Day6 - Continuous Integration

02476 Machine Learning Operations

Nicki Skafte Detlefsen, Associate Professor, DTU Compute

January 2026

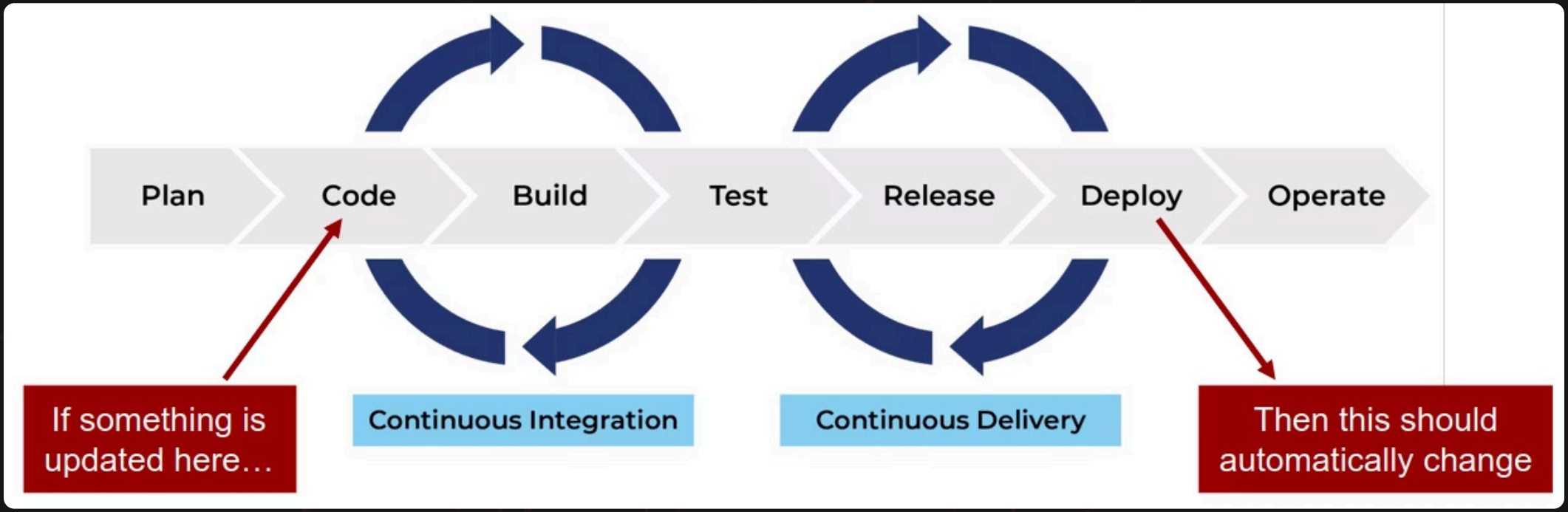
50 people and 5 groups are marked in my little black book...

Why you should care about today

3 years ago, the day before this lecture, the internet went down for a couple of hours because someone f..ked up their continues integration at Fastly

Continues X

🔥Term refers to a set of software practices for automating tedious tasks and make sure changed in a pipeline are continuously propagated through the pipeline



The different types

CI	CD	CML
Continues Integration	Continues Deployment	Continues Machine Learning
💡How to automatically secure that code does not break during development?	💡 How to get your code/application to the user automatically? + monitor life cycle	💡 How to automatically retrain machine learning models when data and code changes?
💻App independent concept	💻App dependent concept	💻Specific to ML applications

MLOps levels

The **Maturity model** overall describes the DevOps practices to run a successful MLOps environment.

Intended to identify gaps in an existing organization's attempt to implement such an environment.

💡 Estimate the scope of the work for new engagements.

💡 Establish realistic success criteria.

💡 Identify deliverables you'll hand over at the conclusion of the engagement.

Level	Description	Highlights	Technology
0	No MLOps	• Difficult to manage full machine learning model lifecycle • The teams are disparate and releases are painful • Most systems exist as "black boxes," little feedback during/post deployment	• Manual builds and deployments • Manual testing of model and application • No centralized tracking of model performance • Training of model is manual
1	DevOps but no MLOps	• Releases are less painful than No MLOps, but rely on Data Team for every new model • Still limited feedback on how well a model performs in production • Difficult to trace/reproduce results	• Automated builds • Automated tests for application code
2	Automated Training	• Training environment is fully managed and traceable • Easy to reproduce model • Releases are manual, but low friction	• Automated model training • Centralized tracking of model training performance • Model management
3	Automated Model Deployment	• Releases are low friction and automatic • Full traceability from deployment back to original data • Entire environment managed: train > test > production	• Integrated A/B testing of model performance for deployment • Automated tests for all code • Centralized tracking of model training performance
4	Full MLOps Automated Operations	• Full system automated and easily monitored • Production systems are providing information on how to improve and, in some cases, automatically improve with new models • Approaching a zero-downtime system	• Automated model training and testing • Verbose, centralized metrics from deployed model

This lecture: continues integration

Core task:

🔥How to automatically secure that code does not break during development? 🔥

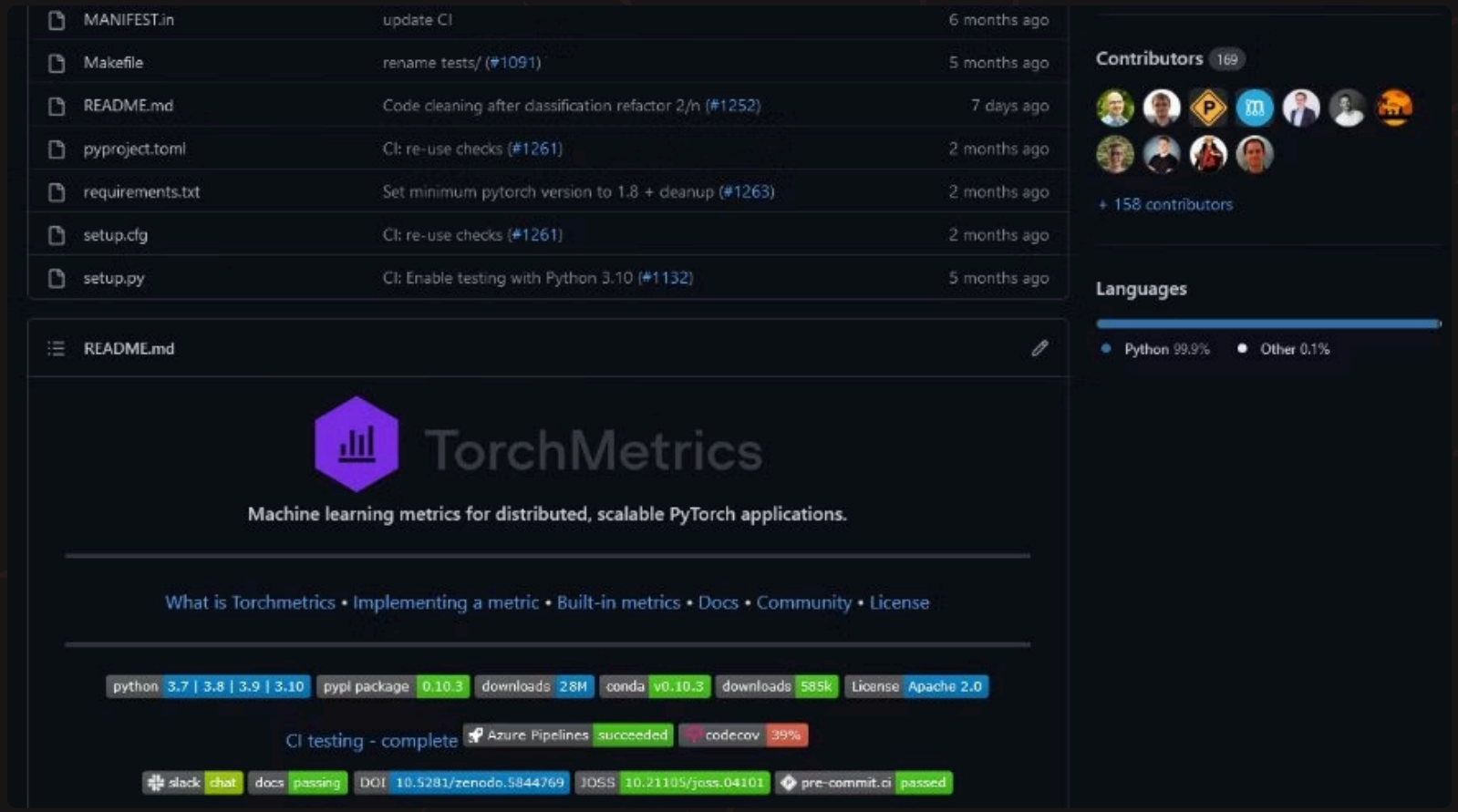
3 steps to do this:

💡Use version control: Frequently committing code to a shared repository

💡Write (unit)test for your code: Should capture unwanted bugs in your code

💡Automate build + testing: Automatically run test so code cannot be merged without working

A small case study for continuous integration



CI step 1: version control

User version control:

💡Code changes are tracked

💡Branches for parallel work

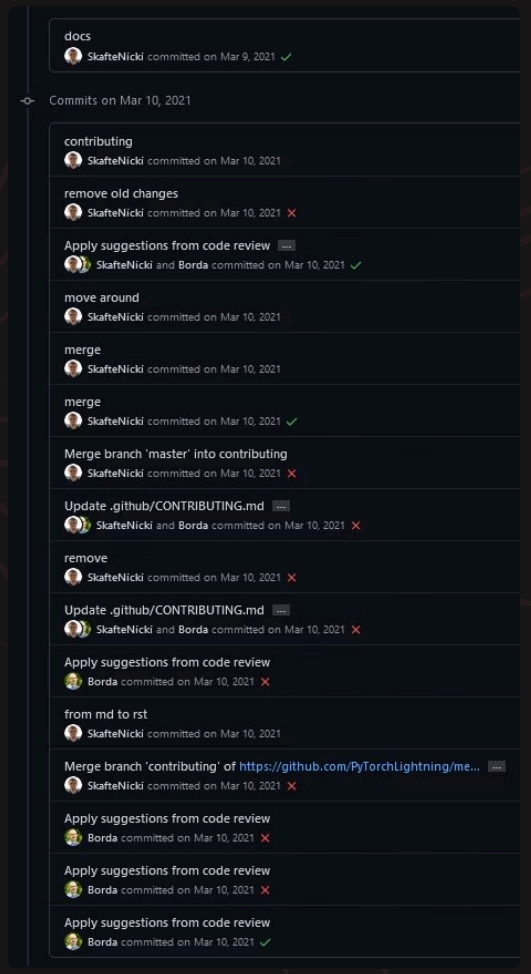
Commit frequently:

💡Catch errors sooner than later

💡Revert back easily to when things were working

💡Merging can be done automatically

Create it → Break it → Fix it →...



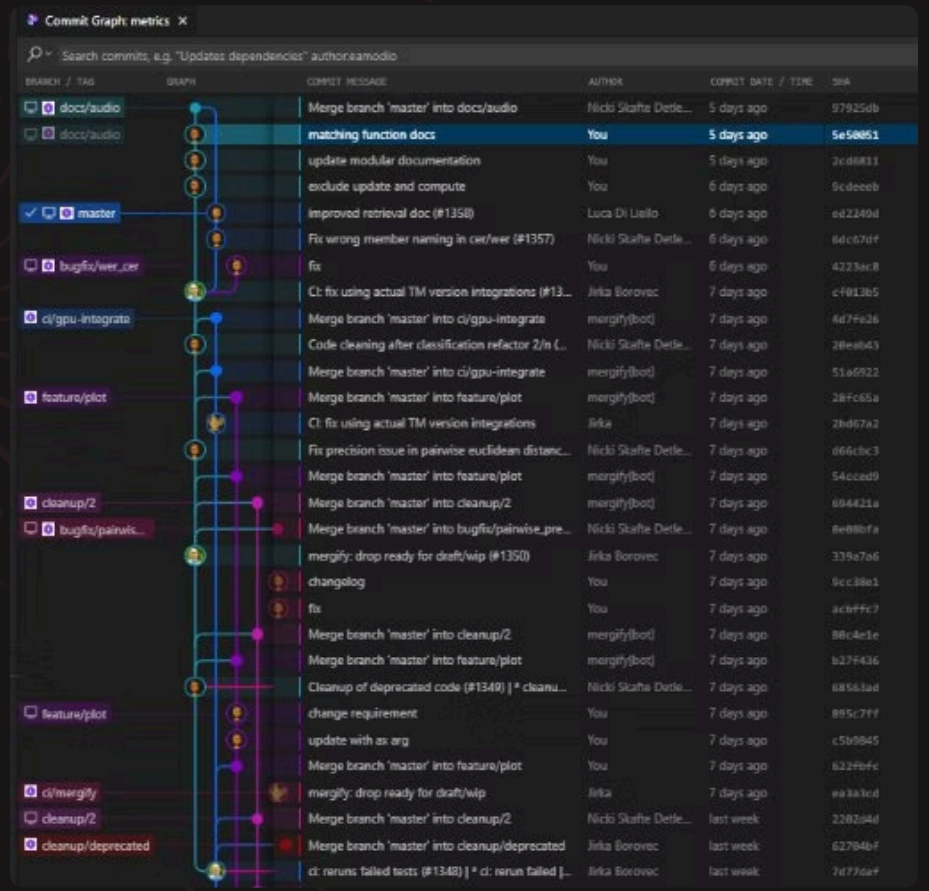
CI step 1: Use branches

Parallel workflow

Experimental features changes are kept away from master/main

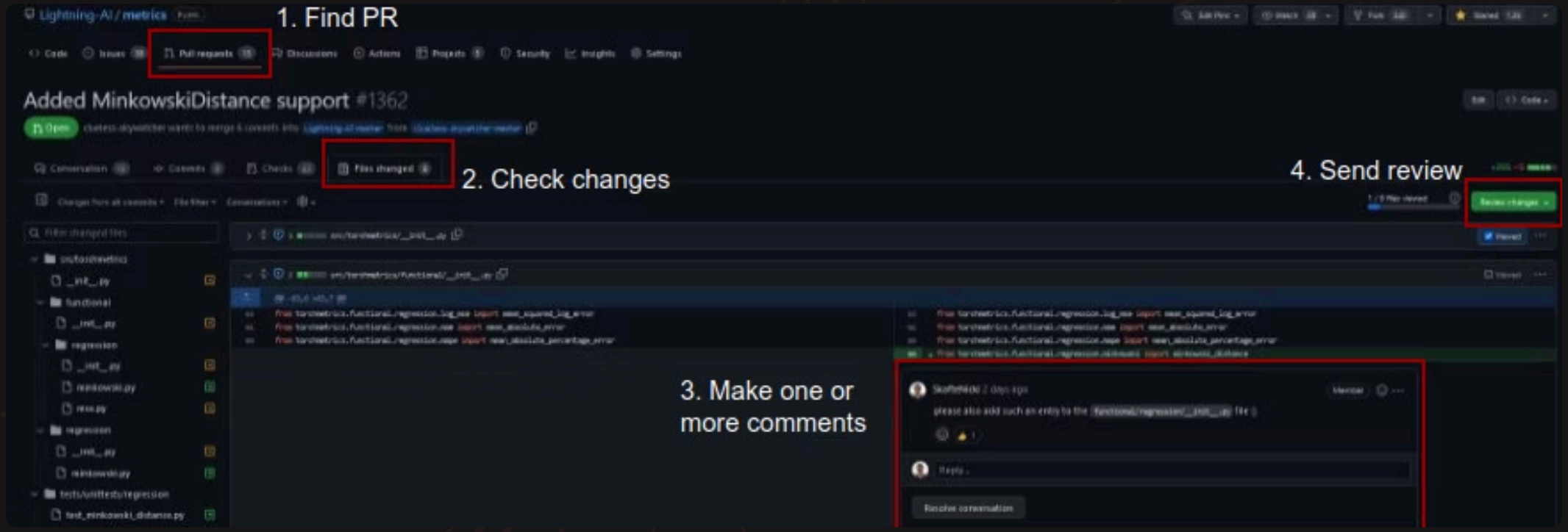
Recommend extensions for VS code:

- GitLens
- Git Graph
- GitHub Pull Requests



CI step 1: Use pull requests

⚠️ No commit can be pushed to master without being in a pull request



CI step 1: pre-commit

✅Check that everything is up to standard before commits are created

.pre-commit-config.yaml

```yaml
default_language_version:
  python: python3

repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.4.0
    hooks:
      - id: end-of-file-fixer
      - id: trailing-whitespace
      # - id: check-json
      # - id: check-yaml
      - id: check-toml
      - id: check-docstring-first
      - id: check-executables-have-shebangs
      - id: check-case-conflict
      - id: detect-private-key

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.3
    hooks:
      - id: ruff
        args: [--fix, --exit-non-zero-on-fix]

  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.1.3
    hooks:
      - id: ruff-format

  - repo: https://github.com/codespell-project/codespell
    rev: v2.2.5
    hooks:
      - id: codespell
        additional_dependencies: [tomli]
```

```
dtu_mlops on  main [!?⇡] via  v3.11.5 (mlops)
❯ git commit -m "implementation of client"
fix end of files.....................................................Failed
- hook id: end-of-file-fixer
- exit code: 1
- files were modified by this hook

Fixing s8_monitoring/exercise_files/client.py

trim trailing whitespace.....................................................Passed
check toml.....................................(no files to check)Skipped
check docstring is first.....................................................Passed
check that executables have shebangs.....................................Passed
check for case conflicts.....................................................Passed
detect private key.....................................................Passed
ruff.....................................................Failed
- hook id: ruff
- exit code: 1
- files were modified by this hook

s8_monitoring\exercise_files\client.py:17:12: S113 Probable use of requests call without timeout
Found 2 errors (1 fixed, 1 remaining).

ruff-format.....................................................Passed
codespell.....................................................Passed
markdownlint-docker.....................................(no files to check)Skipped
```

CI step 2: write tests

Tests are the cornerstones of continuous integration

💡 *unit tests* are arguable the most important.

💡A single unittest, tests a small part of your code

💡By testing code in small pieces, bugs are easier to find

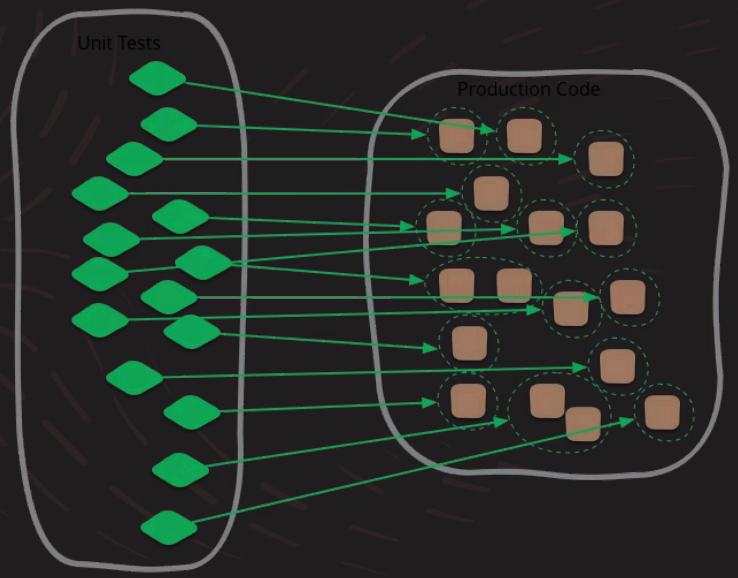
Other test types worth considering:

🔥Integration tests

🔥Regression tests

🔥Performance tests

🔥Security tests



CI step 2: write tests

💡 By Python convention your source code should either be

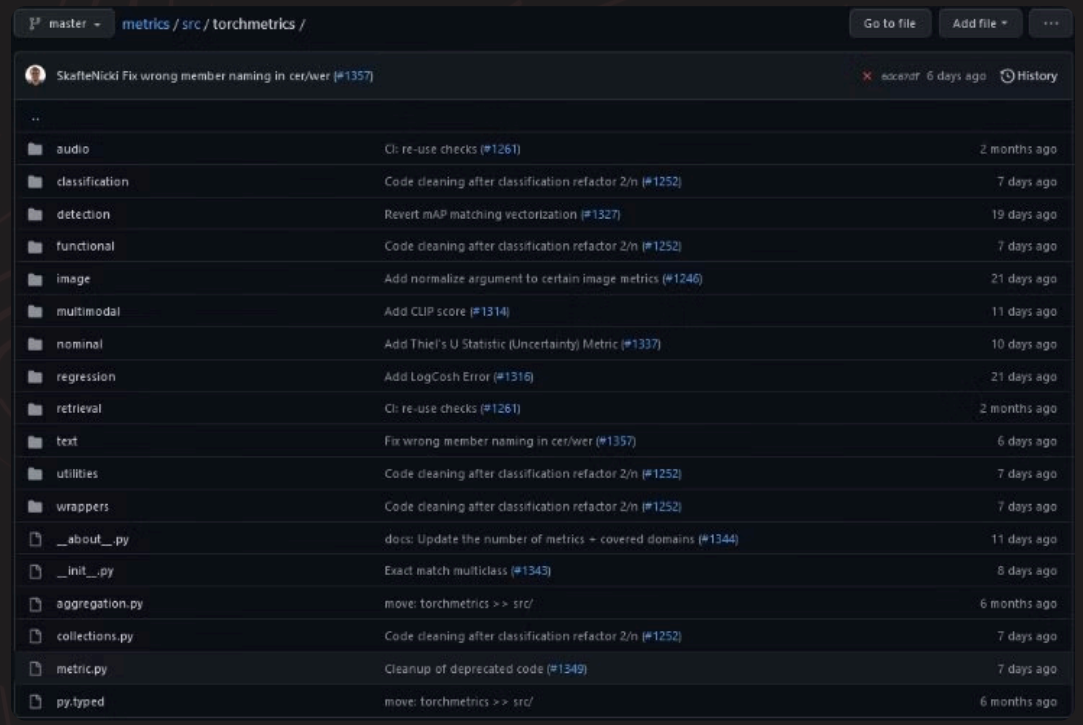
src/<project_name>

(src-layout)

or

<project_name>

(flat-layout)

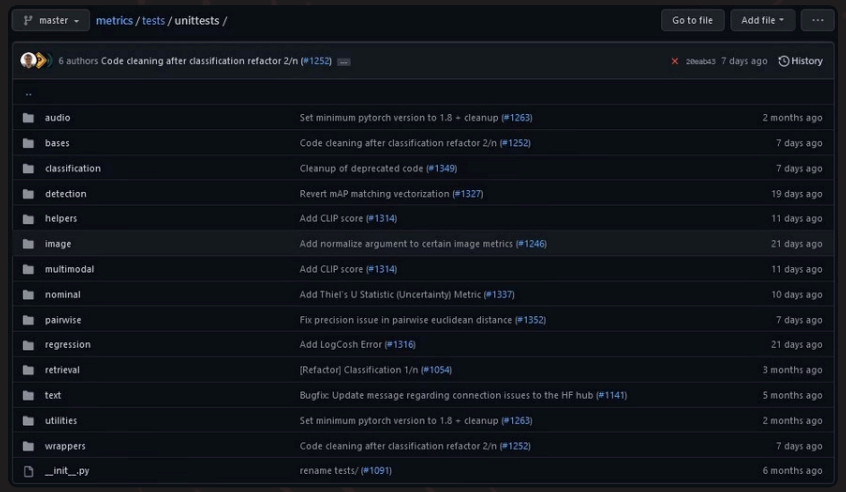


CI step 2: write tests

For tests, the convention is to either place the tests in a separate tests folder, or put the tests in the same folder as the function/class/submodule they are testing.

```
├─── README.md
├─── src/
│    ├─── __init__.py
│    └─── important_functions.py
├─── tests/
│    ├─── __init__.py
│    └─── test_important_functions.py
```

```
├─── README.md
├─── src/
│    ├─── __init__.py
| ├─── submodule/
│    ├──────────── __init__.py
│    ├──────────── important_functions.py
│    └──────────── test_important_functions.py
```



CI step 2: write tests

💡 In python, we recommend using the **pytest** framework.

💡 Test are simple functions that start with *test_* and uses *assert*

```python
import torch
from torch.nn.functional import mse_loss

def test_mse_loss_zeros():
    # (0 - 0)**2 = 0
    assert mse_loss(torch.zeros(1,), torch.zeros(1,)) == 0

def test_mse_loss_ones():
    # (1 - 0)**2 = 1
    assert mse_loss(torch.ones(1,), torch.zeros(1,)) == 0
```

CI step 2: write tets

Test can be simple...

```
def test_warning_on_nan(tmpdir):
    preds = torch.randint(3, size=(20, ))
    target = torch.randint(3, size=(20, ))

    with pytest.warns(
        UserWarning,
        match='.* nan values found in confusion matrix have been replaced with zeros.',
    ):
        confusion_matrix(preds, target, num_classes=5, normalize='true')
```

CI step 2: write test

or complicated

```
@pytest.mark.parametrize("normalize", ['true', 'pred', 'all', None])
@pytest.mark.parametrize(
    "preds, target, sk_metric, num_classes, multilabel",
    [(_input_binary_prob.preds, _input_binary_prob.target, _sk_cm_binary_prob, 2, False),
     (_input_binary_logits.preds, _input_binary_logits.target, _sk_cm_binary_prob, 2, False),
     (_input_binary.preds, _input_binary.target, _sk_cm_binary, 2, False),
     (_input_mlb_prob.preds, _input_mlb_prob.target, _sk_cm_multilabel_prob, NUM_CLASSES, True),
     (_input_mlb_logits.preds, _input_mlb_logits.target, _sk_cm_multilabel_prob, NUM_CLASSES, True),
     (_input_mlb.preds, _input_mlb.target, _sk_cm_multilabel, NUM_CLASSES, True),
     (_input_mcls_prob.preds, _input_mcls_prob.target, _sk_cm_multiclass_prob, NUM_CLASSES, False),
     (_input_mcls_logits.preds, _input_mcls_logits.target, _sk_cm_multiclass_prob, NUM_CLASSES, False),
     (_input_mcls.preds, _input_mcls.target, _sk_cm_multiclass, NUM_CLASSES, False),
     (_input_mdmc_prob.preds, _input_mdmc_prob.target, _sk_cm_multidim_multiclass_prob, NUM_CLASSES, False),
     (_input_mdmc.preds, _input_mdmc.target, _sk_cm_multidim_multiclass, NUM_CLASSES, False)]
)
class TestConfusionMatrix(MetricTester):

    @pytest.mark.parametrize("ddp", [True, False])
    @pytest.mark.parametrize("dist_sync_on_step", [True, False])
    def test_confusion_matrix(
        self, normalize, preds, target, sk_metric, num_classes, multilabel, ddp, dist_sync_on_step
    ):
        self.run_class_metric_test(
            ddp=ddp,
            preds=preds,
            target=target,
            metric_class=ConfusionMatrix,
            sk_metric=partial(sk_metric, normalize=normalize),
            dist_sync_on_step=dist_sync_on_step,
            metric_args={
                "num_classes": num_classes,
                "threshold": THRESHOLD,
                "normalize": normalize,
                "multilabel": multilabel
            }
        )
```

Exercise 1: The data contract

> Imagine you are building a model to predict **House Prices**. Aside from 'the code doesn't crash,' list 3 automated checks you would run on the **input data** before training starts to ensure the data is healthy."

Example dataset:

www.kaggle.com

Housing Prices Dataset

Housing Prices Prediction – Regression Problem

Solution: Exercise 1 - The Data Contract

Automated checks for healthy input data before model training:

Data Type & Range Validation

Confirm numerical features like 'square footage' and 'number of bedrooms' fall within plausible ranges (e.g., non-negative) and maintain their expected data types.

Missing Values Threshold

Verify that no essential columns (e.g., 'price', 'address') contain missing values above a predefined, acceptable percentage, preventing incomplete data from skewing results.

Categorical Consistency

Validate categorical features such as 'neighborhood' or 'property type' against a list of known, expected values to catch typos or new, unhandled categories.

Exercise 2: The Smoke Test

> You've just finished training a new model version in your CI pipeline. What is the very first thing you should do with that model file before saving it to a Model Registry? Write down one 'sanity check'.

Solution: Exercise 2 - The Smoke Test

Model Output Validation

Load the newly trained model and perform a prediction on a small, known-good data sample. Verify that the model's output is not empty or erroneous, its data type matches expectations (e.g., float, integer), and it falls within a sensible range for the problem (e.g., a positive house price, a probability between 0 and 1).

Exercise 3: ML Bug Hunting

> A developer pushes code that passes all unit tests, but the model's accuracy on the test set drops from 95% to 40%. Discuss with your partner: Should the CI pipeline 'fail' this build? How would you write a test to catch this automatically?"

Solution: Exercise 3 - ML Bug Hunting

Implement a Model Performance Test

Introduce an automated regression test in the CI pipeline. This test should:

- Load the newly trained model.
- Evaluate its performance on a stable, versioned test dataset.
- Compare key metrics (e.g., accuracy, F1-score) against a predefined threshold (e.g., accuracy >= 90%).
- Fail the build if performance falls below the threshold, blocking deployment.

CI step 2: execute locally

```
(lightning) C:\Users\nsde\Documents\metrics>pytest tests\unittests\regression\test_mean_error.py
============================================================================================================================================ test session starts ============================================================================================================================================
platform win32 -- Python 3.8.13, pytest-6.2.5, py-1.11.0, pluggy-1.0.0
rootdir: C:\Users\nsde\Documents\metrics, configfile: setup.cfg
plugins: cov-4.0.0, doctestplus-0.12.1, timeout-2.1.0
collected 116 items

tests\unittests\regression\test_mean_error.py ssssssssssssss.............sssssssssssss.....................................................xxxxxxxx.................

============================================================================================================================================ warnings summary ============================================================================================================================================
..\..\Anaconda3\envs\lightning\lib\site-packages\_pytest\config\__init__.py:1183
  C:\Users\nsde\Anaconda3\envs\lightning\lib\site-packages\_pytest\config\__init__.py:1183: PytestDeprecationWarning: The --strict option is deprecated, use --strict-markers instead.
    self.issue_config_time_warning(

-- Docs: https://docs.pytest.org/en/stable/warnings.html
============================================================================================================================ 80 passed, 28 skipped, 8 xfailed, 1 warning in 16.44s ========
```

.	Test passed
F	Test failed
s	Test skipped (`pytest.skipif`, `pytest.skip`)
x	Test was expected to fail (`pytest.xfail`)

Do you remember to do this before each commit?

Let's automate doing it instead

CI step 3: Automating stuff

What can be automated: EVERYTHING🤖

💡Unit testing

💡Integration testing

💡Documentation creation

💡Linters (style formatting)

💡Security checks

💡Code coverage

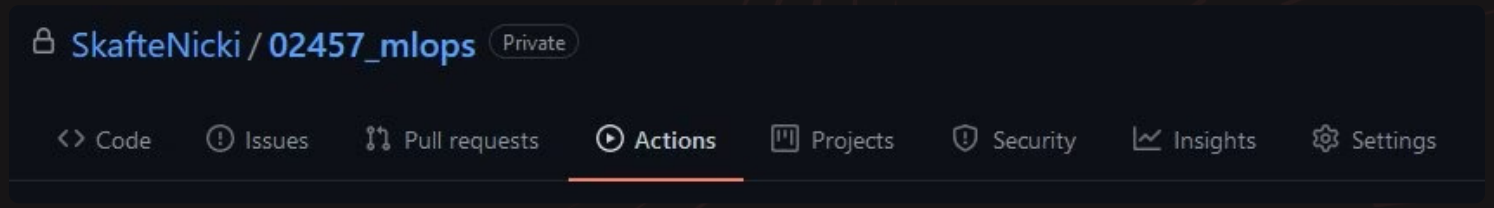
💡Custom checks...

Only your imagination is the limit...

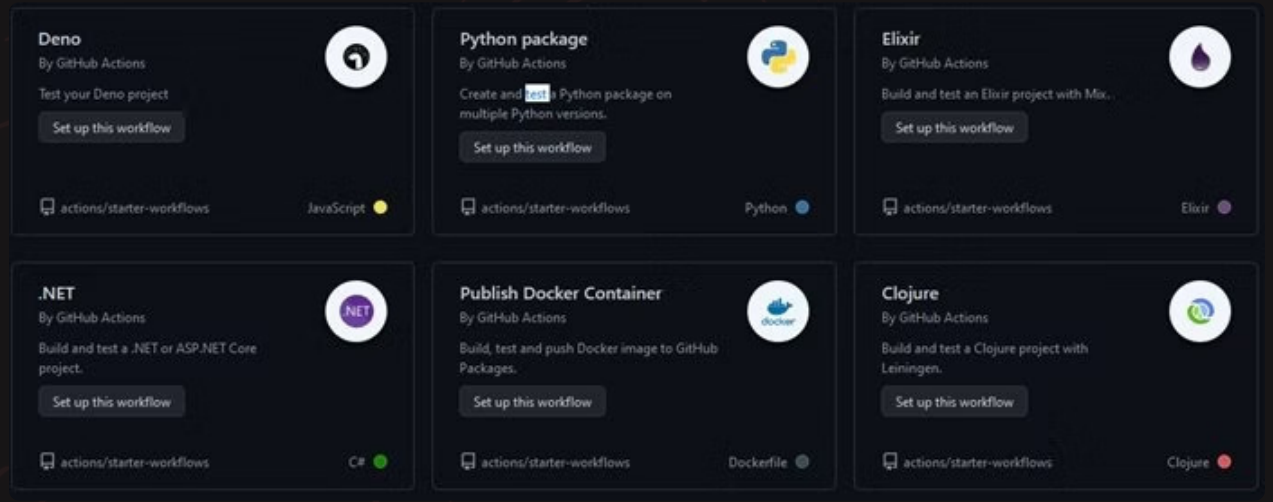
CI step 3: Github actions

Build-in continuous integration in Github.

Free 2000 automation minutes/month (public repository)



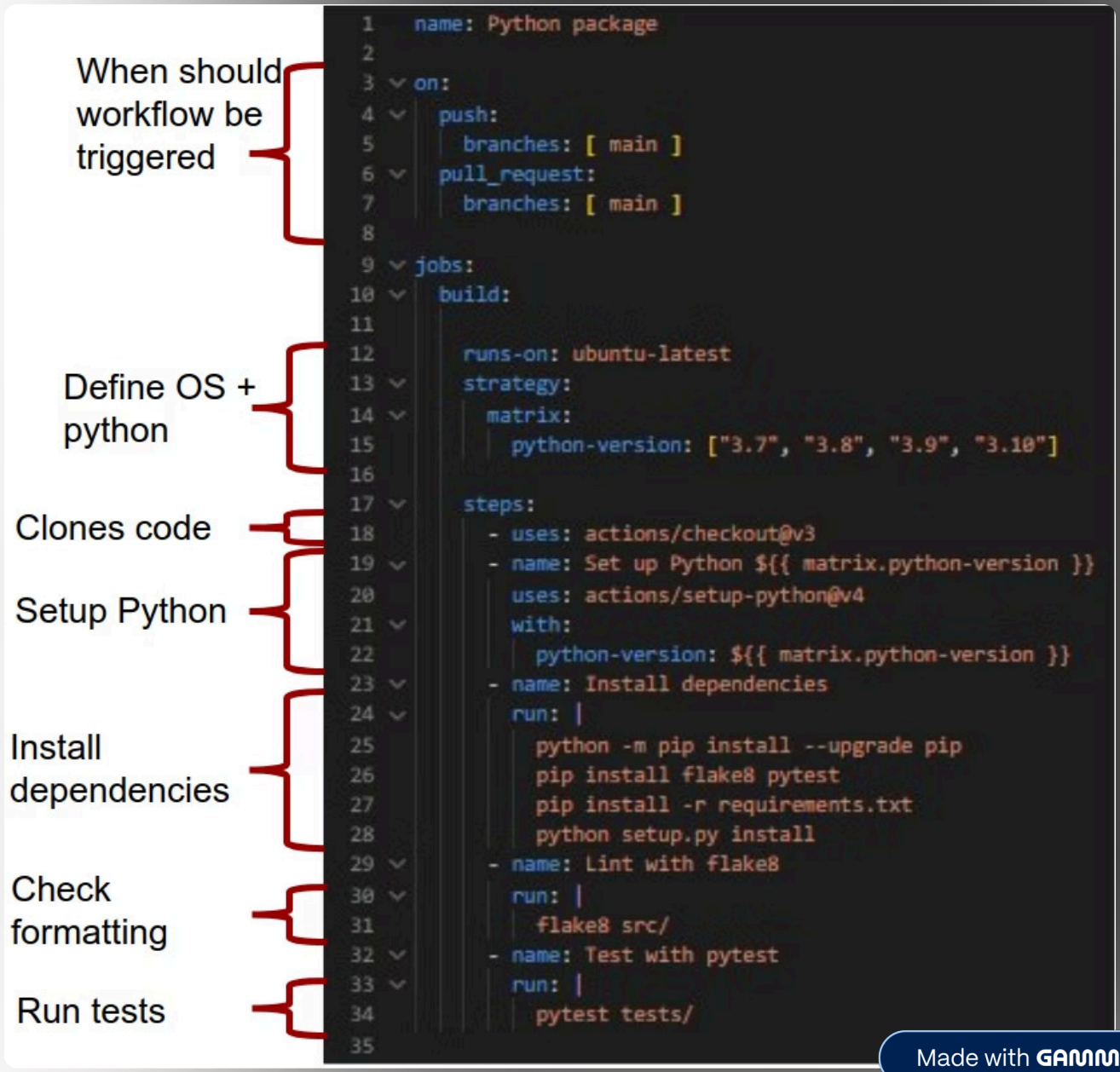
Many ready to go workflows



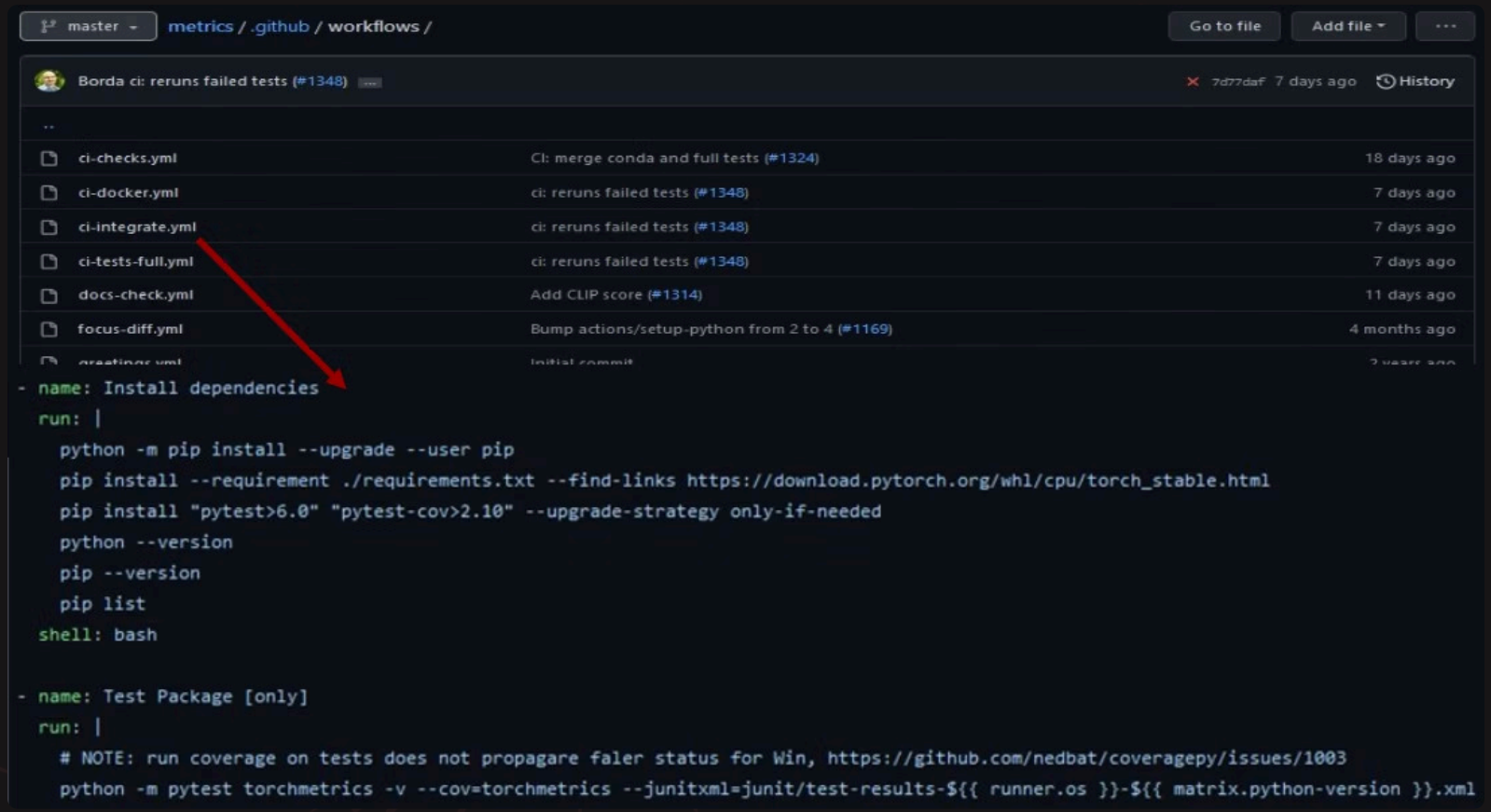
CI step 3: workflow files

Workflow files are a set of instructions that should be executed on a virtual machine hosted by Github

You can have one or many workflow files (runs in parallel)



CI step 3: workflow files



CI step 3: workflow files

✅43 checks in total

Test a combination of

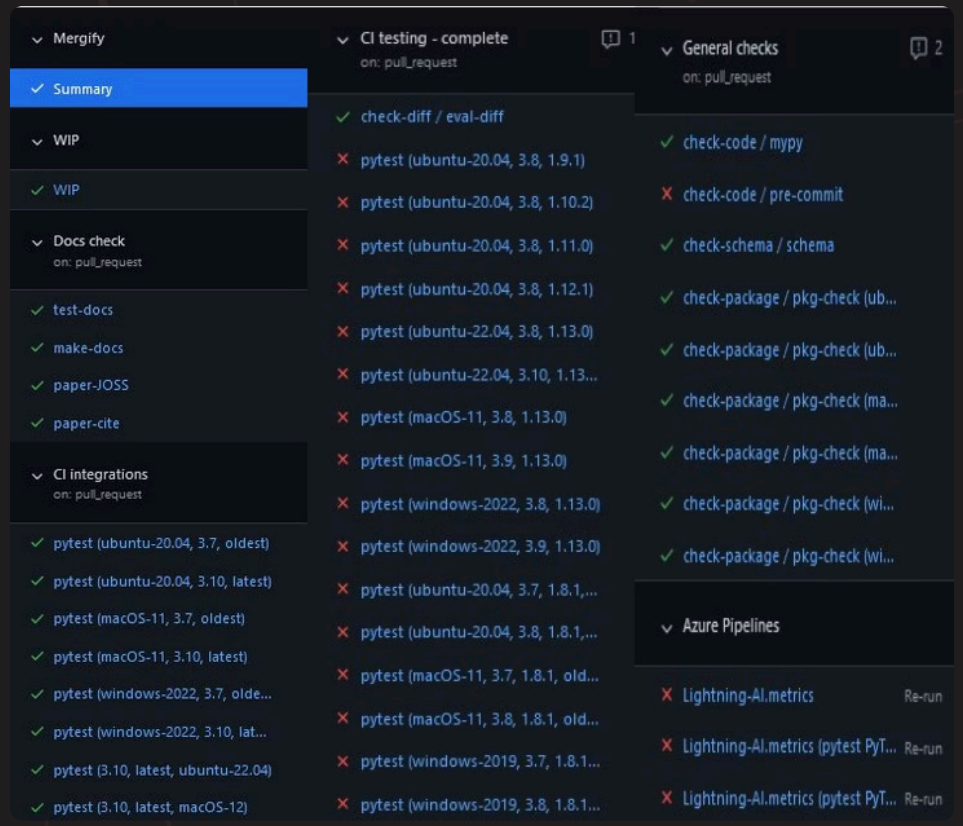
💡Hardware setup

💡Operating system

💡Python version

💡Core dependencies

Runs unit tests, build documentation, test coverage, linting of code, package installer etc.



CI step 3: Code is checked before merging

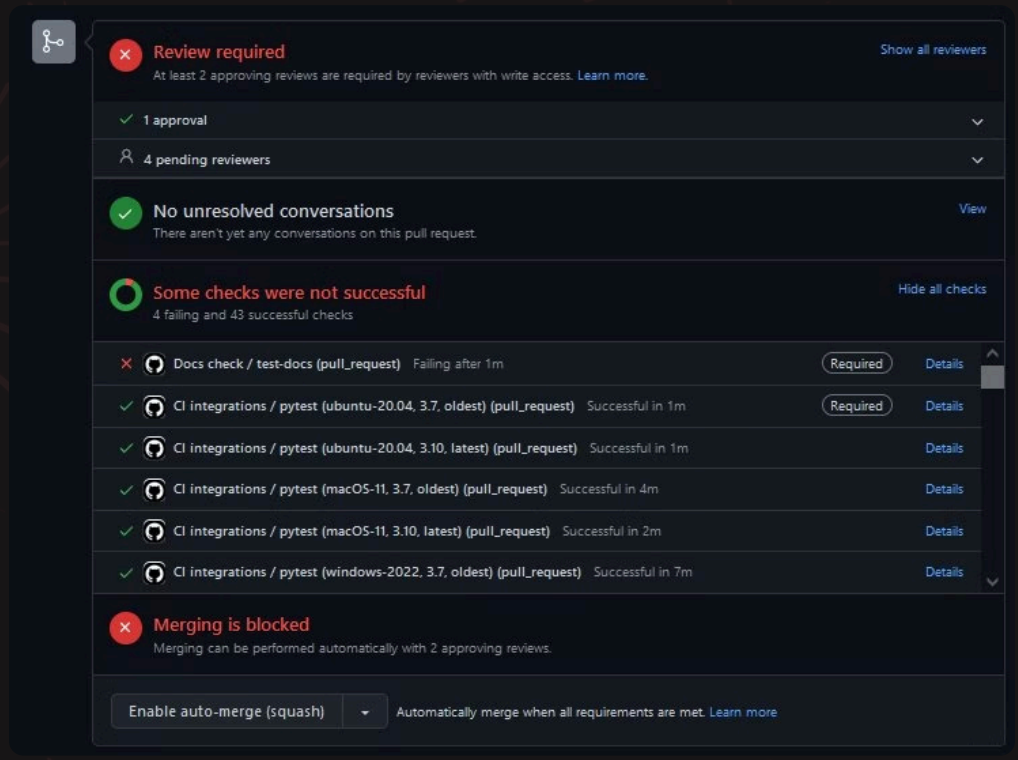
Branch protection rules:

⚠️All/some tests should pass

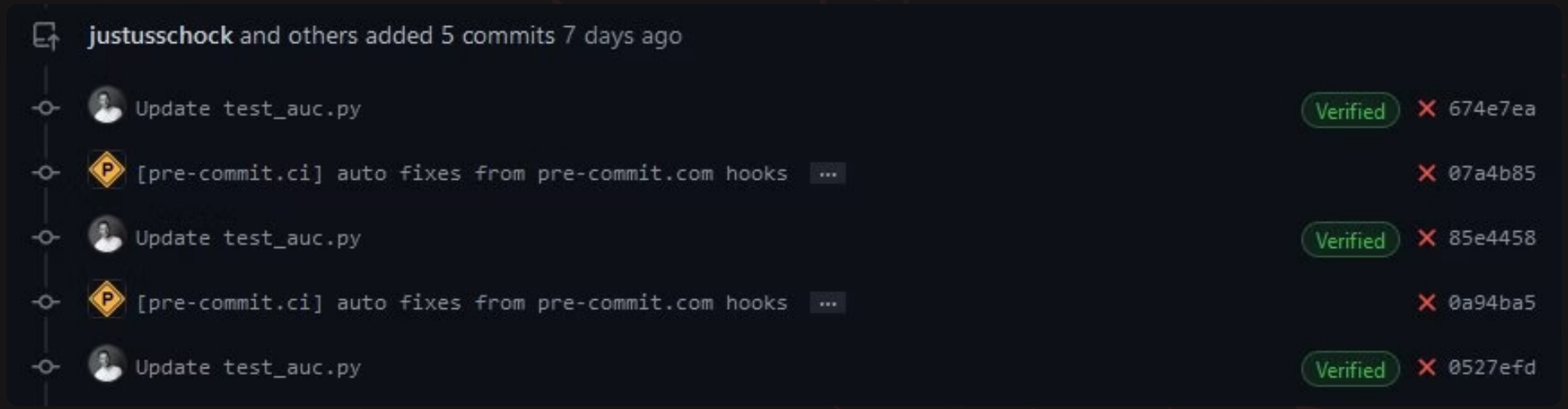
⚠️At least x core members need to approve

⚠️Comments should be taken care of

View more **here**

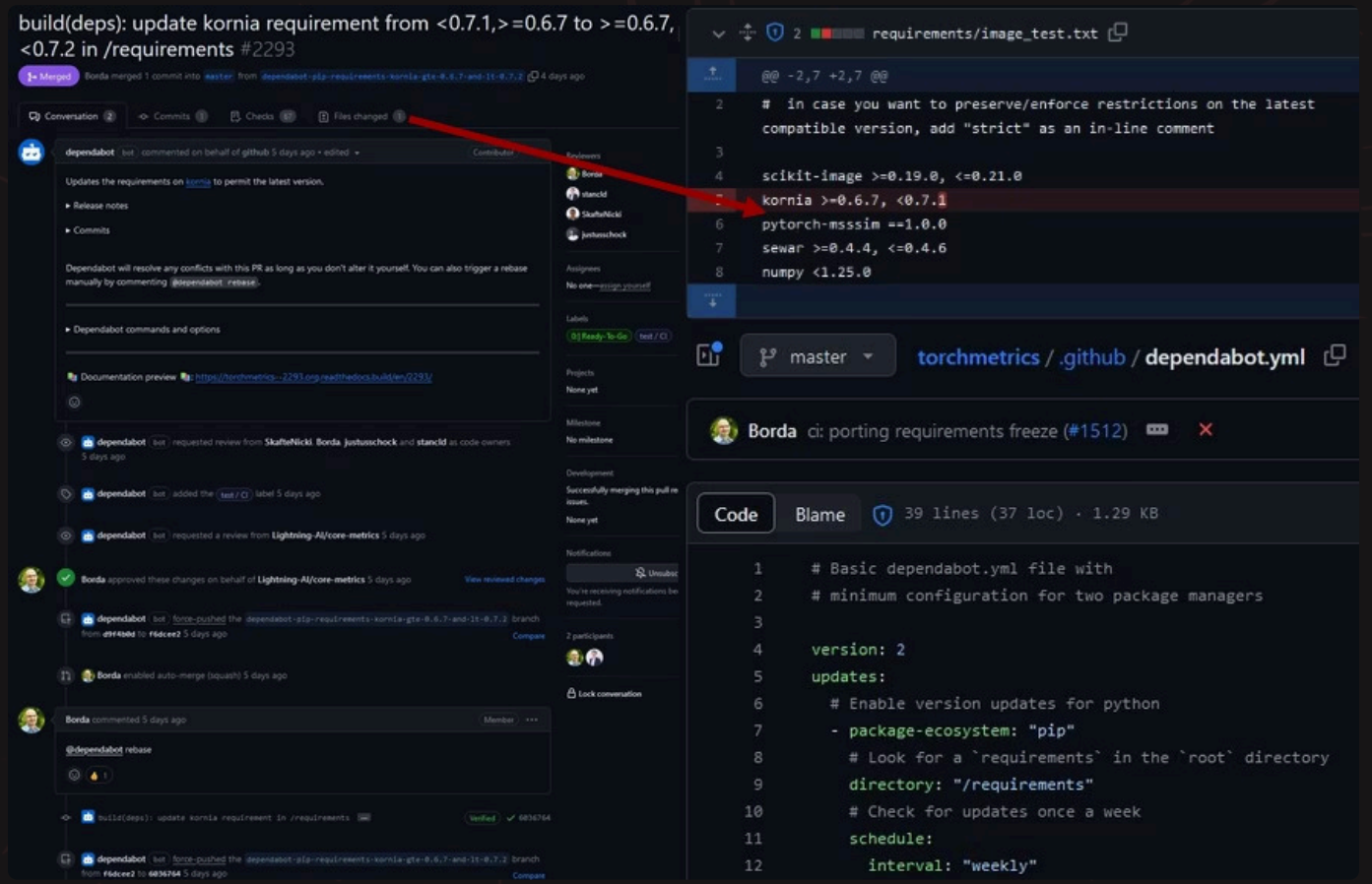


CI step 3: Automate tedious tasks with bots



CI step 3: Automate tedious tasks with bots

🤖Dependabot can help auto checking new releases of dependencies in your project



Summary of continues integration

Use version control	
Write (unit-)test for your code	pytest
Automate build + test	

The agents are here

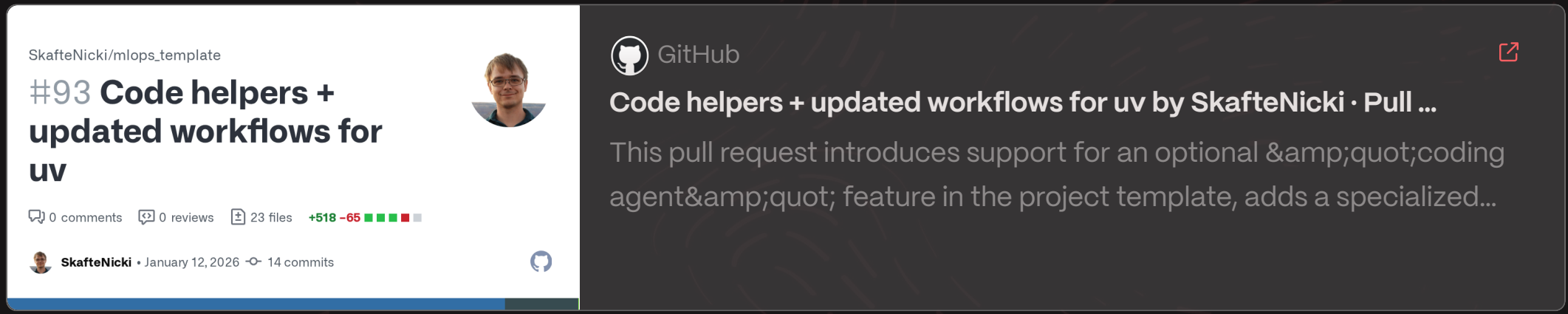
Agents can help you solve issues, review code and solve bugs...

GitHub

[bug] CLI exercise classifier dataset mismatch · Issue #480 · S...

In exercise of Command line arguments at s2_organisation_and_version_control/cli/, part 3 mentions: ... trains a...

A few corrections to last weeks template



- Smaller updates to agents
- Specific works for pip and uv workflows
- Correction of Docker uv files

Meme of the day