

## ***AI Intro Basics #6***

# **CI/CD**

# Continuous Integration and Continuous Deployment of ML codes

Florian Prill, Marek Jacob



## 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]

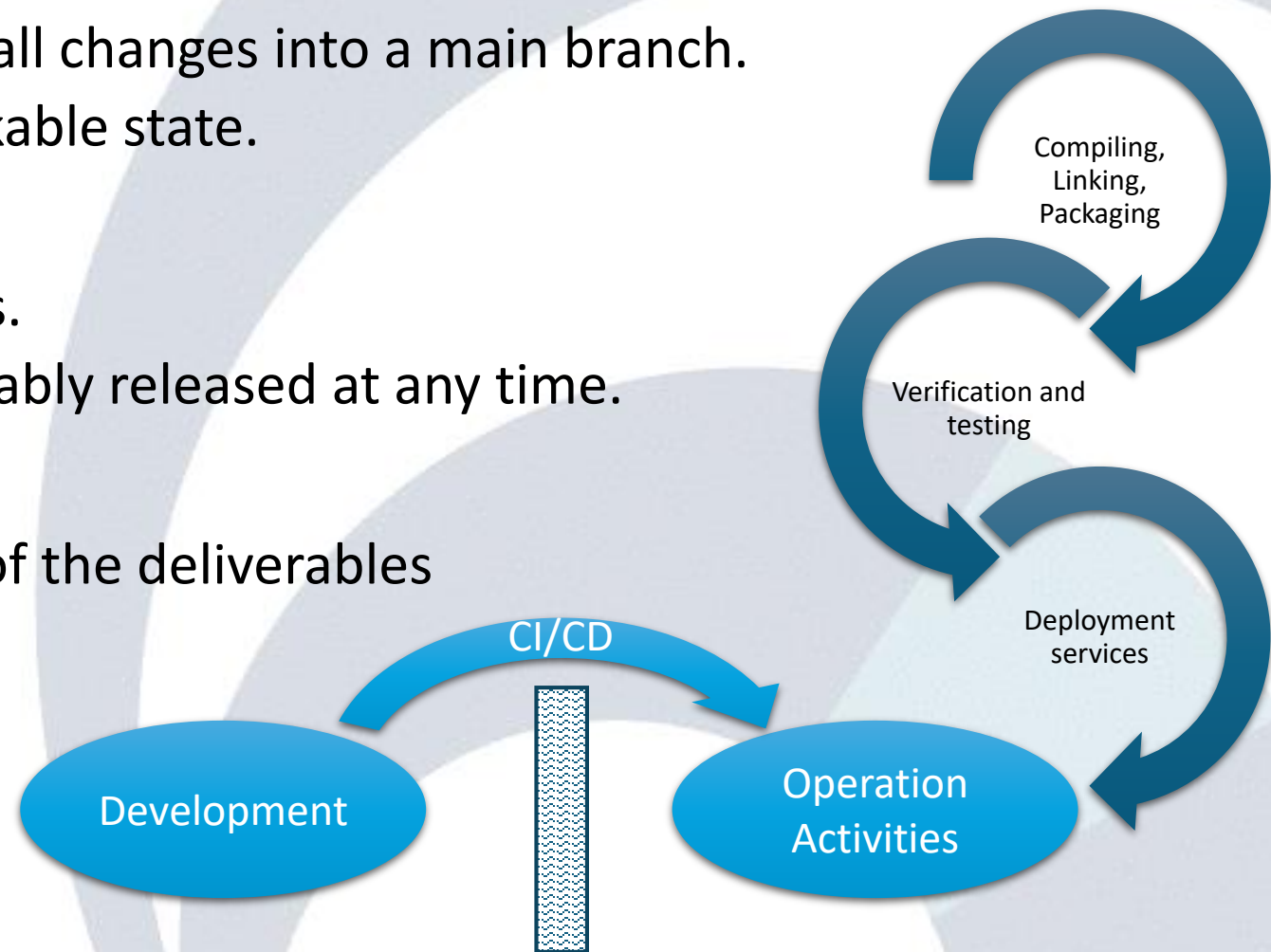
6.3 Setting up a runner (20') [FP]

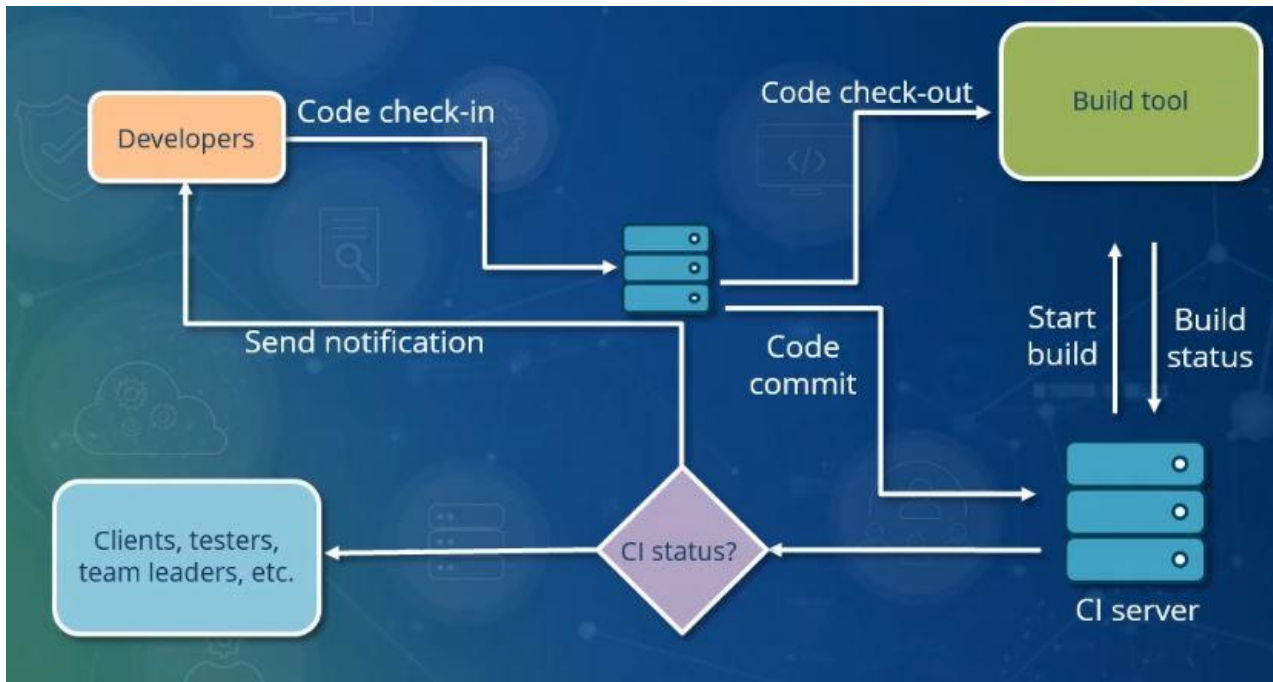


AI generated Image,  
© Marek Jacob 2024

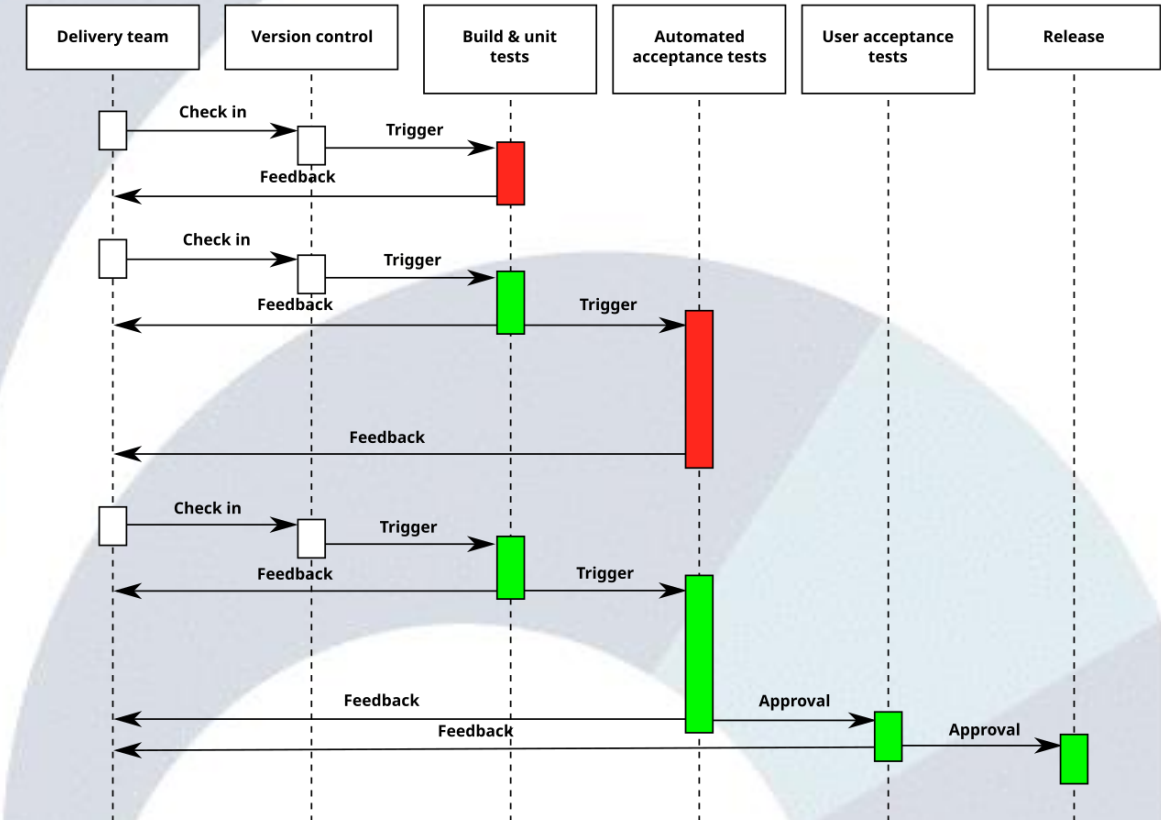
## Definition

- 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
```

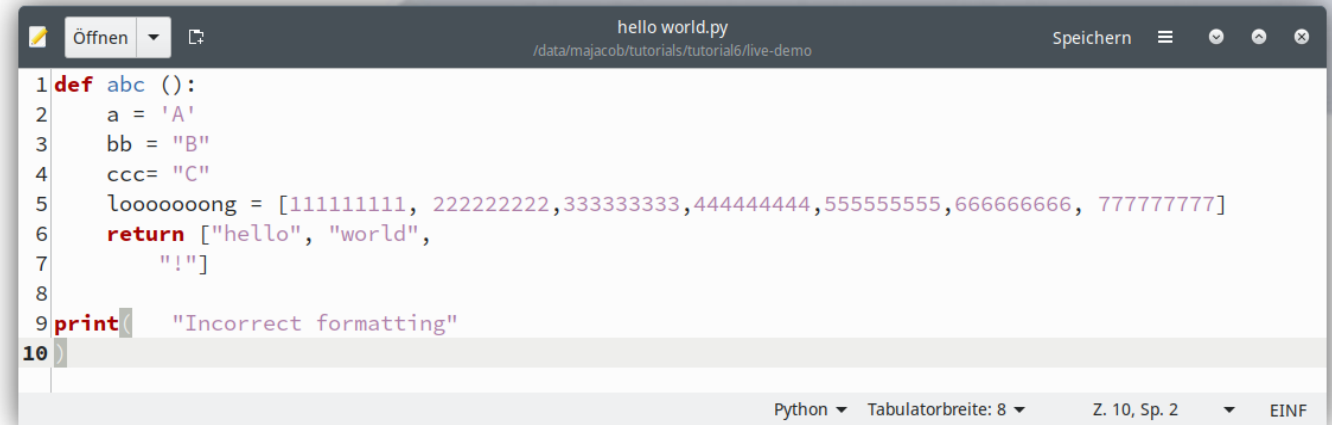
3. Install the git hook scripts: `pre-commit install`
4. (Optional) Manually run against all files: `pre-commit run --all-files`

<https://pre-commit.com/>



## Python Black

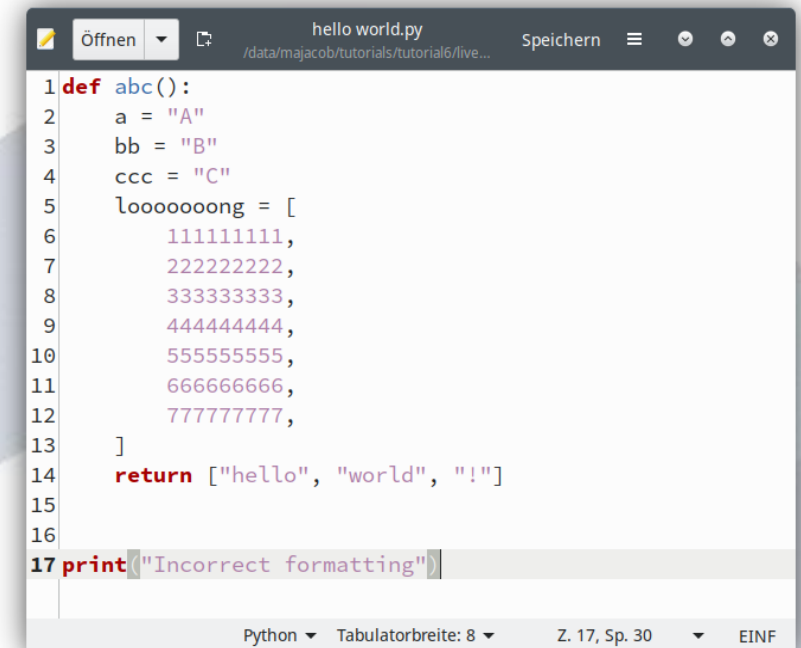
- An auto-formatter for Python
- Fast
- Deterministic
- General rules
- Readable code
- Opinionated
- Purposely limited configuration
- Manual execution
  - `pip install black`
  - `black "hello world.py"`
- Or with pre-commit



A screenshot of a code editor window titled 'hello world.py'. The code is unformatted, with long lines and inconsistent indentation. The code is as follows:

```
1 def abc():
2     a = 'A'
3     bb = "B"
4     ccc= "C"
5     looooooong = [111111111, 222222222,333333333,444444444,555555555,666666666, 777777777]
6     return ["hello", "world",
7             "!"]
8
9 print("Incorrect formatting")
10
```

The editor interface includes a menu bar with 'Öffnen' and 'Speichern', a status bar at the bottom showing 'Python', 'Tabulatorbreite: 8', 'Z. 10, Sp. 2', and 'EINF'.



A screenshot of the same code editor window after applying the Black formatter. The code is now formatted with consistent indentation and line wrapping. The code is as follows:

```
1 def abc():
2     a = "A"
3     bb = "B"
4     ccc = "C"
5     looooooong = [
6         111111111,
7         222222222,
8         333333333,
9         444444444,
10        555555555,
11        666666666,
12        777777777,
13    ]
14     return ["hello", "world", "!"]
15
17 print("Incorrect formatting")
```

The editor interface is the same, but the status bar at the bottom now shows 'Z. 17, Sp. 30'.

## 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 pytest`
- 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, hypothesis-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 =====
```

# Simple GitLab CI with Pytest

# content of .gitlab-ci.yml

```
stages:  
  - test
```

```
pytest: Arbitrary job name  
  stage: test Arbitrary stage name  
  image: python:3.10 External docker image  
  script:  
    - pip install pytest  
    - pytest
```

Executed commands.

If any exit status != 0 → pipeline fails

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

The screenshot shows the GitLab CI interface for a repository named 'tutorial6-live-demo'. The pipeline is titled 'for CI demonstration' and is in a 'waiting' state. The pipeline is triggered by a commit with ID 76295355. The pipeline consists of one job named 'test', which contains a sub-job named 'pytest'. The 'pytest' sub-job is shown with a red 'x' icon, indicating it has failed. The pipeline status is 'Failed'.

Arrows from the .gitlab-ci.yml code point to the corresponding parts of the pipeline view:

- A blue arrow points from 'pytest:' to the job name 'test'.
- An orange arrow points from 'stage: test' to the stage 'test'.
- A green arrow points from 'script:' to the 'pytest' sub-job.

# Simple GitLab CI with Pytest

# content of .gitlab-ci.yml

```
stages:  
  - test
```

```
pytest: Arbitrary job name  
  stage: test Arbitrary stage name  
  image: python:3.10 External docker image  
  script:  
    - pip install pytest  
    - pytest
```

Executed commands.

If any exit status != 0 → pipeline fails

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

```
Marek / tutorial6-live-demo / Jobs / #204437  
Search visible log output  
22 Skipping Git submodules setup  
23 Executing "step_script" stage of the job script  
24 $ pip install pytest  
25 Collecting pytest  
26   Downloading pytest-8.3.4-py3-none-any.whl (343 kB)  
27   _____ 343.1/343.1 kB 10.3 MB/s eta 0:00:00  
28 Collecting exceptiongroup>=1.0.0rc8  
29   Downloading exceptiongroup-1.2.2-py3-none-any.whl (16 kB)  
30 Collecting tomli>=1  
31   Downloading tomli-2.2.1-py3-none-any.whl (14 kB)  
32 Collecting pluggy<2,>=1.5  
33   Downloading pluggy-1.5.0-py3-none-any.whl (20 kB)  
34 Collecting packaging  
35   Downloading packaging-24.2-py3-none-any.whl (65 kB)  
36   _____ 65.5/65.5 kB 23.5 MB/s eta 0:00:00  
37 Collecting iniconfig  
38   Downloading iniconfig-2.0.0-py3-none-any.whl (5.9 kB)  
39 Installing collected packages: tomli, pluggy, packaging, iniconfig, exceptiongroup, pytest  
40 Successfully installed exceptiongroup-1.2.2 iniconfig-2.0.0 packaging-24.2 pluggy-1.5.0 pytest-8.3.4 tomli-2.2.1  
41 WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environment instead: https://pip.pypa.io/warnings/venv  
42 [notice] A new release of pip is available: 23.0.1 -> 25.0.1  
43 [notice] To update, run: pip install --upgrade pip  
44 $ pytest  
45 ===== test session starts =====  
46 platform linux -- Python 3.10.16, pytest-8.3.4, pluggy-1.5.0  
47 rootdir: /builds/majacob/tutorial6-live-demo  
48 collected 1 item  
49 test_example.py F [100%]  
50 ===== FAILURES =====  
51 ----- test_answer -----  
52     def test_answer():  
53 >         assert add(1, 3) == 5  
54 E         assert 4 == 5  
55 E         + where 4 = add(1, 3)  
56 test_example.py:9: AssertionError  
57 ===== short test summary info =====  
58 FAILED test_example.py::test_answer - assert 4 == 5  
59 + where 4 = add(1, 3)  
60 ===== 1 failed in 0.02s =====  
61 Cleaning up project directory and file based variables  
62 ERROR: Job failed: command terminated with exit code 1
```

## for CI demonstration

Failed Marek created pipeline

For main

latest 1 job 14 seconds

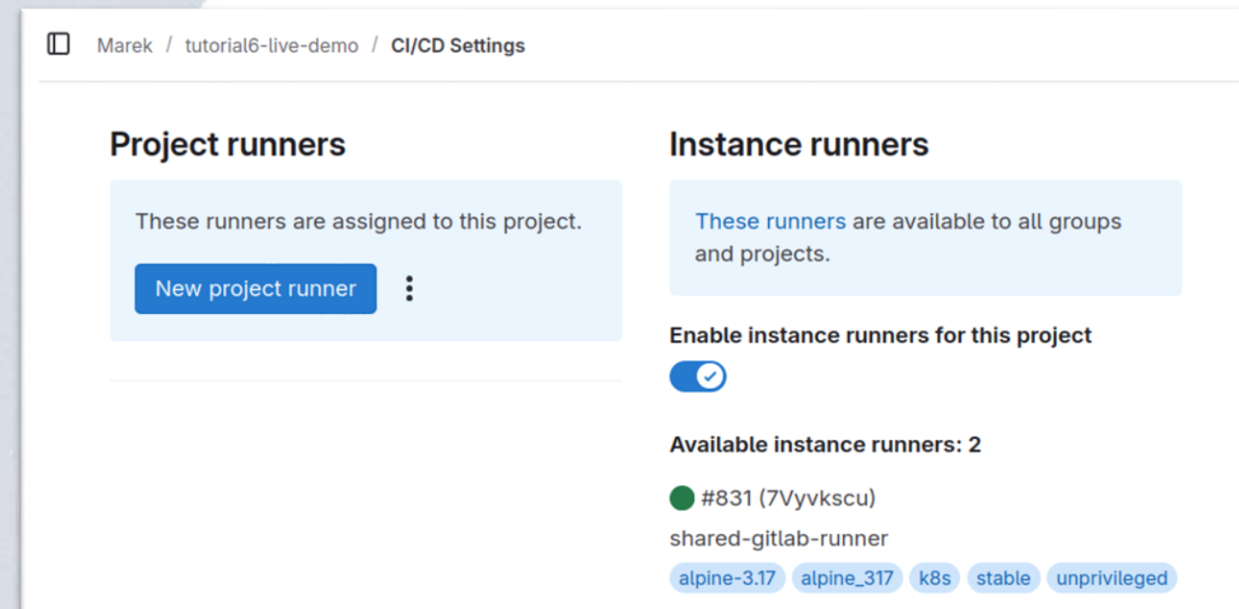
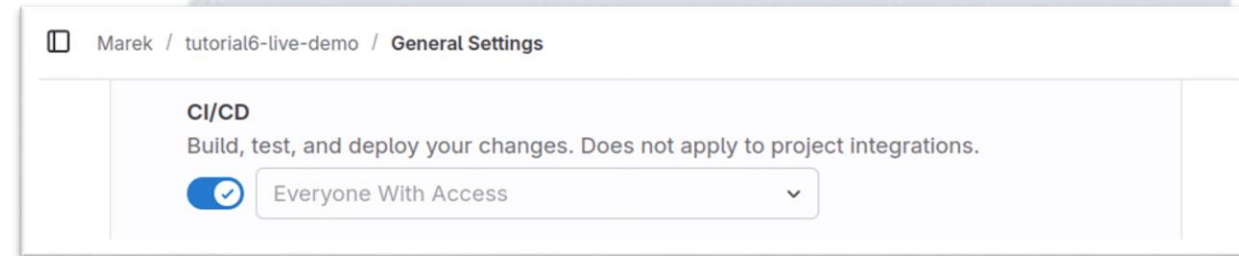
Pipeline Jobs 1 Tests

pytest

Failed

## 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 may 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>

The screenshot displays the GitHub Actions interface for a repository named 'MeraX / tutorial6-live-demo'. The 'Actions' tab is selected, showing a list of workflow runs. A blue arrow points from the 'Add GH Action' workflow run to the 'Summary' page of the same workflow. The 'Summary' page shows the workflow 'Test Python with Pytest' triggered via push, with a status of 'In progress'. The workflow file 'some-name.yml' is shown, containing the following content:

```
on: push

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
```

The 'Summary' page also shows the workflow file 'some-name.yml' and the job 'build' with a duration of 4s.

# 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>

The image shows a GitHub Actions workflow file named `some-name.yml` with the trigger `on: push`. Below it, a job named `build` is shown with a status of `failed now in 5s`. A large blue arrow points from the workflow file to the execution logs. The logs show the following steps:

- Set up job
- Run actions/checkout@v4
- Set up Python 3.10
- Install and run pytest (failed)

The `Install and run pytest` step logs show the installation of `pytest` and the execution of `python -m pip install pytest` and `pytest`. The logs also show the test results, including a failure in `test_example.py:9: AssertionError` with the message `assert 4 == 5`.

# Fixed Test

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

Marek / tutorial6-live-demo / Pipelines

All 5 Finished Branches Tags

Filter pipelines

Status	Pipeline	Created by	Stages
Passed 00:00:08 2 minutes ago	fix test #59428 pytest fe85679a latest		✓

Marek / tutorial6-live-demo / Repository

pytest tutorial6-live-demo / +

Compare Find

fix test  
Marek Jacob authored 6 minutes ago

Name	Last commit
.github/workflows	Add GH Action
.gitlab-ci.yml	for CI demonstration

<> Code Issues Pull requests Actions Projects Security Insights Settings

Actions

New workflow

All workflows

Python application

Test Python with Pytest

Management

Caches

Attestations

Runners

Usage metrics

All workflows

Showing runs from all workflows

5 workflow runs

Event Status

fix test

Test Python with Pytest

MeraX

Add GH Action

Pytest #1 Pull request

Open MeraX wants to merge 3 commits into main from pytest

Conversation 0 Commits 3 Checks 1

MeraX commented 4 hours ago

Test PR

MeraX added 3 commits 4 hours ago

- Simplify GH Action 22901d7
- Add GH Action f0f4429
- fix test fe85679

All checks have passed  
1 successful check

No conflicts with base branch  
Merging can be performed automatically.

Merge pull request You can also merge this with the command line. [View command line instructions.](#)

## The Matrix Strategy (here with GitHub)

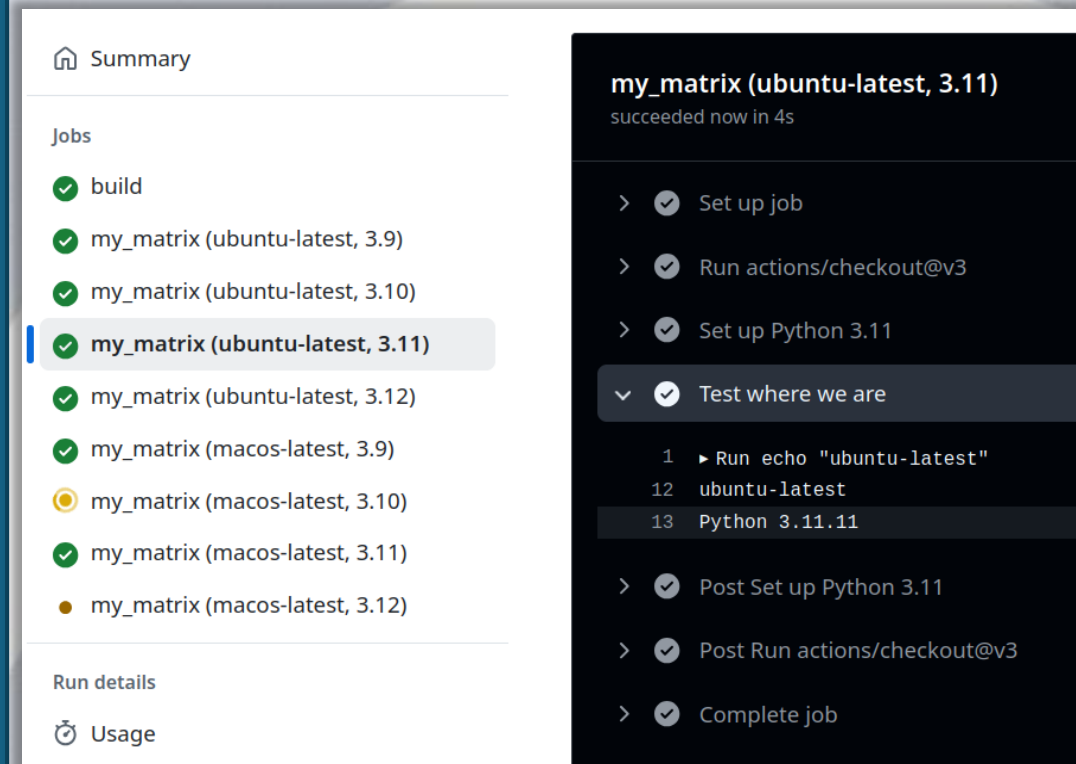
```
# content of .github/workflows/some-name.yml

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 }

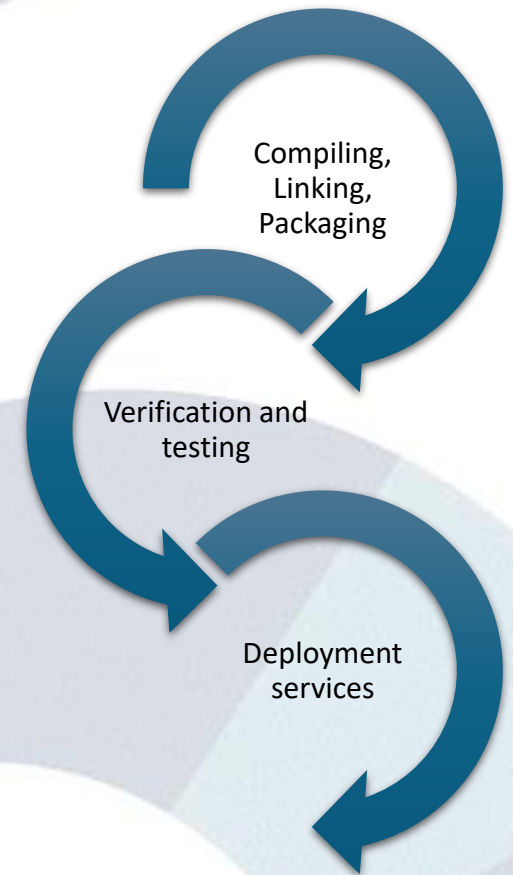
    steps:
      - uses: actions/checkout@v3
      - name: Set up Python ${ matrix.python-version }
        uses: actions/setup-python@v5
        with:
          python-version: ${ matrix.python-version }
      - name: Test where we are
        run: |
          echo "${ matrix.platform }"
          python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



The screenshot shows the GitHub Actions interface for a workflow run. The main panel displays a list of jobs, all of which are successful (indicated by green checkmarks). The selected job is 'my\_matrix (ubuntu-latest, 3.11)'. To the right, a detailed view of this job shows the steps executed: 'Set up job', 'Run actions/checkout@v3', 'Set up Python 3.11', 'Test where we are' (which is expanded to show three sub-steps: 'Run echo "ubuntu-latest"', 'ubuntu-latest', and 'Python 3.11.11'), 'Post Set up Python 3.11', 'Post Run actions/checkout@v3', and 'Complete job'. The overall status is 'succeeded now in 4s'.

## 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 pytest`
- 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, hypothesis-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 =====
```

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

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

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

```
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

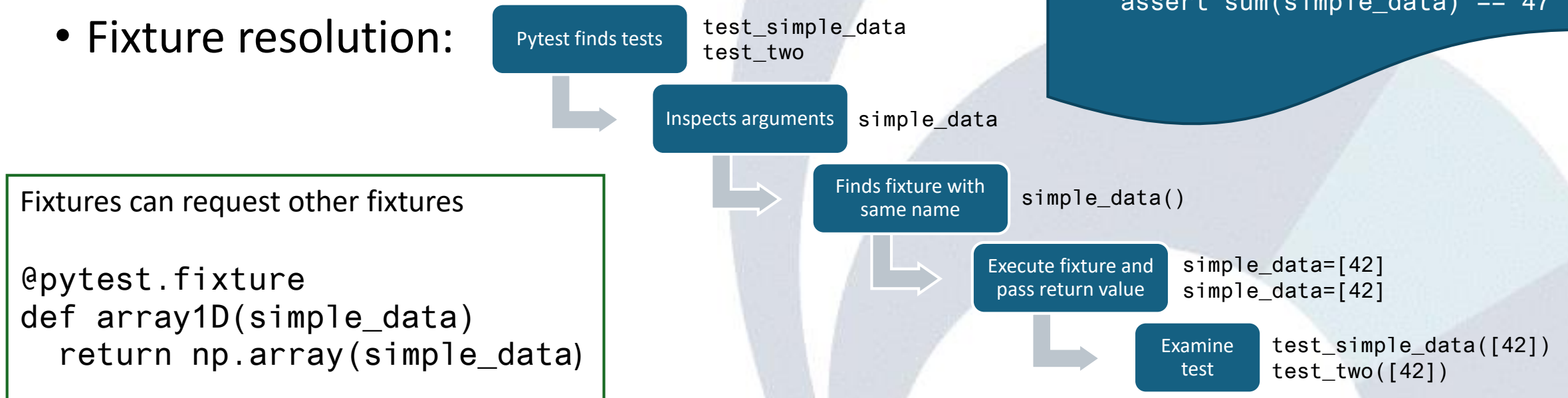
- A Fixture sets up the system state and input data needed for test execution
- Define fixture as factory function with the `@pytest.fixture` decorator
- Fixture resolution:

```
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
```



## 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()

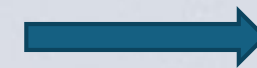
@pytest.mark.gpu
def test_cuda(x_gpu):
    assert x_gpu.is_cuda
    assert not x_gpu.cpu().is_cuda
```

GPU System  
\$ pytest



pass

CPU System  
\$ pytest



fail

CPU System  
\$ pytest -m "not gpu"



pass

## Mark test functions with attributes

- Add metadata on tests with `pytest.mark`

```
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.

```
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]

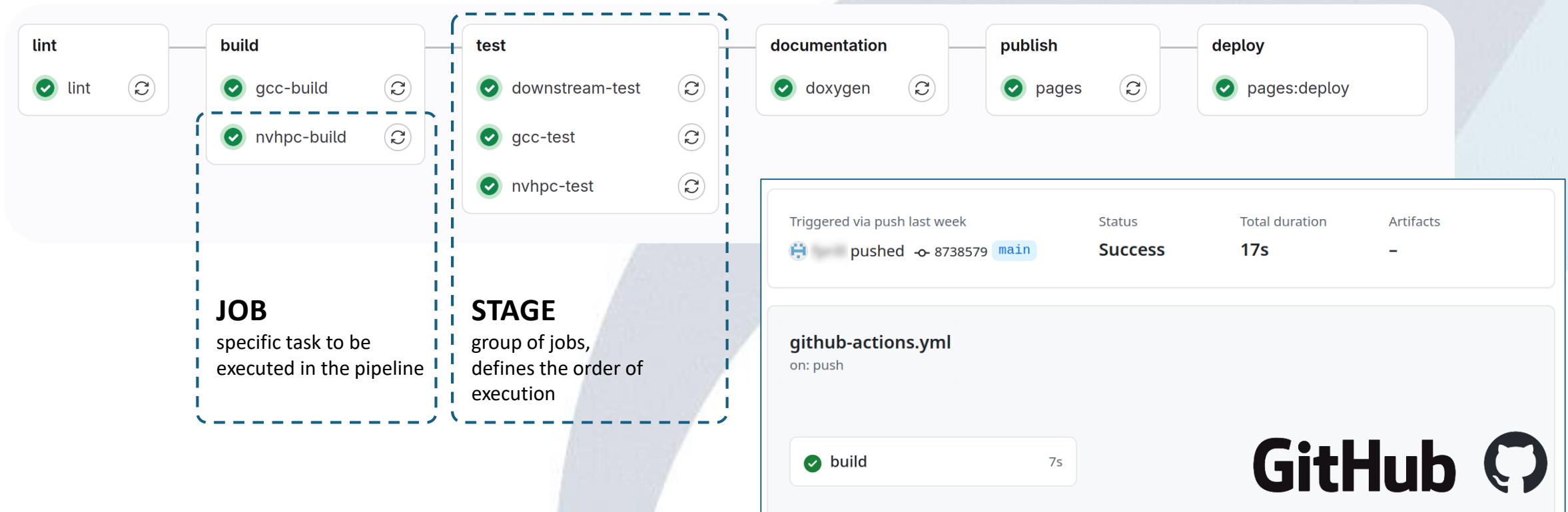
6.3 Setting up a runner (20') [FP]



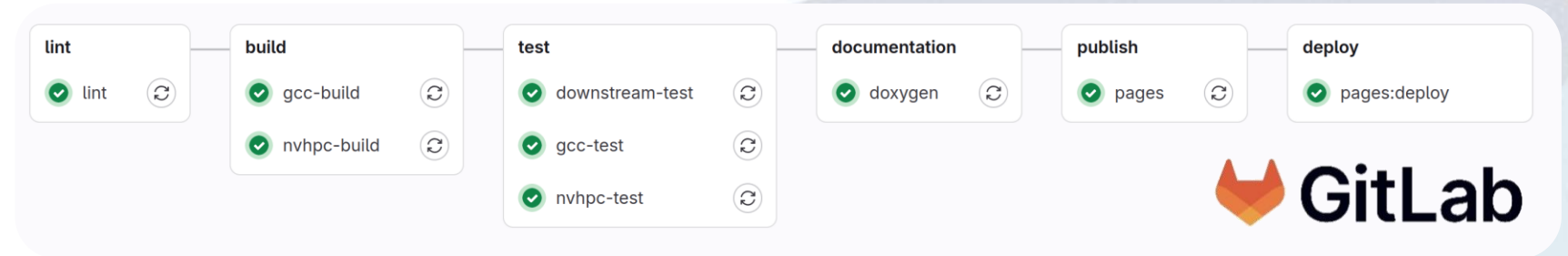
AI generated Image,  
© Marek Jacob 2024

# TYPICAL CI/CD PIPELINE

## Tools for Continuous Integration and Continuous Deployment



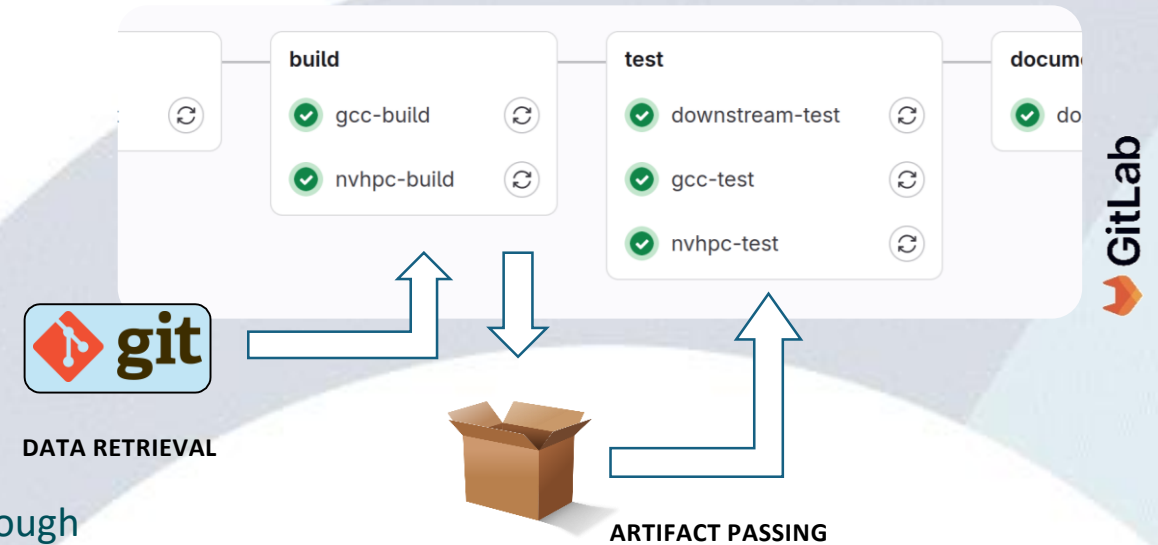
## EXAMPLES FOR CI/CD JOBS



- **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:

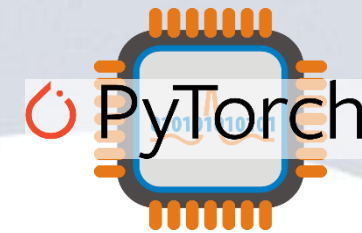
- custom software and dependencies: compiling source code with different compilers or running tests with different versions of Python or PyTorch
- specialized hardware: e.g. test execution on a runner with GPU capability

### 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!**

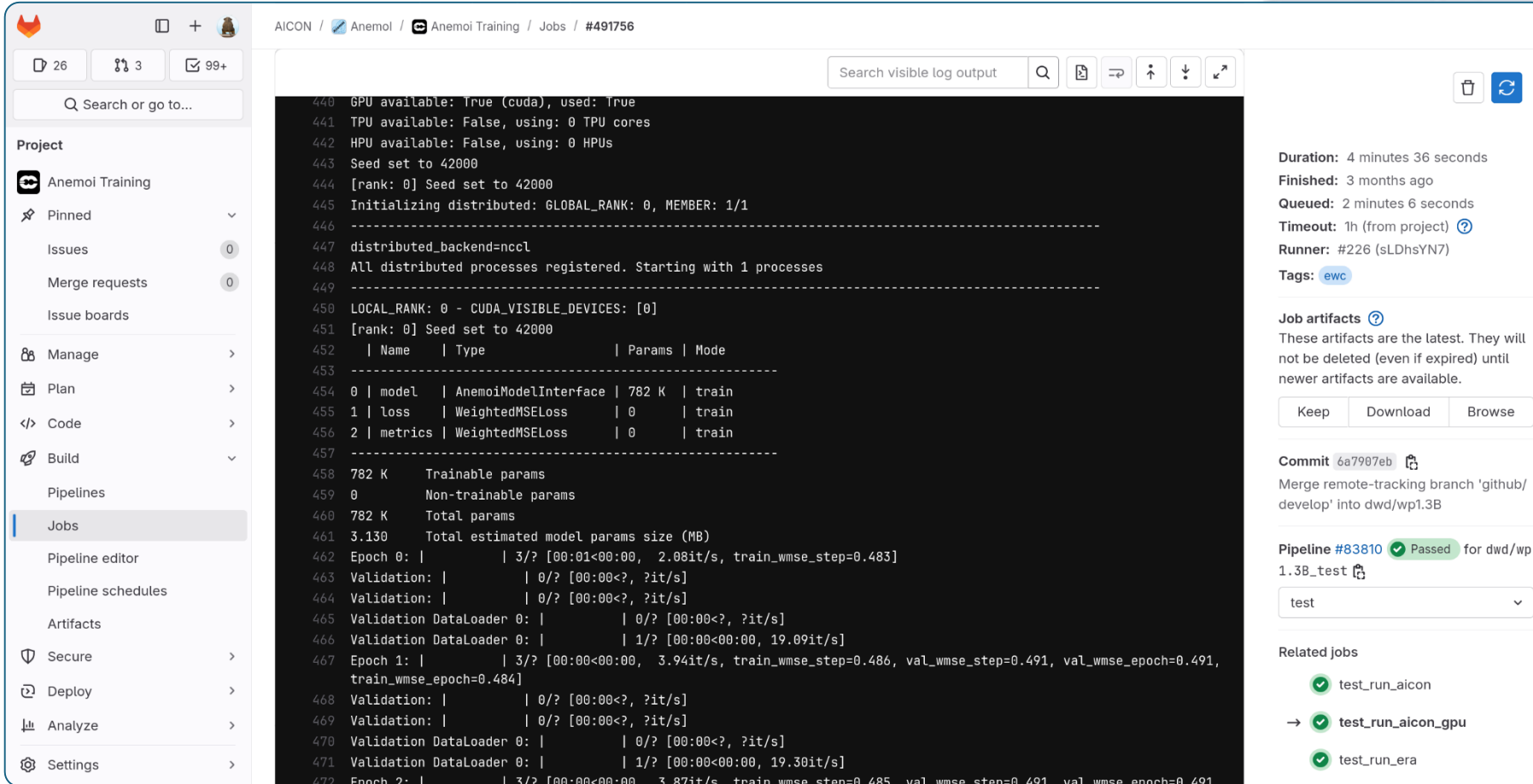


- Setting up your own runner is even more relevant for Gitlab because Gitlab is often set up on a local server while Github offers GitHub-hosted runners.





## EXAMPLE (SCREENSHOT)



The screenshot shows a GitLab CI/CD interface for a project named 'Anemoi Training'. The left sidebar contains navigation options like 'Project', 'Pinned', 'Issues', 'Merge requests', 'Issue boards', 'Manage', 'Plan', 'Code', 'Build', 'Pipelines', 'Jobs', 'Pipeline editor', 'Pipeline schedules', 'Artifacts', 'Secure', 'Deploy', 'Analyze', and 'Settings'. The main area displays the execution log of a job, with a search bar at the top. The log shows the following details:

- GPU available:** true (cuda), used: true
- TPU available:** False, using: 0 TPU cores
- HPU available:** False, using: 0 HPUs
- Seed set to 42000**
- [rank: 0] Seed set to 42000**
- Initializing distributed: GLOBAL\_RANK: 0, MEMBER: 1/1**
- distributed\_backend=nccl**
- All distributed processes registered. Starting with 1 processes**
- LOCAL\_RANK: 0 - CUDA\_VISIBLE\_DEVICES: [0]**
- [rank: 0] Seed set to 42000**

The log also includes a table of parameters and a summary of the job's performance:

Name	Type	Params	Mode
0	model	AnemoiModelInterface	782 K   train
1	loss	WeightedMSELoss	0   train
2	metrics	WeightedMSELoss	0   train

Summary of the job's performance:

- 782 K** Trainable params
- 0** Non-trainable params
- 782 K** Total params
- 3.130** Total estimated model params size (MB)
- Epoch 0:** | 3/? [00:01<00:00, 2.08it/s, train\_wmse\_step=0.483]
- Validation:** | 0/? [00:00<?, ?it/s]
- Validation:** | 0/? [00:00<?, ?it/s]
- Validation DataLoader 0:** | 0/? [00:00<?, ?it/s]
- Validation DataLoader 0:** | 1/? [00:00<00:00, 19.09it/s]
- Epoch 1:** | 3/? [00:00<00:00, 3.94it/s, train\_wmse\_step=0.486, val\_wmse\_step=0.491, val\_wmse\_epoch=0.491]
- Validation:** | 0/? [00:00<?, ?it/s]
- Validation:** | 0/? [00:00<?, ?it/s]
- Validation DataLoader 0:** | 0/? [00:00<?, ?it/s]
- Validation DataLoader 0:** | 1/? [00:00<00:00, 19.30it/s]
- Epoch 2:** | 3/? [00:00<00:00, 3.87it/s, train\_wmse\_step=0.485, val\_wmse\_step=0.491, val\_wmse\_epoch=0.491]

The right sidebar shows job details:

- Duration:** 4 minutes 36 seconds
- Finished:** 3 months ago
- Queued:** 2 minutes 6 seconds
- Timeout:** 1h (from project)
- Runner:** #226 (sLDhsYN7)
- Tags:** ewc
- Job artifacts:** These artifacts are the latest. They will not be deleted (even if expired) until newer artifacts are available.
- Commit:** 6a7907eb
- Pipeline #83810:** Passed for dwd/wp 1.3B\_test
- Related jobs:** test\_run\_aicon, test\_run\_aicon\_gpu, test\_run\_era



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

(Gitlab, execution  
environment: EWC)

## 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.



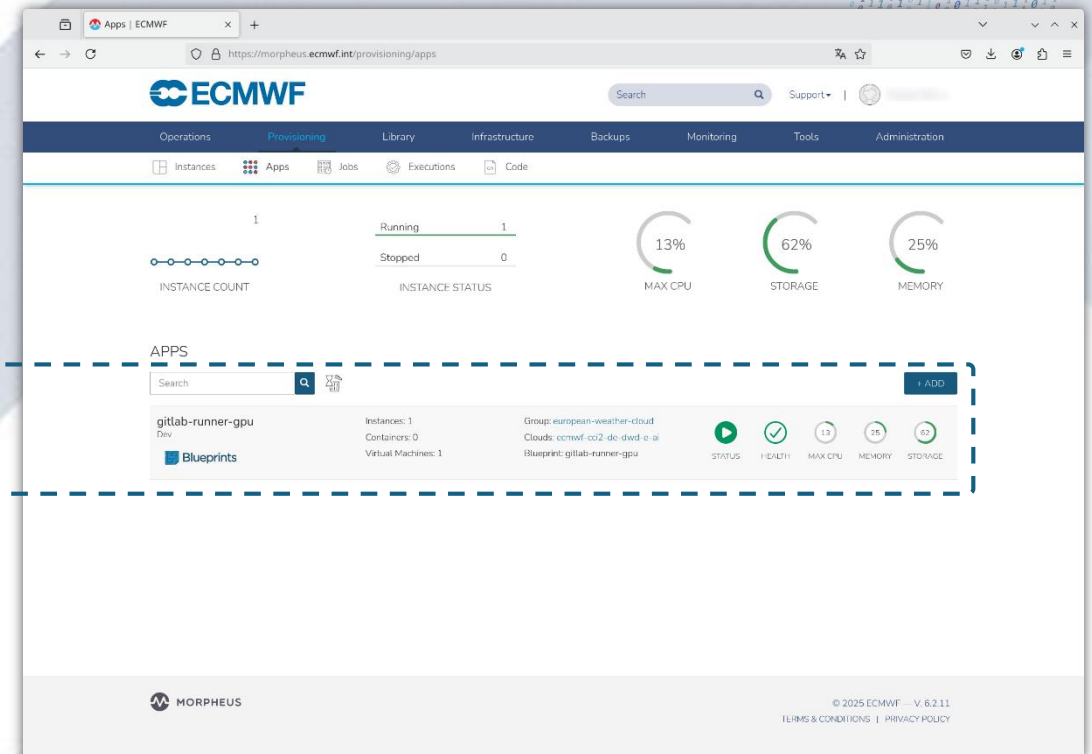
# EWC APPLICATION BLUEPRINT

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

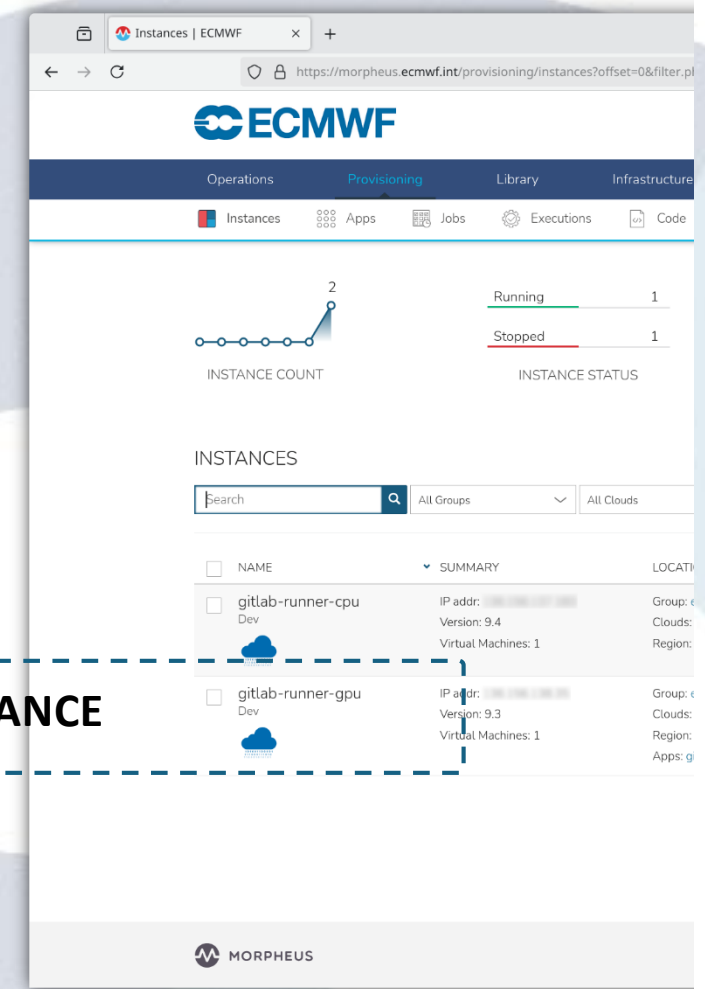


## INITIALIZING THE VM

- **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)

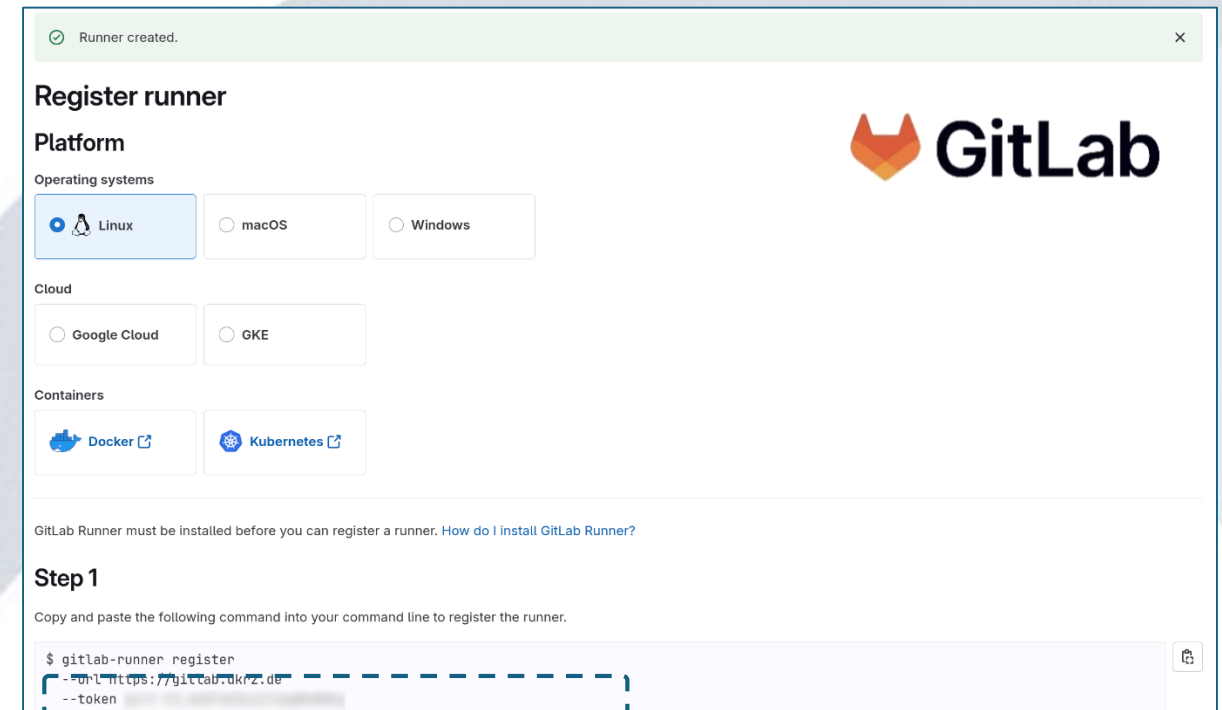
### Assign the new Project Runner to the Gitlab Project (navigate: Settings/CICD/Runners)

- 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



The image shows the GitLab 'Register runner' web interface. At the top, a green banner says 'Runner created.' with a close button. The main heading is 'Register runner' with the GitLab logo. Under 'Platform', there are three sections: 'Operating systems' with radio buttons for Linux (selected), macOS, and Windows; 'Cloud' with radio buttons for Google Cloud and GKE; and 'Containers' with checkboxes for Docker and Kubernetes. Below this, a note states 'GitLab Runner must be installed before you can register a runner. [How do I install GitLab Runner?](#)'. The 'Step 1' section instructs to 'Copy and paste the following command into your command line to register the runner.' and shows a terminal command: `$ gitlab-runner register`  
`-url https://gitlab.ukfz.de`  
`--token [REDACTED]`

**CI/CD RUNNER TOKEN**



## 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`:

```
test_run_aicon:
  stage: test
  tags:
    - ewc
  image: anemoui-gpu:latest
  script:
    - pytest -v -s tests/integration_aicon.py
  artifacts:
    paths:
      - output_training/mr13/checkpoint/*/inference-last.ckpt
  expire_in: 1 hour
```



### 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`).

```
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 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>

## Further Information on E-AI

- Slides available at GitHub

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

- Recording will be available at EUMETNET SharePoint

<https://tInt19059.sharepoint.com/:f:/r/sites/E-AI/Shared%20Documents/Tutorials>

- Register for E-AI updates and SharePoint Access:

[marek.jacob@eumetnet.eu](mailto:marek.jacob@eumetnet.eu)

- Contacts:

- Florian Prill:

[Florian.Prill@dwd.de](mailto:Florian.Prill@dwd.de)

- Marek Jacob:

[Marek.Jacob@dwd.de](mailto:Marek.Jacob@dwd.de)