Infrastructure as Code

Continuous integration & deployment

Roel Van Steenberghe
Sven Knockaert

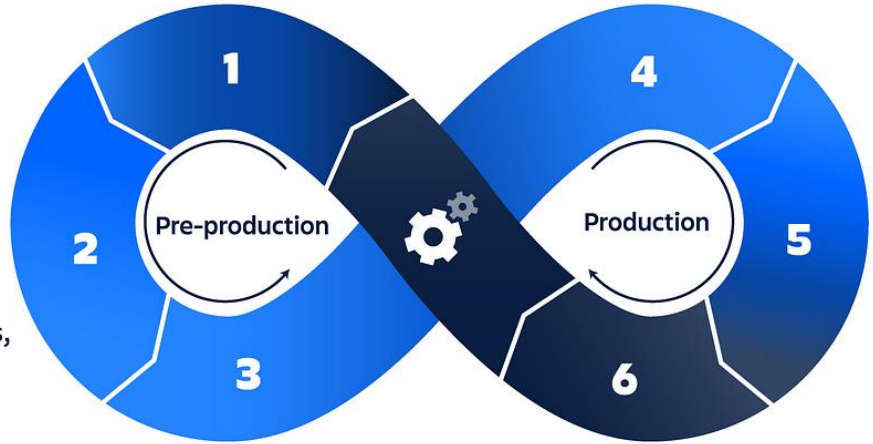
Buzzword bingo!

Devops
Devsecops
Gitops
MLops
Ci/cd
Agile
Lean
Scrum
...

DevSecOps

Plan
(PRE-PRODUCTION)
Threat modeling, change impact analysis

Deploy
(PRODUCTION)
Access and configuration management, chaos engineering, pen testing



Build
(PRE-PRODUCTION)
Pre-commit hooks, software composition analysis, SAST, code review, container security, vulnerability scanning, DAST

Operate
(PRODUCTION)
Log collection, RASP, Patching, WAF

Test
(PRE-PRODUCTION)
DAST

Monitor
(PRODUCTION)
SIEM, vulnerability monitoring, access control

Continuous Integration

Environments

- **Development**
  - Waar je je ontwikkelwerk doet
  - Soms met mocked resources
- **Test**
  - Lijkt op de productie-omgeving, maar vaak veel kleiner
  - Kan test-tools bevatten die de testfase automatiseren
- **Staging**
  - Omgeving die lijkt op productie, met productiedata of een gelijkaardige dataset
  - Idealiter een replica van productie, om bepaalde loads te simuleren, ...
- **Production**
  - Omgeving die zichtbaar is voor eindgebruikers

Continuous Integration: wat

Typische taken:

- BUILD: code compileren, packaging en versioning, database migration, container pushen naar registry
- TEST: unit tests uitvoeren, statische code-analyse, integration testing, security-validatie
- DEPLOY/RELEASE: artefact publiceren als download op een website, code/container vervangen

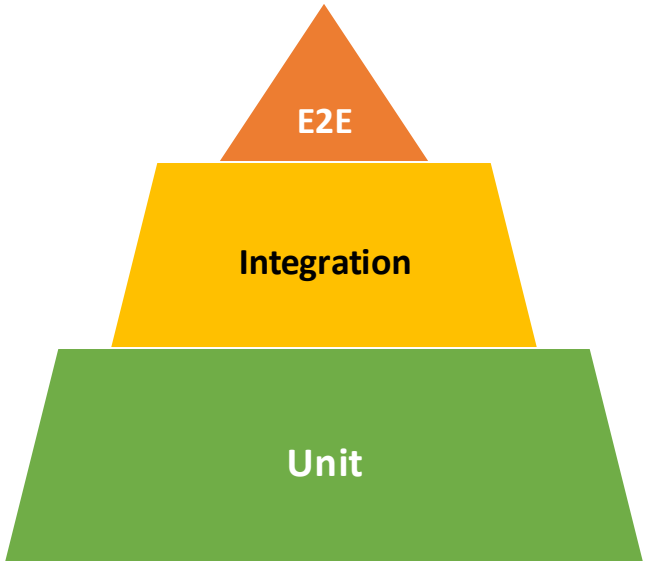
Continuous Integration: waarom

- Reproduceerbaarheid
  - elke versie/build is exact reproduceerbaar
  - geen 'it works on my machine'-discussies meer
- Artefact-generatie
  - executables na compilatie
  - running configs voor netwerkapparatuur
  - …
- Quality assurance
  - kwaliteit verbeteren
  - waarborgen voor kwaliteitscontrole
- Testing
  - codekwaliteit valideren voor deployment

Testing pyramid

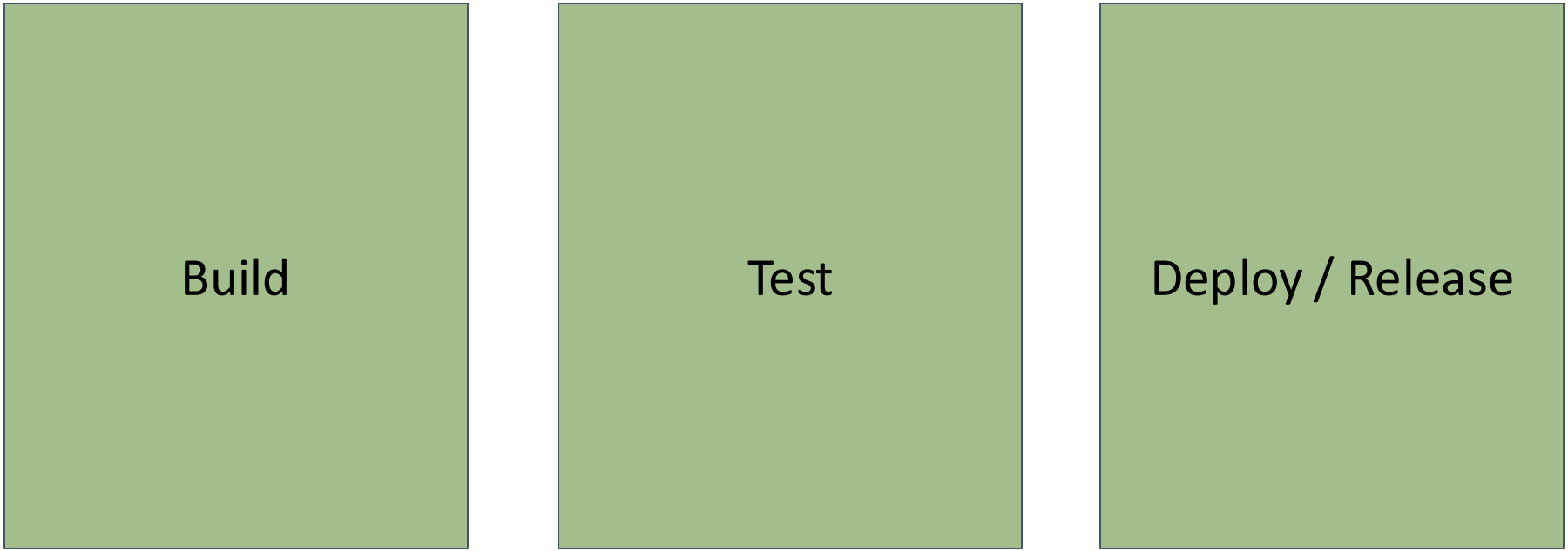
- **End-to-end (E2E)**
  - Volledige user flows door de UI — traag, fragiel, duur. Houd het beperkt
- **Integration / API tests**
  - Modules samen testen: API + DB, services onderling. Middelmatig snel
- **Unit tests**
  - Individuele functies/classes — snel, goedkoop, draaien bij elke commit

Vuistregel: ~70% unit, ~20% integration, ~10% E2E. Snelle feedback = vaak deployen



Oefening

Ontwerp een CI-pipeline voor een PHP-project..



Continuous deployment

Continuous Delivery vs Continuous Deployment

- **Continuous Delivery**
  - Elke commit doorloopt automatisch build + test + staging
  - Release naar productie is een **manuele** stap (één klik op de knop)
  - Geschikt voor: regulated industries, kritieke systemen, change-management
- **Continuous Deployment**
  - Elke groene commit gaat **automatisch** naar productie — geen menselijke tussenstap
  - Vereist sterke test coverage, feature flags en goede monitoring
  - Geschikt voor: SaaS, web apps, snelle iteratie
- **Onthoud**
  - "CD" kan beide betekenen — vraag altijd door
  - Het enige verschil: wie drukt op de knop?

Deployment strategy: basic deployment

Upgrade alles tegelijk

Pro:

- eenvoudig

Contra:

- moeilijk om terug te draaien
- risicovol
- risico op downtime tijdens upgrade

Deployment strategy: blue-green

Testen op de blue-omgeving, eindgebruikers op de green-omgeving

pro:

- eenvoudige rollback

contra:

- kostprijs: alle services draaien parallel
- al het verkeer tegelijk omschakelen kan side effects veroorzaken die niet zichtbaar waren bij een beperkte testgroep

Deployment strategy: rolling deployment

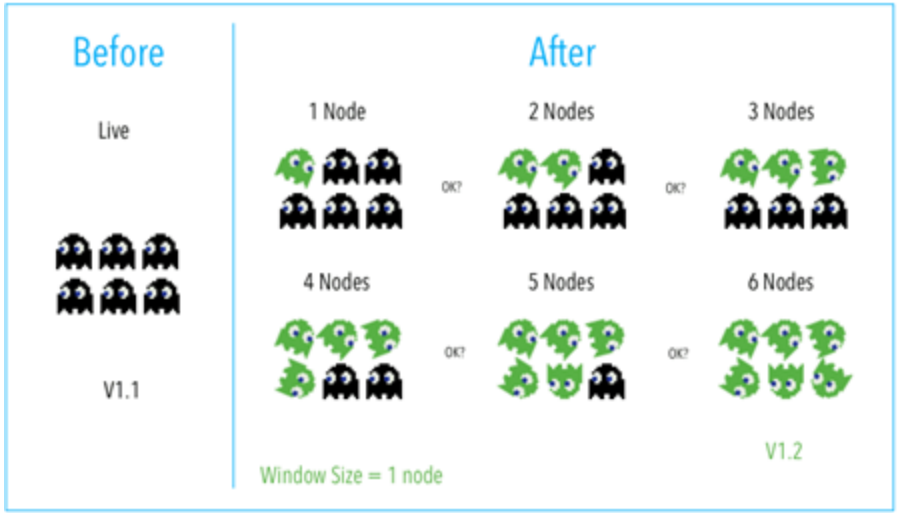
uitrollen in **kleine groepen servers/nodes/containers**

Pro:

- Eenvoudige rollback
- Eenvoudige implementatie

Contra:

- Omgeving moet beide versies ondersteunen
- Kan traag zijn



Oefening: blue-green deployment

- **Doel**
  - Deploy een nieuwe versie van je app zonder downtime
- **Setup**
  - Twee Docker containers (blue + green) achter een Nginx reverse proxy
  - Gitlab pipeline met stages: build → test → deploy-green → smoketest → switch
- **Opdracht**
  - Deploy v1 naar blue; switch traffic; deploy v2 naar green
  - Manuele approval-stage voor de switch
  - Demo rollback: switch terug naar blue

Deployment strategy: canary

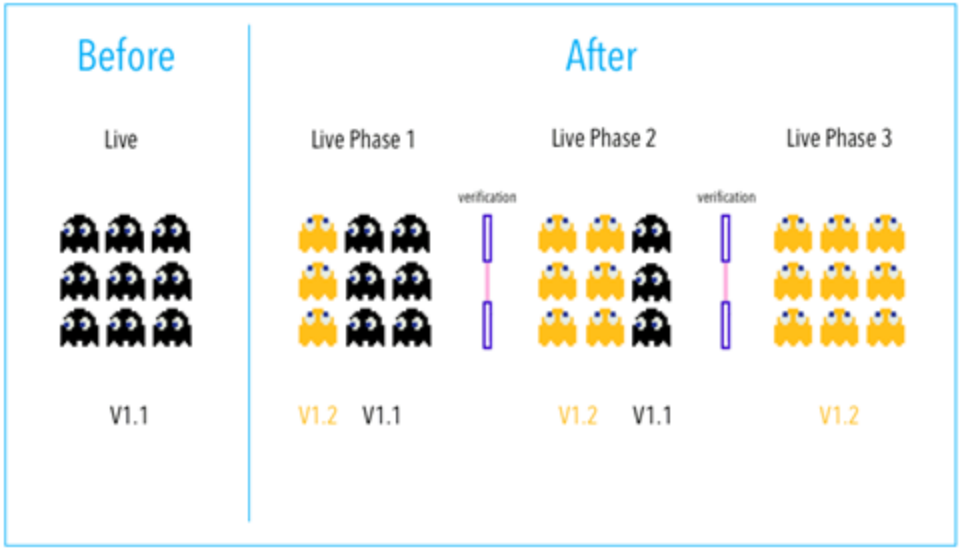
uitrollen voor **kleine groepen eindgebruikers**

Pro:

- Echte tests met echte gebruikers
- Geen extra resources nodig
- Eenvoudig terug te draaien

Contra:

- Moeilijker te monitoren



CI/CD-tools en -processen

CI/CD Tools

- On-premise:
  - Jenkins
  - Gitlab CI/CD
- SaaS:
  - Bamboo
  - TeamCity
  - AWS CodePipeline
  - Cloud Build
  - Gitlab CI/CD
  - Azure DevOps
  - Circle CI
  - Buddy
  - Travis CI
  - Codeship
  - ...

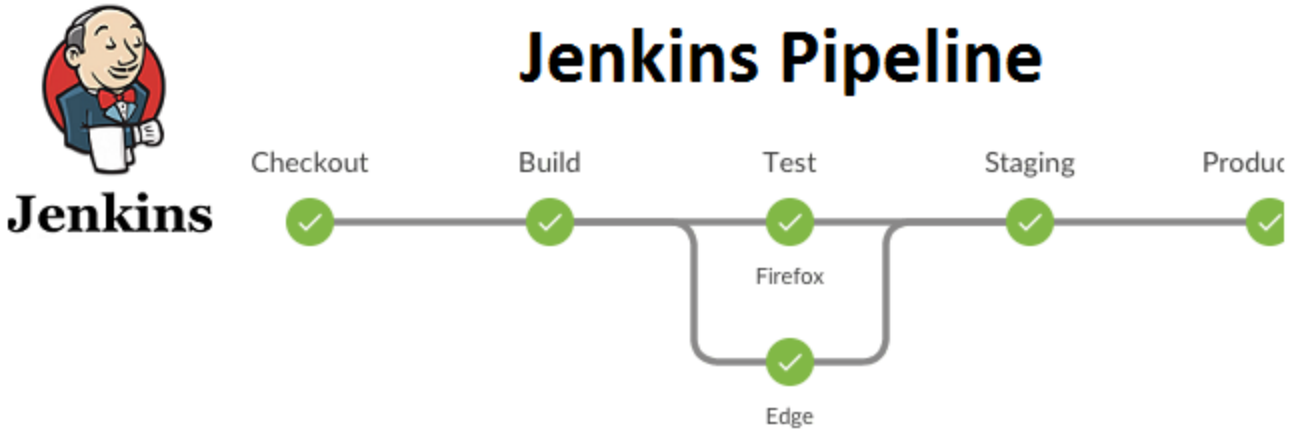
Containers in CI/CD

- **Waarom containers?**
  - "Works on my machine" probleem opgelost — zelfde image overal
  - Reproducible builds: dev = test = staging = productie
- **Pipeline stappen**
  - **Build**: docker build -t app:$CI_COMMIT_SHA .
  - **Test**: tests draaien IN de image, niet ernaast
  - **Push**: push naar registry (GitLab Container Registry, Docker Hub, ACR/ECR)
  - **Deploy**: pull diezelfde image en run op staging/productie (k8s, ECS, Docker swarm)
- **Tag strategie**
  - Gebruik commit SHA of semver (v1.2.3), niet alleen *latest* — anders weet je niet wat draait
- **Image security**
  - Scan op vulnerabilities (Trivy, Snyk, GitLab Container Scanning)
  - Minimal base images (alpine, distroless) — kleiner = sneller = veiliger

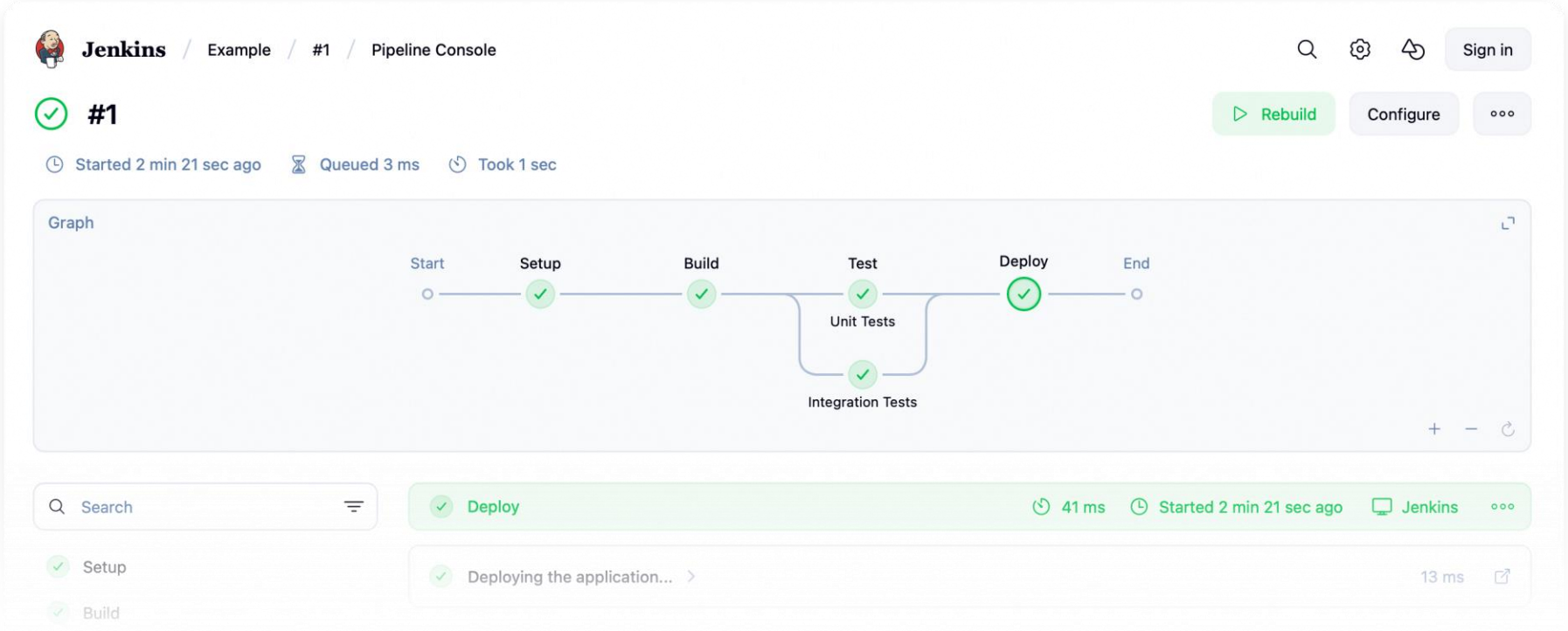
Tools: Jenkins

Jenkins (1 van 2)

- Open source build server en CI/CD-tool
- Kan on-premise gedeployed worden als server of container
- Geschreven in Java
- >1000 plugins, integratie met bijna alle DevOps-tools



Jenkins (2 van 2)



Jenkins: pipeline as code

- **Jenkinsfile**
  - Tekstbestand dat de pipeline beschrijft
  - Wordt mee gecommit in je repository (versioning, code review)
  - Groovy-syntax
- **Voordelen t.o.v. GUI-configuratie**
  - Reproduceerbaar & herbruikbaar
  - Pull request review op pipeline-wijzigingen
  - Rollback naar oudere pipeline-versie via git

Jenkins: declarative vs scripted

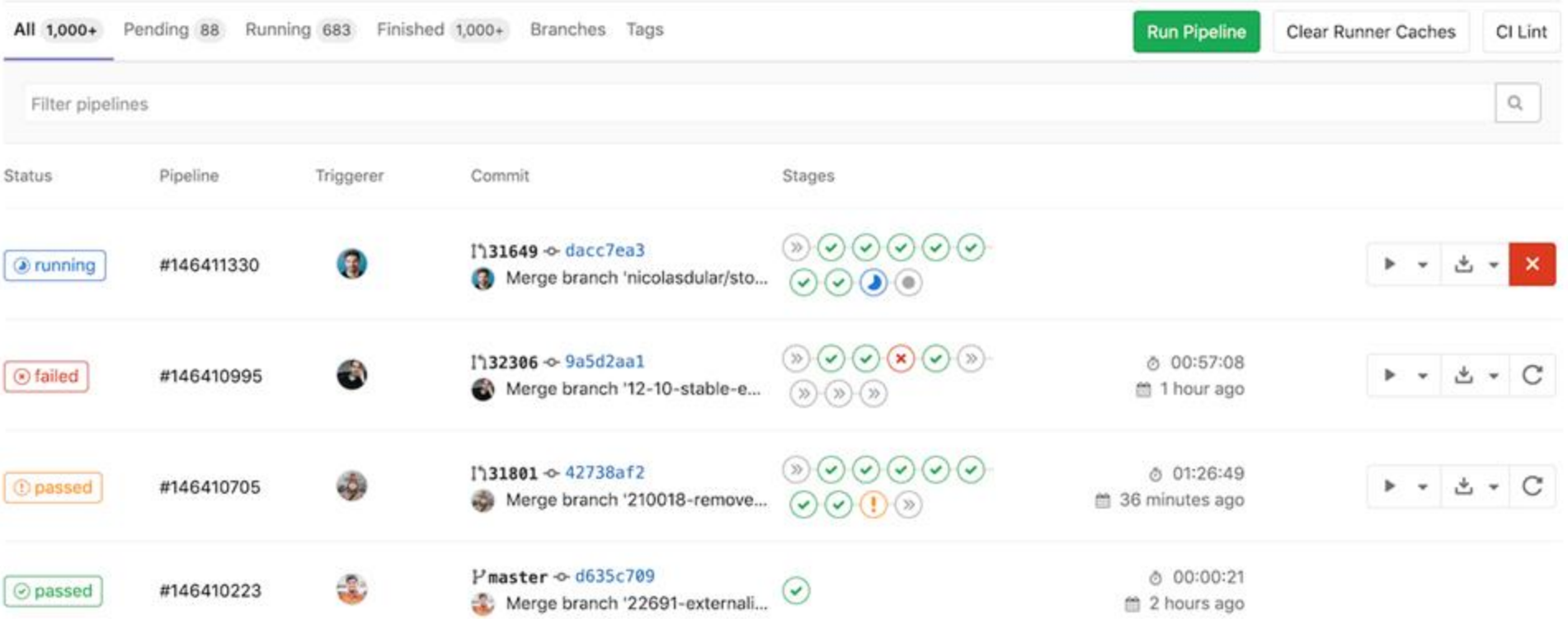
- **Declarative pipeline**
  - Vaste structuur: pipeline { agent { } stages { stage('Build') { steps { } } } }
  - Eenvoudiger te lezen — aanbevolen voor de meeste projecten
  - Goede validatie & foutmeldingen
- **Scripted pipeline**
  - Volledige Groovy-code: node { stage('X') { ... } }
  - Meer flexibiliteit (loops, complexe logica)
  - Steile leercurve; minder leesbaar
- **Vuistregel: start declarative; switch alleen bij echte noodzaak**

Tools: Gitlab CI/CD

Gitlab CI/CD

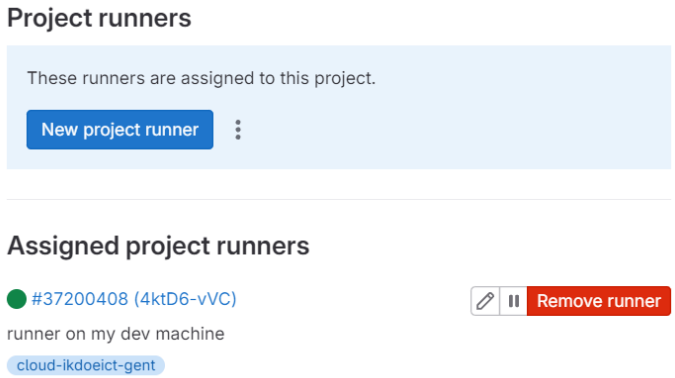
- Build/test/deploy van code met runners
  - Shell
  - Docker
  - Kubernetes
  - …
- Definieer de workflow in een **.gitlab-ci.yml**-bestand
- artefacten opslaan
- (geheime) variabelen gebruiken

Gitlab CI (1 van 2)



Gitlab CI: Runners

- Dit zijn de VM's of containers die je pipelines uitvoeren
- Je kan die lokaal of in de cloud deployen
  - Runners betaal je per minuut
  - Elk type runner heeft zijn eigen kostprijs: https://docs.gitlab.com/runner/executors/
  - Kies verstandig en wees zuinig!
- Sharing levels:
  - **Shared**: gedeeld over alle projecten
  - **Group**: te gebruiken door een bepaalde groep
  - **Specific**: voor een specifiek project
- Keuze van executor (https://docs.gitlab.com/runner/executors/)
  - Op een server (side note: interessant om je pipeline lokaal te testen zonder pushen)
  - In Docker-containers (mogelijkheden: Docker, Docker in Docker, Kubernetes, ..)
  - Op remote SSH-servers

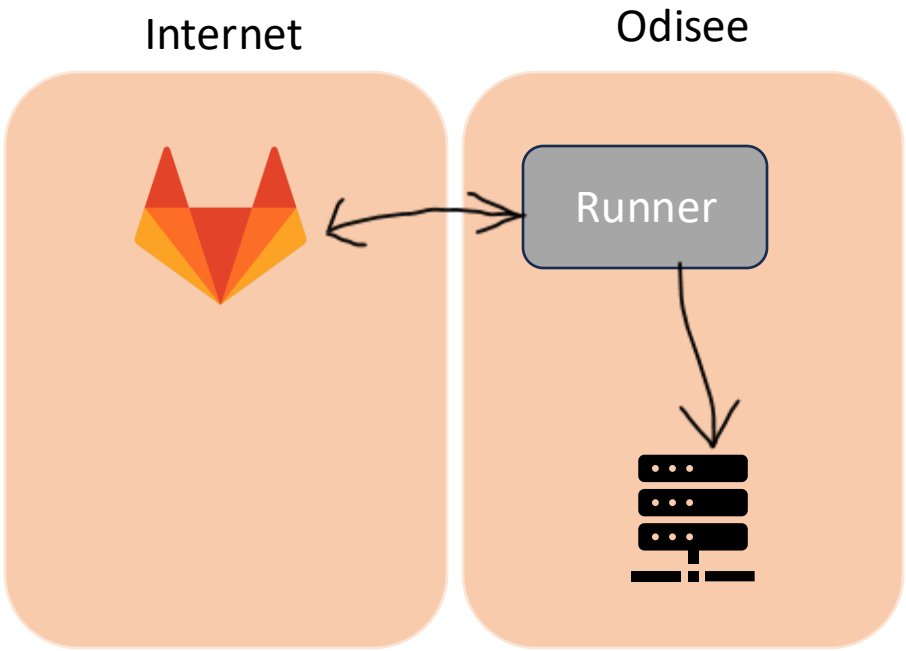


Gitlab CI: Runners

Settings -> CI/CD -> runners

Gitlab CI/CD met Odisee

- Runner op je eigen cloud
- Runner heeft een persistente connectie naar Gitlab.com, op die manier raak je voorbij de Odisee-firewall
- connectie naar je eigen server
  - SSH
  - Ansible op runner naar je eigen servers over SSH
  - sFTP
  - …



Gitlab pipelines: jobs & stages

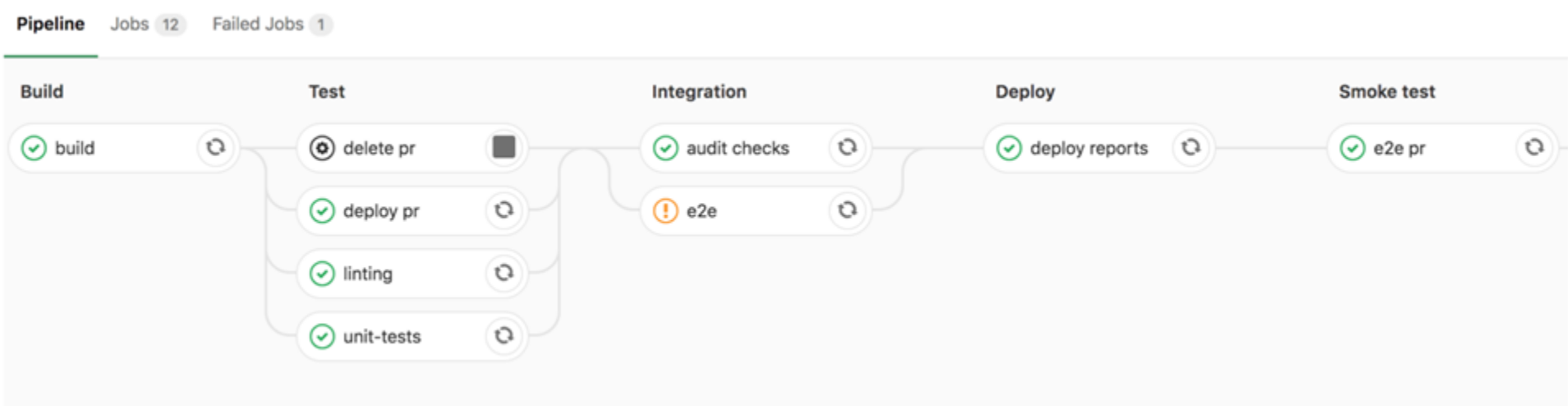
Een job is de kleinste eenheid die uitgevoerd wordt in GitLab CI/CD. Vaak wordt dit een “build step” genoemd. Het kan een build- of compile-taak zijn; unit tests uitvoeren; code quality checks zoals linting of code coverage thresholds; of een deployment-taak.

Een enkele job kan meerdere commando's (scripts) bevatten.
voorbeeld:

```
nameofthejob:
   script:
      - npm install
      - npm run test
```

Gitlab pipelines strategie 1: jobs & stages

Elke job hoort bij één stage. Een stage kan nul, één of meerdere jobs bevatten. Alle jobs binnen dezelfde stage worden parallel uitgevoerd. De volgende stage start pas als alle jobs van de vorige stage succesvol zijn afgerond (*) - of als ze gemarkeerd zijn als "allowed to fail".
(*) tenzij je stageless pipelines gebruikt



Gitlab voorbeeld: basic pipeline

```yaml
stages:
  - build
  - test

build_job:
  stage: build
  script:
    - echo "Building the application..."
    - npm install
    - npm run build

test_job:
  stage: test
  script:
    - echo "Running tests..."
    - npm test
```

Gitlab pipelines: variables

- Purpose: Define reusable values within your pipeline.
- Types: Project-level, group-level, and job-level.
- Benefit: Reduces duplication, makes configurations more flexible.

```
variables:
  APP_VERSION: "1.0.0"
  BUILD_DIR: "dist"

build_job:
  stage: build
  script:
    - echo "Building version $APP_VERSION into $BUILD_DIR"
    - mkdir $BUILD_DIR
```

Gitlab CI/CD: secrets management

Probleem: nooit passwords, API keys of tokens hardcoden in .gitlab-ci.yml of in je code!

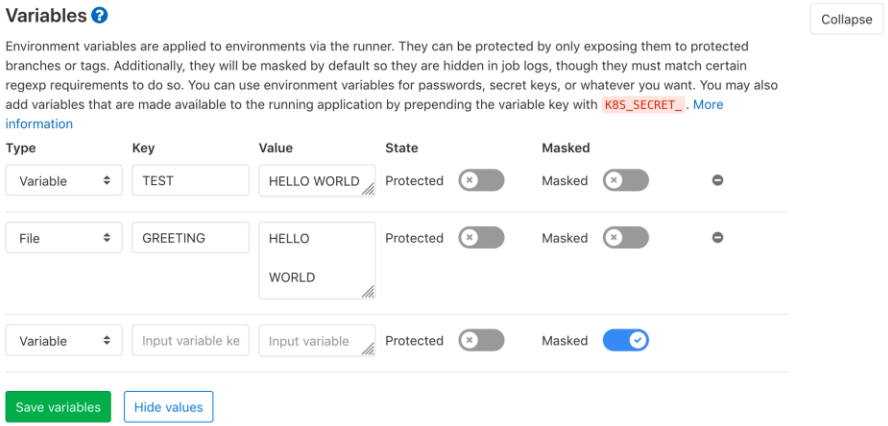
- **Oplossing 1: CI/CD variables (Settings > CI/CD > Variables)**
  - **Masked**: wordt verborgen in job logs (***)
  - **Protected**: enkel beschikbaar op protected branches/tags (vb. main)
  - **File type**: voor SSH keys, kubeconfig, certificaten
- **Oplossing 2: externe vault (productie)**
  - HashiCorp Vault, AWS Secrets Manager, Azure Key Vault
  - Pipeline haalt secret op tijdens runtime — audit trail, rotatie
- **Best practices**
  - Roteer secrets na een leak — git history vergeet niets
  - Gebruik secret scanning (GitLab Ultimate / gitleaks) om commits te checken
  - Principle of least privilege: tokens met minimale scope & korte levensduur

Gitlab CI/CD: variables

- Doel: definieer herbruikbare waarden in je pipeline.
- Types: project-level, group-level en job-level.
- Voordeel: vermindert duplicatie, maakt configuraties flexibeler.

```
variables:
  APP_VERSION: "1.0.0"
  BUILD_DIR: "dist"

build_job:
  stage: build
  script:
    - echo "Building version $APP_VERSION into $BUILD_DIR"
    - mkdir $BUILD_DIR
```



Gitlab pipelines: artifacts

Doel: specificeert bestanden of mappen die als output van een job moeten worden meegegeven aan volgende stages/jobs.

Voordeel: deel build-outputs (vb. gecompileerde binaries, testrapporten) tussen stages.

```
build_job:
  stage: build
  script:
    - npm run build
  artifacts:
    paths:
      - build/
    expire_in: 1 week
```

Gitlab pipelines: cache

- Doel: bestanden bewaren tussen pipeline runs voor snellere uitvoering.
- Voordeel: versnelt builds door hergebruik van dependencies (vb. node_modules, vendor/).

```
build_job:
  stage: build
  script:
    - npm install
    - npm run build
  cache:
    paths:
      - node_modules/
```

Gitlab pipelines: Directed Acyclic Graphs (DAG)

- Probleem: standaard worden jobs in verschillende stages sequentieel uitgevoerd. Jobs binnen dezelfde stage draaien parallel. Wat als je een job in een latere stage afhankelijk wil maken van één job uit een vroegere stage, zonder te wachten op de andere jobs in die stage? Of als jobs binnen dezelfde stage onderlinge afhankelijkheden hebben?
- Oplossing: het sleutelwoord needs
- Doel: definieert expliciet afhankelijkheden tussen jobs, en overschrijft zo de standaard stage-gebaseerde volgorde. Resultaat: creëert een Directed Acyclic Graph (DAG) voor je pipeline, wat efficiëntere parallellisatie toelaat.
- Voordeel: verkort de totale pipeline-uitvoertijd doordat onafhankelijke jobs gelijktijdig kunnen draaien, ook over stage-grenzen heen.

Gitlab pipelines: Directed Acyclic Graphs (DAG)

```yaml
stages:
  - build
  - test
  - deploy

# Job in 'build' stage
build_frontend:
  stage: build
  script:
    - echo "Building frontend..."
    - sleep 5 # Simulate work

# Job in 'build' stage
build_backend:
  stage: build
  script:
    - echo "Building backend..."
    - sleep 3 # Simulate work

# This test job needs 'build_frontend' to complete before starting
test_frontend:
  stage: test
  needs: ["build_frontend"] # Explicit dependency
  script:
    - echo "Testing frontend after build_frontend is done."

# This deploy job needs both backend and frontend builds to complete
deploy_to_dev:
  stage: deploy
  needs: ["build_frontend", "build_backend"] # Multiple dependencies
  script:
    - echo "Deploying to dev after both builds."
```

Gitlab pipelines: conditional execution

- Legacy oplossing: 'only' en 'except' keywords
- Betere oplossing: rules

Clause	Description
`if`	Add or exclude jobs from a pipeline by evaluating an `if` statement. Similar to `only:variables`.
`changes`	Add or exclude jobs from a pipeline based on what files are changed. Same as `only:changes`.
`exists`	Add or exclude jobs from a pipeline based on the presence of specific files.

```
job_name:
  rules:
    - if: $CI_COMMIT_TITLE =~ /-draft$/
      when: never
    - if: $CI_PIPELINE_SOURCE == "merge_request_event"
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
```

Gitlab CI: conditional execution: examples

```yaml
deploy_production:
  stage: deploy
  script:
    - deploy-to-production.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual # Requires manual trigger for 'main' branch
      allow_failure: false
```

Gitlab CI: conditional execution: examples

```yaml
heavy_integration_tests:
  stage: test
  script:
    - run-heavy-tests.sh
  rules:
    - if: '$CI_COMMIT_MESSAGE =~ /\[skip-integration-tests\]/'
      when: never # Don't run if commit message contains specific tag
    - when: on_success # Otherwise, run normally
```

Gitlab CI: conditional execution: examples

```yaml
unit_tests:
  stage: test
  script:
    - npm test
  rules:
    - if: '$CI_COMMIT_BRANCH =~ /feature\/.*/ || $CI_COMMIT_BRANCH == "main"' # Run on
feature branches and main

deploy_staging:
    stage: deploy
    script:
      - deploy-to-staging.sh
    rules:
      - if: '$CI_COMMIT_BRANCH == "main"' # Only deploy staging from main
```

GitOps

GitOps

- **Wat is GitOps?**
  - Operationele praktijk waarbij Git de single source of truth is voor infrastructuur en applicaties
  - Term bedacht door Weaveworks (2017), gegroeid met Kubernetes
- **De vier principes (OpenGitOps)**
  - **Declaratief** – beschrijf gewenste staat, niet de stappen
  - **Versiebeheer & immutable** – alles staat in Git, volledige history
  - **Automatisch toegepast** – agent pullt & synct, geen handmatige kubectl apply
  - **Continu verzoend** – drift wordt gedetecteerd en automatisch hersteld
- **Push vs Pull model**
  - Klassieke CI/CD = push: pipeline deployt naar cluster (credentials in CI)
  - GitOps = pull: agent in cluster trekt de gewenste staat zelf binnen

GitOps in de praktijk: ArgoCD & Flux

- **De twee grote tools**
  - **ArgoCD** – CNCF graduated, draait in Kubernetes, sterke web-UI, Application CRD
  - **Flux** – CNCF graduated, CLI-first, modulair (source/kustomize/helm controllers)
- **Typische workflow**
  - Developer pusht code → CI bouwt image → image tag wordt in config-repo geüpdatet
  - ArgoCD/Flux detecteert wijziging en synct cluster naar nieuwe staat
  - App- en config-repo gescheiden houden (best practice)
- **Voordelen**
  - Audit trail via Git, rollback = git revert
  - Geen cluster-credentials in CI, minder aanvalsoppervlak
- **Beperkingen**
  - Voornamelijk Kubernetes-georiënteerd; secrets nog steeds een aandachtspunt (SealedSecrets, External Secrets)

Monitoring & observability

- **Waarom?**
  - Een succesvolle deploy ≠ werkende app — meet wat productie doet
- **De drie pijlers**
  - **Metrics**: getallen over tijd (CPU, latency, errors) — Prometheus, Grafana
  - **Logs**: tijdsgestempelde events — ELK / Loki
  - **Traces**: verloop van een request door services — Jaeger, OpenTelemetry
- **Golden signals (Google SRE)**
  - Latency · Traffic · Errors · Saturation

Alerting & rollback

- **Alerting**
  - Alleen alerten op symptomen die de gebruiker raken
  - Routeer naar Slack / e-mail / PagerDuty
  - Vermijd alert fatigue: drempels & tijdvensters
- **Rollback strategie**
  - Elke deploy moet omkeerbaar zijn
  - Blue-green & canary maken rollback snel & veilig
  - Automatische rollback: pipeline triggert bij overschrijden van error-budget
- **Sluiten van de loop**
  - Monitoring data → verbeter testen → volgende pipeline run