Veebirakenduse projekt (ITI0302)

Loeng 3

CI/CD: a pipeline that deploys on every push · Siim Rebane

Today

1. The problem: the five-step deployment by hand, and what goes wrong with it
2. What a pipeline is, and where it runs: GitLab CI, runners, jobs in containers
3. The file that describes it, `.gitlab-ci.yml`, stage by stage: build, image, deploy
4. Secrets: CI/CD variables, a Docker Hub token, a deploy key
5. `/api/version`: how the running application learns which commit it was built from
6. Branches, merge requests, and the green-pipeline rule
7. What the lab does with all of this

The problem

Every change to the application means, by hand:

1. build the jar
2. copy it to the server
3. build the image there
4. stop the old container, start the new one
5. check the log and the URL

Five steps, two machines. Each one is a chance to forget something, to copy the wrong jar, or to leave the old container running. And it depends on one laptop that has the right key and the right files on it.

Individual assignment 1 requires that these steps run **without you**, on every push, and that the result is visible as green in GitLab.

What goes wrong when it is done by hand

What happens	What you see
The jar on the server is older than the code in git	"it works on my laptop"
A teammate deploys, you deploy over it	the last one wins, nobody knows what runs
The laptop with the key is at home	nothing can be deployed today
Tests were not run before deploying	the bug is found by the user
Nobody remembers what was deployed when	a bug appears; which change caused it?

Every one of these has the same cause: the steps live in a person's head, not in the repository.

The idea: the steps live in the repository

Put the five steps into a file, next to the code. Have a machine read that file and run the steps on every push. Then:

- every commit is built the same way, on a clean machine, every time
- the result is recorded: which commit, which step, passed or failed, with the full log
- the running application always corresponds to a commit in git
- anyone with push rights can deploy, from any laptop, by pushing

That machine-run sequence of steps is a **pipeline**. Writing the steps down once, and letting the machine repeat them, is the whole idea of this lecture.

Two terms: CI and CD

Continuous Integration (CI): every push is built and tested automatically. The point is to find out within minutes, not weeks, that a change breaks the build or the tests. "Integration" refers to integrating each developer's change into the shared code, continuously, instead of in one big merge at the end.

Continuous Delivery / Deployment (CD): every build that passes is also put on a server automatically. Delivery: made ready to deploy. Deployment: actually deployed, without a human step.

Assignment 1 asks for both: the push is built (CI) and the result runs on your server (CD). The pipeline you write today does CI in its first stage and CD in the other two.

Where it runs: GitLab CI

GitLab, where your repository lives, has a pipeline system built in: **GitLab CI/CD**. Nothing to install and no other service to sign up for.

How it fits together:

1. You commit a file named `.gitlab-ci.yml` in the root of the repository
2. On every push GitLab reads the file and creates a **pipeline**: the set of jobs for that commit
3. Each **job** is handed to a **runner**: a machine that executes the job's commands
4. The job's output is stored as a log; the job ends green (succeeded) or red (failed)
5. The pipeline is green only if every job is green

Runner is the term to hold on to: it is the machine that does the work, and it is not your laptop.

The path of one push

Five machines take part, and each hands one thing to the next:

Step	From	To	What travels
1	your laptop	GitLab	the commit, by `git push`
2	GitLab	a runner	the job: which image, which commands, which commit to clone
3	the runner	Docker Hub	the image, by `docker push`
4	the runner	your server	three SSH commands: copy the Compose file, pull, restart
5	your server	Docker Hub	the image, by `docker compose pull`

Your laptop is involved only in step 1. Everything after the push happens between servers, and it happens the same way whether you pushed from home, from the lab, or from a teammate's machine.

Runners at TalTech

A runner is a program, `gitlab-runner`, registered with the GitLab server. It polls for jobs, runs them, and sends the log back. The university runs runners for everyone on gitlab.cs.taltech.ee; these are **shared runners**.

The runner that ran the pipeline in this deck:

```
Running with gitlab-runner 19.3.1 (a16f5092)
Using Docker executor with image maven:3.9-eclipse-temurin-25 ...
```

"Generic runner on Taltech cloud with Docker executor". **Docker executor** means: every job runs in a fresh container, from an image the job names. The container is created for the job and thrown away after it. Nothing you install in one job exists in the next.

A new project has shared runners switched off. Settings → CI/CD → Runners → "Enable shared runners for this project". Until you do that, every pipeline stays *pending* forever.

A job is a fresh container plus your commands

Because of the Docker executor, each job is:

1. pull the image the job names (`maven:…`, `docker:…`, `alpine:…`)
2. start a container from it
3. clone your repository into it, at the commit being built
4. run the `script` lines, one after another, in a shell
5. save what you asked to keep, throw the container away

Consequences you will meet:

- the job has only what the image has. A Maven image has Maven and a JDK; an Alpine image has almost nothing
- a file made in one job is gone in the next, unless it is declared an **artifact**
- the first line that exits with an error stops the job and makes it red

What a job's log tells you

Every job log has the same shape. The one from the build job of this deck's pipeline:

```
Running with gitlab-runner 19.3.1 (a16f5092)            ← which runner took the job
Using Docker executor with image maven:3.9-eclipse-temurin-25 ...
Pulling docker image maven:3.9-eclipse-temurin-25 ...    ← the container is being prepared
Getting source from Git repository                       ← your commit is cloned
$ ./mvnw -B clean package -DskipTests                    ← each script line, prefixed with $
[INFO] BUILD SUCCESS
$ cp target/*.jar app.jar
Uploading artifacts for successful job
Job succeeded                                            ← or: ERROR: Job failed: exit code 1
```

Lines starting with `$` are your commands; what follows each one is its output. Everything before the first `$` is the runner setting up, and a problem there is the image name or the runner, not your code.

Exit codes: how the runner knows a step failed

Every program on Linux ends with a number, its **exit code**: `0` means success, anything else means failure. `mvn` exits with `1` when compilation fails; `docker push` exits with `1` when the registry refuses; `ssh` returns the exit code of the command it ran on the server.

The runner runs the `script` lines one by one and checks the code after each:

- code `0`: the next line runs
- anything else: the job stops there, is marked failed, and the log ends with `ERROR: Job failed: exit code 1`

Two consequences. A line that fails does not let the lines after it run, so a red `dockerize` never pushes half an image. And a command that prints an error but exits with `0` does not stop the job; the runner trusts the code, not the words.

The file: `.gitlab-ci.yml`

YAML, in the root of the repository, exactly this name. The skeleton:

```yaml
stages:
  - build
  - dockerize
  - deploy

build:        # a job; runs in stage "build"
  stage: build
  image: maven:3.9-eclipse-temurin-25
  script:
    - ./mvnw -B clean package -DskipTests
```

- `stages` lists the stages in order. Jobs of one stage run before the next stage starts; if any job fails, the later stages do not run
- everything else at the top level is a job. Its name is the key; `stage` puts it in a stage; `image` picks the container; `script` is the list of shell commands

Indentation is the syntax. Two spaces, never tabs.

YAML: the notation of the file

YAML is a text format for structured data, chosen by GitLab for this file because it is readable and has little punctuation. Three shapes are enough:

```yaml
key: value              # a pair
name: build
image: maven:3.9-eclipse-temurin-25

list:                   # a list: one item per line, "- "
  - first
  - second

nested:                 # nesting: indent by two spaces
  inner: value
  deeper:
    - item
```

Rules that bite:

- indentation is the structure. Two spaces per level, and **never a tab**; a tab is a syntax error
- a `:` followed by a space inside a value starts a new pair by mistake. Quote such lines: `- echo "port: 8080"`
- `[a, b]` is the same list on one line; `{ k: v }` the same nesting on one line
- `#` starts a comment

Variables GitLab gives every job

An **environment variable** is a named value that a process gets from whoever started it, readable as `$NAME` in a shell. The runner starts every job with a set of them describing the pipeline. The ones this deck uses, with their values from the run it was made from:

Variable	Value in the run	Meaning
`CI_COMMIT_SHORT_SHA`	`93c82ab8`	first eight characters of the commit being built
`CI_COMMIT_SHA`	`93c82ab8…` (40 characters)	the full commit id
`CI_COMMIT_BRANCH`	`main`	the branch that was pushed
`CI_PROJECT_NAME`	`iti0302-ci-test`	the repository's name
`CI_PIPELINE_ID`	`158739`	the number in the pipeline's URL
`CI_JOB_NAME`	`build`	the job running right now

They exist in every job without any setup. Your own variables, added in the project settings, appear the same way, and the shell does not distinguish the two.

The three stages of assignment 1's pipeline

Stage	Job	Runs in	Does	Leaves behind
build	`build`	Maven or Gradle image	compiles, packages `app.jar`	`app.jar` as an artifact
dockerize	`dockerize`	`docker` image	builds the image from `app.jar`, pushes it to Docker Hub	an image on Docker Hub, tagged with the commit
deploy	`deploy`	`alpine` image	connects to your server over SSH, pulls the image, restarts the container	the new commit running

Real times from the run this deck was made from, on the TalTech shared runner:

build	dockerize	deploy	push to live
37 s	69 s	17 s	about 2 minutes

Stage 1: build (Maven)

```
build:
  stage: build
  image: maven:3.9-eclipse-temurin-25
  script:
    - ./mvnw -B clean package -DskipTests
    - cp target/*.jar app.jar
  artifacts:
    paths:
      - app.jar
```

- `maven:3.9-eclipse-temurin-25` : Maven 3.9 with Java 25, the same Java your project declares. The image tag must match your `java.version` , or the build fails with "invalid target release"
- `-B` : batch mode, no progress bars in the log
- `-DskipTests` : build without running tests. Dropping it runs them, and a failing test makes the job red
- `cp target/*.jar app.jar` : one fixed name, so that the Dockerfile does not depend on your project's version number

Stage 1: build (Gradle)

```
build:
  stage: build
  image: gradle:9-jdk25
  script:
    - chmod +x ./gradlew
    - ./gradlew clean bootJar
    - cp build/libs/*.jar app.jar
  artifacts:
    paths:
      - app.jar
```

- `gradle:9-jdk25` : Gradle 9 with Java 25. The wrapper still downloads the exact Gradle version the project pins, `gradle-9.7.1-bin.zip` in the run below
- `chmod +x` : the wrapper script sometimes loses its executable bit in git on Windows
- `bootJar` builds the runnable jar and nothing else, so `build/libs` holds one file. `./gradlew build` would also produce a `-plain.jar` without dependencies, which is not the one to ship

Stage 1 output: the same job with Gradle

From the run on the test project's Gradle branch:

```
$ chmod +x ./gradlew
$ ./gradlew clean bootJar
Downloading https://services.gradle.org/distributions/gradle-9.7.1-bin.zip
Welcome to Gradle 9.7.1!
BUILD SUCCESSFUL in 29s
5 actionable tasks: 4 executed, 1 up-to-date
$ cp build/libs/*.jar app.jar
Uploading artifacts for successful job
Job succeeded
```

The first line of Gradle's output is the wrapper fetching Gradle itself, which the image's own Gradle does not prevent: the wrapper always uses the version pinned in `gradle/wrapper/gradle-wrapper.properties`, so every developer and the runner build with the same one.

Maven or Gradle, the result is the same: `app.jar` in the repository root of the job, declared as an artifact.

Artifacts: what survives a job

The container is thrown away when the job ends. `artifacts` names files that GitLab uploads and keeps:

```
Uploading artifacts for successful job
app.jar: found 1 matching artifact files and directories
Uploading artifacts as "archive" to coordinator... 201 Created
```

Every job in a later stage gets the artifacts of earlier jobs downloaded into its container before its `script` runs. That is how `dockerize` finds `app.jar` without building it again.

Artifacts are also downloadable from the job page in GitLab, which is a convenient way to get the jar of any commit.

Tests in the pipeline

Remove `-DskipTests` and Maven runs the tests before packaging. A Spring Boot project has one from the start, the context-loads test, and the log shows it:

```
[INFO]  T E S T S
[INFO] Running ee.taltech.lab1.Lab1SampleApplicationTests
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 2.363 s
[INFO] BUILD SUCCESS
```

A failing test makes `mvn` exit with `1`, the job turns red, and nothing after it runs: no image is pushed and the server is not touched. That is the whole value of the build stage coming first.

Test reports on the pipeline page

GitLab can show test results as a list instead of a log, if the job reports them. Maven's Surefire plugin writes one XML file per test class; declaring them as a report makes GitLab read them:

```
artifacts:
  paths:
    - app.jar
  reports:
    junit: target/surefire-reports/TEST-*.xml
```

Gradle writes the same files under `build/test-results/test/TEST-*.xml`. With the report, the merge request page lists which tests passed and which failed.

Cache: dependencies that do not change between builds

Every job starts in a fresh container, so every build downloads the project's dependencies from Maven Central again: hundreds of files, a large part of the build time. `cache:` keeps a directory between jobs on the runner:

```yaml
build:
  stage: build
  image: maven:3.9-eclipse-temurin-25
  variables:
    MAVEN_OPTS: "-Dmaven.repo.local=.m2/repository"
  cache:
    key: maven
    paths:
      - .m2/repository
  script:
    - ./mvnw -B clean package
```

`MAVEN_OPTS` moves Maven's download directory into the project folder, where the runner can save it. `key` names the cache; every job using the same key shares it.

Cache: what it changed, measured

Two consecutive builds on the test project, the first with an empty cache:

	First build	Second build
log	`WARNING: Cache file does not exist` … `Created cache`	`Successfully extracted cache`
downloads from Maven Central	hundreds	none
Maven's own time	13.9 s	5.7 s
the whole build job	34 s	20 s

The first build pays for the download and saves the directory at the end; every later build restores it before `script` runs.

A cache is a speed-up, not a guarantee: the runner may have dropped it, so a job must still work when the cache is empty. That is the difference from an artifact, which is always there for the next stage.

Cache for Gradle

The same idea with Gradle's names. Gradle keeps downloads in its *user home*; moving that into the project folder lets the runner save it:

```
build:
  stage: build
  image: gradle:9-jdk25
  variables:
    GRADLE_USER_HOME: $CI_PROJECT_DIR/.gradle
  cache:
    key: gradle
    paths:
      - .gradle/
  script:
    - chmod +x ./gradlew
    - ./gradlew clean bootJar
    - cp build/libs/*.jar app.jar
```

That directory holds the downloaded Gradle distribution as well as the dependencies, so the second build skips both. Measured on the test project: the build job went from 65 s to 30 s, Gradle's own time from 29 s to 8 s. Add `.gradle/` to `.gitignore`; it must not be committed.

Stage 1 output: what a green build looks like

```
[INFO] BUILD SUCCESS
[INFO] Total time:  13.894 s
$ cp target/*.jar app.jar
Uploading artifacts for successful job
Job succeeded
```

And a red one, from a compile error:

```
[ERROR] COMPILATION ERROR :
[ERROR] /builds/.../VersionController.java:[17,33] cannot find symbol
[ERROR]   symbol:   variable committ
[INFO] BUILD FAILURE
ERROR: Job failed: exit code 1
```

The job after it, `dockerize`, is marked *skipped*, and the pipeline is red.

The log is the whole story. When a job is red, read the log from the first `ERROR` line, not from the end.

Stage 2: where the image goes

Stage 1 made a jar. The server needs an image. In lecture 2 the image was built on the server, from a jar copied there by hand. The pipeline does it the other way round: build the image once, in the pipeline, and put it somewhere the server can fetch it from.

That somewhere is a **registry**: a server that stores images under a name and a tag, the way git stores commits under a branch. `docker push` uploads an image to a registry; `docker pull` downloads one. Docker Hub, hub.docker.com, is the public registry Docker uses by default and where `eclipse-temurin`, the base image of your Dockerfile, comes from.

You use it in the other direction: your pipeline pushes your image there, your server pulls it from there. A free Docker Hub account is enough.

Image names on Docker Hub

An image on Docker Hub is named `<account>/<repository>:<tag>`:

```
supax/iti0302-ci-test:93c82ab8
supax/iti0302-ci-test:latest
```

- `supax` is the Docker Hub account; yours is your Docker Hub username
- `iti0302-ci-test` is the repository, created automatically on the first push. **Repositories created that way are public**: anyone can pull the image. For assignment 1 that is fine. To keep one private, create it on hub.docker.com first and tick Private
- the tag says which build. The pipeline pushes every image twice: once tagged with the **commit** it was built from, once as **`latest`**

Two tags, one image. `latest` is what the server pulls; the commit tag is there so that any past build can be pulled back by name.

`latest` is only a name

A tag is a label on an image, nothing more. `latest` does not mean "the newest"; it means "the image someone last pushed under the name latest". If anything other than `main` ever pushed `latest`, the server would pull that, which is why `dockerize` runs only for `main`.

That is why every build is pushed twice. The commit tag is the honest one: `supax/iti0302-ci-test:93c82ab8` is the build of commit `93c82ab8`, today and in a year. `latest` is the convenience the server pulls by default.

The commit tag gives you a way back. If the deploy of `11d55df0` turns out to be broken, on the server:

```
cd /opt/app
sed -i 's/:latest/:93c82ab8/' compose.yaml     # or edit the tag with nano
docker compose up -d
```

The previous build runs again within seconds, from Docker Hub, with no rebuild. Fix the code, push, and the next deploy replaces it with a new `latest`. Change the tag back afterwards, or the deploy job will keep replacing your `compose.yaml` with the one from git, which says `latest`.

Stage 2: dockerize

```
dockerize:
  stage: dockerize
  image: docker:latest
  services:
    - docker:dind
  variables:
    DOCKER_TLS_CERTDIR: ""
  script:
    - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
    - docker build --build-arg COMMIT=$CI_COMMIT_SHORT_SHA
        -t $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA -t $DOCKER_IMAGE:latest .
    - docker push $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
    - docker push $DOCKER_IMAGE:latest
  only:
    - main
```

Four commands: log in, build with two tags, push, push. `only: main`: images are pushed only for `main`; a feature branch stops after `build`. `$DOCKER_IMAGE` is `youraccount/yourrepo`, set once as a variable. `$CI_COMMIT_SHORT_SHA` is the first eight characters of the commit being built, provided by GitLab in every job.

Docker inside a job: docker-in-docker

The job runs in a container. `docker build` needs a Docker daemon, and a container has none. `services: - docker:dind` starts a second container next to the job, running a Docker daemon ("Docker in Docker"), and the `docker` command in the job talks to it.

```
  image: docker:latest        # has the docker command
  services:
    - docker:dind             # has the docker daemon
  variables:
    DOCKER_TLS_CERTDIR: ""    # plain connection between the two; no certificates to set up
```

This block is copied as is. The log shows the daemon starting, and once the build runs you see the familiar lines:

```
#11 naming to docker.io/supax/iti0302-ci-test:93c82ab8 0.0s done
#11 naming to docker.io/supax/iti0302-ci-test:latest done
93c82ab8: digest: sha256:8dce4402... size: 2203
latest: digest: sha256:8dce4402... size: 2203
```

What the dockerize log shows

From the run this deck was made from, the lines that matter:

```
Pulling docker image docker:dind ...
*** WARNING: Service runner-…-docker-0 probably didn't start properly.
Pulling docker image docker:latest ...
$ echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
Login Succeeded
WARNING! Your credentials are stored unencrypted in '/root/.docker/config.json'.
$ docker build --build-arg COMMIT=93c82ab8 -t supax/iti0302-ci-test:93c82ab8 -t supax/iti0302-ci-test:latest .
#11 naming to docker.io/supax/iti0302-ci-test:93c82ab8 0.0s done
$ docker push supax/iti0302-ci-test:93c82ab8
93c82ab8: digest: sha256:8dce4402… size: 2203
Job succeeded
```

- the `WARNING` about the service appears on the TalTech runners even when the daemon is fine; the proof that it is fine is `Login Succeeded` a few lines later
- the credentials warning is about the job's own container, which is thrown away after the job
- `digest: sha256:…` is Docker Hub confirming the upload; the same digest appears for both tags, because it is one image

Stage 3: deploy

```
deploy:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh && chmod 700 ~/.ssh
  script:
    - scp -o StrictHostKeyChecking=no compose.yaml ubuntu@$SERVER_IP:/opt/app/compose.yaml
    - ssh -o StrictHostKeyChecking=no ubuntu@$SERVER_IP
        "cd /opt/app && echo DOCKER_IMAGE=$DOCKER_IMAGE > .env && docker compose pull && docker compose up -d"
  only:
    - main
```

The job is an SSH client and nothing else. It logs in to your server the way you do, with a key, and runs the same commands you ran in lab 2, in `/opt/app` instead of your home directory: `/opt` is where installed applications belong on a Linux server, and a directory there is set up once with `sudo mkdir -p /opt/app && sudo chown ubuntu:ubuntu /opt/app`.

Stage 3, line by line: getting the key into the job

`before_script` runs before `script`, in the same container. Here it prepares SSH:

Line	Does
`image: alpine:latest`	the smallest Linux; it has no SSH, so the first line installs it
`apk add openssh-client`	Alpine's package manager; gives `ssh` and `scp`
`eval $(ssh-agent -s)`	starts the agent that holds keys for this shell
`echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -`	loads the private key from a variable into the agent; `tr` strips Windows line endings that break keys
`mkdir -p ~/.ssh && chmod 700 ~/.ssh`	the directory SSH expects, with the permissions it insists on

After these four lines the job can open SSH connections as the deploy key's owner, exactly as your laptop does with your own key.

Stage 3, line by line: what happens on the server

Line	Does
`StrictHostKeyChecking=no`	accepts the server's fingerprint without asking; a job cannot answer `yes`
`/opt/app`	where the application lives on the server: `/opt` is the conventional place for installed applications. Created once by hand, owned by `ubuntu`, so the job may write there
`scp compose.yaml …`	the Compose file is in git; the server gets the current one every deploy
`echo DOCKER_IMAGE=… > .env`	Compose reads `.env` next to the file; that is how `${DOCKER_IMAGE}` in it gets its value
`docker compose pull && docker compose up -d`	fetch the new `latest`, replace the running container
`only: - main`	deploy only for pushes to `main`; other branches build and dockerize but do not touch the server

Host keys, and why the deploy job says `StrictHostKeyChecking=no`

A server has its own key pair too, its **host key**. On the first connection `ssh` shows its fingerprint and asks whether to trust it; a `yes` stores it in `~/.ssh/known_hosts`, and from then on `ssh` refuses to connect if the server's key changes. That is how you would notice that something else is answering at your server's address.

The deploy job runs in a fresh container with an empty `known_hosts`, and nobody is there to answer `yes`. `-o StrictHostKeyChecking=no` accepts whatever key the server presents, once, and goes on. The log shows it:

```
Warning: Permanently added '193.40.157.28' (ED25519) to the list of known hosts.
```

What is given up: the job would not notice an impostor at that IP. For a student server reached from a university runner that is an acceptable trade. The stricter version is to store the server's public host key in a variable and write it into `known_hosts` in `before_script`; the deck's version is the one to use today.

The Compose file the server gets

Lab 2's `compose.yaml` had `build: .`, because the image was built on the server. Now the image comes from Docker Hub:

```
services:
  app:
    image: ${DOCKER_IMAGE}:latest
    container_name: app
    ports:
      - "8080:8080"
    restart: unless-stopped
```

`docker compose pull` looks at `image:` and fetches it. `docker compose up -d` sees that the image changed and replaces the container. Real deploy log:

```
 app Pulled
 Container app  Creating
 Container app  Created
 Container app  Starting
 Container app  Started
Job succeeded
```

Compose and the `.env` file

`compose.yaml` on the server says `image: ${DOCKER_IMAGE}:latest`, and nothing in that file says what `DOCKER_IMAGE` is. Compose fills `${…}` from two places, in this order: the environment of the shell that runs `docker compose`, and a file named `.env` in the same directory.

The deploy job writes that file before every `pull`:

```
$ ssh ubuntu@193.40.157.28 "cd /opt/app && echo DOCKER_IMAGE=supax/iti0302-ci-test > .env && …"
```

On the server, `cat /opt/app/.env` shows `DOCKER_IMAGE=supax/iti0302-ci-test`, and `docker compose config` prints the Compose file with every `${…}` replaced, which is the fastest way to see what Compose will actually run.

Why not write the image name into `compose.yaml` directly? Because the file is in git and the same for everyone who clones it; the name is yours. The same mechanism carries any value that differs per server, a password included: the value in `.env` on the server, the placeholder in git.

Secrets: what the pipeline must know and git must not contain

The three stages need:

Value	Used by	Secret?
Docker Hub username	dockerize	no
Docker Hub access token	dockerize	yes
image name `account/repo`	dockerize, deploy	no
server IP	deploy	no
private key that opens the server	deploy	yes

Nothing secret goes into `.gitlab-ci.yml` or any other file in git. The repository is readable by every member and by anyone it is ever shared with, and git never forgets: a password committed once is in the history for good, even after it is removed.

A password committed once is committed forever

Git keeps every version of every file. If a token is committed, then removed in the next commit, it is still in the history, readable with `git log -p` by anyone who has the repository, and visible in GitLab's file history for good. Removing it from the current version changes nothing about that.

The only fix is to make the leaked value worthless: revoke the token on Docker Hub, remove the deploy key's line from `authorized_keys` on the server, and issue new ones. Then keep the new ones out of git.

Three habits that prevent it:

- values the pipeline needs go into CI/CD variables, never into `.gitlab-ci.yml`
- values the application needs on the server go into `.env` or `application-local.properties` on the server, both listed in `.gitignore`
- before every commit, `git status` and `git diff --cached`: read what is about to be committed

CI/CD variables

GitLab keeps values outside the repository and injects them into every job as environment variables: Settings → CI/CD → Variables → Add variable.

Key	Value	Flags
`DOCKER_USERNAME`	your Docker Hub username	
`DOCKER_PASSWORD`	your Docker Hub access token	**Masked**
`DOCKER_IMAGE`	`username/iti0302-2026`	
`SERVER_IP`	your server's IP	
`SSH_PRIVATE_KEY`	the private deploy key, whole file	

In the job, `$DOCKER_PASSWORD` is the value, the same way `$HOME` is a value in any shell. The file in git names the variable; only the running job sees what is in it.

Variable flags: masked, protected, expanded

Each variable has three checkboxes. What they do:

Flag	Meaning	For assignment 1
Mask variable	the value is replaced by `[MASKED]` wherever it would appear in a job log	on for `DOCKER_PASSWORD`. A multi-line key cannot be masked; never print it
Protect variable	the variable exists only in pipelines on protected branches	off for all five, or the variable is missing on feature branches and `dockerize` fails with a login error
Expand variable reference	`$name` inside the value is replaced by another variable	off for `SSH_PRIVATE_KEY`; a key can contain `$` characters that would be mangled

Docker Hub: an access token, not your password

The pipeline logs in to Docker Hub. It must not use your account password: it would sit in a variable readable by every Maintainer of the project, and it opens everything, including deleting your repositories.

Docker Hub issues **personal access tokens**: app.docker.com, signed in, → **Personal access tokens** in the left menu → Generate new token.

- Permissions: **Read & Write** (push needs write)
- Repository access: all, or the one repository
- Expiry: the end of the semester

The token is shown once. Paste it into the `DOCKER_PASSWORD` variable. It can be revoked any time without changing your password, and it is what `docker login` in the job uses:

```
$ echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
Login Succeeded
```

How key-based login works

SSH lets you in without a password because of a key pair: a **private key**, a file on your laptop that never leaves it, and a **public key**, derived from it, which you give to every server you want to log in to. The server keeps public keys in `~/.ssh/authorized_keys`, one per line.

At login the server sends a challenge that only the holder of the private key can answer; the answer proves possession without the private key crossing the wire. So:

- anyone with the private key can log in as `ubuntu` on every server that has the public half
- a public key is harmless to publish; a private key is a password that opens every one of those servers

A line in `authorized_keys` is a permission. Adding one grants a login; deleting it revokes it, immediately. That is the mechanism the pipeline's access is built on.

A deploy key: a second key pair, for the pipeline only

The deploy job logs in to your server over SSH. It needs a private key, and it must not be your own: yours is on your laptop and it opens everything you have access to. Make a pair whose only purpose is this:

```
ssh-keygen -t ed25519 -C "gitlab-ci-deploy" -f ~/.ssh/iti0302-deploy -N ""
```

- `-N ""`: no passphrase. A job cannot type one
- `-C`: a comment, so that the key is recognisable in `authorized_keys` later
- two files: `iti0302-deploy` (private) and `iti0302-deploy.pub` (public)

Where each half of the deploy key goes

File	Goes to	How
`iti0302-deploy.pub`	your server, `~/.ssh/authorized_keys`, as a new line	`cat` it on the laptop, paste it into the file with `nano` on the server
`iti0302-deploy`	GitLab variable `SSH_PRIVATE_KEY`	`cat ~/.ssh/iti0302-deploy`, copy the whole thing including the BEGIN and END lines

The server now accepts two keys: yours, and the pipeline's. If the pipeline's leaks, remove its line from `authorized_keys` and make a new pair. Your own key is untouched.

The application must say which commit it is

Assignment 1 requires:

```
GET /api/version  →  { "commit": "93c82ab8" }
```

The commit the running application was built from. The check after the deadline pushes a commit to your repository, waits, and expects this value to change. That proves the pipeline builds **and deploys**, on its own, from a push.

The value has to travel from GitLab, which knows the commit, to a Java method on the server. Four hand-overs, one line each:

1. GitLab → the `dockerize` job, as `$CI_COMMIT_SHORT_SHA`
2. the job → the image, as a build argument
3. the image → the process, as an environment variable
4. the environment → Spring, as a property the controller reads

Hand-over 1 and 2: pipeline → Dockerfile

In `.gitlab-ci.yml`, `docker build --build-arg COMMIT=$CI_COMMIT_SHORT_SHA …` hands the commit to the build as a **build argument**. In the `Dockerfile`, two lines after `FROM` receive it and bake it into the image:

```
FROM eclipse-temurin:25-jre-alpine
ARG COMMIT=local
ENV APP_COMMIT=$COMMIT
RUN addgroup -S app && adduser -S app -G app
WORKDIR /app
COPY app.jar app.jar
RUN chown app:app app.jar
USER app
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "app.jar"]
```

`ARG COMMIT=local`: a value that exists during the build, with a default for a build by hand. `ENV APP_COMMIT=$COMMIT`: copies it into an environment variable that every process in the container sees.

Hand-over 3 and 4: environment → Spring → JSON

`application.properties`:

```
app.commit=${APP_COMMIT:local}
```

Spring reads `APP_COMMIT` from the environment; `:local` is the fallback when it is absent, as in IntelliJ or a jar started by hand. The controller:

```
@RestController
public class VersionController {

    @Value("${app.commit}")
    private String commit;

    @GetMapping("/api/version")
    public Map<String, String> version() {
        return Map.of("commit", commit);
    }
}
```

`@Value` injects the property; a `Map` becomes the JSON object `{"commit":"…"}`.

The chain, checked at both ends

Locally, in IntelliJ or with `java -jar app.jar`:

```
$ curl http://localhost:8080/api/version
{"commit":"local"}
```

On the server, after the pipeline deployed commit `93c82ab8`:

```
$ curl http://193.40.157.28/api/version
{"commit":"93c82ab8"}
```

And after one more push, two minutes later:

```
$ curl http://193.40.157.28/api/version
{"commit":"11d55df0"}
```

That second change is what the assignment check looks for. If the value does not change within 15 minutes of the check's push, the pipeline did not deploy, and the submission fails.

The pipeline list is the deployment history

Build → Pipelines lists every pipeline the project ever ran. The test project's list at the time of writing:

Pipeline	Status	Branch	Commit	Created
158748	success	main	d5c46e1f	2026-09-16 22:36
158747	success	pipeline-cache-tests	5b49899c	2026-09-16 22:34
158742	failed	broken-build	bf8577c9	2026-09-16 22:02
158741	success	main	11d55df0	2026-09-16 21:57
158739	success	main	93c82ab8	2026-09-16 21:50

Read from the top: the last green pipeline on `main` is what the server runs, and its commit is what `/api/version` reports. The red one on a branch never reached the server. Who pushed, when, and every job's log stay available: "what changed, and when" without asking anyone. For assignment 1 the pipeline of the latest `main` commit must be green at check time.

Branches and the pipeline

A pipeline runs for every push, on every branch. With `only: - main` on `dockerize` and `deploy`:

Push to	build	dockerize	deploy
a feature branch	runs	not created	not created: the pipeline has one job
`main`	runs	runs	runs

So a feature branch proves that the code compiles and the tests pass, and nothing else: no image is pushed, the server is not touched. Merging it into `main` triggers the pipeline on `main`, which builds again, pushes the image and deploys it.

Checked on the test project: a push to a branch created a pipeline with only `build`; the merge into `main` created one with all three jobs, and the server reported the merge commit two minutes later.

What a merge request does with the pipeline

A merge request (MR) is GitLab's page for "take this branch into `main`": the diff, a discussion, and the pipeline status of the branch's latest commit. Done on the test project, from a branch named `pipeline-cache-tests`:

1. push the branch; GitLab offers "Create merge request" for it
2. the MR page shows the branch's pipeline: one job, `build`, green after about half a minute
3. Merge. GitLab creates a **merge commit** on `main`, `d5c46e1f`, whose parents are the old `main` and the branch
4. the push to `main` that the merge is starts a new pipeline, with all three jobs; `deploy` puts `d5c46e1f` on the server

After the merge

```
$ curl http://193.40.157.28/api/version
{"commit":"d5c46e1f"}
```

The value is the merge commit, not the branch's last commit: what runs is exactly what `main` points at, and `main` points at the merge.

Two things on the MR page worth knowing:

- while the branch's pipeline is still running, the Merge button reads **Set to auto-merge** (GitLab 19 on gitlab.cs.taltech.ee; older versions call it "Merge when pipeline succeeds"). It merges by itself once the pipeline is green and does nothing if it turns red
- **Delete source branch** is ticked by default. After the merge the branch is gone from GitLab; your local copy stays until you delete it with `git branch -d`

A merged MR stays in the project's list forever, with its discussion and its pipeline: that is the record of why a change was made, next to the pipeline list that records when it went live.

`only` and `rules`: two ways to say the same thing

`only:` is the older syntax, still supported and enough for assignment 1. GitLab's current documentation writes the same condition with `rules:`, which can express more:

```yaml
deploy:
  stage: deploy
  only:
    - main
```

```yaml
deploy:
  stage: deploy
  rules:
    - if: $CI_COMMIT_BRANCH == "main"
```

`rules` is a list, checked top to bottom; the first match decides whether the job is created. Without any match the job is not created, which is what `only` does when the branch does not match. Either form; do not combine both in one job, GitLab rejects that.

Merge requests and the green-pipeline rule

From this week, the standing rule of the course:

- all work happens on **feature branches**; nobody pushes to `main` directly
- a branch reaches `main` through a **merge request (MR)**, in GitLab
- an MR is merged only when its **pipeline is green**
- in a team of two or three, a teammate reviews the MR before merging. Solo work still uses branches and MRs

GitLab shows the pipeline status on the MR page and can block the Merge button until it is green: Settings → Merge requests → Merge checks → "Pipelines must succeed". Turn it on in your repository, and leave "Skipped pipelines are considered successful" off; it makes the rule impossible to forget.

Your git history is reviewed by hand: branches, MRs, green pipelines on the MRs, meaningful commits. A history of commits straight to `main` with red pipelines fails that review even if the application works.

Protected branches: making the rule impossible to skip

GitLab protects the default branch of every project from the start. The test project's setting, read from its configuration:

```
main   allowed to push: Maintainers   allowed to merge: Maintainers
```

Developers cannot push to `main` at all; they must open a merge request. You are the Owner of your own repository, so the default does not stop you. To make it stop you too: Settings → Repository → Protected branches → `main` → "Allowed to push and merge": **No one**. From then on every change to `main`, yours included, arrives through a merge request, and the merge button is the only way in.

Together with "Pipelines must succeed" in Settings → Merge requests, the rule of the course is enforced by the tool rather than by memory: nothing reaches `main` without a green pipeline, and nothing deploys that did not reach `main`.

The whole file

```yaml
stages: [build, dockerize, deploy]

build:
  stage: build
  image: maven:3.9-eclipse-temurin-25
  script:
    - ./mvnw -B clean package -DskipTests
    - cp target/*.jar app.jar
  artifacts:
    paths: [app.jar]

dockerize:
  stage: dockerize
  image: docker:latest
  services: [docker:dind]
  variables: { DOCKER_TLS_CERTDIR: "" }
  script:
    - echo "$DOCKER_PASSWORD" | docker login -u "$DOCKER_USERNAME" --password-stdin
    - docker build --build-arg COMMIT=$CI_COMMIT_SHORT_SHA -t $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA -t $DOCKER_IMAGE:latest .
    - docker push $DOCKER_IMAGE:$CI_COMMIT_SHORT_SHA
    - docker push $DOCKER_IMAGE:latest
  only: [main]

deploy:
  stage: deploy
  image: alpine:latest
  before_script:
    - apk add --no-cache openssh-client
    - eval $(ssh-agent -s)
    - echo "$SSH_PRIVATE_KEY" | tr -d '\r' | ssh-add -
    - mkdir -p ~/.ssh && chmod 700 ~/.ssh
  script:
    - scp -o StrictHostKeyChecking=no compose.yaml ubuntu@$SERVER_IP:/opt/app/compose.yaml
    - ssh -o StrictHostKeyChecking=no ubuntu@$SERVER_IP "cd /opt/app && echo DOCKER_IMAGE=$DOCKER_IMAGE > .env && docker compose pull && docker compose up -d"
  only: [main]
```

What the pipeline needs to exist, in one list

In the repository:

- `.gitlab-ci.yml`, `Dockerfile` (with `ARG COMMIT` and `ENV APP_COMMIT`), `compose.yaml` (with `image: ${DOCKER_IMAGE}:latest`)
- `app.commit=${APP_COMMIT:local}` in `application.properties`, and the `/api/version` controller

In GitLab, project settings:

- shared runners enabled
- five CI/CD variables: `DOCKER_USERNAME`, `DOCKER_PASSWORD`, `DOCKER_IMAGE`, `SERVER_IP`, `SSH_PRIVATE_KEY`

What the pipeline needs to exist, outside GitLab

On Docker Hub:

- an account; its username is the first half of `DOCKER_IMAGE`
- an access token with Read & Write permission, stored as `DOCKER_PASSWORD`

On the server:

- Docker installed and the `ubuntu` user in the `docker` group (lab 2)
- the deploy public key as a line in `~/.ssh/authorized_keys`
- `/opt/app`, created once and owned by `ubuntu`
- nginx forwarding `/api` to port 8080 (lab 2)

On your laptop: the deploy key pair, and nothing else the pipeline depends on. That is the point: once these exist, the laptop is only where the code is written.

Reading a red pipeline

The pipeline page lists the jobs; a red one is a link to its log. Find the first line with `ERROR` or `fatal` and read from there.

In the log	Cause
pipeline stays *pending*, no job starts	shared runners not enabled for the project
`This job is stuck because you don't have any active runners`	same
`invalid target release: 25`	the build image's Java is older than `java.version` in the project; fix the image tag
`denied: requested access to the resource is denied` on push	`DOCKER_IMAGE` does not start with your Docker Hub username, or the token has no write permission
`unauthorized: incorrect username or password`	wrong `DOCKER_USERNAME`, or the token was pasted with a trailing space

Reading a red deploy job

In the log	Cause
`Permission denied (publickey)`	the public deploy key is not in the server's `authorized_keys`, or the private key was pasted incomplete
`Load key "/dev/fd/63": error in libcrypto`	the private key variable is missing lines, or "Expand variable reference" is on
`permission denied while trying to connect to the Docker daemon socket`	`ubuntu` is not in the `docker` group on the server
`ssh: connect to host … port 22: Connection timed out`	`SERVER_IP` is wrong, or the server is down
`scp: /opt/app/compose.yaml: Permission denied`	`/opt/app` does not exist or is not owned by `ubuntu`
`docker: not found` in dockerize	the job's image is not `docker:latest`

Common mistakes

Symptom	Cause
Green pipeline, but `/api/version` still shows the old commit	`deploy` did not run: the push was not to `main`, or `only:` names another branch
`/api/version` says `local` on the server	the Dockerfile lacks `ARG COMMIT` / `ENV APP_COMMIT`, or `--build-arg` is missing from the build line
An image on Docker Hub for every branch, none deployed	`only: main` is missing from `dockerize`; branches are not supposed to push
The job log shows the token	it was not marked Masked; revoke it on Docker Hub and make a new one
`.gitlab-ci.yml` is ignored	wrong name or not in the repository root; GitLab shows "CI/CD configuration not found"
YAML error on the pipeline page	tabs, or a `:` inside a script line without quotes around the line
Pipeline green, container `Exited (1)` on the server	the jar starts and dies; `docker compose logs` on the server says why

Lab 3

- A `.gitlab-ci.yml` with the three stages in **your** assignment 1 repository
- A Docker Hub account and access token; the five CI/CD variables in your project; shared runners enabled
- A deploy key pair: public half on your server, private half in GitLab
- `/api/version` wired through: build argument, environment variable, property, controller
- Push to a branch, open a merge request, watch the pipeline; merge, watch it deploy
- `curl http://<ip>/api/version` shows the merged commit's SHA

By the end, a push to `main` changes what runs on your server, with no step done by hand. That is what assignment 1 is checked against.



Questions