

# Ein Blick in die Tiefen von GitHub Actions und GitLab CI/CD

## Einleitung

GitLab hat kürzlich die Verfügbarkeit des GitLab CI/CD Catalogs (2023/12), verkündet. Dies stellt eine ideale Gelegenheit dar, den aktuellen Stand der beiden CI/CD-Systeme der beliebtesten Entwicklerplattformen zu vergleichen: GitLab CI & GitHub Actions.

*Disclaimer:* Es gibt zwar viele andere CI/CD-Systeme wie Circle CI, Azure DevOps, AWS CodeBuild, Travis CI und überraschenderweise ist auch Jenkins noch weit verbreitet, aber diese Tools sind auf die Bewältigung reiner CI/CD Aufgaben ausgerichtet und sind nicht Teil einer vollständig integrierten Entwicklerplattform. Außerdem habe ich die meisten von ihnen nicht in ausreichendem Maße genutzt, um eine aussagekräftige Bewertung abgeben zu können. Dieser Beitrag konzentriert sich ausschließlich auf GitLab CI und GitHub Actions, insbesondere deren SaaS-Angebote, aus technischer- und Anwendungssicht.

## Allgemeines CI Setup & Ablauf

Sowohl GitHub Actions als auch GitLab CI definieren ihre CI/CD-Jobs mithilfe von `yml`-Dateien, die vom CI/CD-Task-Scheduler geparsst und verarbeitet werden. Es werden ein oder mehrere Jobs erstellt, welche in einer oder mehrere Stages aufgeteilt werden. Standardmäßig laufen Jobs innerhalb einer Stage parallel ab, während Stages nacheinander ausgeführt werden.

GitHub Actions wie auch GitLab CI bieten die nahtlose Integration in ihre Git-Hosting-Plattformen. Wobei man beide Systeme auch nutzen kann, um Jobs in Repositories von Drittanbietern auszuführen. Allerdings ist dies mit Einschränkungen verbunden, da die Events (Trigger) meist in einem direkten Zusammenhang mit dem git-Repository sind, in dem der Job läuft. Dies macht den Ansatz unpraktikabel.

Während GitHub jede Datei unter `.github/workflows` individuell evaluiert, indem geprüft wird, ob das aktuelle Event mit einem der unter `on:` definierten triggern übereinstimmt, gibt es bei GitLab nur eine einzelne Datei (`.gitlab-ci.yml`). Während diese Datei separate CI-Konfigurationen einbinden kann, verschmilzt GitLab immer alle inkludierten Dateien in eine einzige monolithische CI-Pipeline Definition und prüft für jeden einzelnen Job, ob der Job erstellt und ausgeführt werden soll oder nicht.

GitHub verfügt über eine Vielzahl von Triggern, für alle erdenklichen Ereignisse, die in einem GitHub-Repository auftreten können. Unter anderem kann ein Workflow ausgeführt werden nachdem ein Issue erstellt, gelabelt oder kommentiert wurde. Bei GitLab gibt es nur die vordefinierte `CI_PIPELINE_SOURCE` Variable, welche Bit anderen vordefinierten variablen kombiniert werden kann, um zu entscheiden wann und ob ein CI Job ausgeführt wird. Events wie Issues oder Kommentare werden Definitionen unterstützt. Dafür ist die Einbindung von externen Event-Quellen bei GitLab jedoch flexibler gestaltet, da die Auswahl zwischen API, Triggern und Chat besteht. Diese ermöglicht die direkte Integration verschiedener Anwendungen wie Slack. Bei GitHub gibt es zur manuellen Erstellung von CI-Jobs das `workflow_dispatch` Event, welches via API oder die WebUI getriggert werden kann. Typische Event-Trigger wie das pushen eines bestimmten Branches/Tags oder ein `merge_requests` werden von beiden Systemen unterstützt.

## Erstellung der Jobs

Basierend auf dem aufgetretenen Event und dessen Werten, werden die spezifizierte CI/CD Jobs erstellt. Während das Schema eines Jobs relativ ähnlich ist, ist die Spezifikation und Ausführung des eigentlichen Befehls ziemlich unterschiedlich.

Jeder Job hat einen Namen, einen Selektor, der die zu nutzenden Runner zu definiert, Variablen und `script`- oder `step` Eigenschaften. Außerdem unterstützen beide Systeme die Definition einer Matrix, um einen Job mehrfach in verschiedenen Umgebungen (Environments), Runtime-Versionen oder Betriebssystemen auszuführen. Des weiteren kann man in der Jobdefinition auch Secrets und andere Jobs referenzieren. GitHub stellt sogar ein begrenztes Set von Expressions zur Verfügung, mit welchen man Variablen u. A. vergleichen wie auch verändern kann. Außerdem gibt es ein zusätzliches Property zur Bestimmung, ob ein Job ausgeführt werden soll, nachdem das `on` Property evaluiert wurde. Bei GitLab gibt es nur eine Entscheidungsebene, welche über das `rules`-Property definiert wird.

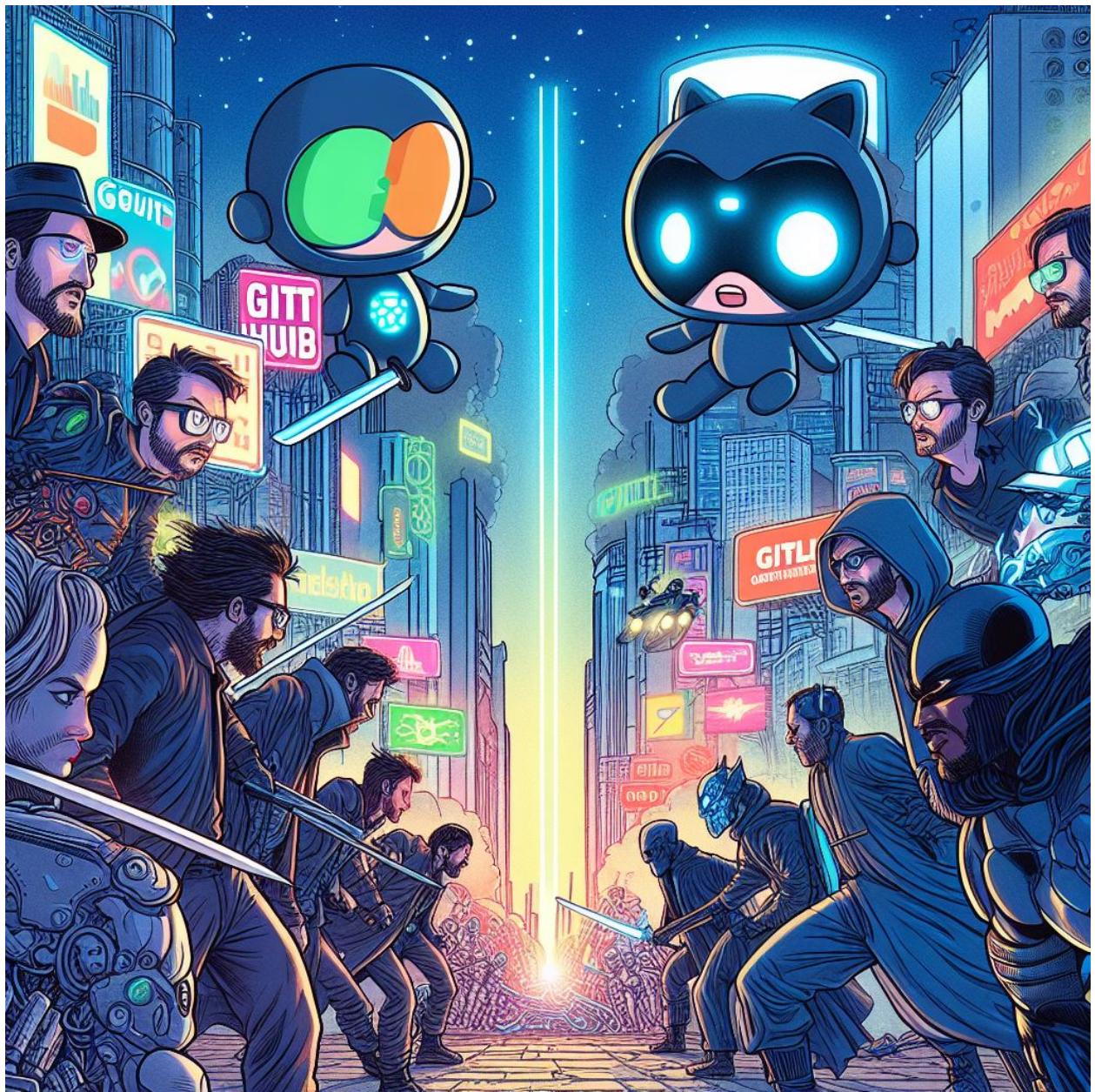


Figure 1: \_c521a4cc-8fe3-4b03-ab22-2286bec58397.jpeg

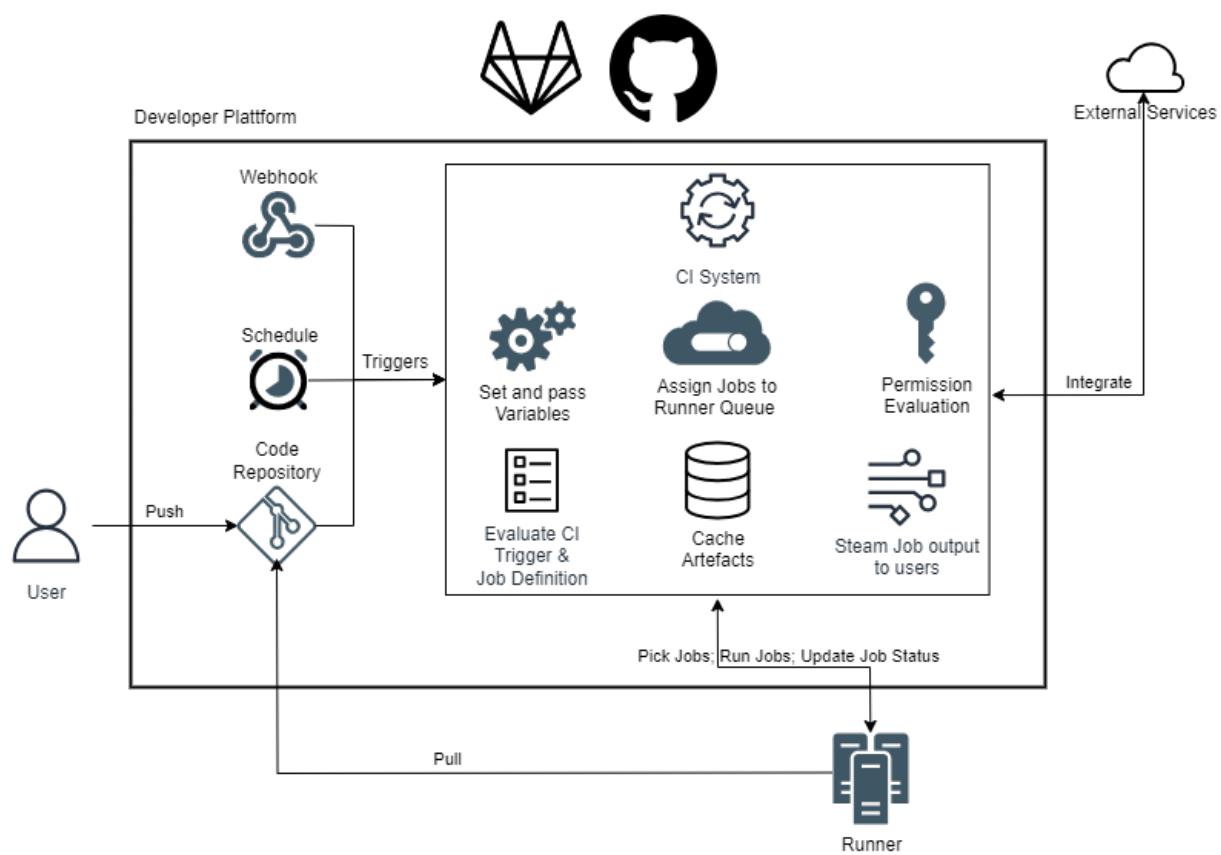


Figure 2: \_c521a4cc-8fe3-4b03-ab22-2286bec58397.jpeg

Zusätzlich kann jeder Job Abhängigkeiten zu anderen Jobs definieren. Die Abfolge der eigentlichen Schritte, die das CI-System ausführen soll, unterscheidet sich bei beiden Systemen:

## GitHub Actions

GitHub Actions bietet Anwendern die Möglichkeit beliebig viele Jobs pro Workflow-File zu erstellen. Jeder Job kann die gewünschte Anzahl an Schritten mittels Action Modulen ausführen. Als Action Modul bezeichnet man die Referenz auf ein anderes Git-Repository, welches einen spezifischen Task innerhalb der CI ausführt. Wenn so ein Job ausgeführt wird, lädt das CI-System die definierten Actions herunter und führt anschließend die darin definierte Logik aus. Aus Sicht des Anwenders sind die meisten Actions deklarativ, was es Nutzern ermöglicht diese zu verwenden ohne Details ihrer Implementierung zu kennen. Beispiele für solche Actions wären die Installation von Docker oder das ausführen von Minikube. Es gibt einen riesigen marketplace, welcher Offizielle- und Community Actions bereitstellt. Das Verhalten dieser Actions kann anhand von Eingabewerten beeinflusst werden, wodurch die Wiederverwendbarkeit sichergestellt wird und die Einstiegs-Barriere gesenkt wird, da Nutzer nicht wissen müssen, wie Actions im Detail funktionieren. Nutzer können auch eigene Actions erstellen.

Neben den Marketplace Actions gibt es auch die Möglichkeit zur Nutzung einer generischen `run` Action, welche jeglichen Shell-Code akzeptiert und es so ermöglicht imperative Befehle auszuführen. Zusätzlich zu Bash und PowerShell ist es direkt möglich interpretierte Sprachen wie Python oder JavaScript zu nutzen. Genau wie bei der Workflow Definitionen und der Job Definition wird bei jedem Schritt entschieden, ob dieser ausgeführt werden soll oder nicht. Die Entscheidung kann auch auf Basis von Werten stattfinden, die in früheren Schritten des Jobs berechnet wurden.

```
# GitHub Action example workflow
name: Example Pipeline

on:
  pull_request:
    branches: ['releases/*', 'main']
  push:
    branches: [main]

jobs:
  code-style:
    runs-on: ubuntu-latest
    steps:
      - name: checkout
        uses: actions/checkout@v4

      - name: Info for main branch
        if: github.ref == 'refs/heads/main'
        run: echo "Running tests in main"

      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - uses: hashicorp/setup-terraform@v3
      - uses: terraform-linters/setup-tflint@v4.0.0
      - uses: pre-commit/action@v3.0.0
```

## GitLab CI

In der GitLab CI hingegen gibt es drei Stages: `before_script`, `script` und `after_script`. Jede Stage ermöglicht das Angeben einer Liste von Shell Befehlen und man kann alles ausführen, was man auch in

einem Script auch könnte. Die `after_script` Stage wird immer ausgeführt, auch wenn ein vorheriger Schritt in einen Fehler gelaufen ist. Dadurch wird das Senden von Webhooks, Error-Handling und das Ausgeben von Debug Nachrichten ermöglicht. Man kann die Skripte auch wiederverwenden, indem man von einem vorhandenen Job erbt, einen vorhanden Job erweitert, oder mit dem `!reference` Tag anderswo definierte Scripte referenziert. Alle Befehle müssen imperative Shell Instruktionen sein und der Nutzer muss selbst dafür sorgen, dass die notwendigen Programme im Job zur Verfügung stehen.

```
# GitLab CI-Pipeline example
tags: [docker]
variables:
  MY_GLOBAL_VAR: Foo
  GIT_DEPTH: 5

include:
  - template: global/templates.yml
  - template: Security/SAST.gitlab-ci.yml

Build-Image-Buildah-Template:
  image: registry.access.redhat.com/ubi8/buildah:latest
  stage: build
  variables:
    REGISTRY: ${CI_REGISTRY}
    REGISTRY_CREDS: ${CI_REGISTRY_USER}:${CI_REGISTRY_PASSWORD}
    BUILD_IMAGE_TAG: ${CI_REGISTRY_IMAGE}:${CI_COMMIT_REF_NAME}
  script:
    - |
      buildah bud --pull \
        --build-arg COMMIT_HASH=${CI_COMMIT_SHORT_SHA} \
        --build-arg COMMIT_TAG=${CI_COMMIT_REF_NAME} \
        --tag "${BUILD_IMAGE_TAG}" \
        --file ${CI_PROJECT_DIR}/Dockerfile ${CI_PROJECT_DIR}
    - buildah push --creds "${REGISTRY_CREDS}" "${BUILD_IMAGE_TAG}"
  after_script:
    - echo "Image build!"
rules:
  - if: $CI_PIPELINE_SOURCE == "schedule"
  - if: $CI_PIPELINE_SOURCE == "push"
  - !reference [.global, default_rules]
```

## Unterschiede

Die vorgefertigten Actions von GitHub ermöglichen eine anwenderfreundliche und schnell konfigurierbare CI-Pipeline. Sie sind ideal für die Wiederverwendbarkeit und die Abstraktion von komplizierten und low-level Tasks. Bei Actions vom Community-Marktplatz müssen Benutzer beachten, dass sie hier fremden Code einbinden. Es wird empfohlen, eine fest spezifizierte Version der Action zu verwenden und alle Actions zu überprüfen, bevor sie verwendet werden, um mögliche Supply Chain Angriffe zu verhindern. Der Ansatz von GitLab erfordert mehr Aufwand im Vorfeld und ein tieferes Wissen über jeden einzelnen Schritt bei der durchzuführenden Aufgabe. GitLab ermöglicht zwar auch die Wiederverwendung von Jobs, ist aber nicht so flexibel, wie die späteren Abschnitte zeigen werden. Die detaillierte und klare Struktur von GitHub Actions, die bestimmt, ob ein Workflow, ein Job oder ein Schritt ausgeführt werden soll, ist ein klarer Vorteil gegenüber dem monolithischen Ansatz von GitLab.

## CI Konfiguration wieder verwenden

Wie der vorherige Abschnitt bereits gezeigt hat, ermöglichen GitHub Actions gute Wiederverwendbarkeit bei einzelnen Schritten. Allerdings bestehen CI-Jobs selten nur aus einem Schritt. Die meisten Jobs umfassen das Auschecken des Codes, die Installation der Laufzeitumgebung, das Herunterladen von Build-Abhängigkeiten und die Durchführung einiger Tests. Ein Docker-Build-Job ist fast immer derselbe, genau wie auch Sicherheitsscans immer gleich sind. Sie benötigen lediglich einige Argumente, die auf den richtigen Code verweisen, und Parameter für die Ausgabe. In einem Unternehmen könnte es wünschenswert sein, diese immer gleichen Schritte gemeinsam zu nutzen, um den Wartungsaufwand zu reduzieren. Dadurch könnte man auch sicherstellen, dass alle Jobs mit internen Standards und Richtlinien konform sind.

GitHub Actions bieten wiederverwendbare Workflows, dies sind CI-Workflows, die aus jedem anderen Projekt aufgerufen werden können. Diese wiederverwendbaren Dateien definieren einen oder mehrere Jobs und bieten Parameter, um die Ausführung der definierten Jobs zu beeinflussen. Da sie auf Git basieren, sind diese Workflow-Dateien versioniert und es kann eine fest spezifizierte Version über dessen Commit Hash verwendet werden.

Bei GitLab ist die Situation ein wenig komplizierter. GitLab verwendet eine monolithische `.gitlab-ci.yml` Datei, in der alle Jobs definiert sind. Die Jobs müssen jedoch nicht innerhalb dieser Datei deklariert werden. Sie können aus einem anderen Projekt oder einer anderen Datei stammen, solange diese über HTTP von der GitLab-Instanz aus zugänglich sind. Im Allgemeinen gibt es in GitLab zwei Arten, anderswo definierte Jobs über `includes` einzubinden. Zum einen gibt es das generische `include` einer beliebigen Job-Definitionsdatei, diese kann in einem gemeinsamen Repository liegen oder von einem Drittanbieter angeboten werden. Zum anderen gibt es die offiziellen GitLab CI/CD-Vorlagen. Bei diesen Vorlagen handelt es sich um gängige CI/CD-Jobs, die direkt von GitLab bereitgestellt werden und mit jeder GitLab-Installation geliefert werden. Beide Typen bieten nicht die Möglichkeit, Parameter direkt an die eingebundenen Jobs zu übergeben. Alle Anpassungen müssen über die CI/CD-Variablen von GitLab oder durch explizites Überschreiben des Jobs vorgenommen werden. Um diese Anpassungen vorzunehmen, ist es oft notwendig, den Quellcode eines bestimmten Jobs zu betrachten, dieser kann über mehrere Ebenen von verschachtelten `includes` definiert sein.

Am 21. Dezember 2023 kündigte GitLab die Beta-Verfügbarkeit des GitLab CI/CD Catalog (2023/12) an. Diese Funktion ermöglicht eine dritte Art von Jobs in der `.gitlab-ci.yml`. Diese bietet einen Community-Store, in dem jeder Benutzer seine eigenen Vorlagen für einen oder mehrere Jobs erstellen kann. Jeder kann diese Komponenten einbinden und sie über eine Reihe von Parametern anpassen. Die Benutzer müssen sich nur die Dokumentation der Components ansehen, um eine Liste aller verfügbaren Parameter und weitere Informationen zu erhalten. Damit gibt es eine viel sauberere Schnittstelle zwischen dem Anbieter und dem Nutzer einer Jobvorlage. Genau wie bei den GitHub-Actions. Nutzer können jedoch nach wie vor nicht die einzelnen Schritte in den Jobvorlagen beliebig anpassen und erweitern, ohne diese explizit zu komplett zu überschreiben. Die Components umfassen ganze Jobs und ähneln daher eher den wiederverwendbaren Workflows von GitHub als den Actions von GitHub. Seit März 2024 sind die Components allgemein verfügbar und verfügen über einen wachsenden Katalog. Eine Liste der bereits verfügbaren GitLab CI/CD Components finden Sie unter [gitlab.com/explore/catalog](https://gitlab.com/explore/catalog).

Die Erstellung einer eigenen Component ist recht einfach. GitLab gibt ein ein bestimmtes Repository-Layout vor und setzt ein Label für das Projekt, um dieses als CI/CD-Component zu kennzeichnen. Mit jedem neuen Release wird eine neue Version dieser Component erstellt. Um diese neue Funktionalität zu testen, habe ich meine eigene Component erstellt, die Docker-Images mit Googles kaniko builder (der kein Docker in Docker benötigt) erstellt und automatisch einige gängige OCI-Spec-Labels anwendet. Probieren Sie es einfach aus. **HINWEIS:** Genau wie bei GitHub-Actions von Drittanbietern sollte eine Component vor der Verwendung überprüft und auf eine exakte Version gepinnt werden.

Die CI/CD-Components sind zwar ein Schritt in Richtung eines benutzerfreundlicheren und wiederverwendbaren CI-Codes, bieten aber immer noch nicht die detailreiche Kontrolle von GitHub Actions. Die feinen Bedingungen von GitHub Actions auf Workflow-, Job- und Step-Ebene bieten eine viel flexiblere Einrichtung und stellen eine sauberere Lösung als von GitLab dar. GitLab verschmilzt Jobs aus vielen Quellen, führt diese zusammen und wertet diese dann aus. Die vollständige Datei `gitlab-ci.yml` mit allen Überschreibungen, Referenzen und Einschlüssen kann im Pipeline-Editor von GitLab angezeigt werden. Aufgrund der vielen

includes kann diese Datei jedoch schnell einige tausend Zeilen lang werden und ist nicht sehr benutzerfreundlich zu lesen.

## Runners

Um die CI/CD-Jobs auszuführen, benötigen sowohl GitHub als auch GitLab sogenannte Runner. Beide Dienste bieten “*Hosted Runners*” an, die dem jeweiligen Anbieter gehören und von diesem gewartet werden (SaaS-Angebot). Jeder Anbieter bietet ein kostenloses Modell für seine CI/CD-Runner an, welches eine bestimmte Anzahl von Rechenminuten umfasst. Die folgende Tabelle gibt einen kurzen, aber nicht vollständigen Überblick über die Angebote der beiden Anbieter. (Stand 05/2024)

Plan	Inklusivminuten	Speicher	Preis	Kosten pro Minute/pro Kern für zusätzliche Minuten
GitHub Free	2000	500 MB	0\$	0.004*  GitHubPro 3000 1GB 4
GitLab Pro	10.000	n/a	29\$	0.005\$

*Hinweis:* Die Kosten pro Minute gelten für zusätzliche Minuten, die über die inbegriffenen Minuten hinausgehen. Sie wurden auf der Grundlage der verfügbaren Preisinformationen berechnet.

Diese Tabelle ist nicht repräsentativ für die realen Kosten, da beide Anbieter eine Vielzahl unterschiedlicher Recheneinheiten mit unterschiedlicher Leistung und verschiedenen Betriebssystemen anbieten. Je nach Größe und Betriebssystem gibt es einen Kostenmultiplikator, der für jede tatsächlich verbrauchte Rechenminute einberechnet wird. Die Zahlen in der Tabelle beziehen sich auf die standardmäßigen (kleinsten) Runner auf einem Linux-System. Spezifische Anwendungsfälle finden Sie in den Preisdokumenten von GitHub oder den Preisdokumenten von GitLab.

Beide Anbieter bieten Linux-, macOS- und Windows-Runner an. Zum Zeitpunkt der Erstellung dieses Artikels befinden sich die Angebote von GitLab für macOS und Windows noch in der Beta-Phase. Die GitHub-Runner werden in der Azure-Cloud unter Verwendung von VMs der Serie “Standard\_DS2\_v2” gehostet und die GitLab-Runner werden auf der Google Compute Platform (GCP) unter Verwendung der Serie “n2d\_machines” gehostet. Beide Runner-Angebote werden derzeit nur in den USA gehostet, was sich auf die Latenzzeit auswirken und Einhaltung von Datenschutzrichtlinien beeinträchtigen kann.

Die Architektur beider CI/CD-Systeme sieht vor, dass die verfügbaren Runner kontinuierlich mit der GitHub/GitLab-Instanz kommunizieren und Jobs aus ihrer Aufgabenliste abarbeiten. Nur die GitHub/GitLab-Instanzen muss für die Runner erreichbar sein, die Runner selbst können in abgeschotteten und nicht mit dem Internet verbunden Netzwerkzonen betrieben werden. Dies kann für bestimmte restriktive Netzwerktopologien wichtig sein. Während die Job Aufgabenverteilung recht ähnlich ist, gibt es einige Unterschiede zwischen GitHub und GitLab bei der Art und Weise, wie die Jobs ausgeführt werden.

## GitHub Runners

Wenn ein Ereignis einen Workflow auslöst, werden die definierten Jobs innerhalb des Workflows mit Vorlagen versehen (mit Lazy Templating) und der Jobwarteschlange hinzugefügt. Verfügbare GitHub-Runner, die mit dem Tag `runs-on` übereinstimmen, rufen automatisch die in der Warteschlange stehenden Jobs ab und führen sie auf einer neuen, nur für diesen Job bereitgestellten, VM aus. Die von GitHub bereitgestellten VMs haben viele Tools vorinstalliert, um eine schnelle Ausführung der Jobs zu ermöglichen. Im Wesentlichen werden alle Befehle direkt auf dem Host ausgeführt, auch wenn die eigentlichen Befehle durch die GitHub-Actions abstrahiert werden. Für jeden Schritt wird ein individueller Befehl erstellt, welcher es dem Benutzer ermöglicht, Werte von einem Schritt oder Job an einen anderen zu übergeben, indem er die spezielle `GITHUB_OUTPUT` Variable verwendet. Die übergebenen Werte müssen nicht vor dem Start eines Jobs bekannt sein, sondern können während der Jobausführung abgerufen, berechnet und weitergegeben werden.

Die Befehlsausgaben der Jobs werden an die GitHub-Instanz zurückgeleitet und sind über die GitHub-Benutzeroberfläche in einem Shell-ähnlichen Ausgabe-Canvas zugänglich. Die Ausgabe erfolgt nahezu in Echtzeit, aber die Oberfläche kann nicht zuverlässig über “Strg-F” durchsucht werden.

## GitLab Runners

Die `gitlab-ci.yml` Datei wird jedes Mal, wenn ein CI-Ereignis eintritt, vollständig zusammengeführt und ausgewertet. Wenn der `rules` Abschnitt eines Jobs mit dem auslösenden Ereignis übereinstimmt, wird der Job zur entsprechenden Jobwarteschlange hinzugefügt. GitLab verwendet `Tags`, um Jobs und Runner zuzuordnen. Ein Beispiel wäre `tags: [linux, large]`, wobei der ausführender Runner dann mindestens diese Tags haben muss.

Die Runner von GitLab können verschiedene Exekutoren (Executors) haben, die bestimmen, wie der benutzerdefinierte Job ausgeführt wird. Die unterstützten Executors sind Shell, SSH, VirtualBox, Docker, Kubernetes und einige mehr. Jeder Executor bringt einige Vor- und Nachteile mit sich. Die Shell- und SSH-Executors werden direkt auf dem angegebenen Rechner ausgeführt. Die Benutzer müssen sicherstellen, dass alle Abhängigkeiten entweder bereits auf dem Rechner installiert sind oder im Skriptabschnitt des Jobs enthalten sind. Diese Executors verändern das Host-Dateisystem und sind nicht idempotent, was zu mangelnder Reproduzierbarkeit und unerwartetem Verhalten führen kann. Daher wird oft die Verwendung eines virtualisierten Executors bevorzugt. Docker und Kubernetes verwenden das im `image` Tag angegebene Container-Image und führen alle Skripte innerhalb dieses Containers aus. Nach Abschluss des Jobs wird der Container gelöscht, und das Hostsystem befindet sich im selben Zustand wie zuvor.

Derzeit verwenden die SaaS-Runner von GitLab den Executor “Docker+Machine”, einen von GitLab gepflegten Fork des veralteten Projekts Docker Machine. Diese Executors erzeugen neue Google Cloud VMs mit installiertem Docker, um das angegebene Container-Image auszuführen und alle Jobskripte auszuführen. Zum Zeitpunkt der Erstellung dieses Artikels arbeitet GitLab daran, diesen Executor durch eine Alternative zu ersetzen.

Obwohl diese Executors idempotent sind, haben sie einige Nachteile, wenn Benutzer einige Low-Level-Systemaktionen durchführen wollen oder selbst Virtualisierung verwenden müssen, was zu verschachtelter Virtualisierung führt. Die Ausführung von Container-Builds in der GitLab-CI erfordert häufig die Verwendung von Docker in Docker, was zu einem Bind-Mount des Docker-Sockets und einer langsameren Build-Leistung führt, es sei denn, es werden Tools wie `kaniko` oder `buildah` verwendet.

Die Skriptausgaben werden auch in regelmäßigen Abständen an die GitLab-Instanz zurückgesendet. Das resultierende Protokoll in der Benutzeroberfläche ist vollständig durchsuchbar. Im Gegensatz zu GitHub lädt GitLab automatisch das Git-Repository an der im Ereignis, das den Job ausgelöst hat, angegebenen Adresse herunter, sofern dies nicht ausdrücklich deaktiviert wurde. Wenn im Job Artefakte definiert wurden, werden diese ebenfalls automatisch hoch- und heruntergeladen. GitLab bietet auch eine größere Anzahl von vordefinierten Variablen, auf die Benutzer während der Ausführung des Jobs zugreifen können. Variablen zwischen Jobs können aneinander weitergegeben werden, indem sie in eine Datei geschrieben werden und diese Artefakte zwischen Jobs weitergegeben werden.

## Vergleich

Die Ausführung von Jobs ist bei GitHub und GitLab unterschiedlich. Die GitLab-Lösung sieht auf den ersten Blick komplexer aus, jedoch ist kein Wissen über die Runner-Architektur für Entwickler erforderlich. Nur GitLab-Administratoren, die ihre eigenen Runner implementieren, müssen die Details kennen, um sich für einen geeigneten Executor zu entscheiden. Je nach Größe der Organisation sollte ein einfacher Docker- oder Kubernetes-Executor die meisten Anwendungsfälle für einen CI/CD-Runner abdecken. GitHub ermöglicht es Benutzern auch, ihre eigenen, selbst gehosteten Runner einzubringen, die genau wie GitLab nicht in Rechnung gestellt werden (zumindest nicht von GitHub/GitLab). Eigene Runner bieten die Möglichkeiten zum Nutzen von konsistenter Leistung oder die Verfügbarkeit anderer CPU-Architekturen und GPU-Unterstützung. Außerdem lassen sich damit Netzwerkrestriktionen überwinden, indem der Runner innerhalb eingeschränkter Netzwerksegmente platziert werden. Eigene Runner könnten auch erforderlich sein, um Latenzprobleme zu überwinden, da sich derzeit sowohl die Runner von GitHub als auch die von GitLab in den USA befinden.

Die GitHub-Runner ermöglichen eine flexiblere Nutzung, da sie eine vollständige VM sind, die für die Dauer des Jobs den Nutzern überlassen wird. Die GitLab-Runner sind eher auf einen bestimmten Anwendungsfall ausgerichtet. Die GitLab-Runner checken automatisch den Code aus und verwalten die Verteilung der

Artefakte für den Benutzer. Wenn das richtige Container-Image gewählt wird, muss nichts weiter installiert werden, da alle Tools bereits im Container vorhanden sind.

Beide Runner sind generell in der Lage, die meisten Anwendungsfälle zu erfüllen. Die Nutzer sollten sich lediglich der jeweiligen Einschränkungen in Bezug auf Standort, Verfügbarkeit, CPU-Architektur und andere Merkmale und Eigenschaften wie Kosten bewusst sein. Erwähnenswert ist noch, dass selbst gehostete Runner je nach Konfiguration mehr als einen Job pro Host gleichzeitig ausführen können. Aus Gründen des Datenschutzes ist diese Funktion bei gehosteten/SaaS-Runnern nicht aktiviert.

Ein Leistungsvergleich ist bewusst nicht Teil dieses Vergleichs, da es zu viele Variablen und Einflüsse gibt, um aussagekräftige Ergebnisse zu erzielen.

## Security and Secrets

Sowohl bei GitHub als auch bei GitLab können Benutzer Variablen und Secrets festlegen, die dann in der CI/CD-Runtime verfügbar sind. Beide Eingabearten können auf Instanz-, Gruppen-/Organisations- und Projektebene und sogar pro erstellter Umgebung festgelegt werden. Secrets sind spezielle Variablen, die als vertraulich gelten und vom CI-System maskiert werden, wenn sie im Ausgabeprotokoll gefunden werden. In GitLab können Benutzer mit den entsprechenden Berechtigungen den Wert eines Secrets auch nach dem Festlegen erneut einsehen. In GitHub wird der geheime Wert nach dem Anlegen nie wieder angezeigt. Auswirkungen im Bezug auf die Sicherheit macht dies jedoch keinen entscheidenden Unterschied, da jeder, der Zugriff auf die CI-Dateien hat, die Secrets beim Ausführen eines Jobs ausgeben kann. Die geheimen Werte werden zwar maskiert, dieser Sicherheitsmechanismus kann leicht umgangen werden, indem der Wert umgekehrt und base64-kodiert wird, bevor er in das Protokoll geschrieben wird. Daher lautet die allgemeine Empfehlung, kurzlebige Token zu verwenden.

Für projektbezogene Ressourcen verwenden beide CI-Systeme ein solches kurzlebiges Token, um auf den in Git gespeicherten Code zuzugreifen, Artefakte hochzuladen oder Releases zu erstellen. Der `GITHUB_TOKEN` oder der `CI_JOB_TOKEN` von GitLab ist nur während der Ausführung des Jobs gültig. Die Berechtigungen für diese Token können jeweils eingeschränkt werden. GitLab erlaubt es, die Berechtigungen von Usern den Anforderungen entsprechend zu setzen. Der `CI_JOB_TOKEN` wird die Berechtigungen des Benutzers, der den CI-Job ausgelöst hat, dann übernehmen. Wenn jedoch ein CI-Job läuft, ist dieses Token mit seinen spezifischen Berechtigungen für alle Jobs und Aufgaben innerhalb dieser Jobs zugänglich.

GitHub hingegen ermöglicht ein umfassendes Berechtigungsmodell für sein Token. Diese Berechtigungen können auch auf Workflow- oder Jobebene definiert werden. Im Gegensatz zu GitLab macht GitHub sein Token nicht standardmäßig für jeden Schritt im CI-Flow zugänglich. Das Erstellen eines Releases über GitHub-Actions erfordert den Zugriff auf die GitHub-Release-API mit einem gültigen Token. Für das Herunterladen von öffentlichen Abhängigkeiten oder das Kompilieren von Code wird jedoch kein Token benötigt. Nur Schritte, die den Zugriff auf den Token anfordern oder explizit über den Parameter `$$ secrets.GITHUB_TOKEN $$` Zugriff erhalten, können dieses Token verwenden. Der standardmäßige, nicht vertrauenswürdige Ansatz ist sehr sinnvoll, da die Wahrscheinlichkeit, dass Benutzer Actions von Drittanbietern in ihrer CI verwenden, also Code, der von Fremden geschrieben wurde, sehr viel größer ist. Vor der Einführung des CI/CD-Katalogs von GitLab gehörte der meiste CI-Code in GitLab den Projekteigentümern oder ihrer Organisation. GitHub verwendet standardmäßig den “least privilege” Ansatz; Schritte erhalten nur Zugriff, wenn sie diesen benötigen. Organisationen können einen noch restriktiveren Berechtigungsumfang für den `GITHUB_TOKEN` durchsetzen und die erlaubten GitHub-Actions auf interne oder auf einer Whitelist von Actions Actions beschränken, um damit die Compliance-Richtlinien des Unternehmens zu erfüllen. GitLab unterstützt eine solche Funktion derzeit nicht von Haus aus. Stattdessen erlaubt GitLab den Benutzern, die Berechtigungen ihres `CI_JOB_TOKEN` zu erweitern, um auf zusätzliche Git-Repositories zuzugreifen. In einem Projekt mit mehreren Repositories/Microservices führt dies zu weniger zu verwaltenden tokens. Wenn GitHub Actions auf mehrere private Repositories zugreifen soll, müssen die Benutzer ihr eigenes Token erstellen, da das CI-System eine solche Funktion derzeit nicht unterstützt und der Standardtoken nur für das aktuelle Repository gültig ist.

Beide Dienste unterstützen die Verwendung von kurzlebigen Token und die Integration von Lösungen anderen Anbietern. GitLab bietet mit HashiCorp Vault, Azure Key-Vault und Google Secret Manager eine First-Party-Unterstützung für die Verwendung externer Secrets. Nutzer können so ohne weitere Konfiguration auf

Secrets aus externen Quellen zugreifen. Einige dieser Funktionen sind jedoch nur in der Pro-Variante von GitLab verfügbar. Jedoch fungieren sowohl GitHub als auch GitLab als *Identity-Provider*, (IDP), um die Verwendung von OpenID Connect (OIDC) zu ermöglichen und eine sichere Kommunikation mit den drei großen Cloud-Anbietern unter Verwendung kurzlebiger Token zu ermöglichen. Je nach Konfiguration erlaubt dies den Benutzern die Verwaltung von Cloud-Ressourcen in der CI/CD, indem sie eine Trust-Relationship etablieren. So können kurzlebige Tokens anstelle von expliziten Anmeldeinformationen verwendet werden. GitLab baut diese Funktionalität direkt in sein CI/CD-System ein, während GitHub auf GitHub-Actions Definitionen wie `configure-aws-credentials` oder `azure-login` zurückgreift.

Eine Zusatzfunktion, die GitLab nicht hat, ist die Möglichkeit, Passwort Maskierungen hinzuzufügen, während das CI bereits läuft. Insbesondere bei der Verwendung von kurzlebigen Zugriffstoken, die das CI nicht im Voraus kennen kann, ist es hilfreich, `echo "::add-mask::MY_TOKEN"` zu verwenden, um eine ungewollte Aufdeckung in den Protokollen zu verhindern.

## Zusätzliche Funktionen und Dienste

Die automatische Ausführung von Aufgaben, wenn ein bestimmtes Ereignis ausgelöst wird, ist die Kernfunktionalität eines CI/CD-Systems. Jedoch steigen die Anforderungen und der Wettbewerb in der CI/CD-Welt steigen an. Funktionen, die das Leben von Entwicklern erleichtern, können ein wichtiges Verkaufsargument sein. Die folgenden Funktionen wurden im vorherigen Vergleich nicht berücksichtigt, sollten aber erwähnt werden, um ein vollständiges Bild von GitHub Actions und GitLab CI zu erhalten.

Sowohl GitHub Actions als auch GitLab CI können neben den Hauptaufgaben zusätzliche Dienste ausführen. Diese Dienste werden als Container definiert. Die kann beispielsweise eine Datenbank oder einen lokalen Webserver innerhalb des CI-Jobs sein. Ein Job kann einen Datenbankmigrationstest auf einer kurzlebigen Datenbank ausführen oder einen End-to-End-Test mit Playwright gegen eine temporär gehostete Website durchführen, alles innerhalb eines CI-Jobs.

Projektdokumentation oder eine statische Website können auf GitHub und GitLab mithilfe ihrer *Pages*-Funktion gehostet werden. Beide CI/CD-Systeme können HTML-Websites erstellen und direkt, ohne explizite Authentifizierung, bereitstellen. Um Tests und Build-Jobs zu beschleunigen, unterstützen beide Systeme Caches. GitHub verfügt über offizielle Action zum Up- und Downlaod von Caches, während GitLab das Schlüsselwort “cache” anbietet und das Up- und Download von Caches als Teil der Jobinitialisierung verwaltet.

Wenn alle Tests erfolgreich waren, kann der Service in einer oder in mehreren Environments bereitgestellt werden. Beide Systeme bieten das **Environment** Schlüsselwort. Jeder Job, der auf eine Umgebung verweist, erzeugt ein neues Deployment für diese Umgebung. Environments können über einen eigenen Satz von Variablen und Secrets verfügen, um eine striktere Trennung zwischen ihnen zu ermöglichen und die Sicherheit zu erhöhen. Im Repository oder im Pull Request UI ist dann ein Link zu dem erfolgreichen Deployment zu sehen. Bei GitHub müssen die Benutzer zuvor Environments manuell erstellen. GitLab kann Environments dynamisch auf der Grundlage von Variablen, wie z.B. dem Branchnamen, erstellen. Darüber hinaus verfügt GitLab über verschiedene Aktionen für Environments, um sie bei Bedarf zu erstellen, darauf zuzugreifen oder sie zu löschen. Environments können sogar automatisch nach einer bestimmten Zeit oder nach dem Schließen eines Pull-Requests entfernt werden. Im Vergleich zu GitHub sind die Environments von GitLab fortschrittlicher und vielseitiger. Für Kubernetes-Benutzer bietet GitLab den optionalen GitLab Agent, um ohne zusätzliche Token auf das Innere eines Clusters zuzugreifen. Der Agent wird innerhalb eines Clusters bereitgestellt und kommuniziert mit einer GitLab-Instanz, auf die über die CI zugegriffen werden kann.

## Gesamtvergleich

GitHub Actions und GitLab CI sind zwei sehr leistungsfähige CI/CD-Systeme, mit denen Benutzer fast jede erdenkliche Aufgabe innerhalb des Lebenszyklus der Softwareentwicklung automatisieren können. Auf funktionaler Ebene sind beide Systeme ähnlich gleichwertig, aber die Art und Weise, wie sie implementiert werden, unterscheidet sich erheblich.

GitHub bietet eine stabile, vielseitige und modulare Plattform für nahezu jede erdenkliche Aufgabe. Es lässt sich nahtlos in jede der Funktionen von GitHub integrieren. Bei Bedarf ist eine Integration mit Standardlösungen möglich, z.B. die Ausführung von Containern oder die Authentifizierung bei Diensten

Dritter über OIDC. GitHub stützt sich in hohem Maße auf seinen riesigen Marktplatz offizieller und Community Actions, die den Großteil der eigentlichen Arbeit erledigen, wie z.B. das Auschecken von Code, die Installation von Software, die Nutzung von Caches oder die Veröffentlichung von Releases. GitHub bietet auch eine sehr detaillierte Kontrolle über sicherheitsrelevante Einstellungen, die deutlich restriktiver als bei seinem Konkurrenten GitLab durchgesetzt werden können.

GitLab hingegen bietet seinen Nutzern eine viel besser verwaltete Lösung. Es ist Bestandteil des CI/CD-Systems den Code auszuchecken, Caches sowie Releases zu verwalten. Es lässt sich direkt mit Vault und den wichtigsten Cloud-Anbietern integrieren. GitLab selbst scheint auch eine einfachere Integration mit Systemen von Drittanbietern zu ermöglichen. CI-Trigger von externen Diensten wie dem Jira Issue-Tracker und Chat-Tools sind vielseitiger, genau wie die Einrichtung der CI-Umgebung. Bei GitLab muss der Benutzer jedoch alle CI-Task Aufgaben selbst schreiben. Die gemeinsame Nutzung und Wiederverwendung von CI-Definitionen wird mit dem monolithischen Ansatz einer großen `.gitlab-ci.yml`-Datei schwierig und schnell unübersichtlich. Es gibt mehrere Ebenen für `includes`, Verweisen und Erweitern, was es schwierig machen kann, einen schnellen Überblick zu bekommen, was ein Job eigentlich tut. Möglicherweise wird dies mit dem neuen CI/CD-Katalog von GitLab besser, aber das ist nicht garantiert, da die Components eher mit den wiederverwendbaren Workflows von GitHub zu vergleichen sind als mit einzelnen GitHub Actions.

GitLab befindet sich definitiv mitten in einem Migrationsprozess in Bezug auf ihre CI, nicht nur um sie wiederverwendbarer zu machen, sondern auch um ihre “Docker+Maschine”-Runner loszuwerden. Im Gegensatz zu GitHubs modularem Ansatz scheint GitLab aufgrund seines stärker verwalteten Ansatzes mit deutlich mehr historischen Herausforderungen zu kämpfen. Aus softwaretechnischer Sicht könnte das Design von GitHub auf lange Sicht vorteilhafter sein, da der modulare Ansatz es ihnen ermöglicht, neue Funktionen hinzuzufügen, ohne dass viele historische Hindernisse im Weg stehen.

Es gibt keine klare Empfehlung für das eine oder das andere System. Beide sind mehr als fähig, jede erdenkliche Aufgabe auszuführen. Die Anforderungen an die eigentliche Entwicklungsplattform, GitHub und GitLab, sollten viel wichtiger sein als ihre CI-Funktionen. Kosten und Datenschutz sowie die Möglichkeit des Self-Hostings sind oft relevantere Faktoren als die Unterschiede zwischen den CI/CD-Systemen.



Figure 3: \_696c50f3-b0c8-4104-abdf-376ca9b8b0fb.jpeg