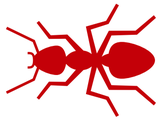
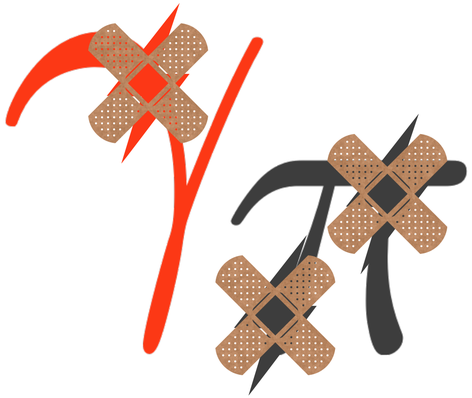
NOTHING WORKS

Lessons Learned from Leading and Maintaining the Open Source Project Gammapy

Axel Donath - July 23rd
Workshop on Open-Source Software Lifecycle

WHY "NOTHING WORKS"?

- When developing software in general we are confronted with the state **"nothing works"** most of the time:
  - We typically spend more time to **debug code** than to write it 
  - We fix **broken CI** tests build failing codecov 71%
  - We find that **design choices do not scale** or do not work as expected
  - We find that our software has **missing features**
  - We find the **performance is not good enough**
  - etc.



```
MemoryError: Unable to allocate 7.11 PiB for an array with shape
 (1000, 1000, 1000, 1000, 1000) and data type float64
```

- **Software is never in a 100% final state**, but rather undergoes a process of constant improvement. As we can never avoid bugs, missing features, changing API, design mistakes etc. completely. So the 2nd best thing we can do is to **setup a working process for the software that tolerates this "imperfections"** and results in the best possible compromise between user and developer experience

WHAT IS GAMMAPY?

H.E.S.S.

cta
cherenkov telescope array

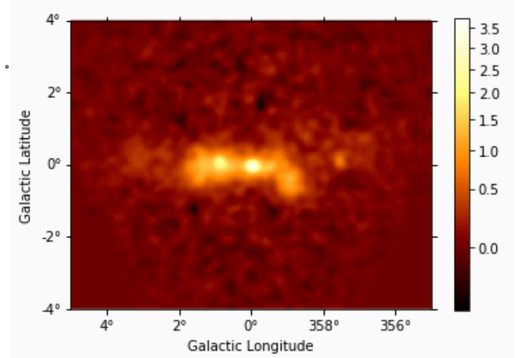
Fermi
Gamma-ray
Space Telescope

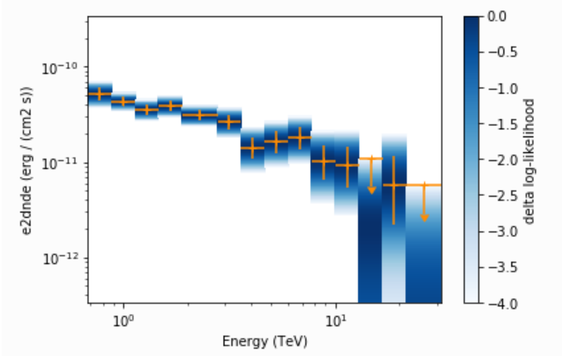
KM3NeT

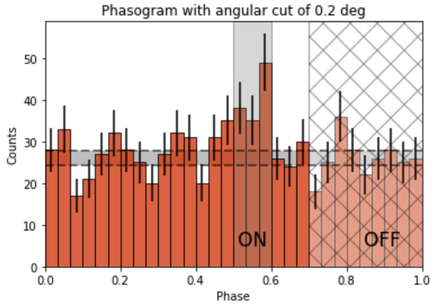
astropy

NumPy

SciPy

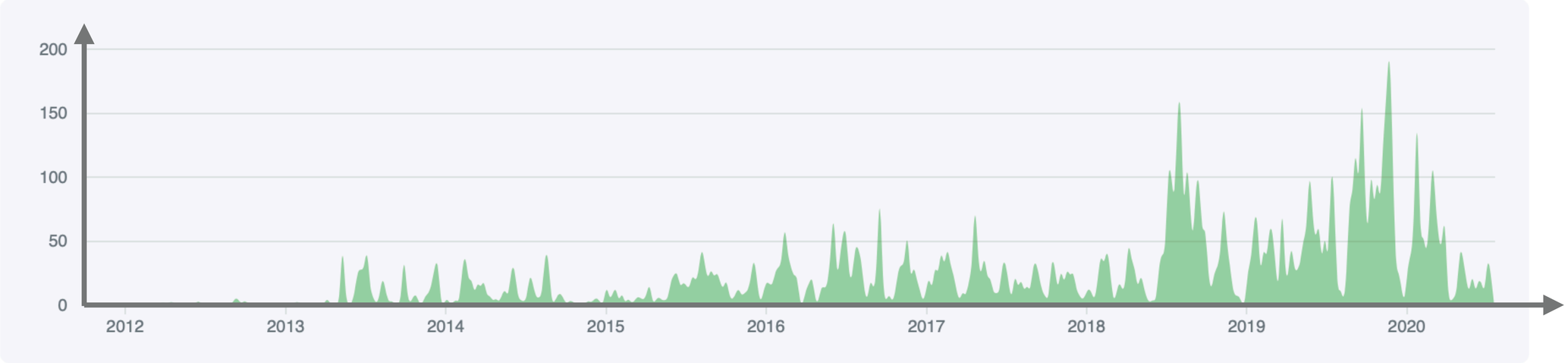






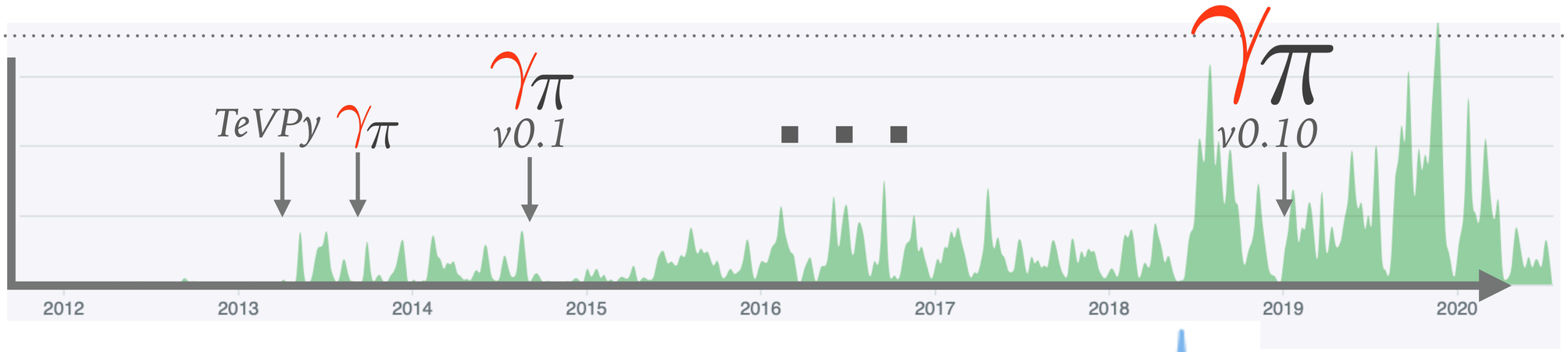
WHAT IS GAMMAPY?

- An openly developed **Python package for Gamma-Ray astronomy**
- Started in ~2012 with a set of Python scripts developed for the HESS Galactic Plane Survey analysis by Christoph Deil and myself
- Approximately 8 years of experience in developing and maintaining an open source Python package and have probably re-written the package already 3 times…:-)
- **~120 forks, ~60 contributors**
- **~13.000 commits**



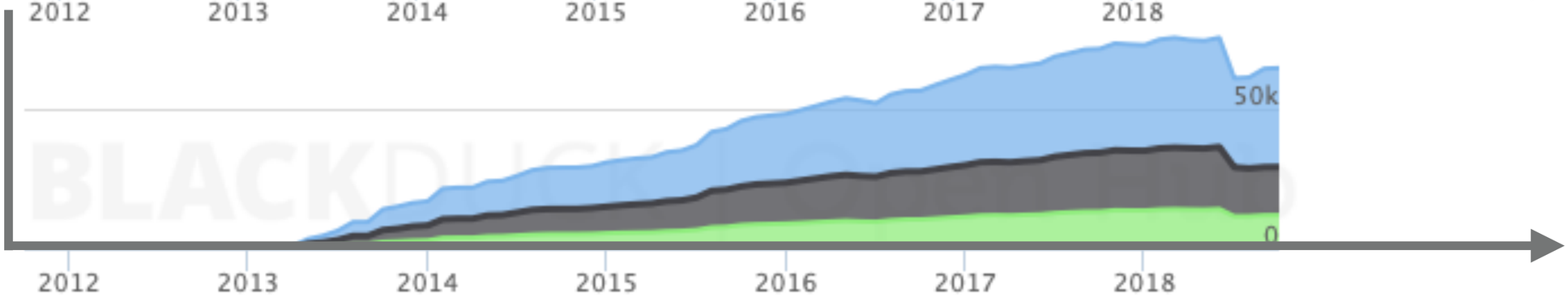
WHAT IS GAMMAPY?

Activitity



Contributors

Lines of code

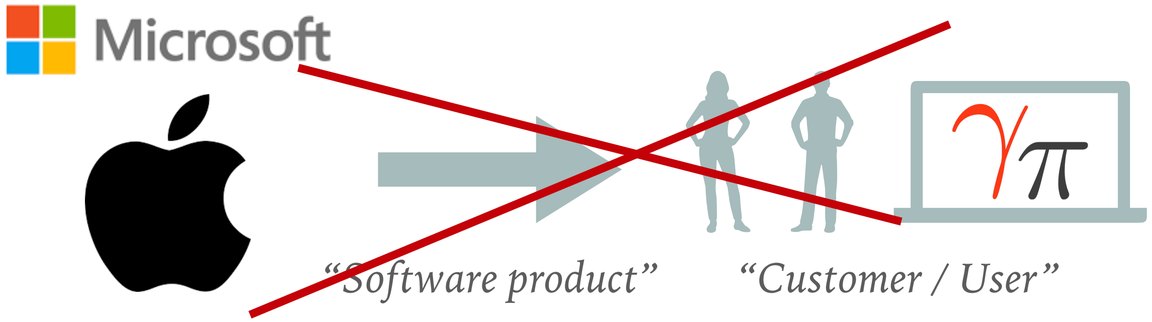


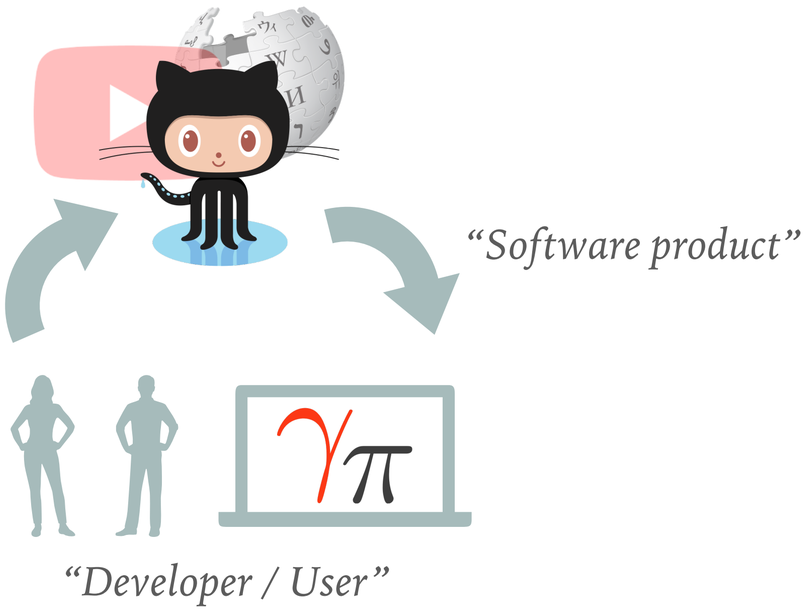
Code Comments Blanks

https://www.openhub.net/p/gammapy

“COMMUNITY DRIVEN” / OPEN DEVELOPMENT

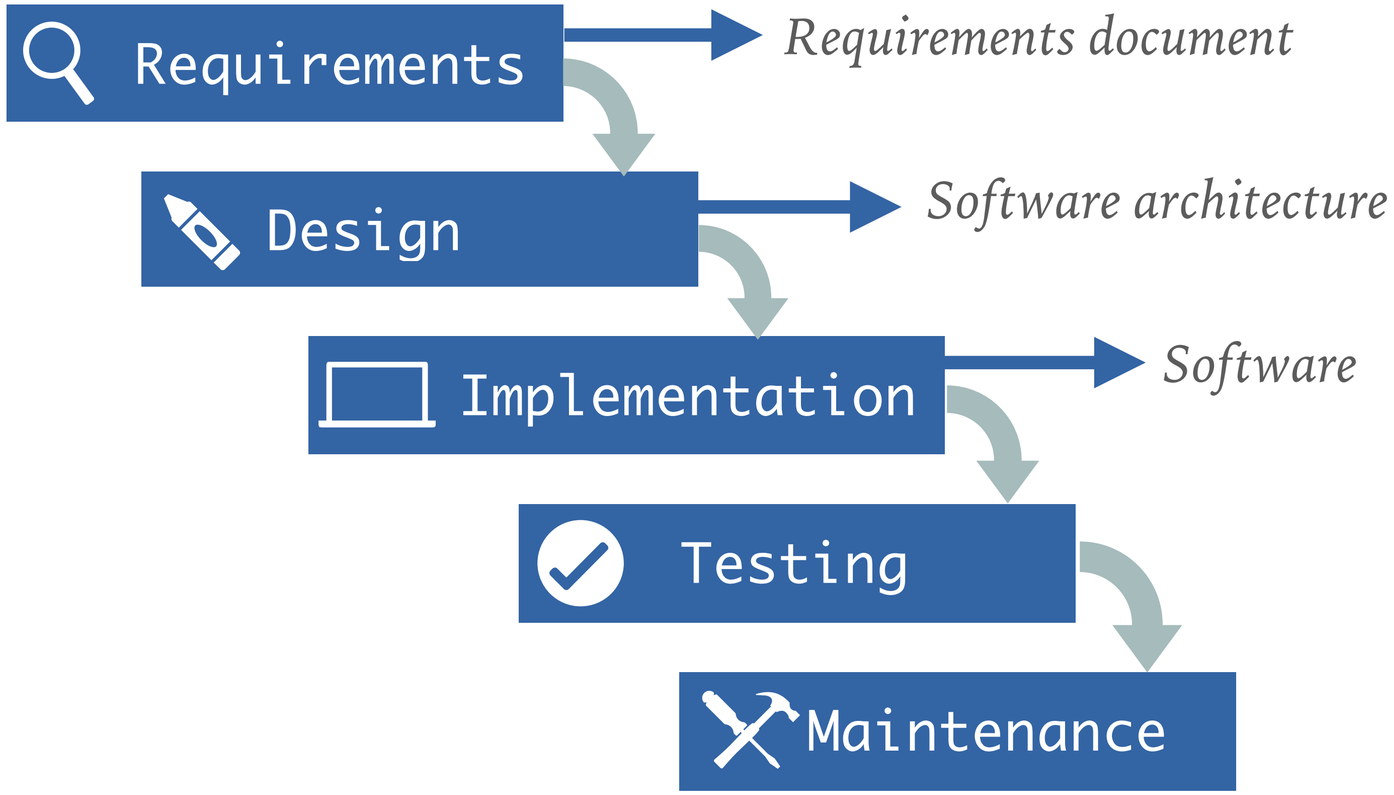
- Not a classical linear “developer -> user” or “company -> customer” relationship
- **No strict, structural boundary between developers and users**
  - Users know best what they need and deliver it for the benefit of the community
- Self organising structure incl. quality insurance, based on “crowd intelligence”
- Maybe **best understood in a Web 2.0 context**? There is no strict boundary between content creator and consumer…(Youtube / Wikipedia etc.)





"CLASSICAL" SOFTWARE DEVELOPMENT LIFE CYCLE

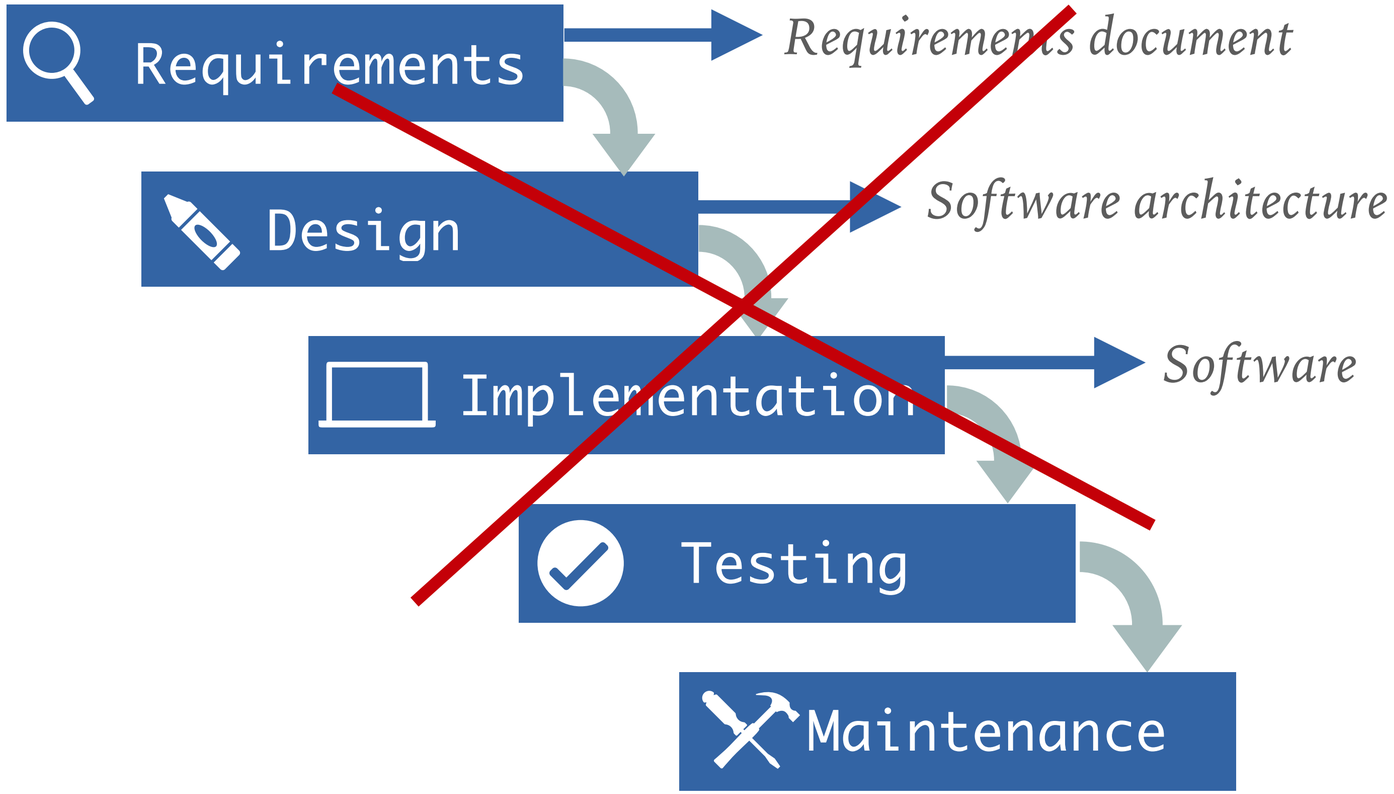
- E.g. **"Waterfall model"**: a company **delivers product to customers** / developers implement software for users



https://en.wikipedia.org/wiki/Waterfall_model

"CLASSICAL" SOFTWARE DEVELOPMENT LIFE CYCLE

- E.g. **"Waterfall model"**: a company **delivers product to customers** / developers implement software for users



Rather static and clumsy process

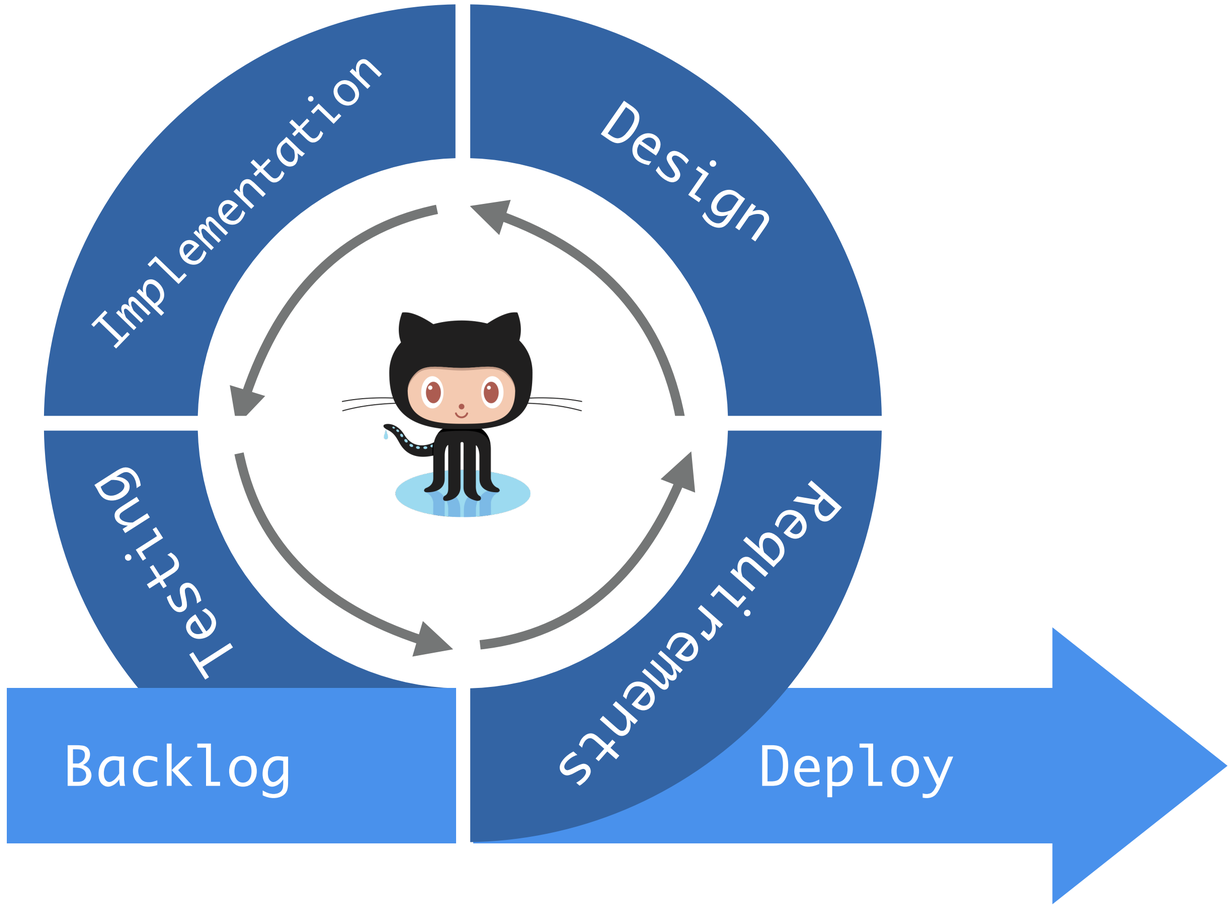
Not really suitable if:

- *there is no classical company / customer situation…*
- *requirements change fast*
- *new (science) use cases arise*
- *developer team changes*

https://en.wikipedia.org/wiki/Waterfall_model

AGILE SOFTWARE DEVELOPMENT LIFE CYCLE

- Constant process of improving the software
- Agile development methods
  - Pair programming
  - Refactoring
  - Test driven development
  - Continuous integration
  - Sprints
  - etc.



"Nothing Works"

Agile process more suited to openly developed software…

https://en.wikipedia.org/wiki/Agile_software_development

"STATE OF THE ART" OPEN DEVELOPMENT SETUP

Open software…

…is hosted on an **open git server** e.g. Github:
https://github.com/gammapy/gammapy

…has **code format standards** or uses an auto format tool e.g. Black

…has the **documentation coupled to the source code** and uses automatic tools e.g. Sphinx to build it

…uses a **continuous integration** system e.g. Travis-CI or Azure Pipelines

…implements tests and uses a **testing framework** e.g. Pytest

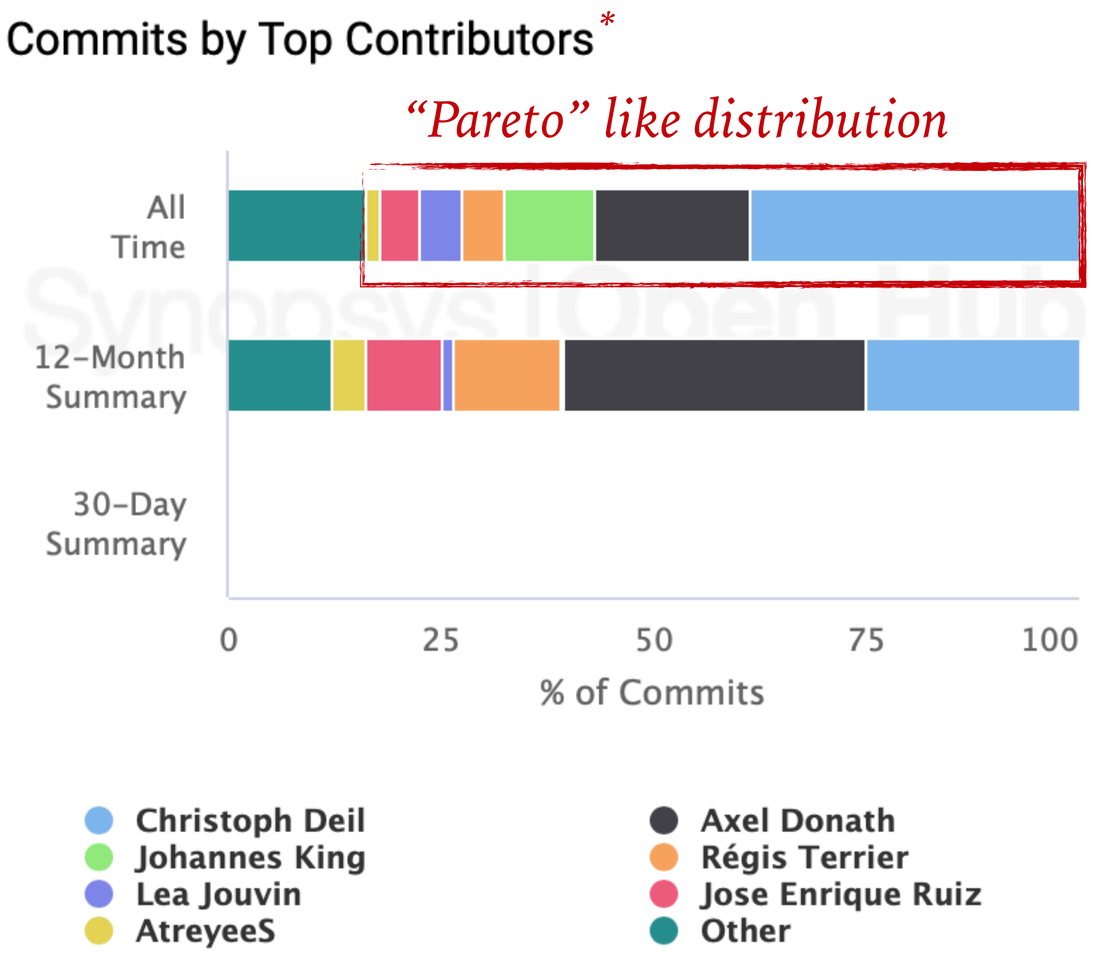
…implements tests for a large fraction of the code "**large coverage**"

…builds and **deploys docs** automatically e.g. on Read the Docs

CONTRIBUTORS

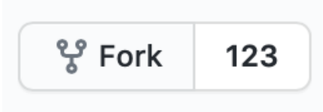
https://www.openhub.net/p/gammapy

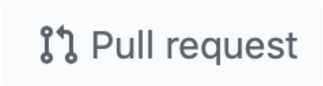
- Approximately **20% of the top contributors did 80% of the commits** ("Pareto" like distribution, seems to be the case for many projects…)
- Only **few long-term contributors** ("core developers")
- **Many single time contributors**, a handful of short-term contributors (< 1 year)
- Often **highly intrinsically motivated contributors**, but often specialised tasks. Sometimes it's needed to slow them down…



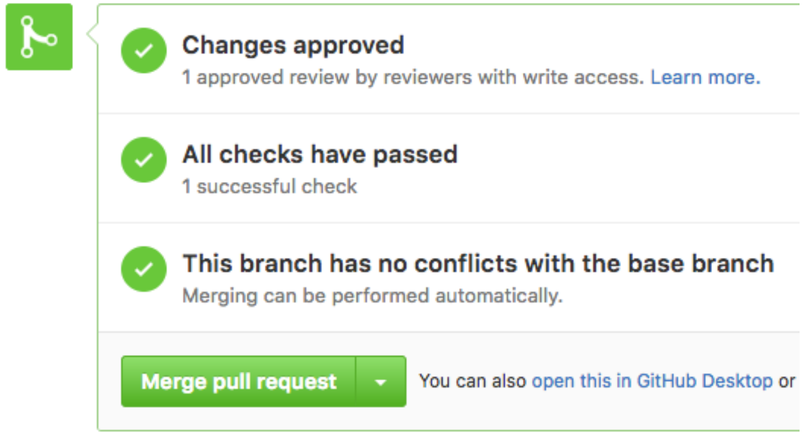
** # of commits are not necessarily a good way to measure contributions…*

“GITHUB” WORKFLOW

- Standard **multi-branch git workflow**:
  - Contributors **fork a repository** 

  - Features are developed in **new a branch** “on the side”
  - A **pull requests** is opened 

- Every pull requests (PR) is **reviewed at least once** by more experienced developers (lead developers). Sometimes “all fine”, sometime “Here is a number of substantial comments”
- Once review comments are implemented and the CI builds pass a PR gets merged
- **Lesson learned**:
  - Often contributors put **too many changes** (or possibly unrelated) **in a single PR**. This is hard to review…
  - Try to ask for **small PRs**
  - Using **automatic code formatting tools** (e.g. Black) can help to to get small diffs
  - **Avoid PRs staying open for a long time:** chances increase with time, that it will never get merged (merge conflicts etc.)



REQUIREMENTS / DESIGN PROCESS

Requirements | Design

PEP *"Proposal for enhancing Python"*

APE *"Astropy proposal for enhancement"*

GAMMAPY "PIG" DOCUMENTS

Requirements | Design

PEP

APE

PIG

"Proposal for Improving Gammapy (PIG)"

PIG 1 - PIG purpose and guidelines

- Author: Christoph Deil
- Created: December 20, 2017
- Accepted: January 9, 2018
- Status: accepted
- Discussion: GH 1239

Abstract

PIG stands for "proposal for improvement of Gammapy". This is PIG 1, describing the purpose of using PIGs as well as giving some guidelines on how PIGs are authored, discussed and reviewed.

What is a PIG?

This article is about the design document . For other uses, see Pig (disambiguation)

Proposals for improvement of Gammapy (PIGs) are short documents proposing a major addition or change to Gammapy.

PIGs are like APEs, PEPs, NEPs and JEPs, just for Gammapy. Using such enhancement proposals is common for large and long-term open-source Python projects.

GAMMAPY "PIG" DOCUMENTS

Requirements | Design

- A "PIG" plans a larger contribution with O(10) PRs
- Written either by experienced developers or by contributors & and experienced developers
- PIGs go through pull request, discussion and an **official acceptance process**
- So far ~20 Pigs in Gammapy, **16 accepted and implement 4 withdrawn / rejected**.
- Lessons learned:
  - PIGs proved to be very **useful in the design process**, lead to better quality code!
  - Keep PIGs small, often failed when the focus was too large and tried to solve to many problem at once
  - In the beginning we also asked non-experienced developers to write a PIG for their project, which was an overburden….

PIG 3 - Plan for dropping Python 2.7 support

- Author: Christoph Deil & Matthew Wood
- Created: Feb 1, 2018
- Status: draft
- Discussion: GH 1278

Abstract

We propose to drop Python 2.7 support in Gammapy v0.11 in March 2019.

All earlier Gammapy versions, up to Gammapy v0.10, support Python 2.7 and of course will remain available indefinitely.

User surveys in 2018 have shown that most Gammapy users are already on Python 3. Gammapy v0.8 shipped with a recommended conda environment based on Python 3.6 that works on Linux, Mac and Windows and can be installed by anyone, also on older machines.

To support Fermipy, which uses gammapy.maps and still requires Python 2.7, as well as other users on Python 2.7 (if any), we will backport bug-fixes and make patch releases in the Gammapy v0.10.x branch as needed, throughout 2019.

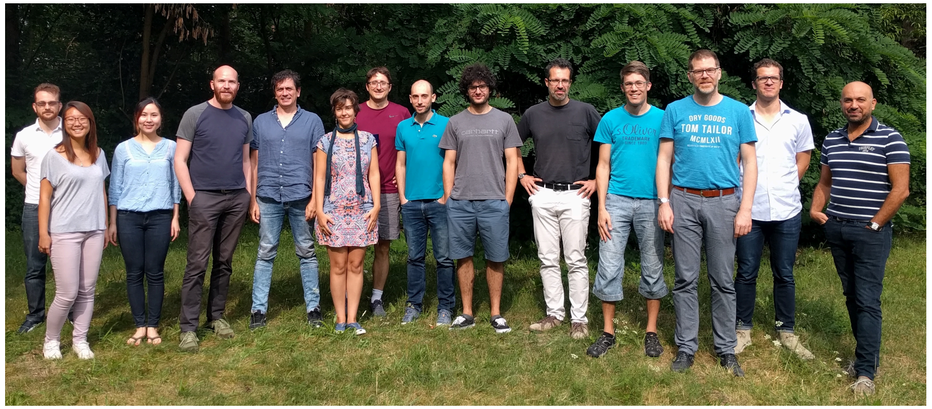
https://docs.gammapy.org/0.17/development/pigs/index.html

PIGs are useful animals…

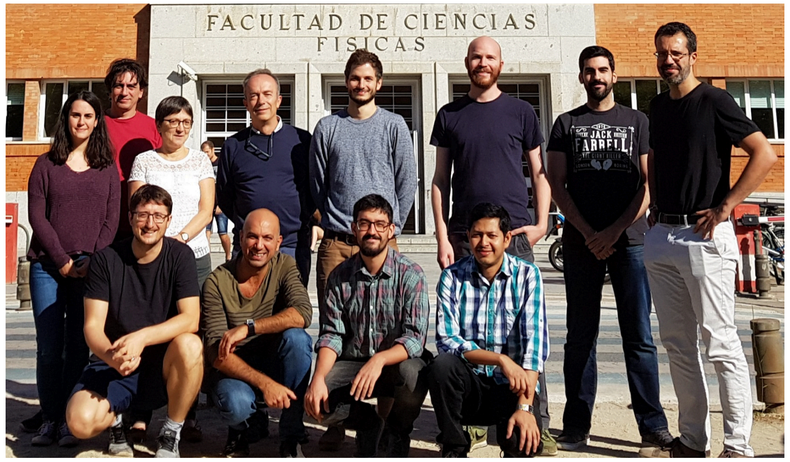
CODING SPRINTS

Implementation

- Proved to be **effective for**:
  - Introducing / **teaching new developers**
  - Effective **decision taking / discussions**
  - Wrap up work before a release
  - **Enhance the community** in general



Erlangen in July 2019



Madrid in October 2018

- Meet other developers / contributors in person for ~ **1 week and work together**
- **Highly recommended!**
- During pandemic: replaced by a **co-working week** (less effective but acceptable…)

https://github.com/gammapy/gammapy-meetings/tree/master/coding-sprints

CODE PARADIGMS

```
import this
```

Implementation

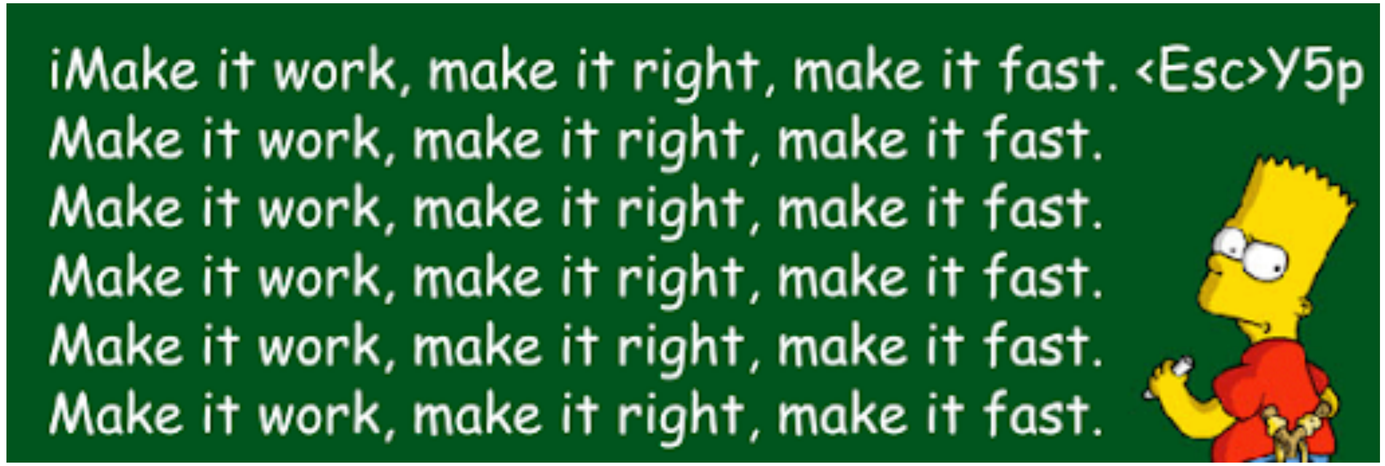
```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

If there is doubt on a design decision, or choice between multiple solutions to implement the same feature…ask the Zen of Python!

CODE PARADIGMS

Implementation



Kent Beck's directive interpreted:

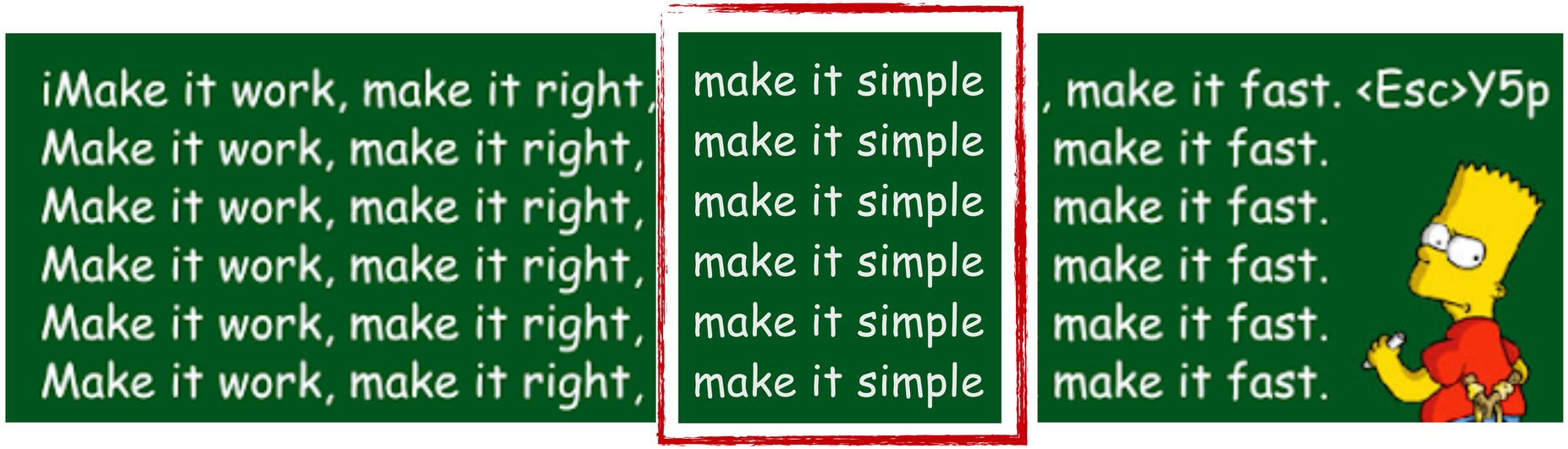
"Make it work correctly,
make the source code clear,
make it run quickly…"

- Code paradigms are helpful…too often forgot in the daily routine

CODE PARADIGMS

Implementation

Let's make it explicit….



- Writing **simple code is important**!
  - Splitting code into **re-usable functions / classes**, meaningful variable names etc.
- In openly developed projects, there is **no "code ownership"**, ideally everyone should be able to understand any piece of code with limited effort
- Simple code (in contrast to "spaghetti code") ensures **long term maintainability** and modification by "non experts" for this specific piece of code
- "**Premature optimization is the root of all evil**" is a similar paradigm and equally true…

OTHER LESSONS LEARNED

State the obvious…?

Implementation

- Many (non-regular) **contributors do not know the whole code base** of a project, so they fix problems in the part of the code, they are working on, even if the problem originates from a completely different position. Take care in the review process that **problems are fixed "up in the hierarchy"** and not locally, so that they apply to the whole code base…
- **Design mistake**: In the beginning we didn't have a design (PIG) process so many data and models abstractions were implemented on a **single use-case driven** basis. Like `CountsSpectrum`, `SkyImage`, `SkyCube` as needed. Long-term this created a proliferation of classes, non-uniform API and required a long refactoring process of unification. So **don't restrict dimensionality of data and models unless you know exactly the requirements**…
- **Many people just use the software, but never start contributing**. This is mostly fine but often users write again their own code and prototypes, **implementing new features but never share it with others**:
  - Negative interpretation: they are "selfish" taking advantage of other's work and not giving back to the community…
  - Positive interpretation: they are "code shy" so motivate them to make the work public…
- And **write simple code**!

RELEASE CYCLE

Deployment

- Rather **short release cycle** of ~2 months
  - Ensures **continuous progress** by working towards "deadlines" regularly
  - Short cycle of user **feedback, design phases and implementation**
  - Found a bit too short for larger development projects…
- Requires **simple deployment system**:
  - Releases are put on `pypi` (**Python package index**) (https://pypi.org/project/gammapy/) and **Conda**
  - Every Gammapy version is delivered with a new **Conda environment file**:

1
```
$ curl -O https://gammapy.org/download/install/gammapy-0.17-environment.yml
```

2
```
$ conda env create -f gammapy-0.17-environment.yml
```

3
```
$ conda activate gammapy-0.17
```

RELEASE CYCLE

Deployment

- The short release cycle and simple install system is a reasonable **compromise for a package with a non-stable API** between:
  - 1. desire from contributors to **develop the package fast and** not spend too much time on backwards compatibility
  - 2. and not require users to update every time, only if they are in need for a certain new feature. Using Conda environments users can keep **multiple version at the same time**
- Currently release process requires manual work so **making a release takes ~0.5 day. Plan to automatise** this in future by setting up a release pipeline (e.g. using GitHub Actions)

GitHub Actions

COMMUNITY EFFORTS AND RELATION TO ORGANISATIONS

- Community driven projects are **typically independent and live from voluntary contributions**.
- Currently mostly indirect support by institutions / companies by giving people the time to work on projects
- In case of success of community driven projects, there is a **need for long-term maintenance. This either requires institutional support** or creating an organisational structure, which receive funding.
  - E.g. Astropy, received 900K funding as an organisation from Moore foundation and Space Science Telescope (JWST) pays software developers to work on Astropy

- **Open question:** How to establish a sustainable collaboration between open source projects and organisations?

cta cherenkov telescope array

H.E.S.S.

Etc.

astropy

James Webb Space Telescope JWST NASA • ESA • CSA

Rubin Observatory

CONCLUSIONS

- **Accepting the state "nothing works" as a normal state**, forces one to develop a working process that allows for "failure" and fast correction of mistakes as well as prevention of mistakes
- For Gammapy **"Agile" inspired methods worked best so far**: GitHub Workflow, code review, coding Sprints, short release cycles, etc.
- Biggest challenge for the "community driven" projects is **long-term maintenance and institutional support**