DevOps Best Practices to Build CI/CD Pipelines, Monitoring, and Logging

3 days

YOUR TEACHER

Jerome Steunenberg

Kubernetes & DevOps architect, consultant and trainer

Full-stack developer

in **https://www.linkedin.com/in/jeromesteunenberg/**

docker

kubernetes

GitLab

Symfony

HELM

ANSIBLE

CI / CD TRAINING

Overview

This hands-on course introduces participants to modern DevOps practices, focusing on building efficient and secure CI/CD pipelines, implementing robust monitoring, and leveraging logging tools to optimize application delivery. Designed for an audience eager to enhance their DevOps expertise, the training uses GitLab as the technical environment. With a mix of theory, demonstrations, and interactive labs, attendees will gain practical insights into creating scalable, reliable workflows aligned with industry best practices.

Highlights

- Learn to design and deploy robust CI/CD pipelines with GitLab.
- Explore best practices in monitoring and logging for enhanced visibility and performance.
- Introduce security as a core element of pipelines, covering best practices and practical tools.
- Gain hands-on experience through guided labs and real-world scenarios (in pairs).
- Build a foundation for implementing DevOps strategies in diverse environments.

CI / CD TRAINING

Day 1

Course contents

✓ High-level presentation on DevOps concepts: "Exploring the DevOps Toolbox: Tools for Scalability and Deployment"

✓ Foundations of CI/CD Pipelines

- Introduction to CI/CD
- GitLab overview and pipeline architecture
- Designing efficient pipelines and managing stages, jobs, and runners
- Introduction to securing CI/CD pipelines

✓ Hands-On Lab (paired)

- Create and deploy a simple CI/CD pipeline in GitLab
- Automate a sample application build, test and deploy
- Secure the pipeline using GitLab's built-in secret management and RBAC features

CI / CD TRAINING

Day 2

Course contents

✓ High-level presentation on DevOps concepts: "GitOps, Branching Strategies, and Repository Design for Scale"

✓ Monitoring and Observability

- Monitoring principles and the "Three Pillars of Observability" (metrics, logs, and traces)
- Setting up Prometheus for pipeline and system metrics collection
- Visualizing metrics with Grafana dashboards
- Creating and analyzing dashboards for system health
- Incident response workflows
- Security logging essentials

✓ Hands-On Lab (paired)

- Use Prometheus and Grafana with GitLab to monitor pipeline and application performance
- Build dashboards to monitor CI/CD application performance
- Configure security logging to detect and analyze failed login attempts and unauthorized changes

CI / CD TRAINING

Day 3

Course contents

✓ High-level presentation on DevOps concepts: "Smart Deployments: Canary Releases, A/B Testing, and Rollbacks"

✓ Logging and Advanced Practices

- Centralized Logging for CI/CD Pipelines and applications
- Enhancing CI/CD Pipelines
- Security Testing in Pipelines
- Incident response workflows and case studies

✓ Hands-On Lab (paired)

- Configure centralized logging for GitLab pipelines
- Simulate and troubleshoot common CI/CD pipeline issues
- Integrate GitLab SAST and DAST scans to identify and fix vulnerabilities in the code

Exploring the DevOps Toolbox

Tools for Scalability and Deployment

DAY 1

EXPLORING THE DEVOPS TOOLBOX

Tools for Scalability, and Deployment

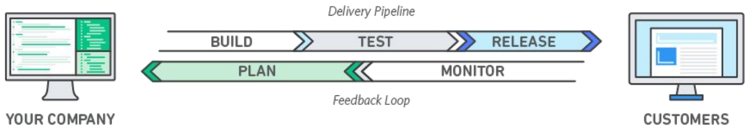
Objective
Provide a high-level overview of essential DevOps tools, their purposes, and how they fit into the CI/CD and operations ecosystem.

Content Outline

- **What is the DevOps Ecosystem?**
    - **A brief history of DevOps and the role of automation in modern workflows.**
    - **Key categories of DevOps tools: Infrastructure, CI/CD, Observability, and Incident Management.**
- **Popular Tools Developers Should Know:**
    - **Infrastructure as Code (IaC): Terraform, Ansible.**
    - **Containerization and Orchestration: Docker, Kubernetes, Helm.**
    - **Deployment Automation: ArgoCD (GitOps for Kubernetes).**
    - **Monitoring and Alerts: Prometheus, Grafana.**

EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

What is DevOps ?



DevOps is the combination of cultural philosophies, practices, and tools that increase an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes. This speed enables organizations to better serve their customers and compete more effectively in the market.

How DevOps Works

Benefits of DevOps

- **Speed: Rapid innovation and adaptation.**
- **Rapid Delivery: Quick releases and bug fixes.**
- **Reliability: Maintain quality and performance at high velocity.**
- **Scale: Manage complex systems efficiently.**
- **Improved Collaboration: Enhanced teamwork across teams.**
- **Security: Retain control and compliance while moving fast.**

Why DevOps Matters

EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

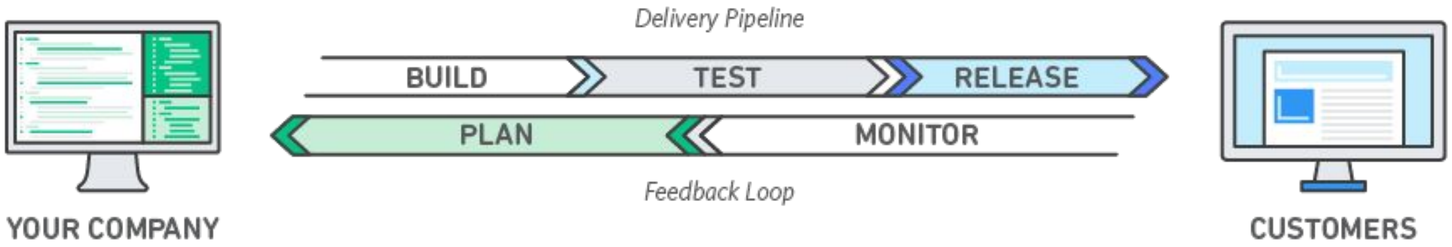
What is DevOps ?

How to Adopt a DevOps Model

- Think declaratively, not imperatively
- Silos become collaboration
- Think proactively -> things are going to break
- The 12-factor app

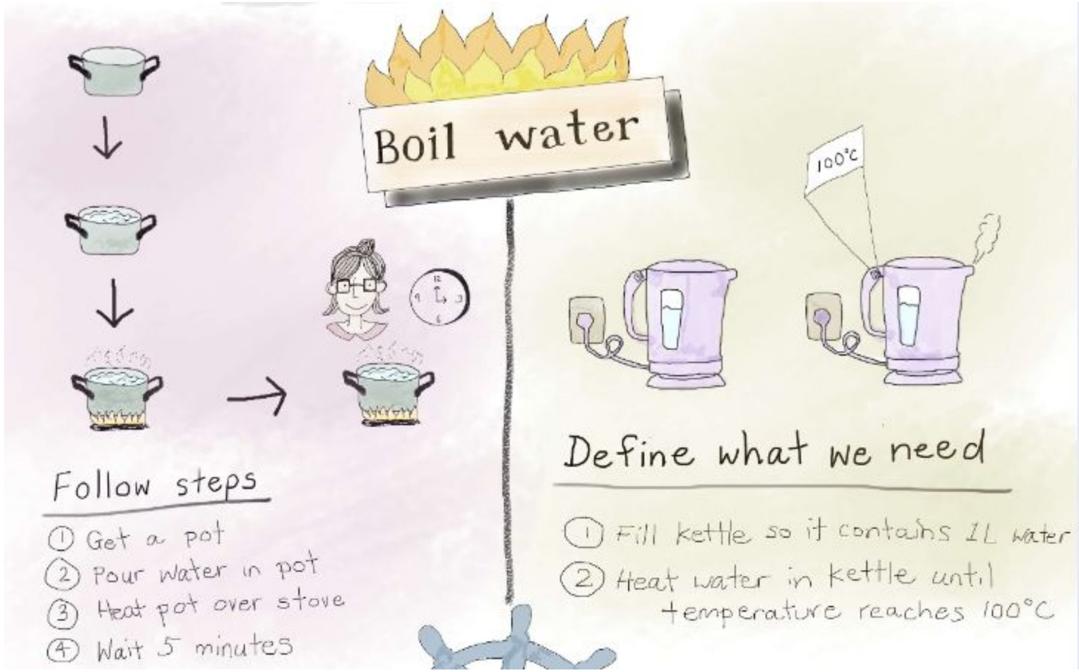
DevOps Practices

- Continuous Integration
- Continuous Delivery
- Microservices
- Infrastructure as Code
- Monitoring and Logging
- Communication and Collaboration
- DevOps Tools



EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

Imperative vs Declarative



EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

History of DevOps

Pre-DevOps Era (Before 2007)

- **2001 – Agile Manifesto is published. Emphasized collaboration, iterative development, and responding to change—principles that later influenced DevOps.**
- **2003 – Google introduces Site Reliability Engineering (SRE). SRE becomes a model for blending software engineering with operations, an early DevOps-like approach.**
- **2004 – First mention of “Infrastructure as Code” concepts with tools like Cfengine and later Puppet (2005) and Chef (2009).**

The Emergence of DevOps (2007–2012)

- **2007 – Velocity Conference (O’Reilly) begins. Where development and operations collaboration problems are highlighted—especially through talks by Flickr engineers (John Allspaw & Paul Hammond).**
- **2008 – “10 Deploys a Day” Talk at Velocity. John Allspaw and Paul Hammond describe how dev and ops can work together for fast, reliable deployments—considered a seminal DevOps moment.**
- **2009 – The term “DevOps” is coined. By Patrick Debois during the setup of the first DevOpsDays event in Ghent, Belgium.**
- **2009 – First DevOpsDays conference in Ghent, Belgium. Marks the formal beginning of the DevOps movement.**

EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

History of DevOps

DevOps Grows and Matures (2013–2017)

- 2011 – CAMS model proposed (Culture, Automation, Measurement, Sharing) by John Willis and Damon Edwards.
- 2013 – The Phoenix Project is published (Gene Kim, Kevin Behr, George Spafford). A fictional business novel that popularized DevOps principles and spread awareness beyond technical circles.
- 2014 – Docker explodes in popularity, bringing containerization to the mainstream and enabling better DevOps workflows.
- 2015 – The DevOps Handbook is released (Gene Kim, Jez Humble, Patrick Debois, John Willis). A detailed guide that formalizes and expands on DevOps practices.
- 2016 – Google publishes the SRE Book. Further reinforces the relationship between DevOps and SRE practices.

Enterprise Adoption & Tooling Explosion (2018–2022)

- 2018 – Kubernetes becomes the de facto container orchestration platform, backed by CNCF and cloud vendors.
- 2019 – GitOps term popularized by Weaveworks. Emphasizes using Git as the single source of truth for infrastructure and application delivery.
- 2020 – DevSecOps gains traction. Security is integrated as a core component of the DevOps pipeline.
- 2021 – Platform engineering emerges as a refined way to build internal developer platforms (IDPs) to support DevOps at scale.

Recent and Ongoing Developments (2023–2025)

- 2023 – Increased adoption of AI and ML in DevOps (AIOps). Automation of monitoring, anomaly detection, and incident response.
- 2024–2025 – DevOps increasingly merges with platform engineering, observability-first development, and developer experience (DevEx) paradigms.

EXPLORING THE DEVOPS TOOLBOX - TOOLS FOR SCALABILITY, DEPLOYMENT, AND INCIDENT MANAGEMENT

Key Categories of DevOps Tools

git

Infrastructure as Code (IaC): Automating infrastructure provisioning and management.

ANSIBLE HashiCorp Terraform

CI/CD (Continuous Integration/Continuous Deployment): Ensuring efficient and reliable software releases.

docker HELM argo GitLab

Observability and Monitoring: Providing visibility into system health and performance.

Prometheus Grafana elasticsearch

Incident Management: Streamlining responses to production issues and ensuring uptime.

Foundations of CI/CD Pipelines

DAY 1

Foundations of CI/CD Pipelines

DAY 1

Foundations of CI/CD Pipelines

Key Topics:

- **Introduction to CI/CD. Benefits of CI/CD and its role in software delivery.**
- **GitLab overview and pipeline architecture.**
- **Designing efficient pipelines with stages, jobs, and runners.**
- **Managing build stages, jobs, and runners.**
- **Introduction to securing CI/CD pipelines:**
    - **Managing secrets and environment variables securely (e.g., GitLab CI/CD secrets).**
    - **Implementing access controls and audit logs for GitLab projects (RBAC).**

Hands-On Lab (paired):

- **Create and deploy a simple CI/CD pipeline in GitLab.**
- **Automate a sample application build and test.**
- **Secure the pipeline using GitLab's built-in secret management and RBAC features.**

Foundations of CI/CD Pipelines

Introduction to CI/CD

What is CI/CD?

Key Benefits of CI/CD

- **Faster development cycles**
- **Improved code quality**
- **Reduced manual effort**
- **Rapid feedback**
- **Lower risk deployments**

CI/CD's Role in Modern Software Delivery

DevOps Best Practices to Build CI/CD Pipelines, Monitoring, and Logging

Training environment

All VMs are Ubuntu 24.04 LTS hosted in Exoscale Geneva, Switzerland. Please make sure you assign one VM to each student.

You will retain the same VM throughout the whole course. You can copy your public key over to the server to ease login.

Studentx	Hostname	IP	User	Password
student1	student1.creativemoods.pt	159.100.244.89	ubuntu	v4TR3R^aKt*#94
student2	student2.creativemoods.pt	91.92.201.68	ubuntu	Mdg&NhaoKk8uSx
student3	student3.creativemoods.pt	159.100.241.129	ubuntu	!xXJVUKo@i99c$
student4	student4.creativemoods.pt	159.100.240.35	ubuntu	t9dBhM@xq&p&hK
student5	student5.creativemoods.pt	159.100.240.205	ubuntu	x*iyzH3KTovQom
student6	student6.creativemoods.pt	194.182.162.200	ubuntu	d!b46@gxHJP#LY
student7	student7.creativemoods.pt	91.92.153.178	ubuntu	Z5Yf%iQ6rt6DJ%
student8	student8.creativemoods.pt	91.92.152.186	ubuntu	E$iUEx%BJY4dN$
student9	student9.creativemoods.pt	194.182.161.163	ubuntu	kc2KmLC2Vx*Q3@
student10	student10.creativemoods.pt	185.19.30.71	ubuntu	2s2$Wqd3&EBQy6

Setting up Gitlab

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Log into your student machine using the provided password:**

```
$ ssh ubuntu@studentx.creativemoods.pt
```

2. **Your kubeconfig file (credentials for accessing the Kubernetes cluster) has been uploaded to the file ~/kubeconfig-studentx.yaml. You will need this later when creating Gitlab CI/CD variables:**

```
$ cat ~/kubeconfig-studentx.yaml
```

3. **Log into gitlab.com with your credentials (studentx@creativemoods.pt or cmstudentx and provided password)**
4. **Add your SSH key to your Gitlab profile**
    a. **Your SSH public key can be found in the file ~/.ssh/id_rsa.pub**
    b. **Your can add SSH keys to your Gitlab profile by going to User Settings > Preferences > SSH Keys > Add new key**
5. **Configure your local git client:**

```
$ git config --global user.name "Studentx"
$ git config --global user.email "studentx@creativemoods.pt"
$ git config -l
```

Foundations of CI/CD Pipelines

Ansible & Terraform demo

Before going into CI/CD, let's have a look at the Ansible and Terraform configurations for the management of these 10 VMs.

Terraform is used with the Exoscale provider to provision 10 VMs with certain properties.

Ansible is then used to configure the VMs.

- Terraform is used for the outside of the VM (VMs, networks, storage, etc.)
- Ansible is used for the inside of the VM (hostname, password, NTP, packages, etc.)
- Infrastructure is also managed as code
- The code is in git, which is the source of truth
- Infrastructure is defined declaratively, not via scripts
- Everything is idempotent. If you execute the Terraform plan or the Ansible playbooks against infrastructure that is the way it's supposed to be, nothing happens. Otherwise, small steps are taken to realign the infrastructure with what has been described in the manifests.

Creating our app (lab1)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **In your home directory, clone the devops-ci-cd-training-lab1 repository:**

```
$ git clone git@gitlab.com:cmtraining1/devops-ci-cd-training-lab1.git
$ cd devops-ci-cd-training-lab1/
$ sudo apt install python3-flask python3-flask-cors
$ python3 backend.py
```

2. **Open a new terminal and shell and issue a few curl requests to verify that the backend works fine:**

```
$ curl -XPOST -H "Content-type: application/json" http://localhost:5000/api/tasks -d
'{"task":"mytask"}'
$ curl http://localhost:5000/api/tasks
$ curl -XDELETE -H "Content-type: application/json" http://localhost:5000/api/tasks -d
'{"task":"mytask"}'
$ curl http://localhost:5000/api/tasks
```

3. **Kill the server (Ctrl-C)**

Adding unit tests (lab2)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/devops-ci-cd-training-lab2.git**
2. **Cd into devops-ci-cd-training-lab2 and inspect the tests directory.**
3. **Run the unit tests with the command:**

```
$ python3 -m unittest discover tests/
```

4. **Try to introduce a bug by uncommenting the following line and relaunching the tests**

```
#tasks = [{"task": "Initial task"}]
```

5. **This should break the tests because they start with an already existing task.**
6. **Fix the error**

Adding a frontend (lab3)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. Clone repository git@gitlab.com:cmtraining1/devops-ci-cd-training-lab3.git
2. Cd into lab3 and inspect the contents (especially frontend/src/App.js).
3. In one terminal, start the backend:

```
$ cd backend
$ python3 backend.py
```

4. In another terminal, start the frontend:

```
$ sudo apt install npm -y
$ cd frontend
$ npm i
$ npm start
```

5. You should now be able to test the app at the URL http://studentx.creativemoods.pt:3000 (use an incognito window if you get stuck with an http -> https redirection)
6. The XHR requests don’t work. What could be the cause ?
7. Fix this by making sure to start the frontend with the appropriate API URL environment variable:

```
$ REACT_APP_API_URL=http://studentx.creativemoods.pt:5000/api npm start
```

8. This should now work (maybe you need to refresh the React client with the Shift-F5 key) and we have a working app that we are going to use throughout our course

Foundations of CI/CD Pipelines

GitLab Overview and Pipeline Architecture

GitLab Overview

- Complete DevOps platform
- Supports SCM, CI/CD, issue tracking, & more
- Single application for the entire SDLC

Pipeline Architecture

- .gitlab-ci.yml defines pipeline structure
- Jobs → Stages → Pipeline
- Runners execute jobs
- Supports parallel and sequential execution

Creating our first pipeline (lab4)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab4.git**
2. **Inspect the new file .gitlab-ci.yml. It contains a minimal pipeline with a single stage and a single job that launches the unit tests of the backend**
3. **Trigger the pipeline:**

```
$ git commit --allow-empty -m "Trigger pipeline" && git push origin main
```

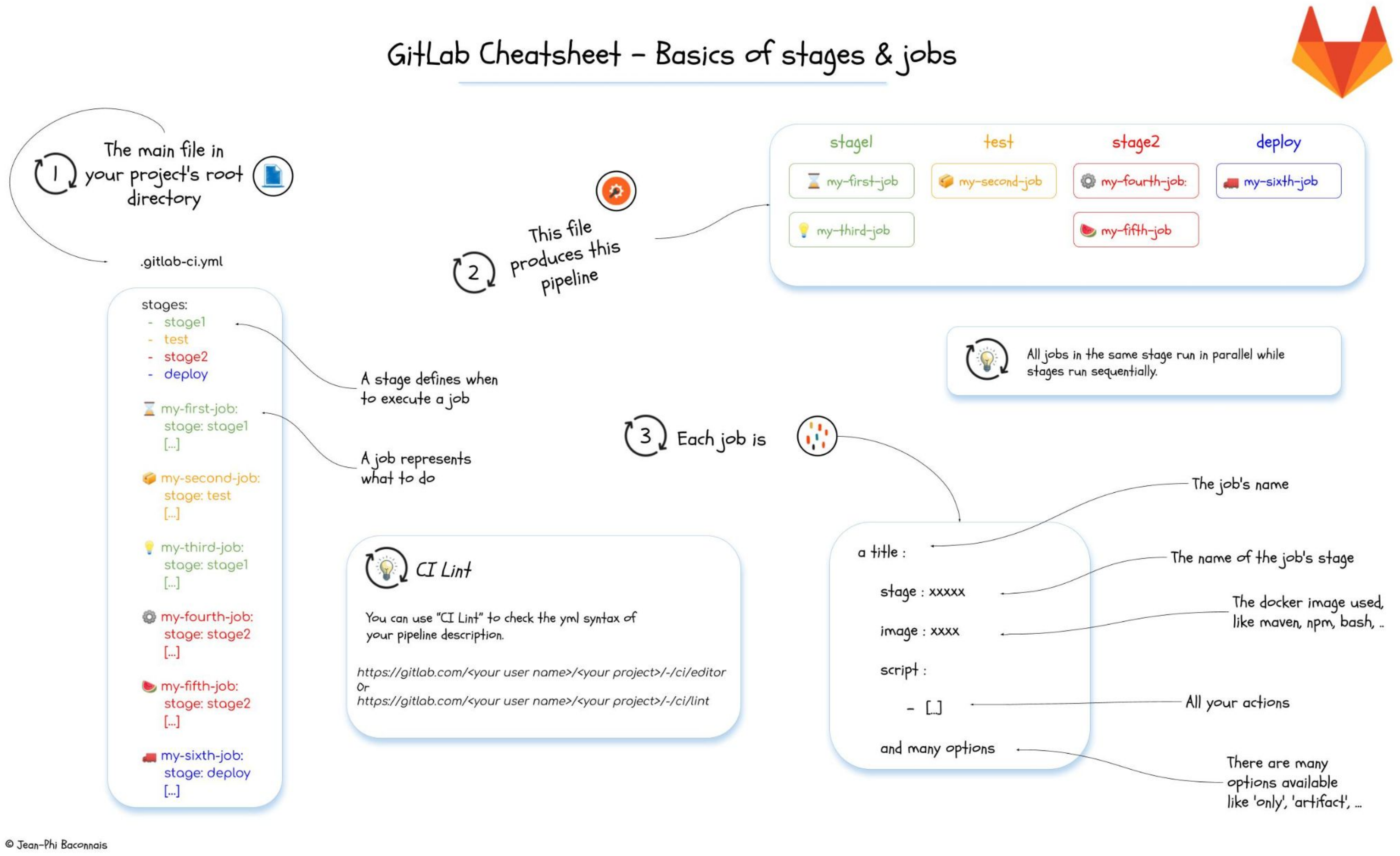
4. **Go to Build > Pipelines. Your pipeline should be stuck because the project doesn't have any runners assigned to it. Cancel the pipeline.**
5. **We need to add a tag to the job so that it knows which Gitlab runner to use:**

```
test_backend:
  stage: test
  image: python:3
  script:
    - cd backend
    - echo "Running unit tests"
    - pip install -r requirements.txt
    - python -m unittest discover tests/
  tags:
    - labrunner
```

6. **Commit the changes and run the pipeline again. It should succeed. Inspect the output of the job.**

```
$ git add .gitlab-ci.yml
$ git commit -m “Fixed missing runner tag”
$ git push origin main
```

Stages and jobs



GitLab Cheatsheet – Basics of stages & jobs

1 The main file in your project's root directory

.gitlab-ci.yml

```
stages:
  - stage1
  - test
  - stage2
  - deploy

⌛ my-first-job:
  stage: stage1
  [...]

📦 my-second-job:
  stage: test
  [...]

💡 my-third-job:
  stage: stage1
  [...]

⚙ my-fourth-job:
  stage: stage2
  [...]

🍉 my-fifth-job:
  stage: stage2
  [...]

🚚 my-sixth-job:
  stage: deploy
  [...]
```

A stage defines when to execute a job

A job represents what to do

2 This file produces this pipeline

All jobs in the same stage run in parallel while stages run sequentially.

3 Each job is

```
a title :
  stage : xxxxx
  image : xxxx
  script :
    - [...]
  and many options
```

The job's name

The name of the job's stage

The docker image used, like maven, npm, bash, ..

All your actions

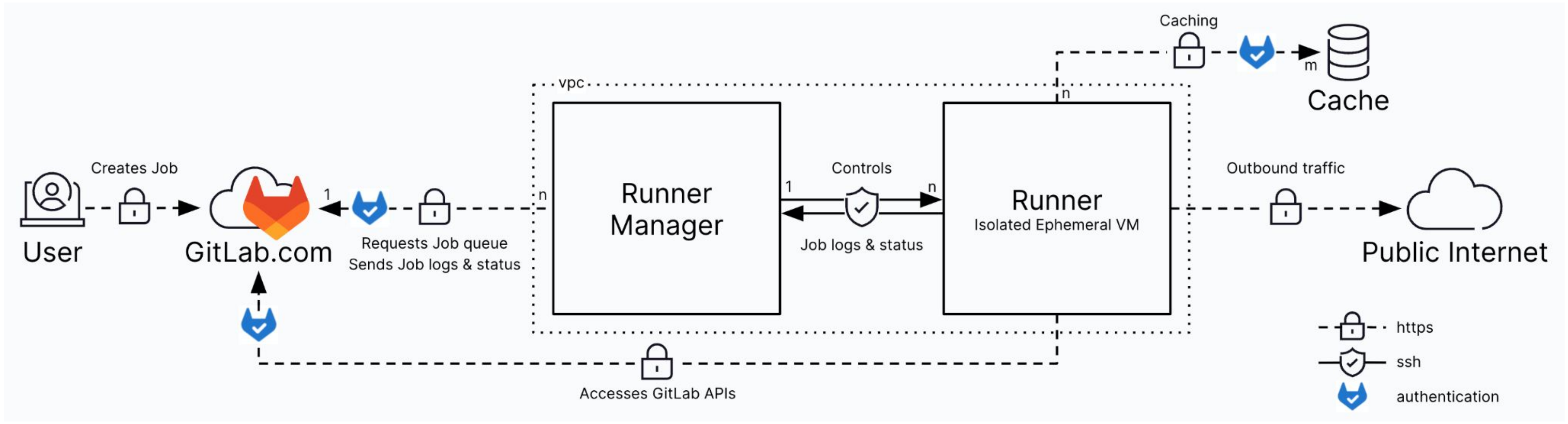
There are many options available like 'only', 'artifact', ...

CI Lint

You can use "CI Lint" to check the yml syntax of your pipeline description.

https://gitlab.com/<your user name>/<your project>/-/ci/editor
Or
https://gitlab.com/<your user name>/<your project>/-/ci/lint

Runners


Caching
m
Cache
vpc
n
Creates Job
User
1
GitLab.com
Requests Job queue
Sends Job logs & status
n
Runner
Manager
Controls
1
n
Job logs & status
Runner
Isolated Ephemeral VM
Outbound traffic
Public Internet
Accesses GitLab APIs
https
ssh
authentication

Foundations of CI/CD Pipelines

Runners

GitLab Runners can run in different executor modes, that define how jobs are run. The two most common are:

- Shell Runner (e.g., Linux Shell)
    - Runs jobs directly on the host system, such as a Linux VM.
    - Uses the host's shell (like bash) to execute CI/CD scripts.
    - No containerization; jobs run with access to the host system.
    - Ideal for:
        - Simple setups
        - Full control over the environment
        - Access to local system resources
- Docker Runner
    - Runs jobs inside Docker containers.
    - Each job runs in a clean container based on the specified image.
    - Better isolation, reproducibility, and consistency.
    - Ideal for:
        - Environments that need consistency across runs
        - Using different dependencies per job
        - CI/CD jobs that need clean environments

Others worth knowing:

- Docker Machine: Similar to Docker, but spins up cloud VMs dynamically.
- Kubernetes: Runs jobs as pods in a Kubernetes cluster. <- the one we're using now
- Custom: Allows for a custom executor script if none of the built-ins work for your use case.
- SSH: Executes jobs via SSH on remote machines.

Foundations of CI/CD Pipelines

Runners

📝 **Steps to Register a GitLab Runner**

- **Install GitLab Runner: https://docs.gitlab.com/runner/install**
- **Register the Runner**

```
gitlab-runner register
```

- **Provide Required Info:**
  - **GitLab URL**
  - **Registration token (from your GitLab project)**
  - **Runner description**
  - **Tags (optional)**
  - **Executor type (e.g., shell, docker)**
- **Start the Runner**

```
gitlab-runner start
```

Adding a build stage (lab5)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab5.git**
2. **Inspect the file .gitlab-ci.yml as well as the backend/docker/Dockerfile and frontend/docker/Dockerfile files. The Dockerfile files instruct docker how to build the containers and the new pipeline now includes a new stage and two new jobs, one for each component.**
3. **Trigger the pipeline:**

```
$ git commit --allow-empty -m "Trigger pipeline" && git push origin main
```

4. **Your pipeline should fail (Build > Pipelines) as it is missing the environment variables it needs. Create the appropriate variables at Settings > CI/CD > Variables:**

Type	Visibility	Protect variable	Expand variable reference	Key	Value
Variable	Visible	No	No	REGISTRY_URL	registry.creativemoods.pt
Variable	Visible	No	No	REGISTRY_USER	studentx (replace x with your student number)
Variable	Masked and hidden	No	No	REGISTRY_PASSWORD	<assigned password for your student number>
Variable	Visible	No	No	API_URL	http://studentx.creativemoods.pt:5000/api (replace x with your student number)

5. **Trigger the pipeline again (Build > Pipelines > Retry all failed or canceled jobs)**
6. **Inspect the outputs of your jobs**

Foundations of CI/CD Pipelines

Gitlab CI/CD variables

🔧 Variable Types

- Environment Variables: Used in .gitlab-ci.yml or GitLab UI.
- File Variables: Store content as files (e.g. certificates, private keys).

🔐 Variable Scope & Protection

- Protected: Only available in protected branches/tags.
- Masked: Hidden in job logs (must match regex rules).
- Masked and hidden: also hidden in the GUI after creation.
- Environment-scoped: Available only for specific environments.

📍 Variable Locations

- Project-level: Most common; set in project settings.
- Group-level: Shared across multiple projects.
- Instance-level: Admins define these globally.
- .gitlab-ci.yml: Inline definitions, less secure.

🧪 Other Options

- Expand Variable References: $VAR_NAME can reference other variables.
- Variable Precedence: YAML-defined < Project < Group < Instance.

Adding a build stage (lab5)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

7. **Log onto the Harbor registry (https://registry.creativemoods.pt/) with the same credentials you used in the pipeline and view the pushed containers**
8. **Run the containers. In one shell:**

```
$ docker login registry.creativemoods.pt
$ docker run --rm --name backend -p 5000:5000 registry.creativemoods.pt/studentx/adesso-backend:latest
```

9. **In another shell:**

```
$ docker run --rm --name frontend -p 3000:3000 registry.creativemoods.pt/studentx/adesso-frontend:latest
```

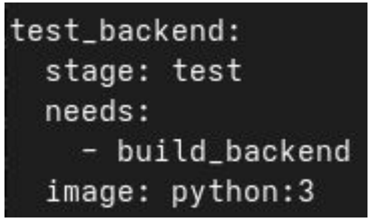
10. **You should now be able to test the app at the URL http://studentx.creativemoods.pt:3000 (use an incognito window if you get stuck with an http -> https redirection)**

Adding job dependency (lab5)

Exercice

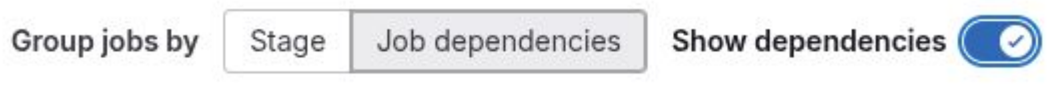
Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

1. The test stage is executed after the build stage. But there is no need for the test job to wait for the frontend build job. Specify a dependency between the test job and the backend build job:



```
test_backend:
  stage: test
  needs:
    - build_backend
  image: python:3
```

2. Commit and push your changes. In Build > Pipelines > latest pipeline:



3. This will show you the dependencies between jobs. You might see that the test job is executed while the frontend is still building.

Refactoring (lab5)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **There is a lot of code duplication between the two build jobs. We can refactor the code using inheritance. Add a new section at the end of the pipeline:**

```
.build_images:
  stage: build
  image: docker:28.1.1
  before_script:
    - docker login -u $REGISTRY_USER -p $REGISTRY_PASSWORD $REGISTRY_URL
  tags:
    - labrunner
```

2. **Replace those lines in the job build_backend:**

```
build_backend:
  extends: .build_images
  script:
    - cd backend
    - echo "Building backend"
    - docker build -t $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest -f docker/Dockerfile
.
    - docker push $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest
```

3. **Replace those lines in the job build_frontend:**

```
build_frontend:
  extends: .build_images
  script:
    - cd frontend
    - echo "Building frontend"
    - docker build --build-arg REACT_APP_API_URL=$API_URL -t
$REGISTRY_URL/$REGISTRY_USER/adesso-frontend:latest -f docker/Dockerfile .
    - docker push $REGISTRY_URL/$REGISTRY_USER/adesso-frontend:latest
```

4. **Commit and push your changes. The pipeline should remain exactly the same**

Monitoring and Observability

Artifacts

- Artifacts are files or build outputs saved between jobs
- Enable reusing compiled code, Docker images, test results, etc.
- Improve performance and consistency by avoiding rebuilds
- Typically stored temporarily (e.g., for 1 hour or until pipeline ends)
- Common use cases:
  - Reusing a built Docker image for testing or deployment
  - Passing compiled binaries to packaging jobs
  - Saving logs, coverage reports, or test outputs
- Artifacts are saved in, and loaded from, the root directory of the working directory

Artifacts (lab5)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Let’s save our built docker image as an artifact in the build_backend job of our pipeline:**

```
build_backend:
  extends: .build_images
  artifacts:
    paths:
      - backend-image.tar
    expire_in: 1 hour
  script:
    - cd backend
    - echo "Building backend"
    - docker build -t $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest -f docker/Dockerfile
.
    - docker push $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest
    - docker save $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest -o ../backend-image.tar
```

Artifacts (lab5)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

2. **Let’s reuse this image in our test job and perform tests inside our docker image:**

```
test_backend:
  stage: test
  needs:
    - build_backend
  image: docker:28.1.1
  script:
    - cd backend
    - echo "Running unit tests"
    - docker load -i backend-image.tar
    - docker image ls
    - docker run --rm $REGISTRY_URL/$REGISTRY_USER/adesso-backend:latest python -m unittest
discover tests/
  tags:
    - labrunner
```

3. **Trigger your pipeline by committing this new code**
4. **There is an error. Can you find it ? Solve and trigger the pipeline again**
5. **Download the artifact from the output of the build job**

Foundations of CI/CD Pipelines

Job dependency

🔄 Job Dependencies Overview

- Default behavior: Jobs run in order of stages (sequential)
- Within a stage: Jobs run in parallel by default.

🔗 Explicit Dependencies

- Use needs: to define direct dependencies between jobs.
- Allows faster pipelines by skipping strict stage sequencing.

📦 Artifacts Passing

- Jobs can pass artifacts to dependent jobs using dependencies: or needs:.

⚠️ Important Notes

- needs: bypasses the stage order — use with care!
- Cycles in needs: are not allowed.

Foundations of CI/CD Pipelines

Caching

🧩 GitLab CI/CD Caching Basics
Cache: Temporary storage to reuse files between jobs or pipelines.

⚡ Boosts speed: Avoid re-downloading or rebuilding.

🔁 Persists between pipeline runs (if defined with a key).

🧺 Stored per job, per branch unless configured otherwise.

Caching (cachelab)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **In Gitlab, create a new blank project “cachelabx”, in the cmtraining1 group**

Do not use your studentx group because you will not have access to the group runner.

2. **In Settings > CI/CD > Runners, uncheck “Enable instance runners for this project”**

If you keep this on, Gitlab will ask you to confirm your identity because it will want you to be accountable for the use of compute minutes. That’s why we have our own Gitlab runner.

3. **Clone your repository and add a file script.sh:**

```
#!/bin/sh
mkdir -p cache_dir
if [ ! -f cache_dir/data.txt ]; then
  echo "Downloading data..."
  echo "This is some data" > cache_dir/data.txt
else
  echo "Using cached data:"
  cat cache_dir/data.txt
fi
```

Caching (cachelab)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

4. **Create a .gitlab-ci.yml file:**

```
default:
  tags:
    - labrunner

stages:
  - prepare
  - use

# Job 1: Download and cache data
download_data:
  stage: prepare
  script:
    - ./script.sh
  cache:
    key: my-cache
    paths:
      - cache_dir/

# Job 2: Reuse cached data
use_cached_data:
  stage: use
  script:
    - ./script.sh
  cache:
    key: my-cache
    paths:
      - cache_dir/
```

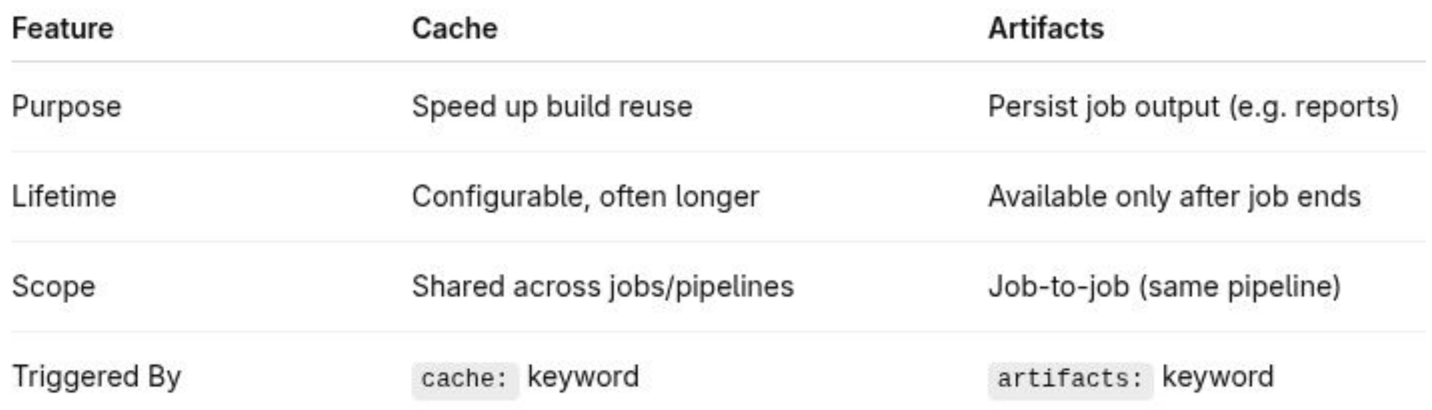
5. **Run your pipeline first with the cache lines commented out and then with the cache lines activated and see the difference in your job outputs.**

🔄 Cache vs 📦 Artifacts

"Cache is here to speed up your job but it may not exist, so don't rely on it."

Foundations of CI/CD Pipelines

Caching



Feature	Cache	Artifacts
Purpose	Speed up build reuse	Persist job output (e.g. reports)
Lifetime	Configurable, often longer	Available only after job ends
Scope	Shared across jobs/pipelines	Job-to-job (same pipeline)
Triggered By	`cache:` keyword	`artifacts:` keyword

Foundations of CI/CD Pipelines

Caching

Using the right key !

If you want a local cache between all your jobs running on the same runner, use the cache statement in your .gitlab-ci.yml:

```
default:
  cache:
    path:
      - relative/path/to/folder/*.ext
      - relative/path/to/another_folder/
      - relative/path/to/file
```

Using the predefined variable CI_COMMIT_REF_NAME as the cache key, you can ensure the cache is tied to a specific branch:

```
default:
  cache:
    key: $CI_COMMIT_REF_NAME
    path:
      - relative/path/to/folder/*.ext
      - relative/path/to/another_folder/
      - relative/path/to/file
```

Using the predefined variable CI_JOB_NAME as the cache key, you can ensure the cache is tied to a specific job.

Adding a deploy stage (lab6)

Exercice

- Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab6.git
- Inspect the pipeline
- Create the following variables in Settings > CI/CD > Variables:

Type	Visibility	Protect variable	Expand variable reference	Key	Value
Variable	Visible	No	No	REGISTRY_URL	registry.creativemoods.pt
Variable	Visible	No	No	REGISTRY_USER	studentx (replace x with your student number)
Variable	Masked and hidden	No	No	REGISTRY_PASSWORD	<assigned password for your student number>
Variable	Visible	No	No	DOMAIN	adessox.creativemoods.pt (replace x with your student number)
File	Visible	No	No	KUBECONFIG	The contents of the kubeconfig file you'll find in your home directory
Variable	Visible	No	No	NAMESPACE	studentx (replace x with your student number)

Adding a deploy stage (lab6)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Trigger the pipeline. It should deploy the containers to the studentx namespace on our Kubernetes cluster.**
2. **You should now be able to test the application at the $DOMAIN URL. Go to https://adessox.creativemoods.pt/ and verify that the application is running correctly.**

A few notes:

- **The Helm deployment contains everything that is needed to instruct Kubernetes to handle SSL termination.**
- **If you want to deploy a new version of the app, you need to increment the appVersion value in backend/helm/Chart.yaml and in frontend/helm/Chart.yaml. This is the value that is used to tag the image that is built in the first stage. If you don’t increment this value, Kubernetes will likely use a cached version of the image during deploy.**
- **We have a dependency between deploy_frontend and deploy_backend because deploy_backend creates the Kubernetes Secret that is used to authenticate to the registry when pulling the images**

Adding a deploy stage (lab6)

Exercice

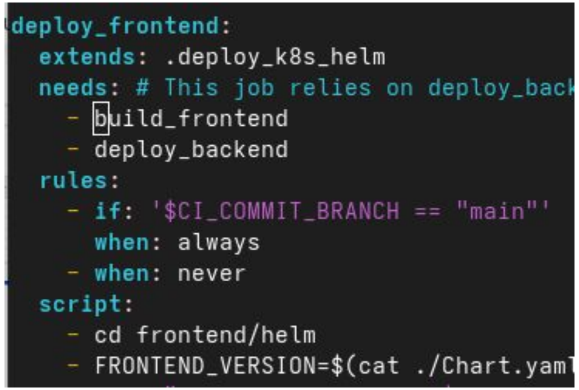
Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. We need to protect our variables. Go to Settings > CI/CD > Variables and mark all variables as protected.
2. Go to Settings > Repository > Protected branches. Verify that main is protected and that Maintainers are allowed to merge and push.
3. Create a new branch from main and push it to Gitlab:

```
$ git branch newbranch
$ git switch newbranch
$ git push origin newbranch
```

4. The pipeline should fail because it doesn’t have access to the environment variables.
5. The error “Cannot perform an interactive login from a non TTY device” happens because the docker login command sees empty variables and tries to ask for a password on the command line but it doesn’t have a TTY to do so. Let’s make this a bit cleaner by implementing a check in the pipeline. Add the following section to each of the four build and deploy jobs (still in newbranch):

```
…
extends: …
rules:
  - if: '$CI_COMMIT_BRANCH == "main"'
    when: always
  - when: never
script:
…
```



```
deploy_frontend:
  extends: .deploy_k8s_helm
  needs: # This job relies on deploy_back
    - build_frontend
    - deploy_backend
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: always
    - when: never
  script:
    - cd frontend/helm
    - FRONTEND_VERSION=$(cat ./Chart.yaml
```

Adding a deploy stage (lab6)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

7. **Commit and push your changes (to newbranch!). The pipeline should run without any errors but only the test_backend job.**

8. **Switch back to main, merge newbranch into main so that main also contains this improvement.**
9. **For the two deploy jobs, replace <always> with <manual> and commit and push. The pipeline should now ask for a manual validation before deployment.**

Foundations of CI/CD Pipelines

Gitlab rules

🧠 **What Are rules:?**

- **Define conditional logic for job execution**
- **More flexible than only/except**
- **Use expressions (e.g., if: '$CI_COMMIT_BRANCH == "main"')**

⏱️ **Using when:**

- **Controls job behavior based on rule match:**
  - **on_success (default): Run if all prior jobs succeed**
  - **always: Run regardless of prior job results**
  - **manual: Requires manual trigger in UI**
  - **never: Prevents job from running**
  - **delayed: Waits before running**

🧪 **Example:**

```
deploy_job:
  script: ./deploy.sh
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      when: manual
    - when: never
```

Foundations of CI/CD Pipelines

GitLab only and except (Legacy Conditions)

📌 **Purpose**

- **Define when jobs should run based on:**
  - **Branch names**
  - **Tags**
  - **Pipeline sources (e.g., merge request, web, trigger)**

🟢 **only:**

- **Job runs only if the condition matches**

🔴 **except:**

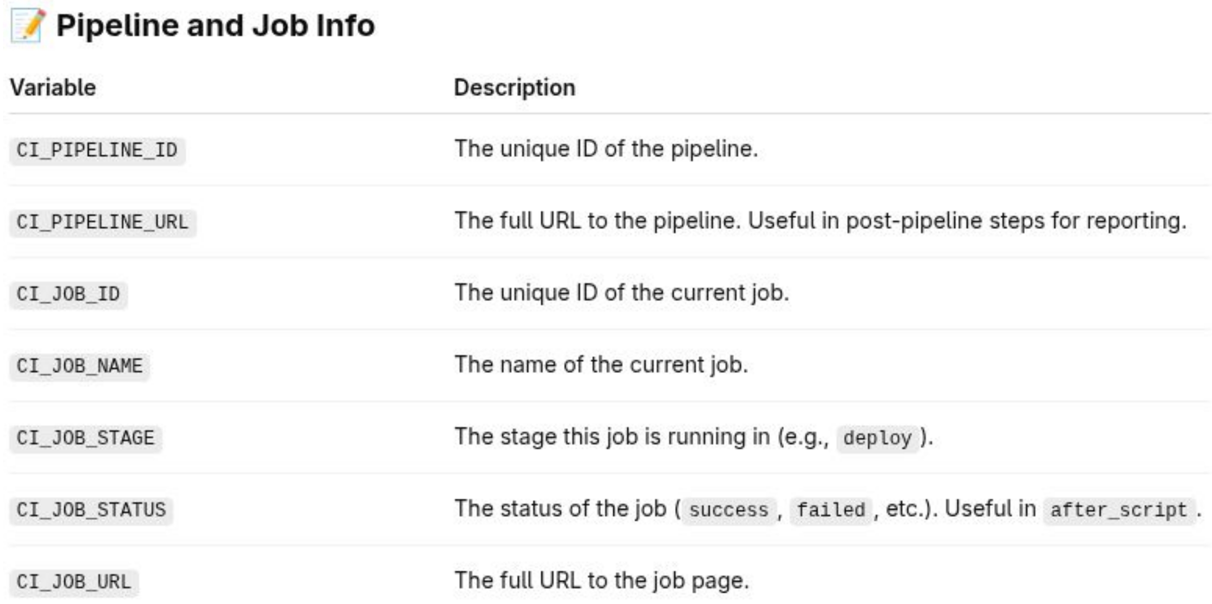
- **Job runs unless the condition matches**

🧪 **Example:**

```
test_job:
  script: ./run-tests.sh
  only:
    - main
    - merge_requests
  except:
    - tags
```

Foundations of CI/CD Pipelines

Gitlab Variables

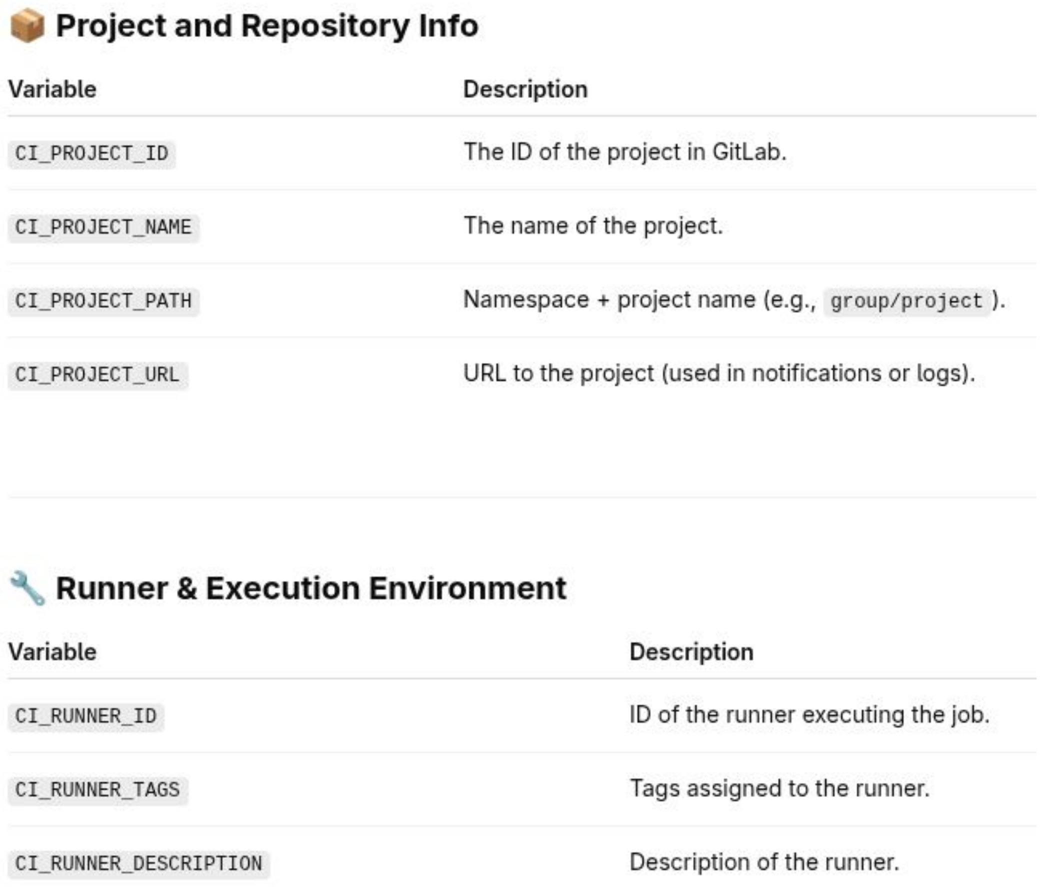


📝 Pipeline and Job Info

Variable	Description
`CI_PIPELINE_ID`	The unique ID of the pipeline.
`CI_PIPELINE_URL`	The full URL to the pipeline. Useful in post-pipeline steps for reporting.
`CI_JOB_ID`	The unique ID of the current job.
`CI_JOB_NAME`	The name of the current job.
`CI_JOB_STAGE`	The stage this job is running in (e.g., `deploy`).
`CI_JOB_STATUS`	The status of the job (`success` , `failed` , etc.). Useful in `after_script` .
`CI_JOB_URL`	The full URL to the job page.

Foundations of CI/CD Pipelines

Gitlab Variables



📦 Project and Repository Info

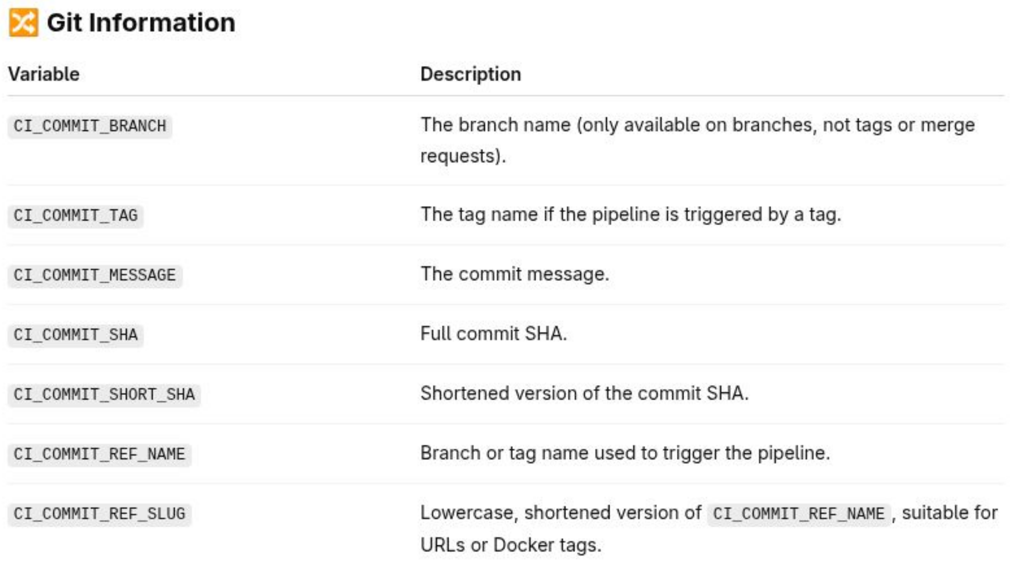
Variable	Description
`CI_PROJECT_ID`	The ID of the project in GitLab.
`CI_PROJECT_NAME`	The name of the project.
`CI_PROJECT_PATH`	Namespace + project name (e.g., `group/project`).
`CI_PROJECT_URL`	URL to the project (used in notifications or logs).

🔧 Runner & Execution Environment

Variable	Description
`CI_RUNNER_ID`	ID of the runner executing the job.
`CI_RUNNER_TAGS`	Tags assigned to the runner.
`CI_RUNNER_DESCRIPTION`	Description of the runner.

Foundations of CI/CD Pipelines

Gitlab Variables

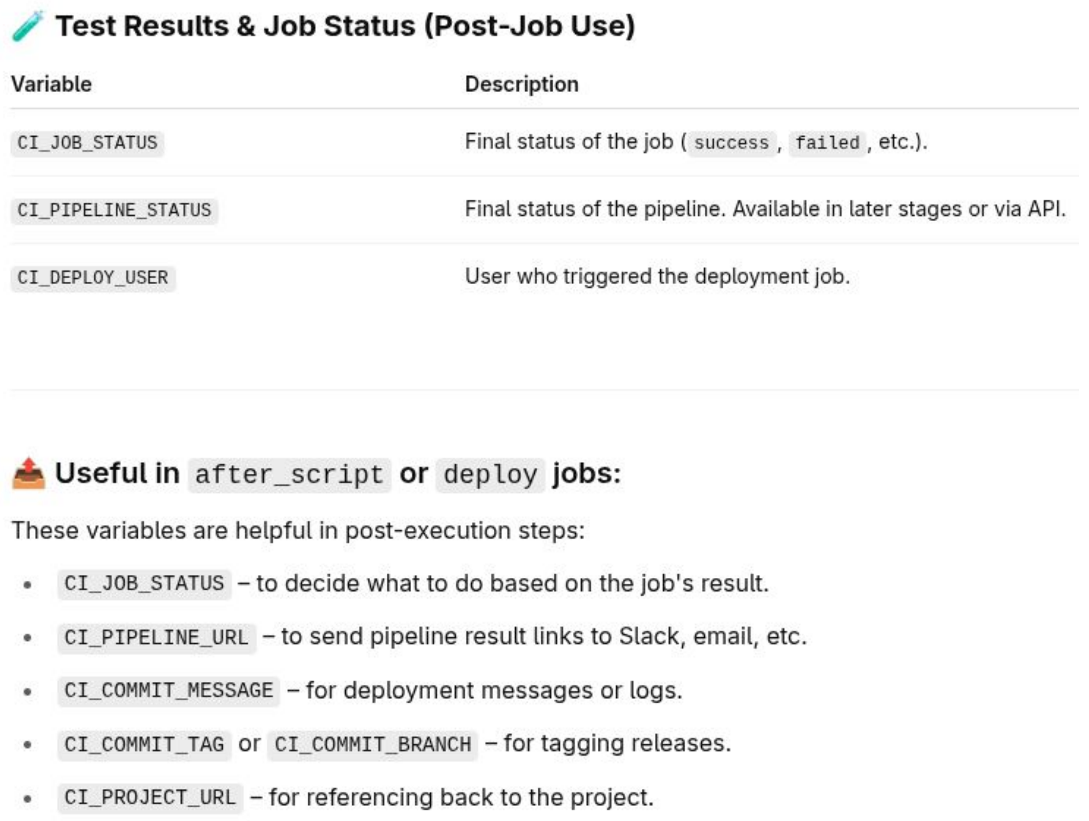


🔀 Git Information

Variable	Description
`CI_COMMIT_BRANCH`	The branch name (only available on branches, not tags or merge requests).
`CI_COMMIT_TAG`	The tag name if the pipeline is triggered by a tag.
`CI_COMMIT_MESSAGE`	The commit message.
`CI_COMMIT_SHA`	Full commit SHA.
`CI_COMMIT_SHORT_SHA`	Shortened version of the commit SHA.
`CI_COMMIT_REF_NAME`	Branch or tag name used to trigger the pipeline.
`CI_COMMIT_REF_SLUG`	Lowercase, shortened version of `CI_COMMIT_REF_NAME`, suitable for URLs or Docker tags.

Foundations of CI/CD Pipelines

Gitlab Variables



🧪 Test Results & Job Status (Post-Job Use)

Variable	Description
`CI_JOB_STATUS`	Final status of the job (`success` , `failed` , etc.).
`CI_PIPELINE_STATUS`	Final status of the pipeline. Available in later stages or via API.
`CI_DEPLOY_USER`	User who triggered the deployment job.

📤 Useful in `after_script` or `deploy` jobs:

These variables are helpful in post-execution steps:

- `CI_JOB_STATUS` – to decide what to do based on the job's result.
- `CI_PIPELINE_URL` – to send pipeline result links to Slack, email, etc.
- `CI_COMMIT_MESSAGE` – for deployment messages or logs.
- `CI_COMMIT_TAG` or `CI_COMMIT_BRANCH` – for tagging releases.
- `CI_PROJECT_URL` – for referencing back to the project.

Foundations of CI/CD Pipelines

Gitlab Variables

If you want to see all predefined variables in your pipeline, you can add this debug job:

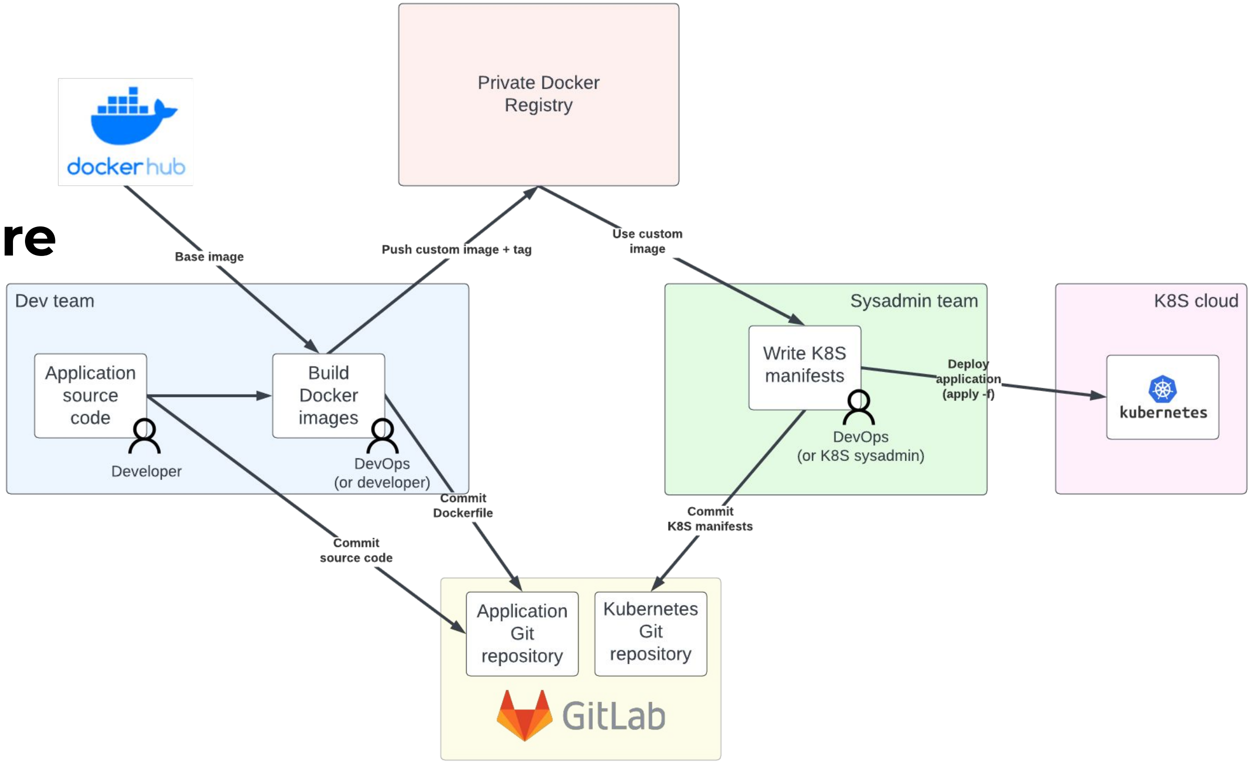
```
print-env:
  stage: debug
  script:
    - printenv | sort
```

Or refer to GitLab's official docs:
https://docs.gitlab.com/ee/ci/variables/predefined_variables.html

Try this in your latest pipeline. Cancel the pipeline once the debug job has passed.

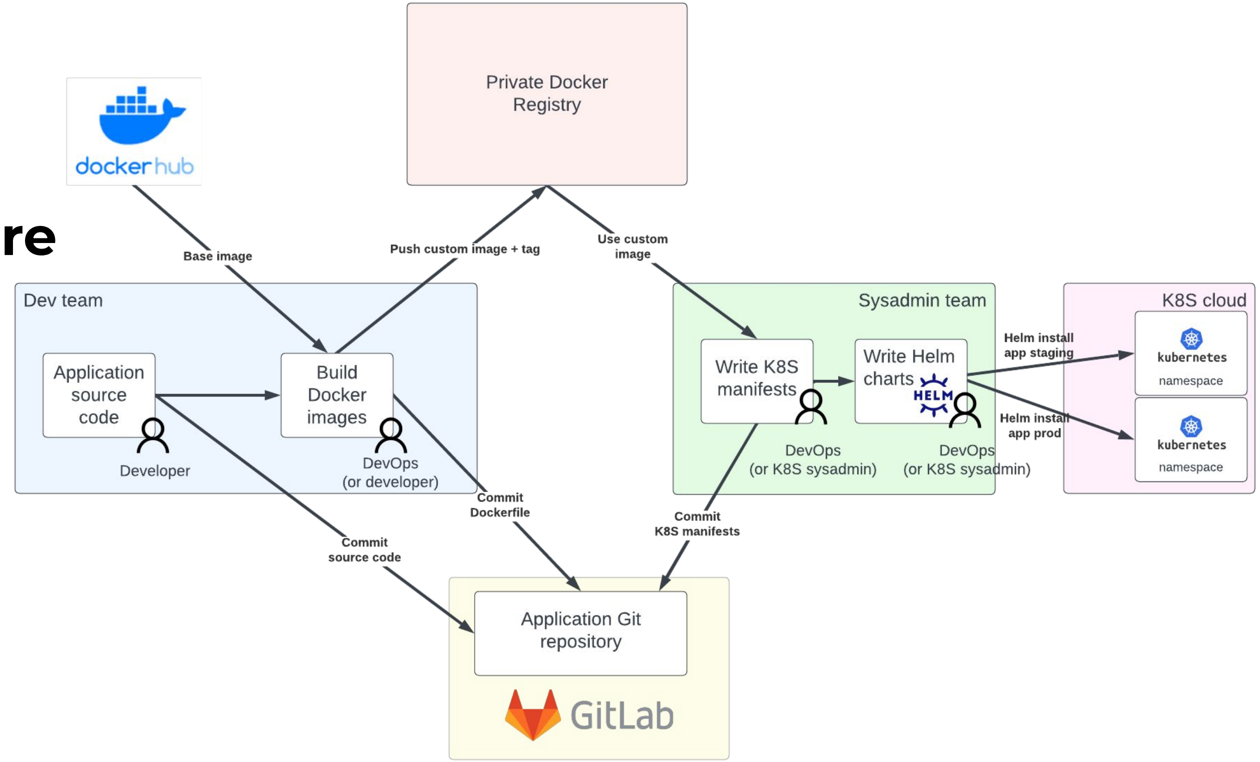
Foundations of CI/CD Pipelines

The big picture



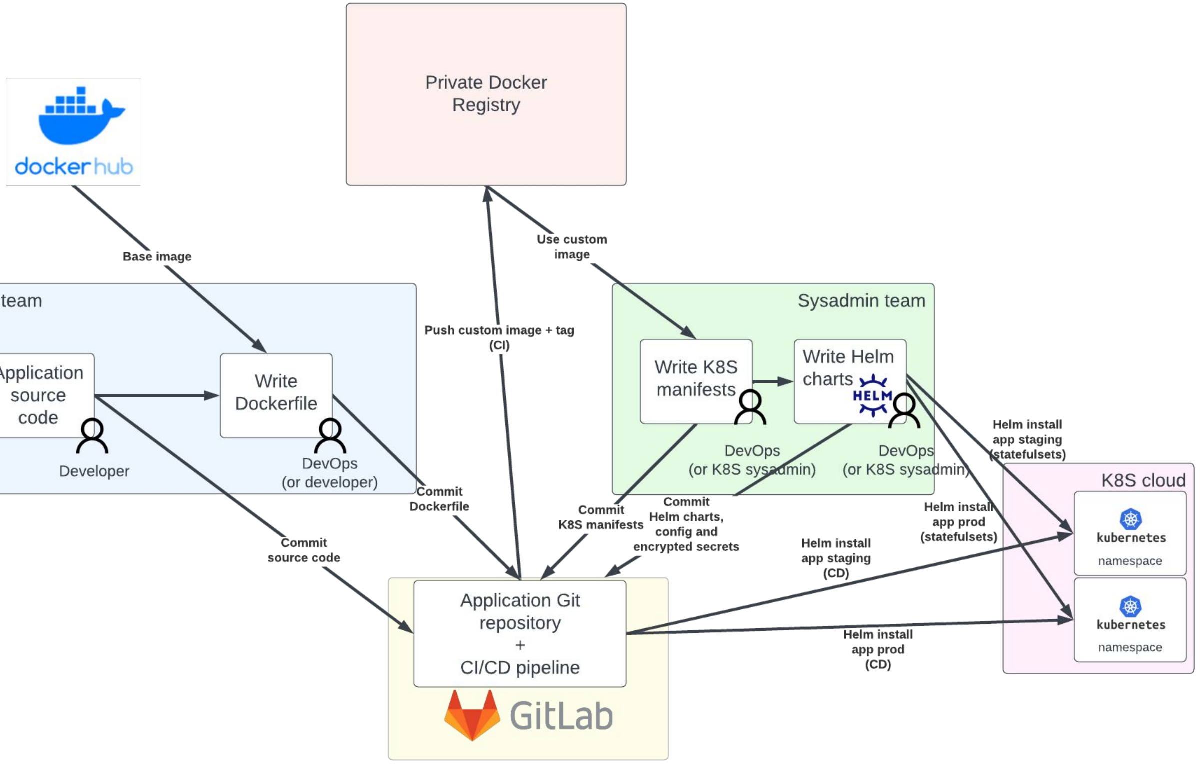
Foundations of CI/CD Pipelines

The big picture



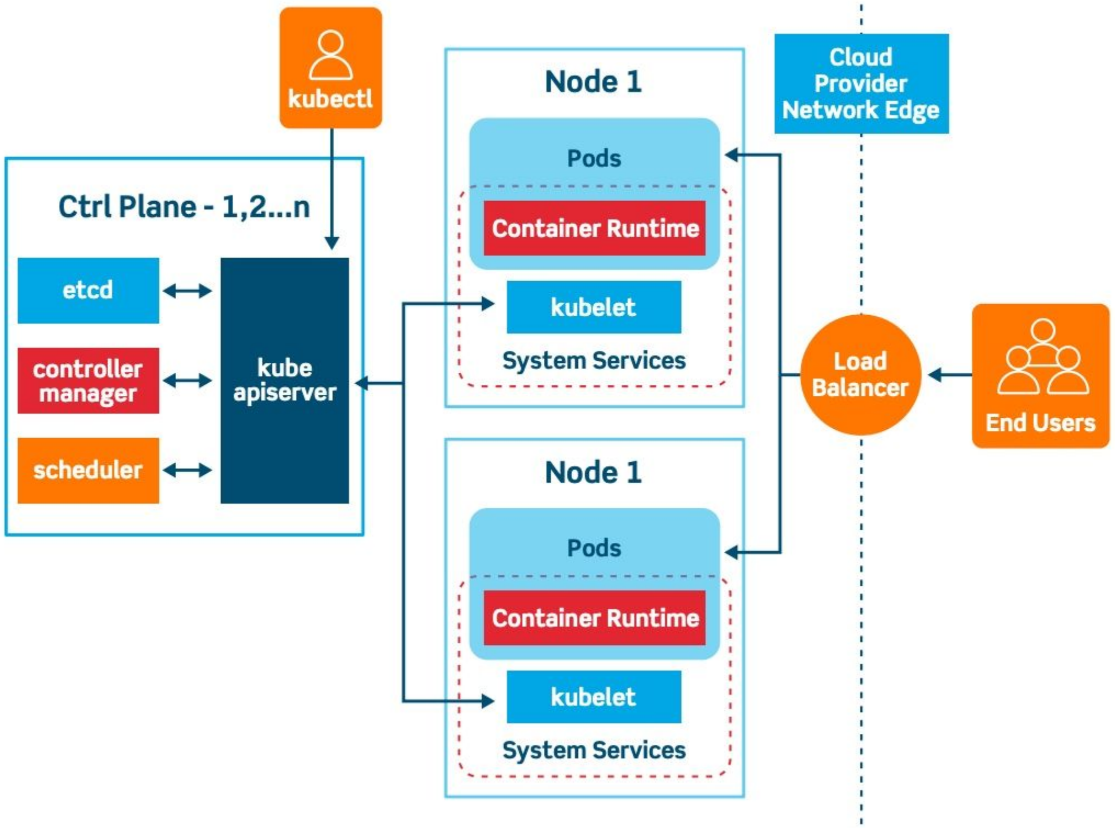
Foundations of CI/CD Pipelines

The big picture



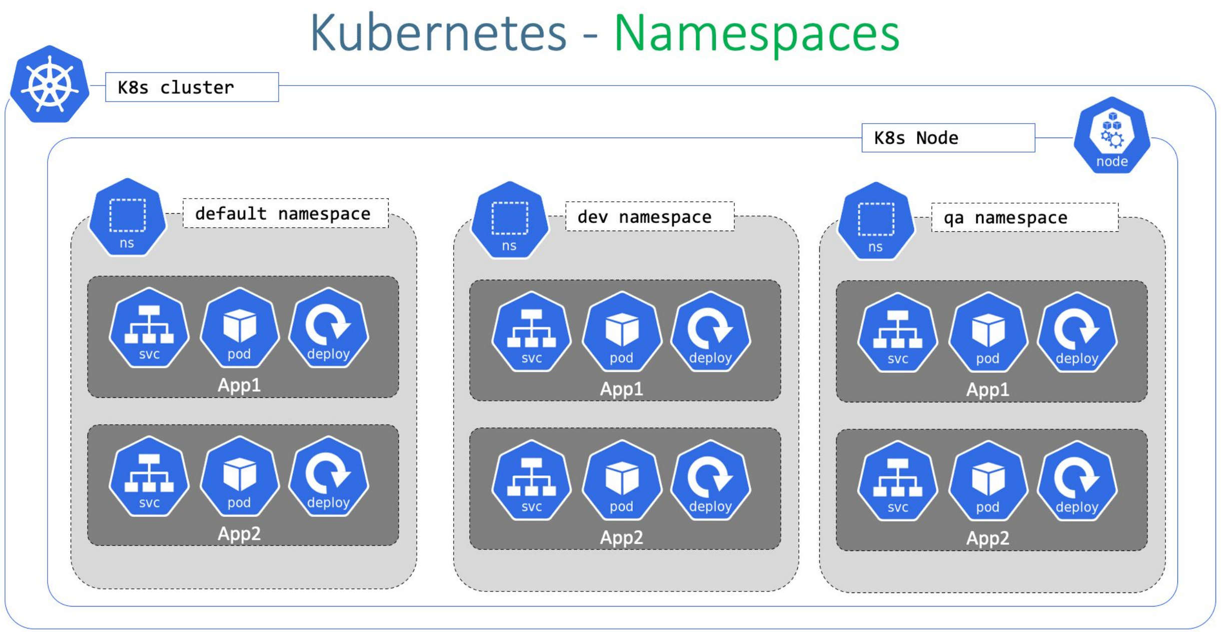
Foundations of CI/CD Pipelines

A note about Kubernetes



Foundations of CI/CD Pipelines

A note about Kubernetes



DAY 2

GitOps, Branching Strategies, and Repository Design for Scale

GitOps, Branching Strategies, and Repository Design for Scale

DAY 2

GitOps, Branching Strategies, and Repository Design for Scale

Objective
Introduce developers to effective branching strategies, GitOps principles, and the pros and cons of monorepo vs. multirepo setups.

Content Outline

- **Branching Strategies for teams:**
  - **Overview of popular branching models:**
    - **GitFlow, Trunk-based Development, and Release Branching.**
  - **How to choose a strategy for your team size and release cycle.**
- **Introduction to GitOps:**
  - **What is GitOps, and why is it important?**
  - **Git as the single source of truth for infrastructure and deployments.**
  - **Tooling examples: Flux, ArgoCD.**
- **Monorepo vs. Multirepo:**
  - **Monorepo: Advantages (simplified dependency management, single source of truth) and challenges (scalability, complexity).**
  - **Multirepo: Advantages (independence, faster builds) and challenges (dependency tracking, fragmentation).**
  - **When to choose each approach.**

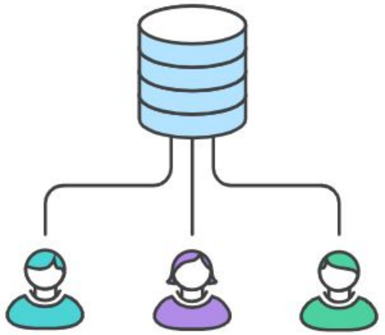
Workflows

Introduction

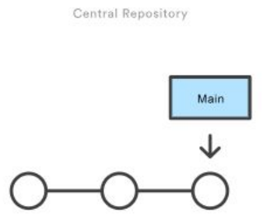
- Git is the most widely used version control system
- A Git workflow provides structure for collaboration
- No single "correct" workflow – Git is flexible by design
- Teams must agree on a shared workflow
- Several common workflows exist to choose from
- Choose or adapt a workflow that suits your team’s needs
- Key evaluation criteria:
  - Scalability with team size
  - Ease of recovering from mistakes
  - Cognitive simplicity

Workflows

Centralized workflow

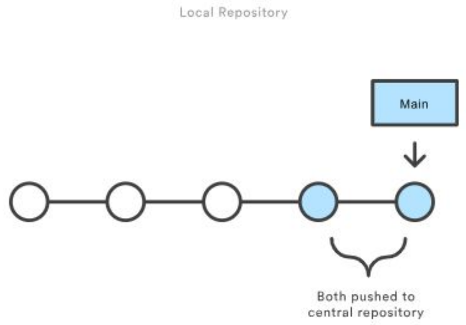


- Ideal for teams transitioning from SVN
- All changes go through a single main branch
- Uses a central repository—no forks or pull requests
- Developers commit locally and push to main
- No additional branches are required
- Key advantages over SVN:
  - Local copy of entire repository
  - Independent, offline work$
  - Git’s powerful branching and merging model
- Best for:
  - Teams new to Git
  - Small teams with simple collaboration needs



Workflows

Centralized workflow



- Developers clone the central repository:

```
git clone ssh://user@host/path/to/repo.git
```

- Each clone is a full local copy of the project
- Developers edit and commit locally
- Changes are isolated until they are pushed
- To publish changes:

```
git push origin main
```

- Equivalent to svn commit, but pushes all new local commits
- Git automatically sets origin as the remote

Workflows

Centralized workflow

- Developers use standard Git workflow:

```
git status       # Check file states
git add <file>   # Stage changes
git commit       # Commit staged changes
```

- Local commits are isolated from the central repo
- Useful for breaking large features into smaller commits
- To share changes:

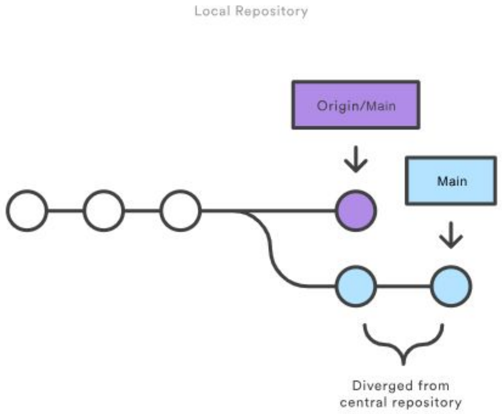
```
git push origin main
```

- Push sends local commits to the central repository
- Push may fail if there are conflicting upstream changes
- In that case, run:

```
git pull
```

Workflows

Centralized workflow



- The central repository is the source of truth
- Git prevents overwriting official history with divergent commits
- To publish changes:
    - Fetch latest changes:

```
git fetch origin
```

-
    - Rebase local commits:

```
git rebase origin/main
```

- Rebase integrates your work on top of upstream changes
- Maintains a clean, linear history
- If conflicts occur:
    - Git will pause and prompt for resolution
    - Use git status and git add to resolve
    - Continue with:

```
git rebase --continue
```

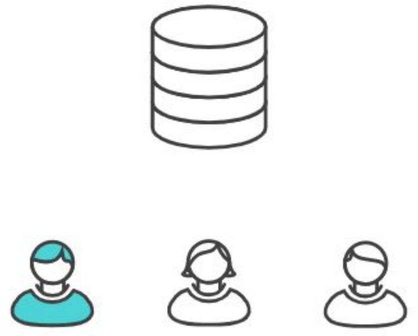
-
    - Or abort with:

```
git rebase --abort
```

Workflows

Centralized workflow

Example. John works on his feature.



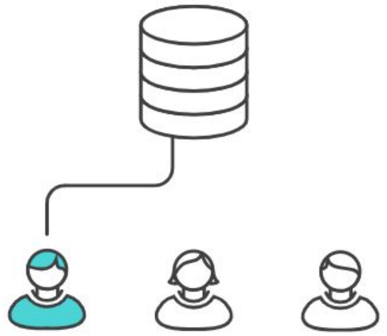
Mary works on her feature

Workflows

Centralized workflow

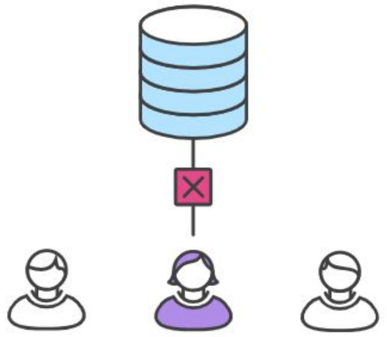
John publishes his feature

```
$ git push origin main
```



Mary tries to publish her feature

```
$ git push origin main
```



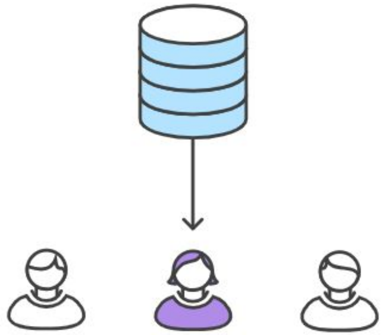
Workflows

Centralized workflow

Mary gets the following error :

```
error: failed to push some refs to '/path/to/repo.git'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. Merge the remote changes (e.g. 'git pull')
hint: before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This prevents Mary from overwriting official commits. She needs to pull John's updates into her repository, integrate them with her local changes, and then try again.
Mary rebases on top of John's commit(s) :



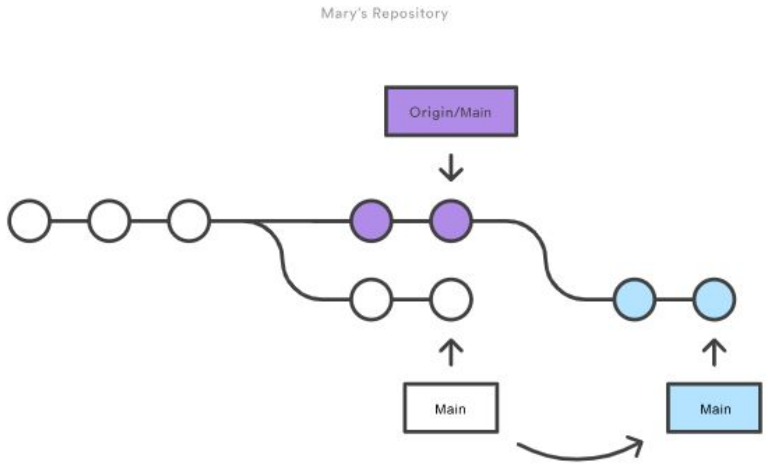
Workflows

Centralized workflow

Mary can use git pull to incorporate upstream changes into her repository. This command is sort of like *svn update*, it pulls the entire upstream commit history into Mary's local repository and tries to integrate it with her local commits:

```
git pull --rebase origin main
```

The --rebase option tells Git to move all of Mary's commits to the tip of the main branch after synchronising it with the changes from the central repository, as shown below:



The pull would still work if you forgot this option, but you would wind up with a superfluous "merge commit" every time someone needed to synchronize with the central repository. For this workflow, it's always better to rebase instead of generating a merge commit.

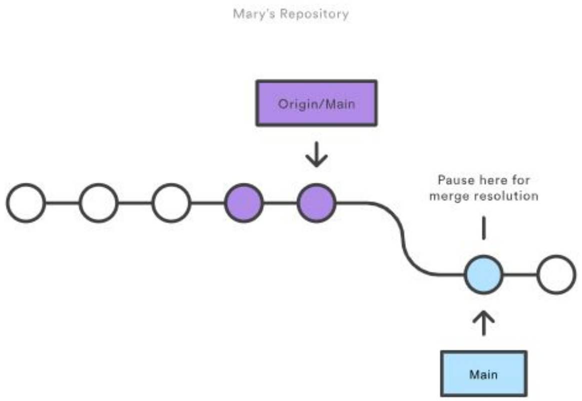
Workflows

Centralized workflow

Rebasing works by transferring each local commit to the updated main branch one at a time. This means that you catch merge conflicts on a commit-by-commit basis rather than resolving all of them in one massive merge commit. This keeps your commits as focused as possible and makes for a clean project history. In turn, this makes it much easier to figure out where bugs were introduced and, if necessary, to roll back changes with minimal impact on the project.

If Mary and John are working on unrelated features, it's unlikely that the rebasing process will generate conflicts. But if it does, Git will pause the rebase at the current commit and output the following message, along with some relevant instructions:

```
CONFLICT (content): Merge conflict in <some-file>
```



Workflows

Centralized workflow

The great thing about Git is that anyone can resolve their own merge conflicts. In our example, Mary would simply run a git status to see where the problem is. Conflicted files will appear in the Unmerged paths section:

```
# Unmerged paths:
# (use "git reset HEAD <some-file>..." to unstage)
# (use "git add/rm <some-file>..." as appropriate to mark resolution)
#
# both modified: <some-file>
```

Then, she'll edit the file(s) to her liking. Once she's happy with the result, she can stage the file(s) in the usual fashion and let git rebase do the rest:

```
git add <some-file>
git rebase --continue
```

And that's all there is to it. Git will move on to the next commit and repeat the process for any other commits that generate conflicts.

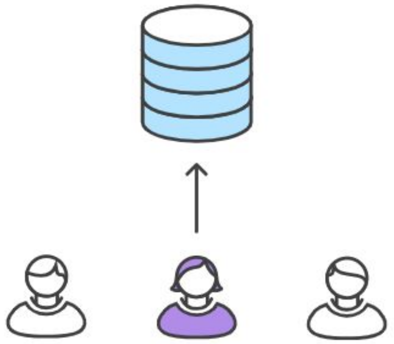
If you get to this point and realize and you have no idea what's going on, don't panic. Just execute the following command and you'll be right back to where you started:

```
git rebase --abort
```

Workflows

Centralized workflow

After she's done synchronizing with the central repository, Mary will be able to publish her changes successfully:



```
$ git push origin main
```

Workflows

Git feature branch workflow

- All feature development happens in dedicated branches
- Keeps main branch clean and stable
- Great for collaborative work and CI pipelines
- Encourages use of pull requests for:
  - Peer review
  - Feedback and collaboration
- Feature branches are:
  - Created from main
  - Named descriptively (e.g., animated-menu, issue-#1061)
- Developers commit changes in the feature branch
- Central repository still holds the official main history

Workflows

Git feature branch workflow

- Feature branches can (and should) be pushed to the central repository
  - Enables collaboration without touching main
  - Serves as a backup of local commits
- main remains the only protected/special branch
- Storing multiple feature branches in central repo is safe and encouraged
- Feature branch lifecycle starts with updating main:

```
$ git checkout main
$ git fetch origin
$ git reset --hard origin/main
```

- Create and switch to a new feature branch:

```
$ git branch new-feature
```

Workflows

Git feature branch workflow

- On your feature branch:
  - Edit, stage, and commit changes as usual
  - Make as many commits as needed

```
$ git status
$ git add <some-file>
$ git commit
```

  - Push the feature branch to the central repository:

```
$ git push -u origin new-feature
```

  - -u sets up a remote tracking branch
  - After this, future pushes can be just git push
- Pushing serves as:
  - A backup of your local work
  - A way for others to collaborate or review
- Open a pull request to:
  - Start a code review
  - Get feedback
  - Prepare for merge into main

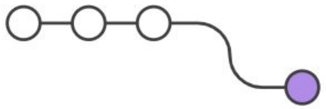
Workflows

Git feature branch workflow

- Teammates review and comment on the feature branch
- You:
  - Resolve feedback locally
  - Commit and push updates
  - See updates reflected in the same pull request
- If there are merge conflicts, resolve them before merging
- Once approved and conflict-free:
  - Merge the branch into main via the pull request
- Pull requests:
  - Enable discussion and code review
  - Prevent premature merges into main
  - Can be opened early, even before the feature is complete
  - Notify teammates automatically for feedback or help

Workflows

Git feature branch workflow



- **After the pull request is accepted:**
  - **Sync local main with upstream:**

```
git checkout main
git pull origin main
```

  - **Merge the feature branch:**

```
git merge marys-feature
```

  - **Push updated main:**

```
git push origin main
```

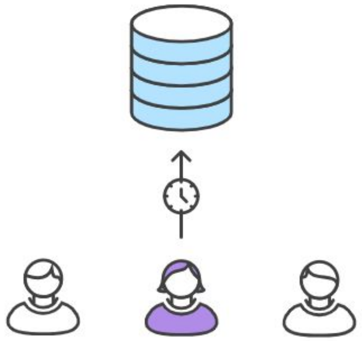
- **Scenario Example:**
  - **Mary starts feature development by creating a new branch:**

```
git checkout -b marys-feature main
```

  - **This gives her an isolated environment for work and collaboration**

Workflows

Git feature branch workflow



- Create a new branch based on main:

```
git checkout -b marys-feature main
```

  - -b creates the branch if it doesn’t exist
- On this branch, Mary:
  - Edits, stages, and commits her changes

```
git status
git add <file>
git commit
```

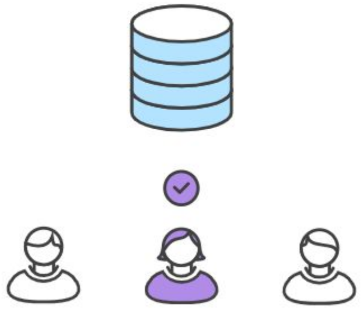
  - Can make multiple commits while building the feature
- Before lunch, she pushes her work:

```
git push -u origin marys-feature
```

  - Acts as a backup
  - Enables collaboration if others need access

Workflows

Git feature branch workflow



- Push the feature branch and set up tracking:

```
git push -u origin marys-feature
```

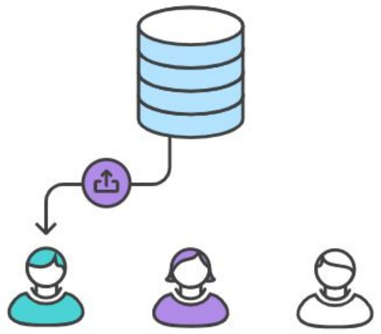
  - -u sets the remote tracking branch
  - Future pushes can use git push with no parameters
- After returning from lunch:
  - Mary completes the feature
  - Pushes final local commits:

```
git push
```

  - This ensures the central repo is up to date
- Now she’s ready to:
  - Open a pull request
  - Let the team know the feature is ready for review

Workflows

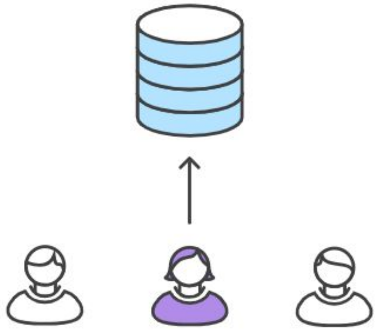
Git feature branch workflow



- Mary files a pull request (PR):
  - Requests to merge marys-feature into main
  - PR created using a Git GUI (e.g., GitLab, GitHub, Bitbucket)
  - Team is automatically notified
- Benefits of pull requests:
  - Comments appear next to relevant commits
  - Easy to ask questions and request changes
- Bill reviews the PR:
  - Requests some modifications
  - Collaborates with Mary through comment threads
- Mary makes updates:
  - Edits, stages, commits, pushes just like before
  - Changes are automatically reflected in the PR
  - Bill continues reviewing updated work

Workflows

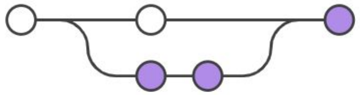
Git feature branch workflow



- Bill can also:
  - Pull marys-feature locally and contribute
  - Any commits he adds will appear in the PR
- To merge the feature into main:

```
git checkout main
git pull
git pull origin marys-feature
git push
```

- This typically creates a merge commit
  - Symbolizes integrating the feature into the codebase
- Prefer a linear history?
  - Use git rebase before merging
  - Enables a fast-forward merge



Workflows

Git feature branch workflow

- Many Git GUIs:
  - Automate merging pull requests with an “Accept” or “Merge” button
  - Automatically close PRs after merging
- Developers can work in parallel:
  - John is working on his own feature branch at the same time
  - No interference between branches
- Isolating features:
  - Enables independent development
  - Keeps the main branch stable
  - Simplifies collaboration and sharing

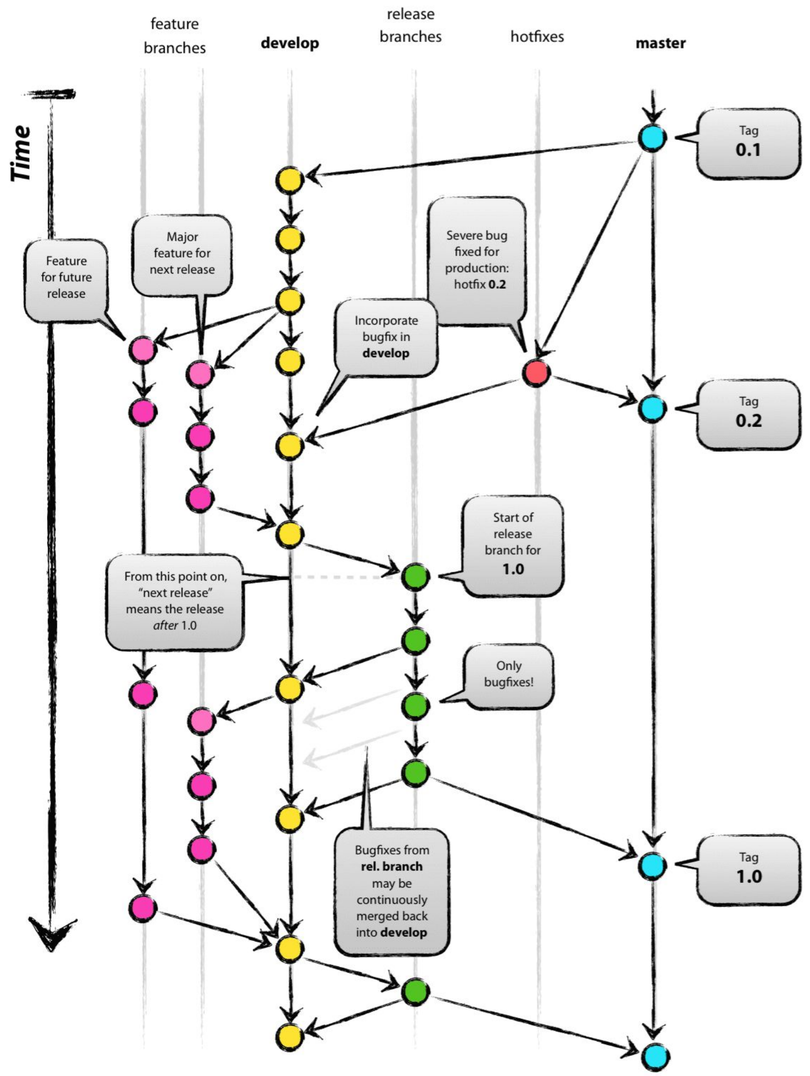
Workflows

Gitflow workflow

- Legacy workflow using multiple long-lived branches
- First popularized by Vincent Driessen at nvie
- Uses:
  - main (or master) for production-ready code
  - develop for integration of features
  - Separate branches for features, releases, and hotfixes
- Long-lived feature branches can:
  - Delay integration
  - Increase merge complexity and risk
- Challenges with CI/CD:
  - Harder to automate and deploy incrementally
- Modern trend: Favoring trunk-based development for simplicity and speed

Workflows

Gitflow workflow


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.0

Workflows

GitHub Flow

- Simple, lightweight workflow
- Only one long-lived branch: main
- New branch for every feature or fix
- Changes go through a pull request
- Emphasizes:
    - Continuous integration
    - Fast review and merge cycles
- Common in cloud-native and web app projects
- Great fit for CI/CD pipelines and small teams

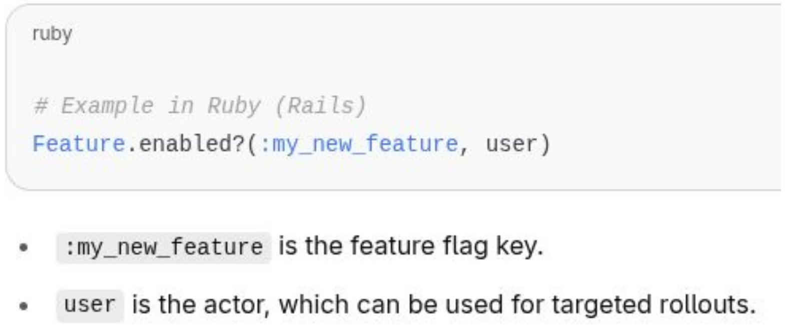
Workflows

Trunk-Based Development

- Developers commit directly to main (or short-lived branches)
- No long-lived feature branches
- Encourages small, frequent commits
- Feature toggles manage incomplete features
- Designed for high velocity and continuous integration
- Supports daily or multiple daily deploys
- Promotes strong team communication and test automation

Workflows

Feature flags



ruby

```ruby
# Example in Ruby (Rails)
Feature.enabled?(:my_new_feature, user)
```

- `:my_new_feature` is the feature flag key.
- `user` is the actor, which can be used for targeted rollouts.

🚦 Progressive Delivery with Confidence

🔧 What Are They?

- Toggles in code to turn features on/off
- Controlled via GitLab UI or API

⚙️ Key Components

- Scopes → Environment-specific flags
- Strategies → Control rollout: % users, user ID, etc.

🎯 Use Cases

- Canary releases
- A/B testing
- Safe deployments
- Quick rollbacks

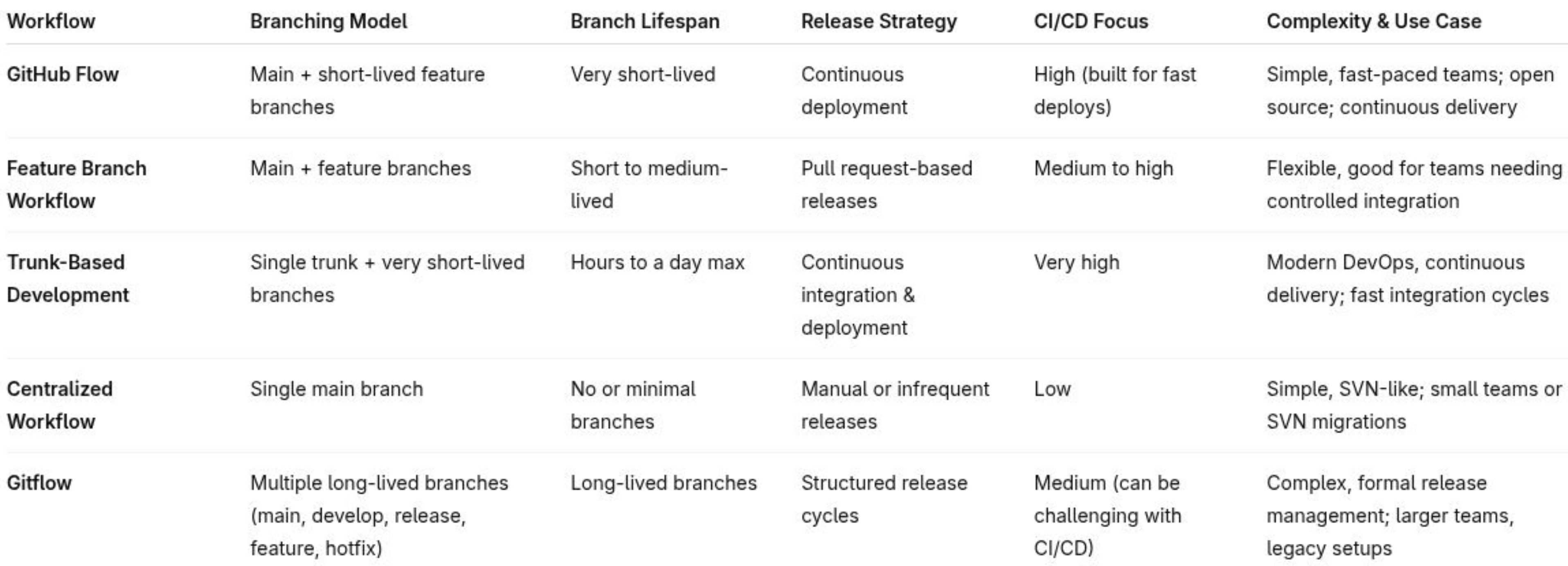
🧪 CI/CD Integration

- Enable features in Review Apps
- Control feature behavior during testing

💡 Tip

Clean up old flags after full rollout!

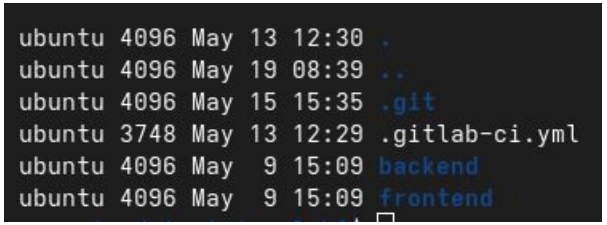
Comparison



Workflow	Branching Model	Branch Lifespan	Release Strategy	CI/CD Focus	Complexity & Use Case
GitHub Flow	Main + short-lived feature branches	Very short-lived	Continuous deployment	High (built for fast deploys)	Simple, fast-paced teams; open source; continuous delivery
Feature Branch Workflow	Main + feature branches	Short to medium-lived	Pull request-based releases	Medium to high	Flexible, good for teams needing controlled integration
Trunk-Based Development	Single trunk + very short-lived branches	Hours to a day max	Continuous integration & deployment	Very high	Modern DevOps, continuous delivery; fast integration cycles
Centralized Workflow	Single main branch	No or minimal branches	Manual or infrequent releases	Low	Simple, SVN-like; small teams or SVN migrations
Gitflow	Multiple long-lived branches (main, develop, release, feature, hotfix)	Long-lived branches	Structured release cycles	Medium (can be challenging with CI/CD)	Complex, formal release management; larger teams, legacy setups

Monorepo vs multirepo

Monorepo



```
ubuntu 4096 May 13 12:30 .
ubuntu 4096 May 19 08:39 ..
ubuntu 4096 May 15 15:35 .git
ubuntu 3748 May 13 12:29 .gitlab-ci.yml
ubuntu 4096 May  9 15:09 backend
ubuntu 4096 May  9 15:09 frontend
```

Advantages

- Simplified dependency management across all projects.
- Single source of truth for all code, increasing visibility.
- Easier code reuse and refactoring across teams.
- Streamlined tooling and CI/CD pipelines.

Challenges

- Scalability issues with very large codebases.
- Complex build systems required to handle unrelated changes.
- Risk of accidental coupling between components.
- Tooling limitations for large monolithic repositories.

Monorepo CI/CD example (lab6)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Modify the build_backend job by adding the following:**

```
build_backend:
  extends: .build_images
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
      changes:
        - backend/**
      when: always
    - when: never
```

2. **Do the same with the build_frontend, deploy_backend and deploy_frontend jobs (you need to adapt the directory of course and keep the manual run for the deploy jobs)**
3. **If you commit and push your repository you should not see any build or deploy jobs running.**
4. **Add a comment somewhere in backend/backend.py and commit and push the change. You should only see the backend jobs running.**

Monorepo vs multirepo

Multirepo

Advantages

- Team autonomy with independent versioning and deployment.
- Faster builds and tests due to smaller codebases.
- Clear ownership and access control per repository.
- Easier to adopt different tech stacks or tooling per repo.

Challenges

- Dependency tracking can become manual and error-prone.
- Risk of fragmentation and duplicated logic.
- Cross-repo changes are harder to coordinate.
- Requires robust release and integration management.

Monorepo vs multirepo

Multirepo CI/CD example

```
stages:
  - trigger
  - validate
  - deploy

variables:
  BACKEND_PROJECT_ID: 12345
  FRONTEND_PROJECT_ID: 67890
  REF: "main"

trigger_backend:
  stage: trigger
  trigger:
    project: $BACKEND_PROJECT_ID
    branch: $REF
    strategy: depend  # waits for
pipeline completion
  only:
    - main

trigger_frontend:
  stage: trigger
  trigger:
    project: $FRONTEND_PROJECT_ID
    branch: $REF
    strategy: depend
  only:
    - main

integration_tests:
  stage: validate
  image: node:18
  script:
    - echo "Running E2E tests
after frontend and backend deploy"
    - npm ci && npm run test:e2e
  only:
    - main

deploy_stage:
  stage: deploy
  script:
    - echo "Final coordinated
deployment step"
    # Optionally deploy infra,
notify teams, or tag releases
  only:
    - main
```

Monorepo vs multirepo

When to Choose Each

Choose Monorepo when:

- Teams share libraries or infrastructure.
- You value centralized governance and consistency.
- You have strong tooling to manage scale.

Choose Multirepo when:

- Teams operate independently with distinct lifecycles.
- You prioritize build performance and repo clarity.
- Projects are loosely coupled or use varied tech stacks.

DAY 2

Monitoring and Observability

DAY 2

Monitoring and Observability

Key Topics:

- Monitoring principles and the "Three Pillars of Observability" (metrics, logs, and traces).
- Setting up Prometheus for pipeline, system metrics and app metrics collection.
- Visualizing metrics with Grafana dashboards.
- Creating and analyzing dashboards for system health.
- Incident response workflows.
- Security logging essentials:
  - Identifying suspicious activities in logs (e.g., unauthorized access attempts).
  - Correlating logs for security incident detection.
  - Proactive alerting for pipeline issues and potential security threats.

Hands-On Lab (paired):

- Integrate Prometheus and Grafana with GitLab to monitor pipeline performance.
- Build dashboards to monitor CI/CD pipeline performance.
- Configure security logging to detect and analyze failed login attempts and unauthorized changes.

Monitoring and Observability

Monitoring Principles and the Three Pillars of Observability

Monitoring Principles

- **Detect and respond to system health and performance issues.**
- **Focus on known failure modes and thresholds.**
- **Alerts should be actionable, timely, and relevant.**
- **Keep dashboards simple and focused on service-level indicators (SLIs).**

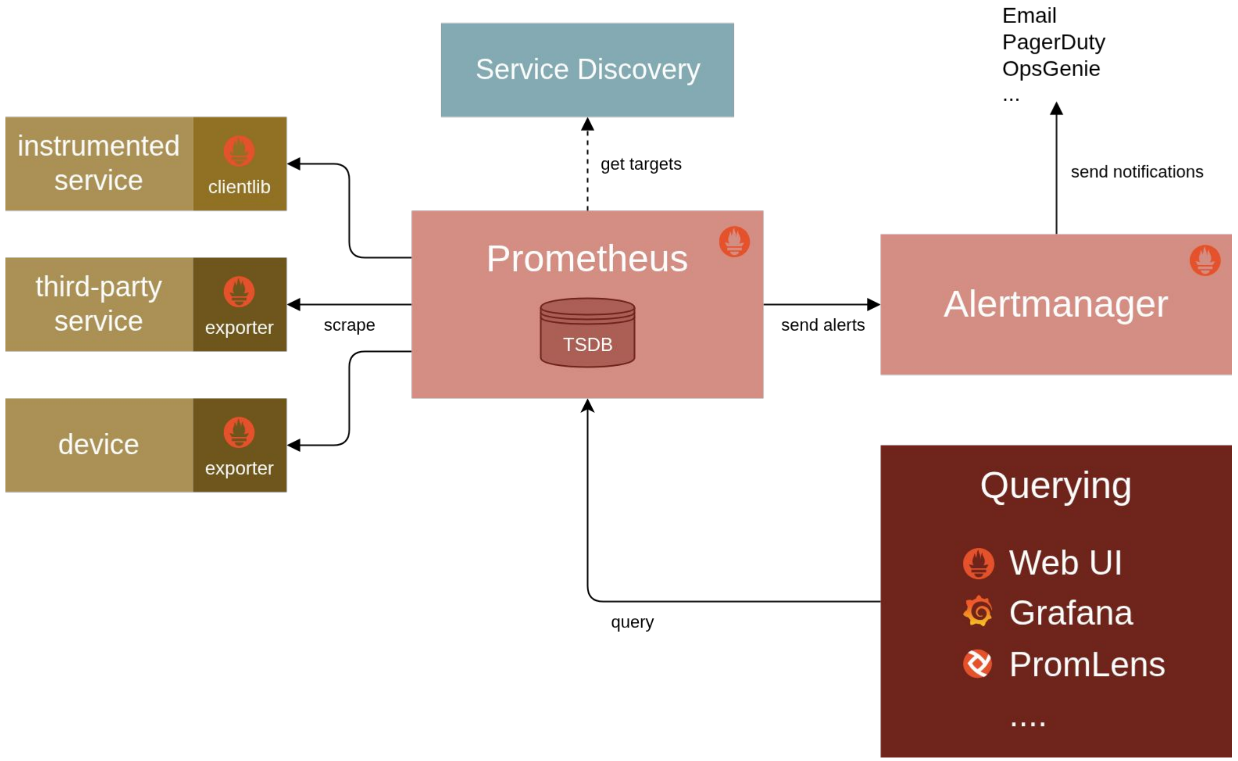
Three Pillars of Observability

- **Metrics: Quantitative data — e.g., CPU usage, request rate, error rate.**
- **Logs: Discrete, timestamped records of events — e.g., stack traces, app events.**
- **Traces: End-to-end record of a request across services — helps identify latency and bottlenecks.**

Together, these pillars provide context, correlation, and causality in complex systems.

Monitoring and Observability

Prometheus



Metrics endpoint (lab7)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab7.git
2. Inspect the /api/metrics endpoint in backend.py
3. If you know Kubernetes, you can inspect the file backend/helm/templates/sm-backend.yaml which defines a ServiceMonitor. This will get picked up by Prometheus so it can scrape data from the svc-backend Service
4. Modify your CI/CD variables to match your environment (they have been pre-created), you just need to change values for:
    a. DOMAIN (replace x)
    b. KUBECONFIG (copy-paste the contents of your kubeconfig-studentx.yaml file)
    c. NAMESPACE (replace x)
    d. REGISTRY_PASSWORD (paste your password)
    e. REGISTRY_USER (replace x)
5. Modify backend/backend.py and modify the 3 occurrences of adessox, replacing x with your student number
6. Edit the file backend/helm/Chart.yaml and increase the minor number of the appVersion on the last line of the file
7. Commit and push your changes to trigger the CI/CD pipeline and deploy your app to your studentx namespace in our Kubernetes cluster. Manually approve the deploy jobs.

Metrics endpoint (lab7)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

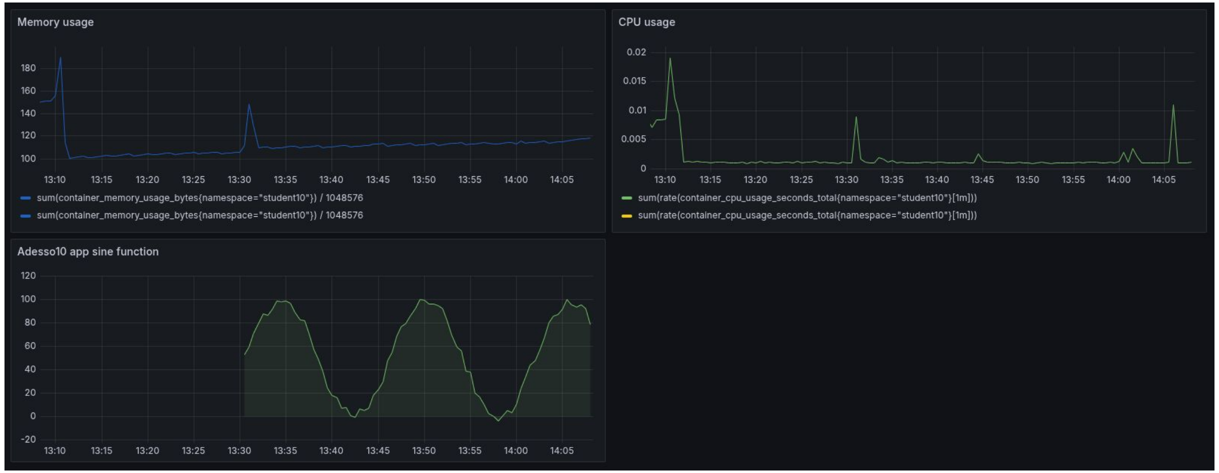
1. **Go to Grafana (https://metrics.creativemoods.pt/) and login with your usual credentials (studentx and your password)**
2. **Go to Explore > Metric "adessox_sine_metric" > Run query. You should see some of the values returned by your application**
3. **Go to Dashboards > Adesso > New dashboard > Add visualization > Select "Prometheus" datasource > Metric "adessox_sine_metric"**
   a. **Change title to "Adessox app sine function"**
   b. **Legend: uncheck Visibility**
   c. **Fill opacity: 10**
4. **Save Dashboard > Title "Studentx dashboard" > Folder "Adesso" > Save**
5. **View your dashboard in Dashboards > Adesso > Studentx dashboard**
6. **Play around with Grafana!**

Metrics endpoint (lab7)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

- Add to your dashboard, a visualization of the amount of CPU and memory your app consumes. As we only know the name of the namespace, we can use the following metrics:
- container_cpu_usage_seconds_total (cumulative CPU time consumed in seconds)
  - Filter by your namespace
  - This is a counter, it always increments. Use a rate function to get values per minute
  - There’s multiple lines because we have the backend, the frontend, the metrics collectors, etc. so use a sum function to get only one line that is the total of all cpu usage for your namespace
- container_memory_usage_bytes (memory usage in bytes)
  - Filter by your namespace
  - Again use a sum function to get a single line
  - Divide by 1048576 to get MiB instead of bytes
- Create a few tasks in your app. You should see a slight increase in CPU usage



```
sum(container_memory_usage_bytes{namespace="student10"}) / 1048576
sum(container_memory_usage_bytes{namespace="student10"}) / 1048576
sum(rate(container_cpu_usage_seconds_total{namespace="student10"}[1m]))
sum(rate(container_cpu_usage_seconds_total{namespace="student10"}[1m]))
```

Alerting (lab7)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

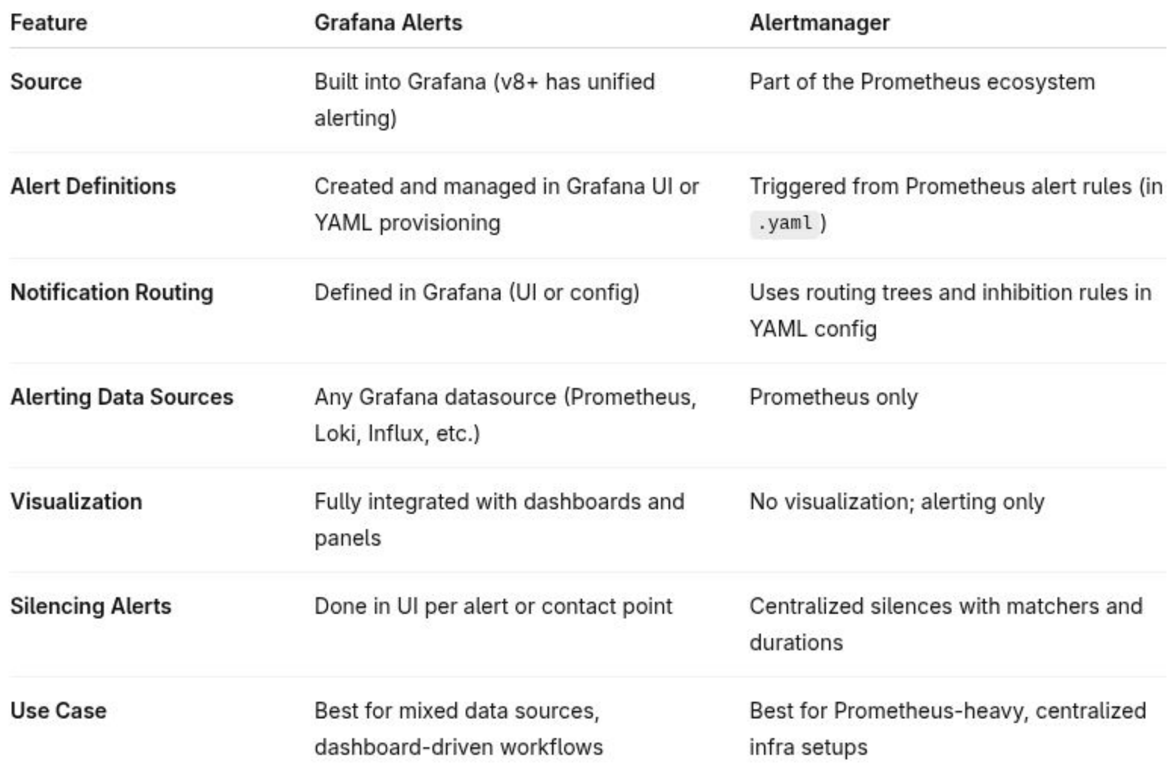
- Create a new contact in Alerting > Contact points
  - Name: Your first name
  - Integration: Email
  - Addresses: Your email address
  - Save contact point
- Alerting > Alert rules > New alert rule
  - Alertx (replace x with your student number)
  - Metric: addessox_sine_metric
  - Alert condition: when query is above 60
  - You can click on Preview alert rule condition to see whether the sine function is above 60 at that moment
  - Folder: Adesso
  - Evaluation group: Create a new evaluation group called Groupx with an evaluation interval of 1m
  - Pending period: 1m
  - Keep firing for: 0s
  - Contact point: the contact you created before
  - Save rule

You should receive an email one minute after the threshold has been reached. During that period, your alert will be in state pending.
You should receive another email immediately when the alert rule is not satisfied anymore.

- Delete the alert rule to prevent receiving emails continuously

Monitoring and Observability

AlertManager



Feature	Grafana Alerts	Alertmanager
Source	Built into Grafana (v8+ has unified alerting)	Part of the Prometheus ecosystem
Alert Definitions	Created and managed in Grafana UI or YAML provisioning	Triggered from Prometheus alert rules (in `.yaml`)
Notification Routing	Defined in Grafana (UI or config)	Uses routing trees and inhibition rules in YAML config
Alerting Data Sources	Any Grafana datasource (Prometheus, Loki, Influx, etc.)	Prometheus only
Visualization	Fully integrated with dashboards and panels	No visualization; alerting only
Silencing Alerts	Done in UI per alert or contact point	Centralized silences with matchers and durations
Use Case	Best for mixed data sources, dashboard-driven workflows	Best for Prometheus-heavy, centralized infra setups

Health endpoints (lab7)

Exercice

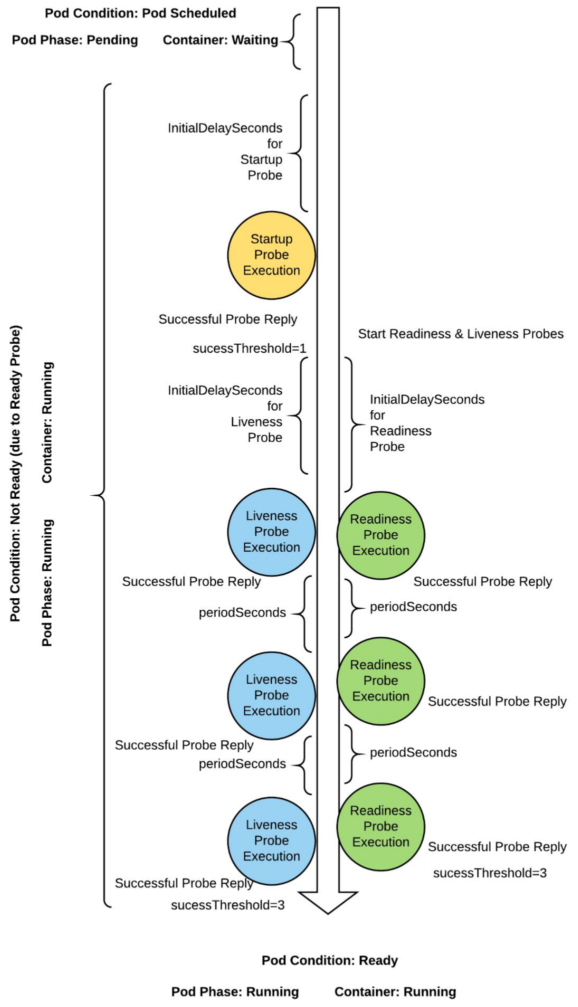
Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

1. Inspect the /api/health and /api/ready endpoints in backend.py
2. In one terminal, run the backend:

```
$ python3 backend.py
```

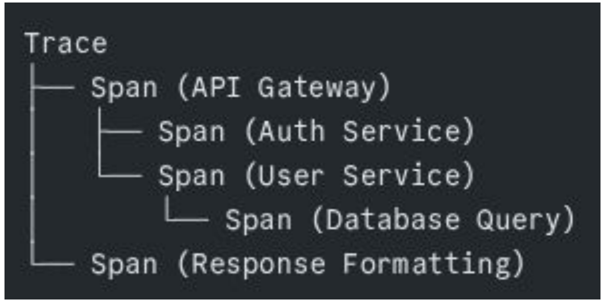
3. In another terminal, issue curl requests to the backend:

```
$ curl -XGET http://localhost:5000/api/ready
$ curl -XGET http://localhost:5000/api/health
$ curl -XGET http://localhost:5000/api/metrics
```



Monitoring and Observability

Traces



```
Trace
├── Span (API Gateway)
│   ├── Span (Auth Service)
│   └── Span (User Service)
│       └── Span (Database Query)
└── Span (Response Formatting)
```

Understanding Traces and Spans: Core Concepts

- **Traces capture the journey of a request as it moves through your distributed system. Think of a trace as the complete story of a request from start to finish—from when a user clicks a button until they see the result.**
- **Spans are the building blocks of traces. Each span represents a unit of work within that journey—like a database query, an API call, or a function execution. Spans nest within each other to show parent-child relationships between operations.**

Benefits of Traces and Spans for DevOps Professionals

- **Find bottlenecks instantly: See exactly which service or function is taking too long**
- **Debug across service boundaries: Follow requests as they jump between services**
- **Understand dependencies: Visualize how your services connect and depend on each other**
- **Improve performance: Identify and fix slow operations with precision**
- **Reduce mean time to recovery (MTTR): Get to the root cause faster when issues arise**

Monitoring and Observability

Traces

Technical Implementation of Traces and Spans

- **Trace Context and Propagation**
- **Span Attributes and Events**
  - **Name: What operation this span represents**
  - **Timing: Start and end times**
  - **Status: Success, error, etc.**
  - **Attributes: Custom key-value pairs (like user_id or cart_size)**
  - **Events: Notable occurrences within the span**
  - **Links: Connections to other spans**
- **Sampling Strategies**

Tracing Implementation Guide: Tools and Frameworks

- **OpenTelemetry: The Industry Standard**
- **The Tracing Toolbox**
  - **Jaeger - Open-source tracing - Best for Self-hosted tracing visualization**
  - **Zipkin - Open-source tracing - Best for Simple distributed tracing**
  - **Grafana Tempo - Tracing backend - Best for Integration with Grafana dashboards**
  - **OpenTelemetry Collector - Data collection pipeline - Best for Processing and routing telemetry data**

Traces (lab8)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab8.git**
2. **Inspect backend/backend.py. We’ve added calls to the opentelemetry Python package. In the API endpoint, we’ve added calls to sleep() to simulate delays.**
3. **Replace the name of the service in backend/backend.py (replace x with your student number):**

```
# Set up the tracer provider and exporter
resource = Resource(attributes={
    "service.name": "backend x"
})
```

4. **Again, replace the 3 adesso_sine_metric with adessox_sine_metric at the end of the file**
5. **Edit the file backend/helm/Chart.yaml and increase the minor number of the appVersion on the last line of the file**
6. **Modify the CI/CD variables to match your environment**
7. **Trigger a commit and watch your build deploy.**
8. **Open https://traces.creativemoods.pt/ where you will see the traces appear. Go to https://adessox.creativemoods.pt/ and create a task. In Jaeger, choose the service that corresponds to your student number and see the trace corresponding to the POST request you just issued.**
9. **What happens if you simulate a failure by creating a task that has less than 3 characters ?**

Monitoring and Observability

Traces

Advanced Tracing Techniques

- Distributed Context Management
  - traceparent: Contains the trace ID and parent span ID
  - tracestate: Allows vendors to add custom context data
- Correlation Between Traces, Metrics, and Logs
  - Exemplar traces: Link metrics to the traces that generated them
  - Trace IDs in logs: Add trace IDs to log messages for cross-referencing
  - Custom attributes: Use consistent attributes across all telemetry types
- Error Handling and Exception Tracking
  - Mark spans with error status
  - Record exceptions with stack traces
  - Add events to spans that show the error's progression
  - Create baggage items that carry error context across service boundaries

Business Value of Traces and Spans: Beyond Technical Benefits

- Track critical user journeys from end-to-end
- Measure the performance of key business operations
- Set SLOs (Service Level Objectives) based on trace data
- Quantify the cost of performance issues in real user terms
- Create business context by adding relevant attributes to spans

DAY 3

Logging and Advanced Practices

Logging and Advanced Practices

DAY 3

Logging and Advanced Practices

Key Topics:

- **Centralized Logging for CI/CD Pipelines:**
  - **Setting up ELK (Elasticsearch, Logstash, Kibana) for log aggregation.**
  - **Troubleshooting pipeline and application issues using centralized logs.**
- **Security Testing in Pipelines:**
  - **Introduction to SAST (Static Application Security Testing).**
  - **Running DAST (Dynamic Application Security Testing) on deployed applications.**
  - **Analyzing and remediating vulnerabilities in SAST and DAST reports.**

Hands-On Lab (paired):

- **Configure centralized logging for GitLab pipelines.**
- **Simulate and troubleshoot common CI/CD pipeline issues.**
- **Integrate GitLab SAST and DAST scans to identify and fix vulnerabilities in the code.**

DAY 3

Elasticsearch

elasticsearch

What is Elasticsearch?

- Distributed, RESTful search and analytics engine
- Built on top of Apache Lucene
- Handles structured and unstructured data
- Used for full-text search, log analytics, and more

Key Concepts

- Index: A collection of documents
- Document: A single JSON object representing a piece of data
- Field: A key-value pair inside a document
- Mapping: Schema definition for an index

How Elasticsearch Works

- Data ingested as JSON documents
- Indexed using inverted indices for fast search
- Distributed across multiple nodes for scalability
- Query with simple REST API or Kibana UI

DAY 3

Elasticsearch

Use Cases

- Log and event data analytics (e.g., ELK stack)
- Website and app search
- Business intelligence dashboards
- Security and monitoring
- CI/CD centralized logging

Kibana Overview

- UI for visualizing and managing Elasticsearch data
- Tools for searching, filtering, and visualizing
- Dashboards to monitor key metrics
- Supports time-series data and alerts

Logs (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

1. Log into https://logs.creativemoods.pt/ as studentx and your password
2. Go to Analytics > Discover
3. Select the "gitlab-traces" Data view
4. Add the following available fields:
   a. job_username
   b. job_stage
   c. job_name
   d. job_status
   e. job_duration
   f. log
5. Make the first columns smaller and the log column larger and in Display options, make sure Cell row height is "Auto fit"
6. Set the Date range to "Last 24 hours"
7. Filter with "job_username:cmstudentx" in order to show only your own build jobs
8. Save your search with the name "Studentx builds"
9. Go to Analytics > Dashboards > Create dashboard
10. Add from Library > Studentx builds
11. Create visualization > Add job_status and choose the Pie chart > Save and return
12. Save your dashboard as Studentx dashboard

Search exercices (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

- Find all jobs that failed among all students since the beginning of the course
    - Filter to only see the deploy jobs
- Find all failed jobs related to the fact that the job did not have access to the variables
- Use the Visualize Library feature to identify the most frequently failing jobs
    - Create a Lens visualization
    - of a Table
    - with job_name in the rows
    - Count of records in the metrics
    - filtered by job_status “failed”
- Find the slowest jobs
    - Create a Lens visualization
    - Show the average of job_duration by job_name

Application logs (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab9.git**
2. **Inspect backend/backend.py. We’ve added the following lines:**

```
import logging
# Set up logging
class TraceIdLogFilter(logging.Filter):
    def filter(self, record):
        from opentelemetry.trace import get_current_span
        span = get_current_span()
        if span and span.get_span_context().is_valid:
            context = span.get_span_context()
            record.trace_id = format(context.trace_id, '032x')
            record.span_id = format(context.span_id, '016x')
        else:
            record.trace_id = "N/A"
            record.span_id = "N/A"
        return True

logger = logging.getLogger("backend")
logger.setLevel(logging.INFO)
handler = logging.StreamHandler()
formatter = logging.Formatter(
    '%(asctime)s %(levelname)s [trace_id=%(trace_id)s span_id=%(span_id)s] %(message)s'
)
handler.setFormatter(formatter)
handler.addFilter(TraceIdLogFilter())
logger.addHandler(handler)
```

Application logs (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

3. **Replace the name of the service in backend/backend.py (replace x with your student number):**

```
# Set up the tracer provider and exporter
resource = Resource(attributes={
    "service.name": "backend x"
})
```

4. **Modify the CI/CD variables to match your environment.**
5. **Commit your changes and manually confirm the deploy.**
6. **Go to Elasticsearch (logs.creativemoods.pt) > Analytics > Discover**
   a. **Data view: filebeat-***
   b. **Add columns kubernetes.namespace, span_id, trace_id, message**
   c. **Filter by kubernetes.namespace:studentx**
   d. **Display last 15 minutes and refresh every 10 seconds**
7. **View your application at https://adessox.creativemoods.pt/ and create a new task.**
8. **You should see a line in your Elasticsearch logs that contains the text “Generated task ID: xxxx”. The fields trace_id and span_id should have data.**
9. **Copy your trace_id and paste it in the “Lookup by trace ID” search box in Jaeger**
10. **You now have correlation between your logs and your traces.**

Logging and Advanced Practices

Metrics correlation

- Correlating trace IDs with metrics is a bit trickier than logs, because most metrics systems (like Prometheus + Grafana) aggregate data — they don't natively track per-request context like traces or logs do.
- Prometheus can be overwhelmed due to cardinality explosion
- Not recommended for high-cardinality environments, but fine for development environments
- Prometheus Exemplars are a Prometheus feature that link specific metric data points to trace IDs.

More info here: https://grafana.com/docs/grafana/latest/fundamentals/exemplars/

Logging and Advanced Practices

Static Application Security Testing (SAST)

SAST: Analyze source code for security vulnerabilities before the app runs.

GitLab SAST - Built-in CI/CD security job

- **Scans code using language-specific analyzers**
- **Simple to integrate:**

```
include:
  - template: Security/SAST.gitlab-ci.yml
```

Bandit (Python-specific)

- **Open-source SAST tool for Python**
- **Detects common security issues (e.g., eval, subprocess misuse)**
- **Easy to run manually or in CI:**

```
bandit -r . -f json -o bandit-report.json
```

Benefits

- **Find bugs early (before deploy)**
- **Lightweight and fast**
- **Prevents vulnerable code from merging**

Best practice:

- **Use both Gitlab SAST and your language's SAST tool**

SAST (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Modify .gitlab-ci.yml to include the following job (between stages and build_backend but it can go anywhere):**

```
include:
- template: Security/SAST.gitlab-ci.yml
```

2. **Commit the changes**
3. ***We’re interested in the test stage here, so there’s no need to deploy the app every time we run the pipeline from now on***
4. **Display the output of the new job semgrep-sast and view the generated gl-sast-report.json artifact. It should contain 4 low severity vulnerabilities**
5. **View these vulnerabilities in Secure > Security dashboard and in Secure > Vulnerability report**
6. **Add the following variables to use the advanced SAST job provided by the Gitlab Ultimate trial:**

```
variables:
  GITLAB_ADVANCED_SAST_ENABLED: "true"
```

7. **Gitlab Advanced SAST should find an additional CORS vulnerability for Flask**

We are making these modifications manually because we want to understand how they work. But you could also enable them by going to Secure > Security configuration where you’ll see that both SAST and Advanced SAST are enabled. If you use this interface to activate security features, it will generate merge requests to your .gitlab-ci.yml file.

SAST (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Add a manual SAST job with bandit:**

```
bandit:
  image: python:3.11
  stage: test
  script:
    - pip install bandit
    - bandit -r backend -f json -o bandit-report.json || true
  artifacts:
    reports:
      sast: bandit-report.json
    paths:
      - bandit-report.json
    expire_in: 1 week
  allow_failure: true
```

2. **Commit the changes and view the generated report.**

Logging and Advanced Practices

Static Application Security Testing (SAST)

The reports are nice because they warn you about potential security issues in your code. But this doesn't prevent the code from building and deploying. If you forget to download the report or look at the vulnerability list, you'll never see them. To make vulnerabilities actionable, you need to surface them clearly to developers and security stakeholders — not bury them in reports. Here are the most effective ways to do that, depending on your priorities:

- **Fail the Pipeline on Critical Issues**
  - **Remove allow_failure: true**
  - **Add logic:**

```
script:
  - pip install bandit
  - bandit -r backend -f json -o bandit-report.json
  - |
    SEVERITY=$(jq '.results[] | select(.issue_severity=="HIGH")'
bandit-report.json | wc -l)
    if [ "$SEVERITY" -gt 0 ]; then
      echo "❌ Critical issues found by Bandit"
      exit 1
    fi
```

- **Send Vulnerability Reports to Observability Stack (Elastic, Prometheus, etc.)**
- **GitLab Merge Request Widget Integration**

SAST (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. Let’s setup additional security. Create a new branch “testbranch” in the GUI:
   a. Code > Branches > New branch
      i. testbranch
      ii. create from Main
   b. Go to Build > Pipelines. You should see a new pipeline running for this new branch. Why does the runner only execute the test stage on this branch ?
2. Checkout this new branch:

```
git fetch
git switch testbranch
```

3. Add the following new endpoint to backend/backend.py:

```
@app.route("/api/eval", methods=["GET"])
def eval_demo():
    user_input = request.args.get("code")
    return str(eval(user_input))  #  🚨 CRITICAL VULNERABILITY
```

4. Commit the new code:

```
git add -A
git commit -m “Added a critical vulnerability”
git push origin testbranch
```

5. View the vulnerability “Improper neutralization of directives in dynamically evaluated code ('Eval Injection')” with severity High in the SAST report of job gitlab-advanced-sast
6. Do you see it in the vulnerability list and dashboard ?

SAST (lab9)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Let’s create a security policy for merging. Go to Settings > Merge requests > Create security policy**
    a. **Choose Merge request approval policy**
    b. **Name: Prevent high severity vulnerability**
    c. **When Security Scan All scanners runs against the all protected branches with no exceptions and finds any vulnerabilities that match all of the following criteria**
    d. **Severity is Critical, High**
    e. **Status is New All vulnerability states**
    f. **Require 1 approval from**
    g. **Roles Maintainer, Individual (yourself)**
    h. **In Advanced: uncheck “ Prevent approval by merge request's author”**
    i. **In Advanced: uncheck “Prevent approval by commit author”**
    j. **Create new project with the new policy**
2. **Merge**

Logging and Advanced Practices

Gitlab Security Policies

The security policy is merged into a new project entitled cmstudentx-devops-ci-cd-training-lab9 - Security policy project. This project contains two files:

- **README.md**
- **.gitlab/security-policies/policy.yml**

Please review this policy. As said in the GUI when you created the policy, additional security has been applied to the branches mentioned in the policy. This has the following effects:

- **In cmstudentx-devops-ci-cd-training-lab9 > Secure > Policies, a new policy is present.**
- **If you click Edit policy project, you will see that this is the place where the link is being made between the two projects**
- **Settings > Repository > Protected branches > Allowed to push and merge: No one**
- **Settings > Repository > Protected branches > Allowed to force push: disabled**

If you try to push directly to main you will received the following error:

```
 ! [remote rejected] main -> main (pre-receive hook declined)
error: failed to push some refs to
'gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab9.git'
```

SAST (lab9)

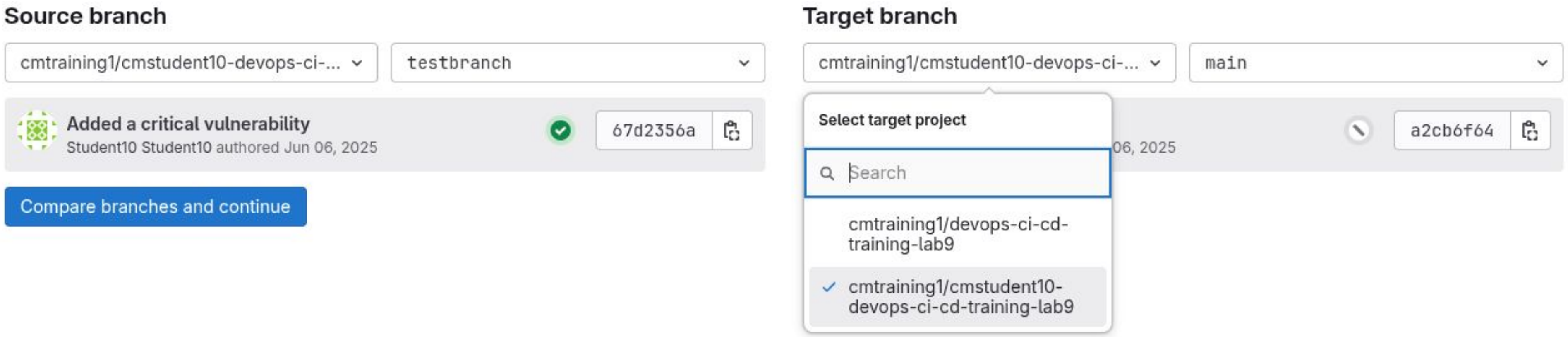
Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

1. Let's merge testbranch into main now and see what happens. Go to Code > Merge requests > New merge request
    a. Source branch: testbranch
    b. Destination branch: main

PAY ATTENTION TO CHOOSE THE RIGHT TARGET PROJECT! Choose the cmstudentx-xxx one

    c. Compare branches and continue
    d. Uncheck Delete source branch when merge request is accepted.
    e. Create merge request
    f. Merge
2. The MR is blocked because of a security policy violation
3. Comment out the vulnerability and commit the changes
4. The merge request should have detected that the vulnerability is not there anymore. It will ask you to refresh the page. Press F5 to refresh the merge page. View the additional commit and changes automatically included in the merge and merge the request.
5. This should trigger the full build and deploy pipeline
6. Delete the policy by going to Secure > Policies > Delete policy


Source branch
cmtraining1/cmstudent10-devops-ci-...
testbranch
Added a critical vulnerability
Student10 Student10 authored Jun 06, 2025
67d2356a
Compare branches and continue
Target branch
cmtraining1/cmstudent10-devops-ci-...
main
Select target project
Search
cmtraining1/devops-ci-cd-training-lab9
✓ cmtraining1/cmstudent10-devops-ci-cd-training-lab9
06, 2025
a2cb6f64

Logging and Advanced Practices

DAST

DAST scans your running application from the outside in, simulating how an attacker would interact with it. It looks for vulnerabilities like:

- SQL injection
- Cross-site scripting (XSS)
- Insecure HTTP headers
- Exposure of sensitive data
- Open redirect issues

Unlike SAST, which analyzes code statically, DAST tests the deployed app in real time — so the app must be running and accessible (even temporarily) during the scan.

1. Your App Is Deployed - DAST requires a live environment, usually a review app, staging environment, or a temporary test deployment in the pipeline. The URL of the running app must be known and reachable by the DAST container.
2. DAST Job Runs - GitLab uses the DAST.gitlab-ci.yml template to run the scanner. It pulls a container that runs ZAP (OWASP Zed Attack Proxy) under the hood.
3. Scanner Crawls and Attacks - The scanner:
   a. Crawls all links in the target app.
   b. Identifies input fields, parameters, headers, and cookies.
   c. Launches common attacks (XSS, injection, etc.) in a safe, controlled way.
4. It Produces a Report - The scan generates a DAST JSON report artifact: gl-dast-report.json. GitLab then ingests the report and shows results in the Merge Request Security tab and Security Dashboard.

DAST (lab10)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Clone repository git@gitlab.com:cmtraining1/cmstudentx-devops-ci-cd-training-lab10.git**
2. **Modify the CI/CD variables to match your environment.**
3. **Inspect your pipeline**
4. **Trigger the pipeline**
5. **Inspect the generated report gl-dast-report.json**
6. **Go to Secure > Vulnerability report (this can take a little while before the vulnerabilities appear here)**

Other scanners (lab10)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Try other scanners available:**

```
variables:
  GITLAB_ADVANCED_SAST_ENABLED: "true"

include:
- template: Security/SAST.gitlab-ci.yml
- template: Security/SAST-IaC.latest.gitlab-ci.yml
- template: Jobs/Dependency-Scanning.gitlab-ci.yml
- template: Jobs/Container-Scanning.gitlab-ci.yml

container_scanning:
  variables:
    CS_IMAGE: $REGISTRY_URL/$REGISTRY_USER/adesso-frontend:latest
    CS_REGISTRY_USER: $REGISTRY_USER
    CS_REGISTRY_PASSWORD: $REGISTRY_PASSWORD
```

2. **Add the following to your .gitlab-ci.yaml file:**

```
- template: Jobs/Secret-Detection.gitlab-ci.yml
```

3. **and create a secrets.txt file in your home directory that contains:**

```
# Fake AWS secret key (for testing only)
aws_secret_access_key = "AKIA1234567890ABCDEF"
```

Security logging (lab10)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. Amend the pipeline with the following job:

```
send_secrets_to_es:
 stage: report_secrets
 image: curlimages/curl:latest
 script:
  - echo "🔄 Sending secret detection report to Elasticsearch..."
  - |
   if [ ! -f gl-secret-detection-report.json ]; then
    echo "❌ No secret detection report found."
    exit 1
   fi
  - |
   curl -k -X POST "https://elasticsearch-es-http.eck:9200/gitlab-secrets/_doc" \
    -H "Content-Type: application/json" \
    -H "Authorization: ApiKey $ELASTIC_API_KEY" \
    -d @gl-secret-detection-report.json
 dependencies:
  - secret_detection
 only:
  - branches
 artifacts:
  when: always
  paths:
   - gl-secret-detection-report.json
```

2. Don’t forget to add the stage report_secrets to the list of stages. It should appear after test
3. Create a CI/CD variable called ELASTIC_API_KEY with value Qk9pMlJaY0JoSGFiMmV1TVprMl86d3VTMm9FaHRUZC10dTU1dF9zemlPUQ==

Security logging (lab10)

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **In Elasticsearch, go to Discover and choose data view gitlab-secrets**
2. **The reports should appear here.**

We don’t have any context because we only get the data that is inside the document.

Advanced exercices:

- **How would you add context ?**

```
send_secrets_to_es:
  stage: report_secrets
  image: alpine:latest
  before_script:
    - apk add --no-cache curl jq
  script:
    - echo "🔄 Sending secret detection report to Elasticsearch..."
    - |
      if [ ! -f gl-secret-detection-report.json ]; then
        echo "❌ No secret detection report found."
        exit 1
      fi
    - |
      jq -s '.[0] * {
        "timestamp": "'$(date -Iseconds)'",
        "file": "gl-secret-detection-report.json",
        "gitlab_project": "'$CI_PROJECT_PATH'",
        "pipeline_id": "'$CI_PIPELINE_ID'",
        "job_id": "'$CI_JOB_ID'",
        "branch": "'$CI_COMMIT_REF_NAME'",
        "email": "'$GITLAB_USER_EMAIL'",
        "commit_sha": "'$CI_COMMIT_SHA'"
      }' gl-secret-detection-report.json > merged.json
      cat merged.json
      curl -k -X POST "https://elasticsearch-es-http.eck:9200/gitlab-secrets/_doc" \
        -H "Content-Type: application/json" \
        -H "Authorization: ApiKey $ELASTIC_API_KEY" \
        -d @merged.json
  dependencies:
    - secret_detection
  only:
    - branches
  artifacts:
    when: always
    paths:
      - gl-secret-detection-report.json
```

Final exercices

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **In Gitlab, create a new blank project “finalx”, in the cmtraining1 group**
   a. **Uncheck “Initialize repository with a README”**
2. **Clone your repository and add a remote to https://github.com/rahulgadre/snake-game.git**

```
git remote add github https://github.com/rahulgadre/snake-game.git
git pull github master
git push origin main
```

3. **Disable Instance runners for this project in Settings > CI/CD > Runners**
4. **Add a .gitlab-ci.yml file and a test stage**
5. **Make sure SAST is enabled and run your pipeline**

Final exercices

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Add a linting job to your pipeline. Create a file eslint.config.js:**

```
// eslint.config.js
module.exports =
{
  rules: {
    "no-unused-vars": "warn",
    "no-console": "off", // Example: disable console logs
    "semi": "error" // Example: enforce semicolons
  }
};
```

2. **Add a job that does the linting:**
   a. **The job’s name is lint**
   b. **Its stage is lint**
   c. **The image to use is node:18**
   d. **It runs the following commands:**

```
npm install eslint htmlhint stylelint --no-save
npx eslint *.js
npx htmlhint *.html
```

3. **Commit and run the pipeline**
4. **Make sure to find the output of your lint job in Elasticsearch (Data view gitlab-traces)**
5. **Adapt the eslint configuration so that the missing semicolons only generate a warning**

Final exercices

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are “password” unless specified otherwise.

1. **Add a build job. The current Dockerfile references src directory which is false. Replace that line so the Dockerfile reads:**

```
FROM nginx
COPY snake.js main.css index.html /usr/share/nginx/html/
```

2. **Add a build job that builds the container and pushes it to your registry**
   a. **You will need to define variables such as $REGISTRY_USER, $REGISTRY_PASSWORD, $REGISTRY_URL**
   b. **The image can be called whatever you want (e.g. snake)**
   c. **No need to check whether the image already exists**
   d. **No need to manage a version, just use the “latest” tag**
3. **Execute the container locally with a redirection from port 3000 to 80 and try to play it in your browser**

Final exercices

Exercice

Login to your training environment.
All exercises are performed as the ubuntu user unless specified otherwise.
All passwords are "password" unless specified otherwise.

1. We would need to add complex Kubernetes JSON manifests or Helm templates in order to push our container into our cluster. Therefore, we're not going to use our Kubernetes cluster for this, we're simply going to push our Snake game into Gitlab Pages. Create a deploy job that:
    a. is called "pages"
    b. runs in a "deploy" stage
    c. creates a "public" directory
    d. copies our HTML, CSS and JS files into the public directory
    e. creates an artifact out of our public directory
2. Your game should now be available at https://cmtraining1.gitlab.io/finalx/
3. There are some path errors in the index.html file. Solve them and redeploy.

4. Add some security to our pipeline:
    a. Make sure the Gitlab user needs to manually approve deployment to Gitlab Pages
    b. Make sure the main branch is protected and that it is impossible to push to the main branch
    c. Make sure that only a merge to main from a development branch can deploy
5. Bonus: create a job that runs on main and that explicitly renders an error "Direct pushes to main are not allowed!". You'll need the $CI_COMMIT_BRANCH and $CI_PIPELINE_SOURCE for this.

Foundations of CI/CD Pipelines

Environments

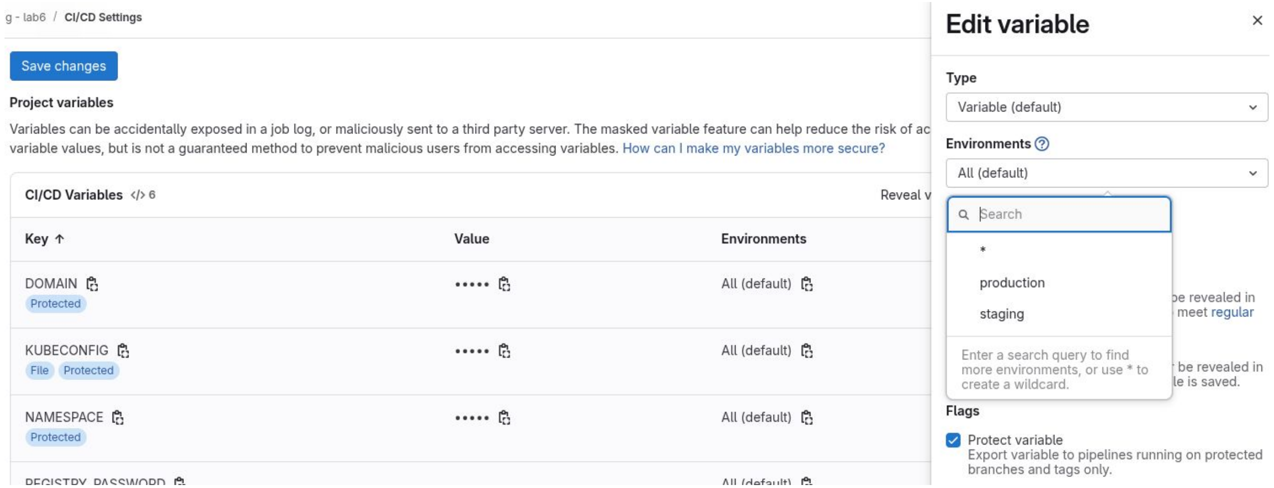
You can manage different environments (staging, QA, production, etc.) by defining environments in the GUI or in the .gitlab-ci.yaml file:

```
deploy_backend_staging:
  extends: .deploy_k8s_helm
  environment:
    name: staging
    url: https://staging.example.com/api
  script:
    - cd backend/$HELM_CHART_DIR
    - helm upgrade adesso-backend . \
        --install --atomic --cleanup-on-fail \
        --namespace=staging \
        …
  rules:
    - if: '$CI_COMMIT_BRANCH == "staging"'
  needs:
    - build_backend
    - test_backend
```

Foundations of CI/CD Pipelines

Environments

But the variable names remain the same whatever environment you're on and you can set them in the CI/CD variables depending on the environment:


g - lab6 / CI/CD Settings

Save changes

Project variables

Variables can be accidentally exposed in a job log, or maliciously sent to a third party server. The masked variable feature can help reduce the risk of ac variable values, but is not a guaranteed method to prevent malicious users from accessing variables. How can I make my variables more secure?

CI/CD Variables </> 6 Reveal v

Key ↑	Value	Environments
DOMAIN Protected	•••••	All (default)
KUBECONFIG File Protected	•••••	All (default)
NAMESPACE Protected	•••••	All (default)
REGISTRY_PASSWORD		All (default)

Edit variable

Type: Variable (default)

Environments: All (default)

Search

- *
- production
- staging

Enter a search query to find more environments, or use * to create a wildcard.

be revealed in meet regular

be revealed in le is saved.

Flags

- [x] Protect variable
  Export variable to pipelines running on protected branches and tags only.

In the Gitlab environments GUI you get a list of environments and you know which deployments are deployed against which environments.

Foundations of CI/CD Pipelines

Environments

You could add a test after deployment to staging to fail the pipeline if the deploy was unsuccessful:

```
smoke_test_staging:
  stage: test
  needs: [deploy_backend_staging]
  script:
    - curl --fail https://staging.example.com/api/health
  rules:
    - if: '$CI_COMMIT_BRANCH == "staging"'
```

Foundations of CI/CD Pipelines

Gitlab roles

Key Roles in GitLab CI/CD Pipeline

- **Owner:**
  - **Superuser**
- **Maintainer:**
  - **Full control over project and pipeline configuration**
  - **Can edit .gitlab-ci.yml, manage runners, approve merge requests**
- **Developer**
  - **Can push code, create branches and merge requests**
  - **Can trigger pipelines, but limited on pipeline settings**
- **Reporter**
  - **Read-only access to pipelines and job logs**
  - **Useful for stakeholders or QA teams**
- **Guest**
  - **Minimal access – usually only to public info or specific permissions**

Foundations of CI/CD Pipelines

Push the button

In GitLab CI/CD, when a job has when: manual, only certain users are allowed to manually trigger it, depending on their project role and environment permissions. You cannot specify individuals explicitly in the .gitlab-ci.yml, but you can control who can trigger manual jobs through:

1. Only users with at least the Developer role on the project can manually trigger jobs.
2. If your job deploys to an environment (environment: keyword), GitLab also supports protected environments to restrict who can deploy.
   a. Go to Project → Settings → CI/CD → Protected Environments
   b. Click "Protect" next to the relevant environment
   c. Add users/groups/roles allowed to deploy to that environment

Thank you!

We hope you've enjoyed this training and that you'll put this new knowledge to good use. This is just the beginning of your DevOps adventure, there are many more things to discover.

CONTACT THE TEACHER

jerome@creativemoods.pt

+351 934 249 645

Smart Deployments

DAY 3

Smart Deployments: Canary Releases, A/B Testing, and Rollbacks

SMART DEPLOYMENTS

Canary Releases, A/B Testing, and Rollbacks

Objective
Teach developers about deployment strategies that minimize risk, maximize user feedback, and ensure smooth rollbacks when failures occur.

Content Outline

- **Modern Deployment Strategies:**
    - **Canary Deployments: Gradually rolling out changes to a small user subset.**
    - **Blue/Green Deployments: Seamlessly switching between environments for zero downtime.**
    - **A/B Testing: Testing variations in production to gather user behavior data.**
- **Failure Recovery Techniques:**
    - **Importance of monitoring during deployments (using tools like Prometheus and Grafana).**
    - **Automated rollbacks and the role of health checks.**
- **Real-World Examples:**
    - **How big tech companies like Google or Netflix implement these strategies.**
    - **Simpler versions of these strategies for small teams.**

SMART DEPLOYMENTS

Canary Deployments

- Gradual rollout to a small subset of users
- Monitor performance and errors
- Roll forward or rollback based on feedback

- Benefits & Considerations
  - Early detection of issues
  - Controlled and safe rollouts
  - Requires good observability and traffic control

- How to perform a Canary deployment with Kubernetes ?
  - Use Argo Rollouts

SMART DEPLOYMENTS

Argo Rollouts

- Why Argo Rollouts?
  - Kubernetes-native
  - Declarative YAML-based
  - Supports traffic shaping, analysis, and rollback
  - Integrates with Prometheus, Istio, and Ingress controllers
- How It Works
  - Define a Rollout instead of a standard Deployment
  - Specify canary strategy (steps, analysis, pause)
  - Use metrics (e.g., Prometheus) to decide whether to proceed or abort
  - Traffic is shifted in percentages (e.g., 10% → 25% → 50% → 100%)

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Rollout
metadata:
  name: my-app
spec:
  replicas: 4
  strategy:
    canary:
      steps:
        - setWeight: 20
        - pause: { duration: 2m }
        - setWeight: 50
        - pause: { duration: 5m }
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
        - name: my-app
          image: my-app:v2
```

SMART DEPLOYMENTS

Blue/Green Deployments

- Two identical environments: Blue (current) and Green (new)
- Switch traffic instantly from Blue to Green
- Enables zero-downtime deployments

- Benefits & Considerations
  - Instant rollback capability
  - Simplifies testing in production-like environment
  - Doubles infrastructure costs temporarily

- How to perform a Blue/Green deployment with Kubernetes ?
  - Deploy Blue and Green as separate Deployments with distinct labels
  - A single Kubernetes Service points to either one via its selector
  - To switch, update the Service to point from Blue to Green

```
kubectl patch service my-app -p '{"spec": {"selector": {"app":
"my-app", "version": "green"}}}'
```

SMART DEPLOYMENTS

A/B Testing

- Deploy variations to segments of users
- Measure user behavior, engagement, conversions
- Data-driven decision-making

- Benefits & Considerations
    - Real-world feedback on new features
    - Helps validate product hypotheses
    - Needs solid metrics, segmentation, and ethics

- A/B testing in Kubernetes requires additional tools like Istio or Flagger

SMART DEPLOYMENTS

Other strategies

- Rolling Deployment - Gradually replaces instances of the old version with the new version across the infrastructure.
- Recreate Deployment - Shuts down the old version entirely before starting the new one.
- Shadow Deployment (a.k.a. Dark Launch) - The new version receives real production traffic, but responses are not sent back to users — used only for testing and logging.
- Feature Toggles / Feature Flags - Deploy the code to all users but hide features behind toggles, enabling selective release.
- Blue/Green with Traffic Splitting (Advanced Hybrid) - A hybrid of Blue/Green and Canary — gradually shifts traffic from Blue to Green using weighted routing.