Coding Continuous Delivery: CIOps vs. GitOps mit Jenkins

Johannes Schnatterer und Daniel Huchthausen, Cloudogu GmbH

Continuous Delivery (CD) hat sich im Umfeld agiler Softwareentwicklung als adäquates Vorgehen erwiesen, qualitativ hochwertige Software in kurzen Zyklen zuverlässig und wiederholbar zu veröffentlichen. Der Einsatz von Containern und Cloud, beispielsweise auf Plattformen wie Kubernetes (K8s), bietet viele Möglichkeiten, um CD-Prozesse robuster und einfacher zu gestalten. Eine solche Möglichkeit ist GitOps. In diesem Artikel werden die Unterschiede zwischen klassischen CD-Pipelines (CIOps) und GitOps-Prozessen anhand von konkreten Beispielen aufgezeigt.

Die Automatisierung bei CD erfolgt mittels Pipelines in Continuous-Integration-(CI)-Servern wie Jenkins. Dabei gibt es zwei Anwendungsfälle, bei denen der Einsatz von Containern Vorteile bietet:

1. Bei der Ausführung der Pipeline können Tools ohne weitere Konfiguration im CI-Server in Containern ausgeführt werden. In Containern können Anwendungen außerdem zum Test isoliert ausgeführt werden, um beispielsweise Port-Konflikte zu vermeiden
2. Container-Images sind ein standardisiertes Artefakt, in das die Anwendung durch die Pipeline verpackt wird

Diese Images können auf vielen verschiedenen Betriebsumgebungen deployt werden, da mittlerweile sowohl Docker-Container und Images als auch das API der Registry durch die Open Container Initiative (OCI) standardisiert sind. In den letzten Jahren haben sich besonders im DevOps-Umfeld Container Orchestration Platforms als flexibles Mittel für Deployments von OCI-Images erwiesen. Dabei hat sich K8s als De-facto-Standard herauskristallisiert, weshalb sich dieser Artikel exemplarisch auf das Deployment auf K8s beschränkt.

Klassische CD-Pipelines – CIOps

Bei einer „klassisch" umgesetzten CD-Pipeline führt der CI-Server aktiv das Deployment in die Betriebsumgebung durch *(siehe Abbildung 1)*. Zur Abgrenzung von später entstandenen Methoden wie GitOps (siehe unten) wird dieses Vorgehen auch als „CIOps" bezeichnet. Manchmal wird es dabei als Antipattern dargestellt *[1]*. Das Verfahren hat sich allerdings jahrelang in der Praxis bewährt und es spricht generell nichts dagegen, es weiterhin zu verwenden.

Eine einfach umsetzbare Logik zur Automatisierung des Deployments in einer CIOps-Pipeline mit Staging- und Produktionsumgebungen ist die Verwendung von Branches in Git.

Dafür nutzen viele Teams Feature Branches oder Git Flow, in denen der integrierte Entwicklungsstand auf dem Develop-Branch zusammenfließt und der Main- (oder Master-) Branch die produktiven Versionen enthält. Darauf kann einfach eine CD-Strategie aufgebaut werden: Jeder Push auf Develop führt zu einem Deployment auf die Staging-Umgebung, jeder Push auf Main geht in Produktion. So steht stets die letzte integrierte Version auf Staging für funktionale oder manuelle Tests bereit. Durch einen Pull Request (PR) oder Merge auf Main wird dann das Deployment in Produktion angestoßen. Zudem ist ein Deployment pro Feature Branch denkbar.

Der Nachteil dieses Vorgehens ist, dass für jedes Deployment ein Build im CI-Server durchlaufen werden muss. Das macht den Prozess langsamer. Denn generell soll ja dasselbe Artefakt auf allen Stages deployt werden. Es wäre also gar kein neuer Build, Test oder gar Versionsname nötig.

Eine solche Deployment-Logik lässt sich mit Jenkins Pipelines einfach realisieren, da der Branch-Name in Multibranch-Builds aus dem Environment abgefragt werden kann. Ein ausführliches Beispiel, das eine vollständige Implementierung mit Jenkins zeigt, ist in Java aktuell 04/2018 beschrieben *[2]*, das vollständige Jenkinsfile kann bei GitHub eingesehen werden *[3]* (Branch „11").

GitOps vs. CIOps

Mittlerweile gibt es im K8s-Umfeld eine Alternative zu CIOps: GitOps. Hier prüft eine im K8s-Cluster laufende, Cloud-native Anwendung (der „GitOps-Operator") kontinuierlich den tatsächlichen Zustand des Clusters gegen den Wunschzustand, der in einem Git-Repository beschrieben ist. Deployments werden durch einen Push auf dieses Repository, beispielsweise durch die Annahme eines PR, ausgelöst *(siehe Abbildung 2)*. Durch GitOps ergeben sich einige Vorteile:

- Weniger schreibender Zugriff von außen auf den Cluster nötig, da der GitOps-Operator Deployments von innerhalb des Clusters durchführt
- Keine Credentials im CI Server, da kein Zugriff auf den Cluster benötigt wird
- Infrastructure as Code (IaC) bietet Vorteile für Auditierung und Reproduzierbarkeit. Außerdem sind Cluster und Git automatisch synchronisiert
- Der Zugriff auf Git ist oft organisatorisch einfacher als der auf den API-Server. Möglicherweise entfällt die Notwendigkeit einer Firewall-Freischaltung

Rolle des CI-Servers bei GitOps

Für das Deployment von Third-Party-Anwendungen (die nicht selbst entwickelt werden), ist ein CI-Server nicht mehr unbedingt nötig. Bei selbst geschriebenen Anwendungen sind nach wie vor Build, Tests, etc. auszuführen. Dies übernimmt weiterhin der CI-Server, genauso wie das Pushen des Images in eine Registry *(siehe Abbildung 3)*. Außerdem kann der CI-Server eingesetzt werden, um einige der Herausforderungen von GitOps zu lösen:

- Lokale Entwicklung mit GitOps ist weniger effizient (Betrieb des Operators, Deployment und Debugging sind umständlicher).
- Manuelle Implementierung von Staging kann umständlich sein (für jede Stage muss ein PR erstellt werden).
- Oft wird bei GitOps der Infrastruktur-Code in einem zentralen Repository gesammelt. Dies bietet den Vorteil, dass der gesamte Zustand des Clusters an einer Stelle gespeichert ist. Der Nachteil: Die Trennung von Anwendungs- und Infrastruktur-Code auf zwei Repositories ist aufwendiger zu warten, beispielsweise bei Review, Versionierung und lokaler Entwicklung.

Durch Unterstützung des CI-Servers kann erreicht werden, dass beides im Repository der Anwendung (im Folgenden als App-Repository bezeichnet) verbleibt. Der CI-Server pusht dann den Infrastruktur-Code in das GitOps-Repository *(siehe Abbildung 4)*.

GitOps am Beispiel

Die Implementierung eines GitOps-Flows, wie in *Abbildung 4* gezeigt, klingt zunächst sehr einfach zu implementieren. Doch wie so oft liegt auch hier der Teufel im Detail: Zum einen warten Herausforderungen bei der Implementierung, zum anderen fallen schnell weitere Punkte auf, die durch die Pipeline automatisiert werden können. So kann am Ende die zunächst einfach erscheinende Implementierung einer solchen Pipeline doch aufwendig werden.

Die größte Herausforderung ist, dass mehrere gleichzeitige Builds laufen können, die in dasselbe GitOps-Repository schreiben. Eine zuverlässige Fehlerbehandlung solcher Concurrency-Issues führt zu

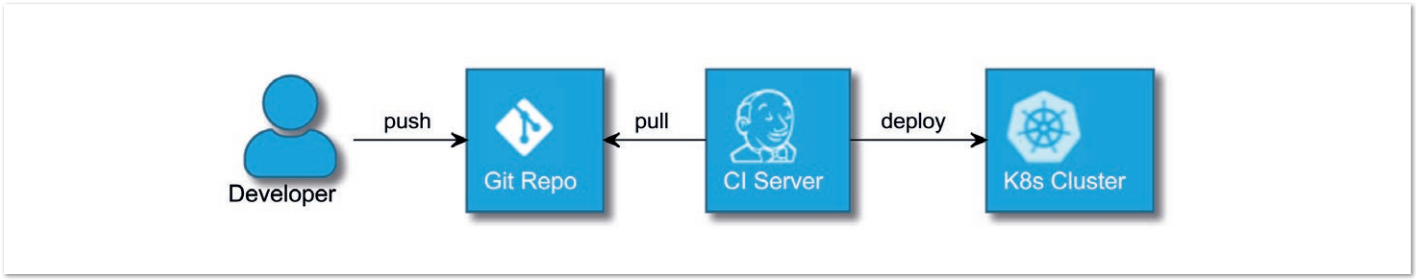

Abbildung 1: „Klassische" CD-Pipeline – CIOps (© Cloudogu)

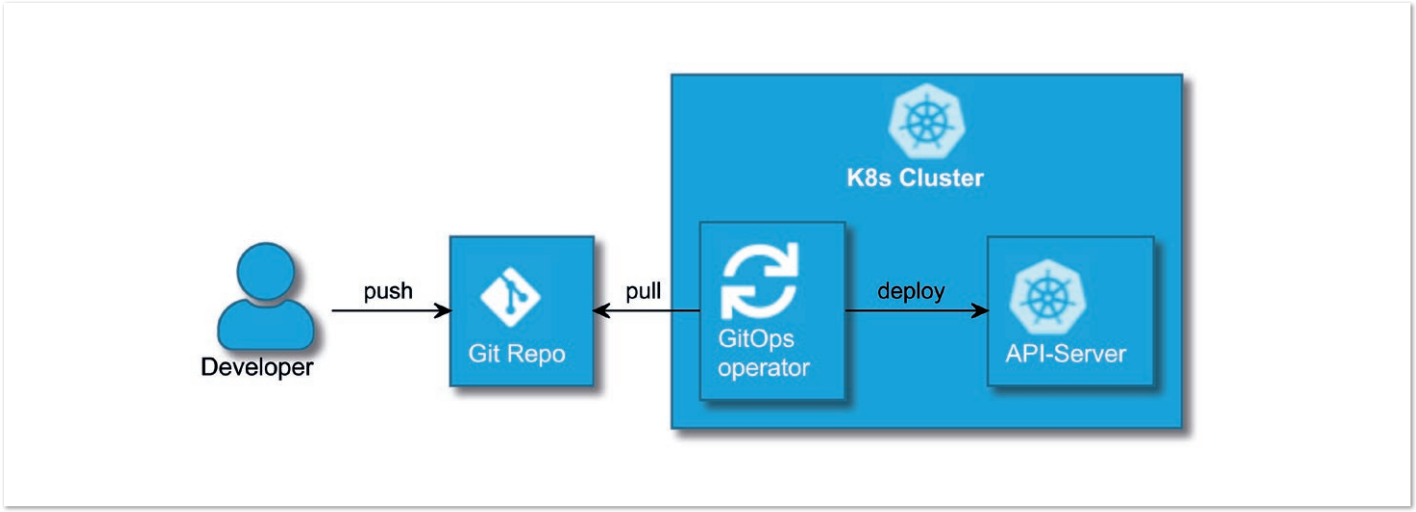

Abbildung 2: Einfaches Deployment mittels GitOps (© Cloudogu)

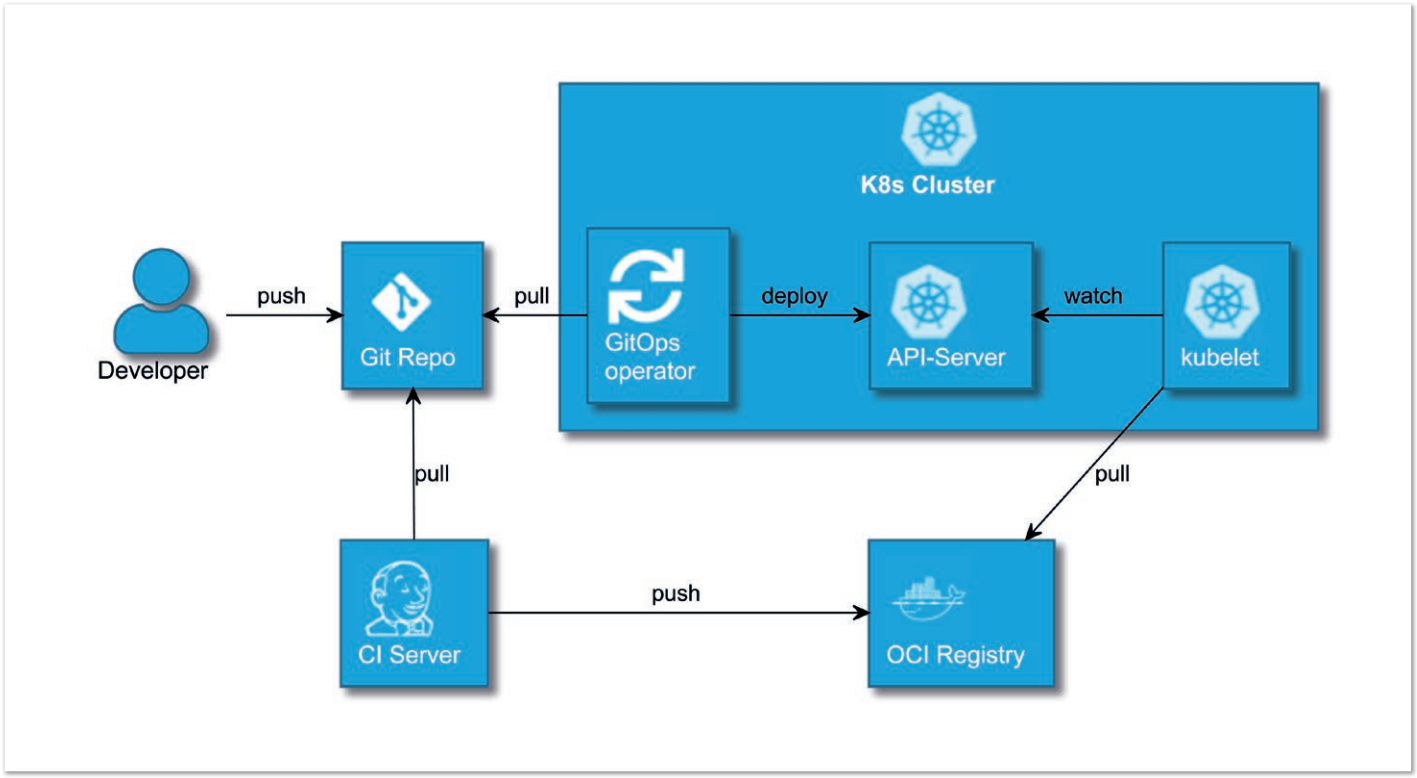

Abbildung 3: GitOps-Deployment von selbst entwickelten Images (© Cloudogu)

überraschender Komplexität. Wenn die Pipeline dann einmal grundlegend funktioniert, kann der Entwicklungsprozess durch weitere Automatisierung effizienter gestaltet werden. Beispiele für solche Erweiterung folgen später.

Konkrete Beispiele für GitOps-Flows bietet der GitOps-Playground *[4]*, mit dem in einem lokal ausführbaren Cluster verschiedene GitOps-Operator, wie Flux (GitOps Toolkit) und ArgoCD (GitOps Engine) im Zusammenspiel mit Jenkins ausprobiert werden können.

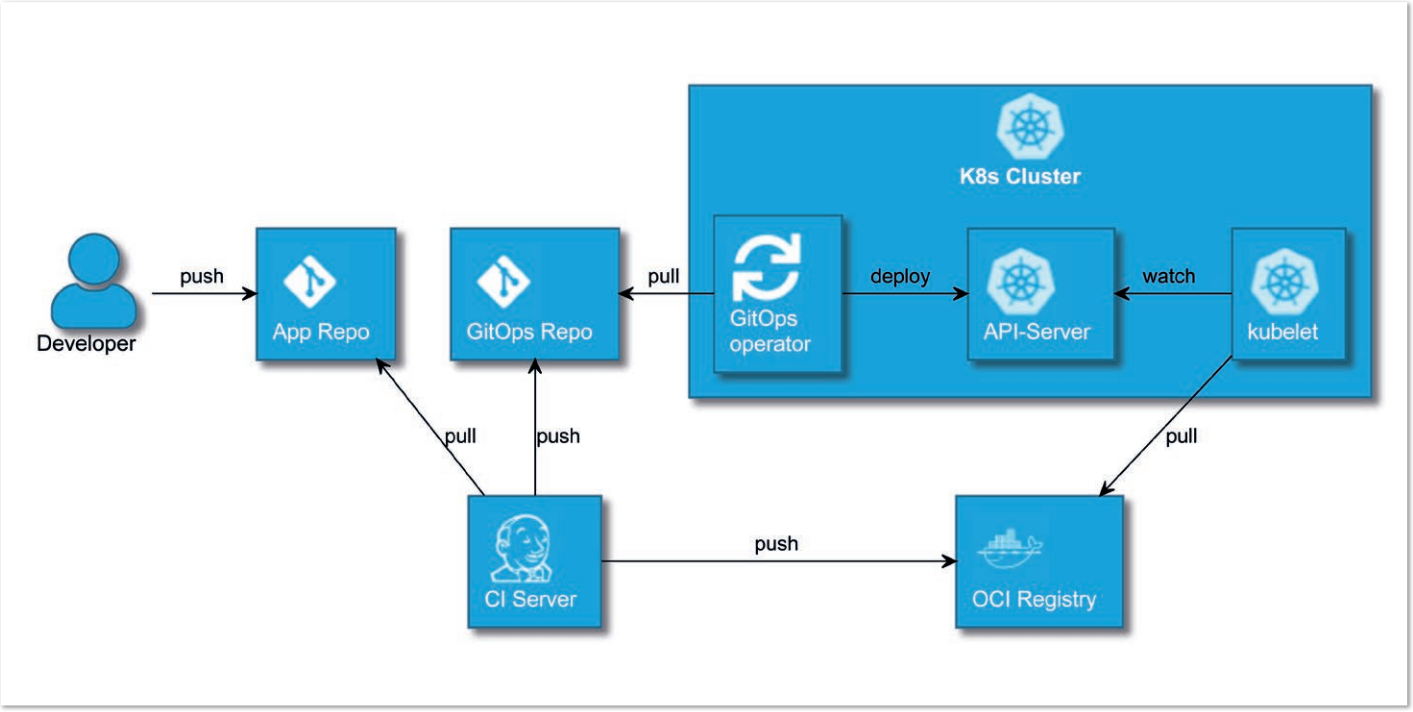

Abbildung 4: Deployment mit App-Repository und GitOps-Repository (© Cloudogu)

Darin enthalten ist auch eine Jenkins Pipeline (Datei „applications/petclinic/fluxv1/plain-k8s/Jenkinsfile"). Diese ist ähnlich aufgebaut wie das erwähnte CIOps-Beispiel. In zwei Punkten unterscheiden sich die Pipelines jedoch grundlegend:

- Die YAMLs werden in das GitOps-Repository gepusht, statt auf den Cluster angewandt
- Die verschiedenen Stages werden komplett im GitOps-Repository realisiert (nicht im App-Repository). Dadurch ist hier kein CI-Server mehr notwendig. Der GoLive ist schneller

Die grundlegende Ordnerstruktur des GitOps-Repository zeigt *Abbildung 5*: Auf oberster Ebene gibt es einen Ordner pro Stage. Darin enthalten ist je ein Ordner pro Applikation. Das Deployment unterscheidet sich dann leicht, je nachdem wie Stages gelöst sind:

- Staging Namespaces im selben Cluster (siehe GitOps-Playground): Es gibt nur einen GitOps-Operator und dieser deployt alle im GitOps befindlichen K8s-Ressourcen in den Cluster. Dabei muss jeweils auf die Angabe des richtigen Namespaces in den K8s-Ressourcen geachtet werden. Auf diese Weise ist es im GitOps-Playground gelöst
- Alternativ können auch Staging Cluster verwendet werden. In diese wird der GitOps-Operator so konfiguriert, dass er entsprechend seiner Stage alles deployt, was im jeweiligen Ordner liegt. Beispiel: Der GitOps-Operator im Staging-Cluster deployt nur K8s-Ressourcen aus dem „Staging"-Ordner

Der Ablauf der erwähnten Pipeline aus dem GitOps-Playground ist wie folgt:

1. Push auf den Main Branch des App-Repository löst den GitOps-Prozess aus
2. GitOps-Repository klonen
3. Staging: Image-Version in Deployment-YAML aktualisieren, in den Applikationsordner der Stage kopieren und auf den Main Branch des GitOps-Repositories pushen.
4. Produktion: Wie in Schritt 3, nur dass die Änderungen im Ordner "Production" gemacht werden und auf einen speziell für die Anwendung erstellten Branch im GitOps-Repo gepusht werden. Anschließend wird ein PR auf den Main Branch geöffnet.

Nachdem diese Pipeline durchlaufen ist, wird die Anwendung vom GitOps-Operator zum Review auf Staging deployt. Darüber hinaus existiert ein PR, der bei Annahme direkt (ohne CI-Server) zu einem Deployment in Produktion führt.

Die oben beschrieben Concurrency-Issues können zwischen dem Klonen des Repository und den Pushes auftreten: Wenn zwischenzeitlich das remote Repository geändert wurde, scheitert der Push und damit der Build. Das macht die Entwicklung komplizierter und langsamer. Die Pipeline im Beispiel hat daher einen einfachen Retry-Mechanismus. Scheitert der Push, erfolgt ein Pull und ein erneuter Push. Diese Lösung ist nicht perfekt, da bei Konflikten der Build trotzdem scheitert. Unter gewissen Umständen kann sogar eine Inkonsistenz entstehen: Beim Pull könnte ein Fast Forward Merge gemacht werden, der die Änderungen aus dem Build mit denen aus einem anderen „vermischt". Hier wäre es also sicherer, nach dem Pull nicht einfach zu pushen, sondern einen Reset auf den Remote-Stand zu machen und die Änderungen erneut durchzuführen.

An anderer Stelle zeigt sich die Pipeline schon recht ausgefeilt. So werden die Commits, die der Job am GitOps-Repository vorgenommen hat, für mehr Transparenz in der Jenkins Job Description angezeigt. Für ein effizienteres Review des PR wird Folgendes in die Commit Message im GitOps-Repository geschrieben (*Abbildung 6* zeigt dies am Beispiel mit SCM-Manager):

- Autor des ursprünglichen Commit im App-Repository
- Autor wird beibehalten, aber „Jenkins" wird Committer. Damit

ist klar, von wem diese Änderung stammt, aber auch, dass der Commit automatisiert erstellt wurde
- Link auf die Issue-ID, geparst aus der ursprünglichen Commit Message. Damit ist eine direkte Verbindung zur Fachlichkeit im Issue Tracker gegeben
- Link auf den ursprünglichen Commit im App-Repository. Damit lässt sich mit einem Klick auf den Source Code der Anwendung wechseln
- Ein Staging-Commit wird jeweils markiert (nicht in der Abbildung zu sehen)

Derzeit sind im GitOps-Playground noch weitere Features in Arbeit, die den Prozess effizienter gestalten. Möglicherweise sind diese zum Zeitpunkt der Veröffentlichung des Artikels schon verfügbar:

- Fail Early: statische YAML-Analyse durch den CI-Server. Damit möglichst selten der aufwendige Weg der Fehlersuche im Log des GitOps-Operators gegangen werden muss, werden die YAML-Files auf syntaktische Korrektheit geprüft, beispielsweise mit dem Tool „yamlint". Ein weiterer Schritt ist das Prüfen der K8s-Ressourcen gegen das K8s-Schema. Dies kann mit dem Tool „kubeval" erfolgen. Bei Helm-Charts mit eigenem Schema könnte auch gegen dieses geprüft werden (mittels „helm lint")
- Automatisch erstelle PRs können durch weitere Informationen angereichert werden. Beispielsweise kann ein bereits bestehender PR ergänzt werden, wenn weitere Commits darauf gemacht werden. Zudem kann ein Link auf den erstellenden Jenkins Job in den Kommentaren erzeugt werden
- Ein Weg, um Konfigurationsdateien oder Scripts in den Cluster zu bringen, ist es, diese als inline YAML zu verpacken, beispielsweise in eine Config Map. Dieses Verfahren hat den Nachteil, dass es in dieser Form kein Syntax-Highlighting oder Linting gibt. Dadurch kommt es häufiger zu vermeidbaren Fehlern oder ineffizientem „hin- und herkopieren". Dieser Nachteil kann mittels Automatisierung behoben werden: Der CI-Server übernimmt das Verpacken einer „echten" Datei in YAML. Dadurch besteht bei der Entwicklung die Möglichkeit, auf dieser Datei zu arbeiten und dort das gewohnte Highlighting und Linting zu bekommen
- Häufig besteht der Bedarf vor Abschluss der Entwicklung eines Features, dieses in der Staging-Umgebung manuell zu testen. Mithilfe der Pipeline lässt sich dies ohne (verfrühten) Merge auf den Main-Branch und ohne PR für Produktion realisieren. Dies kann durch Build-Parameter in Jenkins implementiert werden. Ein solcher Parameter kann beim manuellen Anstoßen eines Builds gesetzt werden. Die Pipeline kann auf den Parameter reagieren, indem ins Staging gepusht, aber kein PR für Produktion erstellt wird
- Eine größere Anzahl Stages kann durch weitere Branches und PRs realisiert werden

```
├── production
│   └── application
│       ├── deployment.yaml
│       └── ...
└── staging
    └── application
        ├── deployment.yaml
        └── ...
```

Abbildung 5: Mögliche Ordnerstruktur eines GitOps-Repositories (© Cloudogu)

Templating-Tools

Die vorgestellten Beispiele zu CIOps und GitOps zeigen, wie einfache K8s-Ressourcen auf den Cluster mit der jeweiligen Methode angewendet werden können. Nachteil dabei ist, dass die K8s-Ressourcen komplett redundant für jede Stage gespeichert werden müssen. In der Praxis werden daher oft Templating-Tools eingesetzt, die eine Parametrisierung einer einzigen Quelle (ohne Redundanz) ermöglichen. Helm, der offizielle Package Manager für K8s, ist eine gängige Lösung. Mit Helm können nicht nur Third-Party-Packages deployt werden. Seine Templating-Funktion kann auch für die lokale Entwicklung genutzt werden.

Für die lokale Entwicklung gibt es einige Alternativen zu Helm, wie das „Template-freie" Tool Kustomize, das mit sogenannten Overlays arbeitet, die mit dem Patch-Mechanismus auf eine Basisdatei angewendet werden.

Bei CIOps lassen sich Templating-Tools relativ einfach anwenden. Die Tools stehen als Kommandozeilenwerkzeug zur Verfügung, das in der Pipeline aufrufbar ist. Ein Beispiel dafür steht bei GitHub zur Verfügung *[3]* (Branch „12"). Hier wird das Helm-Binary als Container ausgeführt, sodass keine weitere Konfiguration seitens des Jenkins-Masters erforderlich ist. Einige weitere wertvolle Erkenntnisse aus der Praxis:

- Durch die Nutzung von `helm upgrade --install` muss nicht aufwendig zwischen Erstinstallation und Upgrade unterschieden werden
- Die in allen Helm-Paketen (sogenannten Charts) vorgeschriebene „values.yaml" beschreibt Standardwerte, eine weitere `values-`Datei pro Stage setzt die jeweils spezifischen Werte. Diese Datei muss dem Helm-Befehl per `-value`-Parameter übergeben werden, die Standard „values.yaml" wird implizit immer angezogen

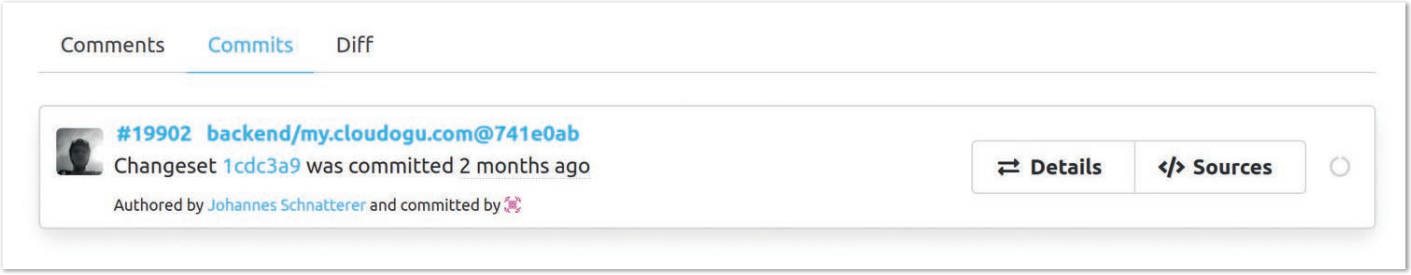

Abbildung 6: Beispiel eines vom CI-Server erstellten Commit im GitOps-Repository (© Cloudogu)

- Name und Version des Image können bequem mittels Parameter gesetzt werden, beispielsweise `--set 'image.tag=...`

Bei der Verwendung von Templating-Tools mit GitOps wartet zu Beginn eine große Herausforderung: Wie kann der imperative Aufruf (beispielsweise `helm upgrade`) in eine deklarative Form gebracht werden, die im GitOps-Repository abgelegt werden kann? Die Lösung: weitere Operatoren in K8s. Für die weit verbreiteten Tools Helm und Kustomize existieren solche Operatoren bereits, für anderen Templating-Tools nicht unbedingt. Auch hier gibt es ein Praxisbeispiel im GitOps-Playground *[4]* (Datei: applications/nginx/fluxv1/Jenkinsfile). Darin wird eine statische HTML-Seite mit dem Webserver NGINX ausgeliefert. Dieses Beispiel käme auch ohne Pipeline aus, allerdings mit den oben erwähnten Nachteilen:

- die HTML-Datei müsste inline in einer YAML-Datei gepflegt werden,
- für die lokale Entwicklung wäre ein Helm-Operator notwendig und
- die values.yamls müssten in pro Stage komplett redundanten HelmRelease YAMLs beschrieben werden.

Insofern ist an dieser Stelle die Verwendung einer Pipeline auch beim GitOps-Deployment von Third-Party-Anwendungen vorteilhaft. Beim Blick auf die beiden Jenkinsfiles aus dem GitOps-Playground fällt auf, dass die Pipelines für die beiden unterschiedlichen Anwendungsfälle „K8s-Ressourcen" und Helm zu großen Teilen gleich sind. Hier bietet sich das Auslagern in eine Jenkins Shared Library an. Diese ist in Arbeit und wird ihren Weg in den GitOps-Playground finden.

Abschließend sei angemerkt, dass die Nutzung eines Helm-Operators auch ohne GitOps Vorteile haben kann: Die Quelle und Version des Charts sind als IaC (in YAML) deklariert, statt innerhalb eines Jenkinsfiles. Dieses wird einfach auf den Cluster angewendet. In der Pipeline wird kein Helm Binary mehr benötigt. Das gleiche Vorgehen funktioniert auch in der lokalen Entwicklung.

Fazit

Die Mehrwerte von CD stehen außer Frage. Dieser Artikel zeigt anhand beispielhafter CD-Implementierungen mit K8s- und Helm-Deployment, dass die Realisierung sowohl mit CIOps als auch GitOps gut mit Jenkins möglich ist. Die Frage „CIOps oder GitOps" ist also ein Implementierungsdetail. Beides kann hervorragend in der Praxis funktionieren. Wer bereits bestehende CD-Prozesse hat, sollte nur umstellen, wenn die Vorteile von GitOps im jeweiligen Anwendungsfall große Mehrwerte bringen. Nicht zu unterschätzen ist dabei der Aufwand für die Migration: Wer viele Pipelines hat, muss auch viele Pipelines migrieren. Für Neueinsteiger bietet es sich aufgrund der vielen Vorteile an, direkt mit GitOps zu starten. Allerdings wird die bereits steile Lernkurve dadurch noch steiler. Die vollständigen Beispiele können bei GitHub in den Repositories *[3]* (CIOps) und *[4]* (GitOps) gefunden werden.

Was dieser Artikel nicht betrachtet, sind die Unterschiede verschiedener GitOps-Operatoren. Dies ist ein Thema für sich. Ein erster Schritt, sich diesem praktisch zu nähern, kann der GitOps-Playground *[4]* sein.

Quellen

[1] *https://www.weave.works/blog/kubernetes-anti-patterns-let-s-do-gitops-not-ciops*

[2] *https://cloudogu.com/de/blog/continuous_delivery_4_de*

[3] *https://github.com/cloudogu/jenkinsfiles*

[4] *https://github.com/cloudogu/k8s-gitops-playground/tree/405d*

Johannes Schnatterer

Cloudogu GmbH

johannes.schnatterer@cloudogu.com

Seit mehr als zehn Jahren verfolgt Johannes mit seiner Tätigkeit in den Bereichen Dev, Ops und Architektur das Ziel, wartbare, sichere Anwendungen schnellstmöglich in Produktion zu bringen. Dabei greift er zu Methoden wie Continuous Delivery & GitOps, Clean Code, IaC, O11y, Dokumentation und Pragmatismus. Er ist begeistert von Open Source, allen Cloud-native-Themen und der lebhaften Community. Als Autor, Trainer und Consultant lehrt und lernt er gerne.

Daniel Huchthausen

Cloudogu GmbH

daniel.huchthausen@cloudogu.com

Daniel Huchthausen ist Consultant bei der Cloudogu GmbH aus Braunschweig. Er steht sowohl Kunden als auch Kollegen als Scrum Master und als Experte für Anforderungs- und Testmanagement sowie Prozessanalyse und -optimierung zur Verfügung.