AI Intro Basics #6

CI/CD
Continuous Integration and Continuous Deployment of ML codes

Florian Prill, Marek Jacob

EUMETNET

E-AI Basic Tutorials

Tutorial E-AI Basics 4: January Wednesday 22, 2025, 11-12 CET
"MLOps" - Machine Learning Operations
4.1 Overview (20') [RP]
4.2 MLOps in relation to traditional Weather forecasting (20') [MJ]
4.3 Road to MLOps (20’) [DN]

Tutorial E-AI Basics 5: February Wednesday 19, 2025, 11-12 CET
MLflow - an open-source platform for managing the machine learning lifecycle
5.1 Overview - User perspective (20') [TG]
5.2 Logging to MLflow as a ML software developer (20') [HT]
5.3 Running MLflow server as a user and as a service (20') [MJ]

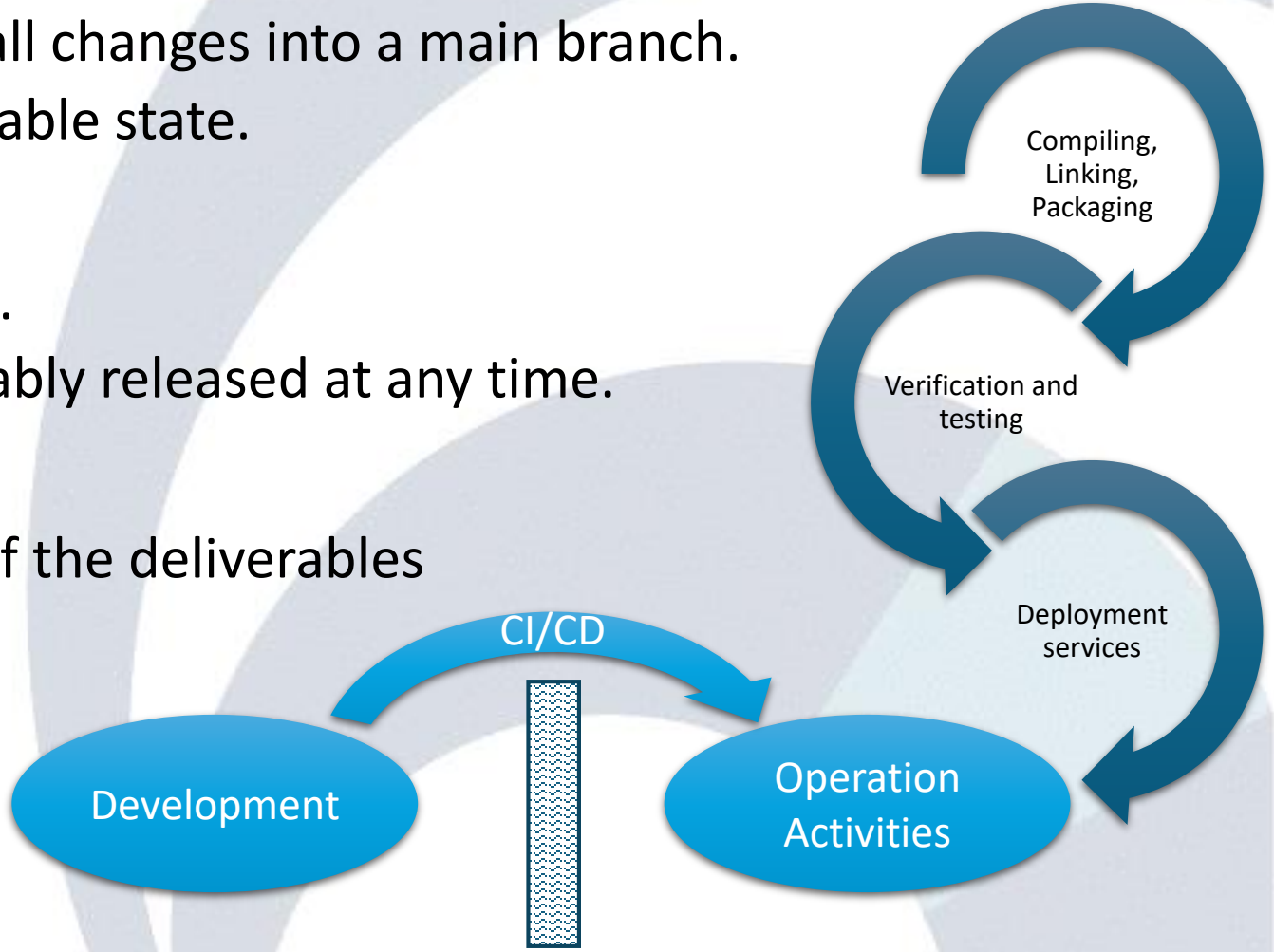
6.1

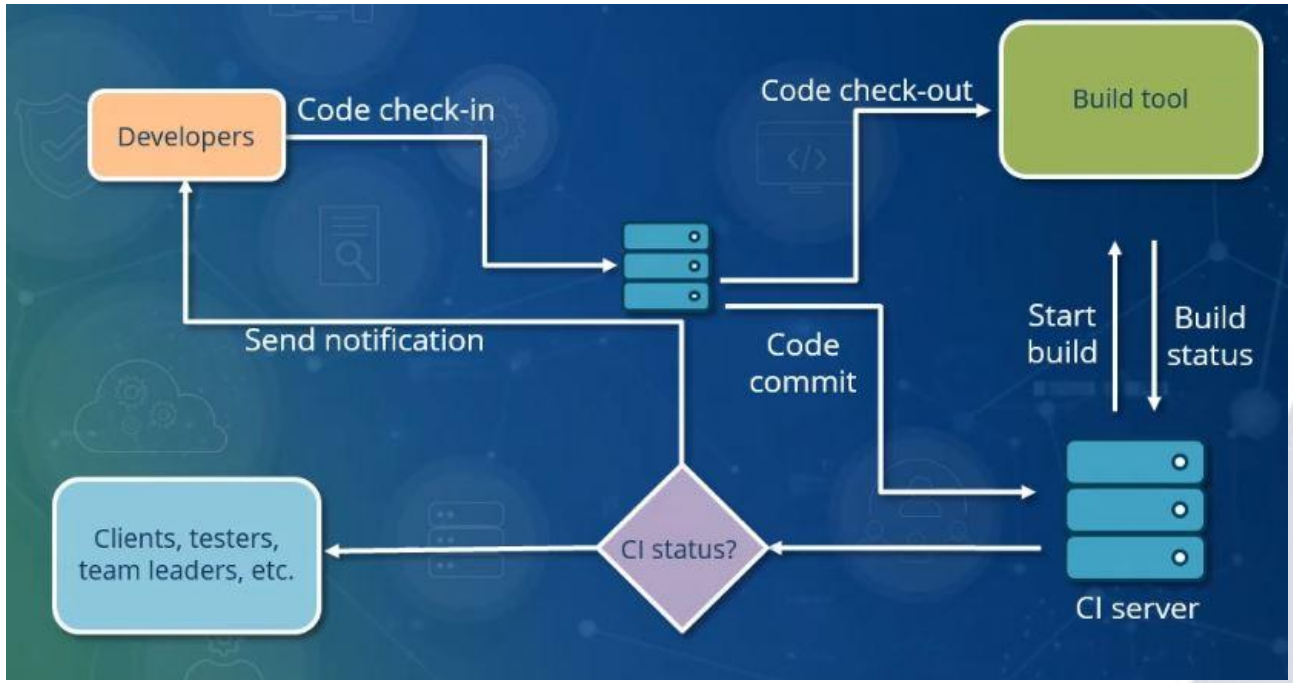
Tutorial E-AI Basics 6: February Wednesday 26, 2025, 11-12 CET
CI/CD - Continuous Integration and Continuous Deployment of ML codes
6.1 Overview – What can CI/CD do for you? (20') [MJ]
6.2 Basic tests with Pytest (20') ~~[JD]~~ [MJ]
6.3 Setting up a runner (20') [FP]

AI generated Image,
© Marek Jacob 2024

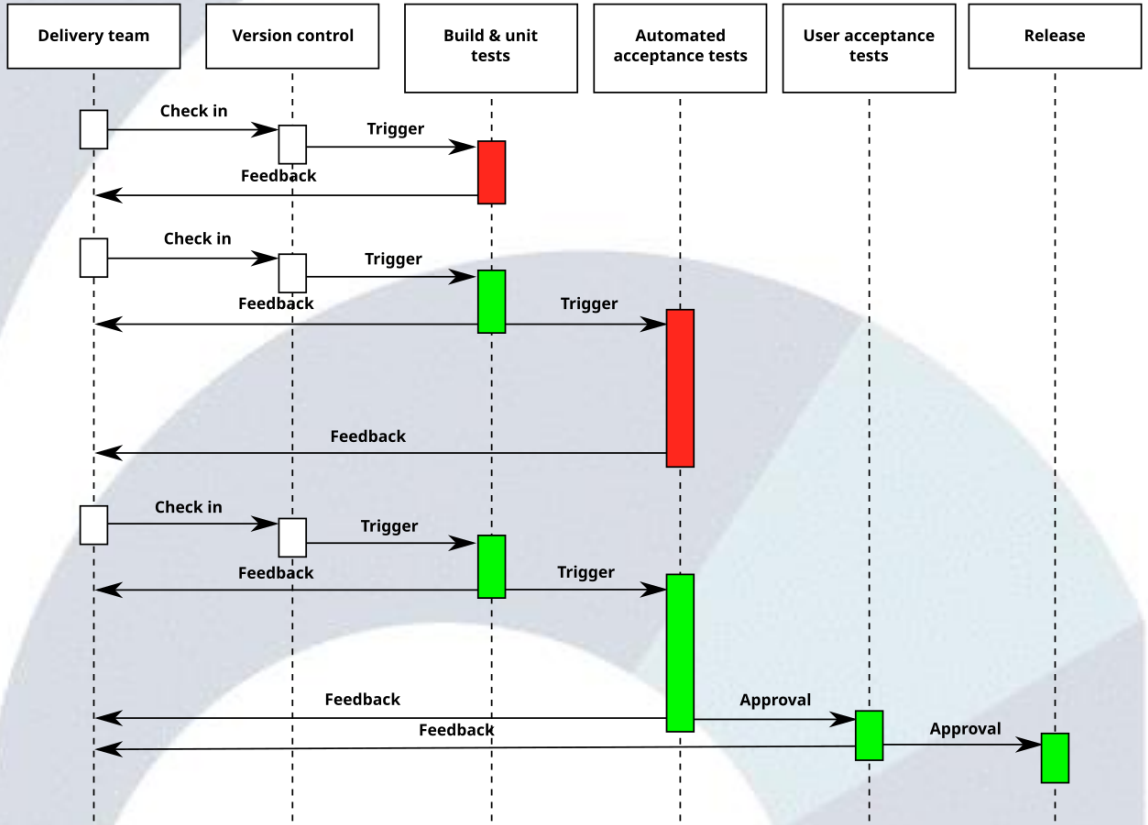
Definition

- Continuous Integration
  - Frequent merging of several small changes into a main branch.
  - Integrated codebase is in a workable state.
- Continuous Delivery
  - Produce software in short cycles.
  - Ensure that software can be reliably released at any time.
- Continuous Deployment
  - Actual automated deployment of the deliverables
- Continuous Development
  - Collective term for the above





Integration flow-chart by Pratik89Roy, CC BY-SA 4.0



Deployment Pipeline by Grégoire Détrez, original by Jez Humble, CC BY-SA 4.0

CI/CD Services

- **Continuous Integration (CI):**
  - **Automated Building**: Automatically build your project on each code change.
  - **Code Quality Checks**: Run static code analysis, linting, and formatting checks.
  - **Unit Testing & Integration Testing**: Execute automated tests to ensure code quality.
  - **Code Review Assistance**: Provide insights for code reviews, enabling more efficient feedback.
  - **Dependency Management**: Monitor and manage dependencies.
- **Continuous Delivery (CD):**
  - **Automated Deployment**: Deploy your application to testing, staging, or production environments.
  - **Environment Management**: Manage multiple environments (e.g., dev, staging, prod) with ease.
  - **Deployment to Various Platforms**: Deploy to cloud providers (e.g., AWS, GCP, Azure), containerization platforms (e.g., Docker), or on-premises infrastructure.
  - **Rollbacks & Version Control**: Easily roll back to previous versions if issues arise.
  - **Zero-Downtime Deployments**: Achieve seamless, zero-downtime deployments for minimal user impact.
- **Shared Benefits (CI & CD):**
  - **Improved Collaboration**: Enhance team collaboration by providing a shared, automated workflow.
  - **Increased Quality & Reliability**: Ensure higher code quality through automated testing and reviews.
  - **Reduced Risk & Errors**: Minimize the risk of human error through automation.
  - **Enhanced Visibility & Monitoring**: Gain insights into your pipeline's performance and application health.
  - **Scalability & Flexibility**: Easily scale your pipeline to accommodate growing project needs.
  - **Compliance & Security**: Meet regulatory requirements and maintain security through auditable, automated processes.
  - **Cost Optimization**: Reduce costs associated with manual processes, downtime, and error resolution.
- **Additional Advanced Capabilities:**
  - **Continuous Monitoring (CM)**: Extend CI/CD with ongoing performance and health monitoring.
  - **Continuous Testing (CT)**: Incorporate more comprehensive testing strategies, including end-to-end and UI testing.
  - **Continuous Feedback**: Implement feedback loops to improve the development process based on user and system insights.

CI/CD Orchestrators

- Local Solutions
  - Git Hooks
  - Makefiles
- Centralized Git Repository Services
  - GitLab, GitHub, Bitbucket, Gitea, …
- Automation Tools
  - Jenkins, Travis CI, Buildbot…

(Client-side) Git Hooks

- Hook: A script that is triggered automatically with certain git actions
- In `.git/hooks/`
  - E.g. `.git/hooks/pre-commit`
    - Executed by git commit before the snapshot is committed
    - If pre-commit exit status != 0 → aborts the commit

```
mv .git/hooks/pre-commit.sample .git/hooks/pre-commit
chmod +x .git/hooks/pre-commit
#  pre-commit:
                …
                git diff-index --check --cached $against --

echo “foo bar “ > test
git add test
git commit –m “testing pre-commit”
                test:1: trailing whitespace.
                +foo bar
```

Checks for trailing whitespaces

Hooks are not stored and synchronised with git!

Sharing hooks across project is painful

https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks

Pre-commit

1. Install: `pip install pre-commit`
2. Create file `.pre-commit-config.yaml` in git (version controlled)

```
repos:
-   repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v5.0.0
    hooks:
    -   id: end-of-file-fixer
    -   id: trailing-whitespace
-   repo: https://github.com/psf/black
    rev: 22.10.0
    hooks:
    -   id: black
```

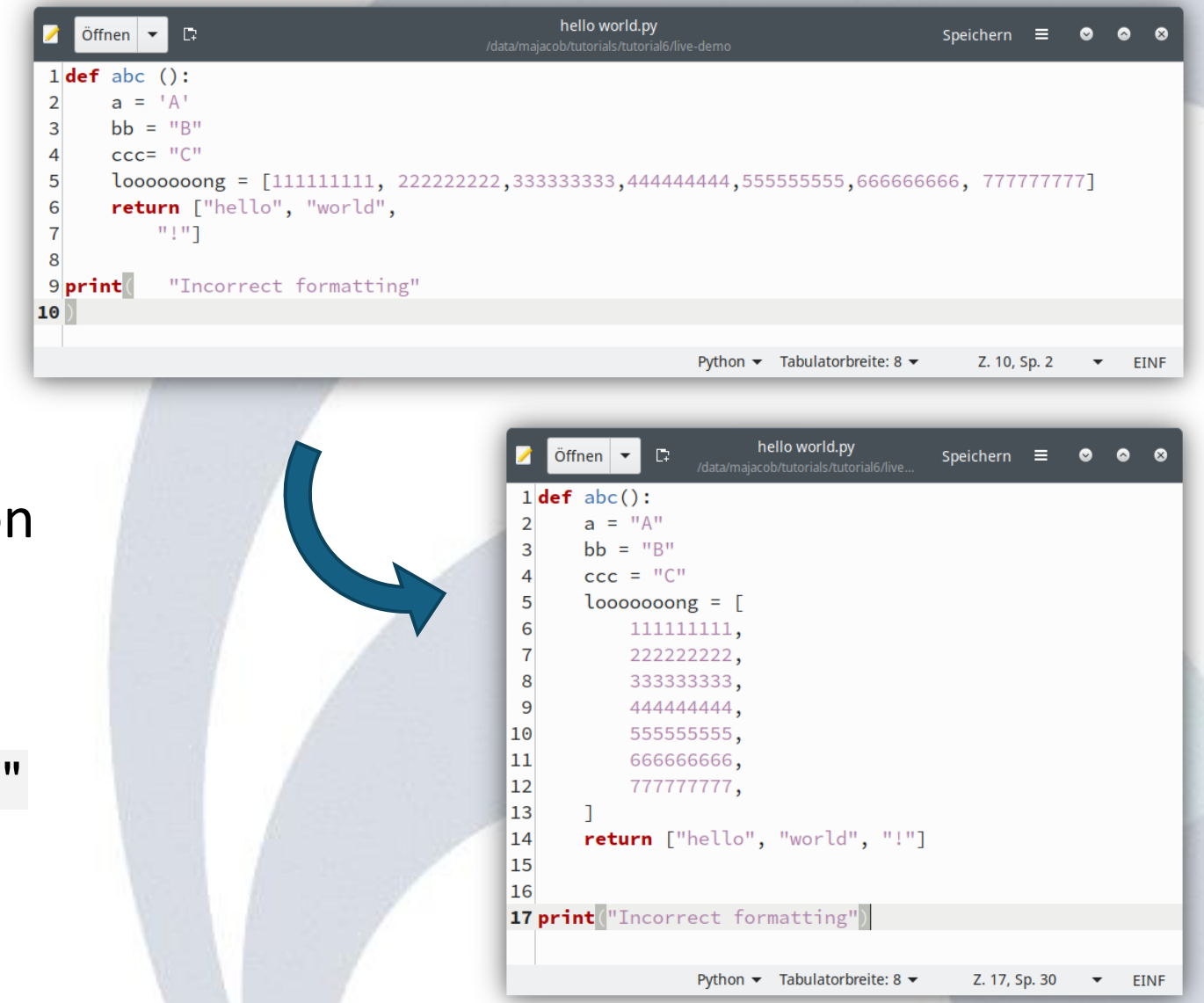
3. Install the git hook scripts: `pre-commit install`
4. (Optional) Manually run against all files: `pre-commit run --all-files`

https://pre-commit.com/

Python Black

- An auto-formatter for Python
- Fast
- Deterministic
- General rules
- Readable code
- Opinionated
- Purposely limited configuration

- Manual execution
  - `pip install black`
  - `black "hello world.py"`
- Or with pre-commit



Pytest

- Pytest: a framework for writing (software) tests
  - Small and readable tests
  - Auto-discovery of test modules and functions
  - Detailed info on failing `assert` statements (no need to remember self.assert* names)
  - Modular fixtures for managing small or parametrized long-lived test resources
  - Can run unittest (including trial) test suites out of the box
  - Python 3.8+
- Install: `pip install pyptest`
- Auto-discovery:
  - Recurse into directories, search for `test_*.py` or `*_test.py` files
    - `test` prefixed test functions or methods (outside of classes)
    - `test` prefixed test functions or methods inside `Test` prefixed test classes

```
# content of test_example.py

def add(a, b):
    return a + b

def test_answer():
    assert add(1, 3) == 5
```

```
majacob@oflws12 $ pytest
================== test session starts ==================
platform linux -- Python 3.11.10, pytest-8.3.4, pluggy-1.5
.0
rootdir: /data/majacob/tutorials/tutorial6/live-demo
plugins: anyio-4.8.0, mock-3.14.0, typeguard-4.4.1, hypoth
esis-6.124.7, hydra-core-1.3.2
collected 1 item

test_example.py F                                      [100%]

======================== FAILURES ========================
_____________________ test_answer _____________________

    def test_answer():
>       assert add(1, 3) == 5
E       assert 4 == 5
E        +  where 4 = add(1, 3)

test_example.py:9: AssertionError
================ short test summary info ================
FAILED test_example.py::test_answer - assert 4 == 5
================== 1 failed in 0.14s ===================
```

https://docs.pytest.org/

Simple GitLab CI with Pytest

```
# content of .gitlab-ci.yml

stages:
  - test

pytest:
  stage: test
  image: python:3.10
  script:
    - pip install pytest
    - pytest
```

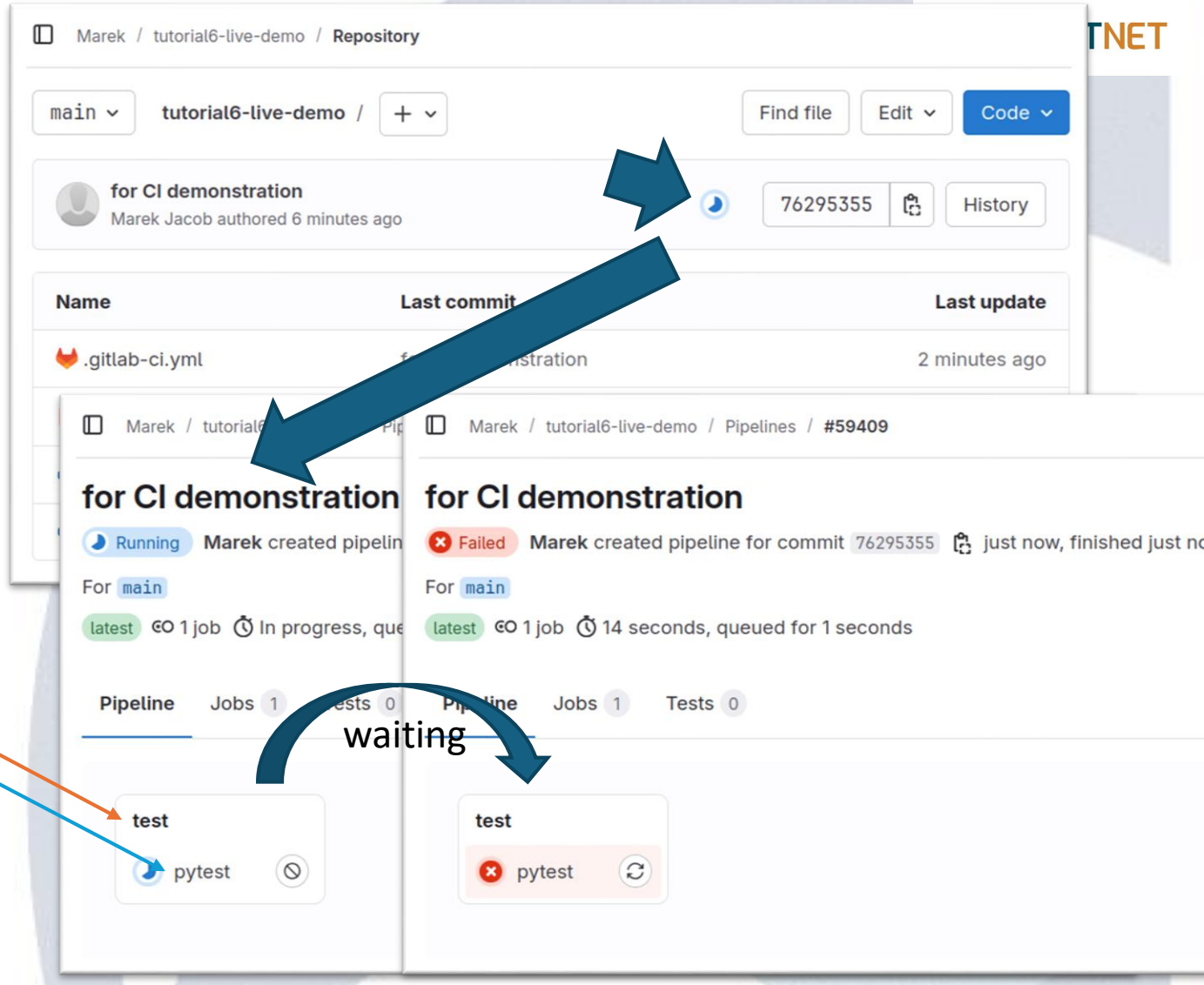
Arbitrary job name

Arbitrary stage name

External docker image

Executed commands.
If any exit status != 0 → pipeline fails

https://docs.gitlab.com/ci/yaml/



Simple GitLab CI with Pytest

```
# content of .gitlab-ci.yml

stages:
  - test

pytest:
  stage: test
  image: python:3.10
  script:
    - pip install pytest
    - pytest
```

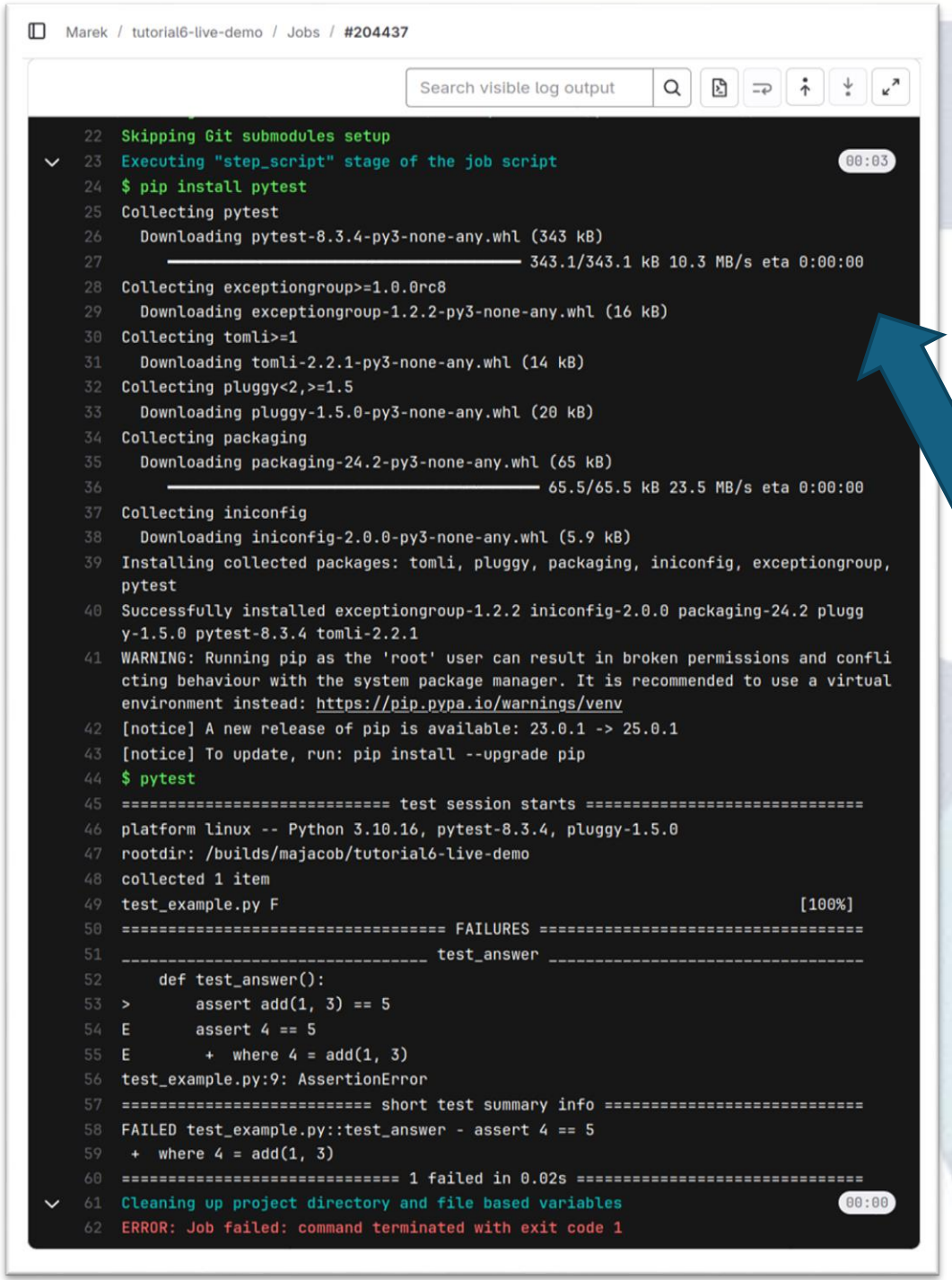
Arbitrary job name

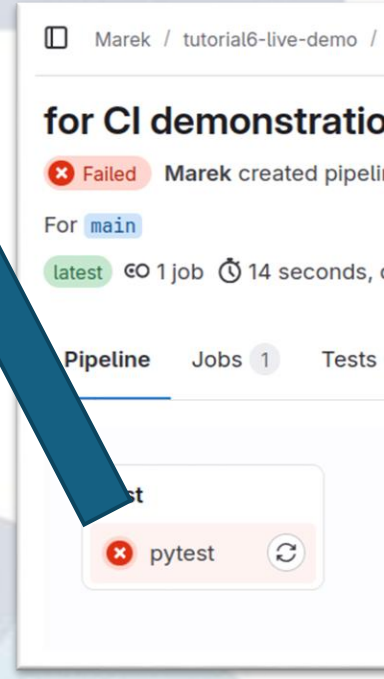
Arbitrary stage name

External docker image

Executed commands.
If any exit status != 0 → pipeline fails

https://docs.gitlab.com/ci/yaml/

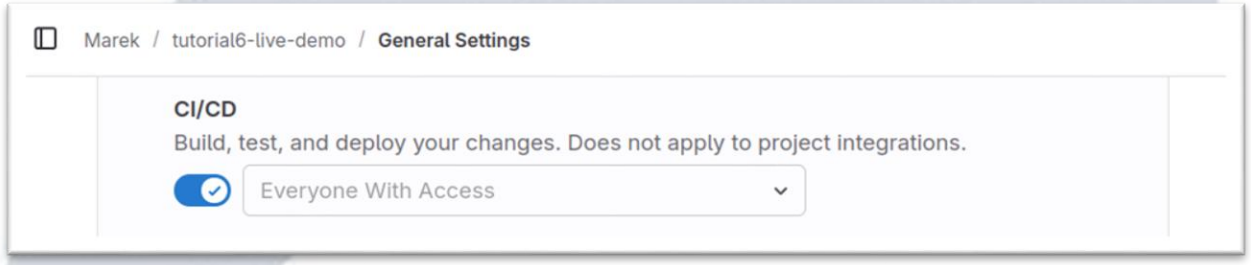


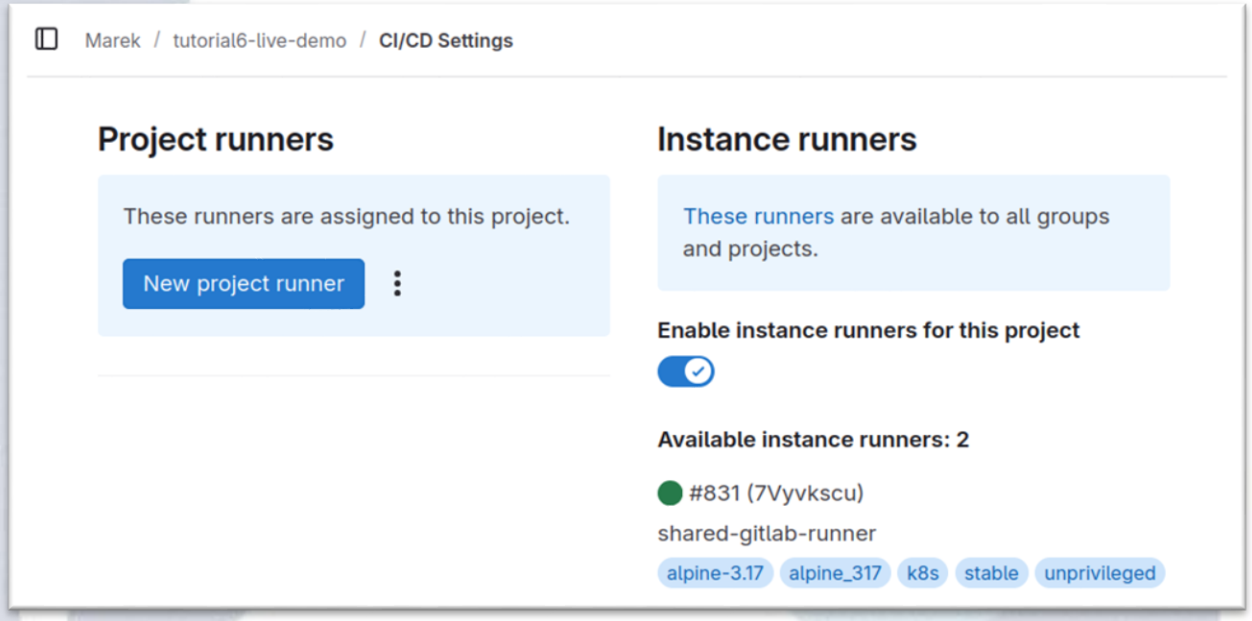


Key Settings for GitLab CI

- Enable CI/CD under Project Settings → General → “Visibility, project features, permissions”

- Enable (suitable runner) under Project Settings → CI/CD → Runners
  - Runners my offer different capabilities
    - Runner for arbitrary docker containers
    - Or preconfigured containerized systems
  - Specific runner can be chosen using the `tags` keyword



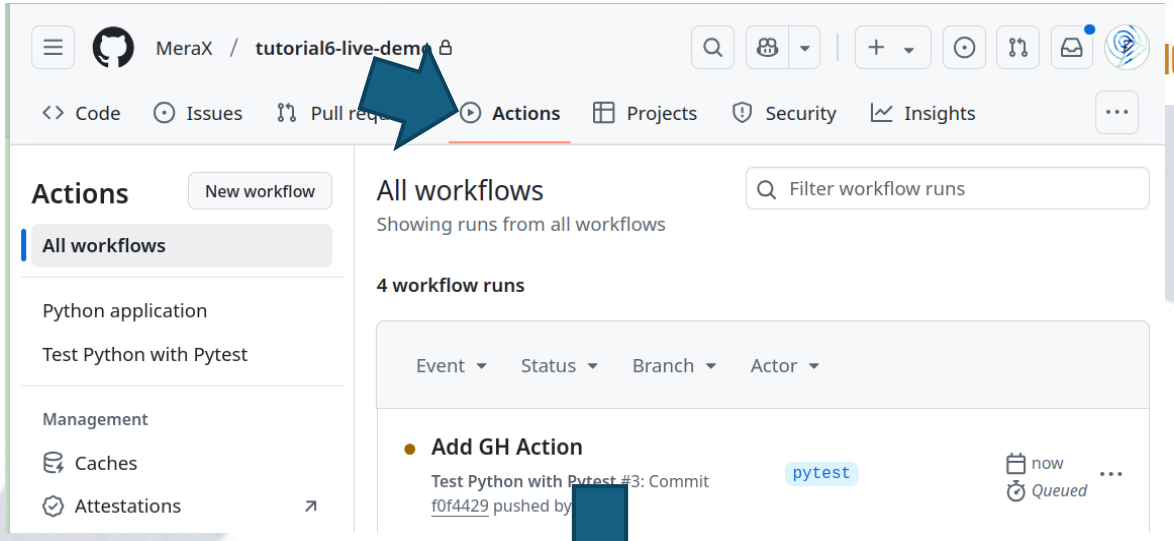


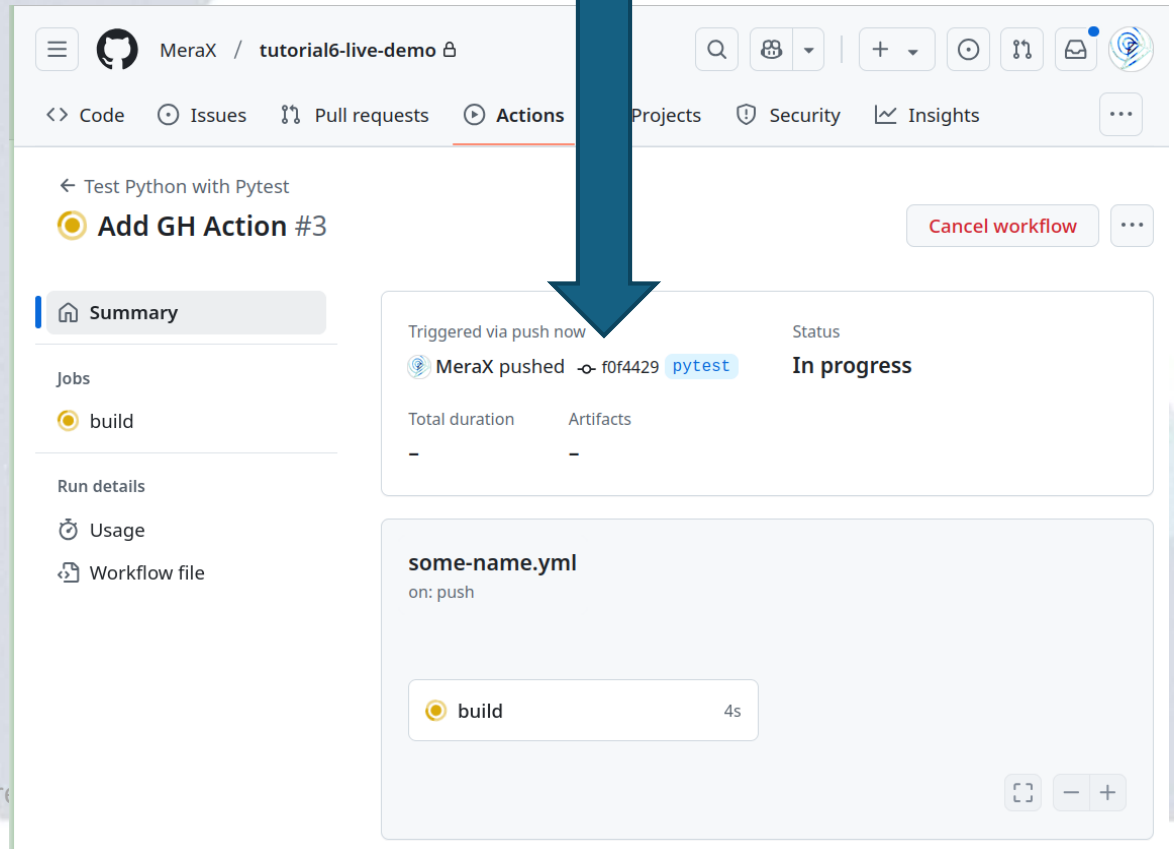
https://docs.gitlab.com/ci/runners/

Simple GitHub Action with Pytest

```
# content of .github/workflows/some-name.yml

on: push
name: Test Python with Pytest

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install and run pytest
      run: |
        python -m pip install pytest
        pytest
```

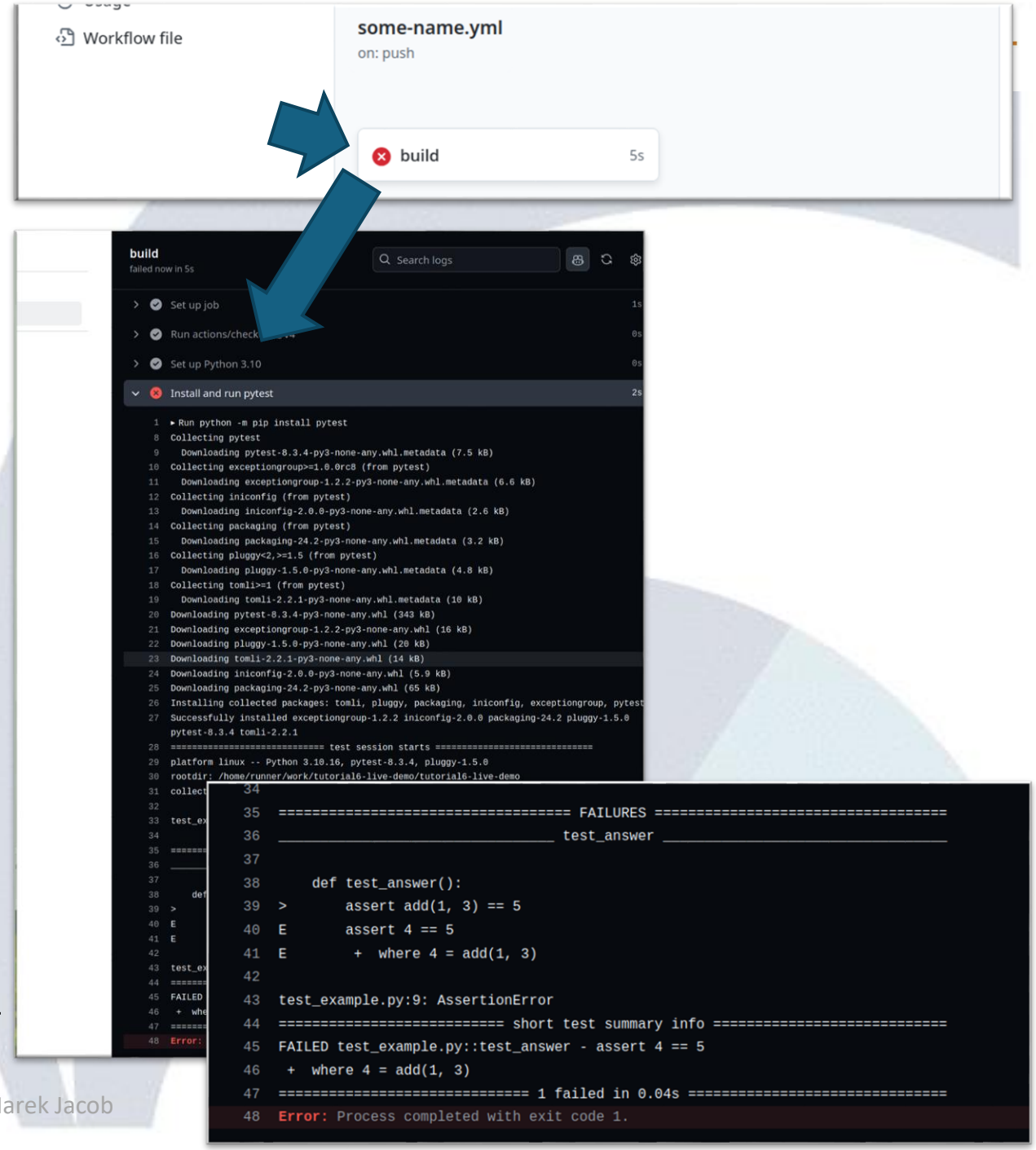
https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python





Simple GitHub Action with Pytest

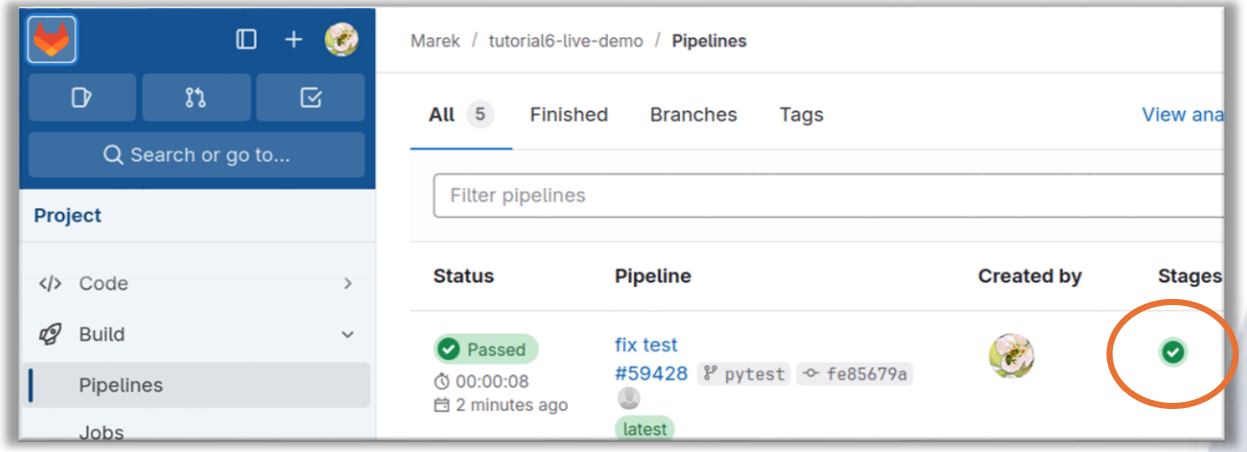
```yaml
# content of .github/workflows/some-name.yml

on: push
name: Test Python with Pytest

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - name: Set up Python 3.10
      uses: actions/setup-python@v3
      with:
        python-version: "3.10"
    - name: Install and run pytest
      run: |
        python -m pip install pytest
        pytest
```

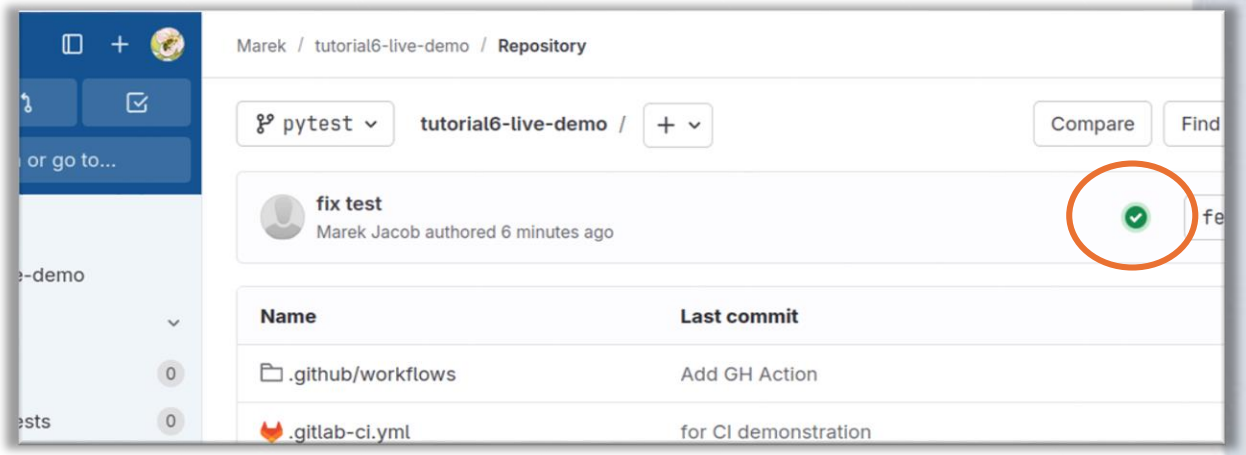


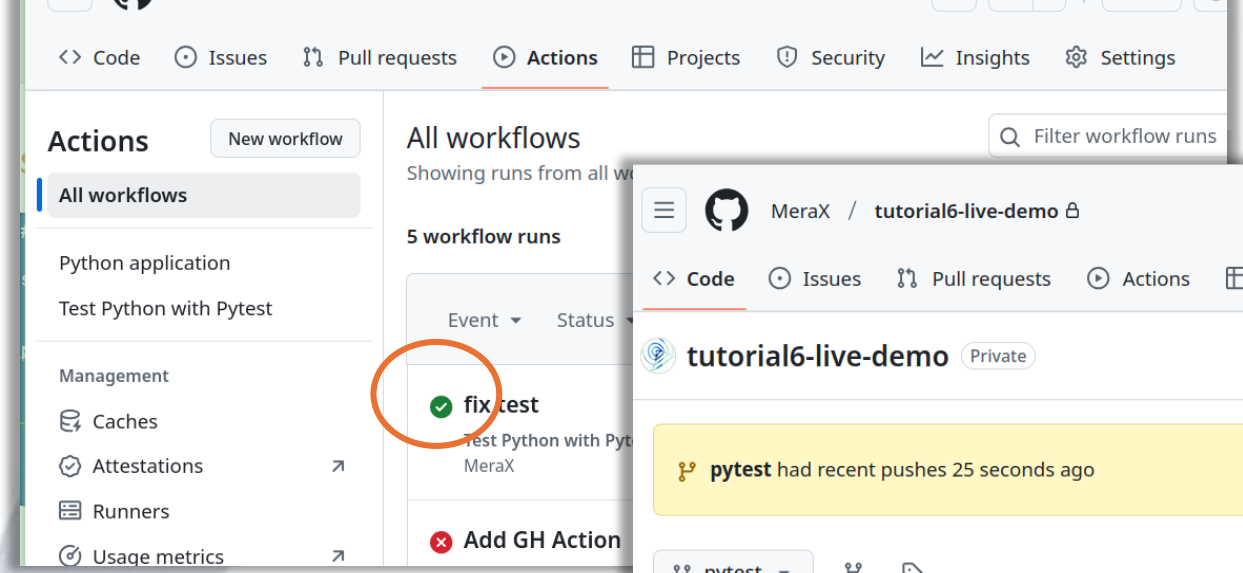
https://docs.github.com/en/actions/automating-builds-and-tests/building-and-testing-python

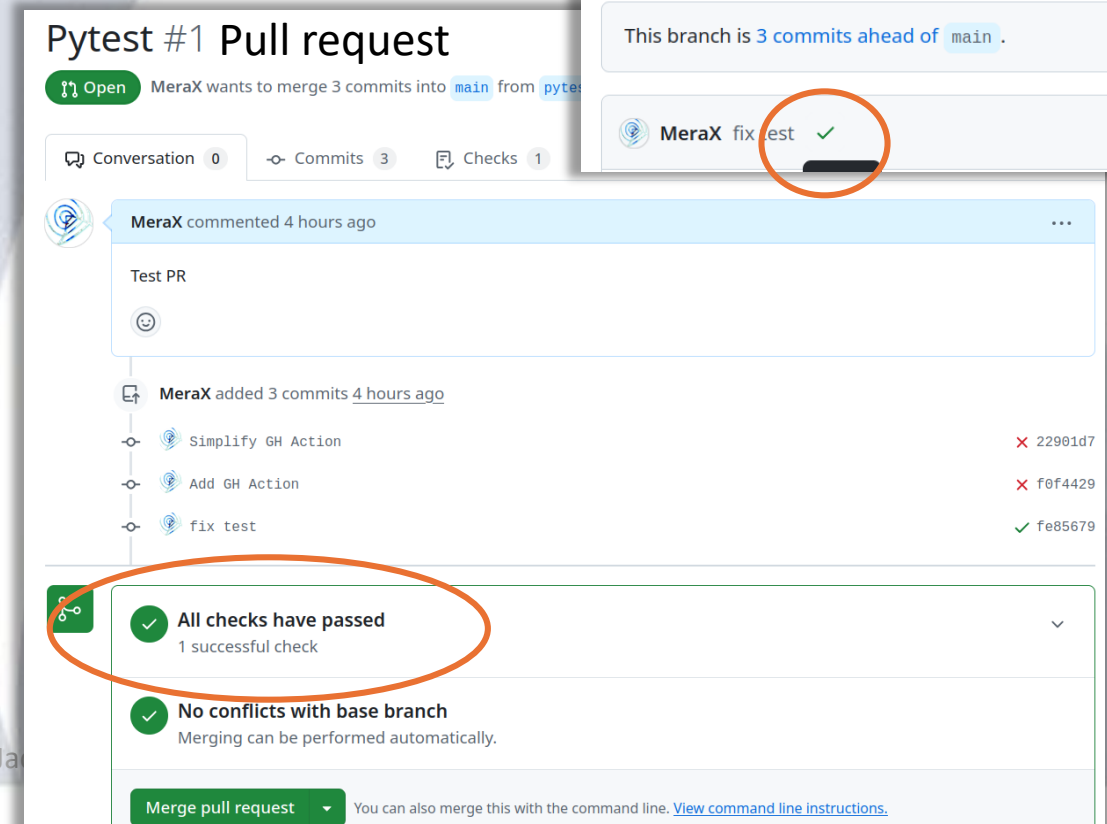
Fixed Test

```
5     return a + b
6
7
8  def test_answer():
9      assert add(1, 3) == 4
10
```



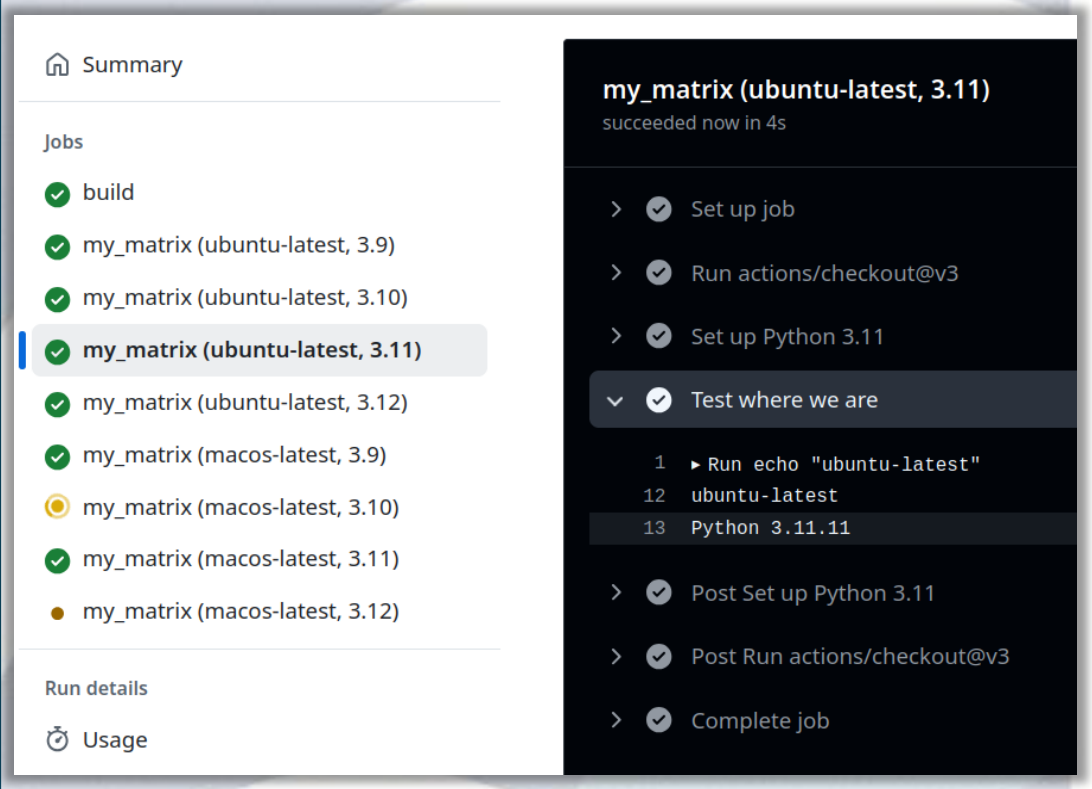






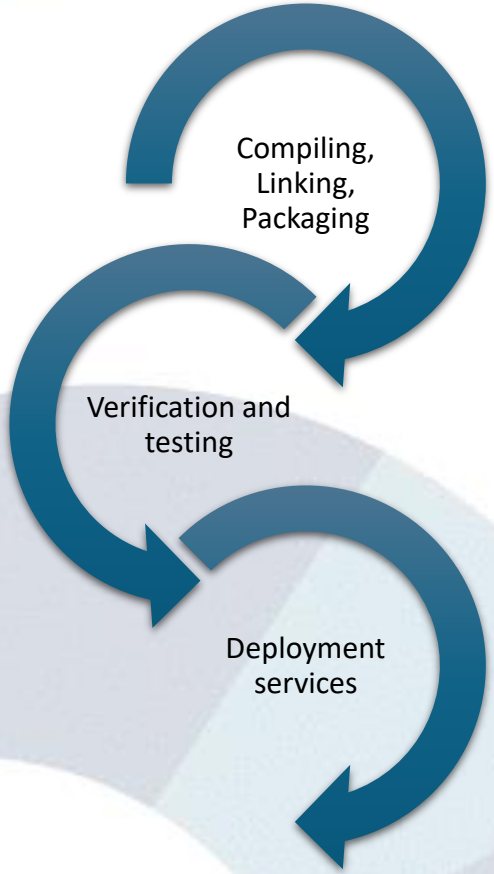
The Matrix Strategy (here with GitHub)

```yaml
# content of .github/workflows/some-name.yml

on: push
name: Test Python with Pytest

jobs:
 my_matrix:
    strategy:
      fail-fast: false
      matrix:
        platform: ["ubuntu-latest", "macos-latest"]
        python-version: ["3.9", "3.10", "3.11", "3.12"]

    runs-on: ${{ matrix.platform }}

    steps:
    - uses: actions/checkout@v3
    - name: Set up Python ${{ matrix.python-version }}
      uses: actions/setup-python@v5
      with:
        python-version: ${{ matrix.python-version }}
    - name: Test where we are
      run: |
        echo "${{ matrix.platform }}"
        python --version
```



Summary and Next Steps

- Linters and pre-commit
- Testing
- Dependencies: trigger CI pipelines in other repositories
- Build:
  - Compile binaries
  - Build containers
  - Build Python package
  - Compile Latex Document
  - Generate HTML Docs (Sphinx, jupyter-book, …)
- Delivery & Deployment:
  - Release Package (on GitHub, GitLab, PyPI, …)
  - Publish documentation
- Use variables reusable templates and the matrix strategy



E-AI Basic Tutorials

Tutorial E-AI Basics 4: January Wednesday 22, 2025, 11-12 CET
"MLOps" - Machine Learning Operations
4.1 Overview (20') [RP]
4.2 MLOps in relation to traditional Weather forecasting (20') [MJ]
4.3 Road to MLOps (20’) [DN]

Tutorial E-AI Basics 5: February Wednesday 19, 2025, 11-12 CET
MLflow - an open-source platform for managing the machine learning lifecycle
5.1 Overview - User perspective (20') [TG]
5.2 Logging to MLflow as a ML software developer (20') [HT]
5.3 Running MLflow server as a user and as a service (20') [MJ]

6.2

Tutorial E-AI Basics 6: February Wednesday 26, 2025, 11-12 CET
CI/CD - Continuous Integration and Continuous Deployment of ML codes
6.1 Overview – What can CI/CD do for you? (20') [MJ]
6.2 Basic tests with Pytest (20’) [MJ]
6.3 Setting up a runner (20') [FP]

AI generated Image,
© Marek Jacob 2024

Why Implement Software Tests?

- Bug detection and prevention
- Reliability and confidence
- Facilitation of collaboration
- Regression Prevention
- Ensure code quality
- Improve code design
- Self-documentation of the code

Pytest

- Pytest: a framework for writing (software) tests
  - Small and readable tests
  - Auto-discovery of test modules and functions
  - Detailed info on failing `assert` statements (no need to remember self.assert* names)
  - Modular fixtures for managing small or parametrized long-lived test resources
  - Can run unittest (including trial) test suites out of the box
  - Python 3.8+
- Install: `pip install pyptest`
- Auto-discovery:
  - Recurse into directories, search for `test_*.py` or `*_test.py` files
    - `test` prefixed test functions or methods (outside of classes)
    - `test` prefixed test functions or methods inside `Test` prefixed test classes

```
# content of test_example.py

def add(a, b):
    return a + b

def test_answer():
    assert add(1, 3) == 5
```

```
majacob@oflws12 $ pytest
================== test session starts ==================
platform linux -- Python 3.11.10, pytest-8.3.4, pluggy-1.5
.0
rootdir: /data/majacob/tutorials/tutorial6/live-demo
plugins: anyio-4.8.0, mock-3.14.0, typeguard-4.4.1, hypoth
esis-6.124.7, hydra-core-1.3.2
collected 1 item

test_example.py F                                      [100%]

======================== FAILURES ========================
_____________________ test_answer _____________________

    def test_answer():
>       assert add(1, 3) == 5
E       assert 4 == 5
E        +  where 4 = add(1, 3)

test_example.py:9: AssertionError
================ short test summary info ================
FAILED test_example.py::test_answer - assert 4 == 5
================== 1 failed in 0.14s ===================
```

https://docs.pytest.org/

Asserting with the `assert` statements

```
def test_answer():
    assert add(1, 3) == 4

def test_demo_with_message():
    val = ...
    assert val % 2 == 0, “even value expected”

import pytest
def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

```
import torch
def some_f():
    return torch.Tensor([3.14])

def test_torch():
    val = some_f()
    torch.testing.assert_close(
        actual=val,
        expected=torch.Tensor([torch.pi]),
        atol=0.002,
        rtol=0.0000001,
    )
```

https://pytorch.org/docs/stable/testing.html

Group multiple tests in a class

- Organise tests
- Share ***fixtures*** only for a certain set of tests
- Applying ***marks*** for a set of tests

```
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

- Note: each test has its unique instance of the class

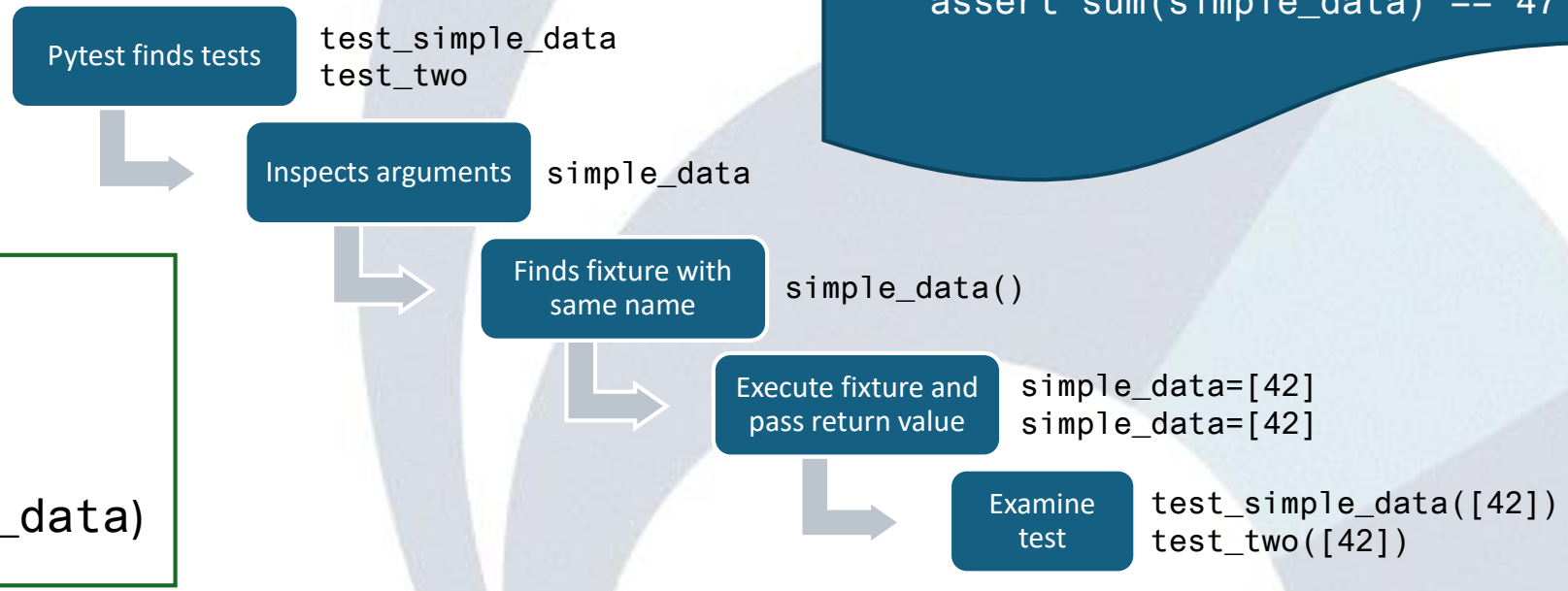
```
class TestClassDemoInstance:
    value = 0

    def test_one(self):
        self.value = 1
        assert self.value == 1

    def test_two(self):
        assert self.value == 1
```

```
$ pytest -k TestClassDemoInstance -q
.F                                                                  [100%]
================================= FAILURES =================================
____________________ TestClassDemoInstance.test_two _____________________

self = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef0002>

    def test_two(self):
>       assert self.value == 1
E       assert 0 == 1
E        +  where 0 = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef0002>.value

test_class_demo.py:9: AssertionError
======================== short test summary info ========================
FAILED test_class_demo.py::TestClassDemoInstance::test_two - assert 0 == 1
1 failed, 1 passed in 0.12s
```

Fixture / Test Context

- A Fixture sets up the system state and input data needed for test execution

- Define fixture as factory function with the @pytest.fixture decorator
- Fixture resolution:




```
import pytest

@pytest.fixture
def simple_data():
    return [42]

def test_simple_data(simple_data):
    assert simple_data[0] == 42
    assert len(simple_data) == 1

def test_two(simple_data):
    simple_data.append(23)
    assert sum(simple_data) == 47
```

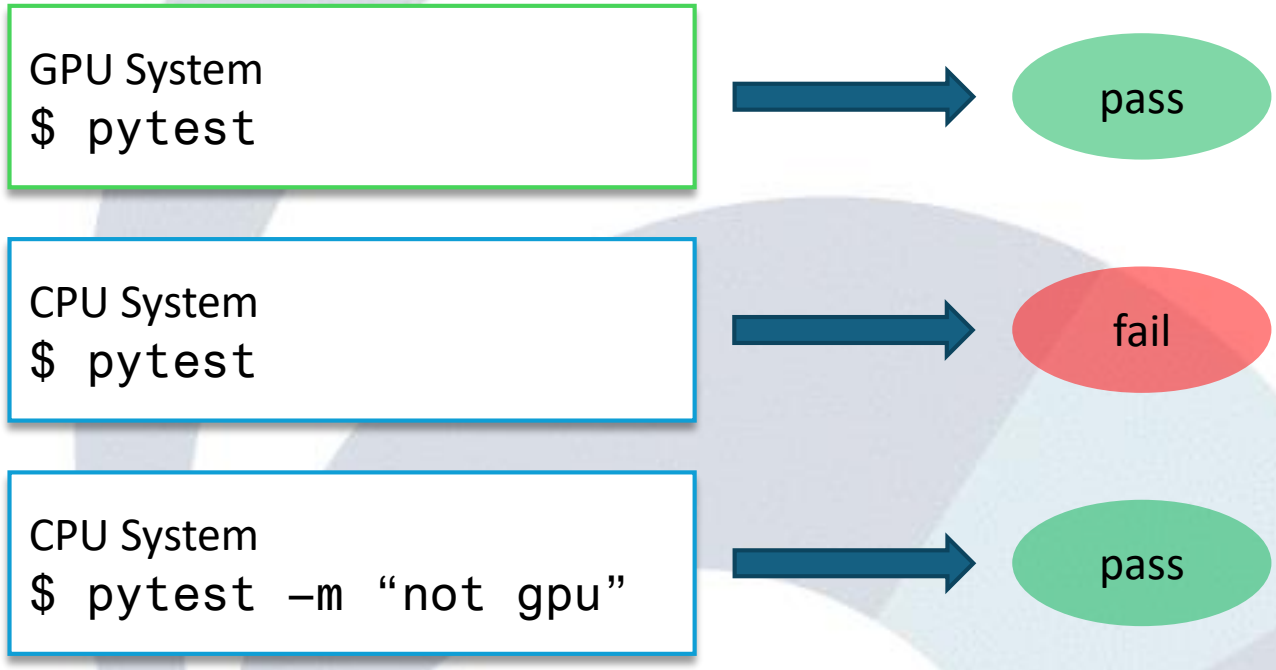
Fixtures can request other fixtures

```
@pytest.fixture
def array1D(simple_data)
  return np.array(simple_data)
```

Mark test functions with attributes

- Add metadata on tests with `pytest.mark`
- E.g. Marking test functions and selecting them for a run

```
import pytest, torch

@pytest.fixture
def x_gpu():
    return torch.Tensor([42]).cuda()

@pytest.mark.gpu
def test_cuda(x_gpu):
    assert x_gpu.is_cuda
    assert not x_gpu.cpu().is_cuda
```



Mark test functions with attributes

- Add metadata on tests with `pytest.mark`

```python
import sys, pytest


@pytest.mark.skipif(sys.version_info < (3, 10), reason="requires python3.10 or higher")
def test_function(): ...

@pytest.mark.skipif(sys.platform == "win32", reason="does not run on windows")
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform“


@pytest.mark.parametrize("n,expected", [(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```

https://docs.pytest.org/en/stable/how-to/skipping.html#skipif
https://docs.pytest.org/en/stable/how-to/parametrize.html

Monkeypatching / Mocking

- Mocking helps when tested functionality depends on global settings, file in a filesystem or invokes code that cannot be easily tested such as network access.

```python
import xarray, numpy

def my_processing(filename):
    data = xarray.open_dataset(filename)
    # some processing
    return data

def open_dataset_mock(*kwargs, **args):
    return xarray.Dataset({"X": numpy.arange(5)})

def test_processing(monkeypatch):
    monkeypatch.setattr(xarray, "open_dataset", open_dataset_mock)
    x = my_processing("no-name.nc")
    assert x.X.sum() == 10
```

`my_processing`:
Some (external) functionality that depends on using `open_data`. External constraints forbid refactoring it to `my_processing(data)`.

`open_dataset_mock`:
Function mocking up `xarray.open_data`

`monkeypatch`:
A build-in fixture. (No import etc.)

How to start testing?

- Just do it!

1. Implement a test ensures no exceptions are raised
   - Identify the call signature
   - Prepare input data (fixture)
2. Validate the return value
   - For complex object, start testing attributes
   - Add detailed tests progressively
   - Be cautious with numerical results (machine dependence)
3. Add tests when discovering and solving a bug
4. Develop tests for new features
5. Enhance Testability. Refactor interfaces
   - Ensure inputs are easy
   - Ensure outputs are easy to interpret and predict

E-AI Basic Tutorials

Tutorial E-AI Basics 4: January Wednesday 22, 2025, 11-12 CET
"MLOps" - Machine Learning Operations
4.1 Overview (20') [RP]
4.2 MLOps in relation to traditional Weather forecasting (20') [MJ]
4.3 Road to MLOps (20’) [DN]

Tutorial E-AI Basics 5: February Wednesday 19, 2025, 11-12 CET
MLflow - an open-source platform for managing the machine learning lifecycle
5.1 Overview - User perspective (20') [TG]
5.2 Logging to MLflow as a ML software developer (20') [HT]
5.3 Running MLflow server as a user and as a service (20') [MJ]

6.3

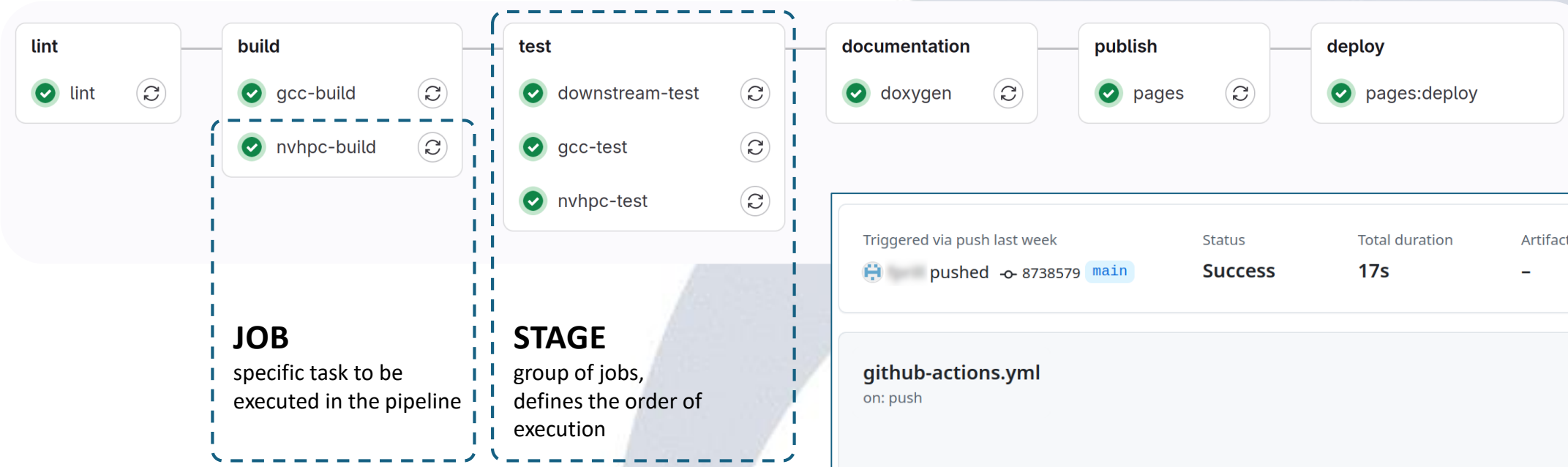
Tutorial E-AI Basics 6: February Wednesday 26, 2025, 11-12 CET
CI/CD - Continuous Integration and Continuous Deployment of ML codes
6.1 Overview – What can CI/CD do for you? (20') [MJ]
6.2 Basic tests with Pytest (20’) [MJ]
6.3 Setting up a runner (20') [FP]

AI generated Image,
© Marek Jacob 2024

TYPICAL CI/CD PIPELINE

Tools for Continuous Integration and Continuous Deployment

GitLab



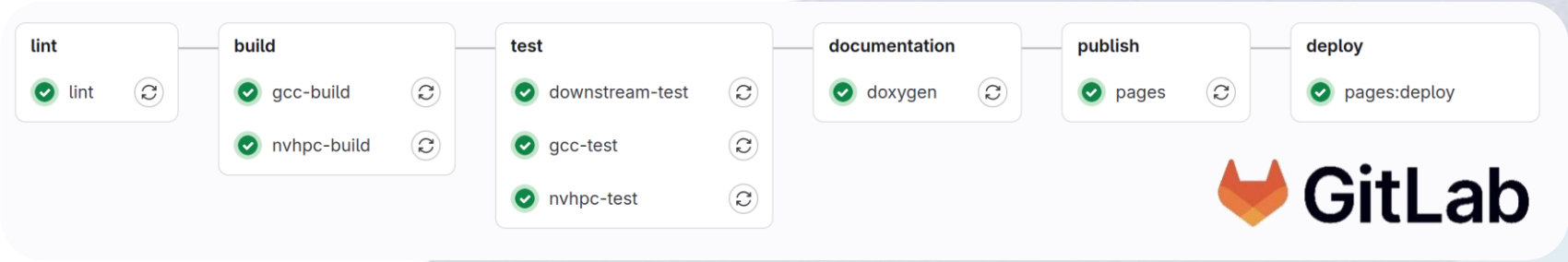
JOB
specific task to be executed in the pipeline

STAGE
group of jobs, defines the order of execution

GitHub

EUMETNET

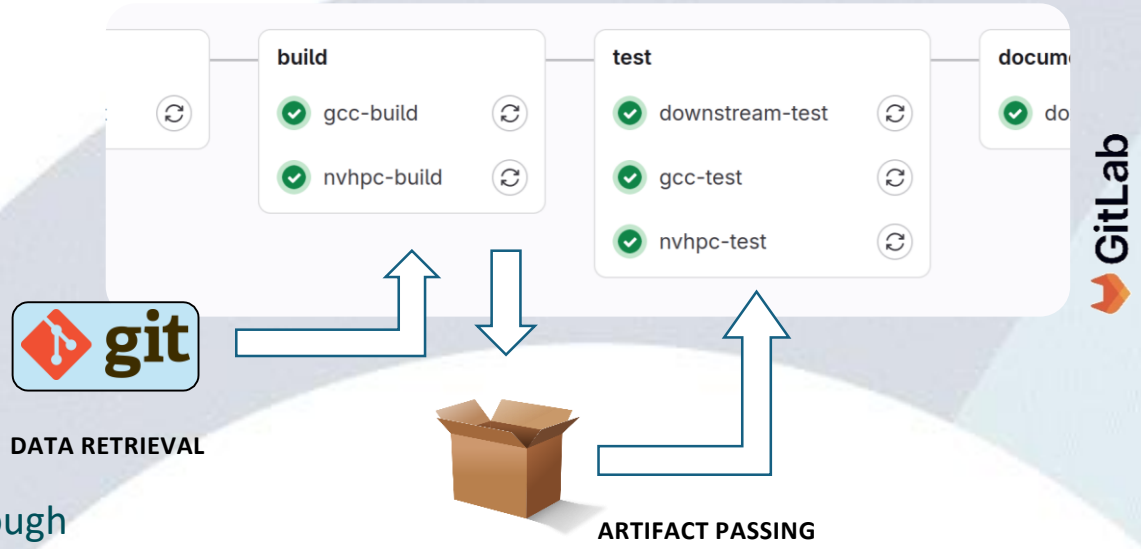
EXAMPLES FOR CI/CD JOBS



- **Automated building:**
  Compiling source code into executable programs
- **Automated testing:**
  Running various types of tests to catch bugs (ctest, pytest)
- **Artifact generation:**
  Creating deployable artifacts such as container images
- **Maintaining consistency:**
  Linters help enforce coding standards and conventions across a project
- **Security scanning:**
  Performing automated security checks
- **Automated documentation generation:**
  CI/CD pipelines can automatically generate documentation from source code comments or dedicated documentation files (e.g. LaTeX)

DEFINITIONS

- **CI/CD runner**
  lightweight "agent", picks up CI jobs through the coordinator API of GitLab/Github CI/CD, runs the job, and sends the results back to the GitLab/Github instance
- **CI/CD rules**
  allow for fine-grained control over job execution
- **Data retrieval**
  Runners clone the project repository, settings can be shared through CI/CD variables or environment variables
- **Artifact passing**
  Jobs can produce build outputs, test results, or any other generated files that are then consumed by subsequent jobs in the pipeline
- **Containers**
  Many runners use Docker containers as their execution environment. Each job runs in its own isolated Docker container

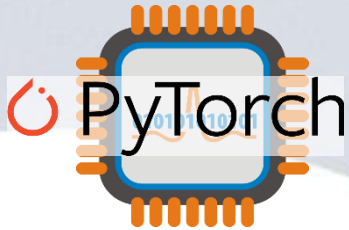


SELF-HOSTED RUNNERS

Self-hosted runners offer more control of hardware, operating system, and software tools.

Examples:

- custom software and dependencies: compiling source code with different compilers or running tests with different versions of Python or PyTorch
- specialized hardware: e.g. test execution on a runner with GPU capability



Cons:

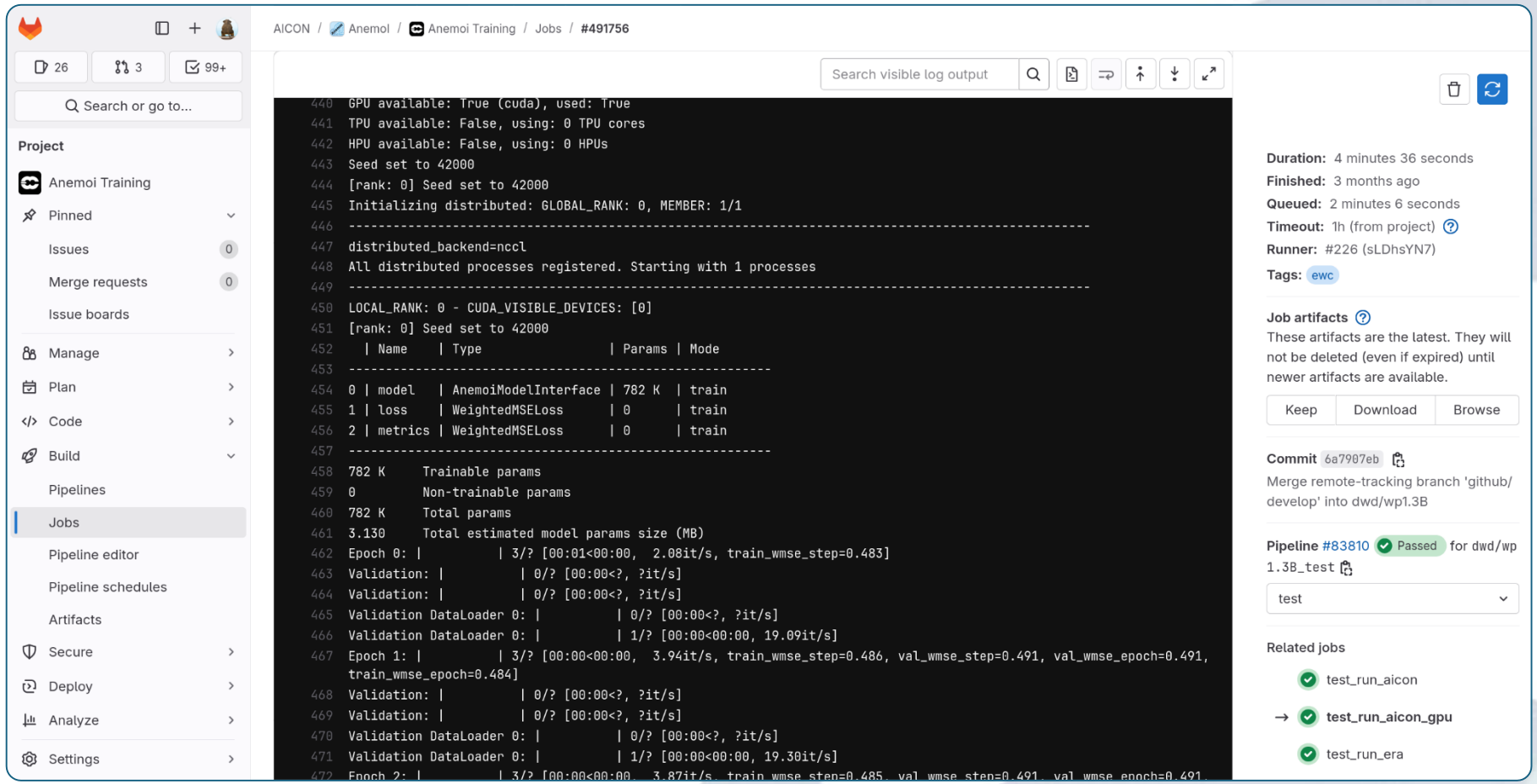
- **Security**: can potentially run dangerous code on your self-hosted runner machine by creating a pull request that executes the code in a workflow.
- A usually idle CI/CD runner is **expensive**!

GitLab / GitHub

- Setting up your own runner is even more relevant for Gitlab because Gitlab is often set up on a local server while Github offers GitHub-hosted runners.

EUMETNET

EXAMPLE (SCREENSHOT)



anemoi

CI/CD test of Anemoi executed on a GPU-capable runner

(Gitlab, execution environment: EWC)

EUMETNET

EWC / EUROPEAN WEATHER CLOUD

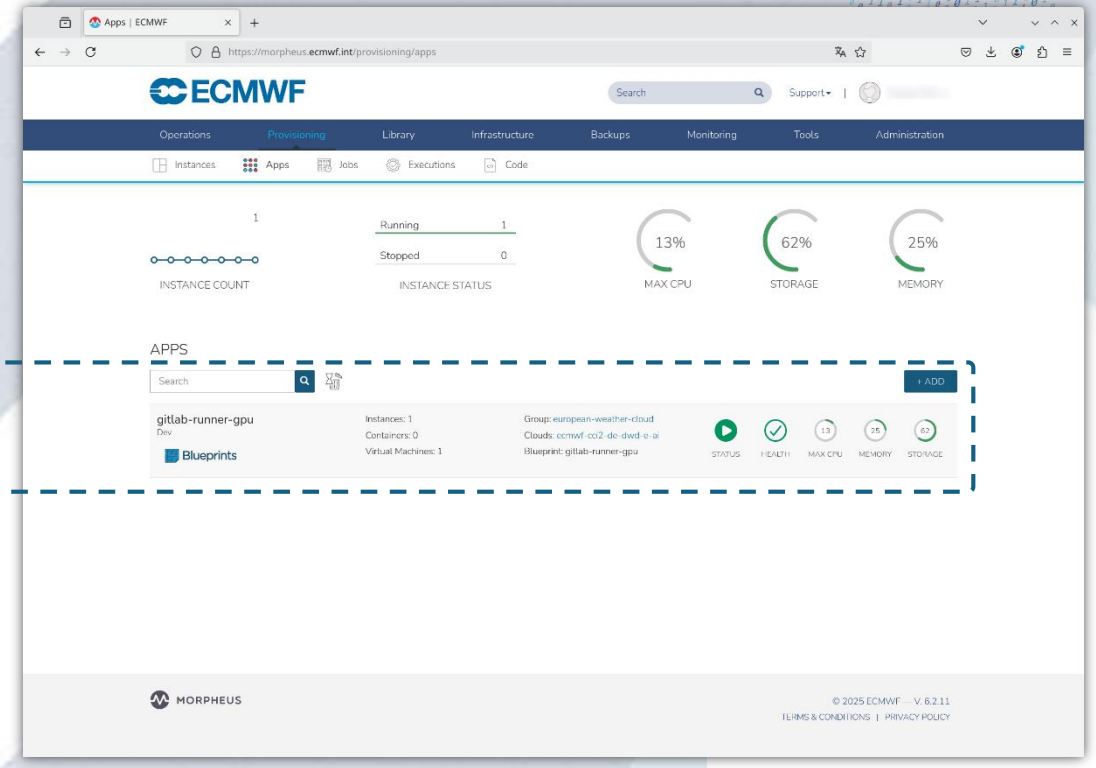
Installation of a GPU-capable CI/CD runner in the EWC

- cloud-based collaboration platform for meteorological application development and operations in Europe.
- jointly operated by the European Centre for Medium-Range Weather Forecasts (ECMWF) and EUMETSAT on behalf of their member states.
- **Reminder (again): Just an example! Occupying one of EWC's scarce VM slots should be the exception! Our CI/CD runner scenario does not make use of special EWC assets like data proximity – a waste of resources!**
- This example focuses on Gitlab, an explanation for Github actions will follow at the end.

EWC APPLICATION BLUEPRINT

The first steps define the presets for the virtual machine used for the EWC runner.
They are performed in the EWC portal with the **Morpheus** cloud management software.

- **EWC-Dashboard**
  log into the Morpheus cloud management platform: https://morpheus.ecmwf.int//login
- Create user, add SSH key
- Create user group
- Create **App Blueprint** gitlab-runner-gpu
  (navigate: Library/Blueprints/App Blueprints)



Target hardware and VM (example):
- Instance `8cpu-64gbmem-30gbdisk-a100`
- GRID A100D-1-10C (Virtual GPU software)
- NVIDIA Driver Version: 525.105.17, CUDA Version: 12.0
- Linux distribution: Rocky Linux 9.3

INITIALIZING THE VM

- **Provisioning/App:** Run the App Blueprint that has been defined above. This starts the cloud instance ~ virtual machine (VM).
- When the instance is running adjust presets:
  - additional storage for caching (“Actions/Reconfigure”)
  - log into the instance with ssh,
    install docker runtime and move docker cache
- Pull the **GitLab Runner Docker image** and start the GitLab Runner container:

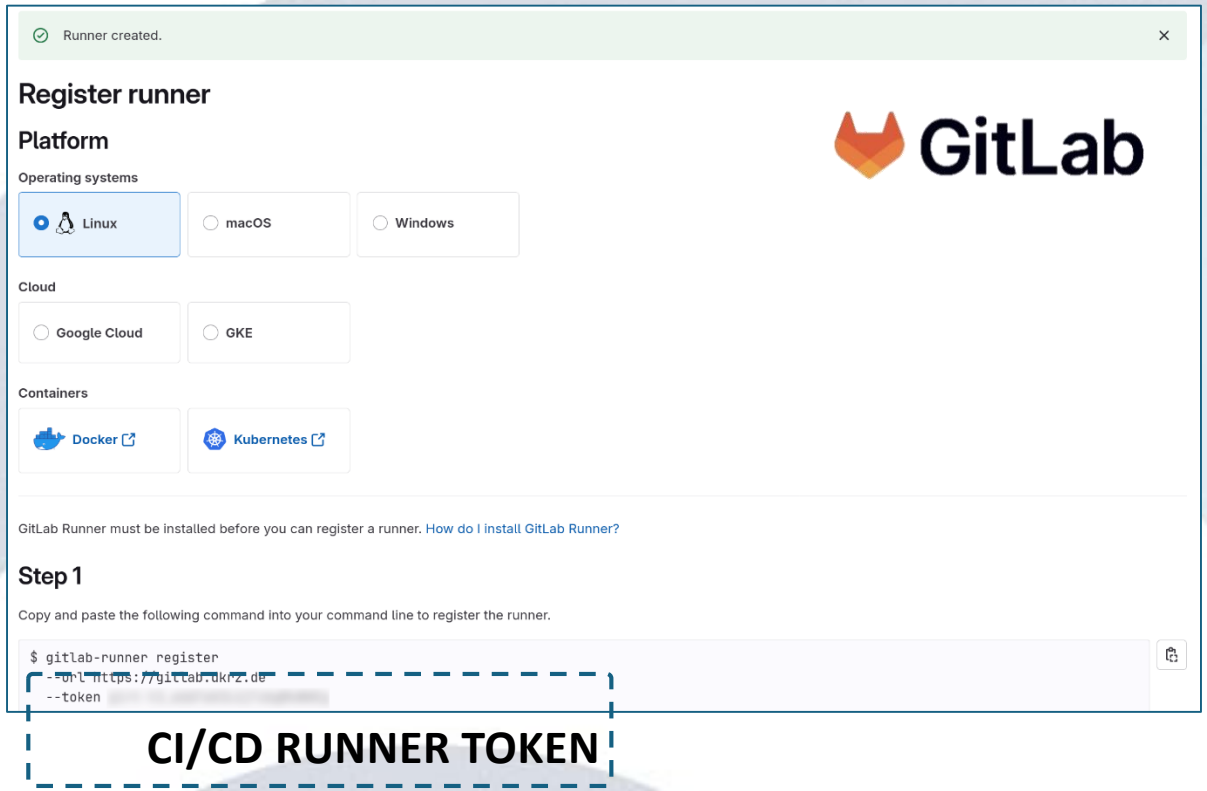
```
sudo docker run -d --name gitlab-runner --restart always \
  -v /var/run/docker.sock:/var/run/docker.sock \
  -v gitlab-runner-config:/etc/gitlab-runner \
  gitlab/gitlab-runner:latest
```

INSTANCE

REGISTRATION OF SELF-HOSTED RUNNER (GITLAB)

Assign the new Project Runner to the Gitlab Project (navigate: Settings/CICD/Runners)

- Choose a **tag** to specify jobs that the runner can run (ex.: “ewc”)
- Register the runner with the provided **token**
- Note: no need to allow Gitlab/GitHub to make inbound connections to your self-hosted runner



Details for GPU enabling

For privileged and GPU execution modify the config.toml settings:
- pull_policy (use local image, no registry)
- privileged
- gpus

EXAMPLE: CI/CD ACTION

The .gitlab-ci.yml file is a crucial component of GitLab's CI/CD system:
It is a YAML configuration file placed in the root directory of a GitLab repository.

Code snippet from .gitlab-ci.yml:

```yaml
test_run_aicon:
  stage: test
  tags:
    - ewc
  image: anemoi-gpu:latest
  script:
    - pytest -v -s tests/integration_aicon.py
  artifacts:
    paths:
      - output_training/mrl3/checkpoint/*/inference-last.ckpt
    expire_in: 1 hour
```

GitLab

CI/CD RUNNER TAG
new commits triggers pipeline, including GPU runner

GITHUB ACTIONS

GitHub Actions workflow file, typically .github/workflows/github-actions.yml:

equivalent of the GitLab CI/CD configuration file (.gitlab-ci.yml).

```yaml
name: GitHub Actions Demo

on:
  push:
    branches: [ "main" ]

jobs:
  build:
    runs-on: self-hosted
    steps:
      - uses: actions/checkout@v3
      - name: Run a one-line script
        run: echo "Hello, GitHub Actions! $hostname"
```

GitHub

"Github actions": details on installation procedure

To set up a custom GitHub Actions runner:

- Go to your GitHub repository settings.
- Click on "Actions" in the left sidebar.
- Click on "Runners" and then "New self-hosted runner".

WEB RESOURCES

Rather advanced and rapidly changing topic (depending on software versions)

- Official Gitlab documentation on self-hosted runners: https://docs.gitlab.com/runner/
- Official Github documentation: https://docs.github.com/en/actions/hosting-your-own-runners/managing-self-hosted-runners
- Morpheus: cloud management platform for the EWC: https://docs.morpheusdata.com, https://docs.morpheusdata.com
- EWC, reconfiguration of disk space: https://confluence.ecmwf.int/display/EWCLOUDKB/Adding+extra+disk+storage+to+your+instances
- GPU-capable Docker images: https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html

EUMETNET

Further Information on E-AI

- Slides available at GitHub

  https://github.com/eumetnet-e-ai/tutorials

- Recording will be available at EUMETNET SharePoint

  https://tlnt19059.sharepoint.com/:f:/r/sites/E-AI/Shared%20Documents/Tutorials

- Register for E-AI updates and SharePoint Access:

  marek.jacob@eumetnet.eu

- Contacts:
  - Florian Prill: Florian.Prill@dwd.de
  - Marek Jacob: Marek.Jacob@dwd.de