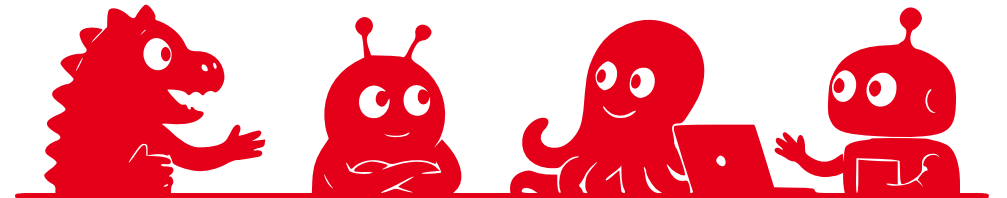




## Software development: from code to software

The challenges of software development in the context of large-scale experiments (collaborations) and long project timescales.



# Approach and use case



**Why does a 50-line script that works today become a software-engineering problem when 20 people need to use it, modify it, reproduce its results, and maintain it for 10 years?**

# Approach and use case

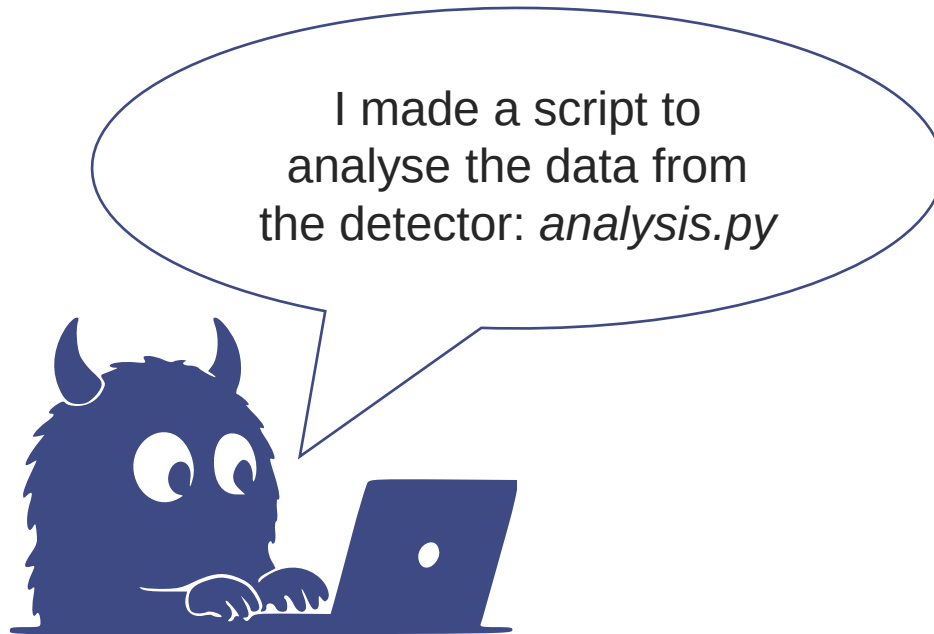


Why does a 50-line script that works today become a software-engineering problem when **20 people** need to use it, modify it, reproduce its results, and **maintain** it for 10 years?

Not all IRFU  
members

With IRFU infra/tools only?

# Introduction: “It's just some code”



# Introduction: “It's just some code”

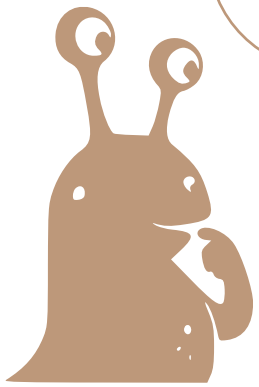
How to install it?

I want to add an input parameter.

Does it works on my machine?

Oh, I don't have *libirfu* installed...

I don't get the same result as you do with the same input data...

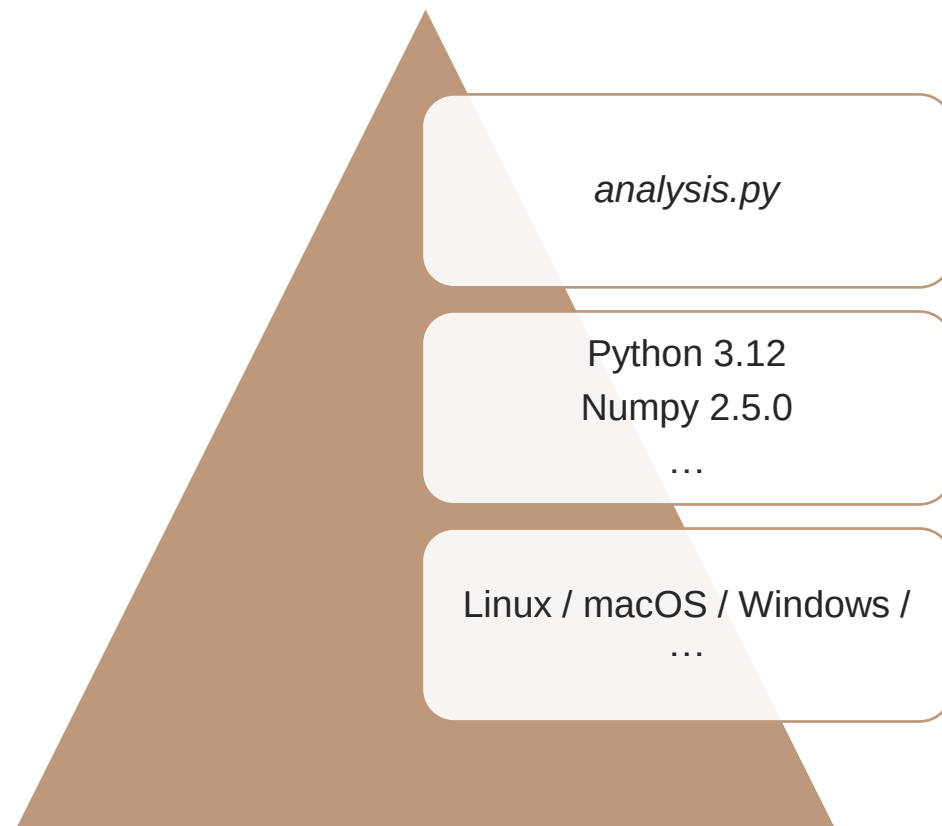
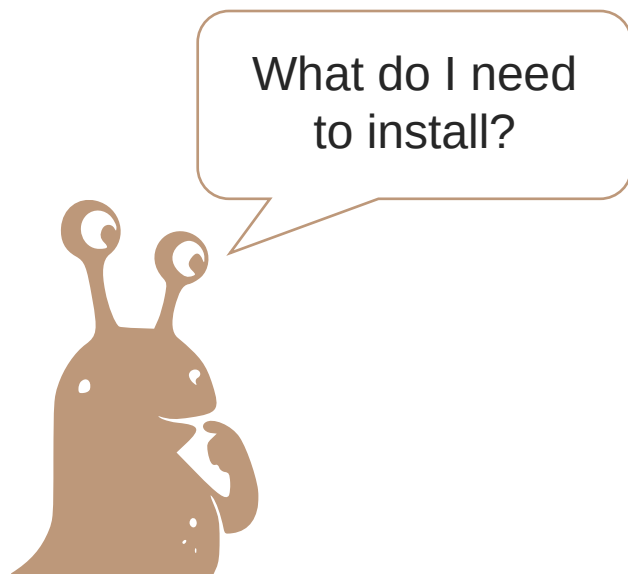


It works on my machine!



We're no longer talking about a script but software development!

# Sharing code vs. sharing software



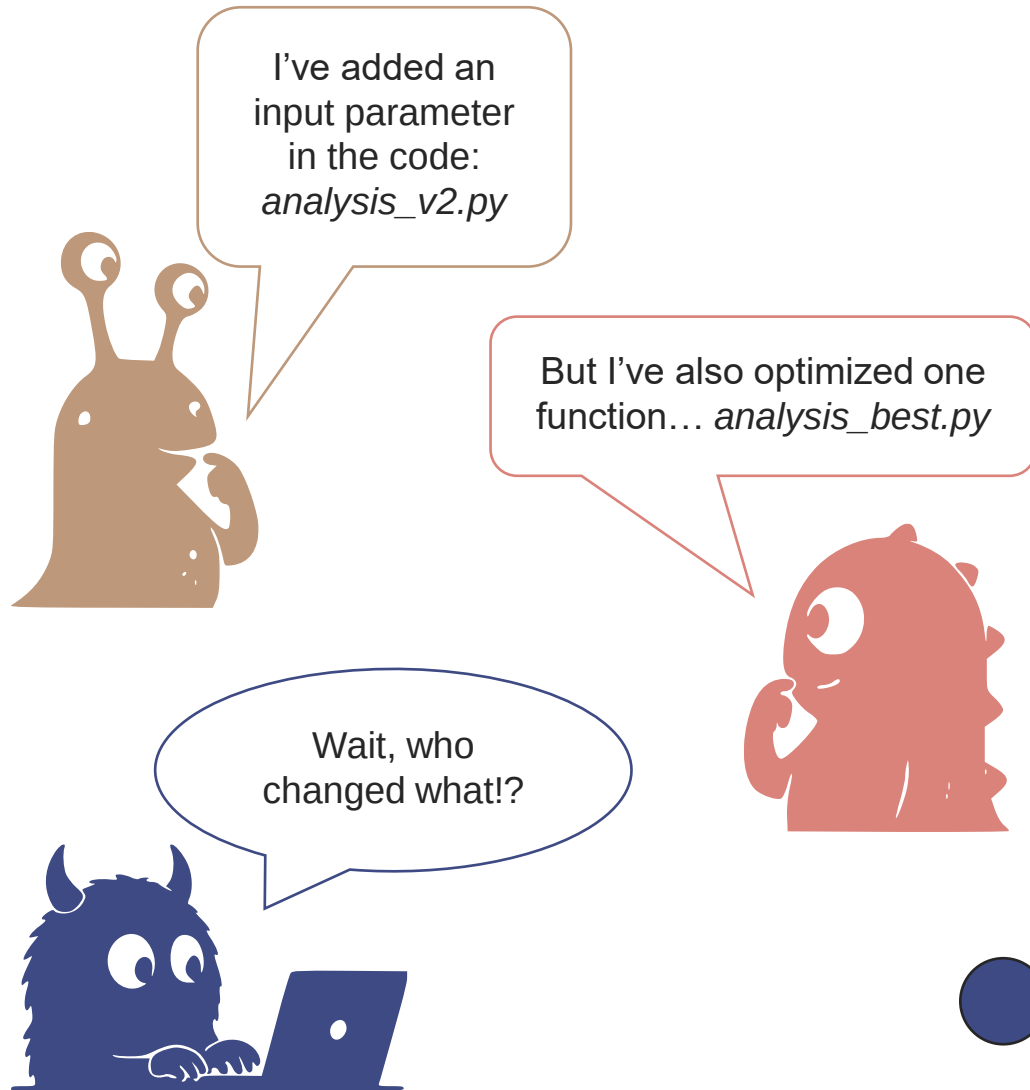
What needs to be shared isn't just *analysis.py* but also the conditions under which *analysis.py* works: reproducibility of the software environment.

## Environment sharing:

- Python virtual environment
- Conda environment
- Virtual machine (Euclid: virtualbox)
- Docker container (Svom)
- Guix
- Spack



# Collaboration and version control



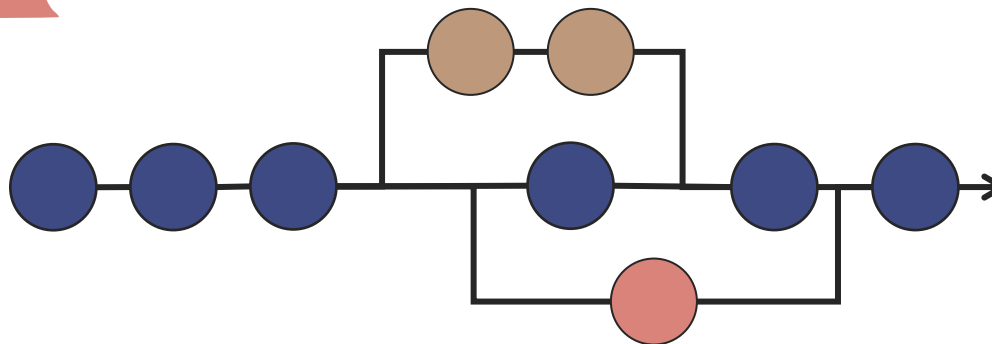
One person writing code → several people developing software.

Version control is not primarily about saving versions. It is about coordinating people:

- What changed?
- Who changed it?
- Why it changed?
- How to handle conflicts?

Version control tools:

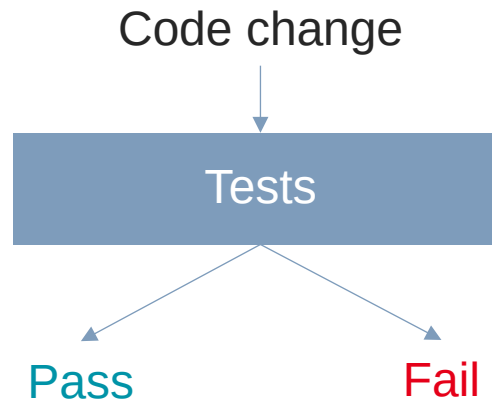
- git (github, gitlab)
- subversion
- ...



# Validation: how do we know we didn't break it



After my changes, and the ones from **Brown**, it runs and gives 42, before it was 24.



## Tests:

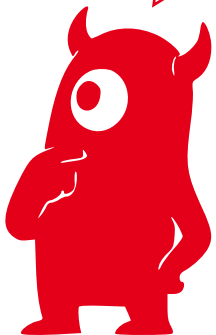
- Unit tests (pytest, boost.test, googletest)
- Tests campaigns (e.g. in Euclid, run the code with the same inputs and check that astrophysical statistics are preserved)
- Comparison with independent implementation
- Test report generation



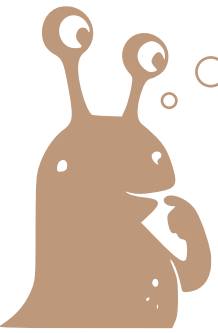
## Checks

- Does it still run?
- Do functions behave like expected?
- Do components (various codes, libraries) still work together?
- Are scientific invariants preserved (when no simple expected answer)?
- Do known science reference cases still give the expected result?
- Execution test (runtime, memory, ...)

Is this really better?



# Do we really have to check this every time?



Modify, build, test, publish,  
modify, build, test, publish,  
modify...

- Actions are triggered automatically (code changes, scheduled run, ...)
- Automation enforces consistency (eliminates human factor)
- Faster feedback → fewer mistakes
- Brings trust in changes/code

Continuous integration and delivery

- GitLab CI
- Github actions
- Jenkins
- pre-commits hooks
- Sonarqube



triggering  
event



build  
code



check  
quality

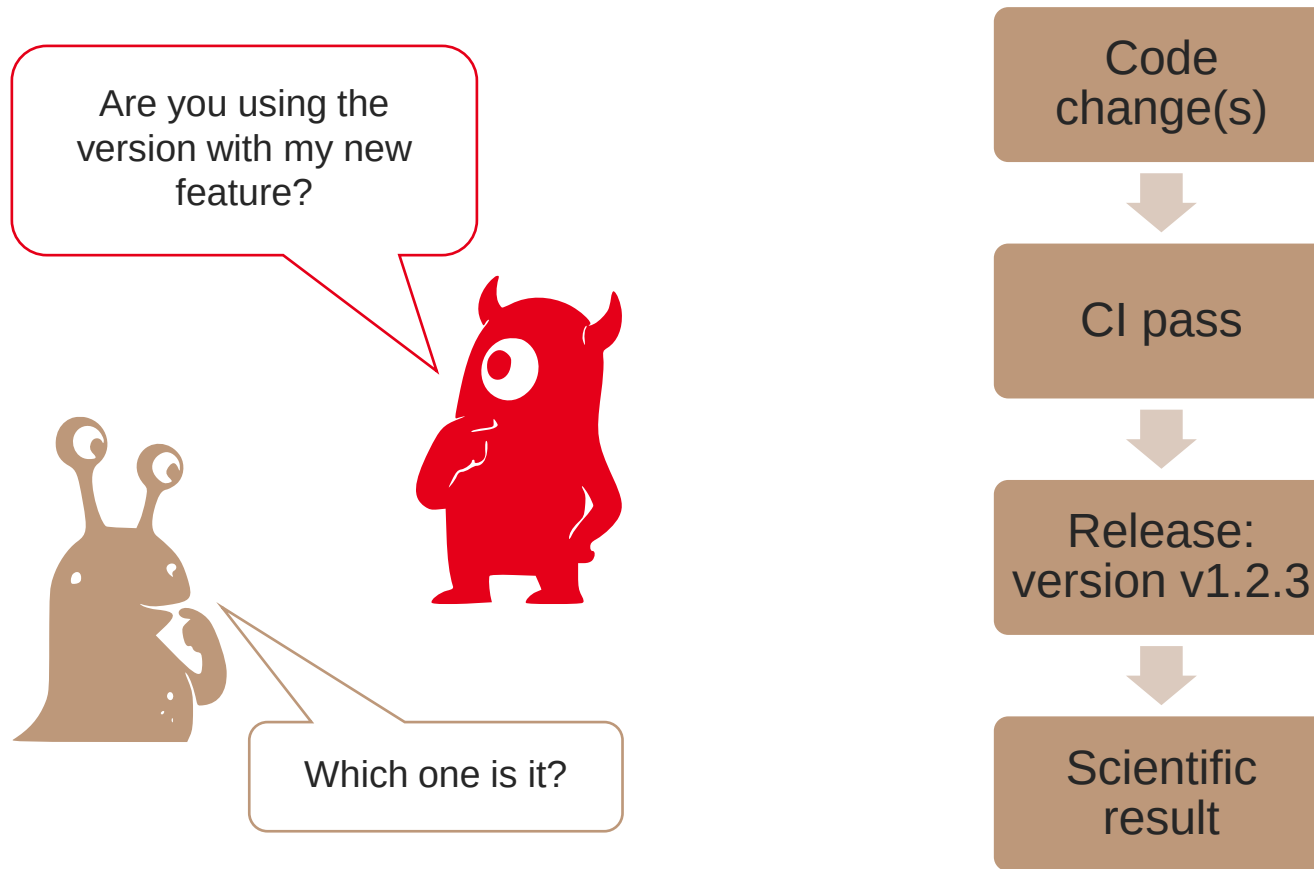


run tests

# Which version produced this result?

CI tells us that the software is still working.

But scientific work also needs us to know *which* software produced a result.



## Versioning

- Git tags
- Changelogs (towncrier)
- Semantic versioning
- Conventional commits



# The software is getting bigger...

Growing software can be difficult to maintain: the spaghetti problem → the code structure matters.

Processing1.py

- Read & calibrate
- Process

Processing2.py

- Read & calibrate
- Process

Processing3.py

- Read & calibrate
- Pre-process
- Process



How to add a processing?  
How to change the calibration  
step only for processing1?

- Keep different concepts separated
- Data structure choices have large impact on the code
- Don't be afraid to break everything, that can save time later (refactoring)
- All-time job: fight entropy as long as the software grows

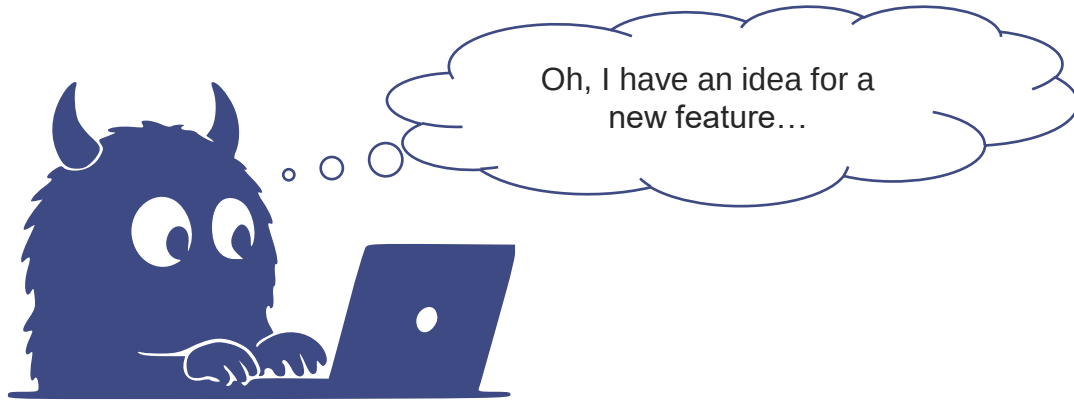
Structuring

- Design patterns
- Class diagrams
- Templates
- Polymorphism
- Inheritance



# Who decides what goes into the software? (and how)

With many contributors, governance and decision-making matters.



- Who can contribute?
- Who reviews?
- Who approves changes?
- Who can merge?
- Who decides releases?
- What requires scientific validation?
- One developer can have habits. A team needs conventions.
- Common practices/shared rules
- A shared codebase needs a shared decision process.
- An important role: the product owner.

## Code formatting

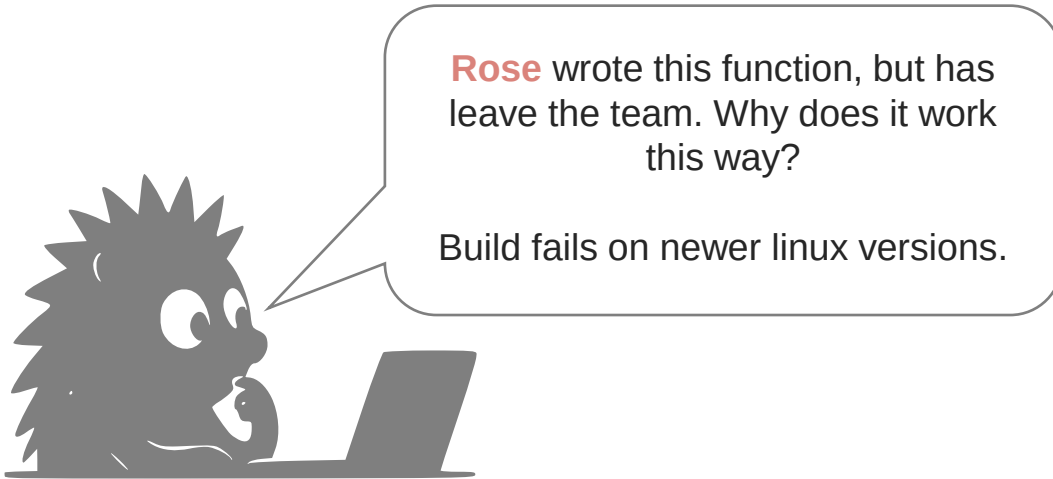
- Pylint, black, ruff
- Clang-format

## Workflows

- Git workflow



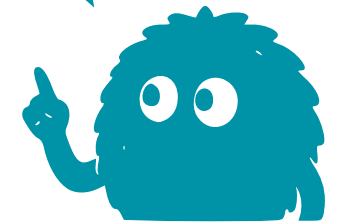
# What happens when the original developer leaves?



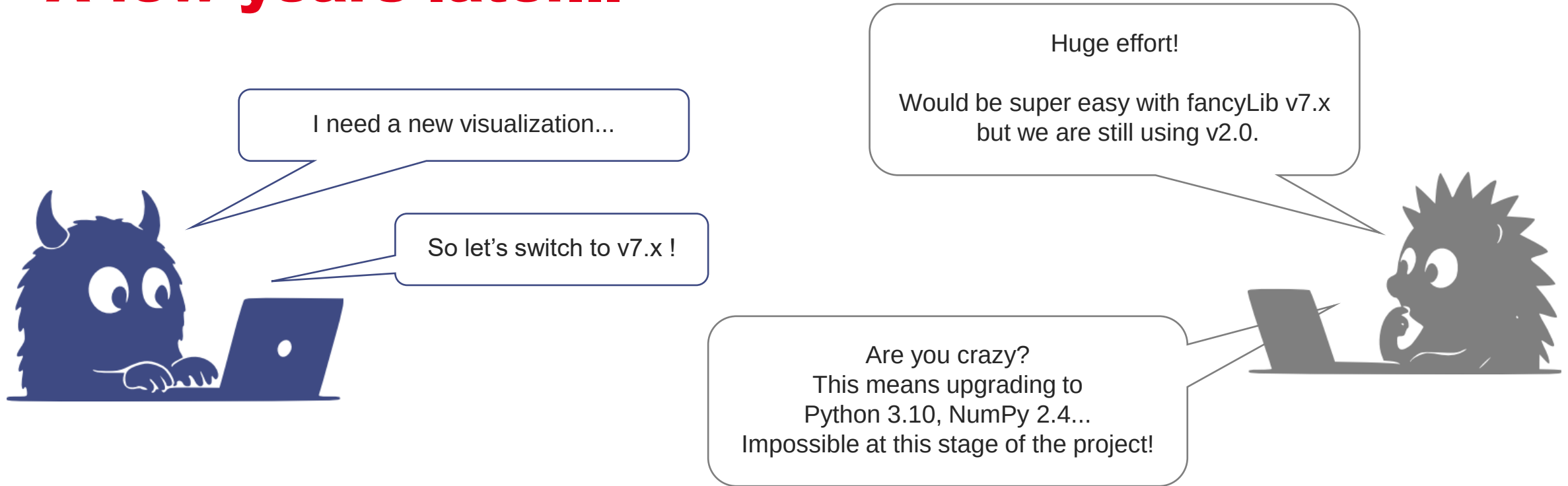
The software may need to survive longer than the people who wrote it.

- Documentation
- Comments in the code (one of the quality metric)
- Tests
- Clear architecture
- Reproducible build/environment
- Release history
- Maintainers / ownership
- Knowledge transfer

- Documentation generation (sphinx, doxygen)
- GitLab/Github pages
- Project documentation
- Commit-issues link



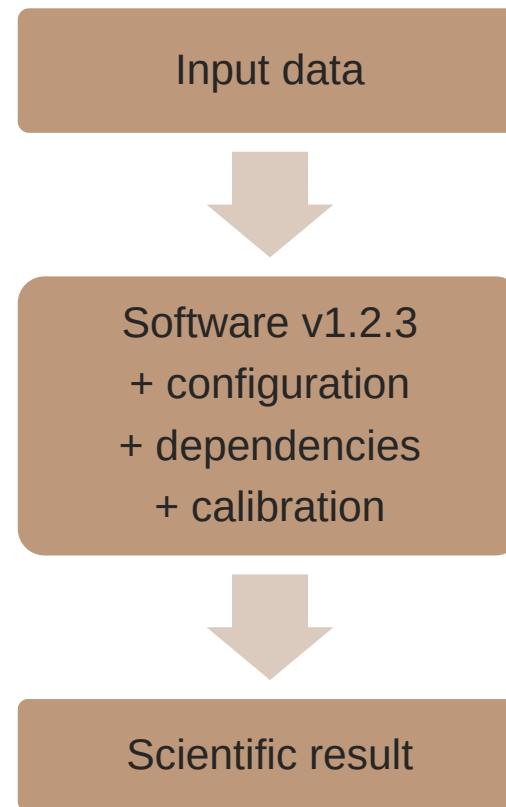
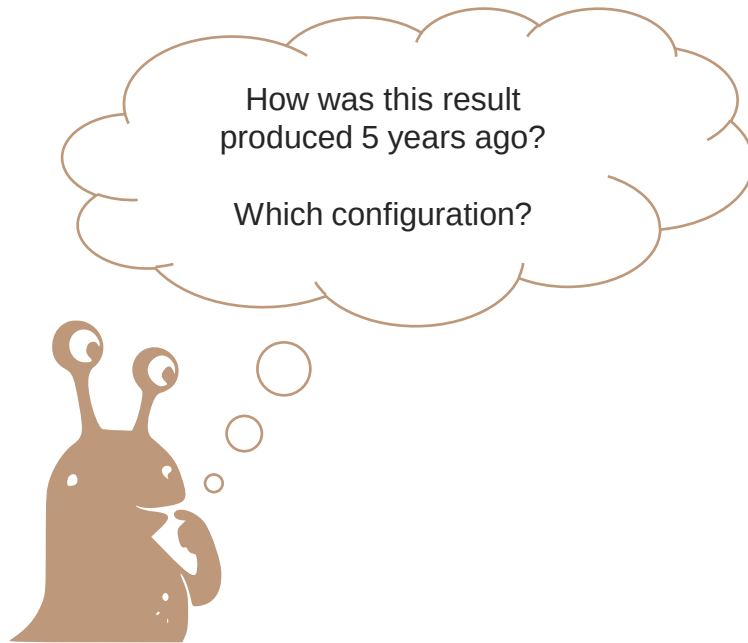
# A few years later...



The software ecosystem is constantly evolving. Make regular updates to your system's dependencies, rather than allowing them to become obsolete and increasingly difficult to upgrade. Neither too quickly nor too slowly.

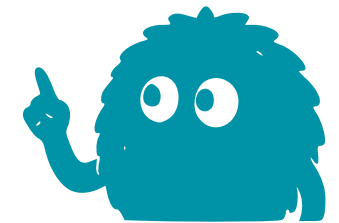
# Can we trace the science back to the software?

A scientific result is not just a number. It has a history.

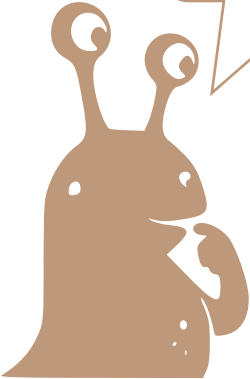


Result history:

- Versioned configuration (Conditions REST)
- Metadata in output products (software version, configuration version)
- Codemeta



# How much engineering do we actually need?

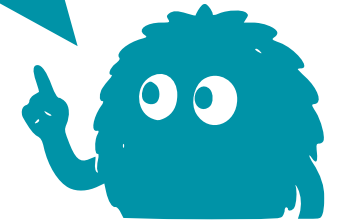


All this job to do for a small script?

Does every piece of scientific code need CI, 100% test coverage, documentation, releases...?

- The more people, time, and scientific impact involved, the more discipline you need.
- Engineering effort should scale with the consequences of failure.
- Rules should apply as soon as possible.

- Euclid Consortium has defined code Maturity Level



50-line experiment

- 1 readme

Research tool

- Git project
- Documentation
- 30% line coverage

Shared analysis

- Automation
- Code review
- 0 blocking violation rules
- 50% line coverage

Community software

- Releases
- 0 critical violation rules
- 70% line coverage

# Why does software engineering matter to science?

- **Trust software**

We know changes haven't silently broken it.

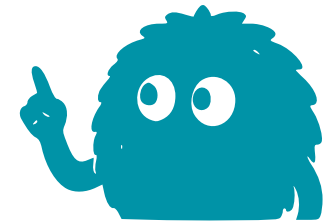
- **Reuse software**

We can continue to use and maintain it over the next ten years.  
Someone else can understand, install and extend it.

- **Reproduce results**

We can reconstruct how a scientific result was obtained.

**Software engineering is not the goal. Reliable science is.**

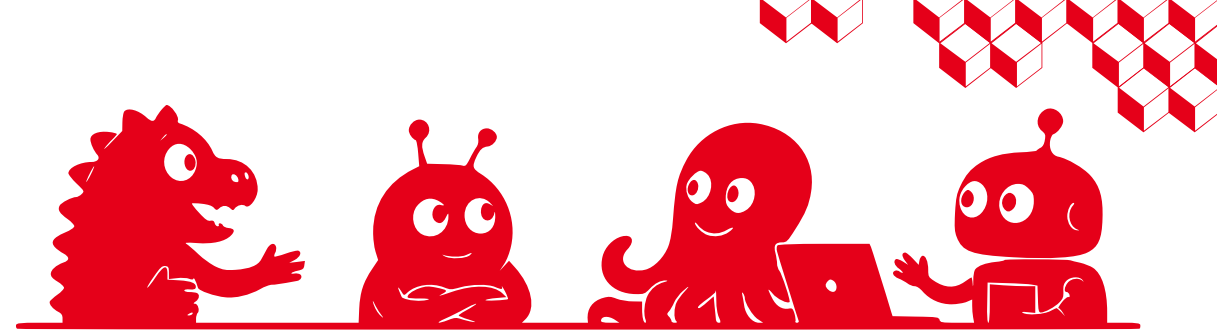


# Software development tools at IRFU

- GitLab (<https://drf-gitlab.cea.fr/>): projects repositories, few runners for CI, private registry for packages, private only pages
  - No public GitLab pages (GitLab in2p3)
  - No container registry (Self-maintained instances for Svom, GitLab in2p3)
- Quality reports (e.g. Sonarcube): not available at IRFU (Self-maintained instances for Svom and Euclid, ESA instance for LISA)
- Deeplab (<https://deeplab.intra.cea.fr/>):
  - GitLab, container registry, nextcloud (but INTRA only)
- IA, agents:
  - Maïa (<https://maia-portail.intra.cea.fr/>): Maïa-CHAT and Maïa-CODE (with IDE integration, but INTRA only and limited choice of models)
- Project documentation: GitLab wiki pages, Redmine in2p3

Setting-up and maintaining all these tools, and training users requires involvement of experts (at IRFU level).

# Questions & discussion



- Where is your code in the impact/maturity scale?
- What would you like to improve first?
- Did you try to implement such concepts?
- What do you think is the most important/difficult step?
- Did you face issues?
- What would you like to be automated?
- You're never alone (code editor, team, IA)
- Were you lacking human resources (DevOps, software developers)?