edv-buchversand.de

GitOps

Grundlagen und Best Practices

» Hier geht's direkt zum Buch

DIE LESEPROBE

Diese Leseprobe wird Ihnen von www.edv-buchversand.de zur Verfügung gestellt.

1 Was ist GitOps?

Hand aufs Herz: Wo schaust du *zuerst* nach, wenn du wissen möchtest, welche Anwendungen konkret in einem Environment deployt sind? Vielleicht findest du dich in einer der folgenden Antworten wieder:

- »Im Cluster selbst – wir deployen nur manuell ...«
- »In den letzten Deploy-Pipelines vom CI-Server.«
- »In einem Git-Repo.«

GitOps kann dich in die Lage versetzen, die letzte der drei Antworten zu geben – und zwar mit sehr großer Zuversicht. Diese psychologische Sicherheit ist nur eines der Resultate von einem Deployment-Workflow auf GitOps-Basis.

In diesem Kapitel stellen wir GitOps ganz grundlegend vor. Dabei starten wir mit einer vagen Gegenüberstellung, um eine grundsätzliche Vorstellung von GitOps zu haben. Dann wollen wir die Hintergründe und Herausforderungen verstehen, aus denen GitOps entstand. Anschließend wenden wir uns den vier Prinzipien von OpenGitOps zu, die GitOps im Kern definieren.

1.1 CIOps vs. GitOps

Begriffe und Synonyme

- »GitOps-Operator« und »GitOps-Controller« werden fast synonym eingesetzt. Darüber hinaus gibt es noch den Begriff »GitOps-Agent« im Sinne von GitOps-Prinzip 4 (siehe Abschnitt 1.3.4 auf Seite 20). In diesem Buch verwenden wir aus Gründen der Einheitlichkeit den Begriff »GitOps-Operator«, auch wenn der von Kubernetes abstrahierte Begriff »GitOps-Agent« an vielen Stellen passender wäre. In der Praxis wird »GitOps-Operator« unserer Erfahrung nach auch am häufigsten verwendet. Allerdings wird auch im Kontext von GitOps gelegentlich der Begriff »Controller« benutzt. Unser Verständnis ist, dass »Controller« der allgemeinere Begriff ist. Unter »Operator« verstehen wir einen speziellen »Controller« samt seiner Erweiterungen der Kubernetes-API (CRDs). GitOps-Operatoren wie Flux und ArgoCD bieten umfangreiche CRDs an, insofern wirkt dieser Begriff für uns am besten passend. Sowohl Argo CD als auch Flux bestehen aus mehreren Komponenten, die teilweise als »Controller« bezeichnet werden. Unter GitOps-Operator verstehen wir in diesem Kontext die Gesamtheit dieser »Controllers«. Flux ist also ein GitOps-Operator, der aus Kustomize-Controller, Helm-Controller, Notification-Controller etc. besteht.
- Mit »Config« meinen wir beispielsweise Kubernetes-Ressourcen. Manche verwenden auch den Begriff »Infrastructure as Code« synonym. Andere sehen zwischen Config und Infrastruktur (beispielsweise virtuelle Maschinen) klare Unterschiede.

CIOps: Der CI-Server deployt in den Cluster.

Bei einer »klassisch« umgesetzten Pipeline für Continuous Deployment (CD) führt der CI-Server aktiv das Deployment in die Zielumgebung durch (Push-Prinzip). In Abb. 1–1 sehen wir ein Beispiel davon: Der CI-Server, in diesem Fall GitLab CI, verbindet sich mit einem Source Code Management (SCM), zum Beispiel GitHub. GitLab CI lädt ein Git-Repository herunter, das beispielsweise Kubernetes-Manifeste enthält. Anschließend verbindet es sich mit einem Kubernetes-Cluster (beispielsweise über eine Kubeconfig-Datei) und rollt diese Manifeste aus (beispielsweise mit einem `kubectl apply`). Dieses Verfahren bezeichnen wir als »CIOps[1]«, weil ausschließlich der CI-Server operativ tätig ist.

Punktuelle Rollouts ermöglichen massiven Drift.

CIOps hat sich jahrelang in der Praxis bewährt – aber es weist an kritischen Stellen entscheidende Mängel auf: Der Rollout wird *nur punktuell* ausgeführt. Dadurch entsteht ab der ersten Sekunde

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

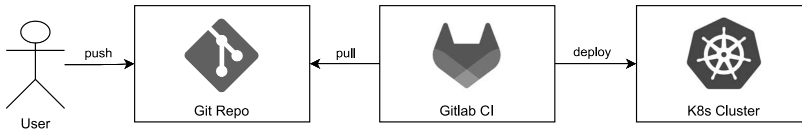


Abb. 1–1
Die »klassische« CIOps Pipeline

nach dem CI-geführten Rollout etwas, das alle Infrastruktur-Menschen fürchten: Drift – die Realität entfernt sich mit jedem Moment mehr vom ursprünglich definierten Wunschzustand. Danach manuell ausgeführte Änderungen bleiben intransparent bestehen. Solche Änderungen können aus Versehen passieren oder absichtlich, etwa durch Angreifende. Und genau dadurch entsteht das Problem, das wir am Anfang des Kapitels angerissen haben: Wer eine Woche lang keinen Commit auf das Repo macht, von dem aus die Pipeline getriggert wird, kann eine Woche lang manuelle Änderungen im Cluster machen, und es wird keinerlei automatisches Zurücksetzen auf den in Git definierten Zustand (Reconciliation) geschehen.

Der CI-Server braucht hochprivilegierten Zugriff auf den Cluster.

Außerdem haben wir gravierende Sicherheitsrisiken: Der CI-Server braucht privilegierten Zugriff auf die Zielumgebung. Meistens bekommt der CI-Server direkt administrativen Zugriff und kann dadurch in den falschen Händen viel Schaden anrichten. Das öffnet sehr gefährliche Einfallstore für Angreifer.

Der CI-Server braucht Sichtkontakt zum Cluster.

Ebenso ist ein Rollout nur dann möglich, wenn eine Netzwerkverbindung vom CI-Server zum Zielsystem besteht. Selbst wenn wir die Rollout-Pipeline im CI-Server automatisch alle 5 Minuten ausführen lassen, wird die Angleichung nur dann passieren, wenn der CI-Server den Cluster erreichen kann.

Außerdem ist in Enterprise-Umgebungen diese Verbindung zwischen Entwicklungsumgebung (CI-Server) und Betriebsumgebung (Kubernetes) aufgrund unterschiedlicher Sicherheitszonen oft eine Herausforderung. Dies kann die Verbindung unmöglich machen oder durch die Notwendigkeit von Firewall-Freischaltungen deutlich erschweren. Zwar unterliegen die umgekehrten Netzwerkverbindungen von der Betriebsumgebung auf die Entwicklungsumgebung (SCM) gegebenenfalls ähnlichen Problematiken. Unserer Erfahrung nach ist es aber meist einfacher, aus der produktiven Betriebsumgebung heraus, als in sie hineinzukommen. Mit GitOps ließe sich selbst dieses Problem lösen: Der GitOps-Operator könnte die Manifeste auch aus einer Open Container Initiative (OCI) Registry lesen, auf die Kubernetes aufgrund der Images ohnehin Zugriff braucht (siehe Abschnitt 4.13 auf Seite 90).

Kein Git-basiertes Löschen

Weiterhin sind wir ziemlich eingeschränkt, weil wir über das Git-Repo, von dem aus der CI-Server deployt, nur bestehende Ressourcen aktualisieren oder neue Ressourcen deployen können. Das Löschen von

Ressourcen hingegen ist nicht von Haus aus über Commits möglich (beispielsweise durch das Löschen einer Manifest-Datei), nur über manuelles Eingreifen oder zusätzliche Pipelines.

Geringe Auditierbarkeit durch imperative Änderungen

Die Pipelines des CI-Servers ermöglichen imperative Änderungen an der Config. Typischerweise schreibt man hier den Tag des aktuellen Images in die Config. Gängig ist aber auch das Einfügen von Secrets aus dem Credentials-Store des CI-Servers, das Spezifizieren von Helm-Charts oder das Setzen von umgebungsspezifischen Parametern. Dies führt dazu, dass die tatsächlich an den Cluster übertragene Config nur transient auf dem CI-Server besteht. Damit sind Änderungen nicht einfach nachvollziehbar und Fehler schwerer zu finden.

GitOps: Der Operator im Cluster deployt.

Wie können wir diesen Problematiken begegnen? Kann es überhaupt einen anderen Weg geben? Mit GitOps können wir ganz anders vorgehen (siehe Abb. 1–2): Der CI-Server verschwindet bei einem Rollout grundsätzlich komplett aus dem Bild. Stattdessen gibt es einen Prozess innerhalb des Zielsystems, der ununterbrochen das relevante Git-Repository pullt und auf Änderungen überprüft (Pull-Prinzip). Dieser Prozess wird »GitOps-Operator« genannt; im Diagramm ist es Argo CD.

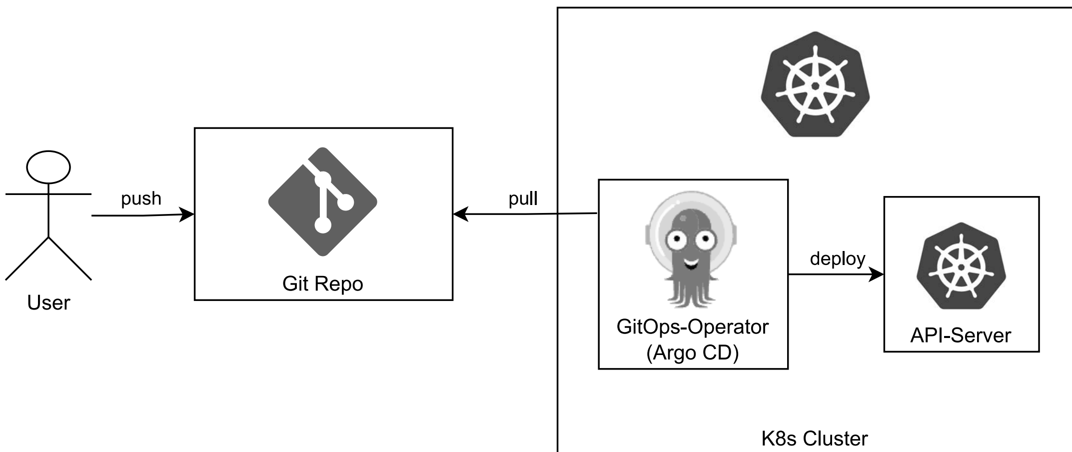


Abb. 1–2
Einfaches Deployment mittels GitOps

Damit können wir den eben genannten Schwierigkeiten bei CIOps sehr gut begegnen:

Selbstheilung durch Continuous Operations

- *kontinuierliches Ausrollen*: Wir verringern den Zeitraum drastisch, in dem Drift auftreten kann, weil die Manifeste im Git-Repository beständig gelesen und angewandt werden. Bei manuellen Änderungen sorgt der GitOps-Operator also für *Selbstheilung*. Prinzipiell implementiert GitOps das Deployment, insofern kann man es als *Cloud-Native Continuous Deployment* (oder Cloud-Native Con-

tinuous Delivery) verstehen. Als Fortführung des Gedankens von Continuous Deployment können wir an dieser Stelle auch von *Continuous Operations* sprechen.

- *bessere Sicherheit*: Da der GitOps-Operator im Cluster läuft, muss netzwerktechnisch gesehen kein administrativer Zugriff von außerhalb des Clusters mehr erfolgen.
  Außerdem zwingt uns die deklarative Natur von GitOps dazu dedizierte Werkzeuge zum Secrets Management zu verwenden, statt diese im CI-Server zu verwalten (siehe Kapitel 5 auf Seite 97). Dies reduziert das Sicherheitsrisiko, das CI-Server darstellen.
- *höhere Stabilität*: Der GitOps-Operator wendet die Manifeste beständig an. Selbst dann, wenn das SCM kurzzeitig nicht erreichbar ist, kann der GitOps-Operator über den lokal zwischengespeicherten Klon des Repositorys weiterhin einen Rollout der zuletzt gesehenen Manifeste durchführen und Drift eliminieren.
- *Löschen per Commit*: Durch geschicktes Interpretieren der Git-Historie ist der GitOps-Operator in der Lage, in Git gelöschte Ressourcen auch im Cluster zu löschen. (Dieser Vorgang wird auch als »Pruning« bezeichnet.)
- *deklarative Beschreibungen und Auditierbarkeit*: Der GitOps-Operator kann nur die finalen Manifeste anwenden, keine imperativen Änderungen vornehmen. Dies zwingt uns dazu, rein deklarativ zu arbeiten, was zu klar nachvollziehbarer Config und Historie führt. Im Nachhinein können wir durch die Historie unserer Commits klar nachvollziehen, wer wann was geändert hat. Im Idealfall beantwortet die Commit-Message auch das Warum.

Weitere Vorteile von GitOps-Operatoren

Über die Behebung der Schwierigkeiten von CIOps hinaus bieten GitOps-Operatoren noch weitere Vorteile:

- *Skalierbarkeit*: Wenn wir mehrere Instanzen von etwas betreiben wollen, beispielsweise eine Anwendung in mehreren Clustern, skaliert dies mit GitOps besser: Wir können die Config zentral in einem Repo ablegen. Aus diesem können sich dann entweder mehrere GitOps-Operatoren bedienen oder wir verwenden GitOps-Features wie das `ApplicationSet` (siehe Abschnitt 3.3 auf Seite 51), um auf mehrere Zielumgebungen zu deployen.
- *Innovationen*: Aus der Entwicklung der GitOps-Operatoren sind einige Features hervorgegangen, die über die Grundaufgabe von GitOps hinausgehen. Mit diesen können wir unsere Prozesse noch weiter verbessern. Zu diesen Innovationen zählen unter anderem das bereits genannte `ApplicationSet` für bessere Skalierbarkeit so-

wie das Lesen von Manifesten aus OCI-Registries zur Vereinheitlichung des Toolings und Vereinfachung der Netzwerkzugriffe.
Außerdem können grafische Oberflächen wie die von Argo CD (siehe Kapitel 3 auf Seite 47) die Developer Experience erhöhen, den Einsteig in Kubernetes vereinfachen und manuelle Interaktion mit dem Cluster verringern.
Mit Preview Environments können wir einfacher mehrere Features gleichzeitig in produktionsnahe Umgebungen bringen (siehe Abschnitt 6.5.2 auf Seite 157).

> Den User, der in beiden Bildern nur am Rand auftaucht, haben wir bisher noch komplett außen vor gelassen. In beiden Bildern ist er nur als jemand abgebildet, der Pushes in ein Repo ausführt. Wir werden uns in Kapitel 2 auf Seite 25 genauer damit auseinandersetzen, welche Auswirkungen die Umstellung auf GitOps für die Menschen bedeutet, die damit arbeiten.

Die vier Prinzipien

Wir stellen jetzt direkt die offiziellen vier GitOps-Prinzipien vor, damit wir sie bereits einmal gesehen haben. Sie werden uns durch das ganze Buch begleiten, und wir werden sie in Abschnitt 1.3 auf Seite 16 im Detail beleuchten. Dies sind die GitOps-Prinzipien:

1. *deklarativ*: Der Soll-Zustand eines durch GitOps verwalteten Systems muss deklarativ beschrieben sein.
2. *versioniert und unveränderlich*: Der Soll-Zustand wird in einer Weise gespeichert, die Unveränderlichkeit sowie Versionierung erzwingt und die vollständige Historie erhält.
3. *automatisch bezogen*: Software-Agenten beziehen den beschriebenen Soll-Zustand automatisch.
4. *kontinuierlich angeglichen*: Software-Agenten beobachten den tatsächlichen Systemzustand und versuchen kontinuierlich, ihn dem Soll-Zustand anzugleichen.

Diese GitOps-Prinzipien klingen anfangs sehr abstrakt; sie sind aber keinesfalls in einem rein akademischen Setting entstanden. Wir befassen uns jetzt mit dem Kontext, in dem GitOps entstand.

1.2 Der Weg zu GitOps

1.2.1 Traditionelle Silos

In den Wasserfall-Methoden konservativer Unternehmen war es üblich, dass Entwicklungsteams, Quality Assurance (QA) Engineers und Betriebsteams strikt voneinander getrennt arbeiteten (siehe Abb. 1–3). Leitende geben einen starren Zeitplan vor, in dem diese drei separierten Teams gemeinsam Software mit geschäftskritischen Features ausliefern müssen – allerdings ohne wirklich miteinander kollaborieren zu können: Entwickelnde bekommen Vorgaben und implementieren diese als Software. Danach übergeben sie den Staffelstab an das QA-Team, das die Software ausführlich testet. Anschließend findet die finale Übergabe an das Betriebsteam statt, die schließlich eine Software ausrollen, die für sie eine völlige Blackbox ist.

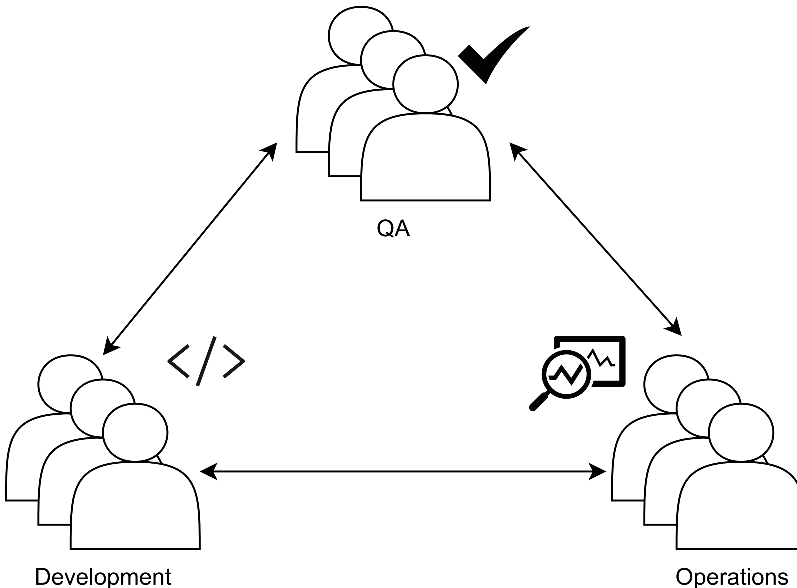


Abb. 1–3
Drei getrennte Teams: Entwicklung, QA und Betrieb

Nach Funktionen separierte Teams liefern langsam, fehleranfällig und ineffizient.

Dieser Prozess war grundsätzlich sehr langsam und hochgradig anfällig für Fehler, weil Kommunikation und Kollaboration zwischen den Teams nicht gefördert wurde. Da jede Phase dieses Entwicklungsvorgangs oft viele Wochen lang dauerte, war ein schnelles Beheben von Problemen in Produktion kaum möglich und Fehler kostspielig.

Die häufigsten Konflikte traten zwischen Entwicklungs- und Betriebsteam auf im Spannungsfeld zwischen Innovation und Stabilität: Während Entwickelnde unter dem Druck ihrer Vorgaben so schnell wie möglich neue Features liefern wollten, wollte das Betriebsteam Ände-

rungen um jeden Preis vermeiden, um die Verfügbarkeit ihrer Anwendungen zu gewährleisten. Statt die wertvolle Erfahrung sowie Sichtweisen der anderen Teams zu verstehen und voneinander zu lernen, entwickelten sich schnell auch zwischenmenschlich tiefe Gräben zwischen den Abteilungen.

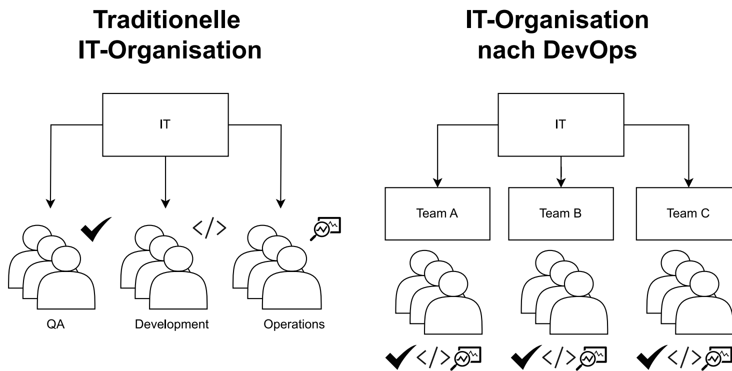
Zwar fing Automatisierung an, eine zunehmende Rolle zu spielen, weil dadurch manuelle Schritte wie ein Build oder ein Deployment deutlich stabilisiert und beschleunigt werden konnten. Dennoch waren diese Verbesserungen immer noch auf nur ein Team jeweils beschränkt. Wir mussten einen anderen und besseren Weg finden.

1.2.2 DevOps

DevOps: Crossfunktionale Teams mit gemeinsamer Verantwortung

Im Zuge von Agile begannen Organisationen bereits zunehmend, Silos abzubrechen und Kollaboration zwischen getrennten Teams zu fördern. Eine Kultur namens DevOps[2] entstand in diesem Kontext, und diese Kultur zielt auf crossfunktionale Teams ab, wie wir in Abb. 1–4 sehen: Teams sind nicht mehr nach Funktion separiert, sondern Entwickelnde, QA Engineers und Betriebler übernehmen eine gemeinsam geteilte Verantwortung (oftmals beispielsweise für ein einzelnes Softwareprodukt). Feedback, Automatisierung und Qualität von Anfang an einzubauen sind wichtige Werte für jedes Team.

Abb. 1–4 *Teamorganisationen traditionell vs. mit DevOps*



In einem solchen Umfeld kann es einem Developer nicht mehr egal sein, ob die Anwendung unperformant in Produktion läuft. Ebenso kann ein Betriebler nicht länger Deployments mit Instabilitätssorgen blockieren. Alle haben eine gemeinsame Verantwortung und lernen deshalb voneinander, was in der jeweiligen Rolle wichtig ist.

[2] *https://martinfowler.com/bliki/DevOpsCulture.html*

Geschwindigkeit und Stabilität stehen nicht mehr im Widerspruch

Dank dieser Fokussierung und Kollaboration und durch hochgradige Automatisierung sind kurze Feedbackzyklen und zügiges Deployen nicht länger ein ferner Wunschtraum. Mittels Continuous Integration (CI) wird auf jedem Commit im Standardbranch eine Pipeline ausgeführt, die den Code testet und einen Build durchführt, an dessen Ende ein deploybares Artefakt steht (beispielsweise ein Java Archive (JAR) oder ein Container-Image)[3]. Mit Continuous Delivery (CD) als nächster Stufe wird das in CI gebaute Artefakt automatisch auf ein Environment deployt, Integrationstests werden ausgeführt und bei Erfolg wird ins nächste Environment deployt bis direkt vor Produktion. Bei Continuous Deployment (ebenfalls mit CD abgekürzt) wird selbst das Deployen nach Produktion vollständig automatisiert.

1.2.3 Infrastructure as Code

Weil Cloud Computing und SaaS-Dienste im Laufe der Zeit zunehmend genutzt wurden, gewannen Themen wie Self-Service und dynamische Infrastruktur an Bedeutung. Im Gegensatz zur »Eisenzeit«, in der man zum Deployen physische Hardware im On-Premises-Rechenzentrum benötigte, konnten wir nun im Cloud-Zeitalter mit sehr geringen Hürden nach Belieben virtuelle Infrastruktur erstellen und wieder abreißen[4]. In diesem Kontext entstand die Praxis von *Infrastructure as Code* (IaC).

Infrastruktur wird mit den gleichen Methoden wie Code behandelt.

Wie der Name anklingen lässt, behandelt IaC die Infrastruktur, die zum Deployen von Code benötigt wird, mit ähnlichen Methoden, wie der Code selbst behandelt wird: Statt manuell sich mit Servern zu verbinden und Änderungen durchzuführen, werden Änderungen an Code gemacht, dieser wird committet, eine Pipeline führt Tests aus und rollt diese Änderungen automatisiert aus.

Geringere Hürden, »Operations by PR«

Mit dieser Herangehensweise sind Änderungen an der Infrastruktur mit weniger Hindernissen verbunden. Sowohl Entwickelnde als auch Betriebler haben ein einheitliches Format, um über Infrastruktur zu diskutieren, und können kollaborativ an Infrastruktur arbeiten, wo es nötig ist. Selbst kleine Änderungen an Infrastruktur-Code können genauso leicht wie große Änderungen durch Pull Requests (PRs) durchgeführt werden, was iterative Herangehensweisen an Infrastrukturaufgaben fördert.

[3] *https://martinfowler.com/articles/continuousIntegration.html*

[4] Morris, Kief. Infrastructure as Code: Managing Servers in the Cloud. 2016.

Pull/Merge Requests

Manche SCMs nennen es »Pull Request« (PR), andere »Merge Request« (MR). Wir meinen damit in jedem Fall denselben Mechanismus, der einen Feature Branch als »Änderungsanfrage« bereitstellt, die bei Annahme in den Standardbranch gemergt und (im Fall von CD) direkt ausgerollt wird. Wir verwenden der Einfachheit halber in diesem Buch immer den Begriff »Pull Request«.

Deklarativer Code beschreibt nur den Zielzustand.

Bei dem Code, der IaC ausmacht, gibt es allerdings deutliche Unterschiede zu konventionellem Anwendungscode: Meist ist er nicht imperativ, sondern *deklarativ*. Imperativer Code beschreibt, *wie* etwas ausgeführt werden soll. Deklarativer Code hingegen beschreibt, *was für ein Zielzustand* erreicht werden soll.

Ein einfaches Beispiel zur Illustration ist die Gegenüberstellung von jQuery und CSS: Mit jQuery können wir beispielsweise ein bestimmtes Element mit einer bestimmten ID rot einfärben:

Listing 1–1 *Imperatives Einfärben mit jQuery*

```
$(document).ready(function(){
    $("p").on({
        mouseenter: function(){
            $(this).css("color", "red");
        }
    });
});
```

Dabei führen wir einen JavaScript-Befehl aus – das ist ein imperatives Vorgehen. Dasselbe Ergebnis können wir mit CSS erreichen, indem wir in einem Block unseren Wunsch ausdrücken, dass alle gewünschten Elemente beim Hovern rot eingefärbt werden:

Listing 1–2 *Deklaratives Einfärben mit CSS*

```
p:hover {
  color: red;
}
```

Deklaratives IaC

IaC gibt es sowohl in imperativer Form (beispielsweise als Ansible-Playbook oder Chef Infra Cookbook) als auch in deklarativer Form (zum Beispiel als Kubernetes-Manifeste oder Terraform-Dateien). Für den Fokus dieses Buches gilt: Wenn wir von IaC schreiben, dann meinen wir ausschließlich deklarative Formen von IaC, weil nur diese im Kontext von GitOps nutzbar sind.

Diese deklarative Formulierung hat einige Vorteile:

- Der Code ist leichter zu lesen, weil die Absicht eindeutiger ist.
- Der Code ist leichter weiterzuentwickeln, weil wir kein Wissen über die konkrete Implementierung benötigen.
- Optimierungen an der dahinterliegenden Implementierung (beispielsweise hinsichtlich Performance) kommen uns im Hintergrund unmittelbar zugute, ohne dass wir selbst dafür tätig werden müssen.

Bei IaC wird genau das gleiche Prinzip von Deklarationen verwendet: Statt ein Shell-Skript zu nutzen, definieren wir unseren Wunschzustand beispielsweise in Terraform-Dateien, Pulumi-Code, Kubernetes-Manifesten oder Docker-Compose-Dateien. Dadurch ernten wir bei Infrastruktur-Code ganz ähnliche Vorteile wie eben bei CSS beschrieben.

Deklarationen abstrahieren imperative Befehle weg.

Selbstverständlich braucht es zur Umsetzung von deklarativem Code an irgendeiner Stelle auch imperative Befehle. Aber diesen imperativen Code können wir durch Deklarationen elegant wegabstrahieren und mehr Zeit auf wertvolle Infrastruktur-Änderungen investieren. Dieser imperative Code für Infrastruktur muss sich nämlich auch mit vielen Edge Cases befassen:

- *Konvergenz*: Ist die Ressource bereits im gewünschten Zustand? Ein einmaliger Befehl wird dafür oft nicht ausreichen. Stattdessen müssen mehrere Befehle ausgeführt werden und bei diesen Befehlen muss immer wieder auf erfolgreiche Ergebnisse gewartet werden.
- *Abhängigkeiten*: Abhängige Ressourcen müssen zuerst erstellt werden. Diese Abhängigkeiten sollten möglichst automatisch erkannt werden.
- *Idempotenz*: Ein mehrfaches Ausrollen derselben Deklarationen sollte zu einem identischen Ergebnis führen.

Wenn wir Infrastruktur mit IaC verwalten, gehen wir davon aus, dass die eben genannten Punkte grundsätzlich vom Interpreter unserer IaC-Deklarationen (beispielsweise die Terraform-CLI bei Terraform-Manifesten) erfüllt werden. Das letzte Kriterium der Idempotenz wollen wir kurz genauer betrachten:

Idempotenz macht IaC-Rollouts risikofrei.

Idempotente Befehle führen bei wiederholter, sequenzieller Ausführung immer zum gleichen Ergebnis. Ein Negativbeispiel, das Idempotenz nicht erfüllt, wäre der Shell-Befehl `echo gitops >> test.txt`, weil nach jeder Ausführung eine zusätzliche Zeile in der Text-Datei steht. Der Shell-Befehl `echo gitops > test.txt` hingegen ist idempotent, weil

nach jeder Ausführung das Resultat identisch ist: Immer steht eine einzige Zeile mit dem Inhalt »gitops« in der Text-Datei.

Ein Controller, der IaC-Manifeste ausrollt, muss letztlich imperative Aktionen ausführen, damit unser gewünschter Zustand Realität wird. Wenn jedoch die Aktionen, die er durchführt, so idempotent wie nur möglich sind, dann sind wir in guten Händen, denn ein wiederholter Rollout von Manifesten, in denen sich nichts geändert hat, ist dann vollkommen ungefährlich.

1.2.4 Kubernetes

Kubernetes[5] ist mittlerweile die am weitesten verbreitete Plattform für Containerbetrieb. Dieses Cluster-Betriebssystem, das aus seiner ursprünglichen Herkunft bei Google 2014 als Open Source zur Verfügung gestellt wurde, hat sich sowohl bei großen Cloud-Providern als auch On-Premises bewährt. Auch wenn keine Plattform alle Anforderungen erfüllen kann, hat Kubernetes einige ganz besondere Eigenschaften.

Eine große Kontrollschleife mit deklarativen Manifesten

Eine dieser Eigenschaften ist, dass deklarative YAML-Manifeste der bevorzugte Weg sind, um mit dem Kubernetes-API-Server (sozusagen dem Backend eines Kubernetes-Clusters) zu reden. Kubernetes arbeitet nämlich mit Controllern, die solche Manifeste entgegennehmen und kontinuierlich dafür sorgen, dass diese »Wunschlisten« Realität werden. Dieses kontinuierliche Konvergieren wird auch als Reconciliation bezeichnet. Kubernetes ist mit seinen Controllern und den zugrunde liegenden Ideen aus der Kontrolltheorie letztlich *als eine einzige große Kontrollschleife implementiert*.

Kubernetes nimmt also einige der Werte und Praktiken von IaC und setzt sie praktisch um. Und wenn wir zurückdenken an unsere initiale Beschreibung von GitOps, dann wird klar, dass ein GitOps-Operator, der kontinuierlich ein Repo überwacht und dessen Manifeste ausrollt, durch die Grundstruktur von Kubernetes nicht sehr schwer zu implementieren sein sollte.

1.2.5 OpenGitOps

Und tatsächlich ist es so, dass die beiden großen GitOps-Operatoren Flux und Argo CD, die heutzutage zunehmend genutzt werden, überhaupt erst auf Basis von Kubernetes entstehen konnten. 2014 wurde die Firma Weaveworks gegründet, und sie setzten früh auf Kubernetes als Container-Orchestrator[6]. 2016 praktizierten Mitarbeiter bereits GitOps, ohne dass es den Begriff überhaupt schon gab, und durch

[5] *https://kubernetes.io*

[6] *https://www.weave.works/blog/the-history-of-gitops*

GitOps konnten sie bei einem Incident ihr gesamtes gelöschtes System innerhalb von weniger als einer Stunde wiederherstellen.

2017: Alexis Richardson definiert erstmals GitOps.

Anschließend beschrieb Alexis Richardson im Blogpost »GitOps – Operations by Pull Request« im August 2017 zum ersten Mal die damaligen Prinzipien hinter GitOps und benutzte dafür auch zum ersten Mal das Wort »GitOps«[7]. Grundsätzlichen sind die meisten der damaligen Prinzipien auch in den heutigen Prinzipien enthalten, allerdings in deutlich verfeinerter Form.

Flux und Argo CD entstehen und reifen.

Dieser Blogpost war auch der Moment, wo Weaveworks ihren GitOps-Operator *Flux CD*[8] vorstellten. Ebenfalls im Jahr 2017 begann bei Intuit die Entwicklung von Argo CD[9,10]. 2019 wurde Flux in die Cloud Native Computing Foundation (CNCF) aufgenommen, das Argo-Projekt folgte im Jahr 2020. 2020 wurde Flux von Grund auf neu geschrieben und als deutlich reifere und stabilere v2 releast. Im Jahr 2022 erreichten beide Tools den höchsten Reifegrad »Graduated«.

2020: Die GitOps Working Group definiert OpenGitOps.

Aufgrund der rasant wachsenden Einführung von GitOps in der gesamten Industrie bildete sich 2020 eine sogenannte Working Group innerhalb der CNCF, die »GitOps Working Group«. Unter den Gründungsmitgliedern waren Mitarbeiter von Firmen wie Amazon, Azure, GitHub, RedHat und Weaveworks[11]. Diese Working Group überarbeitete die ursprünglichen GitOps-Prinzipien von Alexis Richardson und erschuf »OpenGitOps«[12]. Unter diesem Projektnamen wird ein GitOps-Standard erarbeitet und verwaltet, der solide und herstellerneutrale Prinzipien für GitOps gewährt. Im Oktober 2021 wurde die Version 1.0.0 von OpenGitOps und den vier Prinzipien veröffentlicht. Im Jahr 2021 fand auch zum ersten Mal die »GitOpsCon«, eine dedizierte Konferenz zu GitOps, im Rahmen der KubeCon statt.

2023: Zertifizierungen in Vorbereitung

Die GitOps Working Group arbeitet auch weiterhin an der Standardisierung von GitOps. Neben der Bereitstellung von Wissen, Use Cases und Whitepapers, wurde auf der GitOpsCon 2023 das Zertifizierungsprogramm »Certified GitOps Associate (CGOA)«[13] im Rahmen der Linux Foundation vorgestellt. Interessierte können sich auf diese Weise GitOps-Wissen aneignen und die Erfüllung der Anforderungen zur Erlangung eines Zertifikats nachweisen. Zudem wird an einem Sie-

[7] *https://weave.works/blog/gitops-operations-by-pull-request*
[8] *https://fluxcd.io*
[9] *https://argoproj.github.io/cd*
[10] *https://www.cncf.io/reports/argo-project-journey-report*
[11] *https://opengitops.dev/community*
[12] *https://opengitops.dev*
[13] *https://training.linuxfoundation.org/certification/certified-gitops-associate-cgoa*

gel »Certified OpenGitOps Compliance« gearbeitet, damit Tools in der Lage sind, die definierten OpenGitOps-Standards in ihren Produkten nachzuweisen.

1.3 Die vier Prinzipien

Nun sind wir bei OpenGitOps gelandet und können ungefähr sehen, wie die Herausforderungen, denen wir mit DevOps und IaC begegnen und bei denen uns Kubernetes hilft, uns zu den vier Prinzipien führen, die heutzutage GitOps ausmachen. Im Folgenden stellen wir diese vier Prinzipien vor. OpenGitOps besteht im Kern aus diesen vier Prinzipien und einem zugehörigen Glossar.

Beide Materialien wurden bereits in mehrere Sprachen übersetzt, auch ins Deutsche. Wir würdigen diese Übersetzungsleistung und verwenden die deutschen Begriffe in diesem Buch, an manchen Stellen greifen wir jedoch auch auf die entsprechenden englischen Begriffe zurück, da Englisch im Entwicklungsalltag ein ständiger Begleiter ist.

Wir stützen uns auf OpenGitOps v1.0.0[14].
Allerdings sind die deutschen Übersetzungen nicht in diesem Release enthalten, deswegen verlinken wir dafür auf konkrete Commits:

- die Prinzipien auf Deutsch[15]
- das Glossar auf Deutsch[16]

Diese Prinzipien sind anfangs noch sehr abstrakt und schwer vorstellbar. Das ist wichtig, damit sie so wenig wie möglich an eine bestimmte Implementierung oder Plattform gebunden sind. Andererseits ist das nicht hilfreich, weil wir die Konsequenzen daraus nur schwer erfassen können. Deswegen führen wir zuerst jedes Prinzip inklusive seiner Glossareinträge auf und erläutern es anschließend zusätzlich mit eigenen Beschreibungen und Analogien.

1.3.1 Prinzip 1: Deklarativ

»Der *Soll-Zustand* eines durch GitOps verwalteten *Systems* muss *deklarativ beschrieben* sein.«

[14] *https://github.com/open-gitops/documents/releases/tag/v1.0.0*

[15] *https://github.com/open-gitops/documents/blob/c4a016f/i18n/PRINCIPLES_de.md*

[16] *https://github.com/open-gitops/documents/blob/c4a016f/i18n/GLOSSARY_de.md*

Glossareintrag »Soll-Zustand«

Die Gesamtheit aller Konfigurationen, die es braucht, um ein sich gleich verhaltendes System wiederherzustellen. Diese Konfigurationen enthalten im Allgemeinen keine gespeicherten Anwendungsdaten wie zum Beispiel Datenbankinhalte, wohl aber die entsprechenden Zugangsdaten für den Zugriff darauf oder Einstellungen für Wiederherstellungs-Tools des Systems.

Glossareintrag »Softwaresystem«

Ein mittels GitOps verwaltetes Softwaresystem beinhaltet:

- eine oder mehrere Laufzeitumgebungen, die aus verwalteten Ressourcen bestehen
- Verwaltungsagenten innerhalb jeder einzelnen Laufzeitumgebung
- Richtlinien zur Steuerung des Zugriffs sowie der Verwaltung der Repositorys, Deployments und Laufzeitumgebungen

Glossareintrag »Deklarative Beschreibung«

Eine Konfiguration, die den gewünschten Soll-Zustand eines Systems beschreibt, ohne Vorgehensweisen zu definieren, wie dieser erreicht wird. Dies erzielt die Trennung der Konfiguration (Soll-Zustand) von der Implementierung (Befehle, API-Aufrufe, Skripte und so weiter), die verwendet wird, um den gewünschten Zustand zu erreichen.

Infrastructure as Code

Wir erkennen an dieser Stelle das Prinzip von deklarativem Arbeiten wieder, das uns in IaC erstmalig begegnet ist. Imperative Deployment-Skripte, die bei CIOps an der Tagesordnung sind, haben bei GitOps keinen Platz. Deklarative Formate hingegen wie HashiCorp Configuration Language (HCL, das Format von Terraform-Dateien), Pulumi-Code, YAML (bei Kubernetes, Docker-Compose, AWS CloudFormation und vielen mehr) und Azure Bicep sind hervorragend geeignet.

Analogie Hausbau

Gerne nutzen wir in diesem Zusammenhang die Analogie eines Hausbaus: Ein Haus kann man selbst bauen, wenn man entsprechend handwerklich begabt ist. Man legt das Fundament auf einem vorher festgelegten Grundstück, baut Mauern, verlegt die Elektrik und natürlich das Dach. Allerdings ist dieser Prozess sowohl sehr langwierig als auch aufwendig. Der Bauherr muss sich in allen genannten Bereichen gut auskennen, damit nicht »gepfuscht« wird.

Im Gegensatz dazu hat der Bauherr aber auch die Möglichkeit, ein Fertighaus bauen zu lassen. Vorab vereinbaren der Bauherr und der Dienstleister einen zu erfüllenden Vertrag. Der Vertrag und die Baupläne enthalten genaue Vereinbarungen über die Anzahl und die Größe

der Räume, wie die Räume beheizt oder belüftet werden sowie die Ausstattung der Bäder. Die Aufgabe des Dienstleisters ist es, die Beschreibungen dieses Vertrags in konkrete Pläne umzuwandeln und eigenständig auszuführen.

In dieser Analogie entspricht das Haus dem Softwaresystem, Dienstleister dem GitOps-Operatoren, Bauherren den Entwickelnden (die IaC-Code formulieren) und die Baupläne den deklarativen Beschreibungen.

1.3.2 Prinzip 2: Versioniert und unveränderlich

»Der Soll-Zustand wird in einer Weise *gespeichert*, die Unveränderlichkeit sowie Versionierung erzwingt und die vollständige Historie erhält.«

Glossareintrag »Zustandsspeicher«

Ein System, um unveränderliche Versionen der Beschreibung des Soll-Zustands zu speichern. Dieser Speicher sollte Zugriffssteuerung und Audits der Änderungen des Soll-Zustands unterstützen. Git, von dem sich der Name GitOps ableitet, ist das kanonische Beispiel für diesen Speicher, aber jedes System, das die genannten Bedingungen erfüllt, kann benutzt werden. In jedem Fall muss der Speicher ordnungsgemäß konfiguriert sein sowie Maßnahmen getroffen werden, um den Anforderungen der GitOps-Prinzipien gerecht zu werden.

In diesem Prinzip sehen wir zum ersten Mal die Anklänge an den Namen »GitOps«. Streng genommen haben wir es an dieser Stelle mit zwei separaten Kriterien zu tun, die aber eng miteinander verknüpft sind: Versionierung und Unveränderlichkeit.

Versionierung ist heutzutage technisch ziemlich einfach zu erzielen: Wir nutzen beispielsweise eine Versionsverwaltung (Git, Mercurial, Subversion) oder alternativ einen Object Store (AWS S3, Azure Blobs, Google Cloud Storage) mit aktivierter Versionierung.

Ohne Unveränderlichkeit kein deterministischer Rollout

Unveränderlichkeit hingegen ist nicht zuallererst ein technisches, sondern ein *inhaltliches* Kriterium. Es geht nicht primär um Immutable Image-Tags, auch wenn sie bei Prinzip 2 vieles vereinfachen. Stattdessen hilft folgender hypothetischer Test, um zu überprüfen, ob Deklarationen unveränderlich formuliert sind: Wenn ich in der Zukunft dieselben Manifeste erneut ausrolle, wird der Endzustand der gleiche sein?

Negativbeispiele

Schauen wir uns ein paar Beispiele für Inhalte in Deklarationen an, die diesen Test nicht erfüllen:

1. ein Flux-`HelmRelease` ohne definierte Version (es wird immer die neueste Version genommen)
2. ein Kubernetes-`Deployment` mit einem Rolling Image-Tag, beispielsweise `nginx:latest` oder `v3`
3. ein Flux-`HelmRelease` mit einer Semantic Versioning (SemVer) Range als Versionsnummer, beispielsweise `10.1.x`
4. eine Referenz auf ein Secret in HashiCorp Vault ohne Versionsangabe (es wird immer die neueste Version genommen)

Auch wenn diese Beispiele alle nicht exakt das Kriterium von Unveränderlichkeit erfüllen, sehen wir dennoch, dass es Unterschiede in den Auswirkungen gibt: Wenn bei den jeweiligen Anwendungen SemVer befolgt wird, dann nimmt die Schwere der Auswirkungen von oben nach unten graduell ab.

Positivbeispiele

Betrachten wir als Kontrast ein paar Positivbeispiele, die vollständig Unveränderlichkeit erfüllen:

1. ein Flux-`HelmRelease` mit einer fixen Version, beispielsweise `10.1.0`
2. ein Kubernetes-`Deployment` mit einem Immutable Image-Tag (das muss allerdings in der jeweiligen Registry konfiguriert sein)
3. ein Kubernetes-`Deployment` mit einem Image-Tag *und Digest*, beispielsweise `nginx:latest@sha256:9504f3f64a3f16f0...`
4. eine Referenz auf ein Secret in HashiCorp Vault mit Versionsangabe

Unveränderlichkeit hat ihren Preis.

Je exakter man Versionen pinnt, desto mehr Aufwand entsteht dabei aber auch. Manche Aufwände kann man mit Tools wie Renovate[17] automatisieren, aber auch die Instandhaltung davon kommt mit Kosten. Letztendlich ist Unveränderlichkeit also keine binäre Entscheidung, sondern wird immer ein Kompromiss sein. Ist mir ein möglichst exaktes Pinnen einer Version wichtig genug für diese oder jene Ressource – oder kann ich mit einem bestimmten Grad an Nichtdeterminismus leben? Diese Entscheidungen sind letztlich immer eine kontextabhängige Entscheidung, für die es kaum Leitlinien geben kann. Immerhin stößt GitOps uns mit der Nase darauf und fordert uns dazu heraus, diese Entscheidungen bewusst zu treffen.

Analogie Hausbau fortgeführt

Führen wir noch die Analogie des Hausbaus fort: Der Bauherr und der Dienstleister sind mit dem Vertrag und den Bauplänen zum Notar gegangen. Beide Parteien übergeben die Dokumente dem Notar, las-

[17] *https://www.mend.io/renovate*

sen sich eine Kopie mitgeben, und der Notar verwahrt die originalen Dokumente in einem Safe.

Wenn der Bauherr eine Änderung an den Plänen wünscht, dann schlägt er diese dem Dienstleister schriftlich vor. Ist dieser einverstanden, dann unterschreibt er das Dokument und schickt es an den Notar. Der Notar nimmt das Dokument entgegen, schickt den beiden Parteien Kopien des Dokuments und legt es zur ersten Fassung in den Safe dazu. So haben alle Parteien immer einen aktuellen Stand der Dokumente.

Der Safe beim Notar ist in diesem Fall der Zustandsspeicher (beispielsweise ein Git-Repo), der Änderungsvorschlag des Bauherren entspricht einem PR, das Unterzeichnen des Änderungsvorschlags ist der Approval auf dem PR, die Kopien sind dezentrale, lokale Kopien des Git-Repos und das Ablegen im Safe mit dem Verschicken der Kopien entspricht dem Merge des PR. (Der Notar selbst ist in diesem Fall keine wirkliche entscheidende Entität, aber vielleicht ist er mit dem SCM insgesamt gleichzusetzen, alternativ mit einem CI-Server, der nach dem Durchlaufen einer PR-Pipeline automatisch einen Merge durchführt.)

1.3.3 Prinzip 3: Automatisch bezogen

»Software-Agenten beziehen den beschriebenen Soll-Zustand automatisch.«

Der »Software-Agent« ist nichts anderes als der GitOps-Operator, von dem wir bereits am Anfang des Kapitels gesprochen haben. Dieser lädt die deklarativen Beschreibungen aus dem Zustandsspeicher selbstständig herunter (Pull-Prinzip), beispielsweise indem er in regelmäßigen Intervallen einen `git clone` ausführt.

Prinzip 3 und 4 lassen sich praktisch nicht trennen.

Prinzip 3 und 4 gehören im Grunde genommen untrennbar zusammen. Dennoch beschreiben sie zwei unterschiedliche Aspekte von »Continuous Operations«: Prinzip 3 legt den Fokus auf das Herunterladen der Manifeste, während Prinzip 4 den Rollout dieser Manifeste beschreibt.

Die Trennung ergibt in der Theorie und für die Eindeutigkeit der Prinzipien Sinn; in der Praxis werden wir jedoch kein System finden, das nur eines dieser beiden Prinzipien erfüllt. Deswegen wenden wir uns direkt Prinzip 4 zu und betrachten es im Verbund mit Prinzip 3.

1.3.4 Prinzip 4: Kontinuierlich angeglichen

»Software-Agenten beobachten den tatsächlichen Systemzustand und versuchen *kontinuierlich*, ihn dem Soll-Zustand *anzugleichen*.«

Glossareintrag »kontinuierlich«

Mit »kontinuierlich« ist im Kontext der Angleichung gemeint, dass diese regelmäßig, aber nicht zwangsweise sofort erfolgt.

Wenn uns eine sofortige Angleichung wichtig ist, gibt es dennoch Möglichkeiten, die aber mit ihren eigenen Herausforderungen kommen. Diese beleuchten wir in Abschnitt 7.2.4 auf Seite 212 genauer.

Glossareintrag »Angleichung«

Der Prozess, bei dem sichergestellt wird, dass der tatsächliche Zustand eines Systems mit seinem Soll-Zustand übereinstimmt. Im Gegensatz zur traditionellen CI/CD, bei der die Automatisierung im Allgemeinen durch voreingestellte Auslöser gesteuert wird, wird bei GitOps die Angleichung immer dann ausgelöst, wenn eine Abweichung vorliegt.

Die *Abweichung* kann darauf zurückzuführen sein, dass sich der Ist-Zustand unbeabsichtigt geändert hat oder dass eine neue Version der Soll-Zustands-Beschreibung vorliegt. Auf der Grundlage von Richtlinien und *Feedback* des Systems sowie früherer Angleichungsversuche werden Maßnahmen ergriffen, um die Abweichung im Laufe der Zeit zu verringern.

Glossareintrag »Abweichung«

Abweichung bezeichnet eine (beginnende) Entfernung des Ist-Zustandes eines Systems vom gewünschten Soll-Zustand.

Glossareintrag »Feedback«

GitOps folgt der Kontrolltheorie und wird in einem geschlossenen Kreislauf betrieben. In der Kontrolltheorie beschreibt die Rückmeldung, wie frühere Versuche, einen Soll-Zustand anzuwenden, den tatsächlichen Zustand beeinflusst haben. Verlangt beispielsweise der Soll-Zustand mehr Ressourcen, als in einem System vorhanden sind, könnte der Software-Agent versuchen, automatisch zu einer vorherigen Version zurückzurollen oder den menschlichen Betreibern einen Alarm senden.

Mit Abweichung ist genau der Drift gemeint, den wir anfangs als die besonders große Gefahr von CIOps identifiziert haben. Hier sehen wir auch Beispiele für Ursachen von Drift: Dies können entweder manuelle Änderungen im Cluster sein, die wir eliminieren wollen und die vom GitOps-Operator rücksichtlos überschrieben werden, oder es können bewusste Änderungen sein, die wir in einem Git-Repo durchgeführt haben und dann vom Agenten ausgerollt werden.

Die Formulierung, dass »Maßnahmen ergriffen werden, um die Abweichung im Laufe der Zeit zu verringern«, passt haargenau zu Beschreibungen, die man über die generelle Arbeitsweise von Kubernetes-Controllern liest. Wir sehen erneut, warum Kubernetes ein exzellentes »Substrat« bildet, auf dem GitOps hervorragend gedeiht.

Analogie Hausbau fortgeführt

Wenden wir uns noch ein letztes Mal der Hausbau-Analogie zu. Der Vertrag zwischen Bauherr und Dienstleister enthält eine besondere Klausel: Der Dienstleister ist verpflichtet, in regelmäßigen Abständen das Grundstück zu begutachten und Änderungen, die seit dem letzten Besuch passiert sind und die den Bauplänen widersprechen, rückgängig zu machen. Dazu könnten selbst verlegte Leitungen, eigens gezimmerte Anbauten, aber auch beschädigte Türen oder zerstörte Fenster zählen. Solche Änderungen werden rigoros beseitigt und müssen stattdessen durch den schriftlichen Prozess durchgeführt werden, wenn sie permanent sein sollen.

Der Bauherr hat sich damit ein zweischneidiges Schwert eingehandelt: Wenn er manuelle Aufwände investiert, werden sie kurz darauf zunichte gemacht werden. An solchen Stellen wird ihn der zusätzliche Aufwand über den Schriftweg ärgern. In den Situationen jedoch, in denen der Dienstleister anstandslos Schäden repariert, wird er sich sehr glücklich schätzen – und dann geht ihm das Beantragen von Änderungen schon viel leichter von der Hand.

In diesem Fall sind die eigenen Anbauten und beschädigten Elemente die Resultate von manuellen Tätigkeiten im Cluster (beispielsweise Installationen von Helm-Charts oder Löschungen von Ressourcen).

Bei Prinzip 1 ist am meisten ersichtlich, wie es aus IaC entstanden ist. Bei Prinzip 2 können wir zumindest noch erkennen, wie eine DevOps-Kultur von gemeinsamer Verantwortung einen auditierbaren Zustandsspeicher bevorzugt. Prinzip 3 und 4 hingegen sind das, was GitOps am meisten zu GitOps selbst macht. Deswegen wird auch die Hausbau-Analogie Schritt für Schritt surrealer, weil es im echten Leben bisher kaum Situationen gibt, für die man eine gute Entsprechung für den Kern von GitOps findet.

1.4 Fragen und Missverständnisse

An dieser Stelle wollen wir in Kurzform auf einige häufige Fragen eingehen, die sich beim ersten Kontakt mit GitOps stellen:

- *Kubernetes setzen wir nicht ein. Kann ich trotzdem GitOps nutzen?* Ja! GitOps sind zuallererst die vier Prinzipien, und diese beschränken sich mit keinem Wort auf Kubernetes. Jedoch sind die reifsten und am weitesten verbreiteten GitOps-Operatoren Argo CD und Flux auf Kubernetes ausgelegt. Deswegen beschäftigen wir uns auch den allergrößten Teil dieses Buches mit GitOps im Kontext von Kubernetes. In Kapitel 12 auf Seite 333 beleuchten wir aber auch GitOps außerhalb von Kubernetes.
- *Ist mein Projekt bereit für GitOps?* Ja, höchstwahrscheinlich! Einzige Voraussetzung sind deklarative Beschreibungen (zum Beispiel Kubernetes-Manifeste), die du in einem Zustandsspeicher (zum Beispiel einem Git-Repo) speicherst.
- *Muss ich Anwendungscode und Deklarationen zwingend in separaten Repositories lagern?* Nein. Die Praxis, Code und Config zu trennen, hat sich zwar an vielen Stellen bewährt, und wir halten dieses Vorgehen auch in vielen Situationen für angemessen. Es kann jedoch auch Gründe geben, diese nicht zu trennen. In Abschnitt 6.4.3 auf Seite 141 besprechen wir Gründe für und wider und zeigen Kompromisse auf.
- *Wie integriere ich GitOps in meine existierende CI/CD-Pipeline?* In Kapitel 3 auf Seite 47 arbeiten wir als Einstieg ein grundlegendes Tutorial zur Installation eines GitOps-Operators und zum Arbeiten mit einem Config-Repo durch. Bestehende Pipelines, die CIOps implementieren, müssen nicht erweitert werden – sie können sogar gekürzt werden um den Rollout-Schritt. Zusätzliche Schritte vom CI-Server sind nicht nötig, denn der GitOps-Operator führt den Rollout (basierend auf den Manifesten in Git) selbstständig aus. Durch GitOps führen wir Asynchronität in unseren Deployment-Ablauf ein. Die Herausforderungen, die daraus entstehen, betrachten wir eingehend in Kapitel 7 auf Seite 197.
- *Meine CI/CD-Pipeline überträgt rein deklarative Manifeste unverändert an den Cluster. Ist das GitOps?* Mindestens das Prinzip 4 (kontinuierlich angeglichen) ist dabei nicht erfüllt. Vielleicht hilft es, den Begriff »GitOps« von »Git-basierten« Workflows oder Pipelines abzugrenzen, die zumindest Prinzip 1 und 2 erfüllen. Vor der Einführung der Prinzipien gab es einige Missverständnisse darüber, was genau GitOps bezeichnet. Deswegen sind Artikel und Blogposts über GitOps, die vor der Festlegung der vier Prinzipien

veröffentlicht wurden, mit sehr viel Vorsicht zu genießen, weil teilweise Dinge als GitOps bezeichnet werden, die nicht mit den heutigen Prinzipien übereinstimmen. Dennoch ist niemand gezwungen, von ihrem aktuellen Workflow auf GitOps zu wechseln, wenn die Vorteile von GitOps die Einschränkungen des aktuellen Workflows nicht überwiegen.

- *Wie funktioniert GitOps mit Secrets?* Guter Punkt: Der gesamte Zustand muss bei GitOps deklarativ beschrieben sein, so auch Secrets. Natürlich sollten Secrets nie im Klartext in Git stehen. Die Lösung ist die Verwendung dedizierter Werkzeuge zum Secrets Management. In Kapitel 5 auf Seite 97 schauen wir uns verschiedene Optionen für den Umgang mit Secrets im GitOps-Umfeld an.
- *Kann ich nur Container mit GitOps verwalten?* Nein, du kannst letztlich alles mit GitOps verwalten, das sich deklarativ ausdrücken lässt. Da GitOps sich aus Infrastructure as Code (IaC) entwickelt hat, gibt es momentan vor allem deklarative Formate für das Deployen von Containern (zum Beispiel native Kubernetes-Manifeste), für das Verwalten von Cloud-Infrastruktur (zum Beispiel mit Terraform, Pulumi oder Crossplane, siehe Kapitel 11 auf Seite 291) und für das Verwalten von Berechtigungen (beispielsweise mit Kyverno[18] oder Gatekeeper[19] aus dem Projekt Open Policy Agent).

[18] *https://kyverno.io*
[19] *https://open-policy-agent.github.io/gatekeeper*

3 Wie fange ich mit GitOps an?

Nach dem vorherigen Kapitel haben wir eine konkrete Vorstellung davon, welchen Unterschied GitOps im Entwicklungsalltag macht. In diesem Kapitel wollen wir erste praktische Schritte in Richtung GitOps unternehmen. Am Ende dieses Kapitels werden wir einen lokalen Kubernetes-Cluster haben, in dem ein GitOps-Operator läuft, der eine Beispielanwendung kontinuierlich aus einem Config-Repo deployt.

Bevor wir in den Code und die Praxis eintauchen, möchten wir aber zuerst ein paar Werkzeuge an die Hand geben, die für jede Situation nützlich sind, in der man GitOps implementieren will: eine grundsätzliche Empfehlung und eine Orientierungshilfe.

3.1 Agile Empfehlung: zügiger Durchstich

Die vier Prinzipien können einschüchternd wirken, weil sie absolute Standards setzen, die auf den ersten Blick keinen Spielraum lassen. Glücklicherweise ist GitOps aber nicht zuallererst ein himmelhohes Ideal, das kein gewöhnlicher Mensch erreichen kann. Stattdessen reden wir hier von einer *Reise* und einem schönen Reifeprozess. Sehr empfehlenswert zu diesem Thema ist auch der Vortrag »GitOps as a Journey« von Dan Garfield (Codefresh), Scott Rigby (Weaveworks) und Chris Short (AWS) auf der GitOpsCon 2022[1].

Wir empfehlen deshalb grundsätzlich ein *agiles Vorgehen*: Versuche nicht, sequenziell Prinzip für Prinzip lückenlos zu erfüllen. Ziele stattdessen auf einen Durchstich ab, quasi ein Minimum Viable Product (MVP), mit dem du alle vier Prinzipien minimal erfüllst. Danach bringe iterativ immer mehr Ressourcen deines Systems unter GitOps-Kontrolle.

In aller Regel hat man nämlich bereits deklarative Manifeste, die man nicht erst noch erzeugen muss. Das sind beispielsweise Kubernetes-Manifeste, Docker-Compose-Manifeste oder Terraform-Dateien. Damit ist Prinzip 1 grundlegend abgedeckt.

[1] *https://youtu.be/LQgsxT3SlN8*

Git ist sehr weit verbreitet als Versionierungstechnologie. Wenn man seine Manifeste in Git speichert, hat man Prinzip 2 bereits zu einem großen Teil erfüllt.

Den GitOps-Operator installieren

Was GitOps am meisten von anderen Deployment- und Betriebsansätzen differenziert, sind Prinzip 3 und 4: der Operator im Zielsystem, der Deklarationen kontinuierlich überwacht und anwendet. An diesem Punkt ist die erste praktische Änderung nötig. Im Kubernetes-Umfeld haben wir mit *Flux CD*[2] und *Argo CD*[3] zwei reife und etablierte Alternativen an GitOps-Tools. Wir verwenden in diesem Kapitel Argo CD als Startpunkt, weil die eingebaute UI sich gut zur Visualisierung eignet und weil Argo CD meist mehr Anklang bei Entwickelnden findet, während Flux bei Plattformbetreibern beliebter ist.

> In Kapitel 4 auf Seite 69 gehen wir tiefer auf die Unterschiede zwischen Argo CD und Flux ein und welches Tool sich für welche Anforderungen besser eignet.

Config-Repos mit dem GitOps-Operator verknüpfen

Nach der Installation muss Argo CD so konfiguriert werden, dass es die Manifeste im Git-Repo lesen kann. Dafür müssen wir wiederum ein Manifest erstellen, das dieses Git-Repo referenziert. Bei Argo CD nennt man solche Referenzen `Applications`.

> Analog zu den `Applications` bei Argo CD gibt es bei Flux verschiedene Ressourcentypen wie `Kustomizations` und `HelmReleases`.

Sobald wir dieses Referenzmanifest deployt haben, werden die vorhandenen Manifeste von Argo CD überwacht und in GitOps-Manier kontinuierlich angewandt. Ab hier sind auch alle vier Prinzipien erstmals grundlegend erfüllt. *Vollständig* erfüllt sind sie noch nicht, denn folgende Kriterien erfüllt unser Aufbau an diesem Punkt meistens noch nicht:

1. Bei Prinzip 2: Die reine Nutzung von Git erfüllt die Anforderung von Versionierung, aber nicht automatisch die von *Unveränderbarkeit*. Je nach gewünschtem Grad an Unveränderbarkeit sind weiterführende Schritte nötig wie beispielsweise das Pinnen von Image-Tags. (Wir haben bereits bei Abschnitt 1.3.2 auf Seite 18 darüber gesprochen.)
2. Ebenfalls bei Prinzip 2: Oftmals haben wir es auch mit *Secrets* zu tun, die wir nicht direkt in Git versionieren wollen. Diesem Thema wenden wir uns in Kapitel 5 auf Seite 97 zu.

---

[2] *https://fluxcd.io*

[3] *https://argoproj.github.io/cd*

Wenn wir von Vollständigkeit reden, sollten wir noch einmal im Detail Prinzip 1 untersuchen: Dieses Prinzip besagt, dass der Zustand *des von GitOps verwalteten Systems* deklarativ beschrieben sein muss. Es besagt nicht, dass die Gesamtheit des Systems, innerhalb dessen ein GitOps-Operator läuft, vollständig deklarativ beschrieben sein muss.

Prinzip 1 sagt nichts aus über den Umfang des Systems.

Es ist mit Sicherheit gut, wenn so viele Komponenten unseres Softwaresystems wie möglich von GitOps verwaltet werden, und unserer Erfahrung nach entwickelt GitOps schnell eine Sogwirkung, sodass Teams von alleine versuchen, immer mehr Teile des Systems unter GitOps-Kontrolle zu bringen. Dazu kann beispielsweise die zugrunde liegende Infrastruktur gehören (siehe Kapitel 11 auf Seite 291).

Aber Prinzip 1 legt umgekehrt fest, dass es dann erfüllt ist, wenn das System, das von GitOps verwaltet wird, deklarativ beschrieben wird. Nicht alle Bestandteile eines Systems werden wir immer deklarativ ausdrücken können oder wollen. In Kapitel 9 auf Seite 247 betrachten wir Entitäten und Aktionen, die wir möglicherweise bewusst außerhalb der GitOps-Überwachung behandeln möchten.

3.2 Fragen zur Orientierung

Als weiteres Hilfsmittel vor der praktischen Umsetzung wollen wir eine Orientierungshilfe geben, mit der man GitOps-Implementierungen einschätzen und zueinander in Bezug setzen kann.

Zum Thema GitOps finden sich Unmengen an Tutorials, Blogposts, Konferenzbeiträgen und dergleichen, die sehr detailliert eine konkrete Implementierung von GitOps vorstellen. Viele davon sind wertvoll für die Umsetzung, auch wenn manche sich leider auf veraltete Definitionen von GitOps stützen und damit den Begriff »GitOps« verwischen. Ihnen allen ist aber gemein, dass sie denjenigen nicht wirklich helfen, die GitOps und seine Prinzipien zuerst noch grundlegend kennenlernen müssen, bevor sie anfangen können mit einer Implementierung.

Mit den folgenden Fragen wollen wir Verwirrung verringern und dabei helfen, GitOps-Lösungen einzusortieren, damit wir – um das geläufige Sprichwort positiv umzuformulieren – wieder »den Wald trotz lauter Bäumen« sehen können. Sie beschränken sich auf die für GitOps relevanteren Aspekte und lassen damit bewusst manche Dimensionen aus, die für die ganzheitliche Beurteilung eines Softwaresystems relevant sein können.

1. Prinzip 1: *Was für Deklarationen nutzen wir?* Das können unterschiedlichste Dateiformate sein: Kubernetes-Manifeste, Docker-Compose-Manifeste, Terraform-Dateien, Pulumi-Code, Helm-Charts oder andere. Auch Kombinationen von verschiedenen De-

klarationstypen sind möglich, beispielsweise Terraform für Infrastruktur und Helm-Charts für Deployments.

2. Prinzip 2: *Welchen Speicher nutzen wir?* Das kann sich in der Technologie und im Hosting unterscheiden:
   - Die *Technologie* kann beispielsweise Git, Mercurial oder Fossil sein. Sie kann auch seltenere Varianten umfassen wie AWS S3 oder Registries, die mit der Spezifikation der Open Container Initiative (OCI) compliant sind (OCI-Registries).
   - Beim *Hosting* reden wir meistens von der Bereitstellung eines Git- oder Mercurial-Servers, und hier unterscheiden wir nochmals zwischen *Modus* und *Produkt*:
     - Modus: Wird es als Software-as-a-Service (SaaS) bezogen oder selbst betrieben?
     - Produkt: Nutzen wir GitLab, GitHub, Bitbucket oder ein anderes Produkt?
3. Prinzipien 3 und 4: *Welchen GitOps-Operator nutzen wir?* Typische Antworten hier sind Argo CD, Flux, Portainer oder selbst entwickelte Tools.
4. *Welche Secrets-Verwaltung nutzen wir?* (Wir sehen in Kapitel 5 auf Seite 97 genauer, warum das Verwalten von Secrets eine so wichtige Rolle spielt.) Mögliche Antworten können hier sein:
   - verschlüsseltes Speichern im Repository mit SOPS, git-crypt oder Sealed Secrets
   - ein externer Secrets Manager wie HashiCorp Vault oder AWS Secrets Manager, der in Kubernetes eingebunden wird über beispielsweise External Secrets Operator oder als Operator Injector
5. *Welche private Image-Registry nutzen wir?* Weil wir in aller Regel mit Containern arbeiten, ist es zur Orientierung hilfreich zu wissen, wo unsere Container-Images gelagert werden. Meistens lohnt es sich hier zu differenzieren zwischen *privaten und öffentlichen Images*. Öffentliche Images werden oftmals von Docker Hub oder Quay genommen, während private, selbstgebaute Images beispielsweise in der Azure Container Registry, einem selbstbetriebenen Harbor oder einer anderen privaten Registry gelagert werden. Die Antwort auf diese Frage kann sich in manchen Fällen mit der Antwort auf Frage 2 überlappen oder sogar deckungsgleich sein. Ein paar kurze Beispiele dafür: