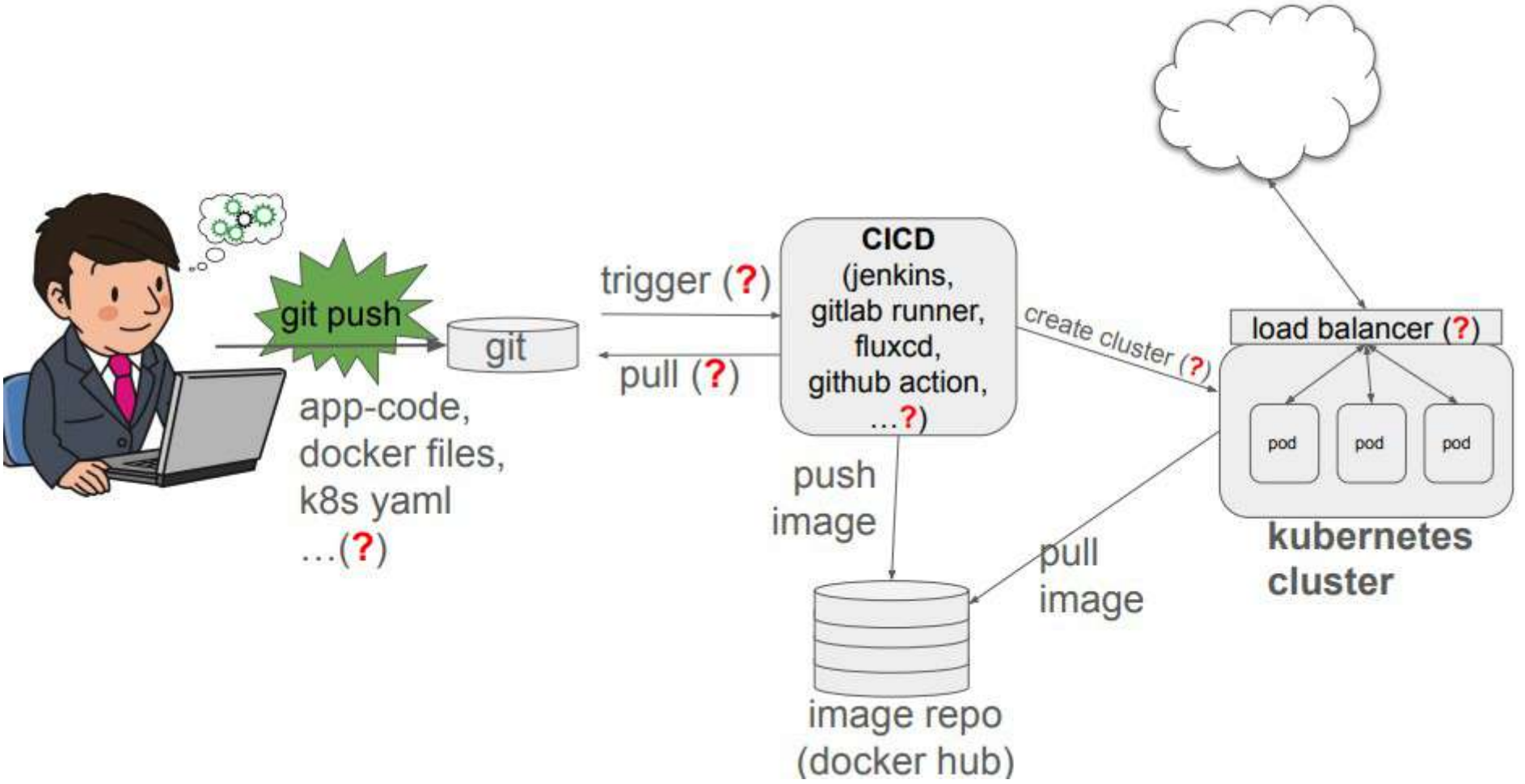
Mijn GitOps ervaringen

matisering

Src: https://www.wordclouds.com/

Een vorige keer:

Andere keer?



DevOps / GitOps?

- DevOps
  - Automatiseren van builds, tests, deployments
  - Verschillende tooling (git, jenkins, scripts, docker, Kubernetes, …)
  - Geen vaste bron van “waarheid”
- GitOps
  - Automatiseren van builds, tests, deployments
  - Git is de waarheid
  - Geen vervanging of opvolger van DevOps

Beide streven naar het automatiseren van de (snelle) uitrol van software.

Container images zijn de “bouwstenen”

Waarom Kubernetes?

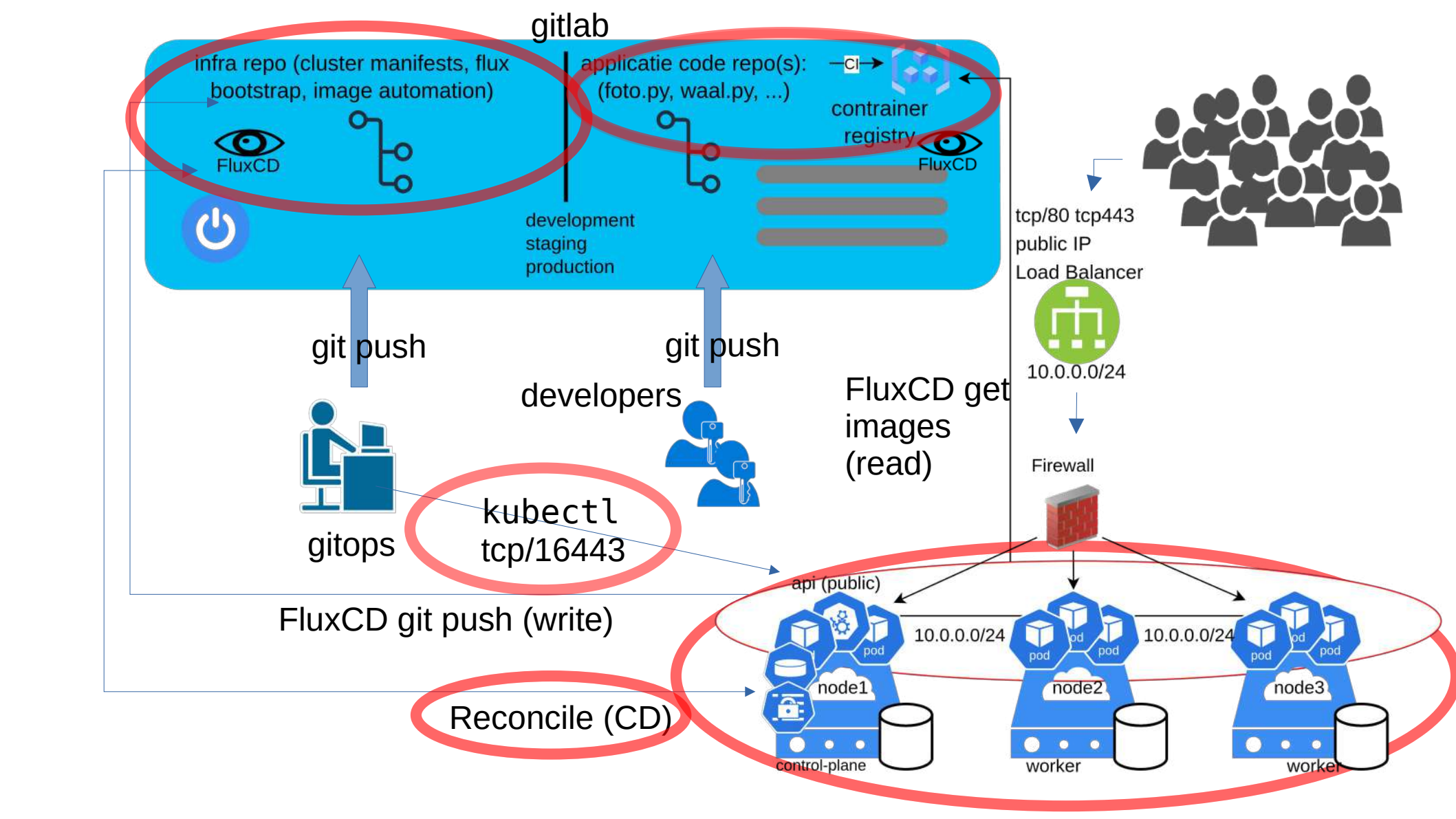
Eerst: waarom containers?

- Bouwstenen voor DevOps / GitOps
- Minder last van “bij mij werkt het..”
- Klassieke server met software gevoeliger voor dependency conflicten
- …

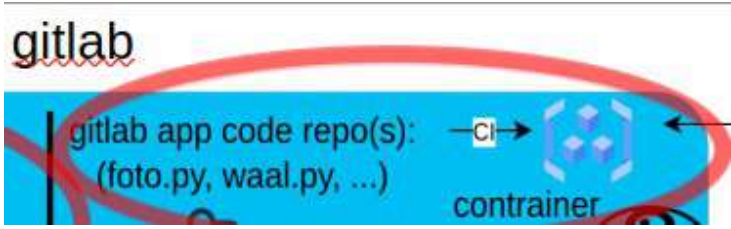
Voor het managen van container images: Kubernetes. Veel extra’s:

- Replica’s (load balancing, (horizontale) schaalbaarheid)
- Geschikt voor meerdere nodes
- Rolling updates
- Declaratief (yaml) → goed voor Git
- Tooling voor automatische deployments (flux, argocd), goed voor GitOps
- Test/productie goed op te zetten (“namespaces”)
- ...

Vergeet nooit: elk voordeel heeft zijn nadeel

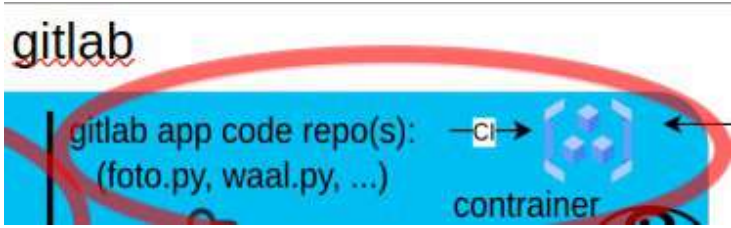

gitlab
infra repo (cluster manifests, flux bootstrap, image automation)
FluxCD
applicatie code repo(s): (foto.py, waal.py, ...)
CI
contrainer registry
FluxCD
development
staging
production
tcp/80 tcp443
public IP
Load Balancer
10.0.0.0/24
git push
git push
developers
FluxCD get images (read)
Firewall
gitops
kubectl
tcp/16443
FluxCD git push (write)
Reconcile (CD)
api (public)
10.0.0.0/24
10.0.0.0/24
pod
node1
node2
node3
control-plane
worker
worker

Applicaties



- Applicatie-code in git
- Dockerfile voor het maken van je image
- Bouwen Container images en pushen naar een container registry
  - Dit gaat met een pipeline gedefinieerd in .gitlab-ci.yml
- Applicatie moet geschikt zijn voor “containerisatie”. Bijvoorbeeld:
  - Gebruik php:fpm, nginx:alpine, mysql:8.x containers voor een “PHP app”
  - Gebruik python:3-x(-slim), postgres:latest containers voor een “Python app”

Het file: .gitlab-ci.yml

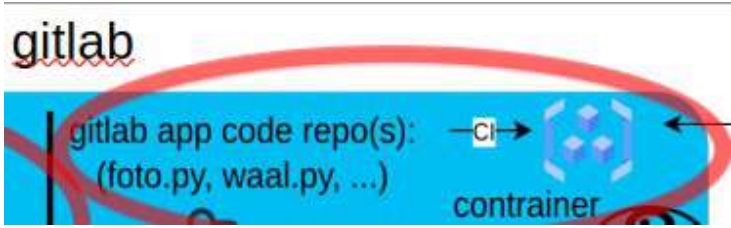


- In de root van je applicatie repositorie
- Maakt een “pipeline” van “commando’s”
- Werkt met stages. Bv.: lint, test, build, deploy
- Wij kijken (deze keer) alleen naar de build
- Opbouw van het file .gitlab-ci.yml:

```
stages    → volgorde van je pipeline
variables → globale variabelen
default   → standaardinstellingen voor alle jobs
jobs      → lint, test, build, deploy
rules     → bepalen wanneer jobs draaien
script    → echte commando’s die uitgevoerd worden
```

```
stages: [build]
variables:
  IMAGEREPO: "$CI_REGISTRY_IMAGE/foto-web"

build_dev:
  …

build_staging:
  …

build_prod:
  …
```

2 build jobs voorbeelden



development

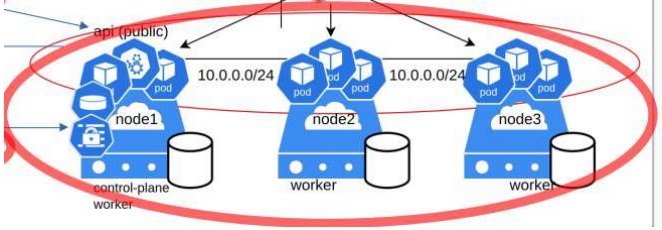
```
# job naam
build_dev:
  # deze job hoort bij de build stage
  stage: build
  # de docker “runner” (docker CLI)
  image: docker:27            #
  Services:
    # een docker daemon is nodig voor bouwen
    - name: docker:27-dind
    - command: ["--mtu=1460"]
  Rules:
    # run deze job niet bij commit tag
    - if: $CI_COMMIT_TAG
      when: never
    # fallback: run in alle andere gevallen
    - when: always
  Script:
    # wat de “runner” runt
    - docker login -u "$CI_REGISTRY_USER" \
              -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - docker build -t "$IMAGEREPO:dev-$CI_PIPELINE_IID-$CI_COMMIT_SHORT_SHA"
  .

    - docker push "$IMAGEREPO:dev-$CI_PIPELINE_IID-$CI_COMMIT_SHORT_SHA"
```

productie

```
build_prod:
  stage: build
  image: docker:27
  services:
    - name: docker:27-dind
      command: ["--mtu=1460"]
  rules:
    # Build only when pushing a tag like v1.2.3
    - if: '$CI_COMMIT_TAG =~ /^v\d+\.\d+\.\d+$/'
      when: always
    - when: never
  script:
    - docker login -u "$CI_REGISTRY_USER" \
           -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"
    - docker build -t "$IMAGEREPO:prod-$CI_COMMIT_TAG" .
    - docker push "$IMAGEREPO:prod-$CI_COMMIT_TAG"
```

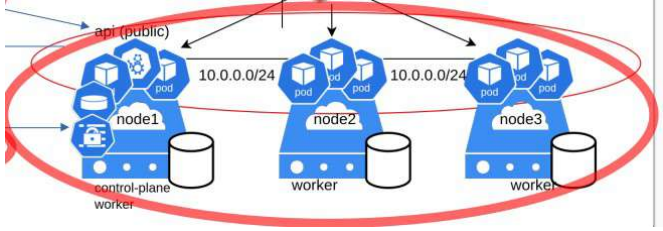
Betere strategie: gebruik goedgekeurd test image voor produktie. Images zijn immutable.

Kubernetes: wat te kiezen?



- Managed kubernetes (AKS, GKE, EKS, Digitalocean, TransIP, Hcloud, ...)
- kubeadm kubernetes (officiele kubernetes.io)
- k3s, microk3s (“minder serieuze” productie, ook lab)
- minikube, kind (lab)

Opbouw Kubernetes 1/2



Control nodes en worker nodes

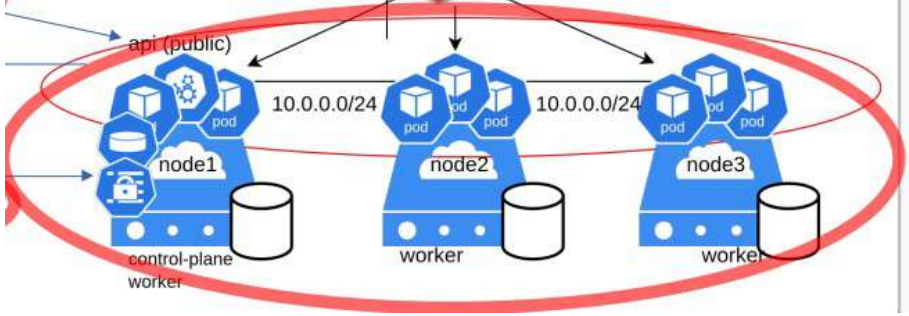
Control nodes. Deze vormen het “control-plane” en bevatten (in principe) alleen de **systeem pods**:

- scheduler - plaatst pods op nodes (houdt rekening met cpu/memory gebruik)
- controller - bewaakt de staat van het cluster
- etcd: key/value store voor het cluster waarin alle resources worden opgeslagen
- api - alle communicatie en commando's gaan via de api (dit is ook waar kubectl tegen praat)
- Minstens 3 control nodes nodig voor High Availability (HA)

Worker nodes. Bevatten de **applicatie pods** maar ook enkele systeem pods:

- DNS pod - DNS-achtige service discovery (koppeling ip's <-> namen)
- proxy pod - verzorgt de routering en load balancing rules
- kube agent ("kubelet") - voor communicatie met de api server. Managed pods
- applicatie pods

Opbouw Kubernetes 2/2



Alle systeem pods runnen in de namespace "kube-system".

Dit wordt allemaal gemaakt tijdens de install

Wat kanttekeningen:

In microk8s zijn veel systeem pods vervangen door systemd services

K3s heeft veel systeem pods vervangen door 1 enkele systemd service (k3s.service)

Kubernetes resources

Kubernetes is opgebouwd uit resources (objects)

Deze hebben het volgende gemeen:

- Hebben een api versie en een “kind” (type)
- CRUD via de kubernetes API
- Worden opgeslagen in de kubernetes database (etcd)

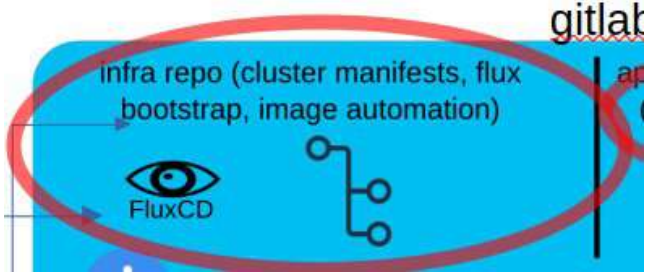
Kubernetes resource	omschrijving
Pod	Kleinste uitvoerbare unit in Kubernetes; bevat één of meer containers
Deployment	Beheert stateless applicaties en zorgt voor scaling, updates en self-healing van Pods
Service	Biedt een stabiel netwerkendpoint en load balancing naar Pods
Ingress	Regelt **externe** HTTP/HTTPS toegang naar Services
PersistentVolume	Concrete opslagresource die aan een PVC gekoppeld wordt
PersistentVolumeClaim	Aanvraag voor persistente opslag door een workload
StorageClass	Definieert hoe opslag wordt aangemaakt
ConfigMap	Opslag van niet-gevoelige configuratiegegevens
Secret	Opslag van gevoelige gegevens zoals wachtwoorden of tokens
StatefulSet	Beheert stateful applicaties met stabiele namen en opslag
DaemonSet	Zorgt dat een Pod op elke node (of geselecteerde nodes) draait
Job	Voert een taak eenmalig uit tot succesvolle voltooiing.
CronJob	Periodieke Jobs volgens een tijdschema.
NameSpace	Logische scheiding van resources binnen een cluster.
ReplicaSet	Zorgt dat een gewenst aantal identieke Pods actief blijft.

Wat zit er in nog meer in git?

We zagen al:

- Je **applicatie code**.
- Je **images**: in een “image (container) registry” (op bv. github of gitlab)

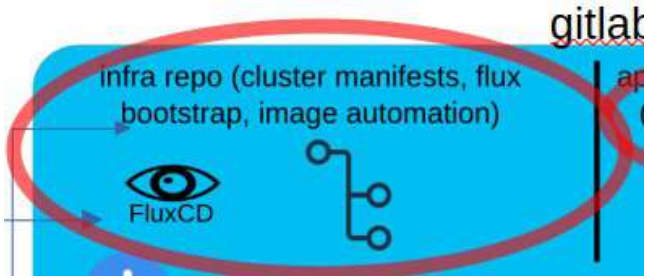
Maar er is meer:



- Je **infra (Kubernetes cluster) code (yaml)**:
  - Config voor je kubernetes resources:
    - deployments, services, ingress, configmaps, secrets, pvc’s, ...
  - Config voor applicaties zonder eigen code (databases, webservers)
  - Config voor deploy (met tools als flux en argocd)
  - Eventuele “helm charts” (de “Kubernetes package manager”)
- Bij gebruik van Terraform: terraform code ook in git (bv. “terraform_repo”)

Voorbeelden directory structuur

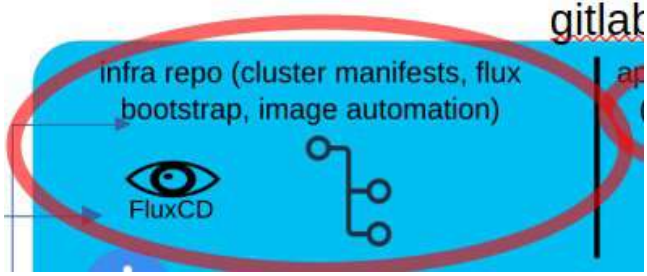
```
microk8s
├── base
│   ├── deployment.yaml
│   ├── kustomization.yaml
│   └── service.yaml
└── overlays
    ├── dev
    │   ├── imagepolicy.yaml
    │   ├── imagerepo.yaml
    │   ├── imageupdateautomation.yaml
    │   ├── ingress.yaml
    │   ├── kustomization.yaml
    │   └── pvc.yaml
    └── prod
        ├── ingress.yaml
        ├── kustomization.yaml
        └── pvc.yaml
```




```
k3s
├── cronjob-sync.yaml
├── deployment.yaml
├── ingress.yaml
├── namespace.yaml
├── pvc.yaml
└── svc.yaml
```

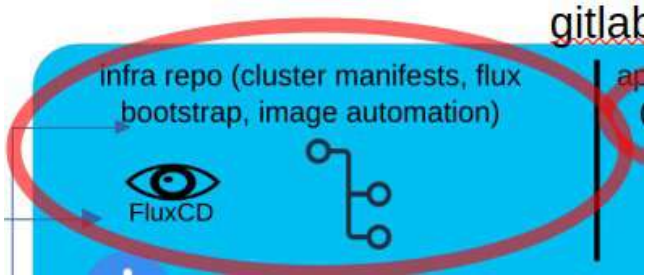
Structuur kubernetes yaml (vb. deployment)

```
Deployment
├── apiVersion (apps/v1, v1, networking.k8s.io/v1, ...)
├── kind       (Deployment, Service, Ingress, ...)
├── metadata
│   └── name
└── spec                          # algemeen
    ├── replicas
    ├── selector
    │   └── matchLabels
    └── template                  # pods
        ├── metadata
        │   └── labels
        └── spec
            └── containers
                ├── name
                ├── image
                └── ports
```



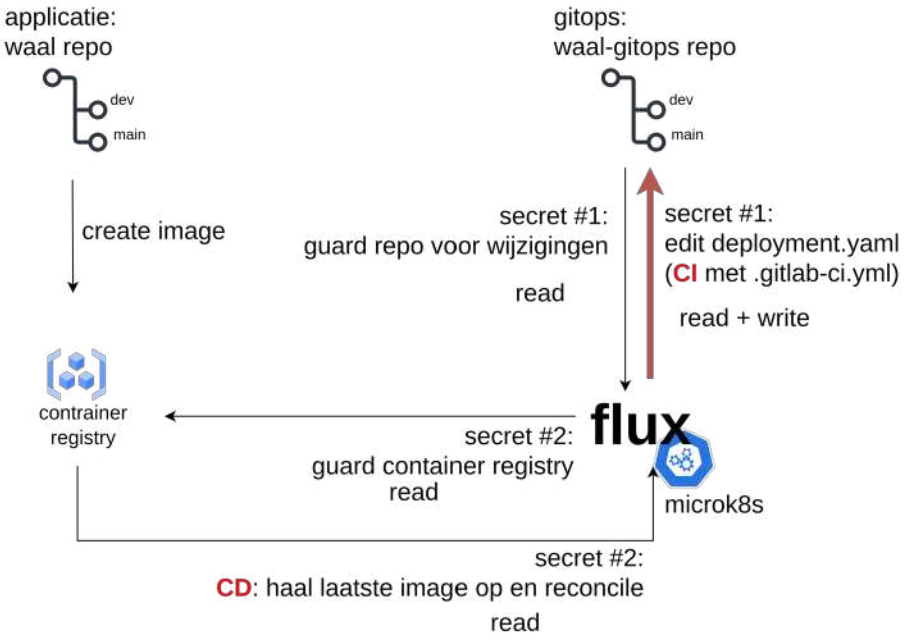
Voorbeeld deployment en service

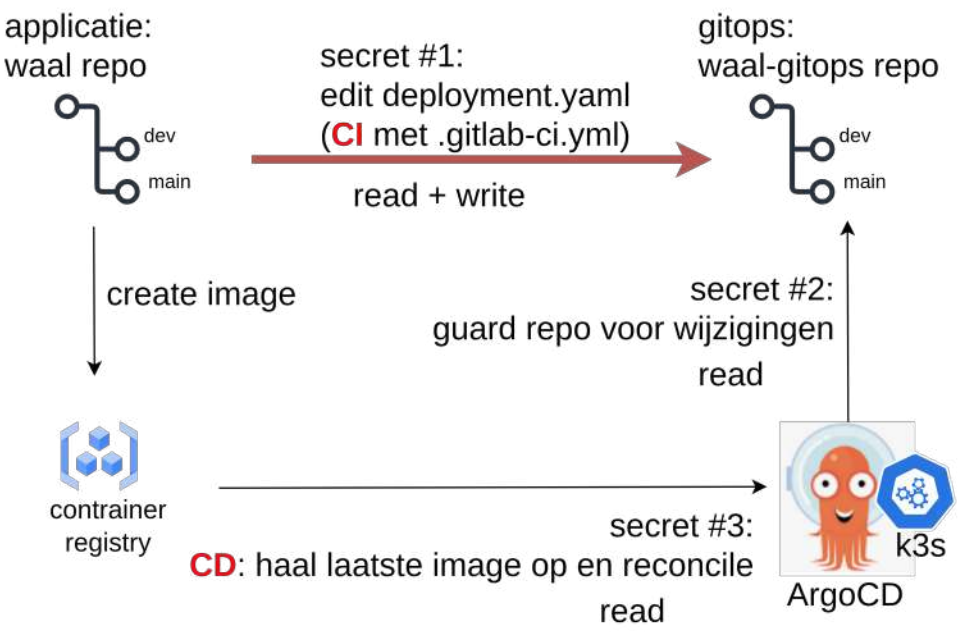
```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: waal-web
  namespace: waal
spec:
  replicas: 1
  selector:
    matchLabels: waal-web
  template:
    metadata:
      labels:
        app: waal-web
    spec:
      imagePullSecrets:
        - name: gitlab-registry
    containers:
      - name: waal-web
        image: registry.gitlab.com/your_name/waal:fd318bc...
```




```
apiVersion: v1
kind: Service
metadata:
  name: waal-web
  namespace: waal
spec:
  selector:
    app: waal-web
  ports:
    - port: 80
      targetPort: 8000
```

Veld	Niveau	Omschrijving
apiVersion	Toplevel	Bepaalt welke Kubernetes API-versie/schema gebruikt wordt
kind	Toplevel	Type Kubernetes resource, zoals Deployment of Service
metadata	Toplevel	Identiteit en metadata van de resource
metadata.namespace	Genest	Namespace waarin de resource bestaat.
metadata.name	Genest	Naam van de resource binnen de namespace.
spec	Toplevel	Beschrijft de gewenste toestand van de resource
spec.replicas	Genest	Gewenst aantal Pod replicas
spec.strategy	Genest	Bepaalt hoe updates/rollouts verlopen
spec.selector	Genest	Selecteert welke Pods bij de resource horen
selector.matchLabels	Genest	Label-filter voor Pod-selectie
spec.template	Genest	Blueprint/template voor aan te maken Pods.
template.spec	Genest	Configuratie van de Pod zelf
template.spec.containers	Genest	Lijst van containers binnen de Pod
containers.name	Genest	Naam van een container
containers.image	Genest	Container image die gestart wordt
containers.volumeMounts	Genest	Mounts volumes in de container


applicatie:
waal repo
dev
main
create image
contrainer
registry
gitops:
waal-gitops repo
dev
main
secret #1:
guard repo voor wijzigingen
read
secret #1:
edit deployment.yaml
(CI met .gitlab-ci.yml)
read + write
flux
microk8s
secret #2:
guard container registry
read
secret #2:
CD: haal laatste image op en reconcile
read


applicatie:
waal repo
dev
main
create image
contrainer
registry
secret #1:
edit deployment.yaml
(CI met .gitlab-ci.yml)
read + write
gitops:
waal-gitops repo
dev
main
secret #2:
guard repo voor wijzigingen
read
secret #3:
CD: haal laatste image op en reconcile
read
k3s
ArgoCD

microk8s / Flux

```
snap install microk8s –classic

curl -s https://fluxcd.io/install.sh | sudo bash

export GITLAB_TOKEN="<your-pat>"
flux bootstrap gitlab \
     --owner=you \
     --repository=your_repo \
     --hostname=gitlab.com \
     --token-auth \
     --path=cluster/development \
     --components-extra=image-reflector-
       controller,image-automation-controller
```

k3s / Argo CD

```
curl -sfL https://get.k3s.io | sh -

kubectl create namespace argocd

kubectl apply -n argocd --server-side --force-conflicts -f \
https://raw.githubusercontent.com/argoproj/argo-cd/stable/
manifests/install.yaml

kubectl apply -f argocd.yaml

kubectl port-forward svc/argocd-server -n argocd 8080:443
```