GitOps

Das Praxisbuch für DevOps-Teams und Systemarchitekten

Kapitel 2
Grundsätzliche strategische Überlegungen

»Kein Plan überlebt die erste Feindberührung.«
– Helmuth von Moltke

Veränderung ist die einzige Konstante, das postulierte schon Heraklit. Je älter man wird, desto besser kann man diesen Satz nachvollziehen, der eine umso stärkere Gewichtung in Bezug auf Systemarchitekturen in der IT hat. Jeder, der länger als drei Tage dabei ist, kennt das Problem: Das, was gestern noch *State of the Art* und superhipp war, ist heute schon wieder Schnee von gestern. Und bedingt durch immer schnellere Innovationszyklen, die sowohl immer effizienteren und noch weiter automatisierten CI/CD- und GitOps-Systemen als auch der zunehmenden KI-Unterstützung geschuldet sind, potenziert sich dies noch einmal.

Alles gut, alles richtig – keine Innovation wäre Stillstand. Aber die extrem schnelle Innovation vereinfacht nicht alles.

2.1 Entscheidungsfindung

Das Problem liegt wie so oft nicht allein in der zunehmend rasant beschleunigten Technik, die es den DevOps, MLOps, Admins, Devs, Architekten und Entscheidern oft nicht einfach macht, den Überblick zu behalten und den Impact bestimmter technologischer Innovationen auf die eigenen Infrastrukturen und Prozessabläufe richtig einzuschätzen.

In logischer Konsequenz betrifft dies, insbesondere in Bezug auf die Personalie der Architekten und Entscheider, natürlich auch die generellen strategischen Ausrichtungen der relevanten Unternehmenszweige. Und die Verantwortung dafür liegt nun einmal erfahrungsgemäß auf der mittleren und oberen Entscheidungsebene, deren Präferenzen wiederum die Strategie bestimmen.

Wie ebenfalls üblich, werden in den Prozess der Entscheidungsfindung pro oder contra eines bestimmten Technologie-Stacks neben den rein (kosten-)technischen Faktoren und Provider-/Vendor-Bindungen erfahrungsgemäß leider auch allzu oft persönliche Präferenzen eingebracht, die in einigen Fällen eher kontraproduktiv sein

können. Die auf Sicht kostspielige Liste solcher ungünstigen Entscheidungen, die viele Unternehmen in der Vergangenheit getroffen haben und auch zukünftig treffen werden, ist lang – und wird erfahrungsgemäß weiter wachsen.

2.2 Changes

In vielen Unternehmen sind Technologie-Stacks und Konzepte erfahrungsgemäß historisch gewachsen. Blickt man auf die getroffenen Entscheidungen zurück, ergibt sich recht häufig ein Muster: Eine initial getroffene Entscheidung, mehr oder weniger erfolgreich, für oder gegen eine bestimmte Software, ein Konzept oder einen Technologie-Stack. Und in den folgenden Jahren wird, selbst sofern die verwendeten Lösungen und deren Modernisierungen suboptimal waren und sind, daran festgehalten, anstatt neue, effizientere Lösungen zu suchen.

Gerade im mittelständischen Bereich ist genau dieses Muster häufig anzutreffen: Irgendwann hat eine Abteilung bzw. die Entscheider-Ebene in Eigenregie eine bestimmte, für den angedachten Job vielleicht nicht optimal geeignete Software eingeführt (oft in Ermangelung besseren Wissens) und als De-facto-Standard im Unternehmen etabliert. Darauf aufbauend wuchsen andere Tools, Konzepte und Verfahren um dieses Konstrukt herum, an dessen evolutionärem Ende irgendwann ein hochkomplexes Ökosystem mit vielen Customizations steht, das für sich genommen zwar funktional sein mag, aber im Vergleich zu anderen, besser geeigneten Lösungen deutlich abgeschlagen rangiert, wenn man seine Effizienz und Kosten betrachtet.

Eine zunehmende Anzahl von Unternehmen nutzt jedoch die Quintessenz aus diesen Erfahrungen und versteht die immer rasanteren Innovationszyklen und neuen Technologien als Herausforderung und Chance, um gegebenenfalls in der Vergangenheit getroffene, suboptimale Entscheidungen wieder in kosten-effizientere Bahnen zu lenken.

Konkretisieren wir das weiter oben beschriebene Problem auf den Scope des Buches: CI/CD- und GitOps-Systeme.

2.3 CI/CD und GitOps

Auch, wenn es Ihnen vielleicht schon bekannt sein dürfte, folgt an dieser Stelle zunächst eine kurze Klarstellung der Begrifflichkeiten und ihrer Überschneidungen.

Nicht selten werden die Begriffe *CI/CD* und *GitOps* in einen Topf geworfen. Aber:

- *Continuous Integration (CI)* steht für das schnelle, regelmäßige Zusammenführen von Codeänderungen, für automatisiertes Bauen und Testen.

- *Continuous Delivery* oder *Deployment (CD)* sorgt dafür, dass daraus entstandene Artefakte automatisiert und reproduzierbar in Zielumgebungen gelangen.
- *GitOps* geht noch einen Schritt weiter und ergänzt CI/CD: Hier wird der gesamte Zielzustand – ob Infrastruktur, Konfiguration oder Anwendung – deklarativ in einem Git-Repository festgehalten. Spezialisierte Controller gleichen kontinuierlich Soll- und Ist-Zustand ab und korrigieren Abweichungen automatisch. Der zentrale Unterschied zur rein Pipeline-getriebenen CD ist die kontinuierliche *Reconciliation*: Ein GitOps-Controller überwacht den Soll-Zustand aus Git permanent, erkennt Drifts im Cluster und setzt den deklarativen Zielzustand – Policy-gesteuert – automatisch wieder durch. Das Schöne daran ist: Git wird so zur zentralen, nachvollziehbaren Wahrheit (die *Single Source of Truth*). Jede Änderung ist versioniert, überprüfbar und jederzeit rückrollbar.

Die Schnittmenge liegt auf der Hand: CI/CD liefert geprüfte, lauffähige Artefakte, GitOps sorgt für den sauberen, auditierbaren Rollout von Applikationen oder kompletten Infrastrukturen – egal, ob in der Cloud, On-Premises oder im Edge Computing. So entsteht dank CI/CD in Verbindung mit GitOps ein durchgängiger, automatisierter Fluss vom Commit bis zum laufenden System.

2.3.1 Die Kernprinzipien von GitOps

GitOps funktioniert nur zuverlässig, wenn Sie die folgenden Prinzipien konsequent anwenden. Sie bilden das Fundament für reproduzierbare Deployments, saubere Governance und schnelle Wiederherstellung.

Git für alles

Git ist die maßgebliche Quelle der Wahrheit, die Single Source of Truth – und zwar nicht nur für Anwendungscode, sondern ebenso für Konfiguration, Applikationen bis hin zur kompletten Infrastruktur. Jede relevante Änderung liegt als deklarative Beschreibung im Repository: Infrastructure-as-Code-Manifeste, Ressourcen, Kustomize-Overlays, Helm-Values, Policies. Damit werden Änderungen nachvollziehbar, überprüfbar und revertierbar: deklarative Konfiguration auf allen Ebenen (Applikationen und Infrastruktur). So entstehen Nachvollziehbarkeit, (automatisierte) Rollback-Fähigkeit und strikte Synchronität zwischen dem gewünschten Soll-Zustand im Git und dem Live-Zustand im Cluster.

Versionierung

Deklarationen in Git sind unveränderlich versioniert und besitzen eine klare Historie. Ein einziger, zentraler Ursprungsort (die *Single Source of Truth*) steuert alles, was Ihre Anwendung ausmacht. Unterschiede zwischen Versionen sind klar über Commits

und Diffs im Versionsverlauf nachvollziehbar – mit der Garantie, dass die Plattform den in Git beschriebenen Zustand abbildet. Drift ist immer sichtbar und auditierbar, Ursachen sind zeitlich exakt zuordnungsfähig.

Pull statt Push

GitOps setzt auf Agenten bzw. Operatoren wie Argo CD im Cluster, die den deklarativen Zustand regelmäßig aus Git abrufen und mit dem Live-Zustand im Cluster abgleichen. Dieses Pull-Modell ersetzt manuelle und fehleranfällige Push-Konstrukte. Bei einem CI-seitigen Single-Shot-Apply handelt es sich nicht um echtes GitOps: ohne laufende Reconciliation findet keine automatische Drift-Erkennung und -Korrektur statt. Das Pull/Reconciliation-Modell stellt sicher, dass Sie jederzeit wissen, ob der reale Zustand dem gewünschten entspricht. Abweichungen werden erkannt und – je nach Policy – automatisch korrigiert.

Reconciliation

Der kontinuierliche Abgleich hält den gewünschten Zielzustand im Cluster deckungsgleich mit der Deklaration im Git. Dieses Reconciliation-Prinzip, das zahlreiche Controller und Operatoren unter Kubernetes nutzen und das Tools wie Argo CD ebenfalls verwenden, liefert fortlaufendes Feedback und gibt volle Kontrolle über alle Synchronisationsabläufe und deren Ausgang. GitOps-Controller schützen die Applikationsstacks in ihren Clustern vor etwaigen manuellen Fehlkonfigurationen, die (»am Git vorbei«) gegebenenfalls direkt im Live Cluster getätigt werden.

2.4 Plattformgebundene vs. generische GitOps-Tools

Plattform-spezifische CI/CD- und GitOps-Werkzeuge – wie GitHub Actions, GitLab CI/CD, BitBucket Pipelines, Azure DevOps oder AWS Code Pipeline – bieten oft einen schnellen Einstieg. Sie sind bequem in die jeweilige Plattform integriert, was anfänglich die Entwicklungs- und Deployment-Prozesse vereinfacht. Beispielsweise ermöglicht *GitHub Actions* als Teil von GitHub die Automatisierung von Workflows direkt in der bekannten Umgebung, ohne dass separate Tools nötig sind. Ähnliches gilt für *GitLab CI/CD* mit seiner vollständigen DevOps-Plattform und vorgefertigten Templates. Kurzfristig mag eine solche enge Integration attraktiv sein – Teams können rasch Pipelines aufsetzen, und vieles ist »out of the box« verfügbar.

Langfristig jedoch stellen sich strategische Fragen: Wie flexibel ist diese Lösung, wenn sich Rahmenbedingungen ändern? Hier kommt der Begriff *Vendor Lock-in* ins Spiel. Ein einfaches Beispiel:

Workflows, die speziell für GitHub Actions oder GitLab CI geschrieben wurden, lassen sich nicht ohne Weiteres in einer anderen Umgebung ausführen. Wechselt ein Unter-

nehmen etwa von GitHub zu GitLab oder in die Cloud bzw. zurück zu On-Premises (»wenn sich der Wind auf Entscheider-Ebene dreht«), müssen CI/CD-Pipelines oft aufwendig neu implementiert werden. Es ist verlockend, die Automatisierungsfunktionen eines Plattform-Anbieters zu nutzen – aber sobald man den Anbieter wechselt, fängt man in der Regel wieder bei null an. Zudem können Änderungen beim Anbieter (z. B. geänderte Preise, SLAs oder kurzfristig eingestellte Services) Projekte unerwartet treffen. Die Lektion daraus ist eindeutig: Abhängigkeiten minimieren und so weit wie möglich plattformunabhängig bleiben.

Generische, standardisierte Tools aus dem CNCF-Umfeld (*Cloud Native Computing Foundation*) oder der CDF (*Continuous Delivery Foundation*) bieten hier einen Ausweg. Diese Open-Source-Werkzeuge sind Vendor-neutral, d. h. nicht an einen bestimmten Cloud- oder Plattformanbieter gebunden. Laut CNCF-Prinzip gilt: *Build once, run anywhere* – einmal erstellt, überall ausführen. Sicher, das ist auch nur ein typischer Werbespruch, aber er trifft in der Regel zu, denn diese Tools laufen typischerweise auf allen Kubernetes-zentrischen Plattformen und funktionieren in jeder Umgebung größtenteils identisch, sei es bei verschiedenen Anbietern in der Public Cloud, in der Private Cloud oder On-Premises.

Das bedeutet konkret: Einmal erstellte Pipeline-Definitionen, Argo-Apps, AppSets, Rollouts oder sonstige GitOps-Konfigurationen lassen sich ohne größere Modifikationen auf verschiedenen Kubernetes-/OpenShift-Clustern ausführen – unabhängig davon, wo diese Cluster laufen. Und sie lassen sich bei einem Strategiewechsel des Unternehmens relativ einfach transformieren. Diese Portabilität verschafft Unternehmen langfristig betrachtet enorme Flexibilität.

2.5 Ein Überblick über die Tools

Die nächsten Seiten geben Ihnen einen Überblick über einige generische CI/CD- und GitOps-Werkzeuge, die im Scope dieses Buches behandelt werden.

2.5.1 Tekton (OpenShift Pipelines)

Tekton ist ein Kubernetes-natives CI/CD-Pipelinesystem. Pipelineabläufe werden als Kubernetes-Objekte (YAML) definiert und können in jedem konformen Cluster ausgeführt werden. Tekton ist ein Open-Source-Projekt der CDF und komplett kostenfrei nutzbar. Es vermeidet Proprietärlogik: Statt spezieller GitHub/GitLab-spezifischer Syntax nutzt man standardisierte Tasks und Pipelines. Red Hat setzt Tekton als Default-CI-Lösung im Operator-gestützten Stack *OpenShift Pipelines* ein. Und Tekton erfreut sich prominenter Kunden, wie auch in Abschnitt 5.1 detaillierter ausgeführt wird: Oscar-prämierte Visual-Effects-Studios, die Ford Motor Company, Air Canada und etliche andere mehr.

In der OpenShift/Kubernetes-Welt integriert sich Tekton nahtlos und eröffnet Möglichkeiten, die klassische Tools wie Jenkins nicht bieten konnten. Durch die Unabhängigkeit von einem einzelnen Anbieter ist Tekton langfristig stabiler – Änderungen der Plattform beeinflussen das Pipeline-System kaum.

2.5.2 Argo CD (OpenShift GitOps)

Argo CD ist ein, wenn nicht gar »das« CNCF-Tool für GitOps-basiertes Continuous Delivery. Argo CD läuft als Dienst in Kubernetes-/OpenShift-Clustern und synchronisiert den gewünschten Zustand (in Git beschrieben) mit einem Cluster oder einer gesamten Cluster Federation. Es ist plattformneutral – es kann Repositories von GitHub, GitLab oder beliebigen Git-Servern nutzen und Deployments auf jede Kubernetes-Distribution durchführen. Red Hats Operator-gestützter Stack *OpenShift GitOps* basiert auf Argo CD und integriert dieses eng in OpenShift mit vollem Enterprise-Support. Da Argo CD ein weit verbreiteter Open-Source-Standard ist, genießen Nutzer eine große Community-Unterstützung. Laut CNCF war Argo CD im betrachteten Stand die GitOps-Lösung mit der weitesten Verbreitung, was seine Zuverlässigkeit und breite Akzeptanz unterstreicht.

2.5.3 Argo Rollouts

Argo Rollouts sind eine Erweiterung aus dem Argo-Projekt für Progressive Delivery (Blue-Green-, Canary-Deployments und mehr). Es arbeitet mit Kubernetes-*Custom Resource Definitions* (CRDs) und lässt sich unabhängig von der restlichen Argo-Suite einsetzen. Dadurch kann man fortgeschrittene Deployment-Strategien umsetzen, ohne auf plattformspezifische Dienste (z. B. proprietäre Cloud-Deployer) angewiesen zu sein. Argo Rollouts funktioniert auf jedem Kubernetes-Cluster gleich – egal ob als Managed Service in der Cloud oder selbst gehostet.

2.5.4 Cluster API

Die *Cluster API* ist ein Open-Source-Framework (initiiert in der Kubernetes-Community) zur einheitlichen Verwaltung von Kubernetes-Clustern über verschiedene Infrastrukturen hinweg. Mit ihm können Ops-Teams Kubernetes-Cluster auf AWS, Azure, vSphere, OpenStack usw. nach dem gleichen Schema erstellen und verwalten. Strategisch bedeutet das: Wechselt die Infrastruktur (Cloud zu on-prem oder zu einer anderen Cloud), bleibt die Cluster-Lifecycle-Methodik gleich. Cluster API abstrahiert die Unterschiede der Anbieter, was Konsistenz und Portabilität fördert – ein weiterer Schutz vor Vendor-Lock-in.

2.5.5 Crossplane

Crossplane ist ein CNCF-Projekt, das einen Multicloud-Control-Plane-Ansatz verfolgt. Crossplane bietet eine Kubernetes-ähnliche Schnittstelle, um Cloud-Ressourcen (Datenbanken, Buckets, DNS etc.) bei verschiedenen Anbietern bereitzustellen und zu managen. Über Pluggable Provider kann Crossplane AWS-, Azure-, GCP-Ressourcen u.v.m. ansteuern. Wichtig aus strategischer Sicht: Unternehmen können ihre Infrastrukturdefinitionen einmalig in Kubernetes-Manifesten modellieren und bei Bedarf den Cloud-Provider austauschen, ohne die Modelllogik komplett neu schreiben zu müssen. Insofern kann Crossplane als Schlüsseltechnologie für Hybrid- und Multi-Cloud-Strategien angesehen werden, da es ermöglicht, jederzeit zwischen Cloud-Anbietern zu wechseln, um einen Vendor Lock-in zu vermeiden. So bleibt die Hoheit über die Infrastruktur beim Unternehmen – nicht beim Cloud-Anbieter.

2.5.6 Security und Compliance

Des Weiteren werden in fast allen Themen und Kapiteln auch die im Unternehmensumfeld extrem wichtigen Aspekte hinsichtlich *Security* und *Compliance* erörtert. Dabei kommen Tools wie *Tekton Chains*, *Vault*, *Kyverno*, *OPA Gatekeeper* und andere selbstverständlich ins Spiel, zusammen mit begleitenden Konzepten, Paradigmen und Best Practices.

2.5.7 Das strategische Gesamtbild

Diese generischen Tools eint, dass sie offene Standards und eine breite Community-Basis mitbringen. Sie werden von unabhängigen Stiftungen (CNCF, CDF) gefördert, was ihre Weiterentwicklung und Interoperabilität sicherstellt. Insbesondere in Verbindung mit Kubernetes/OpenShift spielt das eine große Rolle: Die Innovationszyklen im Kubernetes-Ökosystem werden gemeinschaftlich vorangetrieben, wodurch sich solche Tools über die Zeit als De-facto-Standards etablieren. Firmen wie IBM/Red Hat, Google und andere unterstützen diese Projekte aktiv – OpenShift integriert Tekton und Argo CD und bietet dafür langfristigen Support. Mit anderen Worten: Auch in einer Enterprise-LTS-Umgebung wie OpenShift können moderne CI/CD- und GitOps-Praktiken implementiert werden, ohne sich in eine Sackgasse proprietärer Lösungen zu manövrieren.

Für Entscheider und Architekten ergibt sich daraus ein wichtiger strategischer Vorteil: Zukunftssicherheit. Setzt man auf generische, plattformunabhängige Werkzeuge, können technische Richtungswechsel (sei es durch neue Geschäftsstrategien, Mergers oder regulatorische Anforderungen) deutlich flexibler umgesetzt werden. Die Teams haben ihre Automatisierungs-Pipelines und GitOps-Prozesse in einer Form, die überall lauffähig ist – das reduziert die Reibungsverluste bei einem Wechsel der Plattform erheblich.

2.5.8 Fazit

Plattformgebundene CI/CD- und GitOps-Lösungen mögen kurzfristig bequem sein, doch generische, Kubernetes-native Tools sind langfristig meist die bessere Alternative. Sie fördern Portabilität, verhindern den Lock-in und passen zu Multi-Cloud- oder Hybrid-Strategien. Damit unterstützen sie direkt das agile Prinzip, auf Veränderungen schnell reagieren zu können. Ganz im Sinne Heraklits – Veränderung ist die einzige Konstante – erlaubt ein offener Tool-Stack dem Unternehmen, den Wandel proaktiv zu gestalten, anstatt von proprietären Entscheidungen einzelner Anbieter getrieben zu werden. Generische Tools bieten die robuste Grundlage, um Innovation ohne unnötige Kompromisse voranzutreiben – selbst dann, wenn sich die betrieblichen Rahmenbedingungen künftig grundlegend ändern.

Unter dem Strich machen standardisierte GitOps- und CI/CD-Lösungen das Technologie-Ökosystem eines Unternehmens robuster gegenüber Wandel – sei er technisch getrieben oder durch Management-Entscheidungen verursacht. Sie sind damit ein wesentlicher Baustein einer nachhaltigen DevOps-Strategie, die auch in fünf oder zehn Jahren noch trägt, egal wohin die Reise in der Zwischenzeit geht. Generische Tools sind also nicht *nice to have*, sondern eine zukunftsorientierte Notwendigkeit, um kostspielige Kurswechsel und Sackgassen zu vermeiden.

Wie eingangs erklärt und hier noch einmal aufgegriffen: Nichts ist so beständig wie die Veränderung – und mit den richtigen Werkzeugen in der Toolchain bleiben Sie für diese Veränderung gerüstet.

Ein elementar wichtiges Konzept, das Preflight für alle folgenden Betrachtungen ist und die Foundation für fast alle zuvor beschriebenen Tools und Stacks darstellt und zudem für maximale Automation und Resilienz ebendieser Tools sorgt, schauen wir uns nun an: *Operatoren*.

```
sha256:c9235833a89975273e7867a752ce5173ab0ca7de03c01034167b3c41d1fb9cdb
[skopeo-copy : skopeo-copy] Copying config
sha256:ba5de7dcbe6be7eb9927a54ba437cdc750365766de005c589bca35d072f04cb4
[skopeo-copy : skopeo-copy] Writing manifest to image destination
[skopeo-copy : skopeo-copy] Image: docker://image-registry.openshift-image-registry.svc:5000/
picalc/picalc:1.0 successfully copied from internal to staging registry docker://gcr.io/
cluster-01-999999/staging/picalc-qa:1.0
```

Tipp: Chains und Policy Gates

Um dieses Setup um eine weitere Sicherheitsebene zu ergänzen, können Sie – wie in den Beispielen ab Abschnitt 5.14 noch beschrieben wird – alle TaskRuns und den PR per *Tekton Chains* signieren und attestieren lassen. Des Weiteren können Sie mithilfe von *Kyverno* (ab Abschnitt 7.2), einer passenden Kyverno-ClusterPolicy und einem entsprechenden Verify-SLSA-Task dafür sorgen, dass Skopeo nur Images in den Staging-Bereich kopiert, die eine lückenlose Chain vorweisen können. Entsprechende Manifeste dazu finden Sie in den Beispieldaten.

5.12 Pipeline mit Checks (SonarQube)

Auch SonarQube können Sie in die Pipeline integrieren.

5.12.1 Beispiel-Pipeline: pipeline-golang-sonarqube-trivy

In den Beispieldaten findet sich ebenfalls eine Beispiel-Pipeline mit einer modifizierten SonarQube-Task-Implementierung (Ordner: *../tekton/sonarqube*). Das Besondere an dieser Variante ist, dass die Check-Tasks mit einem externen Scan-Server kommunizieren, der zuvor eingerichtet werden muss. Zudem muss der Zugriff (der SonarQube-Tasks) auf diesen Scan-Server per Zugriffstoken sichergestellt werden.

Achtung: Das SonarQube-Setup unterscheidet sich unter OpenShift von dem der Kubernetes-Variante vor allem im RBAC/SCC-Teil.

Diese Pipeline nutzt das Repo *https://github.com/k8strainer/simple-go-app.git*, das durch eine eigene Test-Unit (`go_test.go`) in der Lage ist, den für den SonarQube-Scan benötigten Report *coverage.out* bzw. *sonar-output.txt* via `golang-test`-Task) zu generieren.

Die Beispiel-Pipeline mit dem Namen `pipeline-golang-sonarqube-trivy` (Datei: *sq-pipeline.yaml*) wird standardmäßig im Namespace `default` ausgerollt. Die beiden SonarQube-Tasks (`sonarqube-scan` in der Datei *sonarqube-task.yaml* sowie der Task `check-quality-gate` in der Datei *sonarqube-qualitygate-task.yaml*) benötigen eine lokale SonarQube-Installation, die im Namespace `sonarqube` erfolgt. Das Setup ist in ähnlicher Form in Abschnitt 5.8.3 beschrieben.

5.12.2 Pipeline-Tasks und Parameter

Tekton-Task-Name	Task-Name (Pipeline)	Ausführung nach Task	Stop on Error	Result ([cre]ator / [con]sumer)	Aufgabe
git-clone	clone-repo		yes		clone Git-Repo
golang-test	golang-test	clone-repo	yes		Go-Tests
trivy-scanner (fs)	trivy-scan-local-fs	clone-repo	yes		Scan files (missing / wrong params etc.)
sonarqube-scan	sonarqube-scan	golang-test	yes	SONAR_TASK_ID (cre)	Scan code
check-quality-gate	check-quality-gate	trivy-scan-local-fs, sonarqube-scan	yes	SONAR_TASK_ID (con)	Quality gate for sonarqube-scan
buildah	source-to-image	golang-test, trivy-scan-local-fs	yes	IMAGE_DIGEST (cre)	Build Image & push to Test-Registry
trivy-scanner (image)	trivy-scan-image	build-image	yes		Check image for CVEs
deploy-using-kubectl	deploy-to-cluster	kube-linter	yes	IMAGE_DIGEST (con)	deploy manifest to cluster

Tabelle 5.10 Tasks

Pipeline-Name	Tasks (geordnet)	PipelineRun-Parameter	Workspace
pipeline-golang-sonarqube-trivy	git-clone golang-test trivy-scanner (fs) sonarqube-scan check-quality-gate buildah trivy-scanner (cve) deploy-using-kubectl	gitUrl (git-clone) gitRevision (git-clone) pathToYamlFile (trivy, deploy-using-kubectl) image (buildah) PROJECT_KEY (sonarqube) ARGS (trivy) IMAGEARGS (trivy)	pipeline-pvc

Tabelle 5.11 Pipeline und Run

5.12.3 Das SonarQube-Setup

Ein wichtiges Preflight ist die Installation des SonarQube-Stacks im Namespace `sonarqube`. Alle dazu benötigten Daten finden Sie in den Dateien *sonarqube-deployment.yaml*, *import_token.sh* und *add_rbac.sh*. Sie können das Setup auch komplett automatisiert mit dem Script *setup-sonarqube.sh* ausführen.

- *sonarqube-deployment.yaml* – enthält den kompletten SonarQube-Stack: die SonarQube-App, das Postgres-DB-Backend, den PVC, das Secret für den Zugriff von SonarQube auf Postgres, das Sysctl-DaemonSet für Elastic-DB-spezifische Settings auf den Nodes, den Service und die Route.
- *add_rbac.sh* – (nur OpenShift) Setzt über den SA `pipeline` erweiterte Privilegien, die für den SonarQube-Betrieb erforderlich sind.
- *import_token.sh* – Erzeugt das (für den Zugriff des Tasks SonarQube-Scan auf SonarQube) erforderliche Token in Form eines Secrets in dem Namespace, in dem die Pipeline betrieben wird.
- Für den Task Build-and-Push installieren Sie den SA `pipeline-account` (Datei: *pipeline-account.yaml*) sowie das assoziierte `ibm-registry-secret` (siehe das Beispiel zur Simple-Pipeline).

Installieren Sie zunächst das SonarQube- und PostgreSQL-Deployment (*sonarqube-deployment.yaml*), das die benötigten Ressourcen im Namespace `sonarqube` und `kube-system` erzeugt.

```
# k create ns sonarqube

namespace/sonarqube created

# k create -f sonarqube-deployment.yaml

persistentvolumeclaim/postgresql-pvc created
persistentvolumeclaim/sonarqube-pvc created
secret/postgresql-secret created
deployment.apps/postgresql created
service/postgresql created
deployment.apps/sonarqube created
daemonset.apps/sysctl-vm-max-map-count created
service/sonarqube created
ingress.networking.k8s.io/sonarqube-ingress created
```

Die Beispieldatei enthält PVCs zur persistenten Datenspeicherung für Postgres und SonarQube, ebenso ein Secret, das die Kommunikation von SonarQube mit Postgres ermöglicht. Ausgerollt wird zudem ein DaemonSet im Namespace `kube-system`, das die für Elastic benötigten sysctl-Anpassungen (**`sysctl -w vm.max_map_count=262144`**) vornimmt. Des Weiteren wird ein Ingress (im folgenden Beispiel: `sonarqube.gke1.it-concepts.cloud`) bereitgestellt, über den Sie auf die SonarQube-UI zugreifen.

5.12.4 SonarQube-Management über die UI

Im Folgenden wird unter anderem beschrieben, wie Sie SonarQube über die UI einrichten können, um z. B. mit angepassten oder selbst erstellten Quality-Gates arbeiten zu können.

SonarQube UI Default Admin Credentials

Die Default-Admin-Credentials der SonarQube-UI sind User: `admin`, Passwort: `admin`, und müssen beim ersten Login zwingend geändert werden.

Nachdem Sie sich auf der SonarQube-UI angemeldet haben, erzeugen Sie ein neues Projekt (Create Local Project). Im folgenden Beispiel heißt es `test`.

`test` ist dann auch gleichzeitig der (SonarQube-)Projekt-Key, der für den PipelineRun benötigt wird. In dem Dialog können Sie auch gleich den Branch angeben (Default: `main`), der eingestellt werden soll. Im nächsten Schritt (Set up project for Clean as You Code) können Sie projektspezifische Check-Settings vornehmen. Für einen einfachen Test reichen Global Settings. Dann wählen Sie Create Project.

Im nächsten Schritt werden Sie zu der Definition der gewünschten Analysis Method aufgefordert. Wählen Sie für Tekton dort Other CI. Erzeugen Sie ein neues Token (z. B. `test_token`) ohne Ablaufdatum per Generate. Sichern Sie den Wert des erzeugten Tokens (z. B. `sqp_fcfca70589f1d00dd973e60927218552c4942a8e`) in einer Datei. Dieses Token wird im Folgenden benötigt, um ein Kubernetes-Secret zu generieren.

Im nächsten Schritt (Analyze your Project) können Sie sich basierend auf OS, Programmiersprache und Architektur die passenden Scanner-Settings anzeigen lassen. Diese können verwendet werden, um z. B. den Default-SonarQube-Task aus dem Tekton Hub über entsprechende Parameter via PipelineRun anzupassen.

Hinweis: Quality-Gates

Sie können für Ihr Projekt entsprechende Quality-Gates definieren und als Default setzen. Werden diese Quality-Gates nicht erfüllt (z. B. Code-Test-Coverage < 99 %) kann ein Task, der diese Werte auslöst, das Beenden des PipelineRuns auslösen.

Erzeugen Sie nun das Secret mit dem zuvor gespeicherten SonarQube-Token in dem Namespace, in dem die Pipeline mit dem SonarQube-Task laufen soll, z. B.:

```
# kubectl create secret generic sonarqube-token --from-literal=SONAR_TOKEN="sqp_
90543510c27f781abdef670357a622dcb0be24c8" -n sonarqube-pipeline
```

```
secret/sonarqube-token created
```

Dieses Secret wird benötigt, damit die SonarQube-Tasks auf den SonarQube-Server zugreifen können.

Quality-Gates

Um z. B. fehlschlagende Quality-Gates zu erzeugen, richten Sie ein neues Quality-Gate für das Project mit extrem restriktiven Einstellungen ein. Das geht über Quality Gates • Create, dann Unlock Editing • Add Condition. Setzen Sie dort Werte für On Overall Code, nicht nur NEW Code, z. B.: `"Issues greater than 0"`, `"Minor Issues is greater than 0"` oder auch `"Coverage is less than 99%"`.

> **Achtung: Quality-Gate-Settings**
>
> Bedingt durch hohe Volatilität ändern sich die Vorgehensweisen zum Einstellen der Quality-Gates, also welches Projekt welches Quality-Gate nutzen soll, leider häufiger. Im Stand 02/2025 gehen Sie auf der Startseite auf das Dropdown-Feld Project Settings (oben rechts) und wählen dort Quality Gates aus. Dort können Sie dann einstellen, welches bereits erzeugte Projekt ein bestimmtes Quality-Gate nutzen soll. Alternativ gehen Sie über Quality Gates • Test • ganz nach unten zu • Projects • All und setzen den Haken bei Test.

Die verschärften Konditionen sollten beim nächsten PipelineRun im Task `check-quality-gate` als Fehler angezeigt werden.

Nun stellt sich aber die Frage: Warum ein separater Task für den Quality-Gate-Check? Man kann doch auch mit dem Flag `-Dsonar.qualitygate.wait=true` im Scanner den Status des Quality-Gates setzen? Richtig, aber die hier verwendete Methode mit gesplitteten Tasks hat ein paar Vorteile:

- In Tekton-Pipelines brauchen wir oft eindeutige und abfragbare Zwischenergebnisse (wie hier unter SonarQube `taskId` oder `analysisId`), um mit diesen in gegebenenfalls nachfolgenden Tasks zu arbeiten. Der direkte Einsatz von `-Dsonar.qualitygate.wait=true` stellt nicht sicher, dass die notwendigen Informationen für den Quality-Gate-Check bereitgestellt werden.
- Zudem läuft in der Standardvariante der SonarQube-Scan-Task (`wait`) im Vordergrund. Je nach Scangröße und -dauer kann dies auf Tekton-Seite zu Timeouts führen. In der Variante mit gesplitteten Scan- und Quality-Gate/Scan-Analyse-Tasks wird das Verfahren logisch entkoppelt. Der Task `sonarqube-scan` übergibt alles an den SonarQube-Server und beendet sich dann, nachdem er die Task-ID empfangen und diese als *Result* hinterlegt hat (die Details dazu folgen in Abschnitt 5.12.5). Der Task `check-quality-gate` prüft nun einfach intervallbasiert, bis das Ergebnis aller Scans und Quality-Gates für den Scan mit der relevanten Task-ID durchgelaufen ist, und meldet sich dann mit einem detaillierten Ergebnis zurück.

Ein eigenständiger Quality-Gate-Check (wie mit `check-quality-gate`) gibt also mehr Kontrolle und Flexibilität, um eine mögliche Asynchronität von Scan und Quality-Gate-Auswertung zu überprüfen und anzupassen:

- Die Implementierung des Tasks `check-quality-gate` gibt Ihnen mehr Kontrolle darüber, wie und wann der Quality-Gate-Status geprüft wird, und ermöglicht eine klare Fehlerbehandlung, insbesondere bei asynchronen Prozessen.
- **Eindeutige Übergabe von IDs** – Der manuelle Prozess, erst die `taskId` und dann die `analysisId` abzurufen, stellt sicher, dass jede Analyse genau überwacht wird und keine ungenaue Zuordnung stattfindet.
- **Bessere Tekton-Integration** – Tekton ist sehr auf modulare Tasks und klare Resultate angewiesen. Die Verwendung von Ergebnissen (*Results*) ermöglicht es, den Pipeline-Fluss robust und nachvollziehbar zu gestalten. Das Warten im Scanner ist weniger flexibel für eine Umgebung, die verschiedene Tasks abhängig voneinander betreibt.

Andersherum gefragt: Wann wäre der Test `-Dsonar.qualitygate.wait=true` sinnvoll bzw. ausreichend? Beispielsweise wenn

- kein komplexes CI/CD-Setup benötigt wird, sondern der Status der Analyse direkt nach dem Scan verfügbar sein soll,
- der SonarQube-Scanner direkt ohne nachfolgende Tasks verwendet wird, die von einer `taskId` oder weiteren Analyseergebnissen abhängen, oder wenn
- eine einfache Jenkins- oder CI/CD-Integration genutzt wird, wo das Resultat direkt und unmittelbar als erfolgreich oder fehlgeschlagen behandelt wird.

5.12.5 Result-Übergabe zwischen SonarQube- und Quality-Gate-Check-Task

Durch die Splittung der Verfahren ist es nötig, verschiedene Parameter, unter anderem die SonarQube-Task-ID, vom SonarQube-Task an den Task `check-quality-gate` per `Result` zu übergeben. Dies geschieht wie folgt: Der SonarQube-Task wertet unter anderem den (zuvor vom `golang-test`-Task erzeugten) *Code-Coverage Report* `coverage.out` aus (siehe dazu die Erläuterungen im nächsten Unterabschnitt), der auf dem Workspace zwischengespeichert wurde.

Da das Processing aller Daten im Hintergrund gegebenenfalls noch nicht abgeschlossen ist, wenn der Task `sonarqube-scan` bereits beendet ist, muss gewährleistet sein, das Folgetasks (wie der Task `check-quality-gate`) die Reports des Scans auswerten können. Dazu schreibt der SonarQube-Task die `SONAR_TASK_ID`, die er aus dem Report extrahiert, in ein Result:

```
# sonarqube-task.yaml
[…]
SONAR_TASK_ID=$(grep -oP 'id=\K[0-9a-fA-F-]+' sonar-output.txt || echo "")
echo -n "$SONAR_TASK_ID" > $(results.SONAR_TASK_ID.path)
```

Die Pipeline übernimmt dieses Result und setzt es automatisch als (Start-)Parameter für den Task `check-quality-gate`:

```
# sq-pipeline.yaml
[...]
    - name: check-quality-gate
      [...]
      taskRef:
        name: check-quality-gate
      workspaces:
        - name: source
          workspace: pipeline-pvc
      params:
        - name: SONAR_HOST_URL
          value: $(params.SONAR_HOST_URL)
        - name: SONAR_TASK_ID
          value: $(tasks.sonarqube-scan.results.SONAR_TASK_ID)
        - name: SONAR_TOKEN
          value: "$(params.SONAR_TOKEN)"
```

Im Task `check-quality-gate` kann nun direkt auf diesen Parameter (`params.SONAR_TASK_ID`) zugegriffen werden:

```
# sonarqube-qualitygate-task.yaml
[…]
   # Überprüfen des Status der Analyseaufgabe
   while ( [ "$STATUS" == "PENDING" ] || [ "$STATUS" == "IN_PROGRESS" ] ) && [
"$TIME_WAITED" -lt "$MAX_WAIT_TIME" ]; do
       echo "Checking Quality Gate status..."
       API_RESPONSE=$(curl -s -u $SONAR_TOKEN: "$(params.SONAR_HOST_URL)/api/
ce/task?id=$(params.SONAR_TASK_ID)")
       echo "API Response: $API_RESPONSE"
       STATUS=$(echo $API_RESPONSE | jq -r '.task.status')
[…]
```

Nachstehend sehen Sie die Logs eines fehlgeschlagenen Tasks `check-quality-gate`, für den die Bedingungen verschärft wurden (0 Errors on Overall Code, 99 % Code Coverage etc.):

```
Checking Quality Gate status...
API Response: {"task":{"id":"536eb8a2-86a8-469b-95cb-c9e559121091",
"type":"REPORT","componentId":"ce5dcd2f-d155-4236-b5a9-49a64260e5c6",
"componentKey":"test","componentName":"test","componentQualifier":"TRK",
"analysisId":"eec15f76-f9d4-4a5d-8fd0-500ca94ce6bb",
"status":"SUCCESS","submittedAt":"2024-11-25T21:44:21+0000",
```

```
"submitterLogin":"admin","startedAt":"2024-11-25T21:44:22+0000",
"executedAt":"2024-11-25T21:44:24+0000","executionTimeMs":2116,
"hasScannerContext":true,"warningCount":0,"warnings":[],"infoMessages":[]}}
Analysis completed. Checking Quality Gate status...
Quality Gate API Response: {"projectStatus":{"status":"ERROR","conditions":
[{"status":"OK","metricKey":"minor_violations","comparator":"GT",
"errorThreshold":"0","actualValue":"0"},{"status":"OK","metricKey":
"new_violations","comparator":"GT","errorThreshold":"0","actualValue":"0"},
{"status":"ERROR","metricKey":"violations","comparator":"GT",
"errorThreshold":"0","actualValue":"3"}],"ignoredConditions":false,"period":
{"mode":"PREVIOUS_VERSION","date":"2024-11-25T15:42:40+0000"},
"caycStatus":"over-compliant"}}
Quality Gate failed!
```

Wozu wird die Datei coverage.out benötigt?

Die Datei *coverage.out* ist der Testabdeckungsbericht (*Code Coverage Report*). Sie enthält die Information bzw. das Protokoll, welche Zeilen des Go-Codes während der Ausführung der Unit-Tests durchlaufen wurden und welche nicht. Der Check-Prozess in unserer Pipeline ist wie folgt:

- **Erzeugung im `golang-test`-Task** – In diesem Task wird der Befehl `go test` mit speziellen Flags ausgeführt. Eines dieser Flags ist `-coverprofile=coverage.out`. Dieses Flag weist das Go-Testwerkzeug an, während der Testausführung genau mitzuprotokollieren, welche Code-Blöcke im Rahmen der Unit-Tests erfolgreich ausgeführt werden konnten. Das Ergebnis wird in die Datei *coverage.out* geschrieben, die dann im Workspace für nachfolgende Tasks bereitgestellt wird.
- **Verwendung im `sonarqube-scan`-Task** – Der SonarQube-Scanner wird mit dem Parameter `-Dsonar.go.coverage.reportPaths=$(workspaces.source.path)/coverage.out` aufgerufen. Damit wird SonarQube mitgeteilt: »Für die statische Analyse, die du jetzt durchführst, findest du hier den passenden Bericht zur Testabdeckung.«

  SonarQube liest diese Datei ein, verarbeitet die Daten und zeigt sie z. B. in der Weboberfläche an. Sie sehen dann Metriken wie »85 % der Zeilen sind durch Tests abgedeckt«. Diese Metrik kann auch als Bedingung im Quality-Gate verwendet werden (z. B. »Die Code-Coverage darf nicht kleiner als 95 % sein.«).

Die *coverage.out* ist die Brücke, die es SonarQube ermöglicht, die Ergebnisse der Go-Unit-Tests zu verstehen und in seine Analyse zu integrieren. Ohne sie wüsste SonarQube nichts über die Qualität der Testabdeckung. Der `sonarqube-scan`-Task prüft deshalb zu Beginn explizit, ob diese Datei vorhanden ist, und bricht ab, falls sie fehlt.

Warum kann SonarQube die Code-Coverage nicht selbst erstellen?

Die Antwort liegt in dem fundamentalen Unterschied zwischen statischer und dynamischer Code-Analyse:

- **Statische Analyse** (das Metier von SonarQube) – SonarQube ist ein Werkzeug für die statische Analyse. Es liest den Quellcode, ohne ihn auszuführen. Dabei findet es Probleme wie Code-Smells (unschöner Code), potenzielle Bugs (z. B. eine Variable, die null sein könnte) und Sicherheitslücken (z. B. hartcodierte Passwörter). Es versteht die Struktur des Codes, aber nicht sein Verhalten zur Laufzeit.
- **Dynamische Analyse** (erfordert Testausführung) – Die Code-Coverage ist eine Metrik der dynamischen Analyse. Um herauszufinden, welche Codezeilen abgedeckt sind, müssen die Tests tatsächlich kompiliert und ausgeführt werden. Man misst, welche Pfade im Code während dieser Ausführung durchlaufen werden.

Der SonarQube-Scanner ist kein Test-Runner. Er ist nicht dafür gebaut, eine sprachspezifische Testumgebung (wie die von Go, Java, Python etc.) aufzusetzen und die Tests auszuführen. Das ist die Aufgabe der jeweiligen Build- und Test-Werkzeuge der Programmiersprache.

Deshalb ist der Prozess branchenweit und sprachunabhängig fast immer zweigeteilt:

1. **Ein Test-Tool** führt die Tests aus: `go test` für Go, Maven/Gradle mit JaCoCo für Java, `pytest-cov` für Python etc. Dieses Tool erzeugt den sprachspezifischen Coverage-Report.
2. **SonarQube** importiert den Report: Der SonarQube-Scanner nimmt diesen extern erstellten Report und integriert die Daten in seine statische Analyse und zur weiteren Auswertung durch das Quality-Gate.

SonarQube verlässt sich also bewusst auf die spezialisierten Werkzeuge des jeweiligen Ökosystems, um die Tests auszuführen, und konzentriert sich auf das, was es am besten kann: die umfassende statistische Analyse und die Aggregation aller qualitätsrelevanten Metriken an einem Ort.

5.12.6 Parameter der SonarQube-Tasks

Der Task `sonarqube-scan` greift über das `SONAR_TOKEN` auf den SonarQube-Server zu. Für den Zugriff werden die Parameter `SONAR_HOST_URL` und `PROJECT_KEY` benötigt. Der Task liefert als Result die jeweils eindeutige `SONAR_TASK_ID` zurück, die vom nachfolgenden Task `check-quality-gate` als Result-Consumer benötigt wird.

Der SonarQube-Task `check-quality-gate` holt sich über das `SONAR_TOKEN` und die per Result vom Task `sonarqube-scan` bereitgestellte `SONAR_TASK_ID` alle Daten aus dem kompletten Report. Dieses Verfahren ist, wie ebenfalls bereits in Abschnitt 5.8.3 beschrieben wurde, deutlich aussagekräftiger und weniger fehlerbehaftet (Stichwort: Timeouts) als das Verfahren per `-Dsonar.qualitygate.wait=true`.

Task	Benötigte Parameter	Produziertes Result	Zweck
`sonarqube-scan`	`SONAR_HOST_URL`, `PROJECT_KEY`, `SONAR_TOKEN` (via Secret)	SONAR_TASK_ID	Startet den Scan auf dem Server und liefert die Task-ID zurück.
`check-quality-gate`	`SONAR_HOST_URL`, `SONAR_TASK_ID` (aus dem Result des vorigen Tasks), `SONAR_TOKEN` (via Secret)	-	Wartet auf den Abschluss der Analyse und prüft das Quality-Gate.

Tabelle 5.12 Parameter der SonarQube-Tasks

5.12.7 Setup der Pipeline und Tasks

Falls Sie das Script *setup-sonarqube.sh* ausgeführt haben, ist bereits alles am richtigen Platz. Falls nicht, erzeugen Sie zunächst den Namespace `sonarqube-pipeline`. In ihm installieren Sie die Pipeline (*sq-pipeline.yaml*), den Pipeline-Service-Account, das Registry-Secret sowie die zugehörigen Tasks aus dem Unterordner *tasks/*: `git-clone`, `golang-test`, `trivy-scanner`, `sonarqube-task`, `sonarqube-qualitygate`, `build-and-push` u. a.

Passen Sie dann den PipelineRun an Ihre Erfordernisse an und starten Sie ihn.

5.12.8 Details und zu beachtende Punkte

Sollte der Task `check-quality-gate` fehlschlagen, obwohl kein schwerer Fehler vorliegt (z. B. ein Issue wie *Use a specific version tag for the image*), können Sie zu Testzwecken auch in der SonarQube-UI das Issue bearbeiten und temporär per OPEN im Dropdown-Menü auf ACCEPTED setzen.

5.13 Überblick: Supply Chain Security (CI)

Nachdem wir uns bis zu diesem Punkt tiefergehend mit diversen Prüfverfahren und -mechanismen für unsere CI-Pipelines befasst haben, geht es in den folgenden Abschnitten um grundsätzlichere Security-Aspekte, nämlich um die unversehrte Integrität der CI-Herstellungskette, die sogenannte *Supply Chain Security*.