ITI8801 · Cloud Architectures and DevOps

CI/CD: from a push to a running deployment

Siim Rebane · TalTech · Autumn 2026

The problem

One changed line in the service reaches the cluster through these steps, each done by a person at a terminal:

Step	Command
run the tests	`pytest`, with a PostgreSQL running for them
build the image	`docker build -t <repository>:<tag> .`
check it for known vulnerabilities	`trivy image <repository>:<tag>`
push it to the registry	`docker push <repository>:<tag>`
change the tag in the manifest	an editor
apply the manifest	`kubectl apply -f deployment.yaml`
check that the new pods are ready	`kubectl rollout status deployment/api`

Seven steps for every change, on the machine of whoever made the change.

What goes wrong when it is done by hand

What happens	Result
a step is skipped: the scan, the test	an image nobody checked runs in the cluster
the image is built on a laptop	its content depends on that laptop: its Docker version, its cached base image, files that are not in git
the tag is typed by hand	two different images get the same tag, or the manifest names a tag that was never pushed
the manifest is applied from a working copy	the cluster runs something that is in no commit
nobody writes down what was deployed	which commit runs now, and since when, is not known
only one person has the cluster credential on their machine	nobody else can deploy

The steps as a file in the repository

The steps are written into a file that lives in the repository next to the code. A server reads the file on every push and runs the steps, in the same order, in a clean environment, and keeps the output of each.

By hand	From the file
the steps are in someone's memory	the steps are in git, with history, changed by commits
run when someone remembers	run on every push
run on a laptop	run in a fresh container
output is in a closed terminal	output is stored per run, per step
a failed check is noticed or not	a failed step stops the steps after it

Every CI system is this idea: a file in the repository and a server that runs it. The file is `.gitlab-ci.yml` in GitLab, a workflow under `.github/workflows/` in GitHub Actions, a `Jenkinsfile` in Jenkins. This lecture uses GitLab; the names differ, the parts do not.

Pipeline, stage, job

- A **job** is one set of commands run in one environment, with one result: passed or failed. Example: build the image.
- A **stage** is a group of jobs. The stages run one after another; the jobs of one stage run at the same time.
- A **pipeline** is one run of all the stages for one commit.

The pipeline of this lecture, the example of the whole lecture, has four stages with one job each:

```
test  ⟶  build  ⟶  scan  ⟶  deploy
```

When a job fails, the stages after it do not run.

Three terms: CI, continuous delivery, continuous deployment

Term	Means
continuous integration (CI)	every push is built and tested automatically, so a change that breaks the build or the tests is seen minutes after it was pushed
continuous delivery	every change that passes is packaged and ready to be released; the release itself is started by a person
continuous deployment	every change that passes is released automatically, with no person in between

CI/CD names the whole: the pipeline from a push to a running deployment. What starts a deploy differs between teams:

Trigger	Deploys	Common where
a merge to `main`	the first environment; in trunk-based teams production as well, often behind feature flags	continuous deployment, web products with many small changes
a Git tag such as `v1.4.0`	a release: the image that already ran in the environment before	versioned products, anything with release notes
a manual job, a button	production, by a named person, sometimes after an approval	a release manager, change windows, regulated environments
a change in a watched repository	whatever the repository says, by a program inside the cluster that pulls it	GitOps: Argo CD, Flux

In this lecture's pipeline the first environment is deployed by the merge and the second by a tag.

Where the pipeline runs: GitLab CI

GitLab CI is the part of GitLab that runs pipelines. Three things take part:

Part	What it is
the file `.gitlab-ci.yml`	at the root of the repository; describes the stages and the jobs
the GitLab server	reads the file on every push, creates the pipeline and its jobs, stores their output and results
a **runner**	a program on another machine that asks GitLab for a job, runs its commands and sends the output back

The GitLab server runs no job itself. A project with no runner has pipelines whose jobs wait forever.

The course uses TalTech's GitLab, `gitlab.cs.taltech.ee`.

The path of one push

1. `git push` sends commits to GitLab.
2. GitLab reads `.gitlab-ci.yml` **as it is in the pushed commit** and creates a pipeline: one job for each job in the file whose conditions hold.
3. The jobs of the first stage get the status `pending`: they wait for a runner.
4. A runner takes a job, starts a container from the image the job names, clones the repository at that commit into it, and runs the job's commands. The job is `running`.
5. The output of the commands goes to GitLab as the **job log**.
6. The last result decides: `passed` or `failed`.
7. When all jobs of a stage passed, the jobs of the next stage become `pending`.

Build → Pipelines in the project shows every pipeline; a click on a job shows its log.

The runner at TalTech

Fact	Detail
its name	`Generic runner on Taltech cloud with Docker executor`
whose it is	an **instance runner**: it serves every project of the GitLab server that allows it (older texts call it a shared runner)
a new project	has instance runners turned off: Settings → CI/CD → Runners → turn on instance runners for the project
its **executor**	Docker: every job runs in a container started from the image the job names
how many jobs at once	few: a job of yours waits as `pending` while other projects' jobs run

A job log starts with the runner that took it:

```
Running with gitlab-runner 19.3.1 (a16f5092)
  on Generic runner on Taltech cloud with Docker executor UXUZxwupi
Using Docker executor with image python:3.13-slim ...
```

A job is a fresh container plus your commands

```
hello:
  image: alpine:3.22
  script:
    - echo "commit $CI_COMMIT_SHORT_SHA on $CI_COMMIT_BRANCH"
    - ls
```

Part	Means
`hello:`	the job's name, your choice
`image:`	the container image the job runs in; pulled by the runner
`script:`	the commands, run in order by a shell inside that container

The container starts empty except for the image's content and a clone of the repository at the pipeline's commit, which is the working directory. Nothing a job installs or writes remains for the next job: each job gets a new container.

Exit codes: how the runner knows a command failed

Every command ends with an **exit code**, a number: 0 means success, anything else means failure.

```
$ ls /etc/hostname;  echo "exit code $?"
/etc/hostname
exit code 0
$ ls /no/such/file;  echo "exit code $?"
ls: /no/such/file: No such file or directory
exit code 1
```

The runner stops a job at the first command that ends with a code other than 0 and marks the job `failed`. A job is `passed` when every command ended with 0.

A check in a pipeline is therefore a command that ends with a code other than 0 when the check does not hold.

The keywords of a job

Keyword	Means
`stage:`	which stage the job belongs to
`image:`	the container image the job runs in
`script:`	the commands
`after_script:`	commands run after `script`, also when `script` failed
`variables:`	environment variables for this job
`rules:`	the conditions under which the job is part of a pipeline
`services:`	further containers started next to the job's container
`cache:`	directories the runner keeps from one job to later jobs
`environment:`	the name of the place this job deploys to
`extends:`	another job whose keywords this job starts from

At the top level of the file, `stages:` lists the stages in their order, and `variables:` sets variables for every job.

Variables GitLab gives every job

GitLab sets environment variables in every job that describe the pipeline. The ones this lecture uses, with the values of one real pipeline:

Variable	Value	Means
`CI_COMMIT_SHA`	`f4e65fbb…` (40 characters)	the commit the pipeline runs for
`CI_COMMIT_SHORT_SHA`	`f4e65fbb`	its first 8 characters
`CI_COMMIT_BRANCH`	`main`	the branch, in a pipeline started by a push to a branch; not set in a tag pipeline
`CI_COMMIT_TAG`	`v1.2`	the tag, in a pipeline started by a pushed tag; not set otherwise
`CI_JOB_ID`	`449131`	a number unique to this job on the whole GitLab server
`CI_PIPELINE_ID`	`165306`	a number unique to this pipeline

CI/CD variables

Settings → CI/CD → Variables in the project. A variable has a key, a value, a type and these settings:

Setting	Effect
Visibility: Masked	GitLab replaces the value with `[MASKED]` wherever it would appear in a job log. Works for a value of one line, at least 8 characters, without spaces
Protect variable	the variable exists only in pipelines that run for a protected branch or a protected tag
Expand variable reference	a `$` in the value is read as a reference to another variable; switched off for a value that contains a `$` of its own

Who can see and change the variables: project members with the role Maintainer or Owner. A member with the role Developer cannot open the page, but can run a job that prints a variable.

The variables of this lecture's project:

Key	Type	Masked	Protected
`DOCKER_USERNAME`, `DOCKER_IMAGE`	Variable	no	no
`DOCKER_PASSWORD`	Variable	yes	no
`KUBECONFIG`	File	no: it has many lines	**yes**

`rules`: which jobs a pipeline has

`rules:` is a list of conditions. The job is part of a pipeline when one of them holds; a job with `rules:` of which none holds is not created.

Condition	Holds in
`if: $CI_COMMIT_BRANCH`	a pipeline for a push to any branch
`if: $CI_COMMIT_BRANCH == "main"`	a pipeline for a push to `main`
`if: $CI_COMMIT_TAG`	a pipeline for a pushed tag

The jobs of this lecture's file, per kind of pipeline:

Job	push to a branch	push to `main`	a tag
`test`	yes	yes	
`build`	yes	yes	
`scan`	yes	yes	yes
`deploy-dev`		yes	
`deploy-staging`			yes

Cache and artifacts: what a job leaves behind

Each job starts in a fresh container. Two keywords carry files out of it:

	`cache:`	`artifacts:`
holds	things that can be downloaded or computed again: dependency directories, a scanner's database	things the job produced: a built package, a test report, a coverage file
goes to	the runner's storage; put back before a later run of the job, when it is still there	GitLab: downloadable from the job page, given to the jobs of later stages
may be missing	yes, at any time; a job must work without it	no: a later job that needs it gets it
typical	`cache: paths: [.trivy/]`, `node_modules/`, `.m2/`	`artifacts: paths: [build/libs/]`, `artifacts: reports: junit: report.xml`

A test report as an artifact (`reports: junit:`) makes GitLab show the failed test by name on the pipeline page and in the merge request, without opening the log. `pytest --junitxml=report.xml`, Gradle's `build/test-results/test/*.xml`.

`services` : a container next to the job

`services:` lists images the runner starts next to the job's container before `script` runs, and removes after the job. The job reaches each one by a host name: the image's name without registry and tag, `postgres` for `postgres:18-alpine`, or a name given with `alias:`.

```yaml
test:
  image: python:3.13-slim
  services: [postgres:18-alpine]
  variables:
    POSTGRES_USER: app
    POSTGRES_PASSWORD: only-for-this-job
    POSTGRES_DB: app
```

Fact	Consequence
the job's `variables:` are given to the service container as well	the `POSTGRES_*` variables above configure the database at its start
the runner waits until the service's port answers, 30 s at most	the first `script` line can connect
a service lives for one job	an empty database for every run; nothing of one pipeline reaches the next
a service is any image	a message queue, a cache, a second service of your own, a Docker daemon (`docker:dind`)

The example of this lecture

One service with a REST API and a PostgreSQL database: `GET /health`, and create, list, read and delete under `/api/boardgames`. Python, about 130 lines.

```
.
├── .gitlab-ci.yml         the pipeline
├── Dockerfile             builds the image
├── main.py                the service
├── requirements.txt       its libraries
├── requirements-dev.txt   pytest and the test client, needed only by the tests
├── pytest.ini             tells pytest where the service's code is
├── tests/
│   └── test_api.py        tests of the API, against a real PostgreSQL
└── k8s/
    ├── api.yaml           Deployment, Service, Ingress
    └── db.yaml            PostgreSQL: Service, StatefulSet
```

Everything the pipeline needs is in the repository, except the credentials. The lecture's concepts are shown on this project; the outputs are from its real pipelines.

The pipeline of this lecture

Stage	Job	Does	Fails when
test	`test`	runs the project's tests, with a PostgreSQL for them	a test fails
build	`build`	builds the image from the commit and pushes it to the registry	the Dockerfile does not build, the registry refuses the push
scan	`scan`	looks for known vulnerabilities in that image	one of severity HIGH or CRITICAL with a fix available is found
deploy	`deploy-dev`	makes the cluster run that image	the new pods do not become ready

```
stages: [test, build, scan, deploy]
```

The tests run first: an image is built only from a commit that passed them. The same image is then scanned and deployed.

What a test in a pipeline proves

A pipeline runs tests so that a change which breaks the service is stopped before it is deployed. Three levels, each proving something the others cannot:

Level	Runs against	Proves	Does not see
unit	one function or class, in the test process, no network	the logic: a wrong result for an input	a wrong query, a wrong wiring, a wrong container
integration	the code together with a real database or queue, in the test process	the code and its dependencies work together: the queries, the mapping, the HTTP layer	the image: a missing library, a wrong port, a wrong start command
smoke, end to end	the running artifact, over the network, as a user would	the deployed thing answers	why it fails; it says only that it does

Unit and integration tests run in the pipeline on every push, before the image is built. A smoke test runs after a deploy, against the environment; the readiness probe is its smallest form.

The test stage of this lecture is integration tests: the service's code with a real PostgreSQL, through its HTTP layer, in one process.

Where the test's database comes from

Arrangement	How	Property
a **service container** of the job	`services: [postgres:18-alpine]`; the tests connect to the host `postgres`	started by the runner, no code for it; the tests do not know they are in a pipeline
Testcontainers	the test code starts the container through the Docker socket; libraries for Java, Go, Python, .NET, Node	the same test runs on a laptop and in the job; the job's container needs the socket
an **in-memory** stand-in	H2, SQLite	no container, but a different database: SQL that passes here fails on PostgreSQL
a **shared** test server	one database for every pipeline	state leaks between runs; two pipelines at once break each other

The first two are the ones to use. Both give the tests a real PostgreSQL that is empty at the start and gone at the end.

The example: the test job

```yaml
test:
  stage: test
  image: python:3.13-slim
  services: [postgres:18-alpine]
  variables:
    POSTGRES_USER: app
    POSTGRES_PASSWORD: only-for-this-job
    POSTGRES_DB: app
    DATABASE_URL: postgresql://app:only-for-this-job@postgres:5432/app
  script:
    - pip install -q -r requirements.txt -r requirements-dev.txt
    - pytest
  rules:
    - if: $CI_COMMIT_BRANCH
```

The project's own tests, run by the project's own test tool, in the language's image. The job's only addition is a database for them.

The test job, line by line

Line	Means
`image: python:3.13-slim`	the language's image: `python`, `pip` and nothing else
`services: [postgres:18-alpine]`	a second container, started by the runner next to the job's container before `script` runs, and removed after the job. The job reaches it under the host name `postgres`
`POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB`	read by the PostgreSQL container at its start: the user, the password and the database it creates. A password for a database that lives for seconds and holds nothing; it is in the file on purpose
`DATABASE_URL`	the variable the service reads: the address of the service container above
`pip install -q -r requirements.txt -r requirements-dev.txt`	the service's libraries and the test libraries, into this job's container
`pytest`	runs every test it finds; ends with 0 when all passed, 1 otherwise
`rules: - if: $CI_COMMIT_BRANCH`	the job exists in pipelines started by a push to a branch

A test tool in another language takes the same shape: `node:22` and `npm test`, `eclipse-temurin:21` and `./gradlew test`, `golang:1.25` and `go test ./...`.

What the tests do

`tests/test_api.py` starts the service inside the test process and sends it requests. The database is the one the job started: the service reads `DATABASE_URL` as it does in the cluster.

```python
@pytest.fixture(scope="module")
def client():
    with TestClient(app) as c:   # starts the service: opens the pool, creates the table
        yield c

def test_health(client):
    assert client.get("/health").json() == {"status": "ok"}

def test_players_is_required(client):
    assert client.post("/api/boardgames", json={"name": "Catan"}).status_code == 400
```

Three tests: `/health`, create → read → delete of one boardgame, and a body without `players` that must be refused. Each `assert` that does not hold fails its test, and one failed test makes `pytest` end with 1.

The test log

```
Starting service postgres:18-alpine...
Waiting for services to be up and running (timeout 30 seconds)...
$ pip install -q -r requirements.txt -r requirements-dev.txt
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.13.16, pytest-8.4.2, pluggy-1.6.0
rootdir: /builds/Siim.Rebane/iti8801-a7-reference
configfile: pytest.ini
collected 3 items
tests/test_api.py ...                                                    [100%]
========================= 3 passed, 1 warning in 0.66s =========================
Job succeeded
```

In the log	Means
`Starting service postgres:18-alpine...`	the runner started the database container before the first `script` line
`Waiting for services to be up and running`	the runner waited until the container's port answered
`...` after the file name	one dot per passed test; a failed one prints `F`
`3 passed`	`pytest` ended with 0

A test that stops a deploy

A commit makes `players` optional with a default of 0. The API description says a body without `players` is answered with 400. The test job:

```
tests/test_api.py ..F                                                    [100%]
_____________________________ test_players_is_required _____________________________
>       assert client.post("/api/boardgames", json={"name": "Catan"}).status_code == 400
E       AssertionError: assert 201 == 400
==================== 1 failed, 2 passed, 1 warning in 0.71s ====================
ERROR: Job failed: exit code 1
```

The pipeline's jobs after it:

```
test         failed
build        skipped
scan         skipped
deploy-dev   skipped
```

No image was built from this commit. The cluster still runs the image of the commit before:

```
$ kubectl get deployment api-dev -o wide
NAME      READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES
api-dev   1/1     1            1           6d12h   service      supax/iti8801-a5-reference:f4e65fbb
```

The image tag: the commit

Every pipeline needs a tag for the image it builds. The tag is the commit's short SHA:

```
variables:
  IMAGE: $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA     # <repository>:f4e65fbb
```

Property	Follows from it
a commit gives one tag, and a tag belongs to one commit	the tag in a Deployment says which commit runs
every pipeline pushes a new tag	no tag is overwritten; an older image is still in the registry under its own tag
the manifest's `image:` changes with every commit	`kubectl apply` sees a change and replaces the pods

A fixed tag such as `latest` has none of the three: the same name points at a different image after every push, a Deployment whose `image:` text did not change is not rolled out again, and a node or a registry cache that already holds `latest` may keep using the old one.

The example: the build job

```
build:
  stage: build
  image: docker:29
  script:
    - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
    - docker build --pull -t $IMAGE .
    - docker push $IMAGE
  rules:
    - if: $CI_COMMIT_BRANCH
```

Line	Means
`image: docker:29`	an image that contains the `docker` command
`docker login … --password-stdin`	log in to Docker Hub; the password comes through the pipe, so it is in no command line
`docker build --pull -t $IMAGE .`	build from the Dockerfile in the working directory; `--pull` asks the registry for the newest base image instead of using a stored one
`docker push $IMAGE`	upload the image under its commit tag
`rules: - if: $CI_COMMIT_BRANCH`	the job exists in pipelines started by a push to a branch

Where `docker build` runs

The `docker` command is a client: it sends requests to a Docker daemon, and the daemon builds and runs. A job's container holds the client. Two places can hold the daemon:

Arrangement	The daemon	Property
docker-in-docker	a second container next to the job (`services: [docker:29-dind]`), started for this job, gone after it	every job gets an empty daemon: no stored images, no other containers
the runner host's daemon	the Docker daemon of the runner machine itself, its socket made available inside the job's container	every job of every project talks to the same daemon

The TalTech runner uses the second arrangement. `docker info` in a job answers with the runner machine:

```
$ docker info --format '{{.Name}} {{.ServerVersion}} containers={{.Containers}} images={{.Images}}'
gitlab-common-runner-2025 29.8.0 containers=8 images=321
```

What a shared daemon means for your jobs

Fact	Consequence
images and build layers stay on the runner after a job	the first build of this lecture's image took 68 s, later ones 11 to 15 s: the layers were already there
a container a job starts itself is not removed when the job ends	a job removes what it started, in `after_script`; the containers of `services:` are started and removed by the runner
container and network names are unique per daemon	`--name api` in two jobs running at the same time: the second fails with `name is already in use`. Names carry `$CI_JOB_ID`
a job can list and inspect every container on that daemon, also other projects'	nothing secret is given to a container a job starts: no real password in `-e`
a base image stored on the runner is reused	`docker build --pull` checks the registry for a newer one

The registry credentials

The job logs in to Docker Hub with an **access token**: a second password made for one purpose, which can be deleted without changing the account's password. Docker Hub → Account settings → Personal access tokens, permission Read & Write.

The job reads three values that are stored in the GitLab project, not in the repository:

CI/CD variable	Holds
`DOCKER_USERNAME`	the Docker Hub user
`DOCKER_PASSWORD`	the access token
`DOCKER_IMAGE`	the repository: `<user>/<name>` on Docker Hub, `ghcr.io/<user>/<name>` on GitHub

GitHub Container Registry works the same way: `docker login ghcr.io`, a token with the scope `write:packages`, and `ghcr.io/<user>/<name>` as the image.

What the build log shows

```
$ docker build --pull -t $IMAGE .
#2 [internal] load metadata for docker.io/library/python:3.13-slim
#5 [build 1/4] FROM docker.io/library/python:3.13-slim@sha256:70729b46c69b…
#11 [build 4/4] RUN pip install --no-cache-dir --prefix=/install -r requirements.txt
#11 CACHED
#14 [stage-1 6/6] COPY main.py .
#14 CACHED
#15 naming to docker.io/supax/iti8801-a5-reference:f4e65fbb 0.0s done
$ docker push $IMAGE
The push refers to repository [docker.io/supax/iti8801-a5-reference]
f4e65fbb: digest: sha256:dc001cef026a… size: 2206
Job succeeded
```

In the log	Means
`$ docker build …`	a line of `script`, printed before its output
`load metadata for …python:3.13-slim`	`--pull`: the registry was asked which image the base tag names now
`CACHED`	the layer was already on the runner's daemon and was not built again
`Job succeeded`	every command ended with 0

Gates: jobs that stop a pipeline

A **gate** is a job whose failure stops every stage after it. The tests are one gate. Others look at the code and the image without running them:

Gate	Looks for	Tools
vulnerability scan	known vulnerabilities in the image's packages and in the dependencies	Trivy, Grype, GitLab Container Scanning
static analysis, SAST	bugs and insecure patterns in the source: SQL built from strings, input used unchecked	SonarQube, Semgrep, GitLab SAST
code quality	duplication, complexity, coverage under a threshold, formatting	SonarQube's quality gate; linters: `ruff`, `eslint`, `golangci-lint`, Checkstyle
secret detection	a token or a key committed to the repository	gitleaks, GitLab Secret Detection
license check	a dependency under a license the project may not use	license-finder, FOSSA

SonarQube is a server that analyses the code a job sends it and keeps the history per project. Its **quality gate** is a set of conditions on the new code of a merge request, such as coverage ≥ 80 %, no new bug, duplication < 3 %. The job fails when the gate fails, and the merge request shows which condition.

A gate is a check with a yes or no answer that needs no person. What needs a person is the merge request.

Why the image is scanned in every pipeline

A vulnerability scanner compares the packages inside an image with a database of known vulnerabilities. The database grows every day.

Fact	Consequence
a vulnerability is published after the image was built	an image with no findings last week has findings today, without any change in the repository
the base image is somebody else's	its packages are in your image, and their vulnerabilities with them
a scan done once by hand shows one day	a scan in every pipeline shows the day of every deploy

The scanner here is Trivy, the same as in Assignment 5, with the same rule: no finding of severity HIGH or CRITICAL for which a fixed version exists.

The example: the scan job

```yaml
scan:
  stage: scan
  image:
    name: aquasec/trivy:0.75.0
    entrypoint: [""]
  script:
    - trivy image --severity HIGH,CRITICAL --ignore-unfixed --exit-code 1 --cache-dir .trivy $IMAGE
  cache:
    paths: [.trivy/]
  rules:
    - if: $CI_COMMIT_BRANCH
    - if: $CI_COMMIT_TAG
```

Part	Means
`entrypoint: [""]`	the Trivy image starts `trivy` by itself; a job needs a shell, so the image's entrypoint is switched off
`trivy image … $IMAGE`	scan the image from the registry
`--severity HIGH,CRITICAL`	report only these two severities
`--ignore-unfixed`	leave out findings for which no fixed version exists
`--exit-code 1`	end with 1 when anything is reported; without it Trivy ends with 0 and the job passes whatever it finds
`--cache-dir .trivy`	keep the vulnerability database in this directory

Cache in the scan job

Every job starts with an empty container, so Trivy would download its vulnerability database, 121 MiB, in every pipeline. `cache:` names directories the runner stores when a job passed and puts back before the same job runs again.

The first run:

```
Checking cache for default-protected...
WARNING: Cache file does not exist
[vulndb] Downloading vulnerability DB...   repo="mirror.gcr.io/aquasec/trivy-db:2"
```

A later run:

```
Checking cache for default-protected...
Successfully extracted cache
…
Saving cache for successful job
.trivy/: found 6 matching artifact files and directories
```

A cache is for things that can be downloaded or computed again. A job must work when the cache is empty: the runner may drop it at any time.

A scan that fails

A commit sets the final stage's base image to `python:3.13.0-slim`, a tag from a year ago, and drops the line that installs its package updates. The tests pass, the build passes. The scan:

```
supax/iti8801-a5-reference:e014ed5e (debian 12.8)
=================================================
Total: 44 (HIGH: 37, CRITICAL: 7)
```

Library	Vulnerability	Severity	Status	Installed Version	Fixed Version
gpgv	CVE-2025-68973	HIGH	fixed	2.2.40-1.1	2.2.40-1.1+deb12u2
libcap2	CVE-2026-4878			1:2.66-4	1:2.66-4+deb12u3
libgnutls30	CVE-2026-33845	CRITICAL		3.7.9-2+deb12u3	3.7.9-2+deb12u7

```
…
ERROR: Job failed: exit code 1
```

```
test         success
build        success
scan         failed
deploy-dev   skipped
```

Every finding is a package of the base image, with a fixed version. Nothing in the service changed; the image is not deployed.

Getting the fixed package into the image

A base image holds the packages of the day it was built. `python:3.13-slim` is rebuilt by its publisher from time to time; between two rebuilds a package in it can get a published vulnerability and a fixed version, and `--pull` then still gives the image with the old package.

One line in the Dockerfile's final stage installs the updates that exist on the day of the build:

```
FROM python:3.13-slim
RUN apt-get update && apt-get upgrade -y && rm -rf /var/lib/apt/lists/*
```

Part	Means
`apt-get update`	fetch the current package lists
`apt-get upgrade -y`	install every newer version of the packages already in the image
`rm -rf /var/lib/apt/lists/*`	delete the package lists again: they are not needed at run time

The commit that puts the line back: scan `success`, deploy ran.

A later build shows `CACHED` for this line: the layer on the runner holds the updates of the day it was built. `docker build --no-cache` builds every layer again.

What a deploy job must do

Task	Because
reach the cluster with a credential	the job runs on the runner, outside the cluster
produce the manifests with this commit's image tag	the tag is new in every pipeline
apply them	the cluster starts replacing the pods
wait until the new pods are ready	`kubectl apply` returns as soon as the API server stored the objects, before any new pod runs
fail when they do not become ready	a deploy job that is green while the new pods crash says nothing

Environments: dev, staging, production

An **environment** is one running copy of the application with its own data and its own address.

Environment	Purpose	Gets a new version
dev	shows the newest state of `main`	automatically, on every push to `main` that passes test, build and scan
staging	a version chosen for release, in the shape of production: more than one replica, production's configuration	when a person decides, by a Git tag
production	the one users use	the image that ran in staging, by a decision and often a second person's approval

This lecture's pipeline has the first two; production would be a third deploy job of the same shape. Both environments of this lecture live in one namespace, so their objects have different names:

	dev	staging
Deployment, Service, Ingress	`api-dev`	`api-staging`
StatefulSet and its Service	`db-dev`	`db-staging`
Secret with the database password	`db-dev`	`db-staging`
host	`000007test-dev.k8s.digivader.com`	`000007test-staging.k8s.digivader.com`

One set of manifests for two environments

The two environments differ in four values. The manifests in the repository hold a placeholder where a value differs:

Placeholder	dev	staging
`$ENV`	`dev`	`staging`
`$HOST`	`000007test-dev.k8s.digivader.com`	`000007test-staging.k8s.digivader.com`
`$IMAGE`	the image of the pipeline's commit	the image of the tagged commit
`$REPLICAS`	`1`	`2`

`envsubst` is a small program that copies its input to its output and replaces `$NAME` with the value of the environment variable `NAME`:

```
envsubst '$ENV $HOST $IMAGE $REPLICAS' < k8s/api.yaml | kubectl apply -f -
```

Part	Means
`'$ENV $HOST $IMAGE $REPLICAS'`	replace only these four; every other `$` in the file stays as it is
`< k8s/api.yaml`	the file with placeholders is the input
`| kubectl apply -f -`	the result goes straight to `kubectl`; `-f -` reads the manifest from standard input

A manifest with placeholders

`k8s/api.yaml`, the Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api-$ENV
spec:
  replicas: $REPLICAS
  selector:
    matchLabels:
      app: api-$ENV
  template:
    metadata:
      labels:
        app: api-$ENV
    spec:
      containers:
      - name: service
        image: $IMAGE
        env:
        - name: DB_PASSWORD
          valueFrom:
            secretKeyRef: {name: db-$ENV, key: password}
        - name: DATABASE_URL
          value: postgresql://app:$(DB_PASSWORD)@db-$ENV:5432/app
```

`$(DB_PASSWORD)` is Kubernetes' own notation for another variable of the same container. It is not in `envsubst`'s list, and `envsubst` does not read the form `$(…)`, so it reaches the cluster unchanged. The file is not valid for `kubectl apply` by itself: `replicas: $REPLICAS` is not a number until `envsubst` ran.

The cluster credential in the pipeline

`kubectl` in the job needs what `kubectl` on a laptop needs: a kubeconfig file. It is stored in the GitLab project as a CI/CD variable of the type **File**:

	A variable of type Variable	A variable of type File
you enter	a value	the content of a file
the job gets	an environment variable with the value	a temporary file with the content, and an environment variable with the **path** of that file

The variable is named `KUBECONFIG`. `kubectl` reads the environment variable `KUBECONFIG` as the path of its configuration, so the job needs no further line: `kubectl get pods` in the job works as on the laptop.

The runner deletes the file when the job ends: `Cleaning up project directory and file based variables`.

Protected branches and tags

A **protected branch** is a branch with a rule about who may push to it and merge into it. The default branch `main` is protected in every new project: Maintainers may push and merge. A **protected tag** is the same for tag names: Settings → Repository → Protected tags, with a pattern such as `v*`.

A protected variable is given only to pipelines for those. `KUBECONFIG` is protected, the registry credentials are not:

Pipeline for	Registry credentials	**`KUBECONFIG`**
a push to `main`	yes	yes
a tag `v1.2`, with `v*` protected	yes	yes
a push to any other branch	yes	no

A branch whose `.gitlab-ci.yml` was changed to deploy from every branch:

```
$ for f in k8s/*.yaml; do envsubst … < $f | kubectl apply -f -; done
error: error validating "STDIN": … Get "http://localhost:8080/openapi/v2?timeout=32s":
  dial tcp [::1]:8080: connect: connection refused
ERROR: Job failed: exit code 1
```

Without the variable `kubectl` has no configuration and tries its built-in default, `localhost:8080`.

What the pipeline's credential can do

Fact	Detail
the kubeconfig in the variable is your own	the pipeline can do in your namespace everything you can: create, change and delete every object, read every Secret
whoever can change `.gitlab-ci.yml` on a protected branch controls what runs with it	the protection of `main` is the protection of the cluster credential
a job log is readable by every project member	a command that prints the file's content puts the credential into the log; a File variable is not masked
the file never goes into the repository	it is a credential; a credential that was committed once is in every clone and in the history, and has to be replaced

A credential with fewer rights, for example one that may only change Deployments, is a separate identity in the cluster with its own role.

The example: the deploy job

```yaml
.deploy:
  stage: deploy
  image: alpine/k8s:1.36.5
  script:
    - export HOST=$NAMESPACE-$ENV.k8s.digivader.com
    - for f in k8s/*.yaml; do envsubst '$ENV $HOST $IMAGE $REPLICAS' < $f | kubectl apply -f -; done
    - kubectl rollout status deployment/api-$ENV --timeout=120s

deploy-dev:
  extends: .deploy
  variables: { ENV: dev, REPLICAS: "1" }
  environment: { name: dev, url: "http://$NAMESPACE-dev.k8s.digivader.com" }
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

A job whose name starts with a dot is a **hidden job**: GitLab never runs it. `extends: .deploy` gives `deploy-dev` every keyword of `.deploy`; the keywords `deploy-dev` writes itself are added to them. The deploy steps are written once and used by one job per environment.

`NAMESPACE: "000007test"` is set in the file's top-level `variables:`.

The deploy job, line by line

Line	Means
`image: alpine/k8s:1.36.5`	an image with `kubectl` 1.36, `envsubst` and a shell
`export HOST=…`	the environment's host name, put together from the namespace and the environment
`for f in k8s/*.yaml; do … done`	for every manifest file: replace the placeholders, apply the result
`kubectl rollout status deployment/api-$ENV`	wait until the Deployment's new pods are ready and the old ones are gone; ends with 0 then
`--timeout=120s`	stop waiting after two minutes and end with 1
`variables: { ENV: dev, REPLICAS: "1" }`	the two values that are fixed for dev; `"1"` in quotes because a CI/CD variable is always text
`environment: { name: dev, url: … }`	GitLab records this job as a deployment to the environment `dev`
`if: $CI_COMMIT_BRANCH == "main"`	the job exists only in pipelines for `main`

The deploy log

The first deploy to an environment creates the objects:

```
$ for f in k8s/*.yaml; do envsubst '$ENV $HOST $IMAGE $REPLICAS' < $f | kubectl apply -f -; done
deployment.apps/api-dev created
service/api-dev created
ingress.networking.k8s.io/api-dev created
service/db-dev created
statefulset.apps/db-dev created
$ kubectl rollout status deployment/api-$ENV --timeout=120s
Waiting for deployment "api-dev" rollout to finish: 0 of 1 updated replicas are available...
deployment "api-dev" successfully rolled out
Job succeeded
```

A later deploy changes one thing, the Deployment's image:

```
deployment.apps/api-dev configured
service/api-dev unchanged
ingress.networking.k8s.io/api-dev unchanged
service/db-dev unchanged
statefulset.apps/db-dev configured
```

`configured` for the StatefulSet on every apply: the API server stores defaults in `volumeClaimTemplates` that the file does not have. The database pod is not restarted by it.

What exists before the first deploy

The manifests refer to one object the pipeline does not create: the Secret with the database password, one per environment. It is created by hand, once, as in Assignment 6:

```
$ kubectl create secret generic db-dev --from-literal=password="$(openssl rand -hex 16)"
secret/db-dev created
$ kubectl create secret generic db-staging --from-literal=password="$(openssl rand -hex 16)"
secret/db-staging created
```

Kept where	What
in the repository	the manifests, with the Secret's **name** in `secretKeyRef`
in the cluster only	the Secret, with the password
in the GitLab project's variables	the registry credentials, the kubeconfig

The password is in no file and in no variable: nothing outside the cluster needs it.

Two environments, one quota

Everything in the namespace counts against one quota. The requests and limits of the three sets in this lecture's namespace:

Set	Pods	CPU requests	CPU limits	Memory limits
Assignment 6	2 service, 1 database	200m	1500m	1024Mi
dev	1 service, 1 database	75m	500m	768Mi
staging	2 service, 1 database	100m	750m	1024Mi
total	8	**375m** of 500m	**2750m** of 6	**2816Mi** of 6Gi

A rolling update starts a new pod before it stops an old one, so for a moment a Deployment has one pod more, and its requests and limits count too. A quota that is full at that moment refuses the extra pod: nothing is created, and `kubectl rollout status` waits until its timeout. `kubectl describe replicaset` then names the resource that was refused.

`kubectl` around a deploy

Command	Does	Reach for it when
`kubectl apply -f -`	sends the manifests; returns when the API server stored them	the deploy job
`kubectl rollout status deployment/api`	waits until the new pods are ready and the old ones gone; ends with 1 at `--timeout`	the deploy job; "is it done?"
`kubectl get deployment api -o wide`	ready count and the wanted image	"which commit runs here?"
`kubectl get pods -l app=api`	each pod: ready or not, restarts, age	a rollout that does not finish
`kubectl describe pod <name>`	the events of one pod: image pull, probe failures, quota	the reason a pod is not ready
`kubectl logs <pod>`, `--previous`	the container's output; of the crashed container before a restart	a pod that restarts
`kubectl rollout history deployment/api`	the revisions the Deployment keeps, 10 by default	before an undo
`kubectl rollout undo deployment/api --to-revision=N`	back to the previous or to a named revision; the pods of that revision's template start	a broken version, now
`kubectl set image deployment/api service=<image>`	changes the image without a manifest; a new revision	a deploy or a rollback by hand, to a known tag
`kubectl scale deployment/api --replicas=3`	more or fewer pods of the same version	load; a test of a rolling update

Every change of the pod template, by `apply`, `set image` or `undo`, is a new revision and a new rollout.

Promotion: the same image, the next environment

Promotion is moving a version that runs in one environment to the next one without building it again. The image that was tested and ran in dev is the image that goes to staging, bit for bit.

The decision is a Git tag on a commit of `main`:

```
$ git tag v1.2
$ git push origin v1.2
```

Part	Means
`git tag v1.2`	give the current commit the name `v1.2`
`git push origin v1.2`	send the tag to GitLab; tags are not sent by a plain `git push`

GitLab starts a **tag pipeline** for the tagged commit. In it `CI_COMMIT_TAG` is `v1.2` and `CI_COMMIT_SHORT_SHA` is the same as in the commit's pipeline on `main`: `$IMAGE` names the image that already exists.

The staging job and the tag pipeline

```
deploy-staging:
  extends: .deploy
  variables: { ENV: staging, REPLICAS: "2" }
  environment: { name: staging, url: "http://$NAMESPACE-staging.k8s.digivader.com" }
  rules:
    - if: $CI_COMMIT_TAG
```

The tag pipeline for `v1.2` has two jobs: no test and no build:

```
scan             success   29 s
deploy-staging   success   35 s
```

The scan runs again on the existing image: a tag may be put on a commit days after its image was built, and the vulnerability database has grown since. An image whose scan failed on `main` fails here as well and does not reach staging.

Two independent mechanisms. A push to `main` builds a new image and puts it in dev. A tag takes an image that already exists and puts it in staging: the newest one, or any older one. Dev follows `main`; staging stays on the tagged commit until the next tag. So the two can run the same commit or different ones:

```
$ kubectl get deployments api-dev api-staging -o wide
NAME          READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS   IMAGES
api-dev       1/1     1            1           6d12h   service      supax/iti8801-a5-reference:55d33c65
api-staging   2/2     2            2           6d12h   service      supax/iti8801-a5-reference:f44b1c4f
```

Production: a decision, and often a second person

Mechanism	In GitLab
a job that waits for a click	`when: manual` on the job: the pipeline is green without it; a person starts it from the pipeline page, and GitLab records who and when
only some people may deploy	Operate → Environments → Protected environments: `production` deployable by Maintainers only; the job fails for anyone else
a required approval	an approval rule on the merge request into `main`, or on the protected environment: a second person before the deploy
a release named for people	a Git tag and a Release in GitLab with notes: what changed, which image

```yaml
deploy-prod:
  extends: .deploy
  variables: { ENV: prod, REPLICAS: "3" }
  environment: { name: production, url: "https://…" }
  rules:
    - if: $CI_COMMIT_TAG
      when: manual
```

A tag pipeline then deploys staging by itself and offers production as a button.

During a deploy: two versions, or none

A deploy replaces the pods of version A with pods of version B. How that happens is the **deployment strategy**, and it decides what users see in between:

Strategy	How	During the switch	Needs
recreate	stop every A pod, then start the B pods	nothing answers	nothing special; the only choice for a thing that cannot run twice
rolling update	start some B pods, wait until ready, stop some A pods, repeat	A and B answer at the same time; the share moves	B must become ready; A and B both work with the database
blue-green	B is started completely next to A; then all traffic is switched at once	A, then B; the switch is one change	two full copies at once; A kept until B is trusted
canary	B gets a small share of the traffic; the share grows while B is watched	most users see A, a few see B	a router that splits traffic by weight; metrics to watch

Kubernetes does the first two itself: `strategy: type: Recreate` or `RollingUpdate` on the Deployment, the second being the default. The other two are built from Deployments and a Service or an Ingress.

A **rollback** is putting the previous version back, by the same mechanism the deploy used.

Rolling update: what the Deployment does

In the Deployment manifest, under `spec:` next to `replicas:`; without it the defaults apply:

```
spec:
  replicas: 4
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 1          # at most 1 pod more than replicas during the update
      maxUnavailable: 0    # never fewer ready pods than replicas
```

Setting	Means
`maxSurge`	how many extra pods may exist: new ones are started before old ones are removed
`maxUnavailable`	how many of `replicas` may be not ready at a time; 0 keeps full capacity throughout
the readiness probe	the signal the rollout waits for: an old pod is removed only after a new one is ready. A new version that never becomes ready stops the rollout with the old pods still serving

The defaults are 25 % for both. The lines `kubectl rollout status` prints are this loop; for one replica with the defaults:

```
Waiting for deployment "api-dev" rollout to finish: 1 old replicas are pending termination...
deployment "api-dev" successfully rolled out
```

Blue-green: two Deployments, one Service

Two Deployments, `api-blue` and `api-green`, both labelled `app: api`, each with its own `version:` label. The Service selects one version:

```
kind: Service
metadata: { name: api }
spec:
  selector: { app: api, version: blue }
```

Step	How
start B next to A	`kubectl apply` of `api-green` with the new image; `rollout status`
test B before users see it	a second Service, or `kubectl port-forward`, that selects `version: green`
send the traffic to B	`kubectl patch service api -p '{"spec":{"selector":{"app":"api","version":"green"}}}'`: the selector now says `green`
put A back if B is wrong	the same patch with `blue`: seconds, no new pods
stop A once B is trusted	scale `api-blue` to 0; it becomes the next B

The price: twice the pods during the switch, and both versions use the same database, so the schema has to serve both.

Canary: a share of the traffic

A canary sends a small part of the requests to the new version and the rest to the old one, and the part grows while the new version is watched. Kubernetes has nothing for this by itself: a Service spreads requests evenly over the pods it selects, with no weights. So a canary is set up partly outside your manifests.

- **A weighted route in the Ingress controller.** Two Services, one per version, and a rule that sends 10 % of the traffic to the second. In Traefik, the cluster's controller, a `TraefikService` with weights; in NGINX Ingress, `canary` annotations; a service mesh does the same per request.
- **A replica ratio.** One Service selecting both versions, 9 pods of A and 1 of B. Works anywhere, but the share is coarse, follows the pod count, and cannot be 1 %.

While the canary runs, the error rate, the latency and the B pods' logs are watched. The share grows in steps; a bad step is reverted by setting the weight back, with no pods to start. Argo Rollouts and Flagger do the steps and the watching from metrics.

What it needs beyond a rolling update: two versions at once, and a way to tell them apart in the metrics.

What two versions at once ask of the application

Rolling update, blue-green and canary all have A and B running together. The application has to allow it:

Requirement	Because
database changes work with both versions	B adds a column and starts; A still writes rows without it. Expand first (add, nullable), deploy, contract later (remove, in a later release)
migrations run once, before B serves	not in every pod's start: a Job before the rollout, or an init step; a migration that can run twice does no harm the second time
no state in the pod's memory	a session kept in A's memory is gone when the next request lands on B; sessions and caches live in the database or a store
graceful shutdown	on SIGTERM the pod stops accepting, finishes the requests in flight, then exits; the Service stops sending to it as soon as it is terminating
a readiness probe that tells the truth	"ready" before the database connection exists sends requests into errors
a backward-compatible API	a client that got A's response format talks to B a second later

Recreate is the strategy for an application that cannot give these: it has downtime instead.

A deploy that cannot finish

A commit changes the readiness probe's path to `/healthz`, which the service does not have. Test, build and scan pass: the service is unchanged, the mistake is in the manifest. The new pod starts and never becomes ready, so the rolling update keeps the old pod. The deploy job:

```
deployment.apps/api-dev configured
$ kubectl rollout status deployment/api-$ENV --timeout=120s
Waiting for deployment "api-dev" rollout to finish: 1 old replicas are pending termination...
error: timed out waiting for the condition
ERROR: Job failed: exit code 1
```

The namespace after the job failed:

```
$ kubectl get pods -l app=api-dev
NAME                       READY   STATUS    RESTARTS   AGE
api-dev-66f7fb6d4f-bmh2b   0/1     Running   0          2m30s
api-dev-f8cf7b9c9-hhmfs    1/1     Running   0          4m46s
$ curl -s -o /dev/null -w "%{http_code}\n" http://000007test-dev.k8s.digivader.com/health
200
$ kubectl describe pod api-dev-66f7fb6d4f-bmh2b | tail -1
  Warning  Unhealthy  37s (x22 over 2m22s)  kubelet  Readiness probe failed: HTTP probe failed with statuscode: 404
```

The old pod still answers. The Deployment stays in the unfinished rollout until something changes it.

Rollback: getting back to a working state

Two ways lead back, and they differ in what the repository says afterwards.

By hand, in the cluster:

```
$ kubectl rollout undo deployment/api-dev
deployment.apps/api-dev rolled back
$ kubectl get pods -l app=api-dev
NAME                       READY   STATUS    RESTARTS   AGE
api-dev-f8cf7b9c9-hhmfs    1/1     Running   0          4m56s
```

The Deployment is back at its previous pod template. `main` still holds the broken commit: the next pipeline deploys the mistake again.

Through the repository:

```
$ git revert HEAD
$ git push
```

`git revert HEAD` creates a new commit that undoes the last one. Its pipeline tests, builds, scans and deploys: 93 s after the push dev ran the reverted state, and `main` describes what runs.

Environments in GitLab

The keyword `environment:` in a deploy job makes GitLab keep a record per environment: Operate → Environments in the project.

The page shows	From
the environments `dev` and `staging`, each with a link	`environment: name:` and `url:`
per environment the commit deployed last, by whom, when	the last successful job with that environment
the history of deployments	every run of such a job
a button to deploy an earlier one again	it runs that earlier pipeline's deploy job again, with that commit's image

The history of this lecture's project, as GitLab lists it:

```
23  staging  success  v1.2  f4e65fbb
24  dev      skipped  main  d3d908a0      the test failed
25  dev      success  main  54650805
26  dev      failed   main  da5a22e9      the rollout timed out
27  dev      success  main  f44b1c4f
28  staging  success  v1.3  f44b1c4f
29  dev      skipped  main  e014ed5e      the scan failed
30  dev      success  main  55d33c65
```

Branches and merge requests

A push to a branch other than `main` starts a pipeline with three jobs:

```
$ git checkout -b report-version
$ git push -u origin report-version
```

```
test    success   23 s
build   success   14 s
scan    success   34 s
```

The change is tested, built and scanned before it is anywhere near `main`, and nothing is deployed: `deploy-dev` is not part of this pipeline, and the cluster credential is not given to it.

A **merge request** asks to merge the branch into `main`. Its page shows the result of the branch's last pipeline next to the changes. When it is merged, `main` gets a new commit, and that commit's pipeline deploys to dev.

Settings → Merge requests → "Pipelines must succeed" makes GitLab refuse the merge while the branch's pipeline is not green.

The pipeline list is the deployment history

Build → Pipelines, newest first:

```
#33  success  main            55d33c65   Revert "Final stage on python:3.13.0-slim, without its package updates"
#32  failed   main            e014ed5e   Final stage on python:3.13.0-slim, without its package updates
#31  success  report-version  8fd44dea   The service reports its commit at /version
#30  success  v1.3            f44b1c4f
#29  success  main            f44b1c4f   Revert "Readiness probe on /healthz"
#28  failed   main            da5a22e9   Readiness probe on /healthz
#27  success  main            54650805   Revert "Players defaults to 0"
#26  failed   main            d3d908a0   Players defaults to 0
#25  success  v1.2            f4e65fbb
#24  success  main            f4e65fbb   The pipeline: test, build, scan, deploy
```

Question	Answer from the list
which commit runs in dev	the newest green pipeline on `main`: `55d33c65`
which commit runs in staging	the newest green tag pipeline: `v1.3`, `f44b1c4f`
why `da5a22e9` never ran	its pipeline failed, in the job the list links to

The whole file, part 1

```yaml
stages: [test, build, scan, deploy]

variables:
  IMAGE: $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
  NAMESPACE: "000007test"

test:
  stage: test
  image: python:3.13-slim
  services: [postgres:18-alpine]
  variables:
    POSTGRES_USER: app
    POSTGRES_PASSWORD: only-for-this-job
    POSTGRES_DB: app
    DATABASE_URL: postgresql://app:only-for-this-job@postgres:5432/app
  script:
    - pip install -q -r requirements.txt -r requirements-dev.txt
    - pytest
  rules:
    - if: $CI_COMMIT_BRANCH

build:
  stage: build
  image: docker:29
  script:
    - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
    - docker build --pull -t $IMAGE .
    - docker push $IMAGE
  rules:
    - if: $CI_COMMIT_BRANCH
```

The whole file, part 2

```yaml
scan:
  stage: scan
  image:
    name: aquasec/trivy:0.75.0
    entrypoint: [""]
  script:
    - trivy image --severity HIGH,CRITICAL --ignore-unfixed --exit-code 1 --cache-dir .trivy $IMAGE
  cache:
    paths: [.trivy/]
  rules:
    - if: $CI_COMMIT_BRANCH
    - if: $CI_COMMIT_TAG

.deploy:
  stage: deploy
  image: alpine/k8s:1.36.5
  script:
    - export HOST=$NAMESPACE-$ENV.k8s.digivader.com
    - for f in k8s/*.yaml; do envsubst '$ENV $HOST $IMAGE $REPLICAS' < $f | kubectl apply -f -; done
    - kubectl rollout status deployment/api-$ENV --timeout=120s

deploy-dev:
  extends: .deploy
  variables: { ENV: dev, REPLICAS: "1" }
  environment: { name: dev, url: "http://$NAMESPACE-dev.k8s.digivader.com" }
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

deploy-staging:
  extends: .deploy
  variables: { ENV: staging, REPLICAS: "2" }
  environment: { name: staging, url: "http://$NAMESPACE-staging.k8s.digivader.com" }
  rules:
    - if: $CI_COMMIT_TAG
```

What must exist outside the file

Where	What
GitLab project → Settings → CI/CD → Runners	instance runners turned on
GitLab project → Settings → CI/CD → Variables	`DOCKER_USERNAME`, `DOCKER_PASSWORD` (masked), `DOCKER_IMAGE`, `KUBECONFIG` (type File, protected)
GitLab project → Settings → Repository → Protected tags	the pattern `v*`
the registry	an access token with write permission; the repository public, so the cluster pulls without a login
the namespace	the Secrets `db-dev` and `db-staging`

Everything else is created by the pipeline.

How long it takes

Measured on the TalTech runner, for this lecture's service:

Job	Time
`test`	22 to 23 s
`build`, layers already on the runner	11 to 19 s
`build`, first time	68 s
`scan`, database from the cache	28 to 37 s
`deploy-dev`, a rolling update of one pod	19 to 27 s
`deploy-dev`, first time, the database created as well	60 s

From `git push` to the new version answering in dev: 81 to 97 s. A job that waits for the runner as `pending` adds to that.

Other CI systems

The parts of this lecture exist under other names in every CI system:

GitLab CI	GitHub Actions	Jenkins
`.gitlab-ci.yml`	files in `.github/workflows/`	`Jenkinsfile`
pipeline	workflow run	build
stage, job	job, step	stage, step
runner	runner	agent
CI/CD variables	secrets and variables	credentials
`rules:`	`on:` and `if:`	`when`

The managed ones run the jobs on the provider's machines: GitHub Actions, GitLab.com, Azure Pipelines, AWS CodeBuild. With Jenkins, or with a GitLab server of your own, the runners are machines you operate.



Questions