Strategien zur Promotion von Releases in GitOps-Umgebungen

Thomas Stadler, BSc
FH-Burgenland, Eisenstadt, Österreich

KURZFASSUNG:
Aus den Prinzipien von DevOps entstand GitOps, als eine Sammlung von Prinzipien und Best Practices für die Operation von Softwaresystemen. Im Zentrum bzw. der Quelle - als einzig wahre Quelle (Single Source of Truth) - steht dabei das Git-Repository bzw. ein Revisions-Kontrollsystem. Der Zustand des zu verwaltenden Systems wird vollständig deklarativ als Code definiert. Ein GitOps-Controller kümmert sich um den kontinuierlichen Abgleich zwischen dem definierten Ziel-Zustand und dem tatsächlichen Systemzustand. Die Promotion neuer Software-Releases zwischen mehreren Umgebungen zeigt sich als derzeit offenes Problem. Es fehlen einheitliche Standardpraktiken, sowie nötige Tools in der Open-Source-Community. Diese Arbeit hat als Ziel, die Promotion von Releases in GitOps-Umgebungen zu addressieren. Es sollen abstrakte Modelle von Deployment-Umgebungen sowie Promotion-Workflows definiert werden. Aufbauend auf den zuvor definierten Modellen soll eine standardisierte Lösung zur Promotion von Releases nach den GitOps-Prinzipien implementiert, und den bestehenden Lösungen gegenübergestellt werden.

1 EINLEITUNG UND PROBLEMHINTERGRUND

Immer mehr Organisationen setzen auf eine DevOps-Kultur, um neue Applikationen und Dienste mit hoher Geschwindigkeit zu entwickeln. Denn eine Kultur, die gemeinsame Verantwortung, Transparenz und schnelles Feedback fördert, hilft dabei, die Lücken zwischen Teams zu verkleinern und damit auch Prozesse zu beschleunigen. Aus dem Bedürfnis nach schneller Innovation entstand GitOps.

GitOps ist eine Reihe von Prinzipien für den Betrieb und die Verwaltung von Softwaresystemen. Diese Prinzipien sind aus dem modernen Softwarebetrieb abgeleitet, haben aber auch ihre Wurzeln in bereits bestehenden und weithin angenommenen Best Practices. Die primären vier Prinzipien, welche als Grundsätze für den gewünschten Zustand eines von GitOps verwalteten System gelten, sind die folgenden:

- **Deklarativ**
  Bei einem von GitOps verwalteten System muss der gewünschte Zustand deklarativ ausgedrückt werden.
- **Versioniert und unveränderlich**
  Der gewünschte Zustand wird auf eine Weise gespeichert, die Unveränderlichkeit und Versionierung erzwingt und eine vollständige Versionshistorie aufbewahrt.

- **Automatisch abgerufen**
  Software-Agenten ziehen automatisch die gewünschten Zustands-Deklarationen aus der Quelle.
- **Kontinuierlich abgeglichen**
  Software-Agenten beobachten kontinuierlich den aktuellen Systemzustand und versuchen, den gewünschten Zustand herzustellen.

(opengitops.dev, 2023b)

Diese Grundsätze sind in der Version 1.0.0 von OpenGitOps festgelegt. OpenGitOps ist ein Cloud Native Computing Foundation (CNCF) Sandbox-Projekt im Rahmen der GitOps Working Group. Die GitOps Working Group ist eine Arbeitsgruppe unter dem CNCF App Delivery TAG, mit dem Ziel, eine klare herstellerneutrale, prinzipiengeleitete Bedeutung von GitOps zu definieren. Damit wird eine Grundlage für die Interoperabilität zwischen Tools, Konformität und Zertifizierung durch dauerhafte Programme, Dokumente und Code geschaffen (opengitops.dev, 2023a).

Es ist anzumerken, dass sich der Begriff GitOps nicht ausschließlich auf die in OpenGitOps definierten Grundsätze beschränkt, sondern darüber hinaus geht.

GitOps kann als eine Weiterentwicklung von Infrastructure as Code (IaC) betrachtet werden, die Git als Versionskontrollsystem für Infrastrukturkonfigurationen verwendet. GitOps senkt die Kosten für die Erstellung von Self-Service-IT-Systemen und ermöglicht Self-Service-Operationen, die bisher nicht zu rechtfertigen waren. Es verbessert die Fähigkeit, Systeme sicher zu betreiben, indem es unprivilegierten Nutzern erlaubt, große Änderungen vorzunehmen, ohne diesen Nutzern direkten Zugriff am Zielsystem zu erteilen. Über sogenannte Pull Requests können gewöhnliche Nutzer einen Antrag ihrer vorgeschlagenen Änderungen stellen. Diese Pull Requests können im Anschluss von höher privilegierten Nutzern überprüft und akzeptiert werden. Gegebenenfalls können Anpassungen vorgeschlagen, oder selbst eingearbeitet werden.

Die Sicherheit kann verbessert werden, indem automatisierte Tests hinzugefügt werden. Diese Tests können beispielsweise Linting (Validierung der Syntax), das Erzwingen von Style-Guides, das Durchführen von Unit Tests, oder das Durchsetzen von Sicherheitsrichtlinien umfassen. Sicherheitsprüfungen und Audits werden einfacher, da jede Änderung nachverfolgt wird. Jede Änderung bzw. Commit wird permanent in der Git-Historie festgehalten (Limoncelli, 2018).

GitOps als Praxis für das Releasen von Software hat mehrere Vorteile, aber wie auch andere Lösungen, hat GitOps auch mehrere Unzulänglichkeiten. Derzeit gibt es mit GitOps einige ungelöste Probleme. Einserseits sind das Probleme, welche meist schon vor der Implementierung von GitOps in einer Organisation ein Problem waren, andererseits bringt eine GitOps-Strategie einige Limitierungen mit sich, welche zu neuen Problemen führen. Die Organisation Codefresh, welche aufgrund ihrer Entwicklungen von diversen GitOps-Tools als treibende Kraft für GitOps gilt, spricht von folgenden Problemen mit dem zurzeit praktizierten GitOps:

- GitOps deckt nur eine Teilmenge des Software-Lebenszyklus ab
- Die Aufteilung von CI und CD mit GitOps ist nicht ganz einfach
- GitOps befasst sich nicht mit der Promotion von Releases zwischen Umgebungen

- Es gibt keine Standardpraxis für die Modellierung von Konfigurationen mit mehreren Umgebungen
- GitOps versagt bei automatischer Skalierung und dynamischen Ressourcen
- Es gibt keine Standardpraxis für GitOps-Rollbacks
- Die Observability für GitOps (und Git) ist unausgereift
- Auditing ist problematisch, obwohl alle Informationen in Git vorhanden sind
- Der Betrieb von GitOps in großem Maßstab ist schwierig
- GitOps und Helm arbeiten nicht immer gut zusammen
- Continuous Deployment und GitOps passen nicht zusammen
- Es gibt keine Standardpraxis für die Verwaltung von Secrets

(codefresh.io, 2023)

Die erforschten Ergebnisse, den erarbeiteten Lösungsansatz, sowie der vorgeschlagene Prototyp, können zum Ende an die CNCF gespendet werden.

2 STAND DES WISSENS / STAND DER TECHNIK

Im Folgenden werden die derzeitigen Praktiken und Best Practices gezeigt zum Thema, wie am besten verschiedene Deployment-Umgebungen mit GitOps modelliert werden.

2.1 Branch-per-Environment-Ansatz

Kapelonis (2021) veröffentlicht in einem Blogbeitrag "Stop Using Branches for Deploying to Different GitOps Environments" die Best Practices von Codefresh zum Thema der Modellierung von Umgebungen in GitOps. Es werden die Probleme des Branch-per-Environment-Ansatzes gezeigt. Dieser Ansatz sei ein Anti-Pattern, und auf jeden Fall zu vermeiden. Es werden einige Punkte angeführt und ausführlich begründet.

- Die Verwendung verschiedener Git-Branches für Deployment-Umgebungen ist ein Relikt der Vergangenheit.
- Pull Requests und Merges zwischen verschiedenen Branches sind problematisch.
- Die Versuchung ist groß, umgebungsspezifischen Code einzubinden und einen Konfigurationsdrift zu erzeugen.
- Sobald man eine große Anzahl an Umgebungen hat, gerät die Wartung aller Umgebungen schnell außer Kontrolle.
- Der Branch-per-Environment-Ansatz widerspricht dem bestehenden Kubernetes-Ökosystem.

2.2 Folder-per-Environment-Ansatz

Kapelonis (2022) liefert in einem Blogbeitrag "How to Model Your Gitops Environments and Promote Releases between Them" einige Best Practices von Codefresh zu der Problemstellung. Codefresh ist die Organisation hinter dem Argo Project (argoproj.io, 2023), welches eine Suite an Open-Source-Tools für GitOps entwickelt (Kapelonis, 2022).

Kapelonis (2022) schlägt vorerst folgende Kategorien der "Umgebungskonfiguration" vor:

- **App-Version** in Form des verwendeten Container-Tags. Das ist jene Einstellung, die sich bei jedem Release erhöht und bei jedem Rollback verringert - und immer zwischen Umgebungen promotet wird.
- **Kubernetes-spezifische Einstellungen** für die Applikation (Manifests).
- **Überwiegend statische Einstellungen** für die Geschäftslogik. Das sind z. B. externe URLs, interne Warteschlangengrößen, UI-Standardwerte, Authentifizierungsprofile usw. Mit "größtenteils statisch" sind Einstellungen gemeint, die einmal für jede Umgebung definiert werden und sich dann in der Regel nicht mehr ändern. Diese Einstellungen sollten niemals zwischen Umgebungen promotet werden.
- **Nicht-statische Geschäftseinstellungen**. Dies ist dasselbe wie der vorige Punkt, aber es umfasst Einstellungen, die sehrwohl zwischen Umgebungen promotet werden. Dabei kann es sich um eine globale Mehrwertsteuer-Einstellung, Parameter für die Empfehlungs-Engine, die verfügbaren Bitraten-Codierungen und jede andere Einstellung handeln, die für das Geschäft bzw. Unternehmen spezifisch ist.

Die App-Version und die nicht-statischen Geschäftseinstellungen gehören zwischen Umgebungen promotet. Kapelonis (2022) schlägt vor, die Konfigurationsparameter je Kategorie in einer separaten Datei zu definieren. Infolgedessen können die vier Kategorien unabhängig voneinander promotet werden. Es wird der Folder-per-Environment-Ansatz vorgeschlagen. Im Allgemeinen sind alle Promotions nur Kopiervorgänge. Im Gegensatz zum Branch-per-Environment-Ansatz kann beim Folder-per-Environment-Ansatz alles von einer beliebigen Umgebung in eine andere Umgebung verschoben werden, ohne befürchten zu müssen, dass die falschen Änderungen übernommen werden. Besonders wenn es um die Rückportierung von Konfigurationen geht, glänzt Folder-per-Environment-Ansatz, da Konfigurationen einfach sowohl "vorwärts" als auch "rückwärts" verschoben werden können, sogar zwischen nicht verwandten Umgebungen.

Szenario: Promotion der App-Version von qa zu staging-us.

```
cp envs/qa/version.yml envs/staging-us/version.yml
```

Szenario: Promotion der Applikation von prod-eu nach prod-us zusammen mit der zusätzlichen Konfiguration. Hier werden auch die Einstellungsdatei(en) kopiert.

```
cp envs/prod-eu/version.yml envs/prod-us/version.yml
cp envs/prod-eu/settings.yml envs/prod-us/settings.yml
```

Szenario: Rückportierung aller Einstellungen von qa zu integration-non-gpu.

```
cp envs/qa/settings.yml envs/integration-non-gpu/settings.yml
```

Kapelonis (2022) merkt hierbei an, dass die cp-Operationen nur zur Veranschaulichung dienen. In einer realen Anwendung würde dieser Vorgang automatisch von einem CI-System oder einem anderen Orchestrierungswerkzeug durchgeführt werden. Und je nach Umgebung sollte vielleicht zuerst ein Pull Request erstellt werden, anstatt den Ordner im Haupt-Branch direkt zu bearbeiten (Kapelonis, 2022).

Kapelonis (2022) nennt folgende Vorteile des Folder-per-Environment-Ansatzes:

- Die Reihenfolge der Commits im Git-Repository ist irrelevant. Wenn eine Datei von einem Ordner in den nächsten kopiert wird, ist die Commit-Historie egal.
- Wenn lediglich Dateien kopiert werden, wird nur genau das promotet, was gebraucht wird, und sonst nichts.
- Es müssen weder Git Cherry-Picks noch andere fortgeschrittene Git-Methoden verwendet werden, um Releases zu promoten. Es müssen lediglich Dateien kopiert werden.
- Es steht dem Benutzer frei, eine beliebige Änderung aus einer beliebigen Umgebung in eine beliebige Umgebung zu übernehmen, ohne dass es irgendwelche Einschränkungen bezüglich der richtigen "Reihenfolge" der Umgebungen gibt.
- Mit Hilfe von Diff-Operationen auf Dateien ist es leicht nachzuvollziehen, was sich zwischen den Umgebungen in allen Richtungen unterscheidet (sowohl in der Quell- als auch in der Zielumgebung und umgekehrt).

3 WISSENSCHAFTLICHE FRAGESTELLUNG

Das Ziel dieser Arbeit ist es, das Problem der Promotion von Releases zwischen verschiedenen Umgebungen in GitOps zu addressieren.

Vor allem große Organisationen verfügen für gewöhnlich über viele Nicht-Produktions- und Produktions-Umgebungen wie beispielsweise: dev, qa, staging-us, staging-eu, production-us, production-eu. In der Regel werden neue Releases automatisch in einer Umgebung wie beispielsweise qa ausgerollt - mittels einer CI-Pipeline. Nun ist es die Aufgabe, neue Änderungen, welche durch ein neues Release herbeigeführt werden, in nachfolgende bzw. andere Umgebungen zu promoten. Die aktuellen GitOps-Tools haben keine einfache Antwort auf die Frage, was der richtige Ansatz für den Prozess der Promotion ist.

Insbesondere sollen in dieser Arbeit existierende Strategien für die Lösung des Problems mit existierenden Tools erforscht werden. Des Weiteren soll ein Prototyp einer neu vorgeschlagenen Strategie entwickelt werden. Im Anschluss soll der Prototyp im Rahmen eines Laborexperiments mit bereits existierenden Strategien verglichen werden.

Um das Ziel der Arbeit zu erreichen, wurden folgende Forschungsfragen identifiziert:

- Wie kann die Promotion von Releases in GitOps-Umgebungen gestaltet werden?
  - Welche Möglichkeiten bieten bestehende Tools für die Promotion von Releases mit mehreren Deployment-Umgebungen?
  - Wie können Deployment-Umgebungen, sowie Promotion-Workflows abstrakt modelliert werden?
  - Wie kann die abstrakte Modellierung verwendet werden, um eine standardisierte Lösung zur Promotion von Releases nach den GitOps-Prinzipien zu implementieren?

4 FORSCHUNGSMETHODIK

Zuerst wird bestehende Literatur zum Thema, sowie der konkreten Forschungsfragen herangezogen. Des Weiteren werden diverse Blogbeiträge von wegweisenden Organisationen im GitOps-Umfeld analysiert und die darin empfohlenen Best Practices sowie vorgestellten Modelle näher betrachtet.

Als nächstes werden existierende Softwarelösungen, die unter Open-Source-Lizenzen zur Verfügung stehen, für den zweckmäßigen Einsatz für die vorliegende Forschung evaluiert. Es gilt herauszufinden, inwieweit die definierten Forschungsfragen mit bereits existierenden Lösungen addressiert werden können. Schließlich wird eine Baseline als State-of-the-Art definiert.

Weiters sollen abstrakte Modelle für Deployment-Umgebungen sowie Promotion-Workflows definiert werden. Diese dienen als Grundlage für die Gestaltung des Prototypen im nächsten Schritt.

Soweit möglich und sinnvoll, wird für den entwickelten Prototypen auf bestehende Toolkits und Best Practices zurückgegriffen. Es ist erstrebenswert, die Kubernetes-API sowie den GitOps-Toolkit lediglich zu erweitern, um so gut wie möglich nativ in das bestehende GitOps-Ökosystem der CNCF integriert zu sein.

Nach Houde und Hill (1997) ist ein Prototyp jede Darstellung einer Design-Idee, unabhängig vom Medium. Ein Prototyp ist etwas, das als Modell oder Inspiration für spätere Entwicklungen dient (Houde & Hill, 1997).

Der entwickelte Prototyp soll im Rahmen eines Laborexperiments den bestehenden Ansätzen zur Lösung der Problemstellung gegenübergestellt werden.

Montgomery (2017) definiert ein Experiment als einen Test oder eine Reihe von Durchläufen, wobei absichtlich Änderungen an den Eingangsvariablen vorgenommen werden, um dann Gründe für Veränderungen am Ausgangsergebnis beobachten und identifizieren zu können (Montgomery, 2017).

5 ERGEBNISSE

Die Ergebnisse der bisherigen Literaturrecherche zeigen, dass für die Promotion von Releases zwischen mehreren Umgebungen keine Standardpraxis existiert. Die bestehenden GitOps-Tools liefern

keinerlei native Lösungen dafür. Als Best Practice wird vorgeschlagen, dass die Promotion-Operationen - in Form von einfachen Datei-Kopiervorgängen - an das CI-System ausgelagert werden sollen. Der Literatur sowie den eigenen Beobachtungen zufolge, fehlt jegliche Art der Standardisierung, wenn es um die Definition von mehreren Umgebungen mit GitOps geht.

Als primäres Ergebnis dieser Arbeit ist eine Antwort auf die Forschungsfragen erwartet, indem eine Softwarelösung als Prototyp vorgestellt wird. Der entwickelte Prototyp soll optimalerweise als modulare Erweiterung zu Kubernetes dienen.

6 SCHLUSSFOLGERUNGEN UND AUSBLICK

Obwohl die Adoption von GitOps einerseits einige Vorteile mit sich bringt, werden andererseits neue Probleme erschaffen, die zuvor noch nicht existierten, oder bei zuvor gefolgten Strategien besser gelöst waren. Außerdem sind mit GitOps die Ansprüche an gute Software - wie so oft - gestiegen, und es wurden imzugedessen neue Probleme identifiziert. Diese neu identifizierten Probleme mögen sich für viele Organisationen derzeit lediglich als Luxusprobleme herausstellen. Jedoch können sich diese derzeitigen Luxusprobleme mit steigendem Wachstum einer Organisation und den damit einhergehenden steigenden Ansprüchen, als reale und akute Probleme entpuppen.

Je höher die Komplexität und der benötigte Automatisierungsgrad einer Organisation, sowie dessen Adoption von DevOps - und den Prinzipien wie Continuous Integration (CI), Continuous Delivery (CD), Continuous Deployment (CDP) - desto wichtiger wird die vollständige Automatisierung der Promotion von Releases zwischen Umgebungen.

References

argoproj.io. (2023). Get More Done with Kubernetes [(Accessed on 01/01/2023)]. https://https://argoproj.io/

codefresh.io. (2023). The pains of GitOps 1.0 [(Accessed on 01/01/2023)]. https://codefresh.io/blog/pains-gitops-1-0/

Houde, S., & Hill, C. (1997). Chapter 16 - What do Prototypes Prototype? In M. G. Helander, T. K. Landauer & P. V. Prabhu (Hrsg.), *Handbook of Human-Computer Interaction (Second Edition)* (Second Edition, S. 367–381). North-Holland. https://doi.org/https://doi.org/10.1016/B978-044481862-1.50082-0

Kapelonis, K. (2022). How to Model Your Gitops Environments and Promote Releases between Them [(Accessed on 01/01/2023)]. https://codefresh.io/blog/how-to-model-your-gitops-environments-and-promote-releases-between-them/

Kapelonis, K. (2021). Stop Using Branches for Deploying to Different GitOps Environments [(Accessed on 01/01/2023)]. https://codefresh.io/blog/stop-using-branches-deploying-different-gitops-environments/

Limoncelli, T. A. (2018). GitOps: A Path to More Self-Service IT. *Commun. ACM*, *61*(9), 38–42. https://doi.org/10.1145/3233241

Montgomery, D. C. (2017). *Design and analysis of experiments*. John wiley & sons.

opengitops.dev. (2023a). GitOps Documents v1.0.0 [(Accessed on 01/01/2023)]. https://github.com/open-gitops/documents/blob/v1.0.0/README.md

opengitops.dev. (2023b). GitOps Principles v1.0.0 [(Accessed on 01/01/2023)]. https://github.com/open-gitops/documents/blob/v1.0.0/PRINCIPLES.md