ITI8801 · Cloud Architectures and DevOps

Practice 7 · Assignment 7: a pipeline from a push to two environments

Siim Rebane · TalTech · Autumn 2026

What you build

A GitLab CI pipeline for your Assignment 5 service. A push to `main` runs your tests, builds the image, scans it and deploys it to a **dev** environment in your namespace. A Git tag deploys the same image to a **staging** environment.

```
http://<your student code in lower case>-dev.k8s.digivader.com/
http://<your student code in lower case>-staging.k8s.digivader.com/
```

Part	Written by you
the repository	your service's source, its Dockerfile, its tests, the manifests, `.gitlab-ci.yml`
the pipeline	five jobs: `build`, `test`, `scan`, `deploy-dev`, `deploy-staging`
the two environments	created and updated by the pipeline, never by `kubectl apply` from your machine

The repository

Requirement	Detail
where	a project on `gitlab.cs.taltech.ee`, in your own user namespace
visibility	**Private**
name	starts with `iti8801-2026`, for example `iti8801-2026-a7`. Add **Siim.Rebane** as a member with the Developer role (Manage → Members → Invite members): the validator reads your pipelines with that account
instance runners	turned on: Settings → CI/CD → Runners
it contains	`.gitlab-ci.yml` at the root, a `Dockerfile`, the manifests of your Deployment
it never contains	a kubeconfig, a manifest of `kind: Secret`, a token, a password, a private key

The credentials are CI/CD variables of the project: Settings → CI/CD → Variables.

The pipeline

The jobs have exactly these names:

Job	Runs in a pipeline for	Must
`test`	a push to a branch	run your service's tests with your language's test tool, against a real PostgreSQL, and fail when one fails
`build`	a push to a branch	build the image from the commit and push it to a public repository on Docker Hub or GitHub Container Registry, tagged with `$CI_COMMIT_SHORT_SHA`
`scan`	a push to a branch, and a tag	fail when the image has a vulnerability of severity HIGH or CRITICAL with a fix available
`deploy-dev`	a push to `main`	make dev run the image of the pipeline's commit, and fail when the new pods do not become ready
`deploy-staging`	a tag	make staging run the image of the tagged commit

The order on `main` is test, build, scan, deploy-dev: a job runs only when the one before it passed. A tag pipeline has no `test` and no `build`: it scans and deploys the image that already exists.

The two environments

Both live in your namespace on the course cluster, next to each other.

	dev	staging
host of the Ingress	`<code>-dev.k8s.digivader.com`	`<code>-staging.k8s.digivader.com`
Deployment of your service	at least 1 ready replica	at least 2 ready replicas
Service	`ClusterIP`	`ClusterIP`
database	its own PostgreSQL	its own PostgreSQL
image	`<repository>:<short SHA of a commit on main>`	the same repository, the short SHA of a tagged commit

The object names are yours to choose; every name is different between the two environments. An item created in dev does not exist in staging.

The database password of each environment is a Secret you create once with `kubectl create secret`, as in Assignment 6.

The end state, checked

- the whole Assignment 5 API answers at both host names
- the image that dev runs is tagged with a commit of your project, and that commit's pipeline on `main` has `test`, `build`, `scan` and `deploy-dev` green, in that order
- the image that staging runs is tagged with a commit that has a tag pipeline with `scan` and `deploy-staging` green and no `build`
- your project has a pipeline on `main` in which `scan` failed and `deploy-dev` did not run, and its commit runs in neither environment
- the repository holds no credential

The validator

`https://iti8801.digivader.com/lab7-validator/` : your student code and your GitLab project, as a URL or as `user/project`.

Box	Green when
Dev	the Ingress for `<code>-dev…` leads to a `ClusterIP` Service and a Deployment with a ready pod; every step of the A5 API test answers through it
Staging	the same through `<code>-staging…`, with 2 ready pods; a different Deployment than dev; an item created in dev is not in staging
Pipeline	the project is private and readable; dev's image tag is a commit of it; that commit's pipeline on `main` has the four jobs green, in that order
Promotion	staging's image tag is a commit with a tag pipeline: `scan` and `deploy-staging` green, no `build`; the same commit has a green `test`, `build`, `scan` on `main`; staging's image repository is dev's
Scan gate	a pipeline on `main` where `scan` failed and `deploy-dev` did not run; its commit runs in neither environment
Repository	`.gitlab-ci.yml`, a `Dockerfile` and a Deployment manifest are there; no Secret manifest, kubeconfig, token or private key in any file

The validator reads your namespace and your GitLab project; it changes nothing in either, except the items its API test creates and deletes. It checks that the end state exists; the assignment is graded by a person afterwards.

The quota with two more environments

Everything in your namespace shares one quota: requests of 0.5 CPU and 2 GiB of memory, limits of 6 CPU and 6 GiB, 4 volume claims.

Running	Counts
your Assignment 6 objects	whatever requests and limits you gave them, and one volume claim
dev	one service pod, one database pod, one volume claim
staging	two service pods, one database pod, one volume claim
a rolling update	one extra service pod for a few seconds

`kubectl describe quota` shows the sums. A deploy job that waits until its timeout, with the new pod never created, is a quota that is full: `kubectl describe replicaset` names the resource.

Your Assignment 6 objects may be deleted once your Assignment 6 check is PASSED.

What to submit, and when

- The validator's code of a check that is **PASSED**. Paste it into Moodle.
- Deadline: Sunday next week, 23:59, as for every assignment.
- A check is stored with its result: what you change in the project or the namespace afterwards does not change it.
- A **PASSED** check is the first step, not the grade: it shows that the end state exists. The assignment is then reviewed by the lecturer: the tests, the pipeline file, the credentials, the history. The slide "What the reviewer looks at" lists it.

Before the first push

the project	New project → Create blank project: name with the prefix `iti8801-2026`, Private, no README; Manage → Members → Invite members: `Siim.Rebane`, Developer; Settings → CI/CD → Runners: instance runners on
the code	your Assignment 5 directory, pushed to `main`
`.gitignore` before the first commit	`.env`, `*.kubeconfig`, `*.tfstate*`, `*.tfvars`, `.terraform/`, your language's build output
the check	`git status` before the commit: no `.env`, no kubeconfig, no state file, no token
the protected tag	Settings → Repository → Protected tags: `v*`, create: Maintainers. Without it a tag pipeline does not get `KUBECONFIG`

Settings → CI/CD → Variables, four times:

Key	Value	Type	Visibility	Protect
`DOCKER_USERNAME`	your registry user	Variable	Visible	no
`DOCKER_PASSWORD`	an access token of the registry with write permission	Variable	Masked	no
`DOCKER_IMAGE`	the repository, e.g. `docker.io/<user>/a5` or `ghcr.io/<user>/a5`	Variable	Visible	no
`KUBECONFIG`	the whole content of your `iti8801-<code>.yaml`	File	Visible	yes

The test job: your decisions

Decision	Options
the image	your language's: `python:3.13-slim`, `eclipse-temurin:21`, `node:22`, `golang:1.25`, `mcr.microsoft.com/dotnet/sdk:9.0`
the command	your test tool, the same as on your machine: `pytest`, `./gradlew test`, `npm test`, `go test ./...`, `dotnet test`
the database	`services: [postgres:18-alpine]` with the `POSTGRES_*` variables, host name `postgres`; or Testcontainers from the test code
the service's address	the variable your service reads, pointing at that database: `DATABASE_URL`, or host, user, password, name
what the tests cover	at least the Assignment 5 API: health, create, read, list, delete, and one body that must be refused
the result	the job is green only when every test passed; the test tool's exit code does that by itself

```yaml
test:
  stage: test
  image: <your language's image>
  services: [postgres:18-alpine]
  variables:
    POSTGRES_USER: app
    POSTGRES_PASSWORD: only-for-this-job
    POSTGRES_DB: app
    <your service's database variable>: <pointing at postgres:5432>
  script:
    - <install the test dependencies>
    - <your test command>
  rules:
    - if: $CI_COMMIT_BRANCH
```

A test that needs nothing from the pipeline is a test you can run on your machine first. Make it pass there.

The build job: your decisions

Decision	Options
the registry	Docker Hub: `docker login` with a Personal access token, Read & Write. GitHub Container Registry: `docker login ghcr.io` with a token of scope `write:packages`
the repository	**public**: the cluster pulls without a login. `DOCKER_IMAGE` holds its name
the tag	`$CI_COMMIT_SHORT_SHA`, through `IMAGE: $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA` in the top-level `variables:`
the Dockerfile	the one from Assignment 5, built from the repository's root with `--pull`

```yaml
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
```

Green when the registry lists a tag that is the commit's first 8 characters. The same `docker build` on your machine finds a Dockerfile error faster than a pipeline does.

The scan job

```
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
```

A green scan ends with a table of zeros. A red one names the package, the installed version and the fixed version; the fix is in the Dockerfile: a newer base image tag, the base image's package updates (`apt-get upgrade`, `apk upgrade`), or a newer version of the library.

An image that passed in Assignment 5 may fail today: the vulnerability database has grown since.

The manifests for two environments

Copy your Assignment 6 manifests into `k8s/` and put a placeholder where dev and staging differ:

Placeholder	Where
`$ENV`	in every object name, label and selector: `api-$ENV`, `db-$ENV`; in the Secret's name in `secretKeyRef`; in the database host of your service's address
`$HOST`	the Ingress host
`$IMAGE`	the Deployment's `image:`
`$REPLICAS`	the Deployment's `replicas:`

Keep the requests and limits of your Assignment 6 manifests. Check the result on your machine before any pipeline does:

```
$ ENV=dev HOST=<code>-dev.k8s.digivader.com IMAGE=<repository>:<tag> REPLICAS=1 \
    envsubst '$ENV $HOST $IMAGE $REPLICAS' < k8s/api.yaml | kubectl apply --dry-run=client -f -
deployment.apps/api-dev created (dry run)
```

`envsubst` is in the package `gettext`: `brew install gettext`, `apt install gettext-base`; Git Bash on Windows has it. Then the two Secrets, by hand, once:

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

The deploy jobs

```
.deploy:
  stage: deploy
  image: alpine/k8s:1.36.5
  script:
    - export HOST=<code>-$ENV.k8s.digivader.com
    - for f in k8s/*.yaml; do envsubst '$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://<code>-dev.k8s.digivader.com" }
  rules:
    - if: $CI_COMMIT_BRANCH == "main"

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

the credential	`KUBECONFIG` as a File variable, protected: `kubectl` reads the path from the variable, nothing else to write
the first deploy	prints `created` for every object and waits for the pod; later ones print `configured` for the Deployment
green	`deployment "api-dev" successfully rolled out`; `curl http://<code>-dev…/health` answers
promotion	`git tag v1.0 && git push origin v1.0` on a commit whose `main` pipeline is green; the tag pipeline is `scan` and `deploy-staging`

The scan gate, on purpose

The validator looks for one pipeline on `main` in which `scan` failed and `deploy-dev` did not run. Produce it: one commit that puts a known vulnerability into the image, then one that takes it out again.

```
# the Dockerfile's final stage: an old tag of the base image, and no line that installs its updates
FROM python:3.13.0-slim
```

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

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

The reverting commit's pipeline is green and deploys. The commit with the old base image runs in neither environment.

A base image tag that is a year old has findings with a fix. A line that installs the base image's updates in the same stage (`apt-get upgrade`, `apk upgrade`) makes the old tag pass: remove that line in the same commit. An old library version with a known vulnerability in your dependency file fails the scan as well.

`kubectl` while a deploy is red

Command	Shows
`kubectl get pods -l app=api-dev`	the new pod next to the old one: `0/1` ready, restarts
`kubectl describe pod <new pod>`	the events: image pull, readiness probe failures, quota
`kubectl logs <new pod>`	the service's own error: the database address, a missing variable
`kubectl get deployment api-dev -o wide`	the image the Deployment wants now
`kubectl rollout undo deployment/api-dev`	back to the previous revision, now; then fix the commit and push
`kubectl describe quota`	the sums, when a pod is never created

The pipeline deploys again on the next push to `main`; nothing you fixed by hand in the cluster survives it.

When a job is red

Job	In its log	Look at
`build`	`unauthorized` or `denied` at login or push	the three registry variables; the token's write permission; a typing mistake in `DOCKER_IMAGE`
`build`	an error in a Dockerfile step	the same `docker build` on your machine
`test`	`connection refused` or `could not translate host name` towards the database	`services:` is missing, or the host name is not `postgres`; the `POSTGRES_*` variables
`test`	`No module named`, `command not found`, `cannot find module`	the test libraries are not installed in the job; the test tool is run from the repository's root
`test`	a failing assertion	the test is right and the service is wrong, or the other way round: the same test on your machine
`scan`	a table with a HIGH or CRITICAL row	the package and its fixed version; the Dockerfile
`scan`	`unable to find the specified image`	`build` did not run for this commit: a tag on a commit that was never built
`deploy-…`	`localhost:8080 … connection refused`	`KUBECONFIG` is missing: its type File, its key; the branch or tag not protected
`deploy-…`	`Forbidden`	the kubeconfig is someone else's or an old one; download yours again
`deploy-…`	`error: timed out waiting for the condition`	`kubectl get pods`, `kubectl describe pod`: the image name, the probe, a missing Secret, the quota
`deploy-…`	`exceeded quota`	`kubectl describe quota`

What the reviewer looks at

The validator checks that the end state exists. The grade comes from reading your project:

Looked at	Good	Not good
the tests	they call the API and check the answers, including a refused body; they pass for a reason	a test that asserts nothing; a test commented out to make the job green
the pipeline file	the five jobs, readable; `rules` that say when each runs	jobs copied with parts that do nothing; a job that always passes
the credentials	in the variables, masked where possible; never in the repository or a job log	a token in a file; a kubeconfig printed by `cat`
the history	commit messages that say what changed; the red pipelines the story needs, and their reverts	a hundred "fix" commits; rewritten history
the environments	dev and staging run different commits at some point; staging has 2 replicas; a scan that stopped a deploy	staging deployed from `main` by hand
the manifests	placeholders, lowered limits, the Secrets created once	a password in the Deployment

What stays where

Thing	Lives in
source, Dockerfile, tests, manifests with placeholders, `.gitlab-ci.yml`	the repository
registry user, registry token, image repository, kubeconfig	the project's CI/CD variables
the database passwords	Secrets in your namespace, made with `kubectl create secret`
the images, one tag per commit	your public registry repository
which commit runs where	the Deployments' image tags; Operate → Environments in the project

The kubeconfig is your password to the cluster. A kubeconfig that reached a repository, a job log or a chat is replaced: tell the lecturer, and download the new one.



Questions