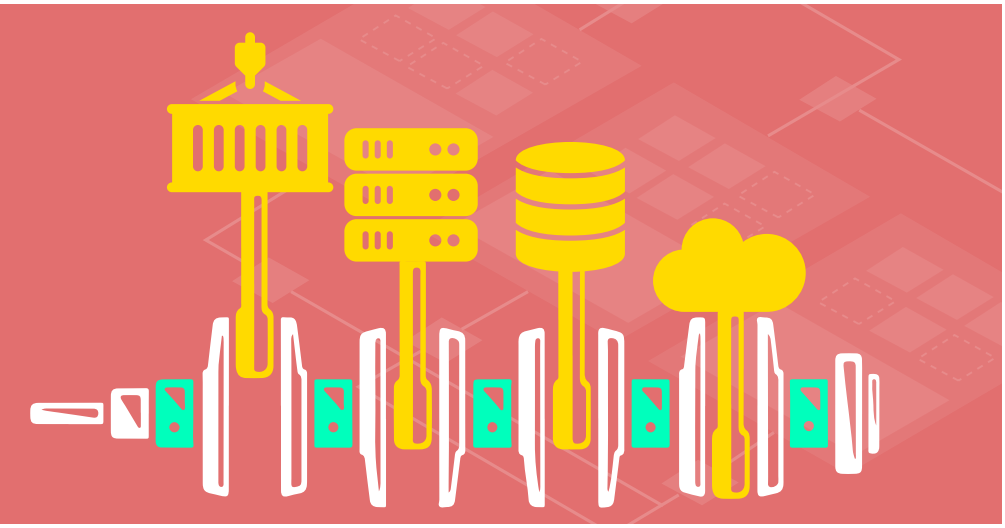


Tutorial GitOps mit Crossplane, Teil 1: Beyond CI/CD

Die Kubernetes-Erweiterung Crossplane orchestriert Ressourcen cloudübergreifend und bietet zugleich eine konsequente Implementierung der GitOps-Prinzipien. Damit kann es Tools wie Terraform zur Provisionierung von Infrastruktur ablösen.

Von Jonas Hecht

GitOps wird gern mit dem Pull-Prinzip assoziiert, doch steht dabei oft das Deployment von Anwendungen im Fokus. Die dazugehörige Provisionierung der Infrastruktur wird meist noch ganz herkömmlich per Push über die CI/CD-Pipeline gesteuert. Das liegt nicht zuletzt an den verwendeten Werkzeugen, die das GitOps-Konzept in Teilen unterlaufen. Hingegen ist Crossplane ein Cloud-Provisionierer, der im Zusammenspiel mit einem GitOps-Operator eine nahtlose GitOps-Implementierung erlaubt, in der Anwendungs-Deployment und Infrastrukturprovisionierung denselben Regeln folgen.

Was hat CI/CD mit GitOps zu tun?

Bevor sich der Begriff GitOps herausbildete, wurden meist CI/CD-Pipelines verwendet, um möglichst jeden einzelnen Prozess der Softwareentwicklung und -auslieferung zu automatisieren. Dabei besteht eine simple CI/CD-Pipeline üblicherweise aus drei Schritten, die alle den im Git-Repository abgelegten Quellcode nutzen.

Im ersten Pipelineschritt wird der Quellcode durch automatisierte Tests validiert. Im zweiten Schritt baut das CI/CD-Werkzeug den Code und verpackt ihn ausführbar in ein Container-Image, das es in die Container-Registry schiebt. Den letzten Schritt bildet das Deployment auf das Zielsystem, etwa auf einen Kubernetes-Cluster (siehe Abbildung 1).

Getriggert wird die Pipeline durch Codeänderungen im Git-Repository, die meist von den Anwendungsentwicklern kommen. Allerdings fehlt in der Darstellung etwas Wesentliches: die Infrastruktur. Denn ohne sie steht kein Deployment-Ziel für die Anwendungen bereit. Daher gilt es, den Blick auf den klassischen CI/CD-Ansatz zu erweitern.

Auch die Infrastruktur braucht CI/CD

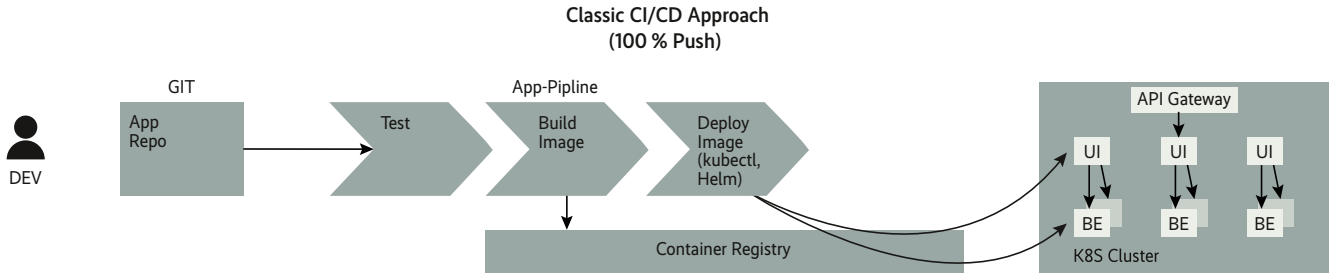
Noch sind die Zeiten nicht vorbei, in denen die Infrastruktur manuell konfiguriert wird. Das gilt auch für Cloud-Ressourcen der Hyperscaler, wenn Kunden die Infrastruktur manuell in den Frontends der Provider zusammenklicken. Glücklicherweise setzt sich aber beim Infrastrukturmanagement die Automatisierung immer weiter durch. Meist kommen auch hier CI/CD-Pipelines zum Einsatz. Das erweitert den Push-Ansatz um die Infrastrukturprovisionierung, ohne die kein Anwendungs-Deployment funktioniert.

Im Beispiel von Abbildung 2 provisioniert die Infrastrukturpipeline unter Zuhilfenahme eines Amazon EKS (Elastic Kubernetes Service) einen Kubernetes-Cluster, der nach Bereitstellung als Deployment-Ziel für die Anwendungspipeline dient. Dabei ist die Infrastrukturpipeline in drei Schritte aufgeteilt: Zuerst

Tutorialinhalt

iX-TRACT

- GitOps stellt mit seinem Pull-Prinzip traditionelle CI/CD-Pipelines auf den Kopf. Das gilt fürs Anwendungs-Deployment genauso wie für die Infrastrukturprovisionierung.
- Herkömmliche Provisionierungswerkzeuge wie Terraform, die nicht von Grund auf für den Einsatz in GitOps-Umgebungen entwickelt wurden, fügen sich hier nicht gut ein. Sie arbeiten meist mit eigenen States außerhalb der Single Source of Truth und eigenen Ressourcenbeschreibungen.
- Das auf Kubernetes aufsetzende CNCF-Projekt Crossplane arbeitet ausschließlich mit Kubernetes-Manifesten und erlaubt ein nahtloses Zusammenspiel von Infrastrukturprovisionierung und GitOps-Werkzeugen wie Argo CD.
- Durch seine Abstraktionsfähigkeiten kann Crossplane zudem Multi-Clouds provisionieren.



Eine simple CI/CD-Pipeline besteht meist aus drei Schritten (Abb. 1).

richtet sie das Netz einschließlich VPC (Virtual Private Cloud), Subnetze und Security Groups ein. Danach kommt der eigentliche Cluster inklusive Node Group und VMs dran und im dritten Schritt die Konfiguration des API-Gateways und anderer Infrastrukturkomponenten.

Das Operations- respektive Plattformteam nutzt selbstredend ebenfalls ein Git-Repository, das der Pipeline als Startpunkt dient. Git verwaltet den Code für eines oder mehrere der vielen bekannten DevOps-Werkzeuge, denn sowohl die Cloud-Provisionierer wie Terraform, dessen Open-Source-Fork OpenTofu, AWS CDK oder Pulumi als auch die Konfigurationsmanagementwerkzeuge wie Ansible nutzen das Repository als Single Source of Truth (SSoT).

Allerdings nutzen OpenTofu und Terraform ein eigenes State-Management. Dadurch existieren bereits beim klassischen Einsatz von Terraform zwei Zustände der Infrastruktur, die Admins miteinander synchronisieren müssen: der Terraform-State und der Status der Infrastruktur. Bei der Nutzung einer SSoT kommt ein dritter State ins Spiel, den es mit dem Terraform-State abzugleichen gilt. Das gilt auch für Pulumi. Besonders in einer GitOps-Umgebung kann sich das als nachteilig erweisen.

Warum eigentlich GitOps?

Man könnte versucht sein, bereits bei diesem Push-Ansatz von GitOps zu sprechen, da auch hier die Verwaltung des Infrastrukturcodes durch Git im Zentrum steht. GitOps verfolgt allerdings vier Kernprinzipien:

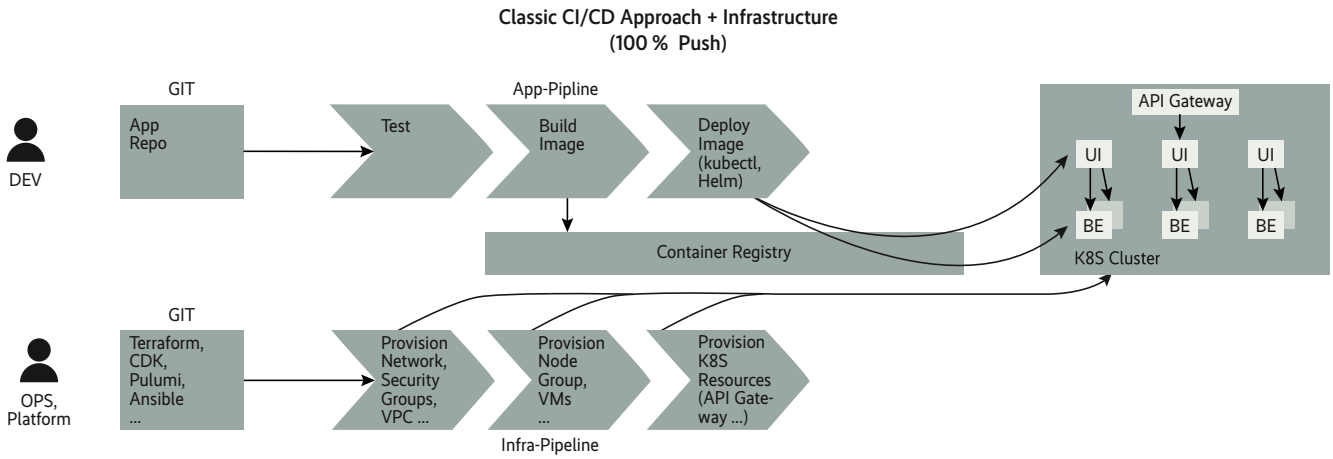
1. Deklarativ: Die Beschreibung des gewünschten Zustands geschieht auf deklarative Weise.
2. Versioniert: Die Beschreibung des Zustands wird versioniert und mit vollständiger Historie in einem Versionsverwaltungssystem abgelegt.
3. Automatisches Pulling: Softwareagenten (Operators) auf den Zielsystemen ziehen sich automatisch den gewünschten Zustand aus dem Versionsverwaltungssystem.
4. Fortlaufende Abgleichung (Reconciliation Loop): Softwareagenten beobachten den tatsächlichen Zustand fortlaufend und wenden den im Versionsverwaltungssystem definierten Zustand an.

Da bereits der klassische Push-Ansatz die Punkte 1 und 2 erfüllt, kommt es beim Begriff GitOps oft zu Verwechslungen. Was GitOps den bereits verwendeten Praktiken hinzufügt, sind die Punkte 3 und 4 Beide setzen einen Softwareagenten oder Operator voraus. Das funktioniert auch ohne Kubernetes. Ein Beispiel hierfür wäre das intensive Nutzen von Scheduled Tasks in CI/CD-Systemen. Doch gemeinhin assoziiert man GitOps mit Operatoren auf einem Kubernetes-Cluster.

GitOps im Applikations-Deployment

GitOps-Prinzipien werden heute immer öfter auf das Applikations-Deployment angewendet. Dabei kommen Tools wie Flux CD, Sveltos oder Argo CD zum Einsatz (alle genannten Tools siehe ix.de/z4rz). Diese Werkzeuge liefern den benötigten Softwareagenten, der als Operator auf einem Kubernetes-Cluster läuft. Oft verwenden Admins einen separaten Kubernetes-Cluster, Managementcluster oder Control Plane genannt. Der Operator kann nun das Anwendungs-Deployment übernehmen. Das Beispiel in Abbildung 3 nutzt Argo CD, ist aber vollständig auf die anderen Werkzeuge übertragbar.

Die Einführung von GitOps ändert auch die Anwendungspipeline. Im Beispiel der Abbildung 3 bekommt der dritte Pipelineschritt eine neue Rolle: Statt wie bisher direkt das gebaute Container-Image auf den Cluster zu deployen, committet die Pipeline nur das Tag des Con-



Der Push-Ansatz wird um die Infrastrukturprovisionierung erweitert (Abb. 2).

tainer-Images – und zwar in ein neu angelegtes separates Git-Repository, hier Deploy Repo genannt. Dieses separate Repository wird nun dem GitOps-Operator zum Pullen und fortlaufenden Abgleichen an die Hand gegeben.

Auch wenn es gängige Praxis ist, muss nicht unbedingt ein separates Git-Repository für ein lauffähiges GitOps-Set-up zum Einsatz kommen. Es gibt Anwendungsfälle, bei denen die Deployment-Konfiguration auch direkt im eigentlichen Anwendungsrepository abgelegt wird. Details dazu beschreibt der Artikel „Introduction to GitOps with Argo CD“ (siehe ix.de/z4rz).

Änderung der Verantwortlichkeiten

Ändert sich nun das Tag des Container-Images im Deploy Repo, liefert Argo CD die neue Version auf den für das Anwendungs-Deployment vorgesehenen Kubernetes-Cluster aus. Das verändert die Verantwortlichkeiten: Nicht die CI/CD-Pipeline spielt das Artefakt aus, sondern der GitOps-Operator übernimmt nun das Deployment, indem er die Anwendung auf das Zielsystem zieht. Dafür gleicht er fortlaufend den Stand in Git mit dem auf dem Anwendungscluster ab.

Dass nun der GitOps-Operator das Anwendungs-Deployment steuert, ist aber nur die halbe Miete. Denn in dem Beispiel läuft die Provisionierung der Infrastruktur noch vollständig per Push über eine CI/CD-Pipeline ab. Würde man sie auf die gleiche Weise handhaben wie das Anwendungs-Deployment, wäre sogar die Homogenisierung der Vorgehensweisen von Applikationsentwicklern und Infrastrukturverantwortlichen in greifbare Nähe gerückt. Genau diese Möglichkeit schafft die Integration von Crossplane mit einem GitOps-Werkzeug wie Argo CD.

Crossplane ist ein Cloud-natives Provisionierungswerkzeug, das vollständig auf Kubernetes aufsetzt, aber auch Nicht-Kubernetes-Infrastruktur zur Verfügung stellen kann. Es ist quelloffen, nutzt die Kubernetes-API und arbeitet rein deklarativ. Mit Crossplane nutzen Entwickler Bausteine, um die Infrastruktur zu provisionieren: Managed Resources bilden die Grundlage und werden mit Composite Resources gebündelt, um komplexere Strukturen in Form von Manifesten bereitzustellen, die Compositions genannt werden und den Plattformteams viele Freiheiten bieten. Sie abstrahieren die Ressourcen der zugrunde liegenden Cloud-Umgebung und erlauben es damit, auch Multi-Cloud-Set-ups aufzubauen.

GitOps auch in der Infrastrukturprovisionierung

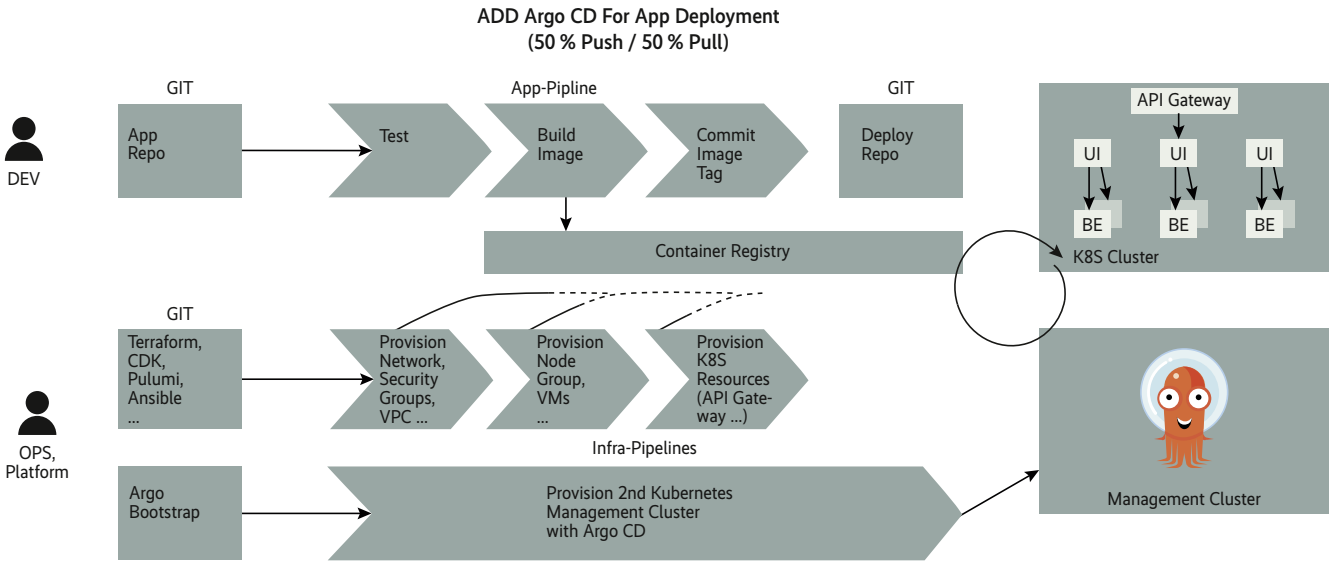
Wer die Infrastruktur in die Hände eines GitOps-Werkzeugs geben will, benötigt ein Kubernetes-natives Werkzeug für ihre Provisionierung. Denn auch Tools wie Flux, Sveltos oder Argo CD arbeiten mit Kubernetes-Manifesten, die sie innerhalb des Managementclusters applizieren. An dieser Stelle kann man also nicht ohne Weiteres einen der traditionellen Provisionierer wie Terraform oder Pulumi verwenden. Vielmehr müssen alle zu provisionierenden Infrastrukturkomponenten der Cloud-Provider als Kubernetes-Manifeste vorliegen. Neben dem cloudunabhängigen CNCF-Projekt Crossplane eignen sich dazu auch die cloudspezifischen Kubernetes-Werkzeuge der US-Hyperscaler (siehe Kasten „Crossplane-Alternativen?“)

Als Multi-Cloud-Werkzeug kapselt Crossplane die zu den Cloud-Anbietern passenden Provider und liefert alle Infrastrukturkomponenten als Kubernetes-Manifeste aus. Die einzelnen Komponenten bezeichnet Crossplane als Managed Resources. Alle verfügbaren Managed Resources sind im Upbound Marketplace zu finden.

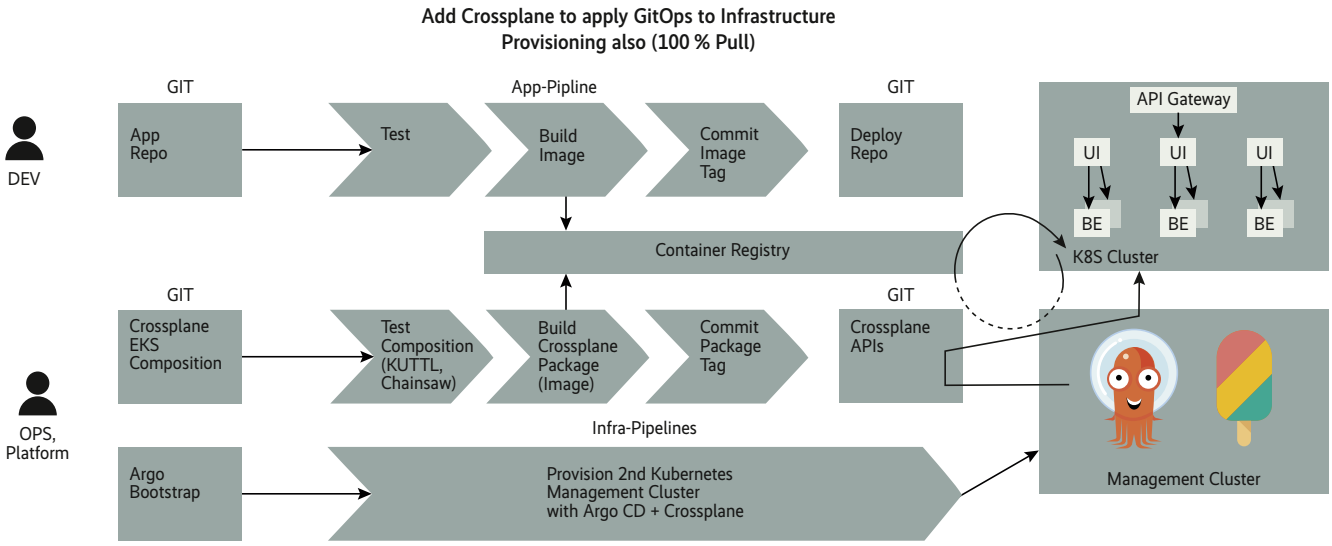
Mit den Manifesten und dem passenden Crossplane-Operator lässt sich ein Set-up aufbauen, bei dem die Infrastrukturprovisionierung ebenfalls vollständig nach den GitOps-Prinzipien funktioniert. Dabei übernimmt der GitOps-Operator eine ähnliche Rolle wie beim Applikations-Deployment und gleicht die in Git eingecheckten Manifeste mit denen im Managementcluster ab. Auf Basis der Manifeste im Managementcluster kann der Crossplane-Operator nun seinerseits

Crossplane-Alternativen?

Wer sich auf einen Cloud-Provider beschränkt, kann auch zu den Werkzeugen der Hyperscaler greifen, also den AWS Controllers for Kubernetes (ACK), dem Azure Service Operator for Kubernetes (ASO) oder dem Google Config Connector. Allerdings haben die Tools einige Nachteile gegenüber Crossplane. Erstens bieten sie nur die Ressourcenbeschreibungen des jeweiligen Cloud-Providers als Kubernetes-Manifeste an. Zweitens liefern die Tools kein Framework fürs Plattform-Engineering mit. Außerdem werden sie unterschiedlich gut gepflegt. Beispielsweise wirkt das GitHub-Repository der AWS Controllers for Kubernetes (ACK) arg vernachlässigt, während die Maintainer des Azure Service Operator for Kubernetes auf GitHub sehr aktiv zu sein scheinen.



Das GitOps-Tool Argo CD übernimmt das Anwendungs-Deployment (Abb. 3).



Mithilfe von Crossplane kann das GitOps-Werkzeug auch die Infrastrukturprovisionierung übernehmen (Abb. 4).

prüfen, ob die Infrastruktur dazu passt – und diese bei Änderungen entsprechend neu provisionieren.

Dadurch wird das GitOps-Prinzip Nr. 4, die fortlaufende Abgleichung, durch die Kombination eines GitOps-Operators mit Crossplane zu einem bidirektionalen Reconciliation Loop aufgebohrt. So entsteht eine durchgehende Kette vom Git-Repository bis zur Infrastruktur, bei der allein der deklarativ in Git eingecheckte Zustand beschreibt, wie die Infrastruktur auszusehen hat (siehe Abbildung 4).

Gleiche Entwicklungsprozesse ohne dritten State

Darüber hinaus gleichen sich auch die Entwicklungsprozesse der Applikationsentwickler und der Infrastrukturverantwortlichen an, was über die GitOps-Architektur hinaus von Vorteil ist: Beispielsweise bauen ähnliche Entwicklungsprozesse Kommunikationsbarrieren ab und fördern ein einheitliches Verständnis. Die Entwicklung von Crossplane-Manifesten, also Compositions, funktioniert ähnlich wie die Applikationsentwicklung. Dadurch sind auch die Pipelines ähnlich strukturiert. Im ersten Pipelineschritt wird die Crossplane-Composition getestet, etwa mit Testwerkzeugen wie kuttl oder Kyverno Chainsaw.

Im zweiten Schritt baut die Pipeline ein Konfigurationspaket als Container-Image, das sie ebenfalls in die Container-Registry schiebt. Wie das funktioniert, beschreibt etwa ein Artikel zum Thema auf codecentric.de (siehe ix.de/z4rz). Im dritten Pipelineschritt wird ebenfalls analog zur Applikationspipeline eine neue Package-Version – die auch einer Container-Image-Version entspricht – in ein separates Git-Repository namens Crossplane APIs committet. Argo CD behält dieses Repository zusätzlich zum Applikations-Deployment im Auge und deployt Änderungen an den Crossplane-Manifesten sogleich im Managementcluster. Hier übernimmt dann Crossplane und provisioniert die Infrastruktur entsprechend dem Stand im Git-Repository.

Damit umgeht Crossplane ein zusätzliches State-Management, wie es Terraform, OpenTofu oder Pulumi verwenden, und vermeidet daraus entstehende Inkonsistenzen. Denn hier wird der Zustand flexibel für die jeweilige Ressource des Cloud-Providers im Kubernetes-nativen Key-Value-Store etcd innerhalb des Managementclusters abgelegt. Der GitOps-Operator übernimmt dann die Verantwortung, diesen State fortlaufend mit dem im Git-Repository abgelegten zu synchronisieren. Da Crossplane seine Infrastruktur mit etcd synchronisiert, entfallen hier sämtliche Workarounds, wie sie bei Terraform, OpenTofu oder Pulumi nötig sind.

Selbstredend bleiben die CI/CD-Pipelines weiterhin relevant. Insbesondere muss das Bootstrapping des Managementclusters mit dem entsprechenden GitOps-Operator und Crossplane irgendwo automatisiert ablaufen. Die untere Pipeline in den Abbildungen 3 und 4 übernimmt genau diese Rolle.

Fazit

Die GitOps-Prinzipien haben die DevOps-Welt mit ihren klassischen CI/CD-Pipelines ordentlich durcheinandergewirbelt. Nun verantwortet nicht mehr die Pipeline das Deployment; etwas zurechtgestutzt committet der letzte Pipelineschritt nur noch neue Versionen in Git. Denn auf den Zielsystemen warten bereits die GitOps-Operatoren, um die Änderung nachzuverfolgen und zu deployen.

Diese Operatoren implementieren auch die GitOps-Prinzipien, die eine klassischen CI/CD-Pipeline nicht bietet. Dadurch bleibt der Zustand im Repository zu dem auf den Systemen jederzeit kongruent. Dabei ist es unerheblich, was es zu deployen gilt. Am weitesten verbreitet sind Applikations-Deployments, umgesetzt mit Tools wie Flux, Sveltos oder Argo CD. Bereits an dieser Stelle kommt oft ein eigener Managementcluster zum Einsatz, auf dem der GitOps-Operator läuft.

Wenn durch Crossplane auch die Infrastruktur per GitOps provisioniert wird, entsteht ein bidirektionaler Reconciliation Loop, der sicherstellt, dass die tatsächlich provisionierte Infrastruktur jederzeit dem Stand in der Single Source of Truth in Git entspricht.

Da Crossplane darüber hinaus ein Framework zum Erstellen höherwertiger Compositions anbietet, lassen sich auch komplexe Anforderungen aus dem Plattform-Engineering umsetzen. Selbst Multi-Cloud-Szenarien sind dadurch umsetzbar. Dabei folgt aber alles den GitOps-Prinzipien und bietet einen einheitlichen Entwicklungsworkflow für die Infrastrukturprovisionierung, der dem der Anwendungsentwicklung stark ähnelt. Das baut zudem die Hürden in der Kommunikation ab.

Der zweite Teil dieses Tutorials im nächsten Heft zeigt die Basisinstallation eines solchen Set-ups mit Crossplane und Argo CD. Der dritte Teil bringt zusätzlich die Crossplane-Provider sowie deren Konfiguration ins Spiel, vervollständigt das Set-up mithilfe fortgeschrittener Argo-CD-Konzepte wie App-of-Apps-Pattern und SyncWaves und macht es bereit für die eigentliche Infrastrukturprovisionierung. (sun@ix.de)

Quellen

Alle Ressourcen und Tools siehe ix.de/z4rz.

JONAS HECHT

ist Senior Solution Architect bei codecentric.