Status	Bedeutung
`Init:N/M`	Der Pod hat insgesamt `M` Init-Container, davon sind `N` erfolgreich beendet worden.
`Init:Error`	Der Init-Container ist fehlgeschlagen.
`Init:CrashLoopBackOff`	Der Init-Container ist mehrmals fehlgeschlagen.
`Pending`	Der Pod hat noch nicht damit begonnen, den Init-Container auszuführen.
`PodInitializing` oder `Running`	Der Pod hat die Init-Container-Phase erfolgreich abgeschlossen.

Tabelle 7.1 Init-Container-Phasen

Betrachten wir ein einfaches, aber mehrstufiges Init-Beispiel. In dem folgenden Manifest sind zwei Init-Container und der eigentliche Applikations-Container verdrahtet. Im Prinzip beinhalten die beiden Init-Container nur Warteschleifen, die per **`nslookup`** so lange prüfen, bis die beiden Service-Namen (`mydb`, `myservice`) über den internen DNS registriert und erreichbar sind. Die im Manifest verwendete *Service*-Ressource wird ab Abschnitt 7.22 behandelt. Sind beide Init-Container erfolgreich durchgelaufen, startet die eigentliche Applikation (`myapp-container`), die ein Echo ausspuckt und danach 6 Minuten am Leben bleibt:

```
# init-container-2.yaml
[…]
---
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  initContainers:
  - name: init-mydb
    image: registry.k8s.io/busybox:latest
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb;
              sleep 2; done;']
  - name: init-myservice
    image: registry.k8s.io/busybox:latest
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice;
              sleep 2; done;']
  containers:
  - name: myapp-container
    image: registry.k8s.io/busybox:latest
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

Rollen Sie das Manifest aus und betrachten Sie auf einer anderen Konsole den Pod wie folgt:

```
# k get pods --watch

NAME        READY   STATUS    RESTARTS   AGE
myapp-pod   0/1     Pending   0          0s
myapp-pod   0/1     Pending   0          0s
myapp-pod   0/1     Init:0/2   0          0s
myapp-pod   0/1     Init:0/2   0          1s
myapp-pod   0/1     Init:1/2   0          2s
myapp-pod   0/1     PodInitializing   0          3s
myapp-pod   1/1     Running           0          4s
```

7.5.5 Init-Container und Compute-Ressources

Siehe zu den Compute-Ressources auch Abschnitt 9.5.

https://kubernetes.io/docs/concepts/workloads/pods/init-containers/#resources

Aufgrund der Ausführungsreihenfolge für Init-Container gelten die folgenden Regeln für die Ressourcennutzung:

- Die höchsten gesetzten Werte für Requests und Limits, die für einen bestimmten Init-Container (von gegebenenfalls mehreren vorhandenen) definiert sind, stellen die effektiven Werte für Request und Limits dar. Wenn für einen Container gar kein Ressourcenlimit angegeben ist, wird dies als das höchste Limit betrachtet (`no limit`).
- Der effektive Request bzw. das effektive Limit eines Applikations-Pods ist der jeweils höhere der folgenden Werte:
  - die Summe aller Requests bzw. Limits aller Applikations-Container des Pods
  - der effektive Request / das effektive Limit des Init-Containers
- Die Planung erfolgt auf der Grundlage effektiver Requests/Limits, was bedeutet, dass Init-Container Compute-Ressourcen für die Initialisierung reservieren können, die während der Lebensdauer des Pods jedoch nicht verwendet werden.
- Die QoS-Klasse des Pods gilt für Init-Container und App-Container gleichermaßen.
- Kontingente und Limits werden basierend auf dem effektiven Pod-Request und dem Limit angewendet.

Siehe dazu auch das Manifest `init-resource.yaml` in den Beispieldaten zu diesem Abschnitt. Die Kontrolle der Ressourcenauslastung durch einzelne Container eines Pods kann so abgefragt werden:

```
# kubectl (oder oc adm) top pods --containers=true
```

7.6 Pod- und Container-Security

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

Ein wichtiges Hilfsmittel, um Pods und Container im Betrieb sicherer zu machen, sind die sogenannten `securityContext`-Settings. Beachten Sie, dass diese zum einen mit spezifischen Settings auf der *Container*-Ebene gesetzt werden können, zum anderen aber auch auf *Pod*-Ebene – dort gegebenenfalls mit zum Teil anderen Settings. Die jeweils verfügbaren Attribute können Sie sich wie bereits vorgestellt ganz einfach per **`kubectl explain`** anzeigen lassen, im Folgenden exemplarisch für ein *Deployment* (siehe ab Abschnitt 7.14):

- Pod-Ebene: **`k explain deployment.spec.template.spec.securityContext`**
- Container-Ebene: **`k explain deployment.spec.template.spec.containers.securityContext`**

7.6.1 SecurityContext für Container: Kernel-Capabilities und mehr

Kernel-Capabilities für Container auslesen und setzen

Siehe dazu auch:

https://kubernetes.io/docs/tasks/configure-pod-container/security-context/#set-capabilities-for-a-container

Ein wichtiger sicherheitstechnischer Aspekt der Gesamt-Security von Container-Clustern sind die *Syscalls*, die ein Container-Prozess im Kernel des Hosts bzw. Nodes nutzen darf. Die zulässigen Syscalls werden üblicherweise über eine Default-Policy (die der Container-Engine zugewiesen wird) mittels sogenannter *seccomp*-Profile festgelegt, die bereits in Abschnitt 2.6.7 im Rahmen einführender Security-Betrachtungen vorgestellt wurden.

Eine weitere Granulierung der *Kernel-Capabilities* (Gruppierungen von Syscalls, siehe **`man 7 capabilities`**), die ein Container in einem Pod nutzen darf, kann über das Attribut `securityContext` eingestellt werden. Dazu muss zunächst verstanden werden, welche Capability-Arten es bezogen auf einen Prozess geben kann. Üblicherweise unterscheiden sie sich zum Teil in der Art ihrer Anwendung bzw. ihres Aufrufs/ihrer Erzeugung, wie z. B. *Inherited Capabilities* (`CapInh`). Betrachten wir dazu die folgende Liste:

- **`Permitted Capabilities`** (`CapPrm`) – die *Obergrenze*, die festlegt, was dieser Prozess maximal an vererbten (`CapInh`) und effektiven (`CapEff`) Capabilities nutzen kann
- **`Inherited Capabilities`** (`CapInh`) – die Capabilities, die *vererbt* werden sollen, wenn ein privilegierter Prozess einen oder mehrere weitere Sub-Programme/-Prozesse per **`execve(2)`** aufruft/erzeugt. Zu beachten ist: Wenn das aufrufende Programm bzw. der aufrufende Prozess nicht unter einem privilegierten Benutzer ausgeführt wird, bleiben die Capabilities des aufrufenden Prozesses nicht per Inheritance/Vererbung erhalten. In diesem Fall müssen explizit `Ambient Capabilities` verwendet werden.
- **`Effective Capabilities`** (`CapEff`) – Das sind die Capabilities, die letztlich *effektiv* zum Einsatz kommen. Sind keine gesetzt, gelten in der Regel die dann üblicherweise vorhandenen `CapInh`.
- **`Bounding Capabilities`** (`CapBnd`) – eine weitere Form der Capability-Limitierung, die dann zum Einsatz kommt, wenn der ausführende Prozess höhere Rechte erlangen will.
- **`Ambient Capabilities`** (`CapAmb`) – Diese Capabilities werden bei einem **`execve(2)`** für die erzeugten Prozesse erhalten, wenn der aufrufende Prozess nicht privilegiert arbeitet. Details hierzu siehe auch:

  https://lwn.net/Articles/636533/

Um Audit-technisch *Kernel-Capabilities* eines (Container-)Prozesses auszulesen, stehen verschiedene Verfahren zur Verfügung: z. B. per **`podman top <containername> capeff capinh ...`** oder beispielsweise über das Auslesen der realen PIDs der Applikationsprozesse des Con-

tainers (z. B. **podman top <containername> hpid pid comm args**) mit anschließender Abfrage des Prozessstatus (**grep ^Cap /proc/<Host-PID-des-Container-Prozesses>/status**) und anschließender Decodierung der in Hex-Form angezeigten Capabilities per **capsh --decode=<HexString>**. Nachstehend ein kleines Beispiel für die Anzeige der Capabilities eines `kube-apiserver`-Container-Prozesses (On-Prem) mit der realen Host-PID 2158:

```
# grep ^Cap /proc/2158/status

CapInh: 00000000080005fb
CapPrm: 00000000080005fb
CapEff: 00000000080005fb
CapBnd: 00000000080005fb
CapAmb: 0000000000000000

# capsh --decode=00000000080005fb

0x00000000080005fb=cap_chown,cap_dac_override,cap_fowner,cap_fsetid,cap_kill,cap_setgid,cap_
setuid,cap_setpcap,cap_net_bind_service,cap_mknod
```

Sollen von den per Default (über die Container-Engine per Default-Seccomp-Profil oder Default-Capabilities) gesetzten Privilegien abweichende Settings vorgenommen werden, kann dies granular über die Container-Ebene erfolgen. Dazu folgt ein kleines Beispiel für das Addieren zusätzlicher Capabilities zu einem Container. Die neuen Capabilities werden denen hinzuaddiert, die diesem Container bereits per Seccomp-Profil vererbt wurden. Achtung: So wie im Beispiel gesetzt (bezogen insbesondere auf `SYS_ADMIN`), sollten die Capabilities nie bzw. nur nach sehr sorgfältiger Risiko-Evaluierung verwendet werden:

```
# cap-add.yaml
apiVersion: v1
kind: Pod
metadata:
  name: capadd
spec:
  containers:
  - name: ubi8-dangerous-caps
    image: registry.access.redhat.com/ubi8
    command: ["/bin/bash","-c","sleep 5000"]
    securityContext:
      capabilities:
        add: ["SYS_ADMIN", "NET_ADMIN", "SYS_TIME"]
```

Neben dem Addieren/Gewähren zusätzlicher Capabilities können diese natürlich auch eingeschränkt/entzogen werden. Der entsprechende `securityContext`-Eintrag im Manifest ist dann `drop` anstelle von `add`. Beides ist auch parallel möglich, die gesetzten Caps sollten sinnigerweise nur nicht miteinander kollidieren

```
securityContext:
  capabilities:
    drop: ["NET_BIND", "AUDIT_WRITE"]
    add: ["SYS_TIME"]
```

Privilege(Escalation) und runAs*

Über das `securityContext`-Setting `allowPrivilegeEscalation` [Default: `true`] kann eingestellt werden, ob ein Container nach seinem Start höhere Privilegien erhalten darf als sein Parent-Prozess (üblicherweise die Container-Engine).

Über das `securityContext`-Setting `privileged` [Default: `false`] kann es Container-Prozessen erlaubt werden, mit root-Rechten zu laufen. Das Attribut `runAsNonRoot` [Default: `false`] unterbindet den Start eines Containers, dessen Primärprozess mit root-Rechten laufen will.

Daneben existieren etliche weitere Settings/Attribute, diese finden sich per:

```
# k explain deployment.spec.template.spec.containers.securityContext
```

Die verfügbaren SecurityContext-Settings für Pods lassen sich wie folgt auslesen:

```
# k explain deployment.spec.template.spec.securityContext
```

7.6.2 Pod Security Admission Controls

Siehe dazu auch: *https://kubernetes.io/docs/concepts/security/pod-security-admission/*

Der Vorläufer der *Pod Security Admission Controls*, die *Pod Security Policies* (*PSP*), wurden mit Kubernetes 1.25 endgültig beerdigt. Der neue Weg funktioniert jedoch komplett anders und bedeutet für alle Dev(Sec)Ops- und MLOps-Teams einmal mehr einiges an Arbeit und Umbaumaßnahmen im Cluster, um von PSP-Objekten auf das neue Verfahren umzustellen.

Während die PSPs noch als separate Ressourcen in einem Namespace implementiert wurden, werden die *Pod Security Admission Controls* (*PSAC*), wie der letzte Teil des Namens vermuten lässt, über *Admission-Controls* umgesetzt, d. h. als »höchstrichterliche« Policy im API-Server, die auf Namespace-Ebene per Label angewendet wird. Dabei bieten die PSAC zunächst grundsätzlich 3 verschiedene Modi, die etwas an SELinux-Policies erinnern:

- **enforce**: Bei Policy-Verletzungen wird der Pod rejected.
- **audit**: Bei Policy-Verletzungen wird eine Audit-Annotation dem Event-Record hinzugefügt, die Aktion an sich ist aber erlaubt.
- **warn**: Bei einer Policy-Verletzung erhält der ausführende User eine Warnung, die Aktion an sich ist aber erlaubt.

Der Security-*Mode* allein ist jedoch noch nicht ausschlaggebend – er muss in den Labeln eines Namespaces mit einem Security-*Level* verheiratet werden. Für Letzteren standen im betrachteten Stand ebenfalls 3 Geschmacksrichtungen zur Verfügung:

8.8 Hands on: PostgreSQL-Operator (Level 5)

Im Folgenden wird anhand eines Level-5-Operators gezeigt, welchen Grad der Vollautomation und Resilienz ein hochfunktionaler, zeitgemäßer Operator in einen Cluster einbringt. Als Beispiel betrachten wir den *Crunchy PostgreSQL Operator*, kurz: PGO. Beachten Sie, dass sich dieser Abschnitt nicht auf PostgreSQL-Cluster an sich fokussiert, sondern nur darauf, wie der PGO diese verwaltet.

Siehe zu allem Folgenden auch:

- *https://github.com/CrunchyData/postgres-operator*
- *https://postgres-operator.readthedocs.io/en/latest/user/*
- *https://access.crunchydata.com/documentation/postgres-operator/latest*
- *https://access.crunchydata.com/documentation/postgres-operator/latest/architecture/high-availability*
- *https://operatorhub.io/operator/postgresql*
- *https://www.postgresql.org/*

8.8.1 Postgres

Alle, die zwei Tage im Job sind, sollten mit dem Begriff PostgreSQL grob etwas anfangen können. Für alle, auf die das nicht zutrifft: PostgreSQL, kurz: Postgres, ist ein freies, objektrelationales SQL-standardkonformes, clusterfähiges Datenbankmanagementsystem (ORDBMS), dessen Entwicklung bereits in den 1980er-Jahren begann. Seit Ende der Neunziger wird das Produkt von einer Open-Source-Community weiterentwickelt. PostgreSQL verfügt über eine umfangreiche Palette an 3rd-Party Tools, z. B. *PostGIS* zur Verwaltung von Geodaten. Mittlerweile existieren zahlreiche PostgreSQL-Derivate, wie z. B. das im Folgenden verwendete *Crunchy Data* oder *EnterpriseDB*, das ebenfalls über eine Level-5-Operator-Unterstützung verfügt, aber ausschließlich per SLA verfügbar ist.

8.8.2 Der Postgres-Operator

Der Postgres-Operator (PGO) wird von *Crunchy Data* bereitgestellt und bietet Level-5-Automation zur Verwaltung von PostgreSQL-Clustern. Der PGO beherrscht (Auto-)Skalierbarkeit, Hochverfügbarkeit, Notfallwiederherstellung, das Klonen von Postgres-Clustern, Rolling Updates und etliches andere mehr. Der PGO basiert auf einem UBI8-Image (*registry.developers.crunchydata.com/crunchydata/postgres-operator:ubi8-5.4.1-0*).

Die Features des PGO im Detail:

- **Bereitstellung von PostgreSQL-Clustern** – Erstellen, Skalieren, Löschen und vollständige Anpassbarkeit der Konfiguration

- **HA** – automatisiertes Failover, unterstützt durch eine verteilte, konsensbasierte Hochverfügbarkeitslösung. Der PGO verwendet (einstellbar gewichtete) Pod-Anti-Affinität, um die Ausfallsicherheit zu verbessern. Race-Conditions durch ausgefallene Primär-/Master-Instanzen werden automatisch behoben. Es besteht die Möglichkeit, Backups intervallbasiert zu schedulen und Aufbewahrungsrichtlinien für Backups festzulegen.
- **Notfallwiederherstellung** – Sicherungen und Wiederherstellungen nutzen das Open-Source-Dienstprogramm *pgBackRest* und unterstützen vollständige, inkrementelle und differenzielle Sicherungen sowie effiziente Delta-Wiederherstellungen.
- **Überwachung** – Der Zustand der PGO-verwalteten PostgreSQL-Cluster kann mithilfe der Open-Source-Bibliothek *pgMonitor* überwacht werden.
- **Klonen** – Sie können Klone aus vorhandenen Clustern oder Backups erstellen.
- **TLS** – Alle Verbindungen erfolgen über (konfigurierbares) TLS.
- **Verbindungspooling** – erweiterte Unterstützung für Verbindungspooling mit *pgBouncer*
- **Affinität** und **Toleranzen** können flexibel eingestellt werden.
- **PostgreSQL-Major-Release-Upgrades** können deklarativ durchgeführt werden.
- **Up- und Downsizing** mit minimaler Disruption
- Verwenden Sie Ihr eigenes **Container-Image-Repository**, einschließlich der Unterstützung von *imagePullSecrets* und privaten Repositories.

Abbildung 8.6 zeigt in vereinfachter Form das Zusammenspiel der Komponenten (siehe dazu auch *https://access.crunchydata.com/documentation/postgres-operator/latest/architecture*).

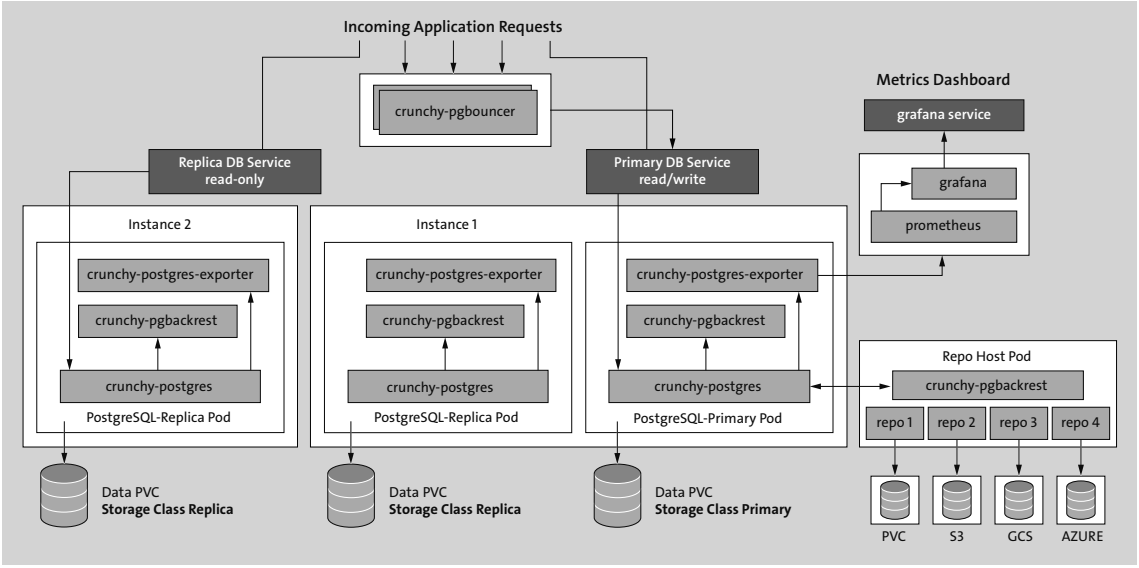


Abbildung 8.6 Der »Crunchy Postgres Operator« – High-Level-View (Quelle: crunchydata.com)

Dort finden sich auch detaillierte Infos über die komplette Architektur, die den Scope an dieser Stelle sprengen würden: HA, Backup-Management, Scheduling (PTSC, Affinities, PDBs, siehe dazu die Abschnitte ab Abschnitt 9.2 über [De-]Scheduling-Konzepte), User-Management, Monitoring und Disaster-Recovery, sowie alle verfügbaren Settings des PGOs.

8.8.3 Verfügbare Versionen

Beim Setup ist zu beachten, welche Version installiert werden soll: Crunchy Postgres stellt verschiedene Varianten bereit, von denen einige mit SLAs verknüpft sind:

- die frei verfügbare *Community-Edition* für alle Kubernetes-Derivate aus dem *operatorhub.io*
- die *Community Production-Edition* für OpenShift: *https://github.com/redhat-openshift-ecosystem/community-operators-prod/tree/main/operators/postgresql*
- die *Red Hat Certified Edition* für OpenShift: *https://github.com/redhat-openshift-ecosystem/certified-operators/tree/main/operators/crunchy-postgres-operator*
- die *Red Hat Marketplace Edition* für OpenShift: *https://github.com/redhat-openshift-ecosystem/redhat-marketplace-operators/tree/main/operators/crunchy-postgres-operator-rhmp*

8.8.4 Hochverfügbarkeit und Datenreplikation

Postgres-Cluster, die vom PGO verwaltet werden, können optional mit synchroner Replikation betrieben werden (Default: asynchron). Ein Vorteil des synchronen Modus liegt darin, dass alle Daten bei einem Failover auf allen Nodes konsistent sind. Allerdings geht dies zulasten der Performance: PostgreSQL muss warten, bis die Transaktion auf allen Replicas erfolgreich geschrieben wurde, bis sie als valide gilt. Verbundene Clients müssen daher, insbesondere bei vielen Schreibvorgängen, gegebenenfalls länger warten, als wenn die Transaktion nur auf dem Master/Primary geschrieben würde und dann bereits als valide gälte (asynchrone Replikation). Darüber hinaus hat der synchrone Replikationsmodus potenzielle Auswirkungen auf die Verfügbarkeit: Wenn in diesem Mode ein Node abstürzt, werden alle Schreibvorgänge auf dem Primärserver blockiert, bis eine (Ersatz-)Replica wieder zu einer neuen synchronen Replica des Masters/Primarys hochgestuft wurde. Die Parameter für den synchronen Mode (`spec.postgresql.parameters.synchronous_mode*`, default: `off`) können über die `PostgresCluster`-CR gesetzt werden.

8.8.5 Setup

Das Setup des PGO ist auf Basis des bereits installierten OLM denkbar einfach: Sie müssen dazu lediglich die passende *Subscription* importieren. Diese sorgt dafür, dass der PGO entsprechend des in der Subscription gesetzten Namespaces installiert wird, in diesem Fall:

`operators`. Die Subscription findet sich in den Beispieldaten, kann aber auch über den OperatorHub bezogen werden (*https://operatorhub.io/install/postgresql.yaml*). Alternativ kann das Postgres-Git-Repo geklont werden:

```
# git clone https://github.com/CrunchyData/postgres-operator.git
```

Wer die maximale Flexibilität wünscht, ist mit dieser Variante gut beraten. Unter *https://github.com/CrunchyData/postgres-operator-examples.git* liegen weitere Beispiel-Manifeste, u. a. Setups für das *pgmonitor*-Setup. Allerdings findet sich in keinem der Repos ein direkt nutzbares Subscription-Template, nur ein Installations-Script-Wrapper.

Im Folgenden wird die Subscription-basierte Variante via OLM verwendet – mit einer kleinen Ergänzung (`installPlanApproval: automatic`):

```
# postgresql-sub.yaml
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: postgresql-operator
  namespace: operators
spec:
  channel: v5
  installplanApproval: automatic
  name: postgresql
  source: operatorhubio-catalog
  sourceNamespace: olm
```

Rollen Sie nun den Operator per Subscription aus:

```
# k apply -f postgresql-sub.yaml
```

```
subscription.operators.coreos.com/my-postgresql created
```

Kurz darauf sollten alle OLM-spezifischen CRs die erfolgte Installation anzeigen und der GPO-Pod sollte aktiv sein:

```
# k get pods,sub,ip,csv -n operators
```

```
NAME                      READY     STATUS      RESTARTS     AGE
pgo-6d5bb8bcc4-css72      1/1       Running     0            4m7s
NAME                                          PACKAGE      SOURCE                  CHANNEL
subscription.operators[…]/postgresql-operator  postgresql   operatorhubio-catalog   v5
NAME                                            CSV                         APPROVAL    APPROVED
installplan.operators.coreos.com/install-xg92z   postgresoperator.v5.4.1   Automatic   true
NAME                                      DISPLAY           VERSION  REPLACES      PHASE
csv.operators[…]/postgresoperator.v5.4.1  Crunchy Postgr[…] 5.4.1    pgo.v5.3.0    Succeeded
```

Dann installieren Sie die nachstehende PostgresCluster-CR, die sich auch in den Beispieldaten findet. Lassen Sie sich von dem Umfang der CR nicht abschrecken: In diesem Beispiel sind bereits einige zusätzliche Settings für Backups, Monitoring etc. vorhanden. Für einen

simplen, aber trotzdem voll Operator-gestützten Postgres-Cluster reicht eine CR mit etwas mehr einem Dutzend Zeilen. Ein entsprechendes Manifest findet sich auch in den Beispieldaten.

Die verfügbaren Settings der `PostgresCluster`-CR für den PGO in Version 5.4.1 lassen Sie sich in bekannter Weise über **`k explain postgrescluster.spec.<subattributes>`** anzeigen oder alternativ über die von Ihnen verwendete PGO-Version in der Dokumentation, z. B. *https://access.crunchydata.com/documentation/postgres-operator/5.3.3/references/crd/*.

```
# postgrescluster.yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PostgresCluster
metadata:
  name: pg-cluster
spec:
 #image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-15.3-2
 #image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.8-3
 #image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.8-2
  image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres:ubi8-14.5-1
  postgresVersion: 14
  instances:
  - name: instance1
    replicas: 3
    minAvailable: 2 # PDB Settings
    resources:
      requests:
        cpu: 1200m
        memory: 4Gi
      limits:
        cpu: 2000m
        memory: 6Gi
    dataVolumeClaimSpec:
      accessModes:
      - "ReadWriteOnce"
      resources:
        requests:
          storage: 20Gi
  backups: # Backup Settings
    pgbackrest:
      global:
        repo1-retention-full: "14"
        repo1-retention-full-type: time
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgbackrest:ubi8-
2.45-2
```

```
      repos:
      - name: repo1
        schedules: # backup scheduling (cronjob syntax)
          full: "0 1 * * 0"
          differential: "0 1 * * 1-6"
          # incremental: <spec>
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 100Gi
      - name: repo2
        volume:
          volumeClaimSpec:
            accessModes:
            - "ReadWriteOnce"
            resources:
              requests:
                storage: 50Gi
      manual: # manual backup, needs annotation in postgrescluster-CR to trigger,
              # eg.: k annotate -n default postgrescluster pg-cluster \
              # postgres-operator.crunchydata.com/pgbackrest-backup="$(date)"
        repoName: repo2
        options:
        - --type=full
  proxy:
    pgBouncer:
      image: registry.developers.crunchydata.com/crunchydata/crunchy-pgbouncer:ubi8-
1.19-2
  monitoring:
    pgmonitor:
      exporter:
        image: registry.developers.crunchydata.com/crunchydata/crunchy-postgres-
exporter:ubi8-5.3.1-0
```

Der hier verwendete `pgBouncer` (»Türsteher«) ist ein Pooler für PostgreSQL-Datenbankverbindungen. Er kann eine große Zahl von externen Verbindungen zum Pooler durch eine geringere Zahl von Verbindungen vom Pooler zur Datenbank bedienen, was auf dem Datenbankserver Ressourcen spart. Die Sektion `backups` mit `pgbackrest` kümmert sich bei Bedarf um Backups des PostgresClusters. Die in der CR gezeigten Backup-Settings sorgen für regelmäßige Cronjob-Backups (*repo1*) entsprechend des Schedulings und der eingestellten Retention,

und über die `manual`-Sektion ist definiert, dass manuelle Backups (Ablage-Repo: *repo2*) bei Bedarf über entsprechende Annotations der Postgres-Cluster-CR angetriggert werden können. Das Backup erfolgt dann automatisch über Single-Shot-Jobs. Alternativ kann dies auch über das *kubectl PGO Plugin* angetriggert werden, z. B. so:

```
# k pgo backup pg-cluster --repoName repo2 --options="type=full"
```

Siehe zum *Backup-Management* auch:

- *https://access.crunchydata.com/documentation/postgres-operator/latest/architecture/backups*

Zum Setup des PGO-Plugins für **`kubectl`** siehe auch:

- *https://github.com/CrunchyData/postgres-operator-client*

Um *Rolling Updates* (nur für Minor-Versionen ohne Disruption, z. B. über das Setzen eines neuen Images) des Cluster-Verbunds triggern zu können, wurde hier zunächst eine ältere Postgres-Version (14.5) gewählt.

Die `monitoring`-Sektion ist zwar gesetzt, aber allein die Ergänzung in der CR ist noch nicht ausreichend, um via Prometheus die Metrik-Daten des Postgres-Clusters zu erfassen. Sie sorgt an dieser Stelle erst einmal dafür, dass ein zusätzlicher Metrics-Exporter-Container in die Postgres-Pods injiziert wird. Um das vollständige Monitoring zu aktivieren, sind leider noch weitere manuelle Tasks notwendig, siehe dazu auch:

- *https://access.crunchydata.com/documentation/postgres-operator/5.3.3/installation/monitoring/*
- *https://access.crunchydata.com/documentation/postgres-operator/5.3.3/installation/monitoring/kustomize/*

Die Monitoring-Capabilities sind zwar Level 4, aber für einen L5-Operator müsste dieser Schritt eigentlich besser automatisiert sein.

8.8.6 Der Zustand nach dem Rollout

Nach dem Rollout sollten Sie folgende Ressourcen sehen:

```
# k get postgresclusters,pods,sts,pdb
```

```
NAME                                                          AGE
postgrescluster.postgres-operator.crunchydata.com/pg-cluster   80m
NAME                                      READY   STATUS      RESTARTS   AGE
pod/pg-cluster-backup-7rfs-t2qkx          0/1     Completed   0          39m
pod/pg-cluster-instance1-dxk8-0           4/4     Running     0          23m
pod/pg-cluster-instance1-knm4-0           4/4     Running     0          22m
pod/pg-cluster-instance1-q6gr-0           4/4     Running     0          23m
pod/pg-cluster-pgbouncer-6dd675c647-vpshd 2/2     Running     0          80m
pod/pg-cluster-repo-host-0                2/2     Running     0          37m
```

```
NAME                                        READY   AGE
statefulset.apps/pg-cluster-instance1-dxk8  1/1     8Om
statefulset.apps/pg-cluster-instance1-knm4  1/1     8Om
statefulset.apps/pg-cluster-instance1-q6gr  1/1     8Om
statefulset.apps/pg-cluster-repo-host       1/1     8Om
NAME                                MIN AVAILABLE   MAX UNAVAILABLE   ALLOWED DISRUPTIONS   AGE
pdb.policy/pg-cluster-set-instance1 2               N/A               1                     8Om
```

Wenn Sie die Logs der StatefulSet-Pods (1 StatefulSet pro Postgres-Instanz des Clusters) prüfen, sollten Sie sehen, dass 2 davon Member sind, 1 der Leader:

```
I am (pg-cluster-instance1-knm4-0), a secondary, and following a leader (pg-cluster-instance1-dxk8-0)
```

8.8.7 Crash-Simulation

Für einen einfachen Resilienz-Test schießen Sie nun den aktuellen Master/Leader Pod des Postgres-Clusters ab, und beobachten Sie dabei den `PostgresCluster` und die Logs des PGOs.

Direkt nach der Löschung wird, wie bei allen Quorums-basierten Systemen in der Regel üblich, einer der Member zum Leader promoted. Diese Funktionalität ist keine des PGOs, sondern eine Built-In-Funktionalität von Postgres, ebenfalls ähnlich wie bei fast allen verwandten Quorums-basierten Systemen. Ab hier übernimmt jedoch der PGO: Er entdeckt über seine Reconcilation, dass der Ist-Stand vom Soll-Stand abweicht. Er weiß, welche Instanz ausgefallen ist, kümmert sich nun entsprechend der Affinity-Settings auf einem geeigneten Node um Ersatz, fügt die neue Instanz dem Cluster wieder hinzu, stellt die Replikation via TLS (sofern in der CR konfiguriert) wieder her, prüft die Replikationskonsistenz zwischen den Instanzen und beendet dann seinen Reparatureinsatz – so lange, bis der nächste Zwischenfall oder ein Upgrade oder eine Skalierung sein Eingreifen wieder erforderlich macht.

8.8.8 Skalierung

Eine Skalierung des Clusters ist ein simpler Job für den PGO, solange genügend Nodes vorhanden sind und er in keine Constraints durch *Pod Topology Spread Constraints* oder Node- und/oder Pod-Anti-Affinities rennt (siehe dazu auch Abschnitt 9.2 und folgende). Modifizieren Sie für eine Skalierung einfach die Anzahl der Replicas (`spec.instances.replicas`), und passen Sie gegebenenfalls das PDB (`spec.minAvailable`) an.

8.8.9 Upgrade

Siehe dazu auch:

https://access.crunchydata.com/documentation/postgres-operator/latest/guides/major-postgres-version-upgrade

Upgrades des PostgresClusters können verschiedene Aspekte berühren, es muss nicht immer ein neues Image oder eine komplette neue Major-Version sein: Es kann z. B. ein Up-

date der Compute-Ressourcen erfolgen oder es können andere Settings geändert werden. Wichtig ist: Der PGO führt die Änderungen sequenziell und mit der gebotenen Umsicht aus. Ohne die im betrachteten Stand (Kubernetes 1.27) leider immer noch nicht funktionierenden In-Place-Upgrades der Compute-Ressourcen ist eine Änderung der Compute-Ressourcen ein destruktiver Prozess. Daher für der PGO zunächst die Änderung sequenziell auf den Replicas aus. Ist dies bei allen Repliken ohne Fehler durchgelaufen, promoted er eine der Replicas zum neuen Master/Primary, bevor der den ehemaligen Primary als letzten updatet.

Ein Upgrade des Clusters auf eine neue *Minor-Version* (durch explizites Setzen eines Images) wird so ausgeführt, wie gerade für die Änderungen der Compute-Ressourcen beschrieben. Bei *Major-Updates* sieht die Lage etwas anders aus. Hier verlangt der PGO (leider) die strikte Einhaltung der altbewährten, aber auch aus der alten Welt stammenden Vorschriften: Backup First, Cluster == Healthy, aber trotzdem kein Rolling Upgrade, sondern Shutdown.

Führen Sie testhalber ein Upgrade des Clusters auf die nächsthöhere Major-Version aus. Dazu verwenden Sie das folgende Manifest aus den Beispieldaten:

```
# pgupgrade.yaml
apiVersion: postgres-operator.crunchydata.com/v1beta1
kind: PGUpgrade
metadata:
  name: example-upgrade
spec:
  fromPostgresVersion: 14
  postgresClusterName: pg-cluster
  toPostgresVersion: 15
```

Das Upgrade wird aus Sicherheitsgründen nicht sofort angestoßen; die CR `PGupgrade` wird anmeckern, dass der Cluster `Not stopped` ist. An dieser Stelle muss ein zweiter Sicherheitsmechanismus angetriggert werden:

```
# k -n default annotate postgrescluster pg-cluster \
postgres-operator.crunchydata.com/allow-upgrade="example-upgrade"
```

Patchen Sie dann das `spec.shutdown`-Feld per Boolean-Wert (kein String) auf `true`. Danach heißt es einfach nur warten. Die `PGupgrade`-CR startet entsprechende Jobs für das Cluster-Upgrade. Sobald der Status des `PGUpgrade`-Objekts folgende Werte hat, ist alles sauber durchgelaufen:

```
# k describe pgupgrades example-upgrade | grep Reason
    Reason:                PGUpgradeCompleted
    Reason:                PGUpgradeSucceeded
```

Patchen Sie danach die `PostgresCluster`-CR, indem Sie `spec.shutdown` auf `false` setzen und unter `postgresVersion` die neue Major-Version (`15`) eintragen. Falls Sie noch explizit ein bestimmtes 14.x-Image gesetzt hatten, ändern Sie dies bitte auch in eine valide 15er-Version

oder kommentieren Sie es aus. In dem Fall wird die aktuelle Minor-Version der neuen 15er-Major-Version genommen.

Danach fährt der Cluster in der neuen Major-Version hoch.

8.8.10 Autoscaling

Autoscaling wäre schön gewesen. Keine unüberwindbare Hürde, aber leider bot im betrachteten Stand keines der PostgreSQL-Derivate und deren Operatoren diese Funktion an. KEDA (siehe Abschnitt 11.9) würde einem direkt einfallen, aber sein PostgreSQL-Scaler kann auch nur Clients auf der Basis von SQL-Metriken der PG-DB skalieren, nicht die DB selbst. Ein (CPU-Load-abhängiges Auto-)Scaling der STS, um z. B. Member zu forken, wenn die Last zu hoch wird, wird natürlich ordnungsgemäß vom Operator wieder auf den Soll-Zustand zurückgesetzt.

Aber neben Enterprise-Operatoren für geclusterte Applikationsverbünde (wie z. B. der kommerziellen ElasticSearch-Version, die auch Autoscaling beherrscht) gibt es auch noch andere Operatoren, die – zwar nur zu Testzwecken gebaut – das volle Programm beherrschen – siehe den nächsten Abschnitt.

8.8.11 War es das zu (L5-)Operatoren?

Nur, was die Einführung angeht. In fast allen folgenden Kapiteln kommen mehr und mehr Operatoren zum Einsatz. Und was einen L5-Operator mit Vollausstattung und Autoscaling-Funktionen angeht: Auch den werden wir noch betrachten, nämlich in Kapitel 13, »Day 3 Operations: In-Cluster Vollautomation mit Operatoren – Advanced Concepts«.

Wer schon vorab in die L5-Operator-Demo hereinschnuppern will: *https://github.com/opdev/l5-operator-demo*.

```
  managementState: Managed # if CR should be managed by the operator or not
  replicationFactor: 2 # defines the policy for log stream replication.
  #size: 1x.extra-small # for testing purposes only: 5 vCPUs and 7,7 Gi RAM
  size: 1x.small # 36 vCPUs, 63 Gi RAM, Datatransfer 0,5TB Day, 50 QPS/200ms
  #size: 1x.medium # 54 vCPUs, 139 Gi RAM, Datatransfer 2TB/Day, 75 QPS/200ms
storage:
    schemas:
    - effectiveDate: "2023-08-09"
      version: v12 # v11/v12
    secret:
      name: logging-loki-s3 # secret with creds for S3 Bucket
      type: s3 # other values: azure,gcs,swift
  storageClassName: gp3-csi #generate PVs for other parts of LokiStack with this SC
  tenants:
    mode: openshift-logging
```

```
# oc apply -f lokistack-opns-aws.yaml
```

Um auf die neue Log-UI zugreifen zu können, patchen Sie noch den OpenShift-Console-Cluster-Operator:

```
# oc patch consoles.operator.openshift.io cluster \
  --patch '{ "spec": { "plugins": ["logging-view-plugin"] } }' --type=merge
```

Da das Log-Dashboard von OpenShift etwas spärlich ist, kann auch ein natives Grafana-Dashboard verwendet werden. Wie Sie Grafana mit allen potenziell zur Verfügung stehenden Dashboard-Erweiterungen nutzen und mit dem In-Cluster-LokiStack verbinden, wird in der Dokumentation beschrieben:

https://loki-operator.dev/docs/howto_connect_grafana.md/

Falls das benötigte Manifest dort nicht mehr verfügbar sein sollte, findet es sich auch in den Beispieldaten zu diesem Abschnitt.

Verfügbare Dashboards finden sich unter:

https://grafana.com/grafana/dashboards/?search=loki

11.4 Cluster-Monitoring mit Prometheus

Siehe zu allem Folgenden auch:

- *https://github.com/coreos/prometheus-operator*
- *https://github.com/prometheus-operator/kube-prometheus*
- *https://prometheus.io/docs/introduction/overview/*

Prometheus muss nicht mehr lange vorgestellt werden. Die seit Langem für das generelle Monitoring und Alerting von IT-Infrastrukturen verfügbare Lösung hat sich längst zu einem Defacto-Standard für das Monitoring von Kubernetes-basierten Clustern entwickelt. Natürlich ist Prometheus als Monitoring-Lösung nicht die einzige, und sicher nicht in jedem Fall die optimale Lösung für jeden erdenklichen Einsatzzweck. Aber sie ist dank *Operator*-gestütztem Stack schnell aufgesetzt und resilient, flexibel konfigurierbar, bietet mit *Grafana* ansprechende Visualisierungen, mit dem *Alertmanager* ein solides Werkzeug, um die DevOps-Teams über Missstände im Cluster jederzeit auf verschiedenste Arten zu informieren, mit *PromQL* die Möglichkeit, Queries/Rules für verschiedenste Anwendungsfälle zu definieren, und mit *Thanos* die Fähigkeit, multiple Prometheus-Instanzen im Sinne einer *Monitoring-Federation* zu überwachen.

Das A und O ist natürlich eine sauber arbeitende Metrik-Erfassung, damit die Metriken des Kubernetes-Core-Stacks, verschiedenster (3rd-Party-)Controller und Applikationen vollumfänglich erfasst und ausgewertet werden können.

11.4.1 Aufbau und Funktionsweise

Time Series Database

Wie alle ähnlich gearteten Lösungen legt Prometheus die erfassten Daten nicht in klassischen Datenbanken ab, sondern in einer *Time Series Database* (TSDB). Siehe dazu auch:

https://prometheus.io/docs/prometheus/latest/storage/

Die TSDB verknüpft die erfassten (Metrik-)Daten mit spezifischen Punkten auf einem Zeitstrahl, der auch den primären Index darstellt. Die Queries der Admins gehen einfach auf den gewünschten Bereich des Zeitstrahls und auf die damit assoziierten Daten. Da die Datenspeicherung in der TSDB zeitreihenbasiert arbeitet, entspricht eine solche Query ihrem nativen Datenmodell und muss nicht erst – wie bei normalen Datenbanken in diesem Einsatzfall – transponiert werden.

Sind Datenpersistenzen des TSDB-Layers auf externem Storage gewünscht, geben die folgenden Links einen Überblick:

- *https://prometheus.io/docs/prometheus/latest/storage/*
- *https://prometheus.io/docs/operating/integrations/#remote-endpoints-and-storage*

Alle Daten, die im TSDB-Layer von Prometheus gespeichert werden, werden üblicherweise in 2-Stunden-Blöcken gruppiert. Werden die Daten nur lokal gespeichert, werden sie jedoch nicht repliziert und sind damit nicht ausfallsicher. Damit die Monitoring-Daten, die je nach Cluster-Größe und Scraping-Settings sehr schnell sehr groß werden können, nicht die Nodes durch volle Plattenauslastung lahmlegen, findet per Default eine Hintergrund-Kompaktierung statt. Dabei werden, vereinfacht ausgedrückt, mehrere ältere 2-Stunden-Blöcke zu größeren Blöcken zusammengefasst und kompaktiert. Dennoch ist die Plattenauslastung durch

Prometheus, ebenso wie bei den bereits gezeigten Storage-hungrigen Logging-Stacks, ein wichtiges Kernthema und sollte permanent überwacht werden. Siehe dazu auch das gleich noch vorgestellte *Thanos*-Projekt.

Label

Jede Zeitreihe wird durch ihren Metrik-Namen und optionale Schlüssel-Wert-Paare (Label) eindeutig identifiziert. Der Metrik-Name (zulässige Zeichen bezogen auf Prometheus wären `[a-zA-Z_:][a-zA-Z0-9_:]*`) definiert eine Beschreibung über die Art der erfassten Daten (z. B. `http_requests_total`). Die Doppelpunkte sind für benutzerdefinierte Aufzeichnungsregeln reserviert. Sie sollten nicht von z. B. Exporter-Instanzen verwendet werden.

Die Label sorgen für ein (multi-)dimensionales Datenmodell innerhalb von Prometheus: Jede beliebige Kombination von Labeln für den gleichen Metrik-Namen identifiziert eine bestimmte dimensionale Instanziierung dieser Metrik (z. B. alle HTTP-Anforderungen, die die `PUT`-Methode im Bezug auf den `/api/xyz`-Handler verwendet haben).

11.4.2 Messwerte: Gauge, Counter, Histogramm und Summary

Prometheus stellt im betrachteten Stand vier Arten von Metrics zur Verfügung. Weitere Informationen und Beispiele finden Sie unter:

https://prometheus.io/docs/concepts/metric_types/

Counter

Ein *Counter* (Zähler) ist eine kumulative Metrik, die einen einzelnen monoton steigenden Zähler darstellt, dessen Wert beim Neustart nur erhöht oder auf null zurückgesetzt werden kann. Sie können beispielsweise einen Counter verwenden, um die Anzahl der bearbeiteten Requests, abgeschlossenen Tasks oder aufgetretenen Fehler darzustellen. Ein Counter sollte nicht verwendet werden, um einen Wert anzuzeigen, der absinken bzw. sich wieder verringern kann. Verwenden Sie daher beispielsweise keinen Counter für die Anzahl der aktuell laufenden Prozesse, sondern stattdessen ein *Gauge*. Unter anderem die folgenden Client-Bibliotheken unterstützen Counter: Go, Java, Python, Ruby und .Net.

Gauge

Ein *Gauge* (im wörtlichen Sinne am ehesten »Messgerät« oder »Messinstrument«) ist eine Metrik, die einen einzelnen numerischen Wert darstellt, der beliebig steigen und fallen kann. Gauges werden typischerweise für Messwerte wie Temperaturen oder die aktuelle Speichernutzung verwendet, aber auch für Counter, die steigen und fallen können, wie etwa die Anzahl gleichzeitig eingehender Requests. Die gleichen Client-Libs, die für Counter benannt wurden, unterstützen auch Gauge.

Histogramm

Ein *Histogramm* erfasst Observations wie z. B. Query-Duration oder Response-Size und zählt sie in konfigurierbaren Buckets. Dabei liefert es auch eine Summe aller beobachteten Werte. Ein Histogramm kann während eines Scrapings mehrere Zeitreihen verfügbar machen und eignet sich je nach Implementierung auch zur Berechnung eines *Application Performance Index Scores*. Bedenken Sie beim Arbeiten mit Buckets, dass das Histogramm kumulativ ist.

Summary

Wie bei einem Histogramm werden auch bei einer *Summary* ähnliche Observations erfasst. Während die Summary auch eine Gesamtzahl der Observations und eine Summe aller beobachteten Werte liefert, berechnet ein Histogramm konfigurierbare *Quantile* (»Wie viele Werte einer Datenmenge liegen unter oder über einer bestimmten Grenze?«) über ein definierbares Zeitfenster.

11.4.3 Prometheus-Komponenten im Überblick

Die wichtigsten Kernkomponenten, die bei einem Operator-gestützten Setup (alles andere ergibt in zeitgemäßen Architekturen wenig Sinn) von Prometheus ausgerollt werden, sind schnell aufgezählt:

- die eigentlichen *Prometheus*-Instanzen, die typischerweise als StatefulSet über die `Prometheus`-CR vom Operator erzeugt, ausgerollt und verwaltet werden
- die *Node-Exporter*, typischerweise als DaemonSet implementiert, um automatisch mit der Cluster-Größe zu skalieren. Details folgen gleich.
- Die *Alertmanager*-Instanzen (ebenfalls StatefulSet), die sich um die Erfassung von Fehlzuständen im Cluster und um die Weiterleitung dieser Infos an die verantwortlichen Personen oder Teams kümmern. Verwaltet werden sie über den Prometheus-Operator mithilfe der `Alertmanager`-CR.
- ein eigenständiger *kube-state-metrics*-Server, siehe weiter unten
- die *Prometheus-Adapter*, die im Grunde Adapter für die *Kubernetes Metrics API* sind und diese mit zusätzlichen Metrik-Schubladen für Custom-, weitere Ressourcen- und External-Metrics ausstatten. Sie sammeln intervallbasiert und detailliert konfigurierbar Metriken von Prometheus und exponieren diese dann in bestimmten Formaten (siehe *https://github.com/kubernetes-sigs/prometheus-adapter*).
- Und dann wäre da noch der *Blackbox-Exporter*, mit dessen Hilfe Prometheus auch mit ihm unbekannten Endpunkten kommunizieren kann und diese mit Proben verschiedenster Protokolle auf Funktion prüfen bzw. Daten exportieren kann. Unterstützt wurden im betrachteten Stand Endpunkt-Proben per HTTP(S), DNS, TCP, ICMP und gRPC.

Kube-State-Metrics

Kube-State-Metrics (KSM) ist ein einfacher Dienst, der den Kubernetes-API-Server abfragt und Metriken über den Zustand der im API-Server vorhandenen Objekte/Workloads generiert, wie z. B. Deployments, Nodes oder Pods. Kube-state-metrics wurde gebaut, um Metriken aus Kubernetes-API-Objekten – ohne Modifikationen an den Workloads – generieren zu können. Dadurch ist gewährleistet, dass die von KSM bereitgestellten Funktionen den gleichen Stabilitätsgrad aufweisen wie die Kubernetes-API-Objekte selbst. Die als Klartext bereitgestellten Metriken werden per Default über den HTTP-Endpunkt */metrics* via Port 8080 exportiert. Die bereitgestellten Metriken spiegeln natürlich im Moment der Abfrage immer den Ist-Zustand des Systems wieder, keine historischen Daten. Sobald ein Workload gelöscht wird, sind dessen historische Daten zwar noch in der TSDB von Prometheus verfügbar, aber logischerweise über den jeweiligen */metrics*-Endpunkt nicht mehr abgreifbar.

Siehe dazu auch: *https://github.com/kubernetes/kube-state-metrics*

Datensammler: Node-Exporter

Der *Node-Exporter* (üblicherweise als DaemonSet implementiert, um mit dem Cluster zu skalieren) sammelt die üblichen Daten über die Cluster-Nodes und die auf ihnen laufenden Prozesse und somit auch über die Kubernetes-Workloads: RAM-Verbrauch, Netzwerk-Traffic, CPU-Load. Da Prometheus bei der Abfrage der Metrikdaten (also dem *Scraping*) eine echte REST-API erwartet, kümmert sich der Node-Exporter um genau das und stellt Prometheus diesen Zugang bereit. Betrachtet man die werkelnden Pods des Node-Exporters tiefer unter der Haube, erkennt man schnell, dass hier kein Voodoo am Werk ist, sondern lediglich die üblichen Abfragen des */proc*- und */sys*-Dateisystems des Hosts durchgeführt werden.

Service- und Pod-Monitore

Prometheus überwacht über `ServiceMonitor`-CRs alle gewünschten Services von Kubernetes und natürlich auch den Zustand der im Cluster betriebenen Ressourcen. Wird Prometheus Operator-gestützt per `kube-prometheus` ausgerollt, wie im Folgenden beschrieben, werden dabei bereits etliche Service-Monitore für wichtige Stacks mit ausgerollt.

```
# k get servicemonitors -A
```

```
NAMESPACE    NAME                      AGE
monitoring   alertmanager-main         5h19m
monitoring   blackbox-exporter         5h19m
monitoring   coredns                   5h19m
 […]
```

Die `ServiceMonitor`-CR ist im Grunde nur eine deklarative Spezifikation, die Prometheus mitteilt, wie es eine bestimmte Applikation zu überwachen hat. Die meisten zum Einsatz kommenden `ServiceMonitor`-CRs verwenden entsprechende Selektoren, um zu definieren, welche Applikationen (typischerweise *a Bunch of Pods/Endpoints behind a Service to be scraped*)

über sie von Prometheus überwacht werden sollen, welche Namespaces durchsucht werden sollen und an welchem Port die Metriken zur Verfügung stehen. Sie stellt auch den Ansatz dar, um eigene Applikationen oder wichtige 3rd-Party-Tool-Stacks wie z. B. Ingress-Controller oder den Argo-Rollouts-Controller zu überwachen. Wie dies funktioniert, werden wir noch betrachten. Außer den Service-Monitoren existieren noch Pod-Monitore, die in bestimmten Fällen zum Einsatz kommen können, um Pods ohne assoziierte Services (und damit ohne Service-Endpoints) selektiv überwachen zu können.

Das Verständnis der Service-Monitore ist essenziell, daher sind die involvierten Komponenten in Abbildung 11.6 zusammengefasst.

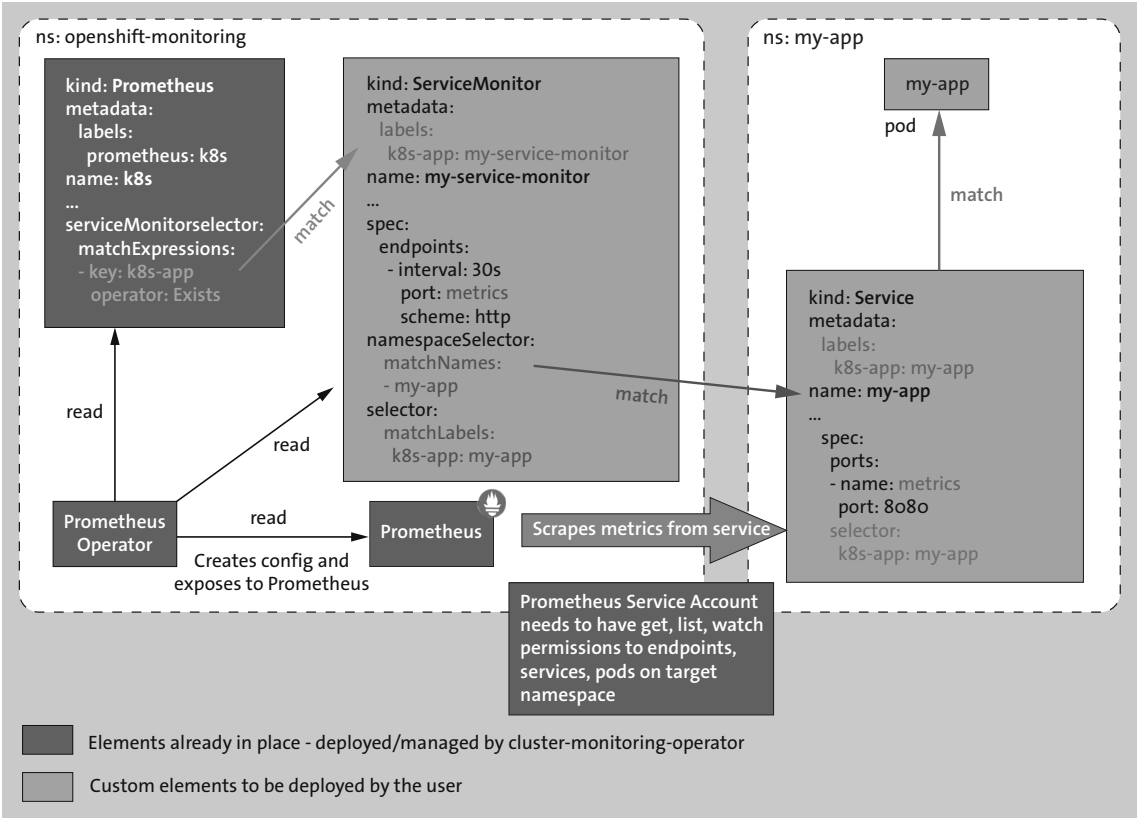


Abbildung 11.6 Service-Monitor-Dependencies in Prometheus (Quelle: fabianlee.org)

11.4.4 Prometheus-Operator und kube-prometheus

Der *Prometheus-Operator* stellt lediglich eine Teilkomponente des Stacks `kube-prometheus` dar. Er verwendet CRDs und darauf basierende CRs, um die Bereitstellung und Konfiguration von Prometheus sowie weiterer zugehöriger Überwachungskomponenten, wie z. B. des

Alertmanagers und vorgefertigter `ServiceMonitore` und `PrometheusRules`, zu vereinfachen. Allerdings bringt der standalone Prometheus-Operator diese Workloads und CRs nicht per Default mit, sie müssten (in der Regel zeitaufwendig) separat erstellt werden.

`kube-prometheus` bietet hingegen einen vollständigen Cluster-Überwachungs-Stack out-of-the-box, basierend auf Prometheus *und* dem Prometheus-Operator in Verbindung mit vorkonfigurierten CRs. Dies umfasst die Bereitstellung mehrerer Prometheus- und Alertmanager-Instanzen, Metrik-Exports über die Node-Exporter-Instanzen zum Sammeln von Knoten-Metriken, fertige Scraping-Targets und -Konfigurationen, die Prometheus mit verschiedenen Metrik-Endpunkten verknüpft, `ServiceMonitore` und Prometheus-/AlertingRules für die Benachrichtigung über potenzielle Probleme im Cluster. Thanos ist jedoch in diesem Stack in der Regel per Default nicht enthalten.

Abbildung 11.7 zeigt das Konstrukt, das wir ausrollen werden, im High-Level-Überblick.

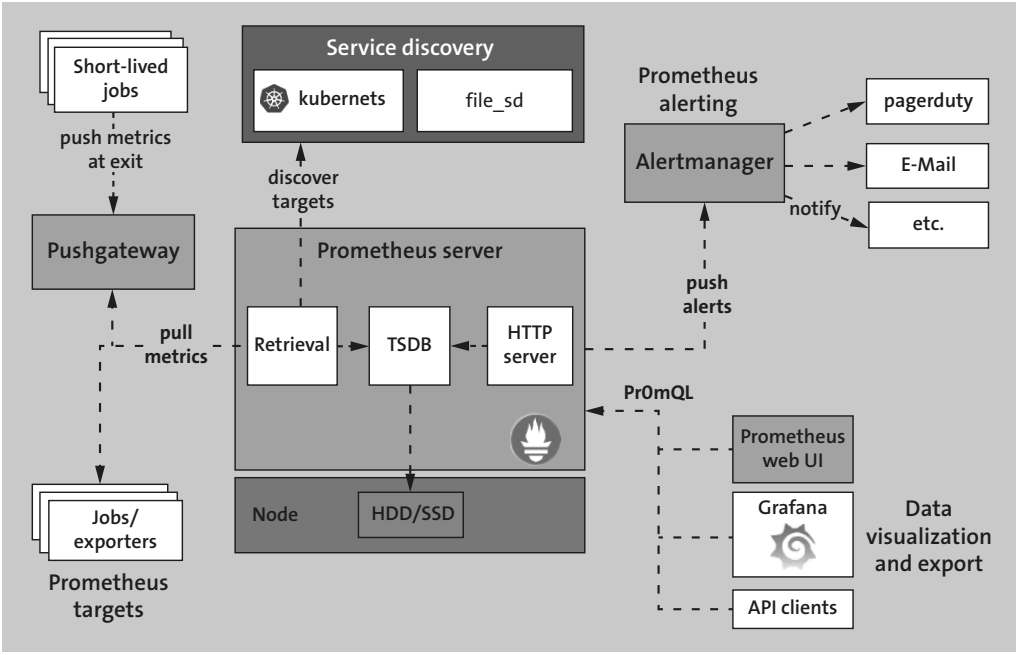


Abbildung 11.7 High-Level-Architektur von Prometheus

11.4.5 Prometheus-Installation und Betrieb – Vorbetrachtungen

Der Prometheus-Operator, dessen Entwicklung bereits 2016 begonnen wurde, kümmert sich Operator-typisch vollautomatisch um viele Belange und bietet vor allen Dingen eine unkompliziertere Abstraktionsebene über CRs. Soll z. B. die Anzahl der Repliken des Alertmanagers oder von Prometheus oder deren Version geändert werden, so sollte dies keinesfalls durch Manipulation in dem korrespondierenden StatefulSet geschehen. Die Änderungen an Work-

loads müssen dem Operator immer über die korrespondierende CR angegeben werden. Andernfalls tut der Operator das, was er soll, und stellt den in der CR definierten Soll-Zustand in Sekundenbruchteilen wieder her.

Die Prometheus-CRDs im Überblick

Der *Prometheus-Operator* arbeitet auf der Basis diverser CRDs, die im Folgenden kurz beschrieben sind:

- `Alertmanager` – definiert, wie der Alertmanager ausgerollt und betrieben werden soll.
- `Prometheus` – die Haupt-CR.
- `ThanosRuler` – definiert die gewünschten Settings des Thanos-Rulers.
- `ServiceMonitor` – legt deklarativ fest, wie Applikationen über ihre Services via Prometheus überwacht werden sollen. Der Operator generiert automatisch Prometheus-Scrape-Konfigurationen, basierend auf dem aktuellen Zustand der Objekte im API-Server.
- `PodMonitor` – legt deklarativ fest, wie Applikationen direkt als Pods überwacht werden sollen. Der Operator generiert auch hier automatisch Prometheus-Scrape-Konfigurationen, basierend auf dem aktuellen Zustand der Objekte im API-Server.
- `Probe` – legt deklarativ fest, wie Ingress-Objekte oder statische Targets überwacht werden sollen. Der Operator generiert automatisch Prometheus-Scrape-Konfigurationen, basierend auf der Definition.
- `PrometheusRule`– Prometheus-spezifische Alerting- und/oder Recording-Rules Der Operator generiert aus den Rules automatisch eine ConfigMap, die von den Prometheus-Instanzen verwendet wird.
- `AlertmanagerConfig` – legt deklarativ Abschnitte der Alertmanager-Konfiguration fest, um z. B. die Weiterleitung von Alerts über bestimmte Verfahren an definierte Empfänger zu ermöglichen.

11.4.6 Setup per kube-prometheus

Das Setup per `kube-prometheus` passt für die meisten Anwendungsfälle bzw. bildet eine gute Ausgangsbasis, da es einen kompletten, funktionsfähigen Prometheus-Stack in einem Rutsch ausrollt. Was noch fehlt, ist Thanos, der jedoch nachinstalliert werden kann. Zudem kann der Stack vor dem Rollout relativ einfach an die jeweiligen Bedingungen/Gegebenheiten angepasst werden.

Hinweis: Gescriptetes Setup

In den Beispieldaten zu diesem Abschnitt findet sich – wie auch zu anderen, komplexeren Abschnitten des Buches – aus Gründen der Zeitersparnis und Reproduzierbarkeit ein umfänglich

kommentiertes, gescriptetes Setup (*setup-prometheus-with-ingress.sh*), das den `kube-prometheus`-Stack Operator-gestützt ausrollt. Kleinere plattformspezifische Anpassungen müssen natürlich von Ihnen vorab geprüft und vorgenommen werden.

Das Script prüft auf das Vorhandensein von ExternalDNS, Cert-Manager und Nginx-Ingress, um einen TLS-geschützten Zugriff auf die Prometheus-UI und Grafana zu ermöglichen, und installiert einige der Komponenten bei Bedarf automatisch nach. Das Script klont das Git-Repo von `kube-prometheus`, prüft die vorhandene Kubernetes-Version, wählt per **`git checkout`** die korrekte Prometheus-Version und installiert sie. Danach legt es ein Wildcard-Zertifikat für eine zu definierende Domain an und bindet es in den Ingress ein.

Die folgenden Setup-Beschreibungen sind daher entsprechend kompakt gefasst.

Die wichtigsten Setup-Schritte für Prometheus sind nachfolgend kurz zusammengefasst. Zunächst wird die aktuelle Version von `kube-prometheus` aus dem Git-Repo geklont:

```
# git clone https://github.com/coreos/kube-prometheus.git; cd kube-prometheus
```

Release-Fragen

Ab diesem Punkt ist es extrem wichtig, dass die korrekte Prometheus-Release für die passende Kubernetes-Version eingestellt wird. Für eine Kubernetes-Version 1.21 oder 1.22 benötigen Sie z. B. den Branch `release-0.9`, für Version 1.24 wie hier beispielhaft gezeigt den Branch `release-0.11` – zumindest in der Theorie:

```
# git checkout release-0.11
```

Danach kann mit der Setup-Prozedur fortgefahren werden.

Tabelle 11.2 zeigt im betrachteten Stand die offizielle Kompatibilitätsmatrix zwischen Prometheus und Kubernetes (siehe hierzu auch *https://github.com/prometheus-operator/kube-prometheus*), die jedoch in der Praxis in einigen Fällen nicht so passt, wie sie in der Dokumentation angezeigt wird.

kube-prometheus-Stack	Kubernetes 1.22	Kubernetes 1.23	Kubernetes 1.24	Kubernetes 1.25	Kubernetes 1.26	Kubernetes 1.27
`release-0.9`	X	–	–	–	–	–
`release-0.10`	X	X	–	–	–	–
`release-0.11`	–	X	X	–	–	–
`release-0.12`	–	–	X	X	–	–
`main`	–	–	–	–	X	X

Tabelle 11.2 Kompatibilitätsmatrix von Prometheus und Kubernetes

Konkret lief der `kube-prometheus`-Stack unter GKE 1.24 bis 1.27 reproduzierbar in keiner Version, die größer als 0.10 ist – und damit für die benannten Kubernetes-Versionen eigentlich ungeeignet ist. Wird die offiziell empfohlene Version eingesetzt, werden Teile des Stacks (typischerweise die Node-Exporter) erst gar nicht ausgerollt und/oder der Prometheus-Stack funktioniert einfach nicht – ohne Auffälligkeiten in den Logs. Einmal mehr gilt: Willkommen im Wunderland der Vanilla-Kubernetes-Versionen – für Menschen mit viel Langeweile.

Datenpersistenzen

Um die von Prometheus erfassten Daten außerhalb von reinen Testsystemen auf persistente Datenspeicherung umzustellen (Default: `emptyDir`), können Sie entweder vor dem Rollout die `Prometheus`-CR (*manifests/prometheus-prometheus.yaml*) um die nachstehend gezeigte `storage`-Sektion ergänzen oder sie nachträglich per Patchfile hinzufügen, das Sie in den Beispieldaten finden (*prometheus-cr-storage.yaml*) (Ausgabe gekürzt):

```
spec:
[…]
  storage:
    volumeClaimTemplate:
      spec:
      # the following SC is GKE-specific
        storageClassName: standard-rwo
        resources:
          requests:
            storage: 100Gi
```

```
# k patch -n monitoring prometheus k8s \
  -p "$(cat prometheus-cr-storage.yaml)" --type merge
```

```
# k get pv -n monitoring
```

```
NAME       CAPACITY ACCESSM. RECLAIMPOL. STATUS CLAIM                          STORAGECLASS    AGE
pvc-2fc[…] 100Gi    RWO      Delete      Bound  monitoring/[…]prometheus-k8s-0  standard-rwo    42s
pvc-96d[…] 100Gi    RWO      Delete      Bound  monitoring/[…]prometheus-k8s-1  standard-rwo    42s
```

Alternativ fügen Sie die unter *https://github.com/prometheus-operator/kube-prometheus/blob/main/examples/prometheus-pvc.jsonnet* aufgelistete Konfiguration (mit der für Ihren Anwendungsfall passenden Storage-Size und `StorageClass`) in die Datei *kube-prometheus-pvc.jsonnet* Ihres gewünschten Stacks ein.

Siehe auch:

https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/user-guides/storage.md

Achtung: Pod-Priority-Kindergarten

Da es seit Erfindung der Pod-Priorities für jeden noch so kleinen Kubernetes-App-Entwickler anscheinend essenziell ist, seine superwichtige Applikation in die Pod-Priority `system-node-critical` oder `system-cluster-critical` einzustufen, kann es beim Rollout wirklich wichtiger Stacks in bestimmten Umgebungen zu einem Quota-bezogenen Fehler kommen, wie in diesem Beispiel z. B. beim Rollout eines Prometheus-Stacks unter GKE. Betroffen ist konkret das *node-exporter*-Daemonset:

»Warning FailedCreate 53s (x14 over 2m15s) daemonset-controller Error creating: insufficient quota to match these scopes: [{PriorityClass In [system-node-critical system-cluster-critical]}]«

Als einfacher Workaround kann in Testsystemen das entsprechende Quota-Limit der Pods dieser Prioritätsklasse für den betroffenen Namespace erhöht werden. In seriös aufgesetzten Systemen sollte jedoch ausdrücklich nicht inflationär mit den o. a. Pod-Priorities der `system`-Klasse umgegangen werden.

```
# k apply -f resquota-mon.yaml -n monitoring
```

Rollout

Achtung: Kein normales kubectl apply für große CRDs

Verwenden Sie bei sehr großen CRDs (wie z. B. einigen der Prometheus CRDs) nicht den Befehl **`kubectl apply`**, denn sonst schlägt das Setup der CRDs mit einer Annotation-spezifischen Fehlermeldung fehl:

»The CustomResourceDefinition `prometheuses.monitoring.coreos.com` is invalid: metadata.annotations: Too long: must have at most 262144 bytes«

Das geschieht, weil das normale **`kubectl apply`** versucht, die aktuelle und in diesem Fall sehr große (Erst-)Konfiguration zusätzlich in den Annotations zu speichern. Alternativ können Sie statt **`kubectl create`** auch ein Server-Side **`apply`** verwenden:

```
# k apply --server-side -f manifests/setup
```

Über den ersten der beiden folgenden Schritte werden zunächst die Prometheus-CRDs installiert, dann folgen die Manifeste mit den Operator-spezifischen Ressourcen und RBAC-Objekten:

```
[~ kube-prometheus]# k create -f manifests/setup
```

Warten Sie, bis die zuvor importierten CRDs im API-Server registriert sind. Es dauert typischerweise immer ein paar Sekunden, bis die importierten CRDs aktiv/verfügbar sind – umso mehr in Multimaster-Setups. Dann erst rollen Sie den eigentlichen Prometheus-Stack inklusive des Prometheus-Operators aus:

```
# k wait --for condition=Established --all CustomResourceDefinition \
  --namespace=monitoring

[~ kube-prometheus]# k apply -f manifests/

# k get crd | grep monitoring.coreos
```

```
alertmanagerconfigs.monitoring.coreos.com             2023-08-04T09:48:25Z
alertmanagers.monitoring.coreos.com                   2023-08-04T09:48:26Z
podmonitors.monitoring.coreos.com                     2023-08-04T09:48:26Z
probes.monitoring.coreos.com                          2023-08-04T09:48:26Z
prometheuses.monitoring.coreos.com                    2023-08-04T09:48:27Z
prometheusrules.monitoring.coreos.com                 2023-08-04T09:48:27Z
servicemonitors.monitoring.coreos.com                 2023-08-04T09:48:27Z
thanosrulers.monitoring.coreos.com                    2023-08-04T09:48:28Z
```

11.4.7 Post-Rollout

Nach dem Rollout sollten sich die wichtigsten Komponenten im Namespace `monitoring` wie folgt darstellen (wie üblich je nach Kubernetes-Version gegebenenfalls abweichend). Die folgenden Ressourcen wurden über das beschriebene Setup-Script aufgesetzt:

```
# k get ing,prometheus,alertmanager,pods,pv -n monitoring
```

```
NAME                CLASS HOSTS                                        ADDRESS         PORTS     AGE
ing/prometheus-ing… nginx grafana.monitoring[…],prometheus.monitor[…] 34.91.230.211   80, 443   12
NAME                                    VERSION   REPLICAS   AGE
prometheus.monitoring.coreos.com/k8s    2.32.1    2          13
NAME                                       VERSION   REPLICAS   AGE
alertmanager.monitoring.coreos.com/main    0.23.0    3          13
NAME                                       READY   STATUS     RESTARTS   AGE
pod/alertmanager-main-0                    2/2     Running    0          13m
pod/alertmanager-main-1                    2/2     Running    0          13m
pod/alertmanager-main-2                    2/2     Running    0          13m
pod/blackbox-exporter-b584f8d57-466qw      3/3     Running    0          14m
pod/grafana-7c55b97ff5-wn64t               1/1     Running    0          13m
pod/kube-state-metrics-d79445c97-lhz8h     3/3     Running    0          13m
pod/node-exporter-psgkj                    2/2     Running    0          13m
pod/node-exporter-pz7c7                    2/2     Running    0          13m
pod/node-exporter-tkfsw                    2/2     Running    0          13m
pod/prometheus-adapter-ffdc5c4f7-7gks5     1/1     Running    0          13m
pod/prometheus-adapter-ffdc5c4f7-mvngd     1/1     Running    0          13m
pod/prometheus-k8s-0                       2/2     Running    0          6m10s
pod/prometheus-k8s-1                       2/2     Running    0          6m9s
pod/prometheus-operator-85f9b9778-4l6cs    2/2     Running    0          13m
NAME       CAPACITY ACCESSM. RECLAIMPOL. STATUS CLAIM                          STORAGECLASS    AGE
pvc-2fc[…] 100Gi    RWO      Delete      Bound  monitoring/[…]prometheus-k8s-0  standard-rwo    12ms
pvc-96d[…] 100Gi    RWO      Delete      Bound  monitoring/[…]prometheus-k8s-1  standard-rwo    12m
```

11.4.8 Externer Zugriff auf die UIs

Soll der Zugriff auf die UIs von Prometheus und Grafana von außen per FQHN und Ingress (mit TLS) erfolgen, sind `ExternalDNS` und Cert-Manager Pflicht. Zum Setup siehe Abschnitt 10.1 und 10.2. Soll der Zugriff über einen GCE-Ingress möglich sein, müssen die entsprechenden Services (`grafana`, `prometheus-k8s`) auf `NodePort` oder `LoadBalancer` gepatcht werden. Sofern Sie einen GCE-Ingress verwenden, können Sie eine Static-IP anlegen mit

```
# gcloud compute addresses create ing-static-ip-prom --global (oder --regional)
```

und diese im Ingress referenzieren, z. B. so:

```
kubernetes.io/ingress.global-static-ip-name: ing-static-ip-prom
```

Leider funktioniert das hier beschriebene Verfahren über die Zuweisung per Annotation nicht mit dem Nginx-Ingress, auch wenn dieser mit statischen IPs arbeiten kann. Die UIs werden wir später betrachten, wenn wir eine Demo-App zur Überwachung implementiert haben.

Prometheus-UI

Das Prometheus-UI ist wie üblich unspektakulär, in der Vanilla-Kubernetes-Variante nicht mit weiteren Helferlein ausgestattet und – wie ebenfalls dort leider üblich – ohne irgendeine Form der Authentifizierung eingerichtet. Die UI kann hilfreich sein, wenn man einen schnellen Überblick über die aktiven Targets, Rules etc. benötigt oder ganz simpel über die Expressions nach vorhandenen Metriken sucht. Die Default-Credentials der Grafana-UI für *User* und *Passwort* sind in der Regel `admin` und `admin`. Die Dashboards können, wie bei Grafana üblich, auf vielfältige Weise angepasst werden.

Installation zusätzlicher Grafana-Plugins

Natürlich können Sie beliebige Grafana-Plugins oder vorgefertigte Dashboards nachträglich installieren. **`list-remote`** zeigt die verfügbaren Dashboards an.

Sobald der Grafana-Container durchgestartet ist, steht das neue Plugin im Dashboard unter Installed Apps zur Verfügung. Alternativ können die Plugins natürlich auch über die GUI installiert werden: Configuration • Plugins

11.4.9 Setup einer Example-App

`kube-prometheus` bringt eine einfache Example-App mit, mit der ein guter Einstieg in die Prometheus-spezifischen CRs (`prometheus`, `prometheusrules`, `servicemonitor` etc.) möglich ist. Die Variante verwendet eine eigene Prometheus-Instanz, sollte aber mit der bereits ausgerollten Instanz im Namespace `monitoring` betrieben werden. Dazu lassen Sie bei der Installation einfach das Manifest *prometheus-frontend.yaml* weg oder löschen nach dem Rollout die Prometheus-Instanz (`frontend`) im Namespace `default` und die assoziierten Services.

Achten Sie vor dem Rollout darauf, in den richtigen Git-Branch zu wechseln, andernfalls werden die Manifeste teilweise nicht in den passenden `apiVersions` generiert:

```
[~ kube-prometheus]# k apply -f examples/example-app
```

```
# k get servicemonitors,svc,deploy,pod
```

```
NAME                                          AGE
servicemonitor.monitoring.coreos.com/frontend   44m
NAME                          TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)          AGE
service/example-app           ClusterIP   10.4.5.221    <none>        8080/TCP         44m
NAME                          READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/example-app   4/4     4            4           44m
NAME                                 READY   STATUS    RESTARTS   AGE
pod/example-app-5b767f9f8f-5k85p     1/1     Running   0          44m
pod/example-app-5b767f9f8f-dstwx     1/1     Running   0          44m
pod/example-app-5b767f9f8f-jlgkg     1/1     Running   0          44m
pod/example-app-5b767f9f8f-svgqc     1/1     Running   0          44m
```

Wie unschwer zu erkennen ist, bringt die Example-App einen eigenen `ServiceMonitor` mit, an dem sich die Funktionalität gut nachvollziehen lässt.

```
# k describe servicemonitors frontend
```

```
Name:         frontend
Namespace:    default
Labels:       tier=frontend
API Version:  monitoring.coreos.com/v1
Kind:         ServiceMonitor
[…]
Spec:
  Endpoints:
    Interval:  10s
    Port:      web
  Namespace Selector:
    Match Names:
      default
  Selector:
    Match Labels:
      Tier:  frontend
  Target Labels:
    tier
Events:  <none>
```

Über den `ServiceMonitor` werden die Metrics der Endpoints der Example-App alle 10 Sekunden erfasst. Sie sollten die neue Applikation in den verfügbaren Targets der Prometheus-UI sehen.

Sie können nun, ähnlich wie im Abschnitt über den HPA (siehe Abschnitt 11.8) gezeigt, die App per Load-Generator testhalber etwas unter Stress setzen und die Auslastung in der Grafana-UI verfolgen (hier nach einem 2. Testlauf und Restart der App durch Hinzufügen von Request und Limits).

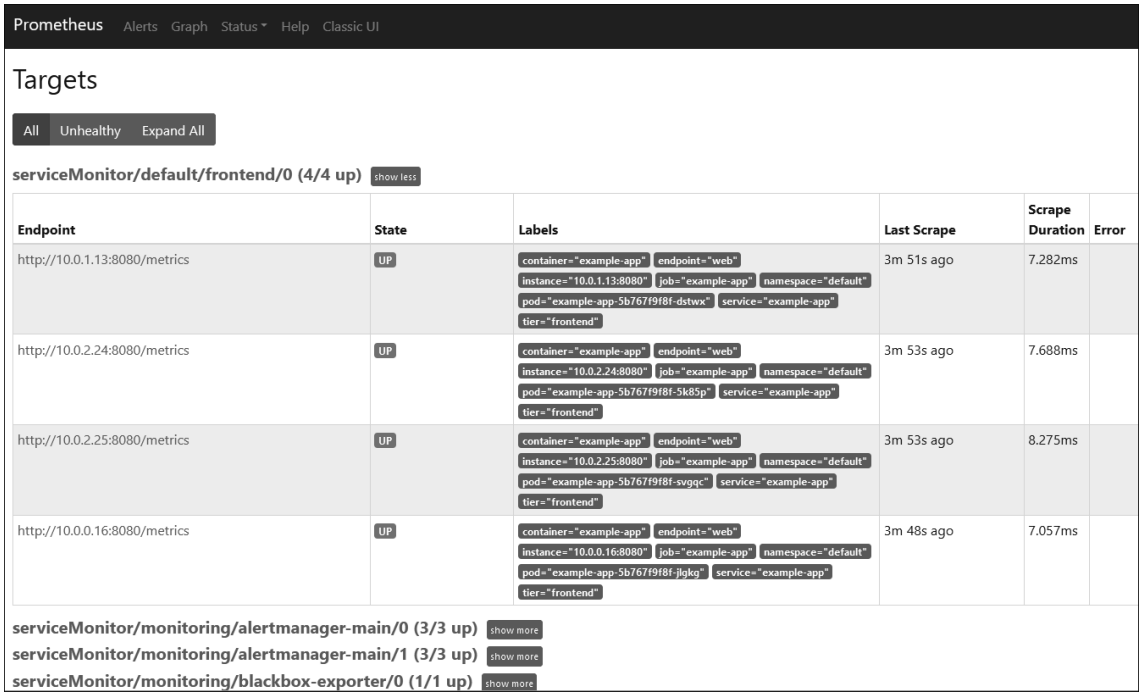


Abbildung 11.8 Die Example-App mit ServiceMonitor als neues Target in der Prometheus-UI

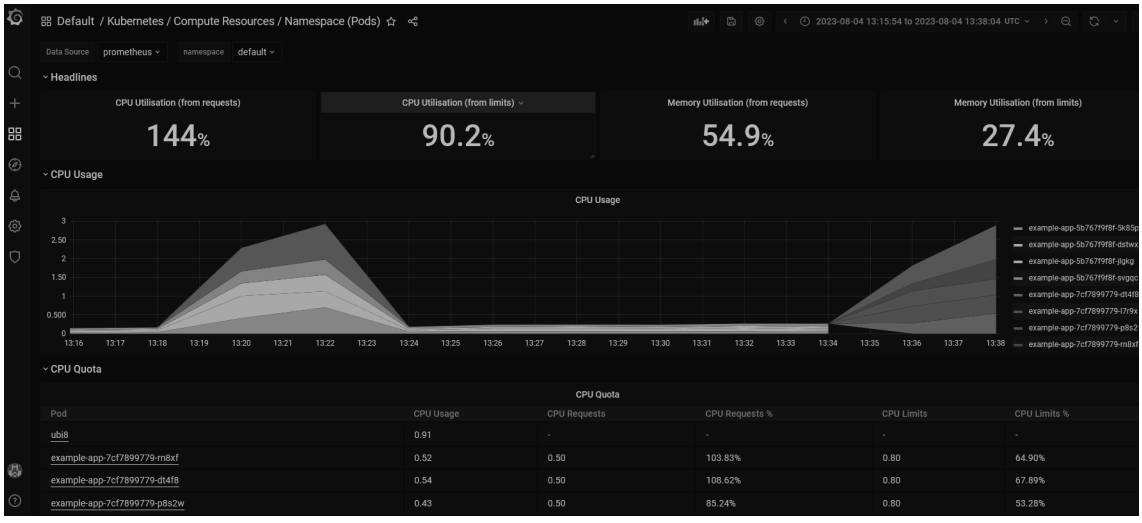


Abbildung 11.9 Example-App in Grafana während eines Load-Tests

11.4.10 PodInfo mit ServiceMonitor

Ein weiteres `ServiceMonitor`-Beispiel ist das zwar schon etwas betagte, aber seit Jahren bewährte Podinfo-Beispiel von Stefan Prodan. Es stellt über den */metrics*-Endpoint der Applikation einige Metriken zur Verfügung, u. a. `http_request*`, `request_duration_seconds` und `requests_total`. Leider existiert keine rein Manifest-basierte Rollout-Variante mehr, sondern nur noch die Chart-basierte Helm-Version. Die Details zur Applikation (u. a. welche Metrics bereitgestellt werden und welche Setup-Schalter existieren) sind unter den folgenden URLs recht gut beschrieben:

- *https://github.com/stefanprodan/podinfo*
- *https://github.com/stefanprodan/podinfo/blob/master/charts/podinfo/README.md*

Um den Stack mit inkludiertem ServiceMonitor auszurollen, gehen Sie wie folgt vor:

```
# helm repo add podinfo https://stefanprodan.github.io/podinfo
```

```
# helm install --wait --create-namespace podinfo --set=serviceMonitor.enabled=true,
serviceMonitor.interval=5s,replicaCount=2,service.type=LoadBalancer podinfo/podinfo -n
podinfo
```

Beachten Sie auch in diesem Fall die benötigten RBAC-Updates für die Prometheus-Instanzen (siehe den nächsten Abschnitt).

```
# k apply -f prometheus-clusterrole-patched.yaml
```

11.4.11 Service-Monitore für infrastrukturrelevante Stacks einrichten: Ingress

Sobald abseits einer Prometheus-Einstiegskonfiguration weitere wichtige Stacks im Cluster deployed werden, müssen für diese auch Service-Monitore eingerichtet werden. Nur so können potenzielle Probleme erkannt und die Metriken z. B. auch zum lastabhängigen Autoskalieren verwendet werden.

Wenn Sie z. B. Metriken des Nginx-Ingress-Controllers – jenseits von CPU, Memory und den üblichen Verdächtigen – via `ServiceMonitor` erfassen möchten, legen Sie zunächst eine passende CR an (findet sich in den Beispieldaten zu diesem Abschnitt):

```
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  namespace: ingress-nginx
  name: ingress-nginx
  labels:
    app: nginx-ingress
    #release: prometheus-operator
spec:
  endpoints:
```

```
  - interval: 30s
    #port: metrics
    port: prometheus #<-- immer den Portnamen aus dem Applikations-Service nehmen!
  selector:
    matchLabels:
      app.kubernetes.io/name: ingress-nginx
      controller-service: "true"
      #release: nginx-ingress
  namespaceSelector:
    matchNames:
    - ingress-nginx
```

k apply -f servicemon-ingress.yaml

```
servicemonitor.monitoring.coreos.com/nginx-ingress-controller-metrics created
```

Zudem müssen Sie den Service und das Deployment des Nginx-Ingress-Controllers wie folgt ergänzen:

```
# nginx-ingress-svc-prometheus-patch.yaml
metadata:
  labels:
    controller-service: "true"
spec:
  ports:
  - name: prometheus
    port: 10254
    targetPort: prometheus
# nginx-ingress-deploy-prometheus-patch.yaml
spec:
  template:
    metadata:
      annotations:
        prometheus.io/scrape: "true"
        prometheus.io/port: "10254"
    spec:
      containers:
        - name: controller
          ports:
            - name: prometheus
              containerPort: 10254
```

**# k patch -n ingress-nginx svc ingress-nginx-controller **
** -p "$(cat nginx-ingress-svc-prometheus-patch.yaml)"**

```
# k patch -n ingress-nginx deploy ingress-nginx-controller \
  -p "$(cat nginx-ingress-deploy-prometheus-patch.yaml)"
```

Sollten kurz darauf weder das neue Nginx-Ingress-Target noch die Metrics verfügbar sein, checken Sie die Logs der Prometheus-Instanzen. Sehr wahrscheinlich liegen dann unzureichende RBAC-Regeln für die ClusterRole `prometheus-k8s` vor, um die Objekte im Namespace `ingress-nginx` zu überwachen. Eine entsprechend gepatchte Variante findet sich in den Beispieldaten:

```
# k apply -f prometheus-clusterrole-patched.yaml
```

Das Prometheus-StatefulSet sollte kurz danach den neuen `ServiceMonitor` erkennen. Falls es Ihnen nicht schnell genug geht, starten Sie (nur auf einem Testsystem) das StatefulSet durch, um die Änderungen sofort bekannt zu machen:

```
# k rollout restart statefulset -n monitoring prometheus-k8s
```

Ist alles korrekt eingerichtet, sollten Sie das neue Target kurz darauf in der Prometheus-UI sehen, und die Ingress-Metrics (im betrachteten Stand über 50) sollten ebenfalls zur Verfügung stehen.

Abbildung 11.10 Das Nginx-Ingress-Target in der Prometheus-UI

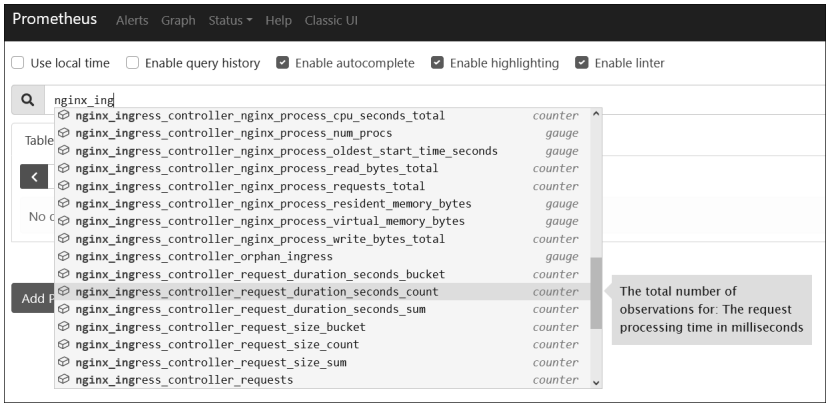


Abbildung 11.11 Metrics zum Nginx-Ingress in der Prometheus-UI

Service-Monitore für infrastrukturrelevante Stacks einrichten: Argo-Rollouts

Die Verfahren, um die Metriken des Argo-Rollout-Controllers per Prometheus zu erfassen, sind ab Abschnitt 21.7 aufgeführt.

11.4.12 Debugging des Service-Monitors

Da das Debugging des Service-Monitors manchmal durchaus anspruchsvoller sein kann, sind nachfolgend einige der wichtigen Verfahren aufgelistet. Weiter Infos finden Sie unter:

https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/troubleshooting.md#troubleshooting-servicemonitor-changes

Wurde der `ServiceMonitor` von Prometheus erfasst? (Hier z. B. für einen `ServiceMonitor` mit der Bezeichnung `ingress-nginx`):

```
# k -n monitoring get secret prometheus-k8s -ojson | \
  jq -r '.data["prometheus.yaml.gz"]' | base64 -d | gunzip | grep "ingress-nginx"
```

```
- job_name: serviceMonitor/ingress-nginx/ingress-nginx/0
      - ingress-nginx
    regex: (ingress-nginx);true
```

Welcher Service wurde vom `ServiceMonitor` erfasst:

```
# k get services -l "$(kubectl get servicemonitors -n ingress-nginx "ingress-nginx" -o
template='{{ $first := 1 }}{{ range $key, $value := .spec.selector.matchLabels }}
{{ if eq $first 0 }},{{end}}{{ $key }}={{ $value }}{{ $first = 0 }}{{end}}')" -n ingress-nginx
```

```
NAME                                 TYPE           CLUSTER-IP    EXTERNAL-IP     PORT(S)          AGE
ingress-nginx-controller             LoadBalancer   10.4.1.2      34.91.230.211   10254:31499[…]   5h56m
ingress-nginx-controller-admission   ClusterIP      10.4.2.105    <none>          443/TCP          5h56m
```

In diesem Fall wurden 2 Services vom `ServiceMonitor` erfasst, er soll aber nur den echten Service erfassen, was in diesem Fall durch ein zusätzliches Label des Service `nginx-ingress-controller` gelöst wurde:

```
controller-service: "true"
```

Die korrespondierende Änderung im `ServiceMonitor`-Objekt lautet dann:

```
selector:
  matchLabels:
    app.kubernetes.io/name: ingress-nginx
    controller-service: "true"
    #release: nginx-ingress
```

Beachten Sie: Wenn der vom `ServiceMonitor` überwachte Service einen Namen statt Port verwendet, muss dieser Name auch im `ServiceMonitor` so verwendet werden, nicht die Portnummer (siehe Abbildung 11.12).

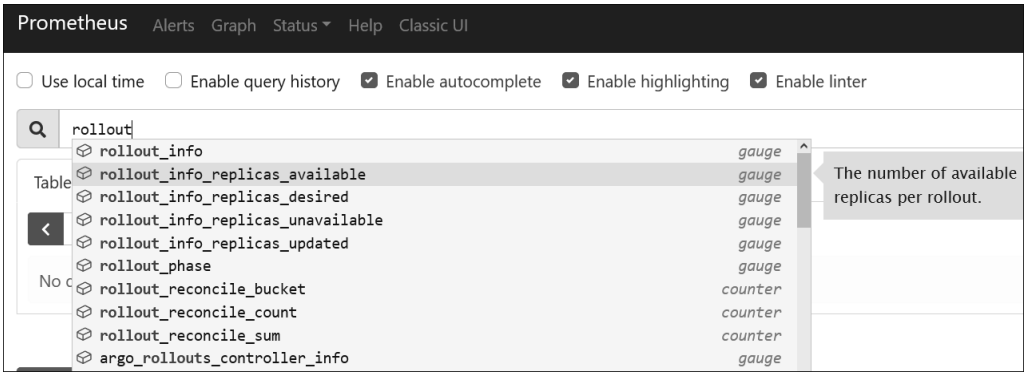


Abbildung 11.12 Metriken des Argo-Rollout-Controllers in Prometheus

Um die Metriken eines Rollouts zu erfassen, um z. B. ein `AnalysisTemplate` auf Prometheus-Basis nutzen zu können, gehen Sie analog vor. Für einen validen Health-Check muss die Applikation natürlich in der Lage sein, passende Metriken zu liefern.

Dashboard-Extension

Wie Sie Grafana um ein für den Argo-Rollout-Controller angepasstes Dashboard erweitern, ist in der Dokumentation beschrieben. Die zweite URL beinhaltet die JSON-Datei, die Sie in Grafana importieren müssen:

- *https://argo-rollouts.readthedocs.io/en/latest/features/controller-metrics/*
- *https://github.com/argoproj/argo-rollouts/blob/master/examples/dashboard.json*

11.4.13 Alertmanager und Alert-Receivers

Eine weitere essenziell wichtige Funktion von Prometheus ist das Alerting. Bei einem Setup via `kube-prometheus` werden etliche vorgefertigte und für Kubernetes-Cluster wichtige `AlertingRules` in den Cluster importiert und über die ebenfalls im `kube-prometheus`-Setup per Default ausgerollten Alertmanager-Instanzen prozessiert. Abbildung 11.13 zeigt auszugsweise eine Übersicht.

Alerts

Um exemplarisch Alerts – z. B. für die Alerts `KubePodCrashLooping` oder `ReplicasMismatch` – zu triggern, sabotieren Sie z. B. einfach die Live- und Readiness-Probes eines Test-Deployments und lassen sich dann die *Pending*- und *Firing*-Alerts anzeigen (siehe Abbildung 11.14).

Die Unterschiede der Alert-States sind folgende:

- **Inactive (grün)** – Kein ausgelöster Alert
- **Pending (gelb)** – Alert wurde bereits (mehrmals) angetriggert, aber die zulässige Fehler-Zeitspanne, bis der Alarm als `Firing` gilt, ist noch nicht erreicht
- **Firing (rot)** – Alert wurde (mehrmals) angetriggert und hat die zulässige `Pending`-Zeitspanne überschritten.

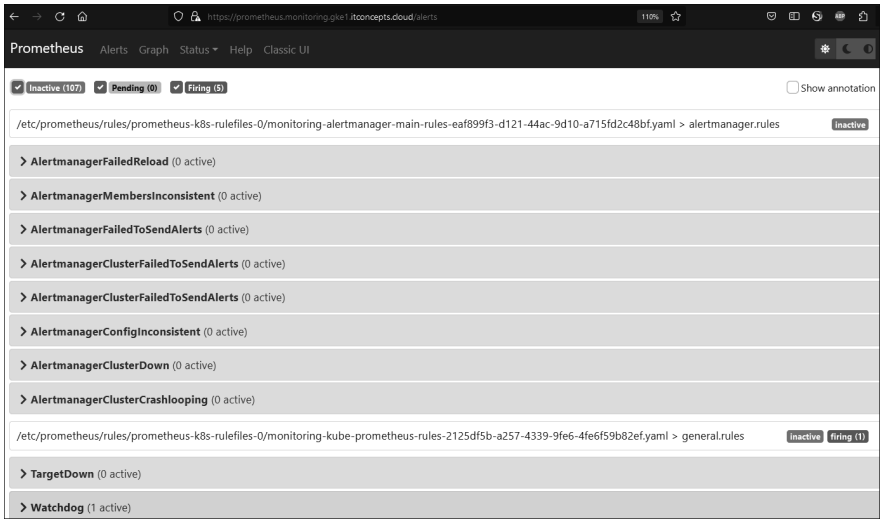


Abbildung 11.13 Alerts in Prometheus

Abbildung 11.14 Ausgelöste Pending- und Firing-Alerts in Prometheus

Alert-Receivers

Logischerweise wird niemand mit halbwegs intaktem Realitätsbewusstsein wie ein Ops-Monkey 24/7 vor der UI sitzen und warten, bis ein Alert getriggert wird, um auf diesen reagieren zu können. Prometheus kennt selbstverständlich zahlreiche Möglichkeiten, um die Mitglieder der DevOps-Teams von Missständen im Cluster zu unterrichten. Die *Alert-Receivers* werden typischerweise in einer entsprechenden `AlertmanagerConfiguration` (CR) festgelegt. Templates finden sich z. B. im `kube-prometheus`-Unterordner (*examples/alertmanager-config.**), üblicherweise mit einem per Default bereits gesetzten `null`-Receiver, der für einen Slack-Channel konfiguriert ist. Als weitere Receiver stehen z. B. E-Mail, Pager, Pushover, Opsgenie oder Webhooks zur Verfügung.

Eine sehr einfache Slack-Channel-Konfiguration könnte wie folgt aussehen. Voraussetzung ist natürlich ein Slack-Channel mit aktivierten Incoming-Webhooks für den Channel. Die Webhook-URL des Channels muss natürlich bekannt sein.

Im folgenden Beispiel enthält die Datei *alertmanager.yaml* unter anderem die Webhook-URL, die typischerweise wie folgt aussieht:

```
https://hooks.slack.com/services/<ID>/<ID>/<ID>
```

Mithilfe dieser Datei wird ein Secret in dem Namespace erzeugt, von dem Alerts in den Slack-Channel gepostet werden sollen:

```
# alertmanager.yaml
global:
  resolve_timeout: 5m
  slack_api_url: https://hooks.slack.com/services/[REDACTED]/[REDACTED]/[REDACTED]
receivers:
- name: slack-notifications
  slack_configs:
  - channel: '#k8s-alerting'
    send_resolved: true
route:
  group_by:
  - alertname
  group_interval: 30s
  group_wait: 30s
  repeat_interval: 60s
  receiver: slack-notifications
```

Achtung: Secret-Name!

Beachten Sie, dass der Name des Secrets immer der Nomenklatur `alertmanager-<NAME des Alert-Managers>` entsprechen muss! In einer Operator-installierten `kube-prometheus`-Konfi-

guration wäre dies per Default üblicherweise `main`. Sie können dies per **`k get alertmanager -n monitoring`** abfragen:

```
NAMESPACE   NAME  VERSION  REPLICAS  AGE
monitoring  main  0.23.0   3         3h41m
```

Dann erzeugen Sie die Secrets, hier beispielhaft für die Namespaces `default` und `monitoring`:

```
# k create secret generic alertmanager-main \
  --from-file alertmanager.yaml -n default

# k create secret generic alertmanager-main \
  --from-file alertmanager.yaml -n monitoring
```

Ebenfalls muss noch die `Alertmanager`-CR *manifests/alertmanager-alertmanager.yaml* angepasst und re-importiert werden, damit sie die Konfiguration, die in den eben importierten Secrets hinterlegt ist, erkennt:

```
[…]
spec:
[…]
  secrets:
    - alertmanager-main
  image: quay.io/prometheus/alertmanager:v0.23.0
  nodeSelector:
    kubernetes.io/os: linux
[...]
```

Ist das Setup korrekt eingerichtet, sollten die Alerts kurz danach im Slack-Channel auftauchen (siehe Abbildung 11.15).

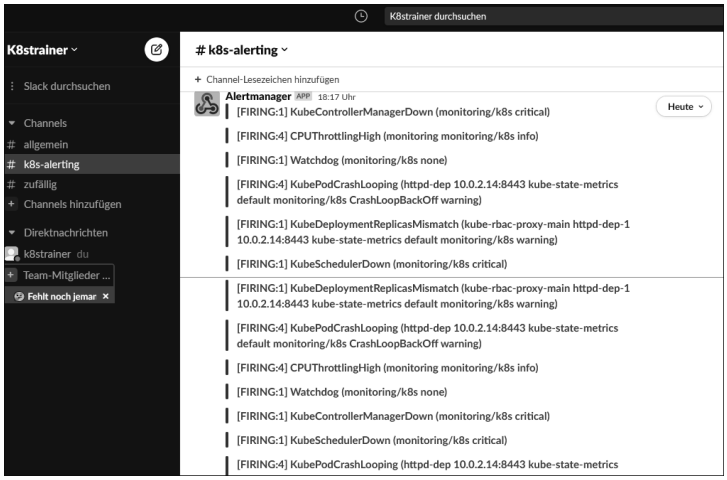


Abbildung 11.15 Alerts des Prometheus-Alertmanagers im Slack-Channel

Für alle Konfigurationsdetails des Alertmanagers siehe auch:

- *https://prometheus.io/docs/alerting/latest/configuration/*
- *https://docs.openshift.com/container-platform/4.12/monitoring/managing-alerts.html*

11.4.14 Eigene Alerting-Rules hinzufügen

Rulefiles von Prometheus (`PrometheusRules`) finden sich augenscheinlich zunächst in der ConfigMap `prometheus-k8s-rulefiles-0` im Namespace `monitoring`, die in die Prometheus-Pods (*prometheus-k8s-0|1*) eingemountet ist. Diese ConfigMap zu modifizieren, bringt jedoch rein gar nichts, da sie über die CR `prometheusrules.monitoring.coreos.com` durch den Operator automatisch (neu) erzeugt bzw. bei Änderungen aktualisiert wird. Möchten Sie also Rulefiles für Ihre Anwendungsfälle modifizieren oder neue hinzufügen, setzen Sie an der passenden `PrometheusRule` an:

```
# k get prometheusrules -n monitoring

NAME                              AGE
alertmanager-main-rules           4h9m
kube-prometheus-rules             4h9m
kube-state-metrics-rules          4h9m
kubernetes-monitoring-rules       4h9m
node-exporter-rules               4h9m
prometheus-k8s-prometheus-rules   4h9m
prometheus-operator-rules         4h9m
```

Für das folgende Beispiel verwenden wir eine neue Alerting-Rule, die auf der Regel `KubePodCrashLooping` basiert (aus dem Manifest *kubernetesControlPlane-prometheusRule.yaml* im Ordner *kube-prometheus/manifests*), aber in einigen Punkten modifiziert ist (`Alert Name`, `Summary`, `Serverity`):

```
# prometheusRule-mod.yaml
apiVersion: monitoring.coreos.com/v1
kind: PrometheusRule
metadata:
[…]
  name: kubernetes-monitoring-rules-addon
  namespace: monitoring
spec:
  groups:
  - name: kubernetes-apps
    rules:
    - alert: KubePodCrashLooping2
      annotations:
        description: 'Pod {{ $labels.namespace }}/{{ $labels.pod }} ({{
$labels.container }}) is in waiting state (reason: "CrashLoopBackOff").'
```

```
        runbook_url: https://runbooks.prometheus-operator.dev/runbooks/kubernetes/
kubepodcrashlooping
        summary: Pod is REALLY crash looping
      expr: |
max_over_time(kube_pod_container_status_waiting_reason{reason="CrashLoopBackOff",
job="kube-state-metrics"}[5m]) >= 1
      for: 10m
      labels:
        severity: critical
[…]
```

```
# k apply -f prometheusRule-mod.yaml
```

```
prometheusrule.monitoring.coreos.com/kubernetes-monitoring-rules-addon created
```

Die korrespondierende CM wird automatisch neu erzeugt und enthält die neue Regel, die sofort aktiv ist:

```
# k get cm -n monitoring prometheus-k8s-rulefiles-0
```

```
NAME                          DATA   AGE
prometheus-k8s-rulefiles-0    7      56s
```

```
# k get cm -n monitoring prometheus-k8s-rulefiles-0 \
  -o yaml | grep -i KubePodCrashLooping2
```

```
      - alert: KubePodCrashLooping2
```

Abbildung 11.16 Die neue Prometheus-Alerting-Rule in der UI

11.4.15 Prometheus-HA, Scaling und Sharding

Siehe dazu auch:

https://github.com/prometheus-operator/prometheus-operator/blob/main/Documentation/user-guides/shards-and-replicas.md

Bis zu diesem Punkt haben wir uns auf das Verständnis der in Unternehmensumgebungen elementar wichtigen Funktionalitäten von Prometheus fokussiert. Weitere wichtige Aspekte in diesem Kontext stellen jedoch auch Hochverfügbarkeit, Skalierbarkeit und Performance der lokalen Prometheus-Instanz dar (das Thema *Federated Prometheus* wird im nächsten Abschnitt mit Thanos erörtert). Hier kommen, ähnlich wie bei dem bereits vorgestellten Elastic, Replicas und vor allem *Shards* ins Spiel.

Um Prometheus hochverfügbar auszuführen, sollten zwei (oder mehr) STS-Instanzen mit derselben Konfiguration und mit PVs im Cluster ausgeführt werden (Prometheus-CR: `spec.replicas`). Die Prometheus-Instanzen scannen dabei dieselben Targets und werten die gleichen Rules aus, daher verfügen alle Prometheus-Instanzen eines lokalen Clusters über dieselben Daten im RAM und On-Disk – aber mit dem kleinen Unterschied, dass die Scrapings und Auswertungen aufgrund ihrer unterschiedlichen externen Bezeichnung nicht exakt zur gleichen Zeit erfolgen. Infolgedessen können identische Queries, die gegen die Prometheus-Instanzen ausgeführt werden, minimal unterschiedliche Ergebnisse liefern. Die Alerts bleiben davon unberührt, da sie im Vergleich zu Metriken relativ zeitunkritisch sind, sobald sie einmal ausgelöst wurden. Für UI-Sessions empfehlen sich daher Service-spezifische Einstellungen zur Session-Stickiness.

Das Ausführen mehrerer Prometheus-Instanzen sorgt zwar dafür, dass kein Single Point of Failure vorhanden ist, hilft aber nicht bei der Skalierung von Prometheus, falls eine Prometheus-Instanz lasttechnisch nicht mehr alle Targets und Rules verarbeiten kann. Hier kommt die experimentelle *Sharding*-Funktion des Prometheus-Operators ins Spiel. Beim Sharding geht es darum, die Scrape-Targets in mehrere Gruppen aufzuteilen, die dann jeweils einem bestimmten Prometheus-Shard zugeordnet und klein genug gehalten werden können, um von einer einzigen Prometheus-Instanz bearbeitet werden zu können. In der Prometheus-CR können die zu verwendenden Shards manuell wie folgt eingestellt werden:

```
spec:
  replicas: 2
  shards: 2
```

Für das o. a. Setting würden sich insgesamt 4 erzeugte Pods pro Cluster ergeben (2 Shards pro Prometheus-Instanz, das Ganze mal 2 Replicas):

```
prometheus-k8s-0                3/3     Running   0          26s
prometheus-k8s-1                3/3     Running   0          26s
prometheus-k8s-shard-1-0        3/3     Running   0          26s
prometheus-k8s-shard-1-1        3/3     Running   0          26s
```

Die Prometheus-Entwickler empfehlen, wenn möglich, ein funktionales Sharding einzurichten: In diesem Fall scraped Prometheus-Shard X alle Pods von Service A, B und C, während Shard Y Pods von Service D, E und F scraped.

Ist ein funktionales Sharding nicht möglich, kann der Prometheus-Operator auch automatisches Sharding unterstützen: In dem Fall werden die Targets anhand der Hashwerte ihrer

Source-Label (»*Sharding is done [based]on the content of the* `__address__` *target meta-label*«) bestimmten Prometheus-Shards zugewiesen. Die gesamte Anzahl der vom Prometheus-Operator erstellten Pods ergibt sich aus der Anzahl der kalkulierten Shards multipliziert mit dem aktuell eingestellten Replica-Wert des Prometheus-STS. Beim Up- oder Downscaling von Shards ist jedoch zu beachten, dass durch das Skalieren nicht automatisch eine Umverteilung auf die verbleibenden bzw. insgesamt vorhandenen Shard-Instanzen erfolgt. Die Daten müssen im betrachteten Stand noch manuell verschoben/rebalanced werden.

Eine Automation dieser Aufgabe durch den Prometheus-Operator wäre wünschenswert. Daher gilt: Solange der Prometheus-Operator das Auto-Sharding/-Rebalancing beim Up- und Downscaling noch nicht vollumfänglich beherrscht, stellen Sie die Größe lieber manuell ein.

Ein weiterer Nachteil der Sharding-Lösung ist die zusätzliche Komplexität: Um alle Daten abzufragen, muss eine Query-Federation (z. B. *Thanos Query*) und eine verteilte Regelauswertungs-Engine (z. B. *Thanos Ruler*) eingesetzt werden, um alle relevanten Daten für Queries und Regelauswertungen erfassen zu können.

11.5 Federated Prometheus mit Thanos

Siehe auch: *https://thanos.io/*

Das Monitoring eines Clusters per Prometheus ist nett, aber in der Praxis nur der erste Schritt von vielen. Typischerweise sind in Unternehmensumgebungen, je nach Größe, Dutzende oder Hunderte Kubernetes-basierte Cluster in verschiedenen Staging- und Tenant-Umgebungen vorhanden, die per Monitoring überwacht werden müssen. Natürlich existieren neben »Kubernetes-nativen« Lösungen wie Prometheus und dem in diesem Abschnitt vorgestellten Thanos mittlerweile gefühlt zahllose (zum großen Teil kommerzielle) Monitoring- und APM-Lösungen (siehe Abschnitt 11.7), die alle den Anspruch erheben, *das* ultimative Tool für diese Aufgabe zu sein. In der Praxis bietet jedes dieser Tools seine Vor- und Nachteile, wie in dem o. a. Abschnitt später betrachtet wird.

11.5.1 Thanos

Wer jedoch seine einzelnen Prometheus-Cluster mit dem frei verfügbaren und durchaus leistungsfähigen *Thanos* zu einem Federated-Verbund zusammenführen will, ist damit nicht schlecht beraten und befindet sich in der Gesellschaft bekannter Namen wieder: Auch Unternehmen wie Adobe, Bytedance, eBay, FreeNow, ProSiebenSat.1 Media, Tencent Holdings Ltd, XING und etliche andere, z. T. milliardenschwere Unternehmen setzen auf diese Lösung.

Federated-Prometheus-Cluster, Hochverfügbarkeit der gescrapten Daten aller Cluster, eine Single-Query-API, um auf Metriken verschiedener Prometheus-Instanzen zuzugreifen, skalierbare Langzeitaufbewahrung von Metriken via ObjectStore und anderes mehr: Im Prometheus-Umfeld kann sich *Thanos* um diese Aspekte kümmern, und das – ebenso wie Pro-

wünscht sein, z. B. in Testclustern live und schnell Konfigurationsänderungen zu testen, ohne die CR zu verbiegen. Für diese Szenarien können Sie den State wie folgt temporär auf *Unmanaged* setzen:

```
# oc patch dns.operator.openshift.io default --type merge \
  --patch '{"spec":{"managementState":"Unmanaged"}}'
```

Die Revertierung auf den Managed State erfolgt analog.

16.12 MachineConfigs, Machines, MachineSets und Scaling

OpenShift Container Platform 3 arbeitete per Default noch mit drei verfügbaren Node-Rollen: *Master*, *Compute* und *Infra*, Letztere für die üblichen Verdächtigen: Logging, Observability/APM und andere. OCP 4 bietet standardmäßig nur noch Master- und Worker-Roles, keine Infra-Role mehr. Dies ist nicht als Manko zu sehen, da die Container-Teams nach dem Rollout selbst wählen können, ob und welche Nodes als Infra-Nodes (oder in anderen Rollen) dienen sollen.

Achtung: IPI vs. UPI

Die folgenden Verfahren beziehen sich ausschließlich auf IPI-(*Installer Provisioned Infrastructure*-)basierte Installationen. UPI-(*User Provisioned Infrastructure*-)basierte Installationen erfordern nachträglich einige Tasks, um die nachstehend beschriebenen Verfahren nutzen zu können. Siehe dazu auch:

https://docs.openshift.com/container-platform/4.12/machine_management/creating-infrastructure-machinesets.html

Die (nachträgliche) Klassifizierung von Nodes bzw. deren Zuordnung zu unterschiedlichen Rollen/Kategorien ist unter OpenShift dank der durchdachten Machine-API und den zugeordneten Operatoren und CRs eine relativ einfache Aufgabe.

Der Machine-API-Stack arbeitet auf der Basis verschiedener Operatoren und CRs, die für bestimmte Aufgabenbereiche zuständig sind und die wir uns hier zunächst in einer auszugsweisen, kurzen Übersicht und später detaillierter ansehen:

- **MachineConfigs** – Über sie können *Machine*-Objekte konfiguriert werden. Sie bestimmen z. B., welche Dateien mit welchem Content auf den RHCOS-Nodes während der Provisionierung landen, wie z. B. SSH-, Container-Engine-, GPU-Node- oder Kubelet- spezifische Konfigurationen. Ressourcen-Kürzel: `mc`.
- **MachineSets** – Sie sind prinzipiell nichts anderes als ReplicaSets für Machine-Objekte. Über sie können Machines hochverfügbar und skalierbar gehalten werden. So können auf der Basis von MachineSets auf manuellen Zuruf oder per Cluster-Autoscaler automatisch,

on-demand weitere Nodes (bzw. *Machines*) gemäß einer per *MachineConfig* hinterlegten Konfiguration erzeugt und dem Cluster hinzugefügt werden. Auch ein (Auto-)Downscaling eines bestimmten MachineSets auf 0 ist möglich, sofern der Cluster genügend verbleibende Compute-Ressourcen hat, um alle aktiven Workloads zu bedienen.

- **MachineConfigPools** – im Prinzip und sehr stark vereinfacht eine logische Gruppierung für Machines verschiedener Kategorien, wie z. B. Master, Worker, Infra, GPU ...
- **MachineAutoScalers** – kümmern sich bei Bedarf um das Auto-Scaling eines bestimmten MachineSets z. B. in einer bestimmten VZ.
- **ClusterAutoscaler** – die den MachineAutoScalers übergeordnete CR. Sie bestimmt, bis zu welchem Limit der Cluster insgesamt skaliert werden kann/darf.

Hinter all den APIs und CRs stecken – wie unter OpenShift üblich – natürlich Operatoren, je nach OpenShift-Version z. B. die Operatoren `machine-config-controller` und `machine-config-operator` im Namespace `openshift-machine-config-operator`, der `machine-api-controllers`-Operator im Namespace `openshift-machine-api` sowie weitere Helfer, wie z. B. der `machine-approver`-Operator im Namespace `openshift-cluster-machine-approver` und einige andere mehr.

16.12.1 MachineConfigs

Die *MachineConfig* ist aus OpenShift-Sicht zunächst auch nur eine Ressource im YAML-Format, jedoch eine spezielle. Nachfolgend sehen Sie eine Auflistung der vorhandenen MachineConfig-Objekte eines IPI-basierten vSphere-Setups, bereits mit Anpassungen für GPU-Nodes:

```
# oc get machineconfig
NAME                                        GENERATEDBYCONTROLLER                      IGNITIONVERSION   AGE
00-master                                   01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
00-worker                                   01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
01-master-container-runtime                 01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
01-master-kubelet                           01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
01-worker-container-runtime                 01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
01-worker-kubelet                           01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
51-gpu                                                                                 3.2.0             31d
99-master-generated-registries              01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
99-master-ssh                                                                          3.2.0             31d
99-worker-generated-registries              01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
99-worker-ssh                                                                          3.2.0             31d
rendered-gpu-6ce5da58d425409c962[…]         01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
rendered-master-b0c37a628e9e1cef[…]         01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
rendered-worker-9cbc9ced282960d9[…]         01f999aa710dd62d43bee8cf1e2ca6a226c7dce3   3.2.0             31d
```

Die MachineConfigs mit dem Präfix `00`, `01` (`master`/`worker`) etc. sind – je nach Version – in der Regel die eigentlichen Templates, aus denen dann während der Setups »gerenderte« Varian-

ten abgeleitet werden. Über die MachineConfigs wird die exakte Konfiguration der RHCOS-Nodes entsprechend der ihnen zugedachten Rolle festgelegt.

Der gleich erläuterte *MachineConfig-Operator* kümmert sich dann im Verbund mit assoziierten Subsystemen/-controllern in Operator-typischer Manier darum, dass der Ist-Stand immer möglichst mit dem gewünschten Soll-Stand übereinstimmt.

Die MachineConfig enthält Konfigurationen für jedes Paket und jede Konfigurationsdatei, die auf dem jeweiligen RHCOS-Pattern zum Einsatz kommt. Wer einen detaillierten Blick darauf werfen möchte: Das geht entweder über die GUI, indem Sie über Compute • MachineConfigs das Pattern auswählen, z. B. rendered-master-<ID> und dieses dann per Yaml betrachten. Auf der CLI ist es z. B. über die folgenden Varianten möglich:

```
# oc describe machineconfig rendered-master-<ID>

# oc get machineconfig rendered-master-<ID> -o yaml | less
```

16.12.2 MachineConfig-Operator

Wie bekannt ist in einem OpenShift-4-Cluster das zugrunde liegende RHCOS integraler Bestandteil der Plattform und wird als zum Cluster zugehörige Ressource über OpenShift verwaltet und bei Bedarf aktualisiert und skaliert. Im Hintergrund kümmert sich der *MachineConfig-Operator* (MCO) um diese Aufgabe auf Cluster-Ebene. Über ihn und die mit ihm assoziierten CRs werden die RHCOS-Nodes verwaltet und plattformtechnisch auf dem neuesten Stand gehalten. Über den MCO können Konfigurationssettings innerhalb der RHCOS-Templates für systemd, CRI-O, Kubelet, NetworkManager etc. vorgenommen werden. Zu diesem Zweck erstellt der MCO initial auf Basis eines Templates eine gerenderte/abgeleitete MachineConfig-Ressource (siehe den vorangegangenen Abschnitt), die alle Daten zur Konfiguration der RHCOS-Nodes enthält. Diese Konfiguration wird dann entsprechend des gesetzten Patterns (Master, Worker, Infra, GPU etc.) auf jeden neuen RHCOS-Node bei seiner Erzeugung angewendet. Betrachten wir die Details dazu.

16.12.3 Komponenten des MCO

Der MCO, der *MachineConfig-Operator*, ist einer der komplexesten OpenShift-spezifischen Operatoren und beinhaltet mehrere Subsysteme:

- **MachineConfig-Server** – Alle Nodes, die dem Cluster beitreten, müssen ihre Konfiguration zwangsläufig von einer Komponente erhalten, die innerhalb des OpenShift-Clusters ausgeführt wird. Der *MachineConfig-Server* stellt die Ignition-Endpoints zur Verfügung, über die sich alle neuen Nodes ihre MachineConfigs ziehen können. Jeder Node kann mittels entsprechender Parameter eine bestimmte MachineConfig – sofern verfügbar – anfordern, indem er die entsprechenden Parameter (respektive den Ignition-Pfad) abruft. Zum

Abrufen einer bestimmten Konfiguration (z. B. für die Konfiguration eines RHCOS-Nodes in der Rolle *Master*) kann der neue Node bzw. die neue Machine den Ignition-Endpoint */config/master* anfordern, ein Worker analog */config/worker*. Details zur Implementierung finden sich z. B. unter:

https://github.com/openshift/machine-config-operator/blob/master/docs/MachineConfigServer.md

- **MachineConfig-Controller** – Er kümmert sich darum, den Zustand, der in einer spezifischen MachineConfig hinterlegt ist, auf den konkreten RHCOS-Node anzuwenden. Subkomponenten sind dabei unter anderem die folgenden Controller:
  - zunächst der *Template-Controller* – er verwendet Templates, um auf deren Basis MachineConfigs zu generieren und die Machines auf den definierten Ist-Stand zu bringen und dort zu halten.
  - Des Weiteren wäre da noch der *Render-Controller* – er generiert das gewünschte MachineConfig Objekt, basierend auf dem *MachineConfigSelector*, der wiederum in einem *MachineConfigPool* hinterlegt ist.
  - Der *Update-Controller* führt in Verbindung mit dem MachineConfig-Daemon, der auf jedem RHCOS-Node ausgeführt wird, notwendige Updates/Änderungen durch, um Machines auf den in der MachineConfig hinterlegten Soll-Stand zu bringen.
  - Ergänzt wird der ohnehin schon komplexe Zoo durch den *KubeletConfig-Controller*: Er lauscht auf Änderungen der KubeletConfig-CR, um gewünschte Features zu (de-)aktivieren.

  Alle Implementierungsdetails finden sich unter: *https://github.com/openshift/machine-config-operator/blob/master/docs/MachineConfigController.md*
- **MachineConfig-Daemon** – Er läuft auf allen Cluster-Nodes als DaemonSet und kümmert sich als Exekutiv-Komponente um die praktische Ausführung von Node-/Machine-Updates gemäß der aktuell spezifizierten MachineConfig. Alle Implementierungsdetails finden sich unter:

  https://github.com/openshift/machine-config-operator/blob/master/docs/MachineConfigDaemon.md

16.12.4 MachineConfigPool

Der *MachineConfigPool* dient vereinfacht ausgedrückt als Adapter bzw. Vermittler, ähnlich der bekannten (Cluster-)RoleBindings. Er stellt über Selektoren die Verbindung zwischen den Nodes und den mit ihnen assoziierten MachineConfigs her. Das Ganze lässt sich anhand der bisherigen Vorbetrachtungen am einfachsten anhand der folgenden Abbildungen nachvollziehen.

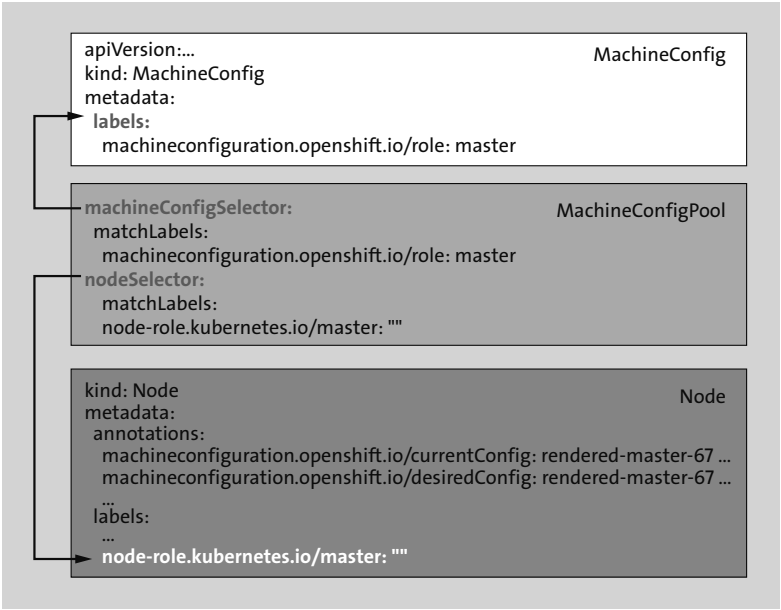


Abbildung 16.8 Beziehung zwischen MachineConfig(-Pool) und den Nodes

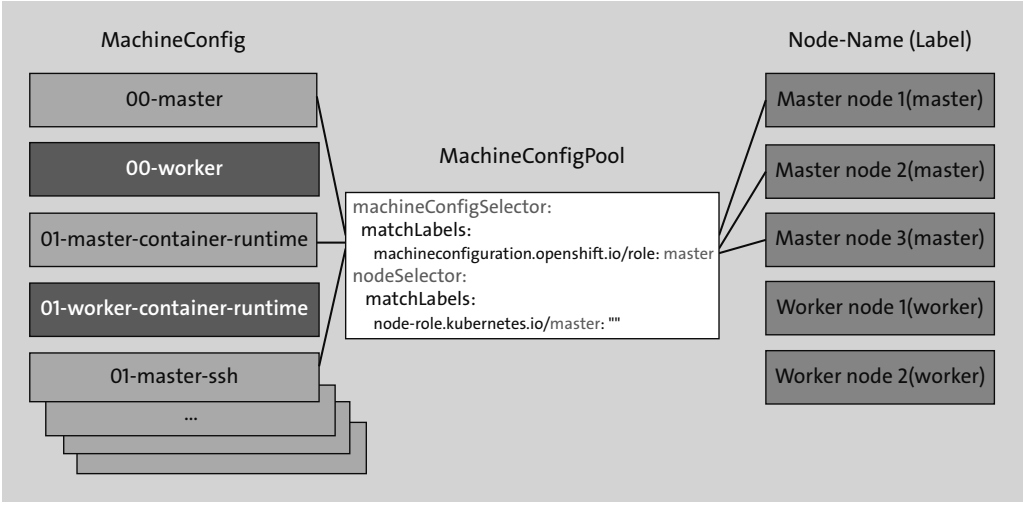


Abbildung 16.9 Selektion von MachineConfigs aus dem Pool für bestimmte Node-Patterns

16.12.5 Machines, MachineSets: manuelle Skalierung

Wie bereits in der Einführung erklärt: Das, was ReplicaSets für Pods sind, stellen MachineSets für Machines dar, nämlich eine Machine-spezifische HA-Metahülle für (Auto-)Skalierung. Nach der erfolgreichen Provisionierung eines Machine-Objekts über die Skalierung eines

MachineSets wird automatisch das auch unter Kubernetes bekannte Node-Objekt erzeugt, das in einer 1:1-Beziehung mit dem Machine-Objekt steht.

Im nächsten Beispiel sehen Sie eine exemplarische, manuelle Skalierung eines MachineSets in einem zuvor per IPI ausgerollten OpenShift-vSphere-Cluster, der zuvor nur mit drei schedulable Mastern ausgerollt wurde, sowie zwei bereits neu erzeugte (unskalierte) MachineSets für GPU-Nodes (Ausgabe gekürzt):

```
# oc get nodes,machines,machineset,mcp -A
```

```
NAME                          STATUS   ROLES                         AGE   VERSION
node/opns1-7rxpj-master-0     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
node/opns1-7rxpj-master-1     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
node/opns1-7rxpj-master-2     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
NAMESPACE              NAME                          PHASE     TYPE   REGION   ZONE   AGE
openshift-machine-api  machine/opns1-7rxpj-master-0 Running                           31d
openshift-machine-api  machine/opns1-7rxpj-master-1 Running                           31d
openshift-machine-api  machine/opns1-7rxpj-master-2 Running                           31d
NAMESPACE              NAME                                    DESIRED CURRENT READY AVAILABLE   AGE
openshift-machine-api machineset/opns1-7rxpj-gpu-pt-r5-vz2 0       0                         30d
openshift-machine-api machineset/opns1-7rxpj-vgpu-t4-vz2   0       0                         31d
openshift-machine-api machineset/opns1-7rxpj-worker        0       0                         31d
NAME                       CONFIG              UPDATED   UPDATING   DEGRADED   MACHINECOUNT
    READYMACHINECOUNT   UPDATEDMACHINECOUNT   DEGRADEDMACHINECOUNT      AGE
machineconfigpool/gpu      rendered-gpu-[…]     True      False      False      0
    0                   0                     0                    31d
machineconfigpool/master   rendered-master[…]   True      False      False      3
    3                   3                     0                    31d
machineconfigpool/worker   rendered-worker[…]   True      False      False      0
   0                   0                     0                    31d
```

```
# oc scale -n openshift-machine-api machineset opns1-7rxpj-worker --replicas 3
```

Nach der Skalierung:

```
# oc get nodes,machines,machineset,mcp -A
```

```
NAME                          STATUS   ROLES                         AGE   VERSION
node/opns1-7rxpj-master-0     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
node/opns1-7rxpj-master-1     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
node/opns1-7rxpj-master-2     Ready    control-plane,master,worker   31d   v1.25.11+c43ddea
node/opns1-7rxpj-worker-6t4qw   Ready    worker                        7m4s  v1.25.11+c43ddea
node/opns1-7rxpj-worker-g4mhs   Ready    worker                        7m4s  v1.25.11+c43ddea
node/opns1-7rxpj-worker-qd4fp   Ready    worker                        7m4s  v1.25.11+c43ddea
NAMESPACE              NAME                          PHASE     TYPE   REGION   ZONE   AGE
openshift-machine-api  machine/opns1-7rxpj-master-0 Running                           31d
openshift-machine-api  machine/opns1-7rxpj-master-1 Running                           31d
openshift-machine-api  machine/opns1-7rxpj-master-2 Running                           31d
NAMESPACE              NAME                                    DESIRED CURRENT READY AVAILABLE   AGE
[…]
openshift-machine-api machineset/opns1-7rxpj-worker        3       3       3     3           31d
NAME                       CONFIG              UPDATED   UPDATING   DEGRADED   MACHINECOUNT
    READYMACHINECOUNT   UPDATEDMACHINECOUNT   DEGRADEDMACHINECOUNT      AGE
machineconfigpool/gpu      rendered-gpu-[…]     True      False      False      0
```

```
    0                  0                    0                      31d
machineconfigpool/master   rendered-master[…]   True      False      False      3
    3                  3                    0                      31d
machineconfigpool/worker   rendered-worker[…]   True      False      False      3
   3                   3                    3                       31d
```

Achtung: Warten!

Der Scale-Out (egal ob manuell oder per Cluster-Autoscaler) kann je nach OpenShift-Version, RZ-Infrastruktur und Auslastung des Cloud-Providers gegebenenfalls 5 bis 10 Minuten und mehr in Anspruch nehmen. Das neue `Machine`-Objekt ist schnell erstellt, aber das `Node`-Objekt lässt in der Regel etwas auf sich warten. Bleiben Sie also geduldig.

Der Aufbau und das Management von MachineSets wird später noch in den Abschnitten über Infra- und GPU-Nodes betrachtet.

16.12.6 MachineConfigs nach dem Deployment anpassen

Wollen Sie die Konfiguration der RHCOS-Nodes nachträglich anpassen, ist das Verfahren prinzipiell relativ simpel, denn es muss lediglich eine neue MachineConfig mit den gewünschten Anpassungen den vorhandenen MachineConfigs hinzugefügt werden. Das neue MachineConfig-Template wird anschließend automatisch gerendert.

Siehe hierzu auch: *https://docs.openshift.com/container-platform/4.12/post_installation_configuration/machine-configuration-tasks.html*

Achtung: Backup

Wie üblich sollten solche tiefgreifenden Modifikationen, insbesondere dann, wenn Master-Nodes involviert sind, zunächst ausschließlich auf Testsystemen validiert werden. Vor der Ausführung in Nicht-Testsystemen sollten entsprechende (etcd-)Backups und/oder Snapshots der Master angelegt werden.

Handelt es sich z. B. um eine geänderte Konfigurationsdatei, wird diese über eine angepasste zusätzliche MachineConfig auf die RHCOS-Nodes angewendet. Wichtig dabei ist unter anderem, dass der Inhalt der Konfigurationsdatei zuvor Base64-kodiert und der Wert so in das entsprechende `source:`-Attribut eingetragen wird.

Achtung: /etc/resolv.conf der RHCOS-Nodes darf nicht angepasst werden!

Die Datei */etc/resolv.conf* wird über diverse Scripte des NetworkManagers generiert und aktuell gehalten. Änderungen gehen daher verloren. Siehe dazu:

- *https://access.redhat.com/solutions/5494101*
- *https://bugzilla.redhat.com/show_bug.cgi?id=1944805*

Nachfolgend sehen Sie die beispielhafte Vorgehensweise für eine Chrony-Konfiguration:

```
# cat << EOF | base64
  pool 0.rhel.pool.ntp.org iburst
  driftfile /var/lib/chrony/drift
  makestep 1.0 3
  rtcsync
  logdir /var/log/chrony
EOF
```

Die Ausgabe sieht z. B. wie folgt aus:

```
ICAgIIHNlcnZlciBj[...cut...]HJOY3N5bmMKICAgIGxvZ2RpciAvy9jaHJvbnkK
```

Dann folgt der Import der neuen `mc`, z. B. über das folgende Manifest (Achtung: Die `source`:-Zeile ist gekürzt):

```
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: master
  name: 99-masters-chrony-configuration
spec:
  config:
    ignition:
      config: {}
      security:
        tls: {}
      timeouts: {}
      version: 3.2.0
    networkd: {}
    passwd: {}
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-8;base64,ICAgIHNlcnZ[…cut…]bnkK
        mode: 420
        overwrite: true
        path: /etc/chrony.conf
  osImageURL: ""
```

Die zusätzlichen (in diesem Fall: Master-)MachineConfigs sorgen nach dem Import dafür, dass die Nodes sequenziell an die neue Konfiguration angepasst werden. Achtung: Der `mode`:-

Wert kann sowohl oktal (z. B. 0644, dabei darf die führende 0 nicht vergessen werden) als auch in seiner dezimalen, kanonischen Repräsentation (420) gesetzt werden. Siehe hierzu z. B.:

https://de.wikipedia.org/wiki/Oktalsystem#Umwandlung_von_Oktalzahlen_in_Dezimalzahlen

16.12.7 Defekte MachineConfigs entfernen und debuggen

Kommt es durch Konfigurationsfehler in einer der MachineConfigs (MC) zu einem korrupten oder fehlerhaften State der Nodes, kann dies über mehrere Verfahren untersucht werden. Die üblichen Verdächtigen wären z. B. die Logs des MachineConfig-Operators, die MachineConfigPools und MachineConfigs.

```
# oc logs -n openshift-machine-config-operator \
  machine-config-controller-887456589-j297k -f
```

oc describe mcp master (dort werfen Sie einen Blick in die `Status`-Sektion)

```
# oc get mcp

NAME     CONFIG                                             UPDATED   UPDATING   DEGRADED
   MACHINECOUNT     READYMACHINECOUNT   UPDATEDMACHINECOUNT   DEGRADEDMACHINECOUNT   AGE
master   rendered-master-3de5da9b9cc2f72a4aa3a9aed80813a9   False     True       True
   3                2                   2                     1                      31d
[…]
```

Eine Quick-and-dirty-Variante, um eine fehlerhafte MachineConfig zu resetten, kann eine Löschung der fehlerhaften MC und des (während des Updates assoziierten) MCP sein.

16.13 Cluster-Autoscaler und Machine-Autoscaler

Siehe zu allem Folgenden auch:

https://docs.openshift.com/container-platform/4.10/machine_management/applying-autoscaling.html

Nachdem wir nun in einem vereinfachten Überblick gesehen haben, wie OpenShift-Cluster auf der Basis von Machine*-spezifischen Objekten/Ressourcen und den assoziierten Operatoren skaliert werden können, geht es in diesem Abschnitt um die Auto-Skalierung von MachineSets und Clustern.

Von *Manuell* zu *Automatik*: Über Cluster-Autoscaler und assoziierte MachineSets können zusätzliche (GPU-)Worker-Nodes beim Überschreiten von spezifischen Thresholds (z. B. GPU/CPU/Memory-Load über den gesamten Cluster) automatisch ausgerollt werden. Zwingende Voraussetzung ist, dass die Machine-API, wie z. B. in jedem Cloud- oder auch in einem IPI-basierten vSphere-Setup, aktiv ist.

16.13.1 High-level Betrachtung

Zunächst ist es wichtig, die Beziehungen der involvierten Komponenten zueinander richtig zu erfassen. Beginnen wir mit den nun bereits bekannten *MachineSets*: Diese werden benötigt, um VM-Templates eines bestimmten Instanz-Typs in einer bestimmten VZ skalierbar zu machen. Das MachineSet selbst besitzt jedoch keine eingebaute Funktionalität für Auto-Scaling, dazu werden *Machine-Autoscaler* benötigt. In diesen CRs können jedoch keine Thresholds festgelegt werden, ab denen das jeweilige MachineSet zu skalieren beginnt: Die Machine-Autoscaler Ressource aktiviert lediglich die grundsätzliche Skalierbarkeit für das assoziierte MachineSet. Die übergeordnete Steuerung erfolgt über den gleich vorgestellten Cluster-Autoscaler.

Die erste, für Auto-Skalierbarkeit wichtige Assoziation wäre also zunächst die zwischen dem *MachineSet* und dem (dem betreffenden MachineSet zugeordneten) *Machine-Autoscaler*.

Da ein MachineSet jedoch in vielen Fällen, egal ob on-premise oder in der Cloud, meist nur für eine bestimmte Verfügbarkeitszone zuständig ist, würde sich so im Falle einer Skalierung schnell eine Schieflage bezogen auf die gesamte Cluster-Auslastung ergeben.

Daher werden *Cluster-Autoscaler*-Mechanismen und assoziierte Objekte/Ressourcen benötigt, welche den/die OpenShift-Cluster ganzheitlich – d. h. mit allen involvierten MachineSets und VZs – erfassen und möglichst symmetrisch skalieren können.

Dies wäre dann auch die zweite wichtige Assoziation: der *Cluster-Autoscaler* und die von ihm betreuten und gesteuerten *Machine-Autoscaler*.

16.13.2 Machine-Autoscaler

Betrachten wir zunächst beispielhaft ein generisches Machine-Autoscaler-Objekt, noch ohne konkreten Bezug zum späteren Setup. Im Prinzip ist der Aufbau extrem simpel: Es werden im einfachsten Fall lediglich das zu betreuende MachineSet (`spec.scaleTargetRef.name`) und die dafür zulässigen Thresholds (`spec.min|maxReplicas`) angegeben:

```
apiVersion: autoscaling.openshift.io/v1beta1
kind: MachineAutoscaler
metadata:
  name: opns2-gpuworker-machineautoscaler-a100-vz2
  namespace: openshift-machine-api
spec:
  minReplicas: 1
  maxReplicas: 10
  scaleTargetRef:
    apiVersion: machine.openshift.io/v1beta1
    kind: MachineSet
    name: opns2-gpuworker-a100-vz2
```

Achtung: Minimale Repliken und Kosten

Für IPI-basierte OpenShift-Installationen kann (nur) unter AWS, Azure, GCP, vSphere oder OpenStack die minimale Anzahl der Repliken auf 0 gesetzt werden. Dies ist wichtig, um im Idle-Zustand Energie und (Cloud-)Kosten zu sparen.

Wie aus dem oberen Listing (bei dem keine Attribute weggelassen wurden) ersichtlich sein sollte, existieren jedoch auf der Ebene des Machine-Autoscalers keine einstellbaren Werte für CPU/GPU/Memory-Pressure bzw. -Load. Hierzu wird der übergeordnete *Cluster-Autoscaler* benötigt.

16.13.3 Cluster-Autoscaler

Kommen wir zum Auto-Scaling-Chef: Der Cluster-Autoscaler orchestriert auf höchster Ebene die Machine-Autoscaler aller VZ/MachineSets, und zwar auf allen Plattformen und Infrastrukturen, auf denen OpenShift verfügbar ist (Bare Metal logischerweise ausgenommen).

Betrachten wir den konzeptionellen Aufbau der Ressource und anschließend die Arbeits-/Vorgehensweise. Eine exemplarische Autoscaler-Konfiguration könnte sich in etwa wie folgt darstellen:

```
apiVersion: autoscaling.openshift.io/v1
kind: ClusterAutoscaler
metadata:
  name: default
spec:
  podPriorityThreshold: -10
  resourceLimits:
    maxNodesTotal: 24
    cores:
      min: 12
      max: 384
    memory:
      min: 128
      max: 2048
    gpus:
      - type: nvidia.com/gpu
        min: 0
        max: 64
  scaleDown:
    enabled: true
    delayAfterAdd: 10m
    delayAfterDelete: 5m
```

```
delayAfterFailure: 30s
unneededTime: 5m
utilizationThreshold: "0.6"
```

Die Arbeitsweise

Der Cluster-Autoscaler berechnet alle 10 Sekunden die Gesamtauslastung der CPU-, GPU- und Memory-Ressourcen im gesamten Cluster, d. h., es werden die Daten aller Nodes (inklusive der Master) berücksichtigt, auch wenn Letztere nicht autoskaliert werden können.

Sobald es Pods gibt, die vom Scheduler aufgrund unzureichender Ressourcen nicht mehr einem der aktiven (GPU-)Worker-Nodes zugeteilt werden können (hier kommen zudem noch Pod-Priorities ins Spiel, dies sprengt jedoch den Scope dieses Buches), beginnt der Cluster-Autoscaler damit, neue (GPU-)Nodes zu skalieren. Sobald die Thresholds wieder unterschritten werden, erfolgt automatisch der Cool-Down bzw. das Herunterskalieren der involvierten MachineSets. Dabei kommen nun einige Thresholds der Cluster-Autoscaler-Ressource ins Spiel.

16.13.4 Thresholds

Die Skalierungsaktionen des Cluster-Autoscalers erfolgen stets unter Berücksichtigung verschiedener Grenzwerte. Betrachten wir auszugsweise einige der elementar wichtigen (`spec.resourceLimits` und `spec.scaleDown`):

- **`maxNodesTotal`** – die maximal zulässige Gesamtanzahl der Nodes im Cluster (inklusive Master/Infra), nicht nur die Zahl der vom Autoscaler kontrollierten und skalierbaren Nodes
- **`cores(.min|.max)`** – minimale und maximale Anzahl der CPU-Cores, die im gesamten Cluster verfügbar sein müssen bzw. dürfen
- **`memory(.min|.max)`** – minimale und maximale Memory-Größe, die im gesamten Cluster verfügbar sein muss bzw. darf
- **`gpu.type(.min|.max)`** – Art der GPU (z. B. NVIDIA/AMD) und ihre minimal/maximal zulässige Anzahl im gesamten Cluster
- **`scaleDown.*`** – In dieser Sektion kann über verschiedene, z. T. optionale Subattribute das Verhalten des Cluster-Autoscalers beim Herunterskalieren gesteuert werden, z. B. mit:
  - `enabled` – Das Herunterskalieren ist aktiviert.
  - `delayAfterAdd` – Wartezeit, bis ein Node herunterskaliert werden darf, nachdem er hinzugefügt wurde
  - `delayAfterDelete` – Wartezeit bis zur nächsten Node-Löschung, nachdem ein Node entfernt wurde)

– `utilizationThreshold` – String für die Auslastung zwischen 0 und 1, z. B. `"0.5"`. Der Auslastungsgrad eines Nodes entspricht dabei der Summe der angeforderten Ressourcen dividiert durch die dem Node zugewiesenen Ressourcen. Wird kein Wert angegeben, verwendet der Cluster-Autoscaler einen Standardwert von `"0.5"`, der einer Auslastung von 50 % entspricht.

Würden wir das maximale CPU-Limit des Clusters auf 64 Cores einstellen und den Cluster-Autoscaler so konfigurieren, dass (GPU-)Worker-Nodes mit minimal je 4 Cores erstellt werden, könnte der Cluster (wenn die Master-Nodes bereits 32 Cores okkupieren) mit den verbleibenden 32 Cores 8 weitere Nodes hinzufügen.

16.13.5 Zu beachtende Punkte

Nachstehend folgen noch einige wichtige konzeptionelle Anmerkungen zum Cluster-Autoscaler in Form einer kurzen Auflistung.

ScaleDown und Auto-Removal von Nodes

Der Cluster-Autoscaler betrachtet einen Node als *Ready-to-remove*, wenn z. B. die folgenden Bedingungen zutreffen:

- ScaleDown ist nicht deaktiviert.
- Die Auslastung des zu entfernenden Nodes liegt unter dem angegebenen bzw. dem Default-Threshold.
- Der Cluster-Autoscaler kann alle Pods, die auf dem zu entfernenden Node ausgeführt werden, auf die verbleibenden Nodes umverteilen.

Pods, die ein Node-Removal bzw. ScaleDown verhindern

Wenn die folgenden Arten von Pods auf einem Knoten vorhanden sind, entfernt der Cluster-Autoscaler den Node nicht aus dem Cluster-Verbund:

- Pods mit *PodDisruptionBudgets* (PDB, siehe *https://kubernetes.io/docs/tasks/run-application/configure-pdb/* und Abschnitt 9.7)
- System-relevante Pods, die standardmäßig nicht auf dem Node ausgeführt werden
- System-relevante Pods, die keine PDB oder eine zu restriktive PDB haben
- Pods, die von einer Custom-Ressource (z. B. per Operator) mit eigenständigen Controller/Reconciler-Mechanismen erzeugt wurden
- Pods mit lokalem Speicher auf dem jeweiligen Node
- Pods, die aufgrund fehlender Ressourcen, inkompatibler Node-Selektoren oder (Anti)-Affinities nicht auf einen anderen Node scheduled werden können
- Pods mit der Annotation `cluster-autoscaler.kubernetes.io/safe-to-evict: "false"`

Inhalt

TEIL II Kubernetes-Architektur, Core-Concepts, Workloads und Day 1 Operations

3 Kubernetes 89

9 Kubernetes-Cluster: Day 2 Operations – Pod-Lifecycle, De-Scheduling, Tenancy und Limits 411

TEIL VIII Vollautomatisierte CI/CD-GitOps-Pipelines

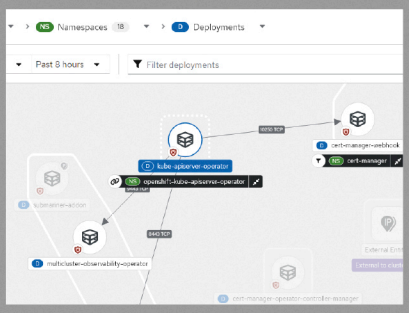
21 Day 3 Operations: CI/CD-Pipelines und GitOps 1005

Skalierbare Container-Infrastrukturen

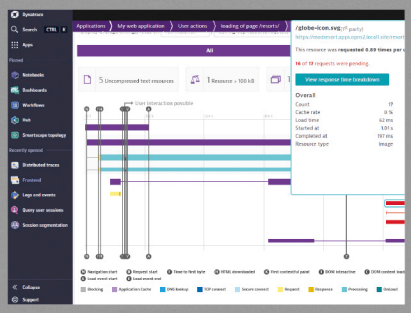
»Die Referenz für DevOps-Teams und Admins«

Container-Cluster umfassend und kompetent erklärt

Dieses Handbuch zeigt Ihnen, wie Sie Kubernetes-basierte Container-Cluster durchdacht planen, auf unterschiedlichen Plattformen installieren und hochautomatisiert betreiben. Mit den richtigen Tools und fundierten Architekturentscheidungen sorgen Sie dafür, dass Ihre Kubernetes- oder OpenShift-Infrastruktur performant und ausfallsicher läuft.



Strukturen verstehen



Tools evaluieren und anwenden

```
apiVersion: config.openshift.io/v1
kind: OAuth
metadata:
  name: cluster
spec:
  identityProviders:
  - ldap:
      attributes:
        email: []
        id:
        - dn
        name:
        - cn
        preferredUsername:
```

Getestete Konfigurationen

Strategie, Theorie und Praxis

Verstehen Sie Container-Technologien und lernen Sie ihre Einsatzmöglichkeiten und Vorteile kennen. Dabei stehen Automation und Effizienz für den zeitgemäßen Unternehmenseinsatz im Zentrum.

Container-Orchestrierung

Ob OpenShift oder Kubernetes, ob managed oder self-managed, ob On-Prem, in der Cloud oder hybrid: Wie Ihre Anforderungen auch sind, Sie werden von erprobtem Praxiswissen profitieren, mit dem Sie einen kosteneffizienten und störungsfreien Betrieb sicherstellen.

Maximale Automation und Effizienz

Praxisszenarien, Diagramme, Codebeispiele und Best Practices helfen Ihnen bei der Konzeption, Installation und Wartung. Stellen Sie auf Knopfdruck operatorgestützte, vollautomatisierte, widerstandsfähige und skalierbare Multi-Cluster-Systeme auf jeder Plattform bereit.

Alle Beispielkonfigurationen zum Download

Linux-Enterprise-Experte **Oliver Liebel** ist Partner von Red Hat und SUSE und arbeitet eng mit NVIDIA zusammen. Er ist seit 28 Jahren als Berater und Systemarchitekt für große Unternehmenskunden tätig. Seine Spezialgebiete sind skalierbare und GPU-beschleunigte Container-Cluster auf Kubernetes-/OpenShift-Basis, High-Availability, Verzeichnisdienste und Software-Defined Storage.

Aus dem Inhalt

Rheinwerk Computing

€ 79,90 [D] € 82,20 [A]

9 783836 297530

ISBN 978-3-8362-9753-0

Netzwerke & Datenbanken