



KAPITEL 7: CONTINUOUS INTEGRATION UND DEPLOYMENT

LERNZIELE

- Erklären, was Continuous Integration, Delivery und Deployment ist
- Bauprozesse mittels GitLab CI Deskriptoren erstellen
- Werkzeuge wie GitLab CI und GitHub Actions differenzieren
- Anforderungen an ein Deployment von Infrastrukturen, die mittels Docker Compose oder Kubernetes-Manifesten beschrieben sind, beschreiben



7.1 EINLEITUNG

Continuous Integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily - leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible. Many teams find that this approach leads to significantly reduced integration problems and allows a team to develop cohesive software more rapidly.

Martin Fowler



GRUNDPRINZIPIEN



Gemeinsame Codebasis

Um innerhalb einer Arbeitsgruppe sinnvoll integrieren zu können, muss eine Versionsverwaltung existieren, in die alle Entwickler ihre Änderungen kontinuierlich integrieren können.

Automatisierte Übersetzung

Jede Integration muss einheitlich definierte Tests wie statische Code-Überprüfungen durchlaufen, bevor die Änderungen integriert werden. Dafür ist eine automatisierte Übersetzung notwendig. Um Testergebnisse von den Arbeitsumgebungen unabhängig zu machen, empfiehlt sich der Einsatz von separaten Test-Umgebungen. Damit können auf diesen Rechnern auch gezielt Verfahren implementiert werden, um die Testlaufzeit zu minimieren.

Kontinuierliche Test-Entwicklung

Jede Änderung sollte möglichst zeitgleich mit einem dazugehörigen Test entwickelt werden (beispielsweise mittels testgetriebener Entwicklung). Mit Hilfe von kontrollflussorientierten Testverfahren (Analyse der Code-Überdeckung, engl.: "Code Coverage Analysis") kann diese Vorgehensweise dokumentiert und kontrolliert werden.

Häufige Integration

Jeder Entwickler sollte seine Änderungen so oft wie möglich in die gemeinsame Code-Basis integrieren. Mit kurzen Integrations-Intervallen reduziert man das Risiko fehlschlagender Integrationen und sichert gleichzeitig den Arbeitsfortschritt der Entwickler in der gemeinsamen Code-Basis (Datensicherung, engl: "backup").



Integration in den Hauptbranch

Jeder Entwickler sollte seine Änderungen in den Hauptbranch des Produktes integrieren, wo dann automatisch ein Build- und Testzyklus gestartet wird. Dieser Build wird der kontinuierliche Integrationsbuild genannt.

Kurze Testzyklen

Der Test-Zyklus vor der Integration sollte kurz gehalten sein, um häufige Integrationen zu fördern. Mit steigenden Qualitätsanforderungen für die einzelnen Integrationen steigt auch die Laufzeit zur Ausführung der Test-Zyklen. Die Menge der vor der Integration durchgeführten Tests muss sorgfältig abgewogen werden, weniger wichtige Tests werden dann nur nach der Integration durchgeführt.



Gespiegelte Produktionsumgebung

Die Änderungen sollten in einem Abbild der realen Produktionsumgebung getestet werden.

Einfacher Zugriff

Auch Nicht-Entwickler brauchen einfachen Zugriff auf die Ergebnisse der Software-Entwicklung. Dies müssen nicht notwendigerweise Quellen sein, sondern kann beispielsweise das in das Testsystem gespielte Produkt für Tester, die Qualitäts-Zahlen für Qualitäts-Verantwortliche, die Dokumentation oder ein fertig paketierte Abbild für Release Manager sein.

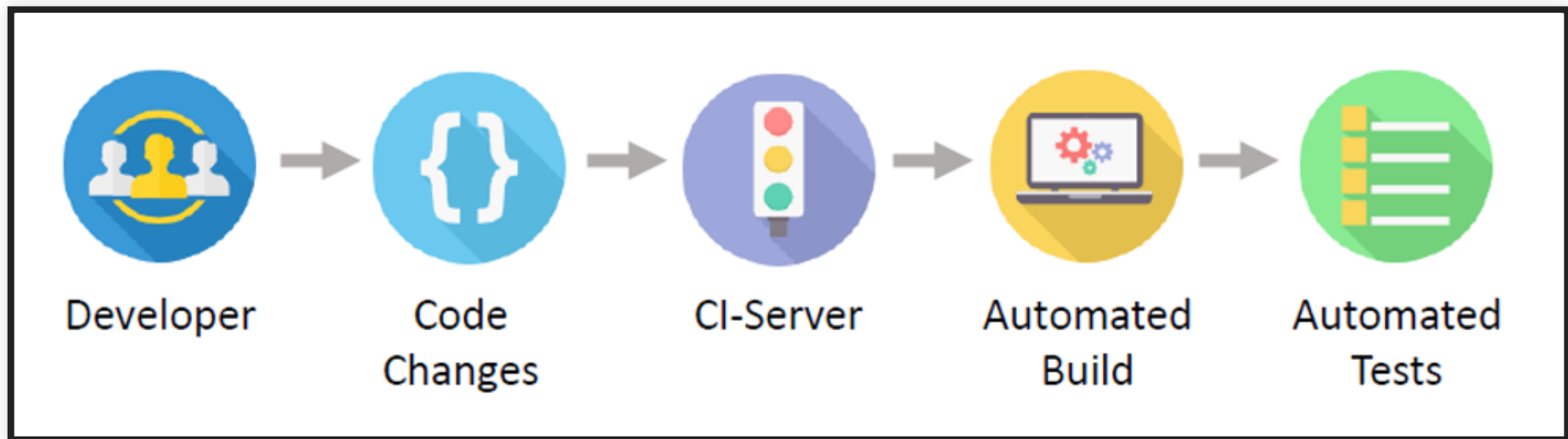
Automatisiertes Reporting

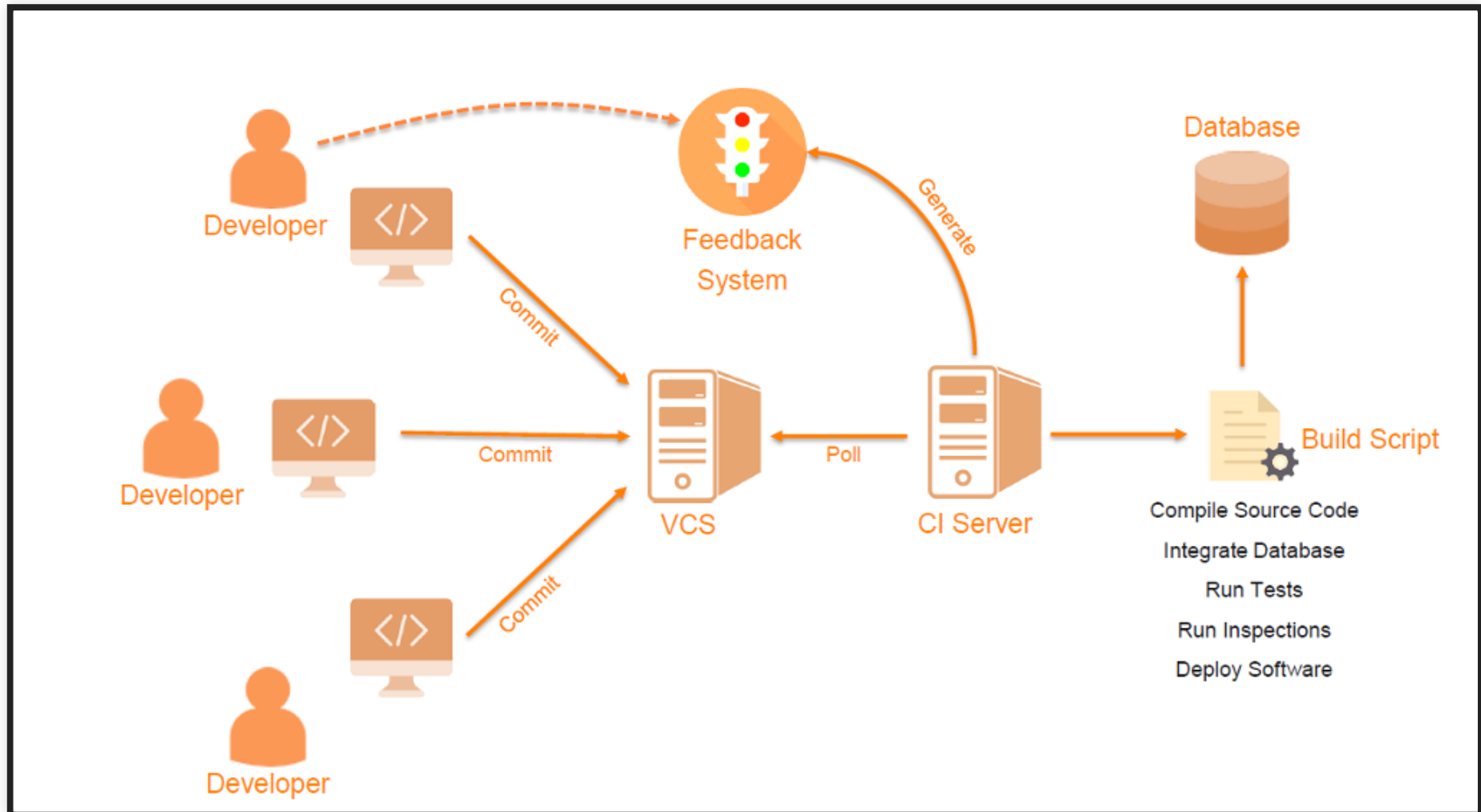
Die Ergebnisse der Integrationen müssen leicht zugreifbar sein. Sowohl Entwickler als auch andere Beteiligte müssen leicht Informationen darüber bekommen können, wann die letzte erfolgreiche Integration ausgeführt wurde, welche Änderungen seit der letzten Lieferung eingebracht wurden und welche Qualität die Version hat.



Automatisierte Verteilung

Jede Version sollte leicht in eine Produktionsumgebung (oder ein Abbild derselbigen) überführt werden können. Hierfür sollte die Softwareverteilung automatisiert sein.







VOR- UND NACHTEILE

Vorteile

- Integrations-Probleme werden laufend entdeckt und behoben (gefixt) – nicht erst kurz vor einem Meilenstein.
- Frühe Warnungen bei nicht zusammenpassenden Bestandteilen.
- Sofortige Unittests entdecken Fehler zeitnah.
- Ständige Verfügbarkeit eines lauffähigen Standes für Demo-, Test- oder Vertriebszwecke.
- Die sofortige Reaktion des Systems auf das Einchecken eines fehlerhaften oder unvollständigen Codes „erzieht“ die Entwickler im positiven Sinne zu einem verantwortlicheren Umgang und kürzeren Checkin-Intervallen. Der Merge-Aufwand wird immer größer, je länger man mit der Integration wartet.

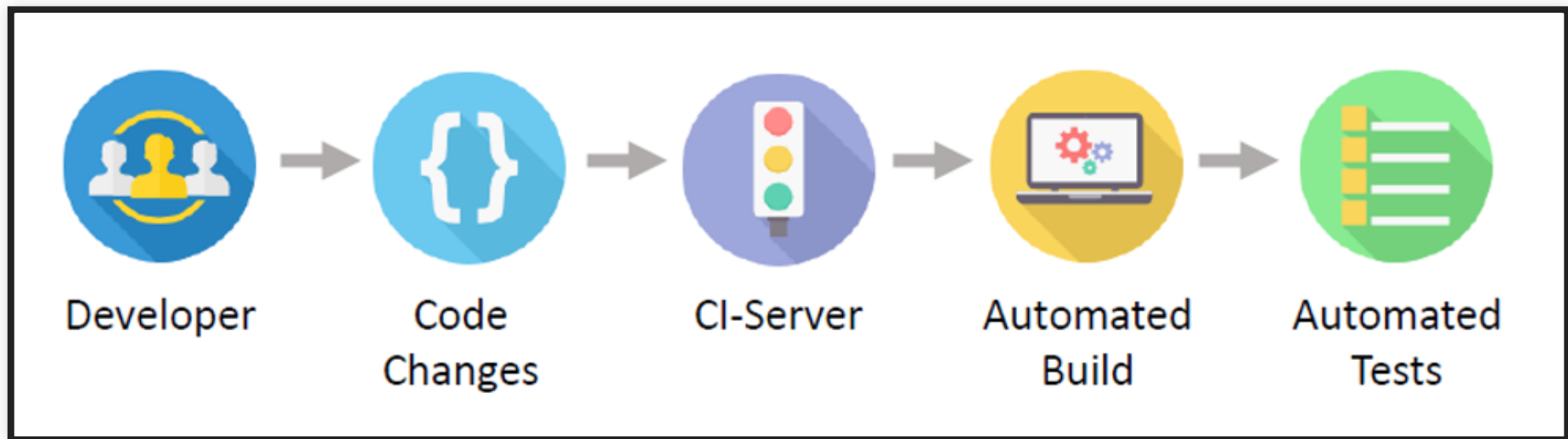
Nachteile

- Umstellung von gewohnten Prozessen
- Benötigt zusätzliche Server und Umgebungen
- Erarbeitung von geeigneten Test-Abläufen nötig
- Es kann zu Wartezeiten kommen, wenn mehrere Entwickler annähernd gleichzeitig ihren Code integrieren möchten



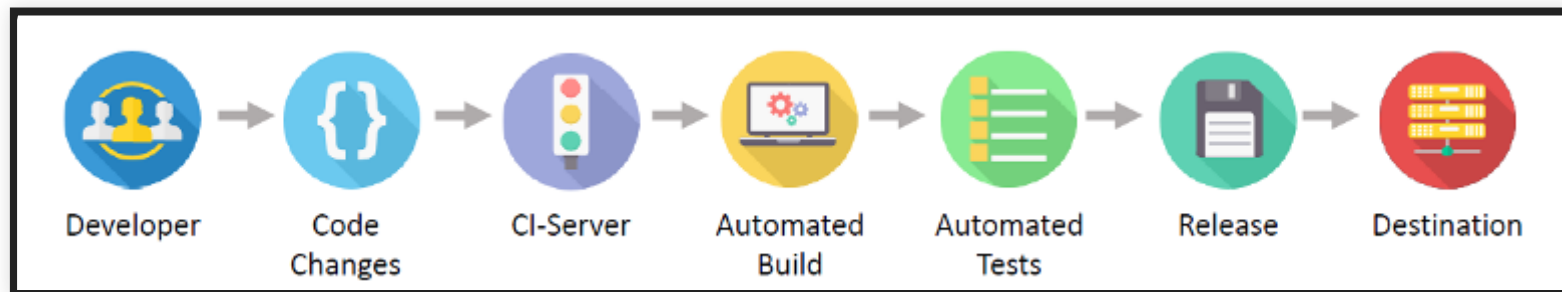
DELIVERY UND DEPLOYMENT

Continuous Integration



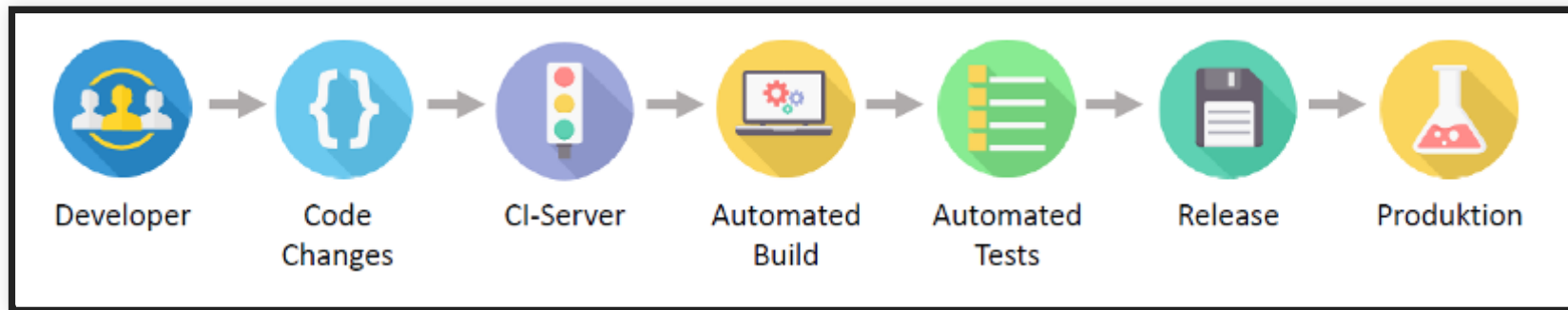
Continuous Integration bezieht sich in der Regel auf das Integrieren, Erstellen und Testen von Code innerhalb der Entwicklungsumgebung.

Continuous Delivery



Continuous Delivery baut darauf auf und befasst sich mit den letzten Schritten, die für die Produktionsbereitstellung erforderlich sind.

Continuous Deployment



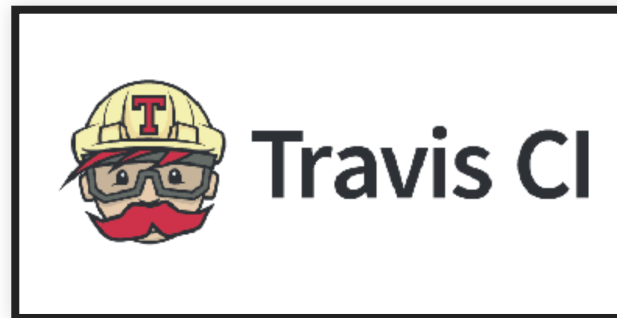
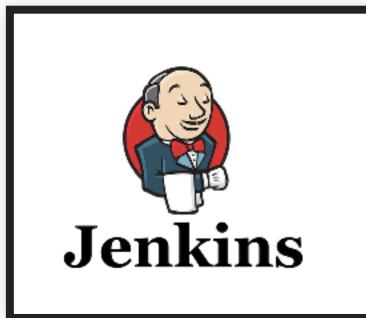
Continuous Deployment bedeutet, dass jede Änderung die Pipeline durchläuft und automatisch in die Produktion übernommen wird, was zu vielen Produktionsimplementierungen pro Tag führt

WERKZEUGE

Zur Umsetzung von CI/CD Ansätzen sind verschiedene Werkzeuge notwendig:

- Zentrale Bereitstellung von Entwicklungsständen und Versionierung, z.B. über Git
- Strategien für Dependencies- und Buildkonfiguration, welche im Idealfall automatisierbar genutzt werden kann, z.B. CMakeLists, Apache Maven oder NPM
- Testfälle welche (automatisierbar) ausgeführt werden können, z.B. in einem Docker-Container
- Möglichkeiten zur Beschreibung von Dienst-Infrastrukturen, insbesondere wenn diese für Tests notwendig sind, z.B. über eine Beschreibung von Infrastruktur wie es mit Docker-Compose möglich ist

- Erstellen von Reports, bzgl. Code-Qualität und Ergebnisse der Test-Ausführung
- Möglichkeiten zur Konfiguration von Systemen, auf denen Tests ausgeführt werden soll, z.B. Ansible
- Möglichkeiten zur Ausführung von Skripten, immer dann wenn Änderungen integriert werden (bzw. in Git ein Commit im Remote bereit gestellt wird)





7.2 GITLAB CI



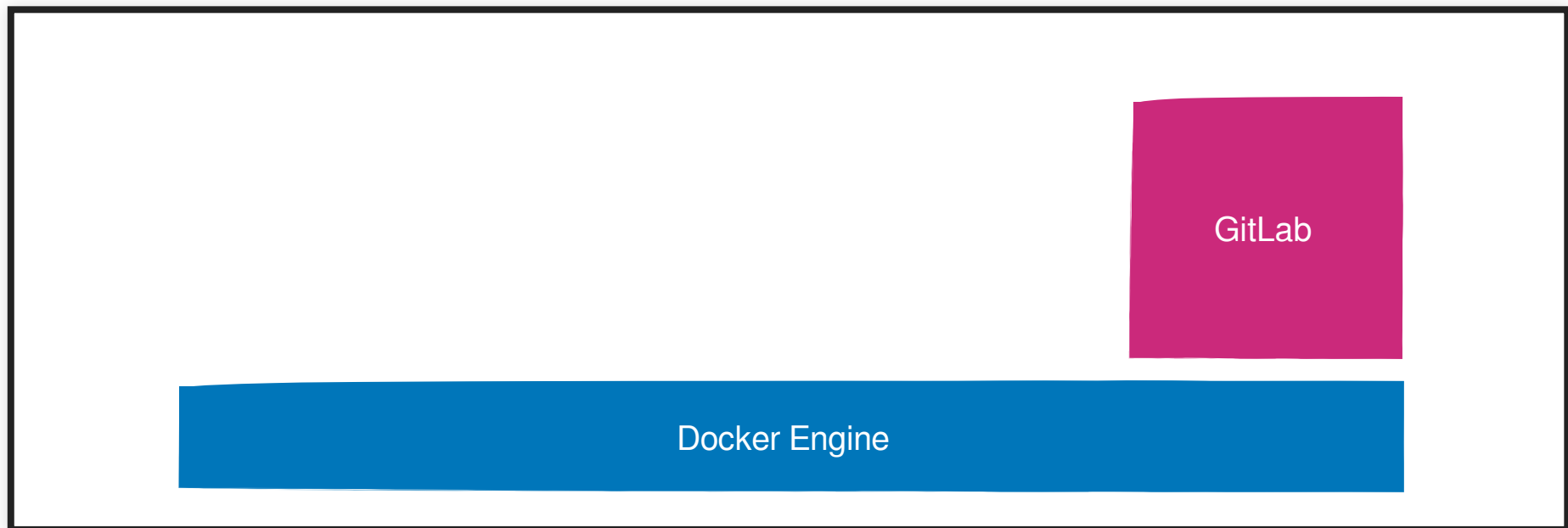
GitLab ist wie GitHub oder Bitbucket eine Plattform zum Management von Software Projekten auf Basis eines Git-Repositories.

Um die Repositories und die Verwaltung und Versionierung von Dokumenten (wie Quellcode) bietet die Plattform Werkzeuge an, u.a. Issue-Tracking, Wiki.

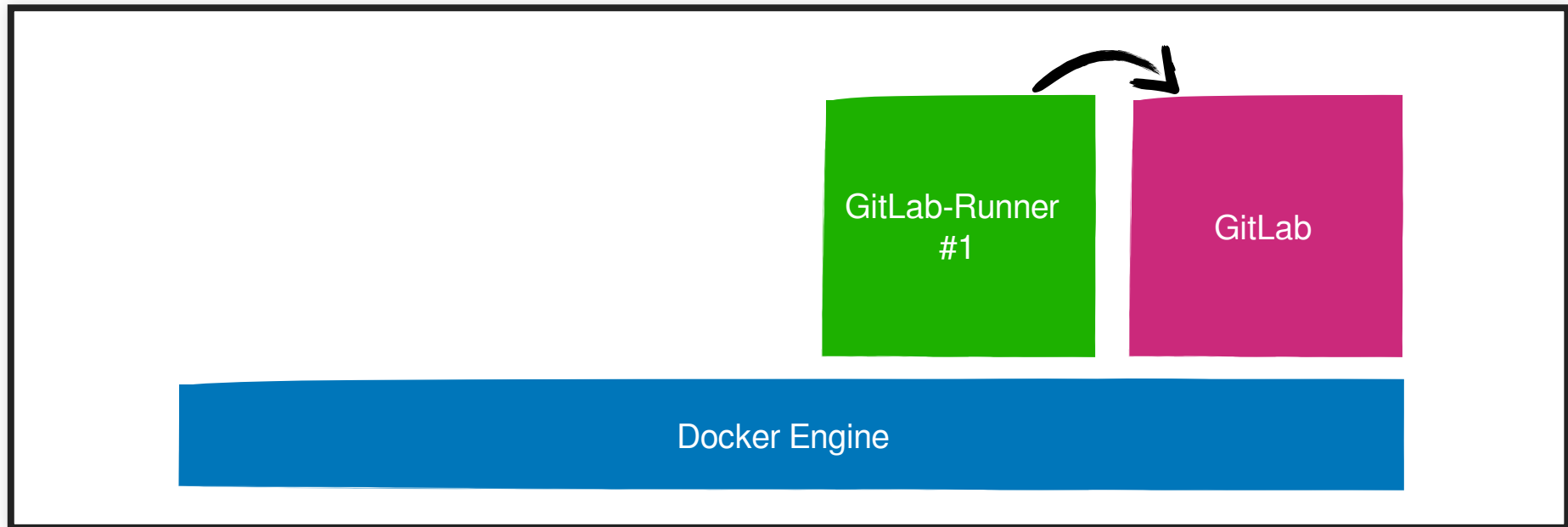
Ein Werkzeug fokussiert auf die Realisierung von CI/CD Pipelines.



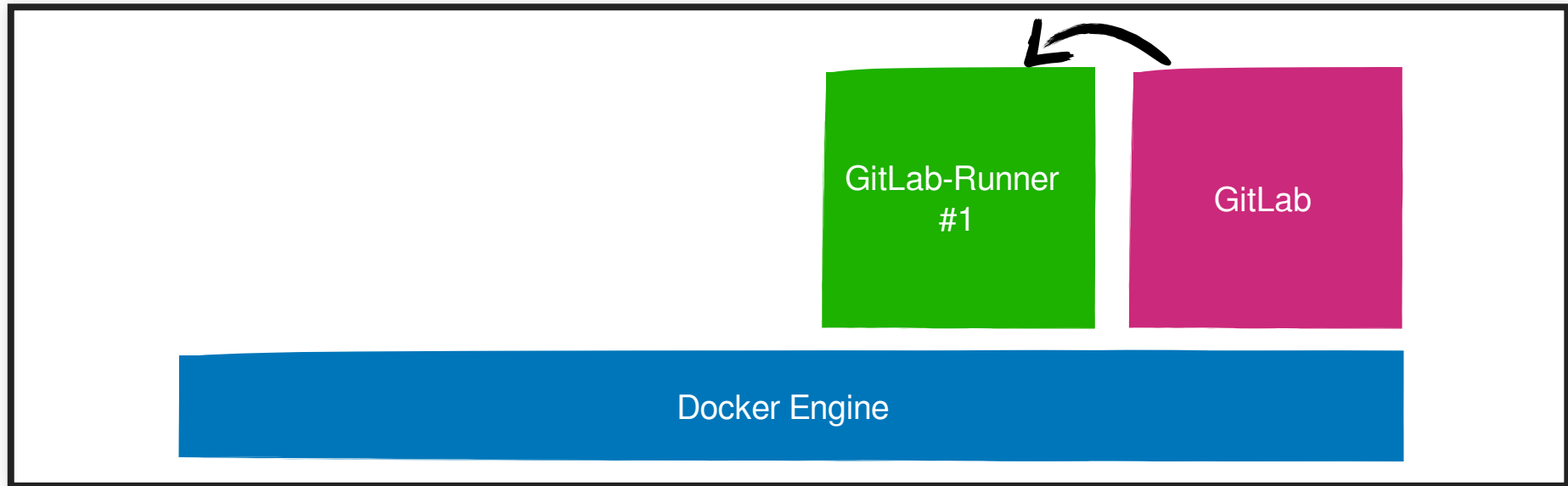
ARCHITEKTUR



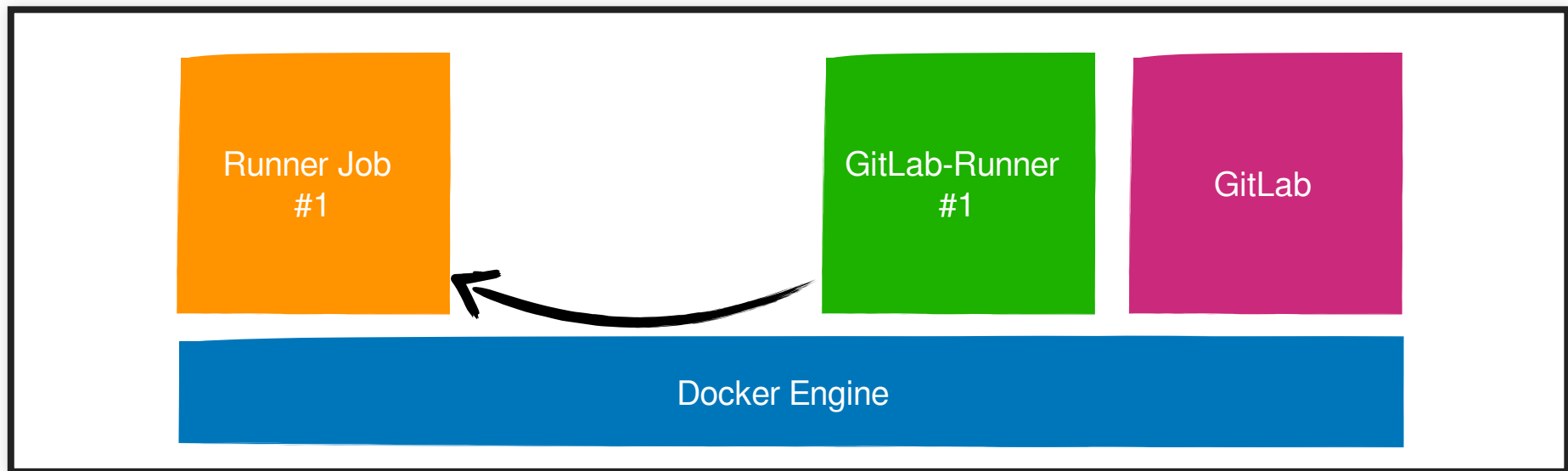
GitLab kann (muss aber nicht) als ein fertiger Docker-Container instanziiert werden.



Zusätzlich zum GitLab ist es möglich sogenannten GitLab-Runner zu deployen, welche die Bauanweisungen abarbeiten. Diese sind unabhängig und registrieren sich bei der GitLab-Instanz.

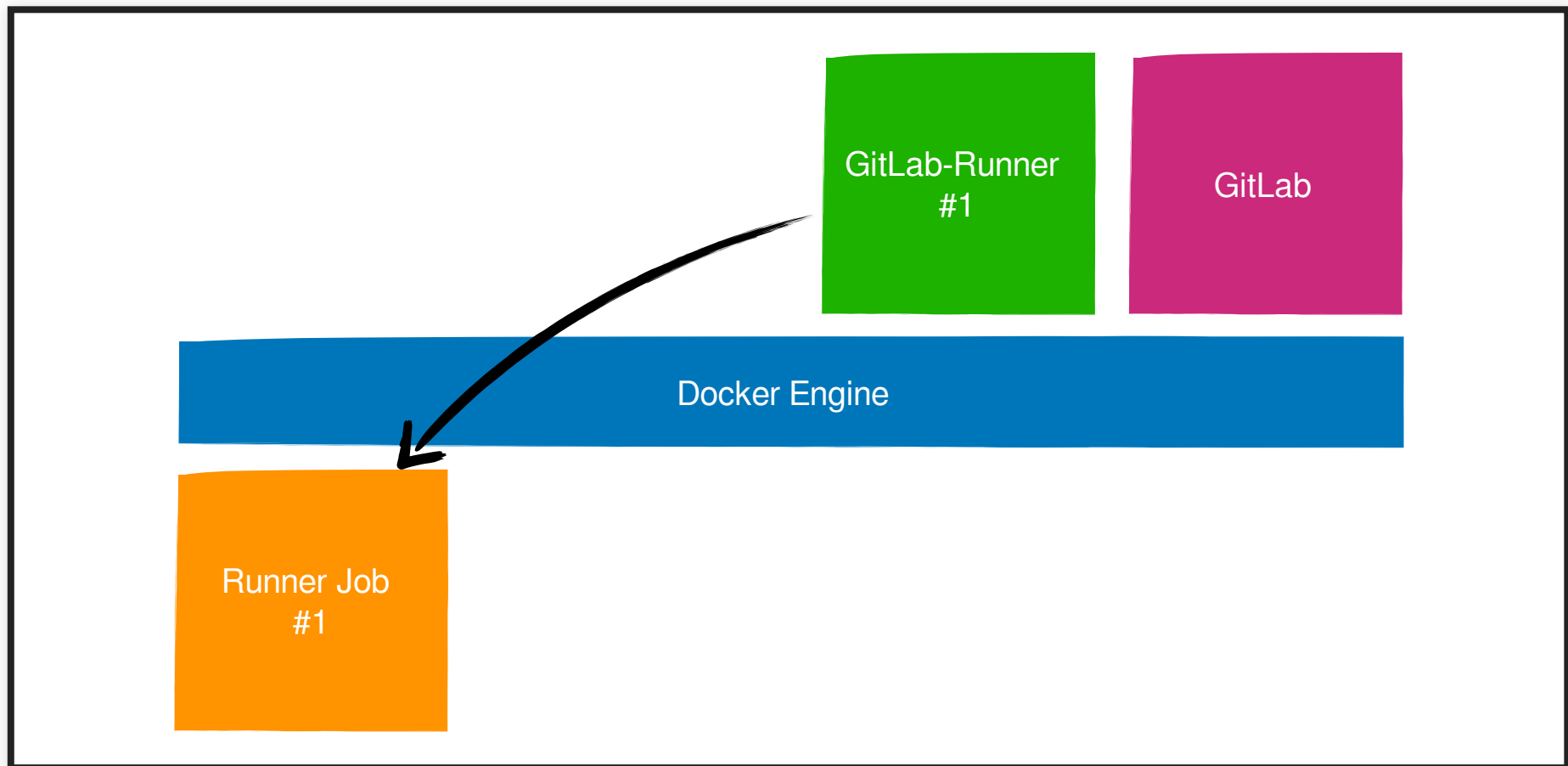


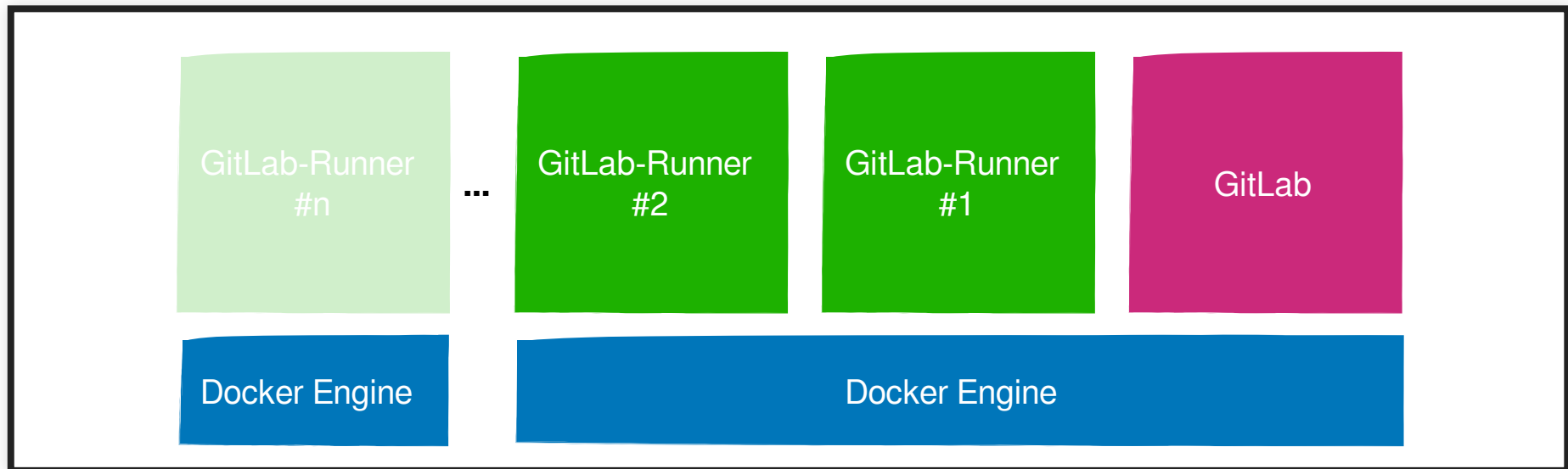
GitLab koordiniert die anstehenden Bauprozesse (aus den Projekten) und weist diese einem Runner zu.



Der Runner selbst arbeitet diese Jobs nicht ab, er startet (z.B.) Container, welche diese Aufgabe übernehmen.

Es besteht die Möglichkeit auch ohne Container direkt auf dem Host zu bauen. Was immer dann hilfreich ist, wenn man systemspezifische Eigenschaften zugreifen möchte.

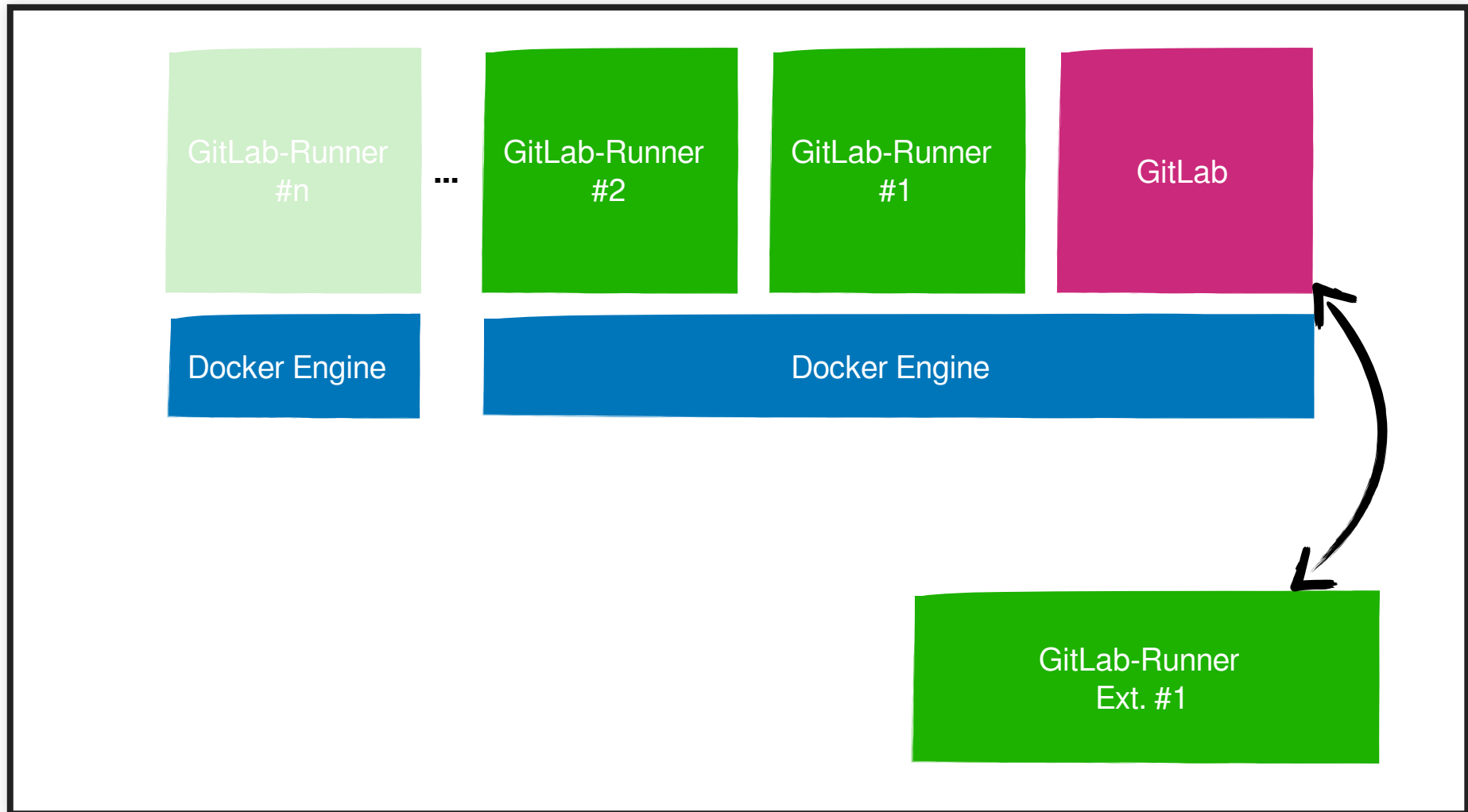




Darüber hinaus sind beliebig viele Runner mit unterschiedlichen Konfigurationen möglich.

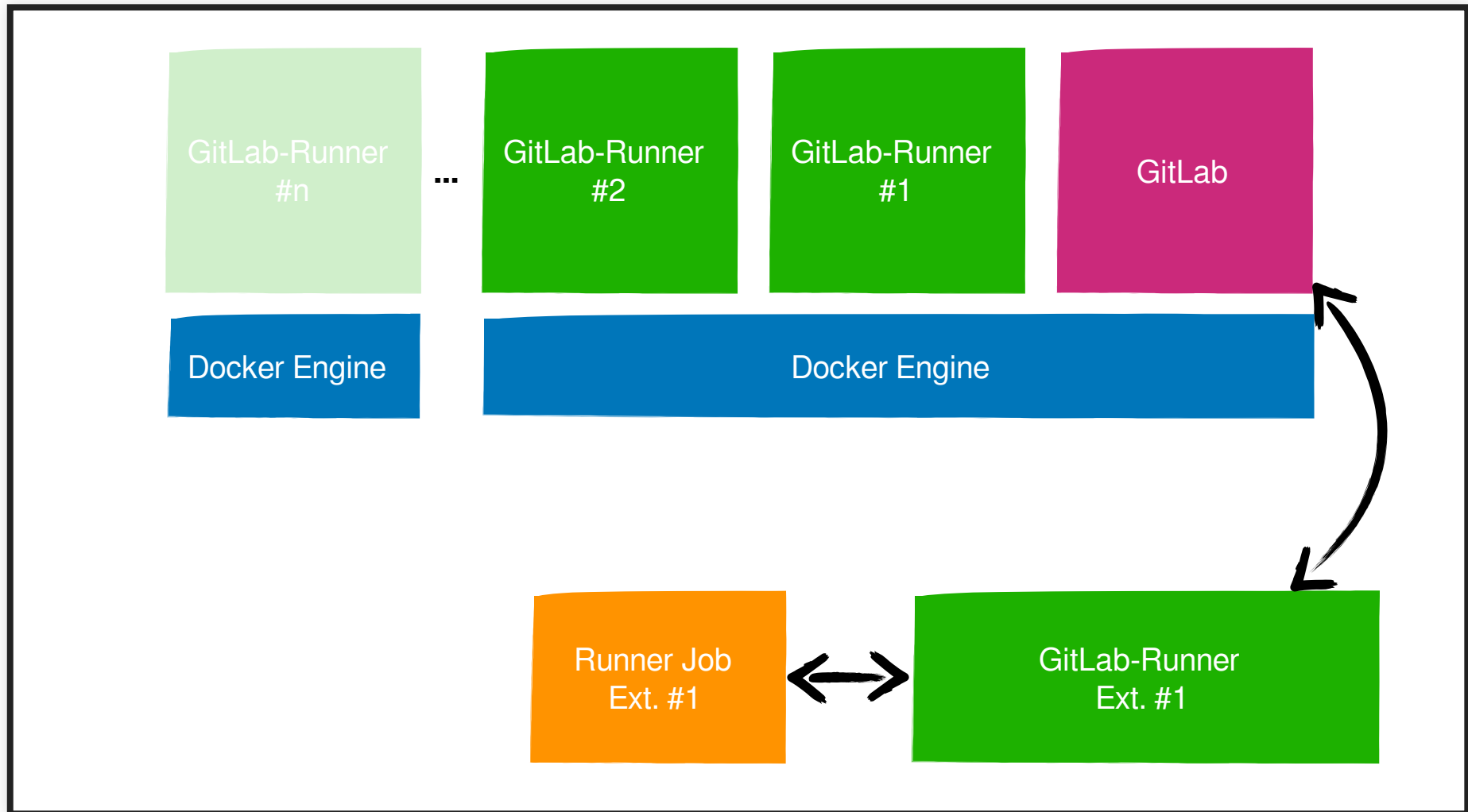


Runner müssen nicht im selben Umfeld ausgeführt werden, nicht einmal innerhalb einer Docker-Umgebung





Auf die Art lassen sich z. B. spezifische Bauumgebungen nutzen (vgl. iOS oder Windows-Anwendungen).





GitLab verwaltet alle verfügbaren Runner und erlaubt den Zugriff auf diese für verschiedene Projekte.

All 1Instance 1Group 0Project 0

Register an instance runner ▾

🕒 ▾

Search or filter results...

🔍

Created date ▾

⌵

Online
1●

Offline
0○

Stale
0🕒

Status ?

Runner

● Online

#1 () Instance
Version 15.3.3 · thi-case-runner-01
🕒 Last contact: 6 minutes ago 🖨 10.10.240.41 🕒 1 📅 Created 2 days ago

✎

⏸

✖

DESKRIPTOR

Wie kann eine CI-Pipeline konfiguriert werden?

- GitLab nutzt eine YAML-Datei je Projektrepository zur Beschreibung der Pipeline
- Im Projekt-Root mit Namen `.gitlab-ci.yml`
- Die `.gitlab-ci.yml` soll die verschiedenen Bauaufgaben in Form von Phasen bzw. Stages beschreiben

```
job1:  
  script: 'do-some-script-stuff'  
job2:  
  script: 'do-other-script-stuff'
```

Ein erstes Beispiel mit GitLab-CI

```
build:  
  script: 'echo "Hello, World!" > foo'
```

Wird der Job ausgeführt, wird die Datei foo erzeugt und mit Inhalt gefüllt.

Dies hat keinen Einfluss auf das Repository und das Ergebnis kann als Artefakt des Bauprozesses angesehen werden.

CI-Skripte können je Job mehrere Anweisungen enthalten

```
build:
  script:
    - 'mkdir build'
    - 'echo "Hello, World!" > build/foo'
```

Mehrere Anweisungen werden als Array in YAML angegeben.

Before- und After-Skript

```
job1:  
  script: 'do-some-script-stuff'  
  before_script: 'do-something-before'  
job2:  
  script: 'do-other-script-stuff'  
  after_script: 'do-something-after'
```

Es gibt Aufgaben die davor und danach ausgeführt werden ...
... mit `before_script` und `after_script`.



Erweiterung des Beispiels

```
build:  
  before_script: 'mkdir build'  
  script: 'echo "Hello, World!" > foo'
```

GitLab-CI und Docker

Wird der Runner mit Docker ausgeführt und hat Zugriff auf den Docker-Socket, können Jobs in Docker-Containern ausgeführt werden.

```
job1:  
  script: "do-some-script-stuff"  
  image: "somedockerimage:latest"  
  before_script: "do-something-before"  
job2:  
  script: "do-other-script-stuff"  
  after_script: "do-something-after"
```

Hierfür kann das Image angegeben werden, in welchem die Anweisungen ausgeführt werden sollen.



Wird das Image vor den Jobs angegeben, gilt dies als Default-Image für alle Jobs

```
image: "somedockerimage:latest"
job1:
  script: "do-some-script-stuff"
  before_script: "do-something-before"
job2:
  script: "do-other-script-stuff"
  after_script: "do-something-after"
```



Das Beispiel könnte ein Alpine-Image als Basis verwenden

```
build:
  before_script: 'mkdir build'
  image: alpine:3.14
  script: 'echo "Hello, World!" > foo'
```


Artefakte der Pipeline

```
job1:
  script: "do-some-script-stuff"
  image: "somedockerimage:latest"
  before_script: "do-something-before"
  artifacts:
    paths:
      - somefile.txt
job2:
  script: "do-other-script-stuff"
  after_script: "do-something-after"
```

Artefakte, die aus dem Bauprozess entstehen müssen benannt werden und können abschließend über die GitLab Oberfläche bezogen werden.



Das Beispiel könnte die foo-Datei und/oder den build-Ordner als Artefakt angeben

```
build:
  before_script: 'mkdir build'
  image: alpine:3.14
  script: 'echo "Hello, World!" > foo'
  artifacts:
    paths:
      - foo
```

... oder ...

```
# ...
artifacts:
  paths:
    - build/
```

Default Stages

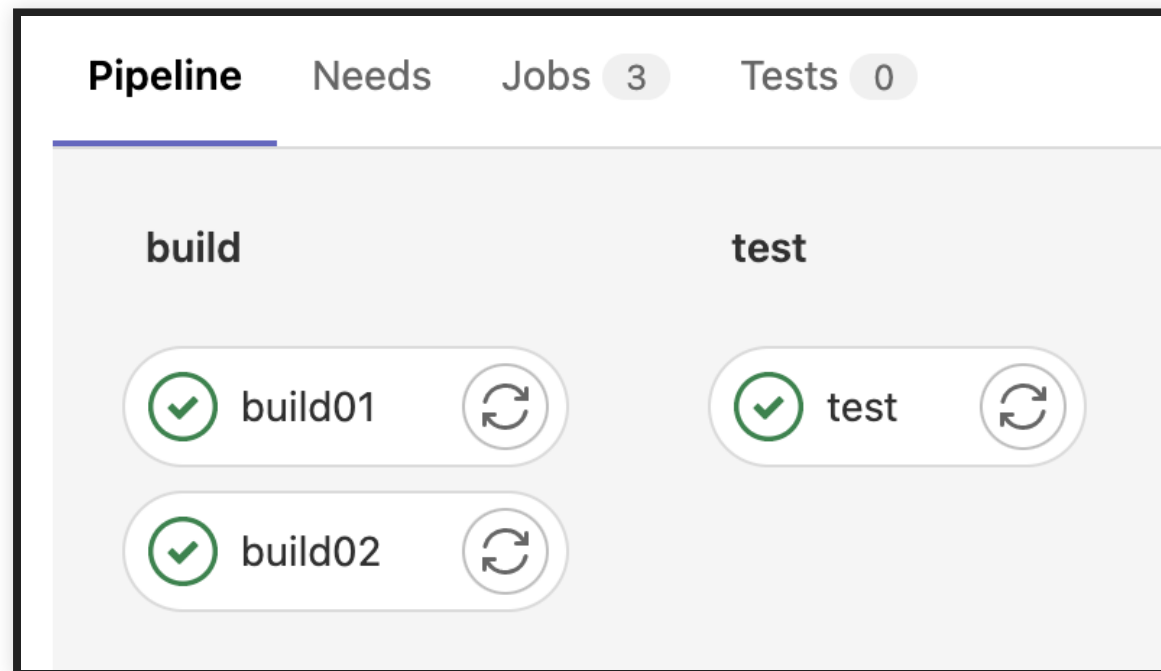
Die Pipeline ist in Stages unterteilt, welche je Job angegeben werden können. Die Default Stages sind: `.pre`, `build`, `test`, `deploy`, `.post`, welche in dieser Reihenfolge abgearbeitet werden.

```
job1:
  script: "do-some-script-stuff"
  image: "somedockerimage:latest"
  stage: build
  before_script: "do-something-before"
  artifacts:
    paths:
      - somefile.txt
job2:
  script: "do-other-script-stuff"
  stage: test
  after_script: "do-something-after"
```

Beispiel mit mehreren Stages

```
image: alpine:3.14
build01:
  stage: build
  script: 'echo "Hello, World!" > foo'
build02:
  stage: build
  script: 'echo "Hello, World!" > bar'
test:
  stage: test
  script: 'echo "Hello, World!" > stuff'
```

Gibt es mehrere Jobs (z.B. `build01` und `build02`) je Stage (z.B. `build`) können die Aufgaben parallel abgearbeitet werden.





BEISPIELE

Beispiel mit CMake

```
build:
  stage: build
  before_script:
    - 'apk add cmake boost-dev build-base'
  script:
    - 'mkdir build'
    - 'cd build'
    - 'cmake ..'
    - 'make'
  artifacts:
    paths:
      - build
```



Beispiel mit Apache Maven

```
maven-build:  
  image: maven:3-jdk-8  
  script: "mvn clean package -B"  
  artifacts:  
    paths:  
      - target/*.jar
```




Erweiterung des Beispiel mit Apache Maven über verschiedenen Phase

```
image: maven:3-jdk-8
stages:
  - build
  - test
  - deploy
maven-build:
  stage: build
  script: "mvn compile -B"
maven-test:
  stage: test
  script: "mvn test -B"
maven-deploy:
  stage: deploy
  script: "mvn deploy -B"
```



Beispiel mit LaTeX

```
build:
  stage: build
  image: blang/latex:ctanfull
  script:
    - latexmk -pdf main.tex
  artifacts:
    paths:
      - "*.pdf"
```

Bedingungen

```
image: maven:3-jdk-8
stages:
  - test
  - deploy
maven-test:
  stage: test
  except:
    - main
  script: "mvn test -B"
maven-deploy:
  stage: deploy
  only:
    - main
  script: "mvn package -B"
```

Das Beispiel führt Tests in jedem Branch außer `main` aus. Java Archive werden nur im Branch `main` erzeugt.



BUILD DOCKER-IMAGE



Wie gestaltet sich der Bauprozess wenn Docker-Images erstellt werden?

Docker-Images werden in Registries bereitgestellt

- Das Projekt benötigt ein `Dockerfile`
- Im CI-Skript muss `docker build` aufgerufen werden
- *Wichtig:* CI-Jobs werden (u.a.) in Docker-Containern ausgeführt; wird ein Image mittels `docker build` gebaut, wird somit ein Docker-Bauprozess aus einem Docker-Container gestartet, was nicht ohne weiteres möglich ist
- Hierfür gibt es einen speziellen Dienst (`docker:dind`) und die Möglichkeit als Image "`docker:latest`" zu verwenden, die Verwendung der Docker-CLI im CI-Skript erlaubt
- Damit ein Bereitstellen des Ergebnisses möglich wird, ist zusätzlich eine Registry notwendig, z.B. `hub.docker.com`
- GitLab bietet eine eigene Registry

GitLab Docker Registry

In jedem Projekt können Docker Images verwaltet werden. Nutzer können sich lokal mittels Nutzernamen und Passwort an der Registry anmelden und Images beziehen:

```
example/ $ docker login
example/ $ docker login case-projects-repositories.rz.fh-ingolstadt.de
Username: ...
Password: ...
```

- Meldet in der Default-Registry an
- Meldet in der Registry vom THI-GitLab für die Vorlesung an

Images bereitstellen

```
example/ $ docker build -t test .  
example/ $ docker tag test case-projects-repositories.rz.fh-ingolstadt.de/yo  
example/ $ docker push case-projects-repositories.rz.fh-ingolstadt.de/your-u
```

... oder ...

```
example/ $ docker build -t case-projects-repositories.rz.fh-ingolstadt.de/yo  
example/ $ docker push case-projects-repositories.rz.fh-ingolstadt.de/your-u
```


GitLab-Credentials

Damit GitLab das selbe ebenfalls in einem CI-Skript machen kann, werden die Zugangsdaten benötigt. Für den Login in die selbe GitLab-Instanz können vordefinierte ENV genutzt werden:

- `CI_REGISTRY`: Host der GitLab-CI-Registry
- `CI_REGISTRY_USER`: Nutzernamen mit dem die Anmeldung durchgeführt werden soll
- `CI_REGISTRY_PASSWORD`: Passwort für die Anmeldung
- `CI_REGISTRY_IMAGE`: Name des Images, enthält den Host



Beispiel für einen Job, der Docker-Images erzeugt und bereitstellt

```
services:
  - docker:dind
container-build:
  stage: dockerize
  only:
    - main
  image: docker:latest
  script:
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t $CI_REGISTRY_IMAGE .
    - docker push $CI_REGISTRY_IMAGE
```

Einloggen in der Registry mittels `login`, anschließend bauen mittels `build` und den gebauten Container mittels `push` in die Registry bewegen.

Für die Kommunikation zwischen Job-Container und umliegender Docker-Engine ist der Service `docker:dind` notwendig.



TEMPLATES

```
.job_template: &job_configuration
  image: ruby:2.6
  services:
    - postgres
    - redis
test1:
  <<: *job_configuration
  script:
    - test1 project
test2:
  <<: *job_configuration
  script:
    - test2 project
```

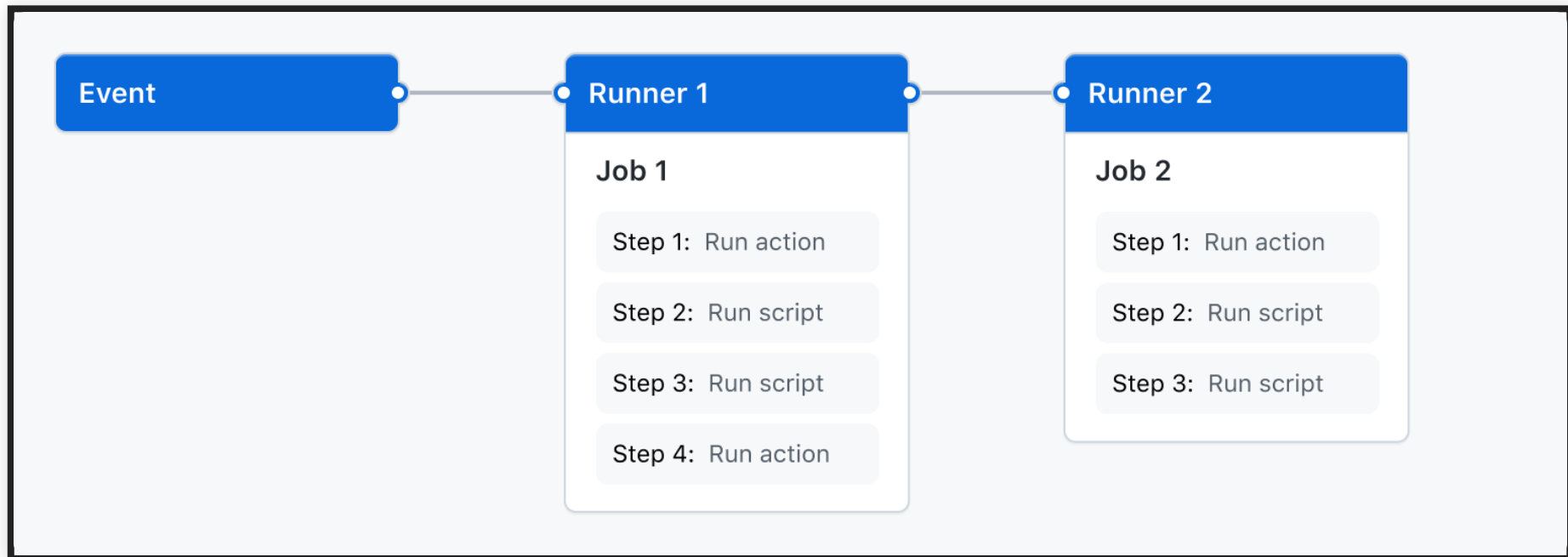
- Versteckte Konfiguration mit dem Namen `job_configuration`
- Verwendung als Basis-Configuration im Job test1...
- ... und test2



7.3 GITHUB ACTIONS



Wie GitLab sind GitHub Actions eine Möglichkeit zur Automatisierung von Bau-, Test- und Deploymentprozessen.



Es lassen sich Pipelines definieren, welche mit Pull Requests ausgeführt werden.

Quelle: <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>

Begriffe im Kontext von GitHub Actions

- Workflows: Automatisierbarer und konfigurierbarer Prozess, der ein oder mehrere Jobs ausführt. Workflows werden in YAML-Dateien konfiguriert, welche im `.github/workflows` Ordner des Repositories abgelegt werden
- Events: Ereignisse im Repository, z.B. Pull-Requests, Issue erzeugt oder ein neuer Commit
- Jobs: Beschreibt eine Menge von Schritten, die von dem selben Runner ausgeführt werden sollen
- Actions: Erlaubt es umfangreiche Aufgaben für CI/CD in eigene Anwendungen auszulagern
- Runners: Ein Server, welche Workflows ausführen.



Beispiel Workflow `.github/workflows/learn-github-actions.yml`

```
name: learn-github-actions
run-name: ${ github.actor } is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

Quelle: <https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>



```
name: learn-github-actions
run-name: ${ github.actor } is learning GitHub Actions
on: [push]
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

- Name des Workflows
- Bezeichnung für eine Ausführung des Workflows
- Event, wann genau dieser Workflow ausgeführt werden soll
- Jobs in diesem Workflow
- Name des Jobs in diesem Workflow



The screenshot shows the GitHub Actions interface for a workflow named 'learn-github-actions'. The workflow is in a 'Success' state. The left sidebar shows the 'Summary' tab selected, with a list of jobs including 'check-bats-version'. The main content area displays the workflow details, including the trigger 'on: push' and a list of jobs. The job 'check-bats-version' is highlighted with a green checkmark and a duration of 8s. The interface includes a 'Re-run all jobs' button and a 'Workflow file' link.

1. Name des Workflows

2. Bezeichnung für diese Ausführung des Workflows

3. Name des Jobs in diesem Workflow

```
# ...
jobs:
  check-bats-version:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-node@v3
        with:
          node-version: '14'
      - run: npm install -g bats
      - run: bats -v
```

- Gibt an, dass dieser Job auf einem Ubuntu ausgeführt werden soll
- Definiert die Schritte die ausgeführt werden sollen, dies können Actions oder Shell-Commands sein
- Verwendet die Action `actions/checkout@v3`, diese Action führt einen Clone im Runner durch
- Verwendet die Action `actions/setup-node@v3`, diese Action installiert NodeJS mit der Version 14
- Führt Shell-Commandos aus



check-bats-version
succeeded 21 seconds ago in 8s

Search logs

> ✓ Set up job2s

> ✓ Run actions/checkout@v31s

> ✓ Run actions/setup-node@v30s

> ✓ Run npm install -g bats1s

▼ ✓ Run bats -v0s

1 ▶ Run bats -v

4 Bats 1.8.2

> ✓ Post Run actions/setup-node@v30s

> ✓ Post Run actions/checkout@v30s

> ✓ Complete job0s

Wird der Workflow ausgeführt, werden die einzelnen Schritte abgearbeitet. Die Aktionen können Post- und Pre-Anweisungen definieren.

ACTION

Actions erlauben die Zusammenfassung verschiedener umfangreicher Aufgaben in einer Action, um diese dann in einem Workflow zum Einsatz zu bringen. Auf diese Art und Weise lassen sich Workflows übersichtlicher gestalten und komplexe Aufgaben auslagern.

Actions lassen sich an den folgenden Stellen platzieren

- Einem anderen, öffentlichen Repository
- Im Repository wo auch die Workflow-Datei liegt
- In einem veröffentlichten Image auf Docker Hub

Verfügbare Actions finden sich im [Marketplace von GitHub](#).



Actions in öffentlichen Repositories

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
```

Hierfür wird die der Owner und der Repository-Name verwendet:
`{owner}/{repo}@{ref}`. Entsprechend lässt sich ein öffentliches **Repository** zu `actions/checkout` finden. In diesem Repository findet sich eine `action.yml`.



Eigene Aktionen hinzufügen

Wird im selben Repository unter `.github/actions/` eine neuer Ordner mit zugehöriger Action-Definition in YAML platziert...

```
├── hello-world (repository)
│   ├── .github
│   │   ├── workflows
│   │   │   └── my-workflow.yml
│   │   └── actions
│   │       └── hello-world-action
│   │           └── action.yml
```

... kann diese wie folgt in einem Job Verwendung finden:

```
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: ../.github/actions/hello-world-action
```



Actions über Docker-Images

```
jobs:
  my_first_job:
    steps:
      - name: My first step
        uses: docker://alpine:3.8
```



ARTIFACTS

Ergebnisse aus den Jobs lassen sich hoch- und runterladen. Hierfür können fertige Actions verwendet werden: `actions/upload-artifact@v3` und `actions/download-artifact@v3`

Beispiel zum Upload

```
# ...  
- run: mkdir staging && cp target/*.jar staging  
- uses: actions/upload-artifact@v3  
  with:  
    name: my-artifact  
    path: staging
```


Beispiel zum Download

```
# ...  
- uses: actions/download-artifact@v3  
  with:  
    name: my-artifact  
    path: stuff  
- name: Display structure of downloaded files  
  run: ls -R  
  working-directory: stuff
```



ÜBUNG: SPRING UND MAVEN MIT GITHUB-ACTIONS

Erstellen Sie ein GitHub-Projekt mit einer einfachen Spring-Anwendung und realisieren Sie die notwendige Workflow-Konfiguration, damit diese gebaut wird.



ÜBUNG: BUILD DOCKER-IMAGE MIT GITHUB-ACTIONS

Erstellen Sie ein GitHub-Projekt mit einer einfachen Spring-Anwendung und realisieren Sie die notwendige Workflow-Konfiguration, damit diese und ein zugehöriges Docker-Image gebaut und publiziert wird.



7.4 DEPLOYMENT

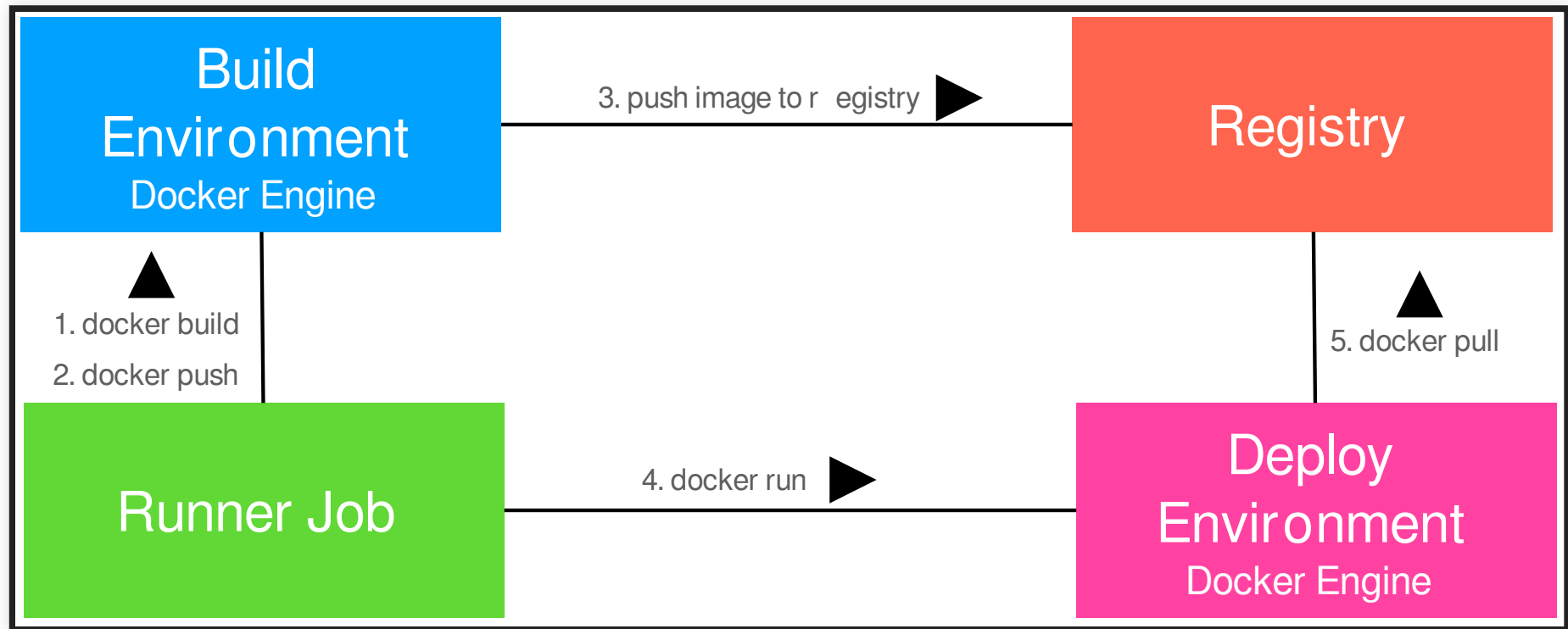


Herausforderung zur Automatisierung von Deployments

Was sind die notwendigen Schritte, um eine VM, einen Container, ein Compose-File oder Kubernetesmanifeste zu deployen?



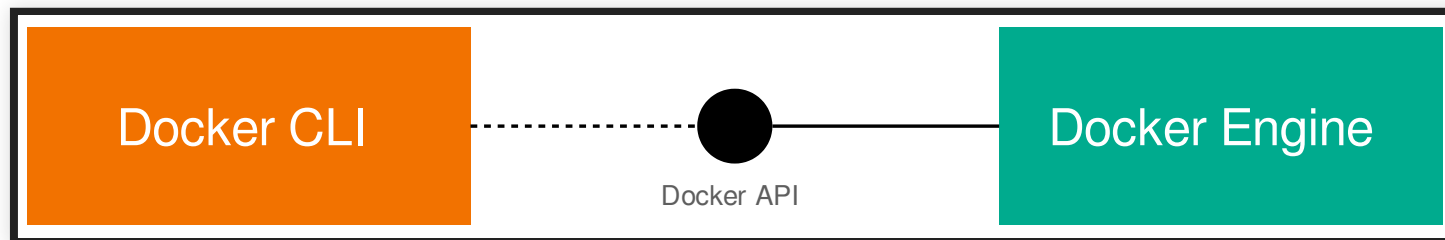
CI/CD WORKFLOW



Zusammenwirkung verschiedener Umgebungen, die die notwendigen Aufgaben abbilden

- Der Runner-Job benötigt Zugriff auf eine Umgebung zum Erstellen von Images, bei einem Setup mit Docker / GitLab-CI wird dies über den speziellen Service `docker:dind` realisiert
- Das in der Umgebung entstandene Image muss anschließend in einer Registry bereitgestellt werden (ggf. sind Credentials notwendig)
- Der Befehl zum Ausführen muss anschließend in einer Zielumgebung platziert werden
- Diese muss anschließend in der Lage sein, das Image zu beziehen (ggf. ebenfalls Credentials notwendig)

- Um den Zugriff z.B. auf die Docker-Engine in einer Zielumgebung zu ermöglichen, ist der Zugriff auf die Docker API notwendig
- Diese TCP/IP basierte Schnittstelle kann über **verschiedene Wege** gesichert werden
 - Per SSH
 - Per TLS (HTTPS) und Server / Client Zertifikaten














GITLAB-CI VARIABLES



Credentials, Zertifikate usw. lassen sich über Variablen in einem Projekt oder einer Gruppe verwalten. Damit kann die Information hinterlegt werden, wie auf die Umgebungen zugegriffen werden soll.

Type	↑ Key	Value	Protected	Masked	Environments	
Variable	EXAMPLE1 	***** 	×	×	All (default)	
Variable	EXAMPLE2 	***** 	✓	×	All (default)	
Variable	EXAMPLE3 	***** 	✓	✓	All (default)	
<div><button>Add variable</button><button>Reveal values</button></div>						



Mit einem einfachen CI-Skript kann ein Blick in die Umgebungsvariablen geworfen werden:

```
step1:  
  script:  
    - printenv
```



Wird der CI-Skript in einem gesicherten Branch ausgeführt, sind alle Variablen verfügbar, die Ausgabe aber ggf. maskiert

```
EXAMPLE1=Hello, World!  
EXAMPLE2=12345!  
EXAMPLE3=[MASKED]
```

Wird in einem ungesicherten Branch (z.B. Feature-Branch) ausgeführt, fehlen Variablen:

```
EXAMPLE1=Hello, World!
```



BEISPIELE



Beispiel mit Docker und TLS

```
deploy-to-abc123:
  variables:
    DOCKER_HOST: tcp://abc123.example.de:2376
    DOCKER_TLS_VERIFY: 1
  image: docker:latest
  script:
    - mkdir -p ~/.docker
    - echo "$ABC123_CA_CERT" > ~/.docker/ca.pem
    - echo "$ABC123_CERT" > ~/.docker/cert.pem
    - echo "$ABC123_KEY" > ~/.docker/key.pem
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker stack deploy --with-registry-auth --compose-file=docker-compose
  environment:
    name: staging
    url: https://abc123.example.de/
```

Die Zielumgebung meldet sich an der Registry von GitLab an, damit Images geladen werden können, anschließend wird das lokal verfügbare Compose-File zur Zielumgebung gesendet, um das Deployment durchzuführen.

Beispiel mit Web-Content

```
deploy-report:
  before_script:
    - 'which ssh-agent || ( apk add --no-cache openssh-client )'
    - eval $(ssh-agent -s)
    - echo "$REPORTS_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - 'echo "$REPORTS_KNOWN" > ~/.ssh/known_hosts'
    - chmod 644 ~/.ssh/known_hosts
  script:
    - 'ssh -p 3022 reports@abc123.example.de "mkdir -p /var/www/reports/${CI_PROJECT_NAMESPACE}"'
    - 'scp -P 3022 -r target/site/. reports@abc123.example.de:/var/www/reports/${CI_PROJECT_NAMESPACE}'
  environment:
    name: reports
    url: https://abc123.example.de/reports/${CI_PROJECT_NAMESPACE}/${CI_PROJECT_NAME}
```

Als weiteres Beispiel ein Upload von Artefakten, z.B. für Reports oder Webseiten, welche ebenfalls Zugriffsmöglichkeiten benötigen.

Hier ist es notwendig alle erforderlichen SSH-Konfigurationen zu realisieren, damit der entfernte Zugriff möglich ist.

Weitere Möglichkeiten

Es gibt über individuelle Konfigurationen und Setups hinaus ebenfalls die Möglichkeit zum Tool-Einsatz. Diverse Werkzeuge erlauben das Durchführen von Deployments. Eine einfache Möglichkeit wäre zum Beispiel Portainer.io. Für einen Service-Stack kann ein Webhook erzeugt werden, welcher das aktualisieren des Deployments erlaubt.

Redeploy from git repository

This stack was deployed from the git repository <https://case-projects.rz.fh-ingolstadt.de/cloud-native-development/portainer-stack-example.git> .
Update `docker-compose.yml` in git and pull from here to update the stack.

Automatic Updates ☒

⚠ Any changes to this stack or application made locally in Portainer will be overridden, which may cause service interruption. Do you wish to continue?

Mechanism Polling Webhook

Webhook Copy link

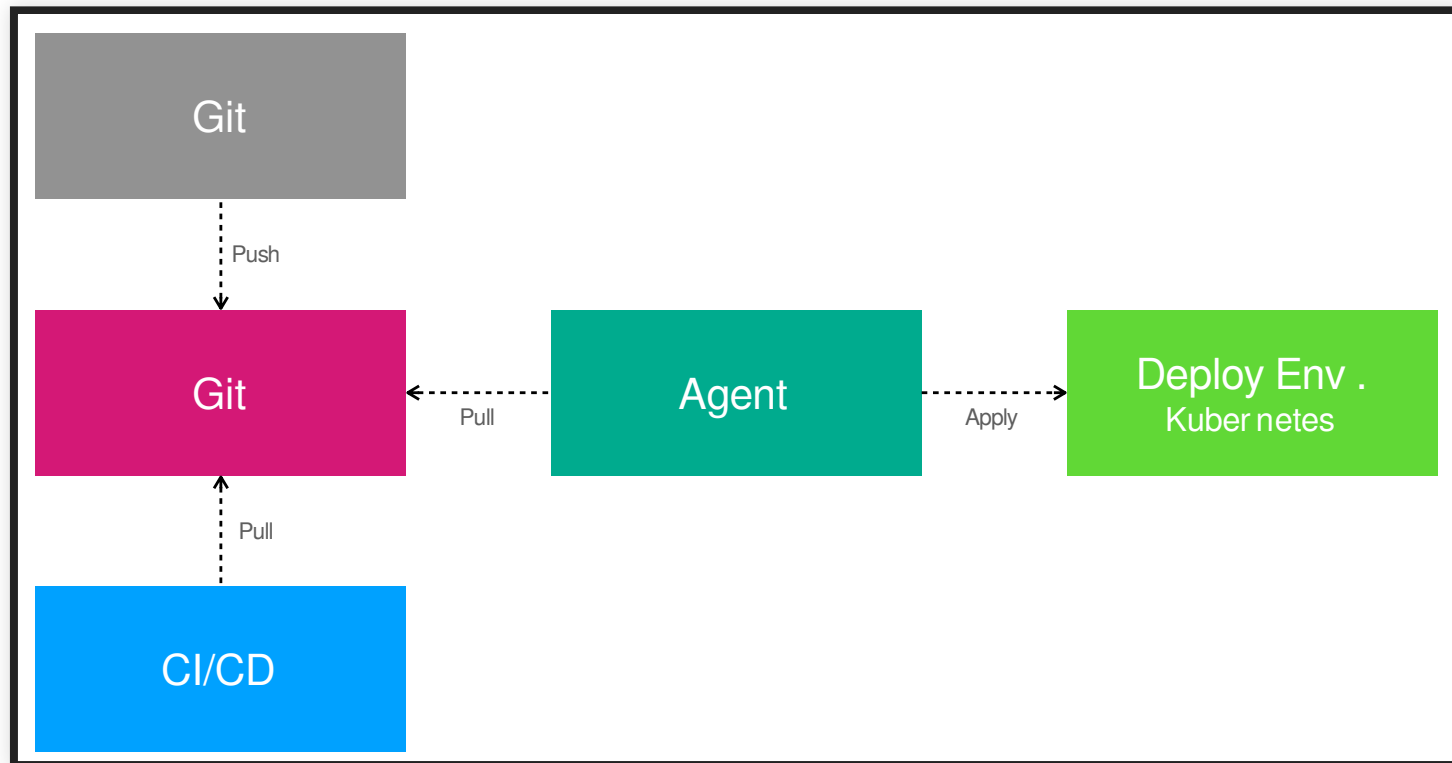
Pull latest image ☐ Business Edition Feature

Force Redeployment ☐ Business Edition Feature



GITOPS MIT GITLAB

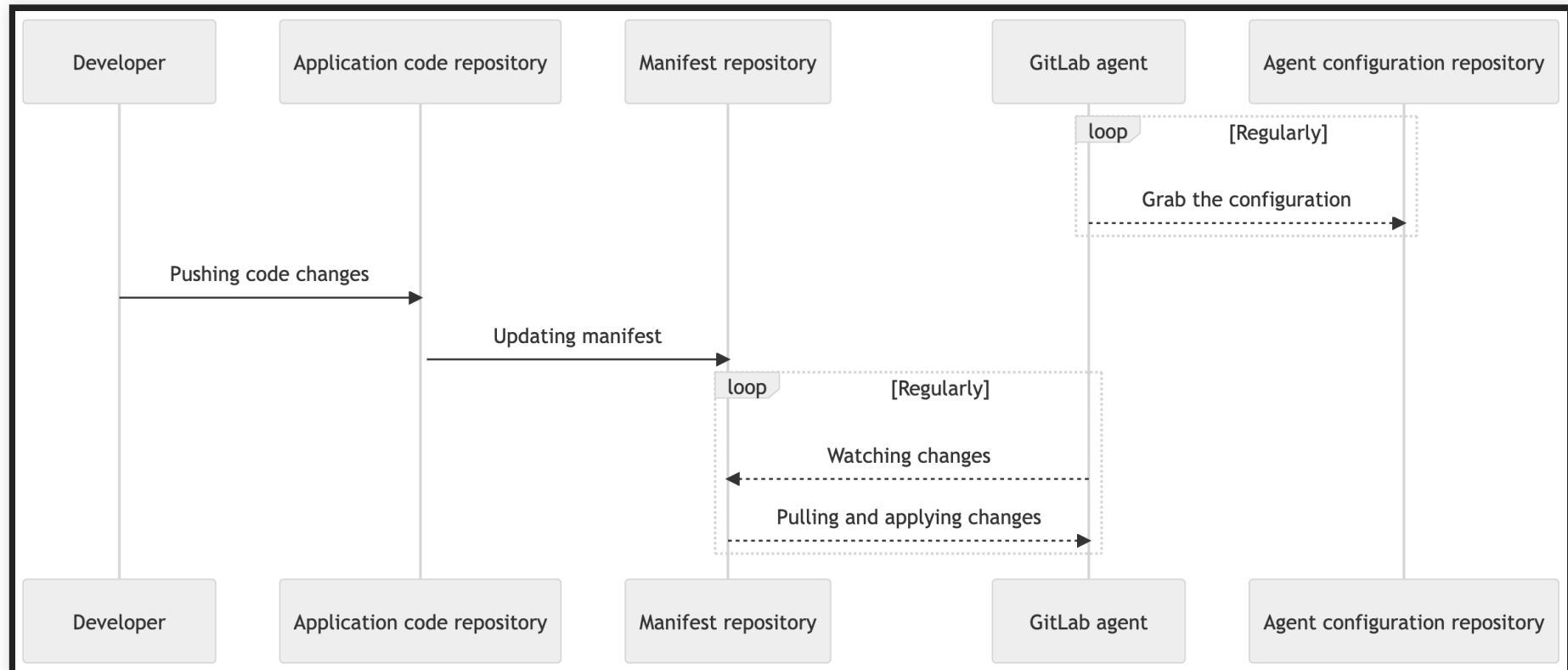
Allgemeiner Ansatz



GitOps setzt Git-Repositories als Single Source of Truth ins Zentrum

- Mittels CI werden Schritte ausgeführt, welche Tests, Package und Delivery umfassen
- Trigger ist hier nicht das externe Event, wie ein Git-Push, sondern über Pull-Requests
- Ein Agent oder Git-Operator überwacht (mittels Pull) das Repository und weiß was zu tun ist, wenn Änderungen existieren (neue Commits)

Vorgehen in GitLab



Quelle: <https://docs.gitlab.com/ee/user/clusters/agent/gitops.html>

Kubernetes und GitLab

Wird **Kubernetes und GitLab** verwendet, wird ein GitOps-Agent im Kubernetes mittels Helm installiert, welcher die Änderungen überwacht.

Im Repository liegt dann eine Konfigurationsdatei, welche die Details für das Deployment enthält.