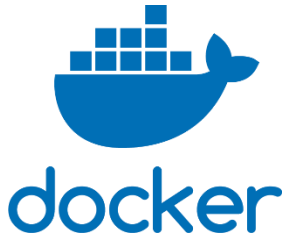
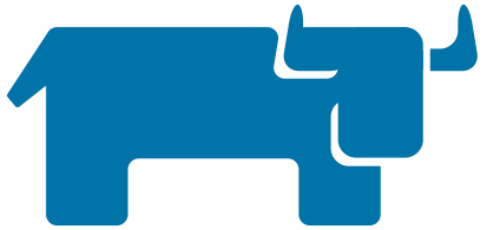




Build, Pack, Deploy

How to get your build into a container to production

Florian Fordermaier, Developer OpenSpace 2017, 13.10.2017

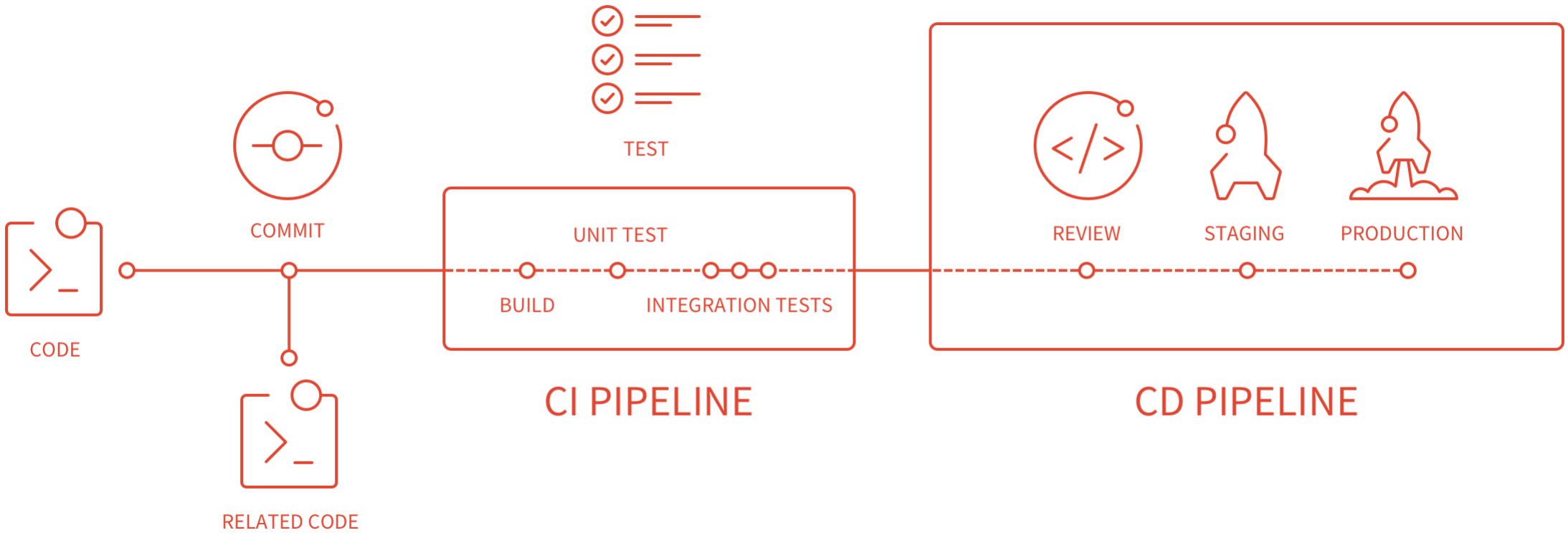


Bild: https://docs.gitlab.com/ee/ci/README.html

Ablauf

- Warm up
- GitLab Setup
- Continous Integration für unser Projekt
- Rancher Setup
- Continous Deployment für unser Projekt

whoami

Florian Fordermaier

@codedevote

Founder & CEO Sarooma GmbH (01/2016)

Freiberuflicher Berater / Entwickler (08/2002)

WARM UP

Startpunkt

- Vagrant Boxes mit Ubuntu 16.04, Docker Engine 17.06-ce und > 20 Docker Images
- Vagrantfiles anpassen
  - CPU und Memory konfigurieren
  - Forwarded Ports / Private Network
- Beide Boxes
  - vagrant up
  - vagrant ssh
  - docker version
  - docker images

LOKALES GITLAB SETUP

Herausforderungen

- Lokale Testumgebung, nicht im Public Internet
- SSL Zertifikate
  - Docker Registries by default https
  - Self-Signed funktioniert, ist aber aufwändig
  - Insecure Registries ebenfalls aufwändig
- SMTP Server
  - Nicht zwingend notwendig
  - Schöner ist‘s mit ☺

gitlab-net – unser Docker-Netzwerk

- Custom Docker Netzwerk
- Alle Container werden in dieses Netzwerk eingehängt
- Docker DNS gewährleistet Namensauflösung über Containernamen

TASK: Erstelle ein Docker Netzwerk mit dem Namen **gitlab-net**

Mailcatcher – SMTP für Testumgebung

- Docker Image **schickling/mailcatcher**
- SMTP auf Port 1025
- WebUI auf Port 1080

TASK: Starte Mailcatcher im *gitlab-net* mit Namen *mailcatcher* mit Portforwarding für Port 1080 und stelle sicher, dass der Container auch nach Neustart des Hosts wieder gestartet wird.

Überprüfe, dass die WebUI via Browser auf dem Hostsystem erreichbar ist. Warum ist sie das?

Das Leid mit dem SSL Zertifikat

- Self-Signed
  - Muss auf „jedem“ Host verfügbar sein und explizit vertraut werden
  - Muss in „jedem“ Container verfügbar sein
  - Ist nicht für Docker Registries vorgesehen → insecure Docker Registry trotz HTTPS
- Workaround
  - Let‘s Encrypt Zertifikat für öffentlich zugänglichen Server
  - **devspace.ff-solutions.de**
  - Nutzung des Zertifikats durch „Umbiegen“ von /etc/hosts bzw. Nutzung eines DNS Servers

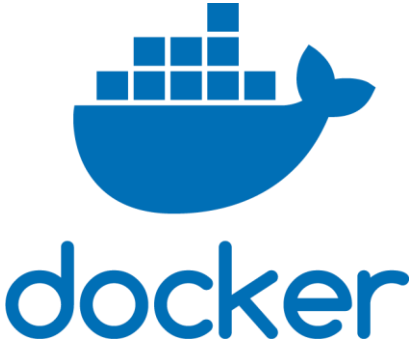
Simple Container DNS

- Docker Image blackikeeagle/developer-dns
- Benötigt Zugriff auf /var/run/docker.sock, um Containerstarts und Stops überwachen zu können
- Erzeugt DNS Einträge für Container, die mit ENV VIRTUAL_HOST gestartet werden

TASK: Starte Container *gitlab-dns* mit Zugriff auf */var/run/docker.sock* im *gitlab-net.* Weise dem Container eine fixe IP aus dem *gitlab-net zu* (inkl. dessen subnet) und stelle sicher, dass der Container auch nach Neustart des Hosts wieder gestartet wird.

GitLab Komponenten

- GitLab
- Redis
- MySQL / PostgreSQL
- Docker Registry



TASK: GitLab starten

- Docker Image: *gitlab/gitlab-ce:10.0.2-ce.0*
- Portfreigaben: 443 (WebInterface) und 5001 (Registry)
- Name *gitlab,* Netzwerk *gitlab-net*
- ENV: VIRTUAL_HOST=devspace.ff-solutions.de (DNS)
- Volumes (Host:Container)
  - /vagrant/gitlab-config:/etc/gitlab
  - ~/gitlab-data/ logs:/var/log/gitlab
  - ~/gitlab-data/data:/var/opt/gitlab
  - ~/gitlab-data/ssl:/etc/gitlab/ssl
- Neustart, falls Host neu gestartet wird

TASK: SSL Zertifikat für Gitlab

http://devspace.ff-solutions.de/devspace.ff-solutions.de.crt

http://devspace.ff-solutions.de/devspace.ff-solutions.de.key

Beide Files in ~/gitlab-data/ssl ablegen

/etc/hosts

- **Server Box und das eigene Hostsystem** um Hosteintrag erweitern:

```
127.0.0.1  devspace.ff-solutions.de
```

- Windows: c:\windows\system32\drivers\etc\hosts
- MacOS, Linux: /etc/hosts
- Verifizieren durch ping

GitLab Konfiguration

- Zentrale Konfigurationsdatei gitlab.rb (Ruby)
- Wir konfigurieren
  - Externe URL (https://devspace.ff-solutions.de)
  - Externe URL für Registry (https://devspace.ff-solutions.de:5001)
  - SMTP (mailcatcher auf Port 1025)
- *gitlab-ctl reconfigure*

TASK: GitLab Konfiguration überprüfen

- Überprüfen von https://localhost und https://devspace.ff-solutions.de mit curl
- Zugriff auf GitLab Web Interface via Browser auf Hostsystem

KURZER GITLAB ÜBERBLICK

TASK: User, Gruppe und Projekt einrichten

- Einen User anlegen (Mailcatcher!)
- Eine Gruppe **devspace** anlegen
- In der Gruppe **devspace** ein Projekt **TodoApi** anlegen
- Projekt **devspace/TodoApi** klonen
- Github Repo klonen: https://github.com/codedevote/devspace2017
- Entweder Inhalt des Ordners **netcore** oder **nodejs** ins Projektverzeichnis von **devspace/TodoApi** kopieren und pushen

TASK: Manueller Build mit Build-Container

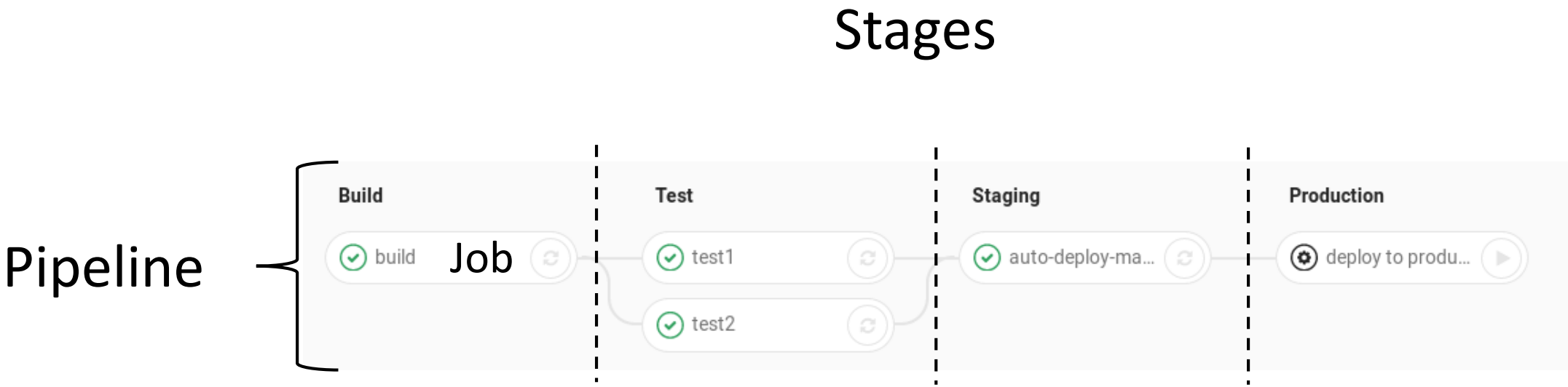
- Das Projekt **devspace/TodoApi** lokal mit Hilfe des Build-Containers bauen
  - **microsoft/dotnet:2.0.0-sdk**
  - **node:8.6.0**
- Ergebnis des Builds lokal mit Hilfe des Runtime-Containers ausführen und mit curl auf die TodoApi zugreifen
  - **microsoft/aspnetcore:2.0.0**
  - **node:8.6.0**

TASK: Test der GitLab Registry

- https://devspace.ff-solutions.de:5001
- Dazu das lokal verfügbare Docker Image **busybox** in die eigene Registry pushen und danach von dort wieder abholen

GITLAB CI

Pipelines, Jobs, Stages



- Pipeline ist eine Gruppe von Stages
- Stage ist eine Gruppe von Jobs
- Jobs einer Stage werden parallel ausgeführt (Voraussetzung: ausreichend Runner)
- Stages werden nacheinander ausgeführt.
- Schlägt ein Job fehl, schlägt auch die Pipeline fehl

.gitlab-ci.yml – Where the magic happens

```yaml
variables:
  - ARTIFACTS: “./build_output”

stages:
  - build
  - test
  - deploy

build_job:
  stage: build
  before_script:
    - execute before script
  script:
    - execute build
  tags:
    - build_runner
  artifacts:
  paths:
    - $ARTIFACTS
```

```yaml
test_job:
  stage: test
  script:
    - execute tests
  when: on_success
  tags:
    - build_runner

deploy_job:
  stage: deploy
  environment: production
  script:
    - perform deployment
  only:
    - master
  tags:
    - deploy_runner
```

GitLab Runners

- Applikation, die die Pipeline-Beschreibung in der .gitlab-ci.yml ausführt
- Shared Runner vs. Specific Runner
- Unterschiedliche Ausprägung (Executors)
  - Shell, Docker, Docker Machine, Kubernetes, SSH, VirtualBox, Parallels
- Docker
  - Containernutzung (d.h. Nutzung eines Build-Containers)
  - Images bauen (d.h. Eigenes Docker-Image bauen)

Docker in GitLab CI

- Containernutzung: Runner mit **docker** Executor
- Nützlich für Build-Container, Test-Container, ...
- Drei Optionen, um **Docker Images** zu bauen
  - Shell Executor (Docker Engine nutzen, die auf Runner Host läuft)
  - DinD Executor (Docker-in-Docker → isolierte Docker-Engine)
  - Docker Socket Binding (Docker-Engine nutzt Engine des Runner-Hosts)

Container Registry

- In GitLab integriert und einfach nutzbar
- Für viele Anwendungsfälle ausreichend
- Externe Registry kann aber konfiguriert und verwendet werden (z.B. Private Docker Hub, Harbor, ...)

TASK: GitLab Runner hinzufügen

- Executor: Docker
- Docker Socket Binding (DinD hat in Testumgebung leider nicht funktioniert ☹)
- Start des **gitlab/gitlab-runner:v10.0.2** Containers
  - DNS
  - Docker Socket Binding
  - Network *gitlab-net*
- Mit RunnerToken (siehe GitLab Web-UI), den Runner bei GitLab registrieren (**gemeinsam** ☺)

GITLAB CI FÜR TODOAPI

TASK: GitLab CI Build Stage für TodoApi

- .gitlab-ci.yml für TodoApi erstellen (im Rootverzeichnis des Projekts)
  - Variable GIT_SSL_NO_VERIFY: "1"
  - Variable ARTIFACTS: "./artifacts" (Ausgabeverzeichnis)
  - Runner: docker
  - in der Build-Stage
    - Before_script: Anzeige der Version des Build-Tools (dotnet, node)
    - Ausführung des Buildskripts
  - Definition der Artefakte

TASK: GitLab CI Build Image Stage für TodoApi

- Dockerfile für TodoApi erstellen
- .gitlab-ci.yml erweitern
  - Image für **build_image** Stage: **docker:17.06.2-ce**
  - in der **build_image**-Stage
    - Runner: docker
    - before_script:
      - Anzeige Informationen über Docker Engine
      - Login in GitLab Registry (*docker login -u gitlab-ci-token -p $CI_BUILD_TOKEN $CI_REGISTRY*)
    - Ausführung des Docker Builds, Push des Images in die Registry mit den Tags [aktuellePipelineID] und [latest] (Nutzung von *$CI_REGISTRY_IMAGE* und *$CI_PIPELINE_ID*)
    - after_script: Cleanup der gebauten Images (Erinnerung: Docker Socket Binding)

TASK: HealthCheck implementieren

- Die TodoApi um einen Endpoint /_health erweitern, der Http Statuscode 200 OK liefert.
- .gitlab-ci.yml erweitern
  - Image für **check_health** Stage: **docker:17.06.2-ce**
  - in der **check_health**-Stage
    - Runner: docker
    - before_script:
      - Login in GitLab Registry
      - Erstellen eines **eindeutigen** Docker Network (Hinweis: PipelineID)
    - script: Start der TodoApi mit **eindeutigem** Namen, danach Start **appropriate/curl** und Überprüfung des Statuscodes des /health Endpoints
      (*curl -sL -w "%{http_code}" "google.de" -o /dev/null*)
    - after_script: Cleanup Images und Docker Network

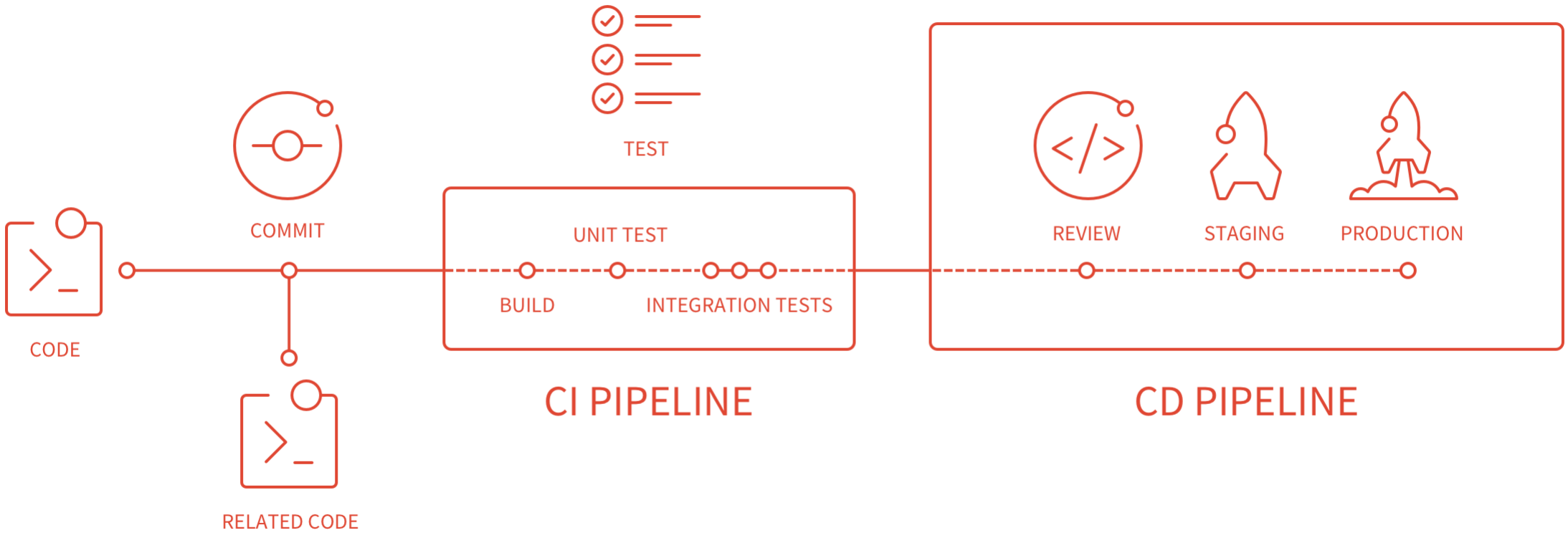
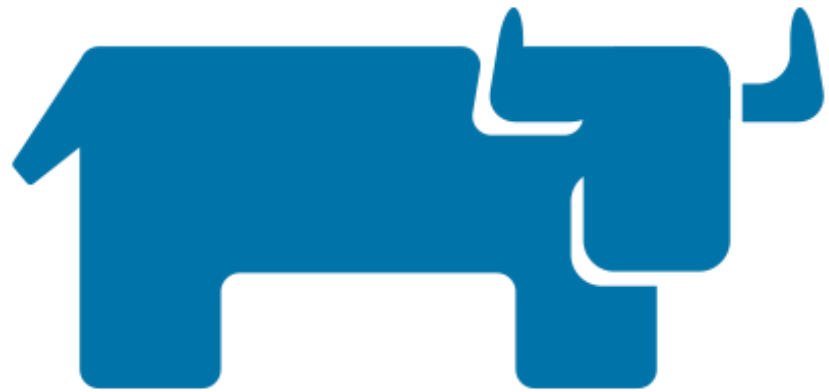
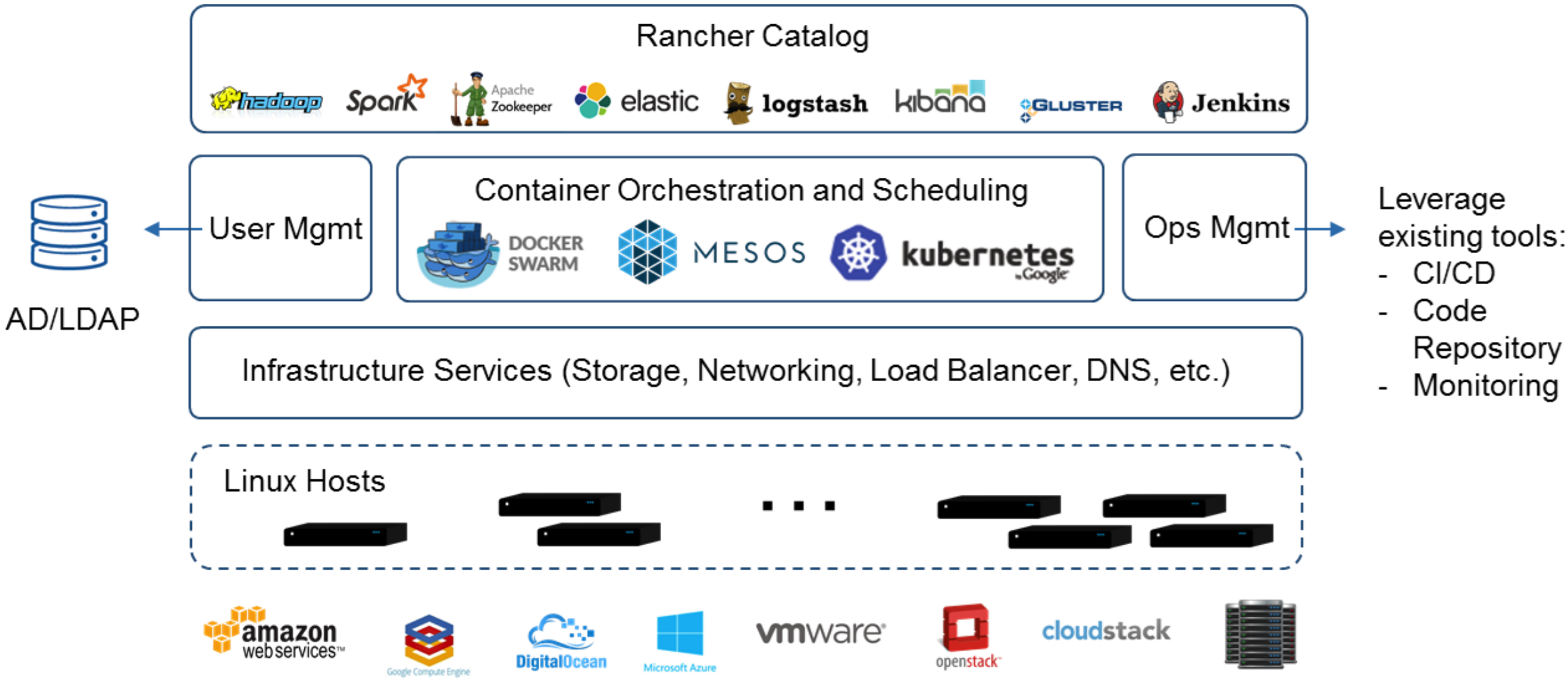


Bild: https://docs.gitlab.com/ee/ci/README.html



RANCHER


Rancher Catalog
hadoop
Spark
Apache Zookeeper
elastic
logstash
kibana
GLUSTER
Jenkins
AD/LDAP
User Mgmt
Container Orchestration and Scheduling
DOCKER SWARM
MESOS
kubernetes
Google
Ops Mgmt
Leverage existing tools:
- CI/CD
- Code Repository
- Monitoring
Infrastructure Services (Storage, Networking, Load Balancer, DNS, etc.)
Linux Hosts
amazon web services
Google Compute Engine
DigitalOcean
Microsoft Azure
vmware
openstack
cloudstack

Aus 1000 Metern

- Infrastructure Orchestration
  - Any cloud, any bare metal
  - Jeder Linux Host ist „nur“ CPU, RAM, lokaler Storage und Netzwerk
- Container Orchestration & Scheduling
  - Unterstützt Swarm, Kubernetes, Mesos, Cattle
- Rancher Catalog
  - Sammlung von „containerized“ Apps, die mit einem Klick deployed werden können
  - Public und Private Catalogs
- Access Control

Infrastructure Components

- Environments
  - „dev“, „test“, „prod“
  - Gruppe isolierter Resourcen mit eigener Zugriffskontrolle
- Hosts
  - Grundlegendste Resource in Rancher
  - Any Linux server (VM oder Bare Metal) + unterstützte Docker Engine Version
- Registries
  - Quelle für Docker Images, public (Docker Hub) oder private (GitLab)
- Certificates
  - SSL Zertifikate (SSL Termination am LoadBalancer)

Hosts

- Any cloud, any hoster
- Können automatisch über Rancher erzeugt werden (für unterstützte Cloudprovider wie Azure, AWS, Digital Ocean, ...)
- Können auch manuell eingebunden werden (Start des **rancher/agent** auf dem entsprechenden Host)

Cattle – Part 1

- IPSec Overlay Netzwerk
  - „Private VPN“ zwischen Hosts und daher auch Containern
  - „**Managed Network**“ je Environment
    - Internal DNS Service
    - Automatische DNS Einträge → Discoverability
- Service
  - 1..n Container desselben Images
  - Service HA
  - Unit of scale
  - Health Checks

Cattle – Part 2

- Stack
  - Gruppe von Services, die zusammen eine Applikation implementieren
  - Eigenes Lifecycle Management
- Load Balancer
  - Managed HAProxy
  - Layer 4 & Layer 7 Load Balancing
  - Alle Container eines Services werden automatisch als Ziele registriert
  - Default: Round Robin
  - Custom HAProxy Konfiguration möglich
- Volumes, Labels, Scheduling, ...

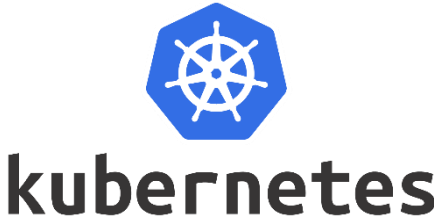
HealthChecks

- rancher/healthcheck Infrastruktur-Service
- 3 HealthCheck Container überwachen 1 Service/Container
- HealthCheck OK, wenn mind. 1 von 3 HealthChecks OK
- Ausnahme: Nur ein Host verfügbar → nur 1 HealthCheck Container
- Funktioniert nur im “Managed” Network

Upgrading Services

- In-Service upgrades
  - Container eines Services in Batches stoppen und neue Versionen starten
  - Mit CLI Tool **rancher-compose** → Granularität steuerbar (Stack, 1..n Services)
  - Mit (containerized) CLI Tool **cdrx/rancher-gitlab-deploy**
- Rolling Upgrades
  - Nur möglich mit CLI Tool **rancher-compose**
  - Ersetzen eines Services durch einen neuen Service
  - Inbound Links (z.B. LoadBalancer, andere Services, …) werden automatisch aktualisiert

Rancher 2.0



The Rancher container management platform has become an increasingly popular way to manage containers—it's been used to deploy tens of thousands of clusters. With Kubernetes becoming the fastest developing technology in the container ecosystem, we saw an opportunity. And rebuilt Rancher on Kubernetes. Rancher 2.0 provides the same great user experience on all Kubernetes clusters, whether they are managed using Rancher or existing Kubernetes clusters. The result? It's now even easier for you to adopt Kubernetes and run containers.

(https://rancher.com/docs/rancher/v2.0/en/faq/)

TASK: Rancher Server starten

- Docker Image **rancher/server:v1.6.10**
- WebUI auf Port 8080 freigeben
- Bind-mount /var/lib/mysql
- Überprüfe, dass die WebUI via Browser auf dem Hostsystem erreichbar ist.
- Zugriffskontrolle aktivieren und eigenen Admin-User anlegen

Environment, Host und Registry

- Environment **production** erstellen
- Custom Host (Vagrant Box 2) hinzufügen
- GitLab Registry hinzufügen
- Stack **devspace** erstellen
- Service **TodoApi** hinzufügen

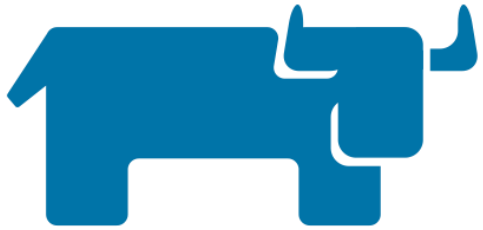
TASK: Rancher Agent starten

- Firewall auf der Vagrant Box konfigurieren: Port 80 auf 8080 weiterleiten
- sudo iptables -t nat -A PREROUTING -p tcp --dport 80 -j REDIRECT --to-port 8080

- Nun benötigen wir die **zweite!** Vagrant Box
- IP der ersten Box im Vagrant Private Network ist 192.168.33.10 → /etc/hosts patchen!

TASK: Rancher HealthCheck hinzufügen

- Upgrade **TodoApi** Service via UI
- Hinzufügen des HealthChecks (wichtig: hier wird der containerinterne Port benötigt)



CONTINOUS DEPLOYMENT FÜR TODOAPI

Rancher API Keys und GitLab Secrets

- Rancher Environment API Key in UI generieren
- Als GitLab Secrets des TodoApi Projekts hinterlegen
  - RANCHER_URL (http://192.168.33.10:8080)
  - RANCHER_ACCESS_KEY
  - RANCHER_SECRET_KEY
- Diese Secrets sind während der Ausführung der Pipeline als Environmentvariablen verfügbar

TASK: Deployment Stage

- .gitlab-ci.yml erweitern
  - **deploy_prod** Stage
  - Environment **production**
  - Image **cdrx/rancher-gitlab-deploy**
  - ☺ *docker run -it cdrx/rancher-gitlab-deploy rancher-gitlab-deploy –help*

GitLab Environments

- Übersicht über Deployments je Environment
- Re-Deploy / Rollback
- Link auf Environment (über URL in .gitlab-ci.yml)

TASK: Die Pipeline genießen

- TodoApi verändern
- Änderungen pushen
- Pipeline in GitLab CI nachvollziehen
- Upgrade des TodoApi Services in Rancher UI beobachten

TASK: Datenbank hinzufügen

- Postgres für .NET Core Projekt
- MongoDB für Node Projekt

- Datenbank manuell in **devspace** Stack deployen
- Upgraden des **TodoApi** Service

TASK: Weitere Box / Cloud Host hinzufügen

- Weitere Vagrant Box oder Cloud Host in Rancher **production** Environment einbinden
- TodoApi Service skalieren (2 Instanzen)
- LoadBalancer hinzufügen