

Continuous Integration (CI)



What is it?

CD/CI - Continuous Delivery / Continuous Integration is a *DevOps* methodology that speeds up software delivery

- Automates analysis, tests each code change
 - Report errors
 - Streamline software releasing
- Improves development / collaboration



What do we need?

We need to combine:

- ~~Reproducible environment(s)~~
- Code analysis tools
 - ~~Formatters~~
 - ~~Linters~~
 - Type Checkers
- ~~Tests~~



The tooling Zoo



- Package managers ([micromamba](#), [uv](#))
- Version control systems ([git](#))
- IDEs - Integrated Development Environments ([VS Code](#))
- Code Editors ([vim](#), [emacs](#))
- Linters, formatters, type-checkers ([ruff](#))
- Compilers and/or Interpreters (e.g. [gcc](#), [python](#))
- Testing frameworks ([pytest](#))

Static analysis

- Syntax errors - beyond parsing:
 - All languages follow a syntax spec.
 - Easy for us to make mistakes/typos
- Some bugs - simple logical problems
- Conform to good practices

```
1 def helloworld():  
2     print('Hello',end='')  
3     print('world')  
4  
5 helloworld()
```

```
$ python3 helloworld.py  
File "/home/reis/dir/helloworld.py", line 3  
    print('world')  
                ^  
IndentationError: unindent does not match any  
outer indentation level
```



Static Analysis - Linters & formatters

The latest tool in the zoo is `ruff`.

It supports +900 linting `rules`

```
$ ruff check --output-format=concise helloworld.py
```

```
helloworld.py:3:1: SyntaxError: Unexpected indentation
helloworld.py:5:1: SyntaxError: Expected a statement
Found 2 errors.
```

```
$ ruff format --diff helloworld.py
```

```
1 --- helloworld.py
2 +++ helloworld.py
3 @@ -1,5 +1,6 @@
4  def helloworld():
5  -     print('Hello',end='')
6  -     print('world')
7  +     print("Hello", end="")
8  +     print("world")
9  +
10
11 helloworld()
```

`--diff` and `--check` only print outcome, remove them to format the source files



Static Analysis - Type checking

`mypy` is the *go-to* tool - leveraging **type hints**

```
mult.py
```

```
1 def mult_int(a: int, b: int) -> float: # note the optional syntax
2     c: int = a * b                    # used for type hinting
3     return c
4
5 print("Mult 3*6 is:")
6 print(mult_int(3, "6"))
```

```
$ mypy mult.py
```

```
mult.py:6: error: Argument 2 to "mult_int" has incompatible type "str"; expected "int" [arg-type]
Found 1 error in 1 file (checked 1 source file)
```

```
$ python3 mult.py
```

```
Mult 3*6 is:
666
```



Use ruff and mypy

- Create a **python** script (e.g. `hello_world.py`) that prints something
- Install `ruff` and `mypy` in your environment
- Use them to lint, format your script



We have the tools, now what?





Setting up a CI pipeline

- Git repos / services are **perfect place!**
 - Simply add the configuration files to the repo
 - Checks are done everytime you push code
 - CI verifies if commits meet expectation
- Provided on popular git hosting systems
 - [GitLab CI](#) as pipelines
 - [GitHub Actions](#)



What to expect?

- Brand-new environments should be created
 - Even with versioned/pinned dependencies
- Static analysis errors should be immediately spotted
 - Code style violations should be raised
 - Compiling code should be done (if compiled language)
 - e.g. `cpython` latest linting check: 
- Tests must run and assess functionality
 - e.g. `cpython` latest tests 

How!?

- Define stages/steps in order of dependency
- For each prepare the environment and tools to be used
- Introduce variants
 - e.g. using different `python` versions
 - e.g. run with a pinned version of your environment
 - e.g. run the latest (version) of your environment



Real case examples

- [GitHub Actions](#) workflow for CPython
- Course lectures on [Gitlab CI](#)



Gitlab CI

Configured with a file `YAML` file named `.gitlab-ci.yml`

- Gitlab CI syntax: docs.gitlab.com/ci/yaml
- DKRZ Gitlab documentation: docs.dkrz.de ⚠
 - A couple of extra configurations are needed to use DKRZ's `gitlab.dkrz.de`'s CI
 - E.g. jobs running in slurm/levante (templates documented) or not

Example of `.gitlab-ci.yml`

```
.gitlab-ci.yml
1 stages:
2   - static-analysis
3
4 default:
5   image: python # container image with basic python tools
6   tags:
7     - docker-any-image # note this is specific to DKRZ's gitlab
8
9 linter:
10  stage: static-analysis
11  script:
12    - pip install ruff
13    - ruff check .
14
15 type-check:
16  stage: static-analysis
17  script:
18    - pip install mypy
19    - mypy .
```

One can **also** edit the file in the Gitlab page under **Build > Pipeline editor**.



Example of `.gitlab-ci.yml`

```
.gitlab-ci.yml
1 stages:
2   - static-analysis
3
4 default:
5   image: python # container image with basic python tools
6   tags:
7     - docker-any-image # note this is specific to DKRZ's gitlab
8
9 linter:
10  stage: static-anaylis
11  script:
12    - pip install ruff
13    - ruff check .
14
15 type-check:
16  stage: static-analysis
17  script:
18    - pip install mypy
19    - mypy .
```

When using DKRZ's gitlab jobs require `tags` to be specified!



Runners

In order to *run* the CI, *code needs to run*¹ *somewhere*.

- Centrally provided (shared) runners
- Self-hosted
- Runners can be tagged, indicating functionality.
 - e.g. different hardware/os/capabilities
- Jobs can select runners using their **tags**.

```
.gitlab-ci.yml
```

```
1 default:  
2   tags:  
3     - gpu
```

1. Gitlab Runners: <https://docs.gitlab.com/ci/runners>



(Docker) Images & Containers

- Images are compiled “*environments*” containing an entire operating system.
 - e.g. `python`: an operating system with Python pre-installed

The `image` keyword specifies an image for the executor to use

```
.gitlab-ci.yml
1 default:
2   image: python # by default resolves to https://hub.docker.com/_/python
```

(Docker) Images & Containers

- Containers are active environments based on an image.
 - i.e. a running virtual machine, created for a single job
- To use a container in Gitlab CI, you need a runner with a “Docker Executor”.
 - e.g. using `docker-any-image` tag on DKRZ gitlab.




Hands-on Session

Create a **simple** Gitlab CI pipeline job on your repository:

- It should use python and print "Hello world!"

Important

 You need to enable CI/CD in Settings->General->"Visibility, project features, permissions" and enable instance runners under "Settings->CI/CD->Runners".

Tip

Since you are using DKRZ's Gitlab refer to the documentation:

- General: <https://docs.dkrz.de/doc/software%26services/gitlab-git-repository-manager.html>
- Runners: <https://docs.dkrz.de/doc/software%26services/gitlab-runner.html#dkrz-runners>

Hands-on Session

```
.gitlab-ci.yml
```

```
1  ---
2  stages:
3    - greet
4
5  say-hi:
6    stage: greet
7    image: python
8    script:
9      - python -c 'print("Hello world!")'
10 tags:
11   - docker-any-image
```

Hands-on Session

- Create a python script that is checked by ruff
 - It should lint and format of your code
- Bonus: use the type-checker mypy as well



Hands-on Session

```
.gitlab-ci.yml
1  ---
2  stages:
3    - static
4
5  ruff:
6    stage: static
7    image: python
8    script:
9      - pip install ruff
10     - ruff check .
11     - ruff format --check .
12  tags:
13    - docker-any-image
14
15  mypy:
16    stage: static
17    image: python
18    script:
19      - pip install mypy
20      - mypy .
21  tags:
22    - docker-any-image
```

Ruff has a [documentation](#) section dedicated for Gitlab CI.



More on Gitlab CI



Artifacts

- Jobs have an ephemeral nature
 - Files they create will be gone once completed
- `artifacts`¹ allows us to specify which files to keep
 - Used to pass data between jobs
 - Save results (e.g. code report, documentation)
 - Expected to disappear (eventually/explicitly)

1. https://docs.gitlab.com/ci/jobs/job_artifacts/



Example with artifacts

```
.gitlab-ci.yml
```

```
1 stages:
2   - create
3   - use
4
5 defaults:
6   tags:
7     - condaforge
8     - dkrz
9
10 job1:
11   stage: create
12   script:
13     - mkdir output
14     - echo "Creating a result!" | tee -a output/result.txt
15
16 job2:
17   stage: use
18   script:
19     - cat output/result.txt
```

```
$ cat output/result.txt
cat: output/result.txt: No such file or directory
```

The pipeline fails, `job2` starts from the same initial point as `job1` and there is no persistency!



Example with artifacts

```
.gitlab-ci.yml
1  stages:
2    - create
3    - use
4
5  defaults:
6    tags:
7      - condaforge
8      - dkrz
9
10 job1:
11   stage: create
12   script:
13     - mkdir output
14     - echo "Creating a result!" | tee -a output/result.txt
15   artifacts:
16     paths:
17       - output/result.txt
18     expire_in: 1 hour
19
20 job2:
21   stage: use
22   script:
23     - cat output/result.txt
```



Matrix jobs

- Using `parallel:matrix`¹ one can run a job in parallel.
 - For each combination of values a job will be spawned.

```
.gitlab-ci.yml
1  stages:
2    - run
3
4  default:
5    tags:
6      - docker-any-image
7
8  job_python:
9    stage: run
10   image: python
11   script:
12     - python --version
```

1. <https://docs.gitlab.com/ci/yaml/#parallelmatrix>



Matrix jobs

- Using `parallel:matrix`¹ one can run a job in parallel.
 - For each combination of values a job will be spawned.

```
.gitlab-ci.yml
1 stages:
2   - run
3
4 default:
5   tags:
6     - docker-any-image
7
8 job_python:
9   stage: run
10  parallel:
11    matrix:
12      - PYTHON_VERSION: ["3.9", "3.10", "3.11", "3.12", "3.13", "3.14", "3.15"]
13  image: python:${PYTHON_VERSION}
14  script:
15    - python --version
```

1. <https://docs.gitlab.com/ci/yaml/#parallelmatrix>



Take home messages

- Tools exist to make everyone's life easier
- CI takes tools to the *next level*
- Makes problem detection and collaboration easier!



Shotgun buffet



Executors

In Gitlab different executors¹ offer different isolation settings.

- With `SSH` and `Shell` executors, a job that runs `rm -rf /` could destroy the state of the runner (persistence).
- The same using `Docker` would only destroy the **newly created isolated** environment (not even the image)

There are [official registries](#) from where to download images.

Your repository could also have an image registry, allowing you to create and use your own images

1. <https://docs.gitlab.com/runner/executors/#selecting-the-executor>



Pre-commit hook &

Automate the locally checking the code with `git hooks` ¹.

1. Inspect `.git/hooks/pre-commit.sample`. What does it do?
2. Install `pre-commit`, configure it to use `ruff-pre-commit`
3. Modify a python script (e.g. `hello_world.py`)
4. Try to commit that modification
5. Did the commit went through? What if you try to commit a syntax error?

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



Solution pre-commit

Using the pre-commit tool:

```
.pre-commit-config.yaml
```

```
1  ---
2  repos:
3    - repo: https://github.com/pre-commit/pre-commit-hooks
4      rev: v2.3.0
5      hooks:
6        - id: check-yaml
7        - id: end-of-file-fixer
8        - id: trailing-whitespace
9    - repo: https://github.com/astral-sh/ruff-pre-commit
10      rev: v0.11.5
11      hooks:
12        - id: ruff
13        - id: ruff-format
```

```
1  pre-commit install
2  pre-commit run --all-files
```

