Introduction to good DevOps practice

A Noob-Friendly Guide to DevOps for Scientific Software

Matheus do Carmo Teodoro @EuXFEL
matheus.teodoro@xfel.eu

Outlook

- Introduction to DevOps for Scientific Software
- Git & Collaborative Development
- Introduction to CI/CD (Continuous Integration & Deployment)
- Versioning & Deployment
- Final Takeaways and Q&A

Introduction to DevOps for Scientific Software

Git & Collaborative Development

Introduction to CI/CD (Continuous Integration & Deployment)

Versioning & Deployment

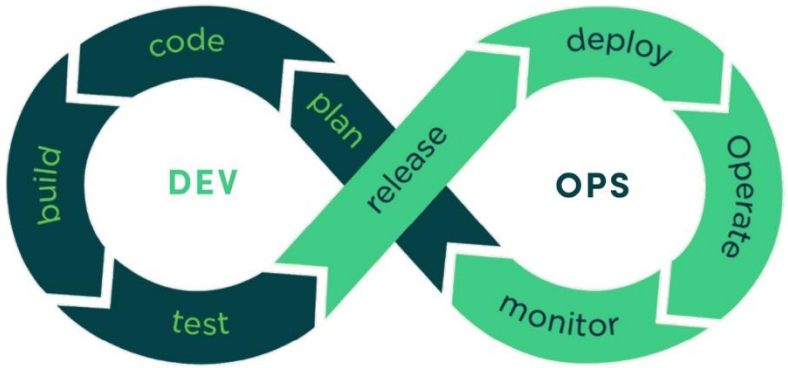
Final Takeaways and Q&A

Introduction to DevOps for Scientific Software

What's DevOps and why does it matter?

Dev + Ops

"DevOps combines development (Dev) and operations (Ops) to increase the efficiency, speed, and security of software development and delivery compared to traditional processes"



Without DevOps

Bugs appear too late and it takes long to deploy the fixed version

Complicated and hard to track development

Hard to maintain software

With DevOps

Automatic testing

Automatic deployment

Reliable and reproducible software

Although effort and time is needed to setup a DevOps good pipeline, it saves one greater effort and time waste later on

Introduction to DevOps for Scientific Software

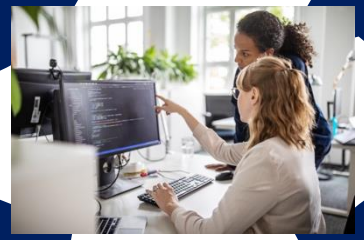
What's DevOps and why does it matter *FOR SCIENCE*?

Typical starting place of a scientific software developer

- I code alone
- Only I use my code
- I test my code by running it
- I'm only interested in the results

The new context that developer might face

- Many people will develop the software
- Many people will use the software
- Bugs might have catastrophic consquences
- Performance, security and scalability matter now



How to adapt

In this new scenario DevOps is needed since it provides:

- Collaborative development
- Automatic testing and deployment
- Catching problems early on
- Build reusable and reliable software

Introduction to DevOps for Scientific Software

Core principles

Collaborative development

Multiple contributors and large projects will require good planning and organisation. GitHub/GitLab will provide a organised way of development, version control, PRs and code reviews.

Versioning and Reproducibility

Scientific software often requires results to be reproducible. Versioning the software will ensure that the software releases are trackable.

Automation and Continuous integration

Time and performance is of the essence. A good CI pipeline will remove the need of manual testing while CD will deliver the application updates fast to users. GitHub/GitLab also provides means to build CI/CD pipelines.

Deployment

Delivering the software to the end user must be fast and (semi)automatic.

Introduction to DevOps for Scientific Software

Git & Collaborative Development

Introduction to CI/CD (Continuous Integration & Deployment)

Versioning & Deployment

Final Takeaways and Q&A

Git & Collaborative Development

Large projects with multiple contributors require organisation, planning, and control.

Collaborative development

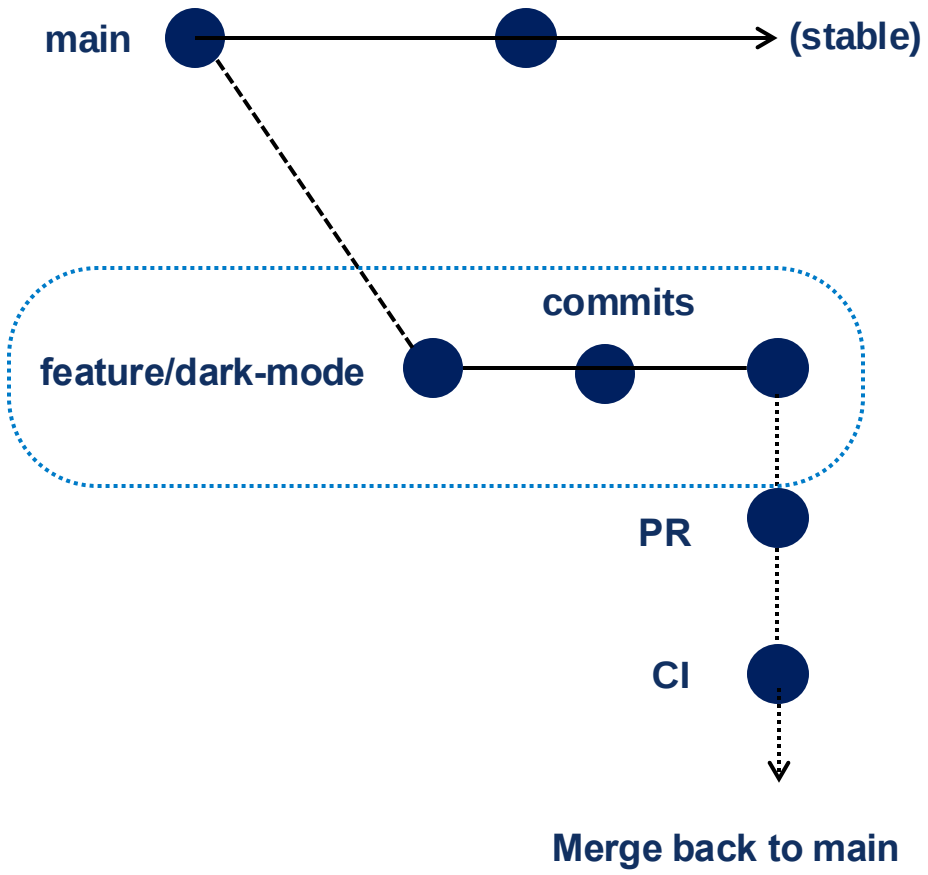
DevOps solution: GitHub/GitLab will provide tools for:

- ❑ **Commit, commit, commit!** Yes, time travel is a thing with git.
- ❑ **Push Requests (PR) and Code reviews!** Now you can get formally roosted by your colleagues and increase the quality of the application.
- ❑ **Branching and merging strategies!** Feeling uncertain? Worry not! You will use a branch you can break at first.

Example: ”Users want a dark mode in the DAMNIT application. I branch the repo, implement the feature, make a PR and, after review I can merge it. The users are now happy”

Git & Collaborative Development

Best practices when developing with git, it's all about *good* communication



❑ Branching and merging strategies

- ❑ **Use feature branches** oringin/main must be kept clean
- ❑ **Follow name convention** stuff526 is a bad name
- ❑ **Consider squashing before the final merge**

❑ Versioning control

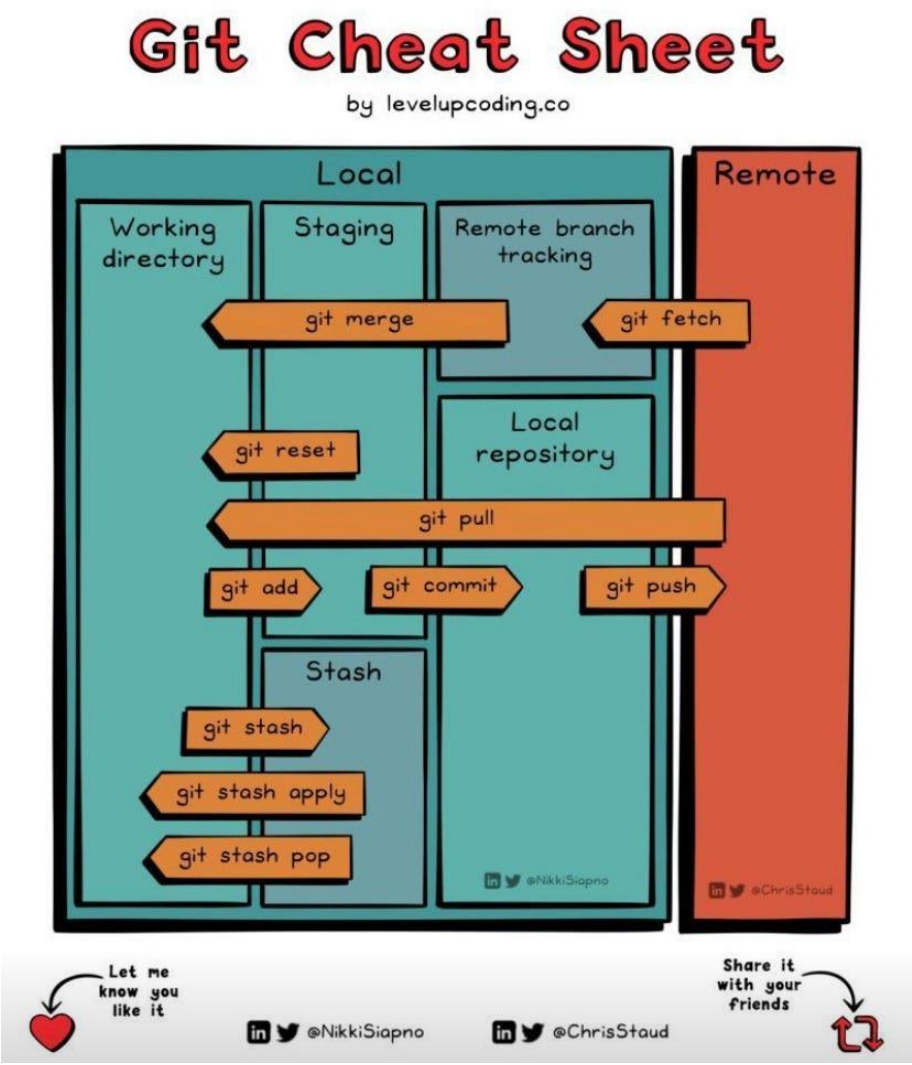
- ❑ **Commit often but keep it meaningful**
- ❑ **Follow conventions and write clear commits:** ”fix: dark mode contrast issue #42” is better than “Oops, my bad.”

❑ PRs and Code reviews!

- ❑ **Abuse of markdown, figures and gifs**
- ❑ **Tell your reviewer how to test your feature**
- ❑ **Point out possible problems**

Git & Collaborative Development

Best practices when developing with git, it's all about *good* communication



❑ Branching and merging strategies

- ❑ **Use feature branches** oringin/main must be kept clean
- ❑ **Follow name convention** stuff526 is a bad name
- ❑ **Consider squashing before the final merge**

❑ Versioning control

- ❑ **Commit often but keep it meaningful**
- ❑ **Follow conventions and write clear commits:** "fix: dark mode contrast issue #42" is better than "Oops, my bad."

❑ PRs and Code reviews!

- ❑ **Abuse of markdown, figures and gifs**
- ❑ **Tell your reviewer how to test your feature**
- ❑ **Point out possible problems**

Introduction to DevOps for Scientific Software

Git & Collaborative Development

Introduction to CI/CD (Continuous Integration & Deployment)

Versioning & Deployment

Final Takeaways and Q&A

Introduction to CI/CD

Manual testing is slow and error-prone. Automation ensures faster and reliable software delivery

Introduction to CI/CD

DevOps will save you from deployment hell:

- **Continuous Integration!** Writing tests is a necessary pain. Now every push will automatically test your code, so you can go home and sleep peacefully after work
- **Continuous Deployment/Delivery:** Passed the test? Approved PR? Is it a new release? Congrats, now deployment will be (semi)automatically done.

Example: ”Dark mode feature is developed and approved! Now we need to assure that the feature won’t break the application and , likewise, that further changes on the application won’t break the feature. The new version of the application needs to be ready for release, users are eager to use it!”

Introduction to CI/CD

Manual testing is slow and error-prone. Automation ensures faster and reliable software delivery

Introduction to CI/CD

How does it work?

- **To set up your CI/CD pipeline all you need** (apart from your code, your tests, your Dockerfile…) **is a YAML (Yet Another Markup Language) file that looks like this:**

```
main ?3 ❯ touch .github/workflows/ci-cd-pipeline.yml
```

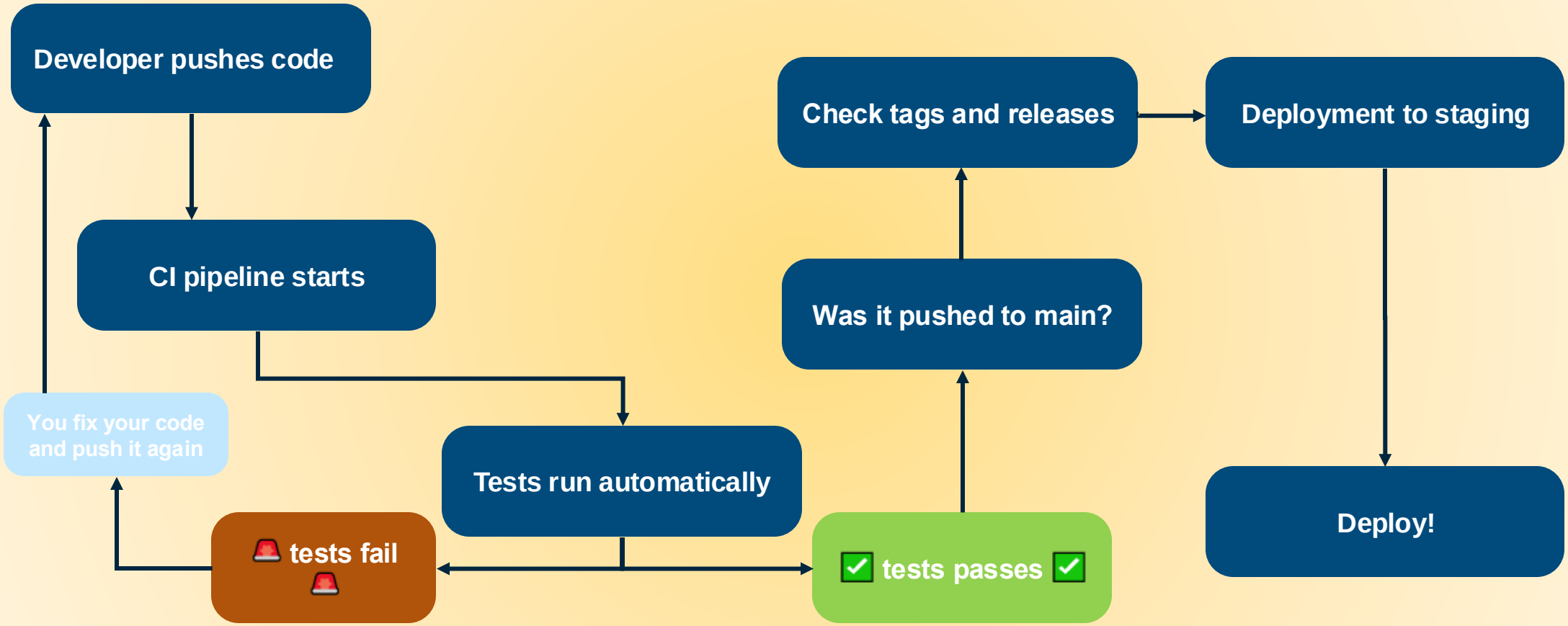
vim test.yml

```yaml
name: My CI/CD Pipeline  # Name of the workflow

on: [push, pull_request]  # Triggers: e.g. runs when code is pushed or PR is created

jobs:
  build-and-test:
    runs-on: ubuntu-latest  # The environment where the job runs
    steps:
      - name: Checkout code
        uses: actions/checkout@v3  # Fetches repo code

      - name: Set up Python
        uses: actions/setup-python@v3   # Sets up python env
        with:
          python-version: '3.9'

      - name: Install dependencies   # Install dependencies
        run: pip install -r requirements.txt

      - name: Run tests
        run: pytest # Run tests
```

Introduction to CI/CD

Manual testing is slow and error-prone. Automation ensures faster, more reliable software delivery

Workflow



Introduction to CI/CD

Manual testing is slow and error-prone. Automation ensures faster, more reliable software delivery

Best practices when building your tests

- ❑ **Write good unit tests**
  - ❑ Keep them fast and isolated
  - ❑ Don’t try to test every single thing
  - ❑ Use mocking if your application needs a DB or APIs
- ❑ **Do you need integration tests?**
  - ❑ Keep in mind that these are slow, so run they after the unit tests
  - ❑ Docker can be useful to make temporary environments (e.g. your code uses a Kafka broker)

Introduction to CI/CD

Minimal example of a YAML setup for GitHub

Continuous integration

vim tests.yml

```yaml
name: Python CI

on: [push]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'

      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          pip install -r requirements.txt

      - name: Run tests
        run: |
          python -m unittest discover
```

- Workflow will be triggered on “push”
- Defines the env where the jobs will run (here the latest ubuntu)
- GitHub action: clone the current repository version into the VM
- GitHub action: Install python environment on the runner
- Install dependencies from requirements.txt
- Run your unit tests

Introduction to DevOps for Scientific Software

Git & Collaborative Development

Introduction to CI/CD (Continuous Integration & Deployment)

Versioning & Deployment

Final Takeaways and Q&A

Versioning and deployment

Publishing your package

Example: DAMNIT

vim CD.yml

```yaml
name: Publish

on:
  push:
    branches: [master]
    tags: ["*"]
  pull_request:

env:
  SYSTEM_PACKAGES: gcc g++ xvfb qtbase5-dev

jobs:
  tests:

  "..."

  publish:
    runs-on: ubuntu-latest
    if: ${{ startsWith(github.ref, 'refs/tags/') }}
    needs: tests
    permissions:
      id-token: write

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up Python 3.10
        uses: actions/setup-python@v5
        with:
          python-version: "3.10"

      - name: Build packages
        run: |
          python3 -m pip install build
          python3 -m build

      - name: Publish package distributions to PyPI
        uses: pypa/gh-action-pypi-publish@release/v1
```

Workflow will be triggered on push into master and if the commit has a tag. Tags are set manually to indicate that this version should be published

```
git tag -a v1.0 -m "First stable release"
git push origin v1.0
```

Creates source (.tar.gz) and wheel (.whl) distributions

Uses PyPI trusted publishing (no need to store API tokens in secrets).

Package is ready to use

```
pip install damnit
```

Versioning and deployment

Pushing a docker image

Example: EUXFEL environments

vim test.yml

```yaml
name: Docker Image CI

on:
  workflow_dispatch:
  push:
    branches: ['main']
    paths:
      - 'custom-recipes/Dockerfile'

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push-image:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Log in to the Container registry
        uses: docker/login-action@65b78e6e13532edd9afa3aa52ac7964289d1a9c1
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}

      - name: Extract metadata (tags, labels) for Docker
        id: meta
        uses: docker/metadata-action@9ec57ed1fcdbf14dcef7dfbe97b2010124a938b7
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}

      - name: Build and push Docker image
        uses: docker/build-push-action@f2a1d5e99d037542a71f64918e516c093c6f3fc4
        with:
          context: ./custom-recipes
          push: true
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
```

- Workflow will be triggered on changes are pushed into ‘custom-recipes/Dockerfile’, at the main branch
- REGISTRY is set to GitHub Container Registry (GHCR)
- Checkout the repository
- Logs in to GitHub Container Registry (GHCR)
- Builds docker image, includes extracted metadata and push it to GHCR

Versioning and deployment

General remarks

Optimize your workflow!

- **Use dependabot:** that way you can keep your dependencies up to date

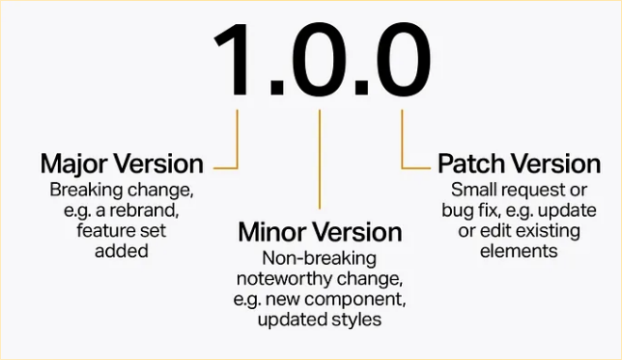
vim dependabot.yml

```yaml
version: 2
updates:
  - package-ecosystem: "pip"
    directory: "/.github/dependabot/"
    schedule:
      interval: "daily"
~
~
~
```

vim constrains.txt

```
Extra-data==1.19.0
ipython==8.32.0
kafka-python==2.0.3
matplotlib==3.10.0
mplcursors==0.6
mpl-pan-zoom==1.0.0
numpy==2.2.3
openpyxl==3.1.5
pandas==2.2.3
pyflakes==3.2.0
```

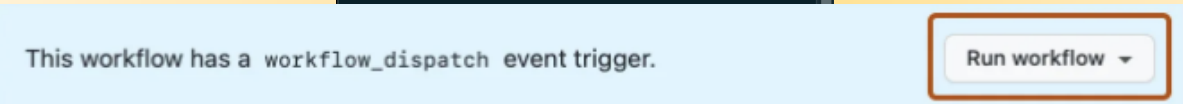
- **Treat your versioning with care <3**



- **Manual Job Triggers:** For long duration jobs that might be optional

vim CD.yml

```yaml
name: Manual workflow

on: workflow_dispatch
```



- **Use caching:** This way you can have faster builds

vim CD.yml

```yaml
  - uses: actions/cache@v4
    with:
      path: ~/.cache/pip
      key: ${{ runner.os }}-${{ matrix.python-version }}
~
```

Introduction to DevOps for Scientific Software

Git & Collaborative Development

Introduction to CI/CD (Continuous Integration & Deployment)

Versioning & Deployment

Final Takeaways and Q&A

👀 Final remarks

What we've learned today? What's next?

Why DevOps?

Setting a DevOps good pipeline will save you greater effort and time waste later on. Also will make your product more scalable and reproduceble.

Collaborate with git

Give your colleagues a robust way to collaborate if you writing good commits and PRs. Also give yourself the ability of travel to past commits and the freedom of branching for a smooth dev setup!

How to CI

Write good tests and automatize them with a CI pipeline, this way your software will be less prone to error.

How to CD

Weather if you are publishing a package in PyPI or pushing a docker image, automatize the process of deployment or delivery so your work can reach the users faster.

What's next?

We've talked a lot about how to automatize deployment, but how does deployment work? Later in this workshop you will learn about:

- Package and framework deployment at European XFEL
- Continuous integration and deployment at DESY FS-SC
- Introduction to containers
- Deployment and containerisation of infrastructure components at HZB
- DECTRIS container services
- CI-, containerisation and virtualisation workflows
- Kubernetes applied to an image processing service

👀 Acknowledgments and Q&A

Powered by teamwork – Thanks, Everyone!

- Cammille Carinan @EuXFEL
- Robert Rosca @EuXFEL
- Fabio Dall'Antonia @EuXFEL
- Luca Gelisio @EuXFEL
- Lisa Amelung @ DESY

Questions? 🧐