Organizing Small Software Projects

Ben Morgan

WARWICK
THE UNIVERSITY OF WARWICK

What is a “Small Software Project”?

- In the context of this meeting, software projects that may be
  - ***Specific**: e.g. to simulate/process/analyse data from an experiment*
  - ***General**: e.g. event generators used by many experiments*
- The metric here for “Small” is ***total developer FTE***
  - *Not always related to size of the experiment/community using the software*
  - *FTE because developers may **only be funded a fraction of their time** to work on the software/experiment*
  - *This is slightly biased towards specific projects, as general ones often have an active, though transient, population of contributors*
- To take the SuperNEMO $0\nu\beta\beta$ search experiment as an example:
  - ***~100 researchers, 1-2FTE on software from 2-8 developers***

Typical challenges faced in smaller projects

- **Limited FTE**, often with high turnover from limited funding/short contracts
  - *“Get it working” pressure can lead to neglect of long term support items*
  - *Easily leads to a critical Bus Factor of 1-2*
- **Not Invented Here** syndrome (*reinventing the wheel*)
  - *Usually driven and exacerbated by the above*
  - *Fewer developers can mean less breadth of knowledge/experience*
- Often a high level of **overthinking/engineering**
  - *Not unique to small projects, but creates further long term support issues*
  - ***Developers ≅ Users***
- Importance of software work to an experiment or the community **may not be recognized** by funding bodies (or even experiment members!)
  - *Can be particularly challenging for community wide projects*

Addressing Challenges

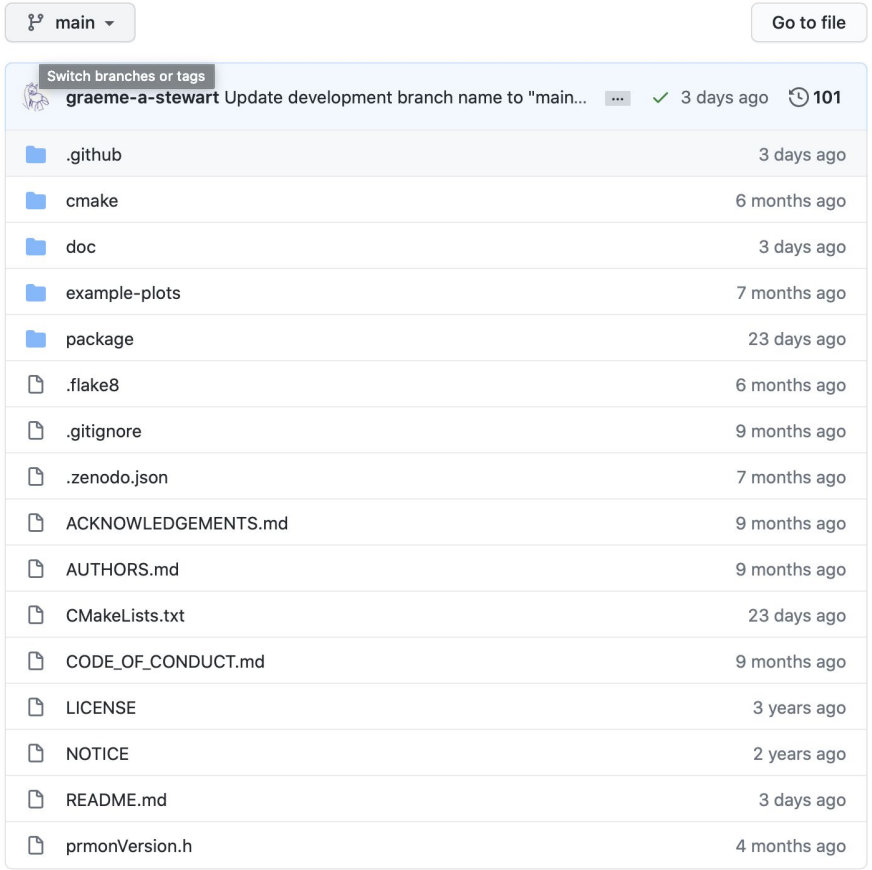
- In all cases, the challenges are best addressed by projects **implementing, maintaining and focussing on the foundations**
  - *Documentation (README/CONTRIBUTING, Sphinx, Doxygen)*
  - *Build and testing (CMake/CTest/etc)*
  - *Code style/quality tools (clang-format/tidy, Black/flake8)*
  - *Package management/deployment (Conda/Pip/Spack/CVMFS etc)*
  - *Version Control and Continuous Integration workflows*
- Might be obvious (*even boring*), but **neglecting these will build up high interest technical debt** that sooner or later will have to be **repaid in FTE**
  - *Been there, done/doing that*
- There is a growing recognition of the criticality of sustainable and reproducible software for research, **but never be afraid to repeat the case!**

Tools and Suggestions for Small Projects

- Managing Code and Documents
- Development Workflows
- Release and Deployment
- Caveats:
  - *Strong C++ bias*
  - *I'm just as guilty of not always doing what I suggest...*
  - *There will no doubt be a lot of "but…" and "what about…" questions (please write them in the Live Notes, they are **always** welcome!)*

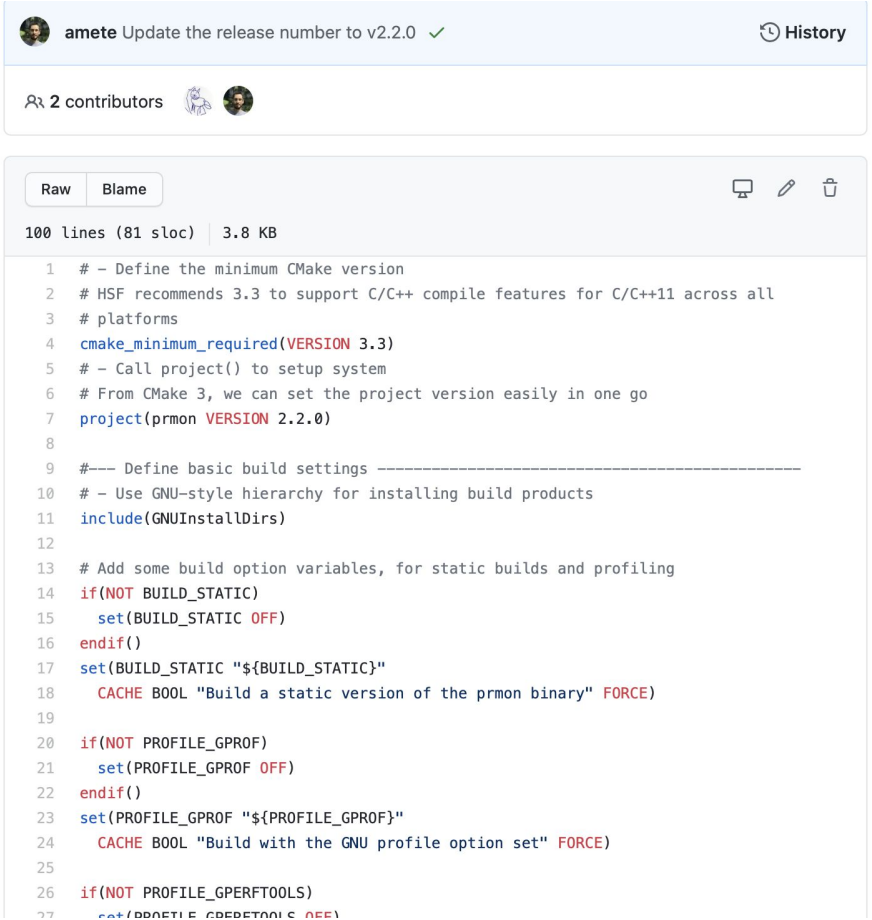
Organizing the Project

- **Always** have needed “first contact” docs like `README`, `CONTRIBUTING`
- **Never** a need for a “special” layout of source or other code
  - *Languages often have a recommended/standard layout*
  - *Keep it simple, logical otherwise*
- Might seem trivial, but you want **principle of least surprise**!
  - *Minimize mental start up and context switching costs*



Building the Project

- Always use a standard build tool
  - *CMake, Autotools, Setuptools*
- **Keep scripts simple and standard**
  - *Avoid bikeshedding and over-engineering!*
- **Build scripts are code** so document and maintain them as such
- **A simple, well maintained build system pays dividends in later items!**



amete Update the release number to v2.2.0 ✓ History

2 contributors

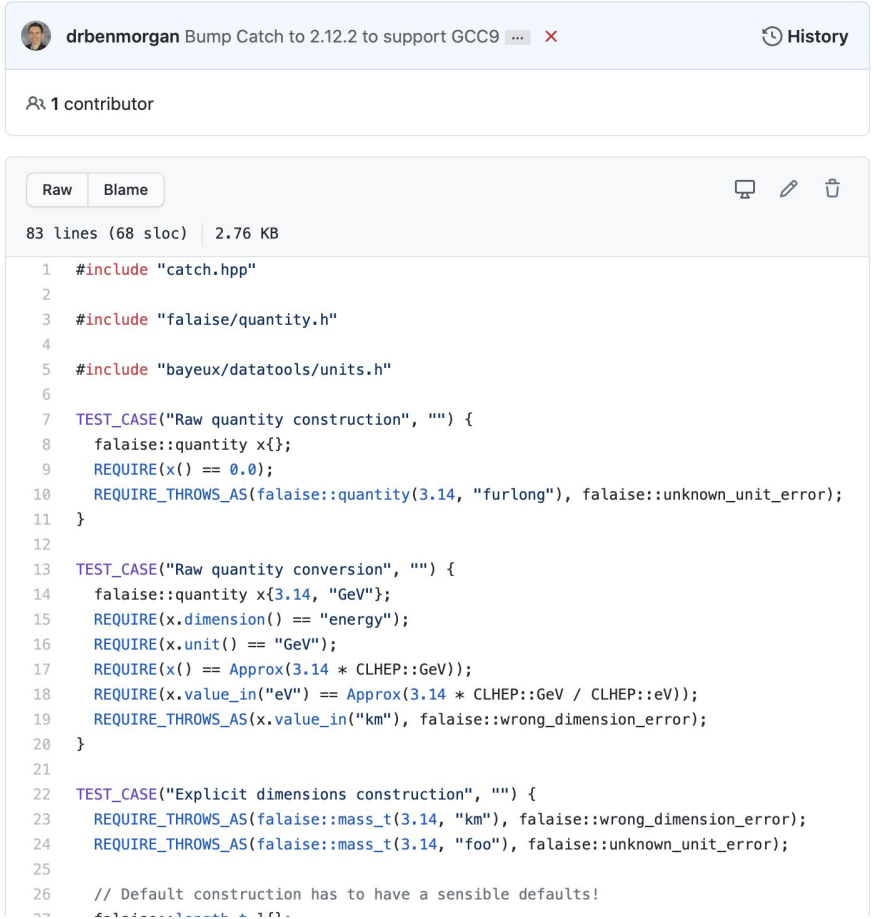
Raw Blame

100 lines (81 sloc) 3.8 KB

```
1  # - Define the minimum CMake version
2  # HSF recommends 3.3 to support C/C++ compile features for C/C++11 across all
3  # platforms
4  cmake_minimum_required(VERSION 3.3)
5  # - Call project() to setup system
6  # From CMake 3, we can set the project version easily in one go
7  project(prmon VERSION 2.2.0)
8
9  #--- Define basic build settings ----------------------------------------------
10 # - Use GNU-style hierarchy for installing build products
11 include(GNUInstallDirs)
12
13 # Add some build option variables, for static builds and profiling
14 if(NOT BUILD_STATIC)
15   set(BUILD_STATIC OFF)
16 endif()
17 set(BUILD_STATIC "${BUILD_STATIC}"
18   CACHE BOOL "Build a static version of the prmon binary" FORCE)
19
20 if(NOT PROFILE_GPROF)
21   set(PROFILE_GPROF OFF)
22 endif()
23 set(PROFILE_GPROF "${PROFILE_GPROF}"
24   CACHE BOOL "Build with the GNU profile option set" FORCE)
25
26 if(NOT PROFILE_GPERFTOOLS)
```

Testing the Project

- Always use a common/standard testing framework
  - *E.g. Catch2/googletest*
  - ***stdout+Mk1 eyeball isn't!***
- **Test-driven development** can be very useful, both to
  - *Ensure tests are written(!)*
  - *Help clarify interfaces/contracts*
- Write tests to **triage+fix bugs**
- Ensure tests **are easily built/run as part of the development workflow**
  - *E.g. `make test`*



Documenting the Project

- Includes `README`, `CONTRIBUTING`, `INSTALL` files etc.
- Use standard tools like Doxygen, Sphinx for API/User Guides
- **Like tests, document-as-you-go**
  - *Will be clearer at time of writing*
  - *Helps you to clarify your design*
- **"Compile" docs as part of build**
  - *Check for mistakes!*
  - *Helps in release stage*
- Encourage users to contribute - **they bring a different perspective**

Formatting the Code

- **Enforce a coding style to ensure consistency and familiarity**
  - *E.g. `clang-format` for C/C++, Flake8 for Python*
- Use of a tool reduces the chances of a holy war over spaces/braces
  - *An area developers can be pointlessly opinionated about*
- Use **Git hooks and IDE integrations** to format on save/commit

Falaise / .clang-format

drbenmorgan Bump column width to 100 ... History

1 contributor

Raw Blame

98 lines (97 sloc) 2.78 KB

```
---
Language:        Cpp
# BasedOnStyle:  Google
AccessModifierOffset: -1
AlignAfterOpenBracket: Align
AlignConsecutiveAssignments: false
AlignConsecutiveDeclarations: false
AlignEscapedNewlinesLeft: true
AlignOperands:   true
AlignTrailingComments: true
AllowAllParametersOfDeclarationOnNextLine: true
AllowShortBlocksOnASingleLine: false
AllowShortCaseLabelsOnASingleLine: false
AllowShortFunctionsOnASingleLine: All
AllowShortIfStatementsOnASingleLine: true
AllowShortLoopsOnASingleLine: true
AlwaysBreakAfterDefinitionReturnType: None
AlwaysBreakAfterReturnType: None
AlwaysBreakBeforeMultilineStrings: true
AlwaysBreakTemplateDeclarations: true
BinPackArguments: true
BinPackParameters: true
BraceWrapping:
```

Analysing the Code

- Use **static analysers** to suggest, or apply, fixes for style, performance
  - *E.g. `clang-tidy` (C/C++)*
- Usually easy to integrate with the build or test systems
  - *CMake makes `clang-tidy` use really easy, for example*
- **Particularly useful for modernizing or simplifying older projects!**

Falaise / .clang-tidy

drbenmorgan Initial clang-tidy config for Falaise ... ✓ History

1 contributor

Raw Blame

159 lines (158 sloc) | 7.47 KB

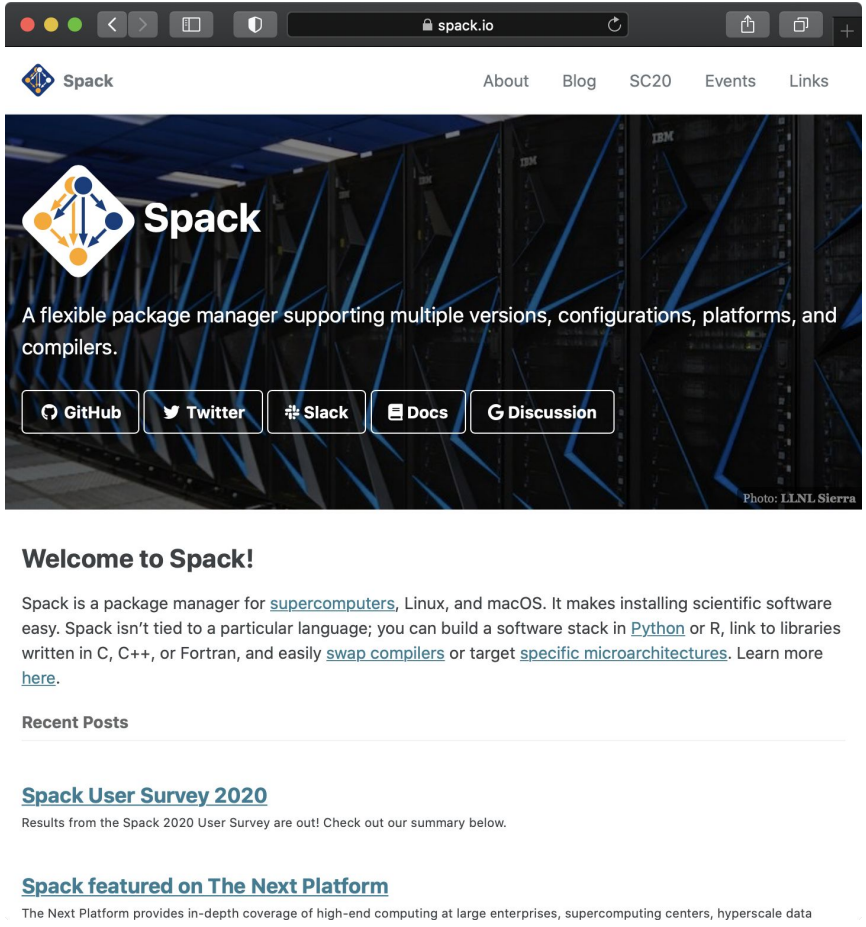
```
1  ---
2  Checks:          'clang-diagnostic-*,clang-analyzer-*,-*,readability-*,modernize-*,perf
3  WarningsAsErrors: ''
4  HeaderFilterRegex: ''
5  AnalyzeTemporaryDtors: false
6  FormatStyle:     file
7  CheckOptions:
8    - key:             cert-dcl16-c.NewSuffixes
9      value:           'L;LL;LU;LLU'
10   - key:             cert-oop54-cpp.WarnOnlyIfThisHasSuspiciousField
11     value:           '0'
12   - key:             cppcoreguidelines-explicit-virtual-functions.IgnoreDestructors
13     value:           '1'
14   - key:             cppcoreguidelines-non-private-member-variables-in-classes.IgnoreCl
15     value:           '1'
16   - key:             google-readability-braces-around-statements.ShortStatementLines
17     value:           '1'
18   - key:             google-readability-function-size.StatementThreshold
19     value:           '800'
20   - key:             google-readability-namespace-comments.ShortNamespaceLines
21     value:           '10'
22   - key:             google-readability-namespace-comments.SpacesBeforeComments
23     value:           '2'
24   - key:             modernize-loop-convert.MaxCopySize
```

Using Other Software in the Project

- All projects use **external packages - dependencies**
  - *Already seen these with build/test/documentation ones!*
- **When the project needs an "X", find a suitable off-the-shelf "X" first**
  - *"Suitable" means "Meets the requirements"*
  - *"I don't like the implementation/style"* ***isn't*** *a requirement*
  - ***One on the biggest time drains for a small project can be the implementation, maintenance, and development of these new wheels***
  - *You can (and should!) contribute back to the projects you use!*
- Make sure the project is not locked to a specific version of the external
  - *All languages have constructs to compile/branch on the version*
- A minimum, or range, of supported versions is fine though
  - *Most build systems support this, .e.g CMake's* `find_package`

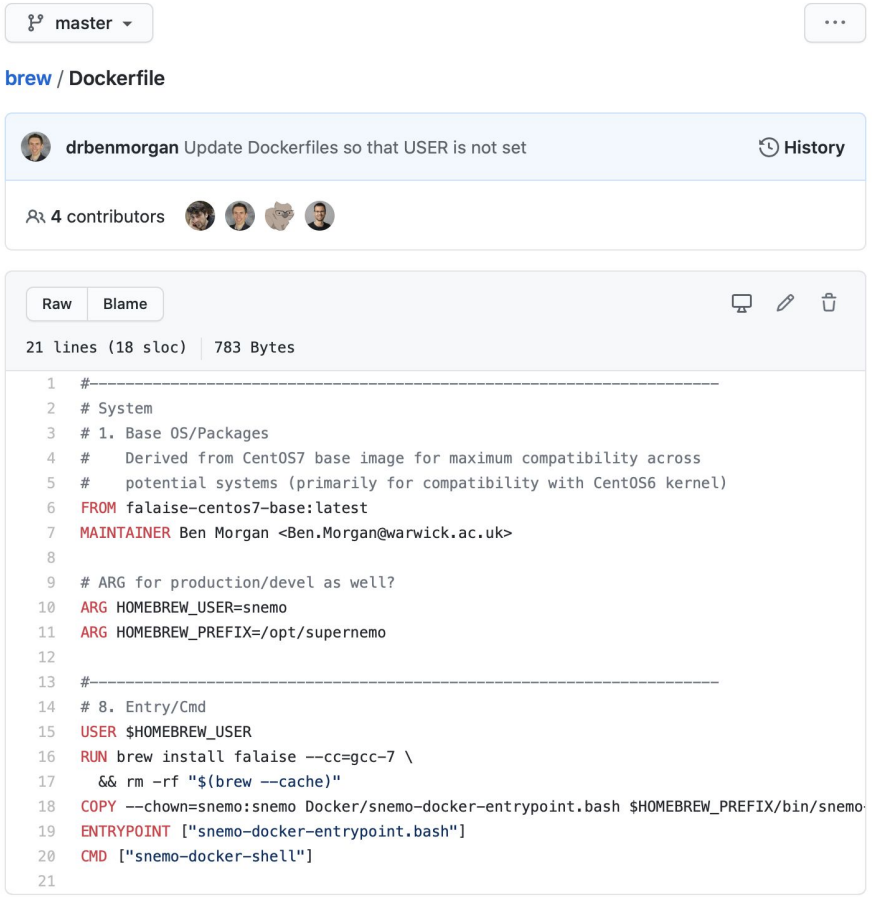
Package Managers

- Always install dependencies with a widely used ***package manager***
  - *Don't write one yourself, please...*
- Choice will depend on languages used and platforms targeted
  - *Language specific, e.g. `pip`*
  - *Platform specific, e.g. `apt`*
  - *General(ish), e.g. conda, spack*
- **Use a "requirements" file/package**
  - *Lists packages and versions for a given "environment"*
  - *Aids easy setup/reproducibility*



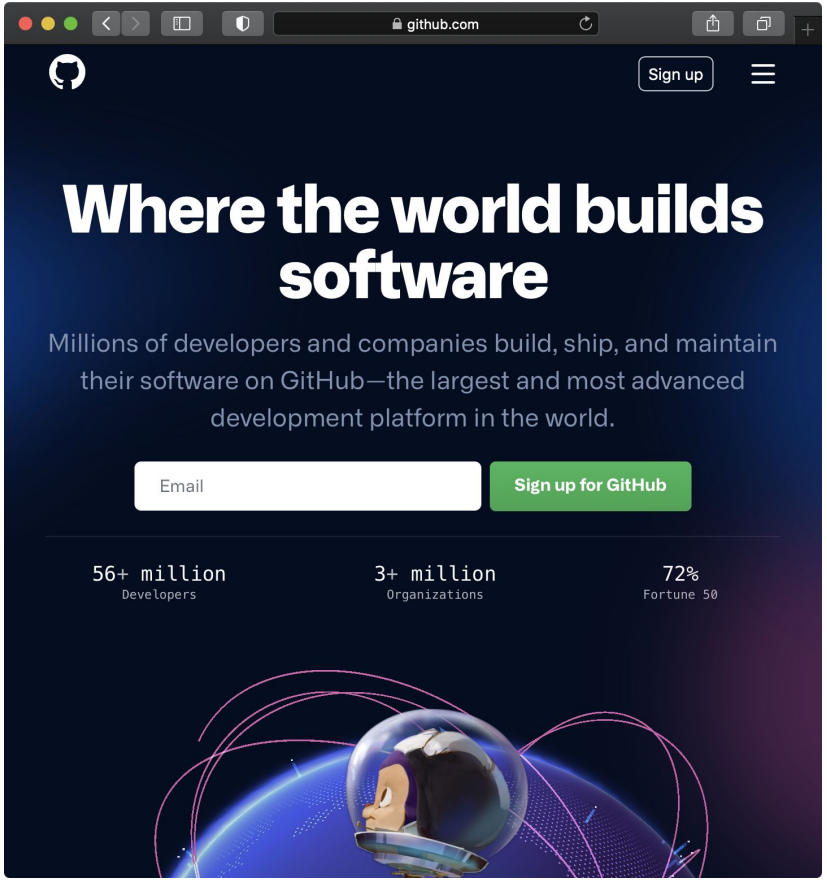
Docker/Singularity

- Containers can aid in distributing a development/use environment
  - *Modern IDEs can use them seamlessly (e.g. VSCode)*
- Package manager “environment” files and VCS/Container tags provide great system for reproducible setups
- **Don’t** use containers as a sticking plaster over a bad build/packaging system - **fix these first**!



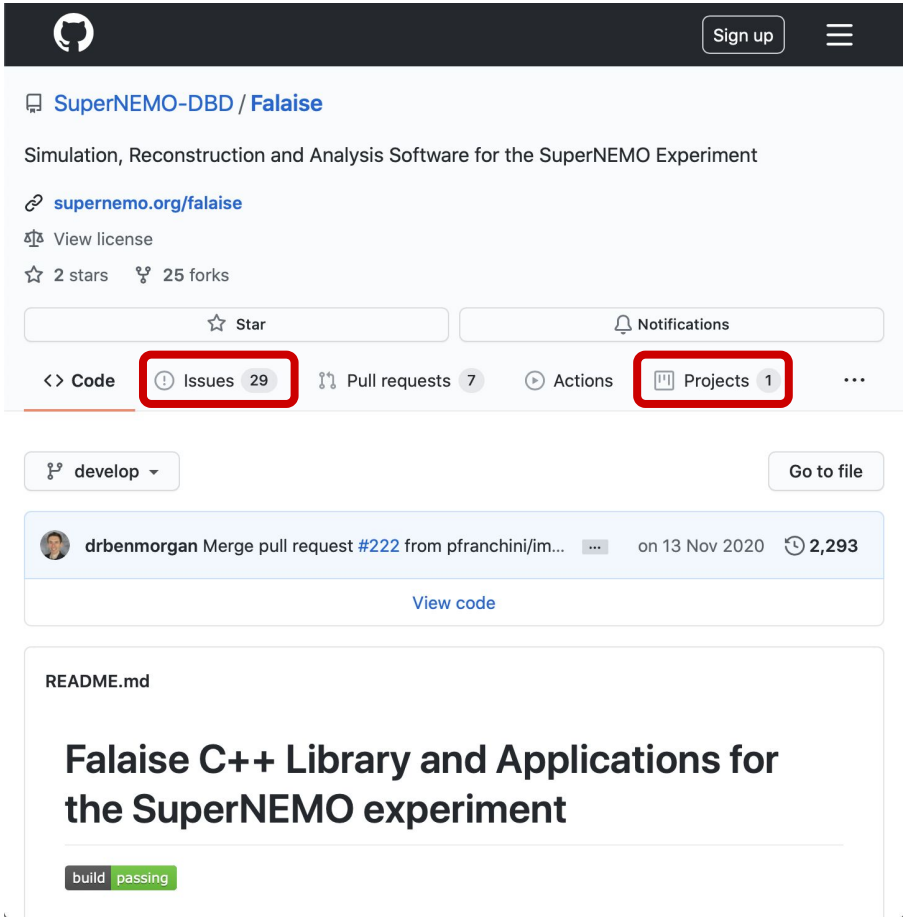
Hosting the Project

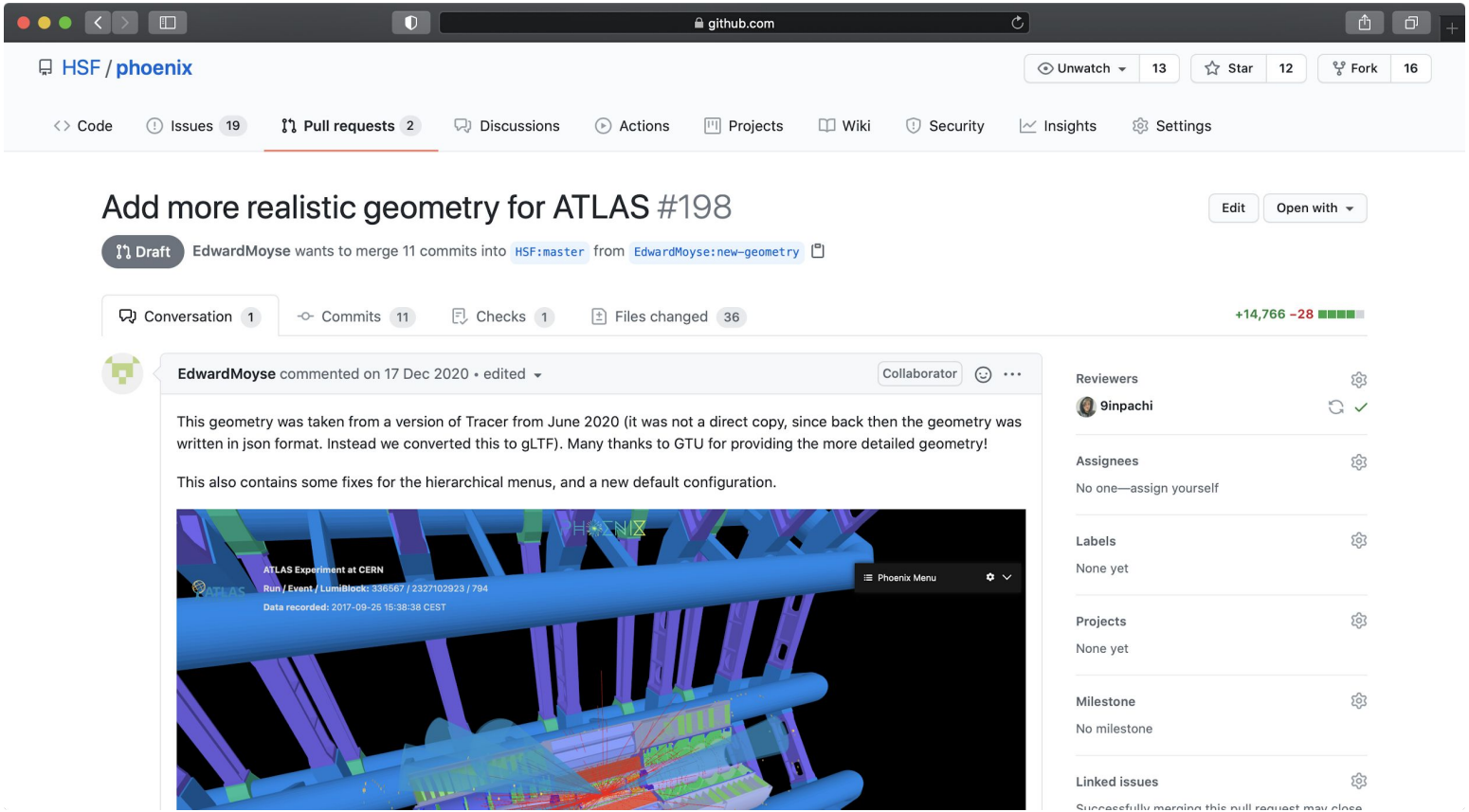
- **Git**+**GitHub** is the primary VCS and hosting solution, though many **GitLab instances at Institutes/Labs**
  - *These may be better if the project has specialist needs like access to GPU hardware for CI*
- Git repos are trivial to move, so you can always change later!
- **Never manage your own hosting** - you **cannot** do it as well as GitHub/Lab or Institute/Labs



Use Issues and Boards

- Use **Issues** as **primary** channel of communication to reduce “where do I go to...” and help engagement
  - *Email/Slack fine, **but encourage use of Issues!***
  - *Beta “**Discussions**” feature may be good for “How do I” questions*
- Use **Projects** to organise core tasks
  - *Can add Issues to specific Projects e.g. “Next Release”*
  - ***Helps focus effort and reduce context switching cost***



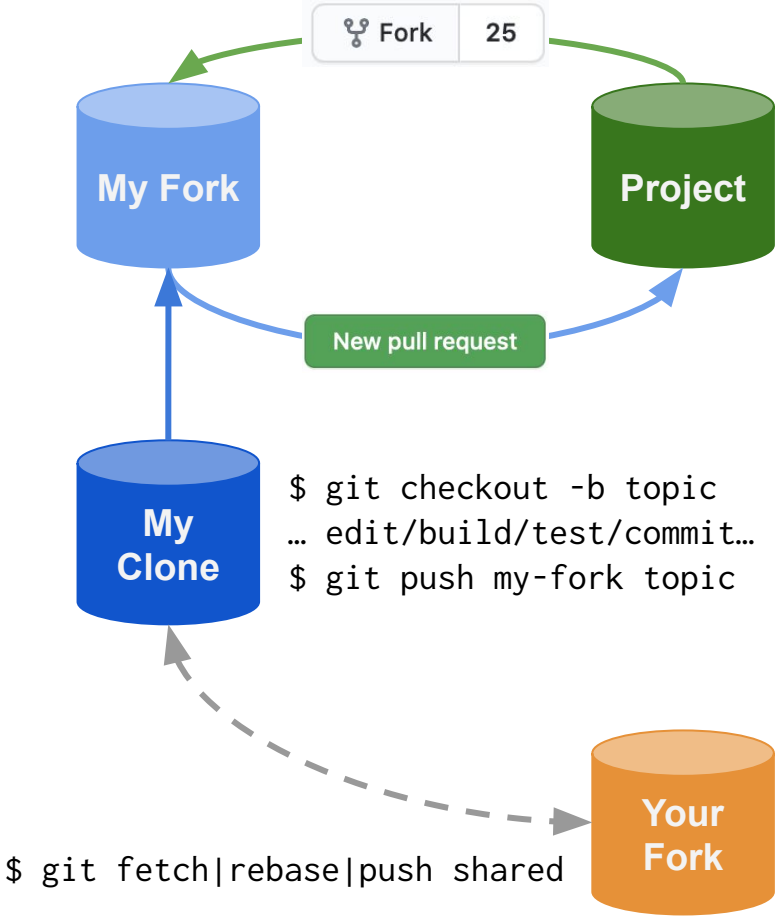


Developing the Code: Topic Branches and Pull Requests

Never commit/push directly to the `main` branch - you will break it at some point!

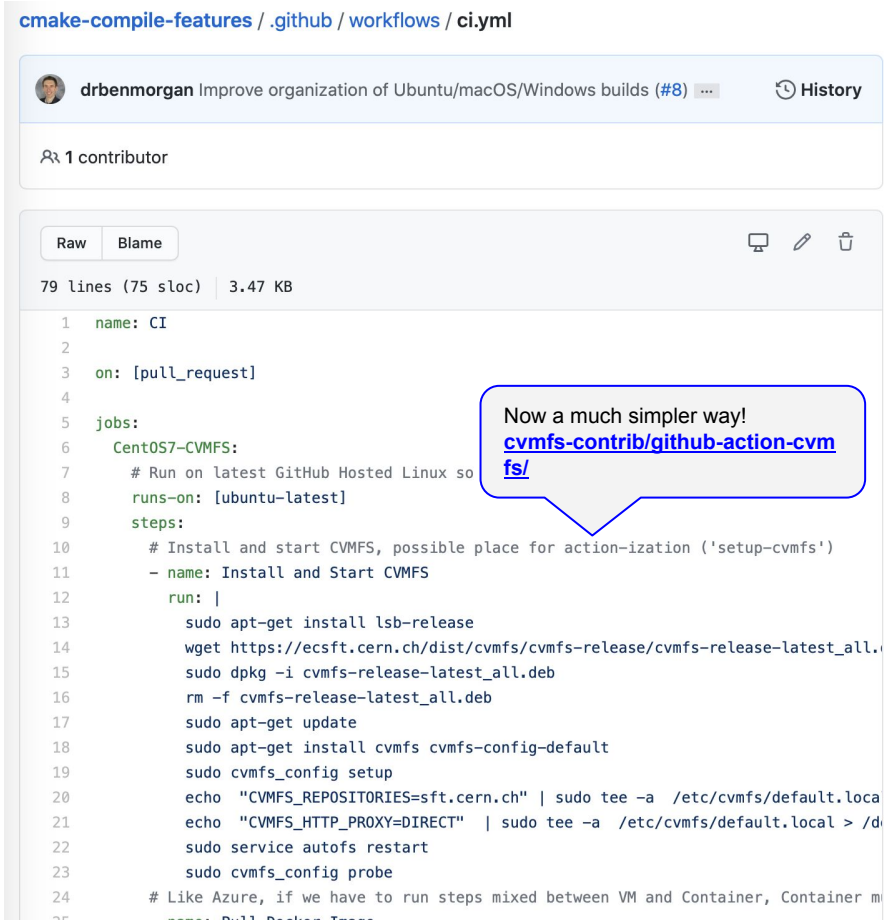
Organising Topic Branches

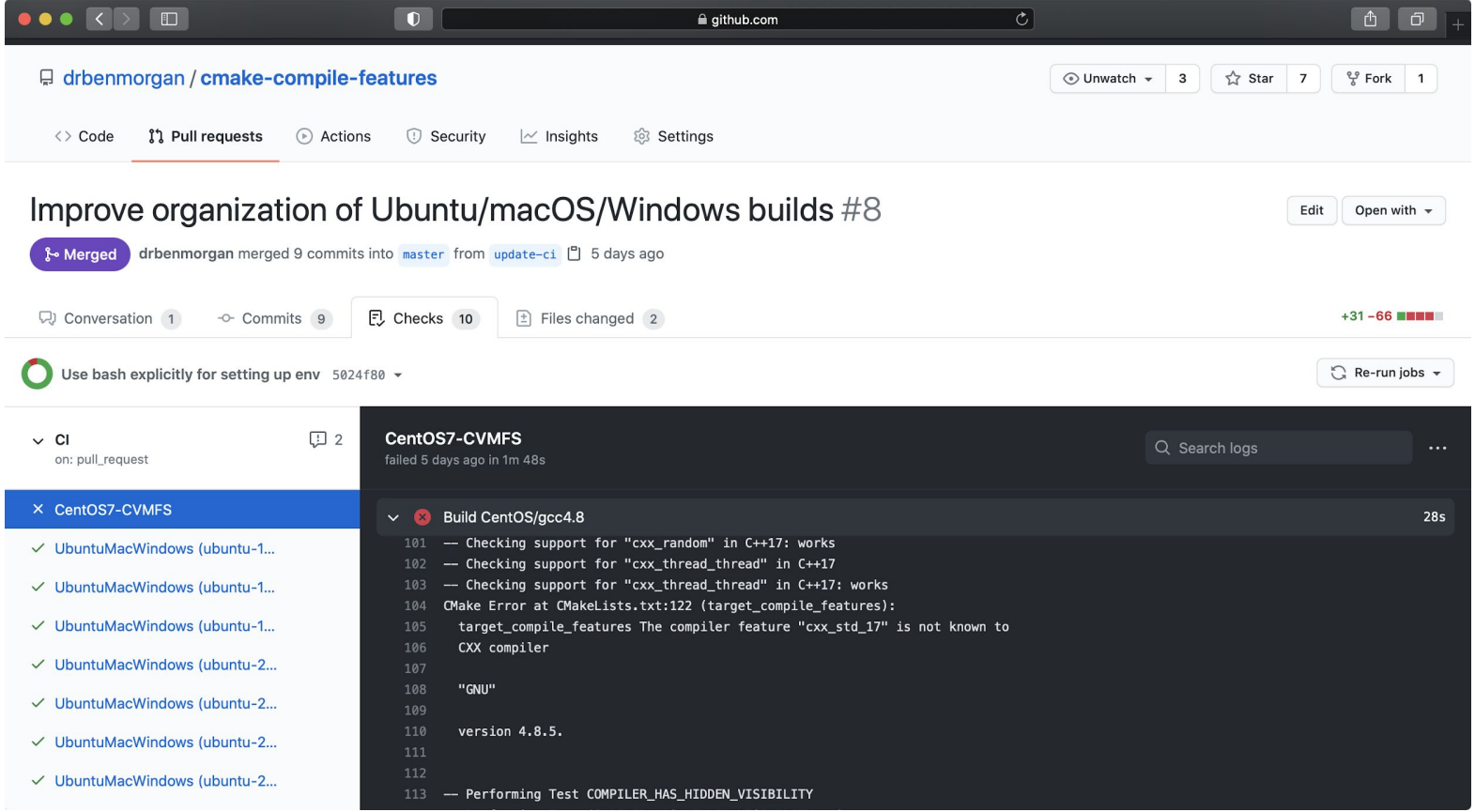
- **Forking Workflow** simplest and best
  - *Topic Branches on Forks*
  - ***Project repo only for merging***
- Tempting to push Topic Branches to project repo itself
  - *Fine for 1-3 developers …*
  - *… but does not scale well*
  - *Forks **don't** stop co-development*
- **Even lead developers should submit work from Forks!**
  - *Help separate development work from release/maintenance tasks*



Testing Pull Requests

- **Always build and test each PR using a CI system**
  - *E.g. GitHub Actions, Azure*
- **A critical time saver,** even if only targeting a single platform/OS
  - *Will quickly reveal any coding oversights or "works on my machine" gremlins*
- This is where earlier work on build and test systems starts to return the time investment!





GitHub Actions in Action

Real value is the record of build/test results, so PR author can find/fix issues quickly

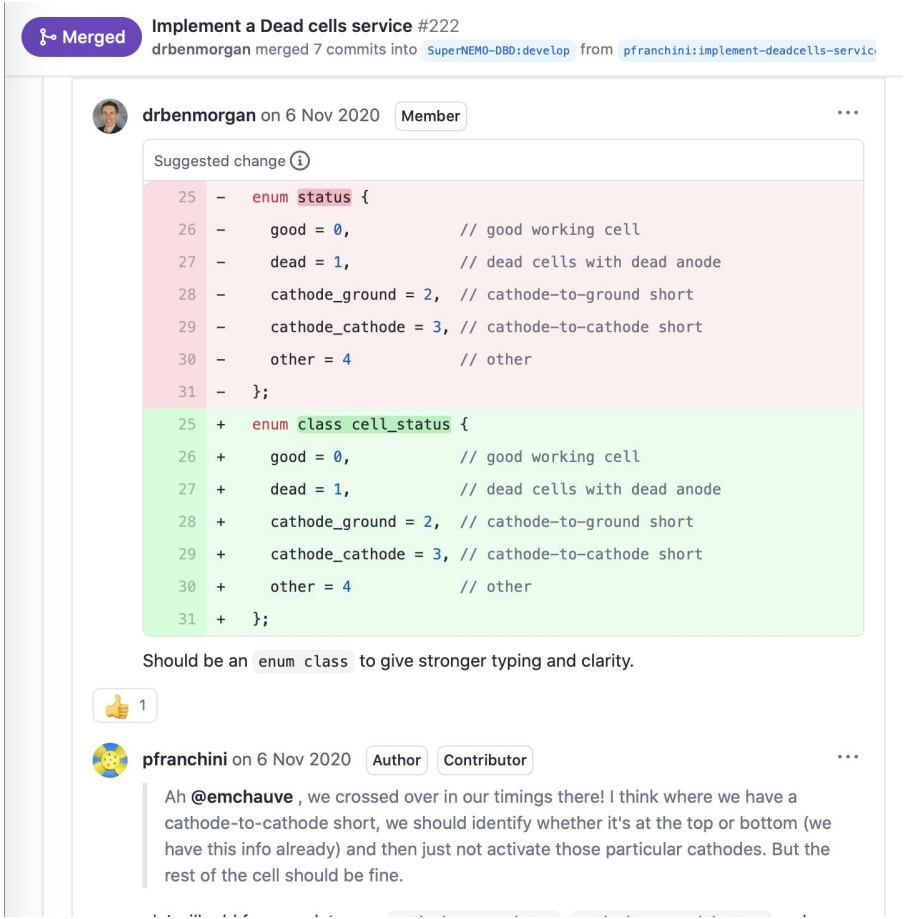
CI Scripting

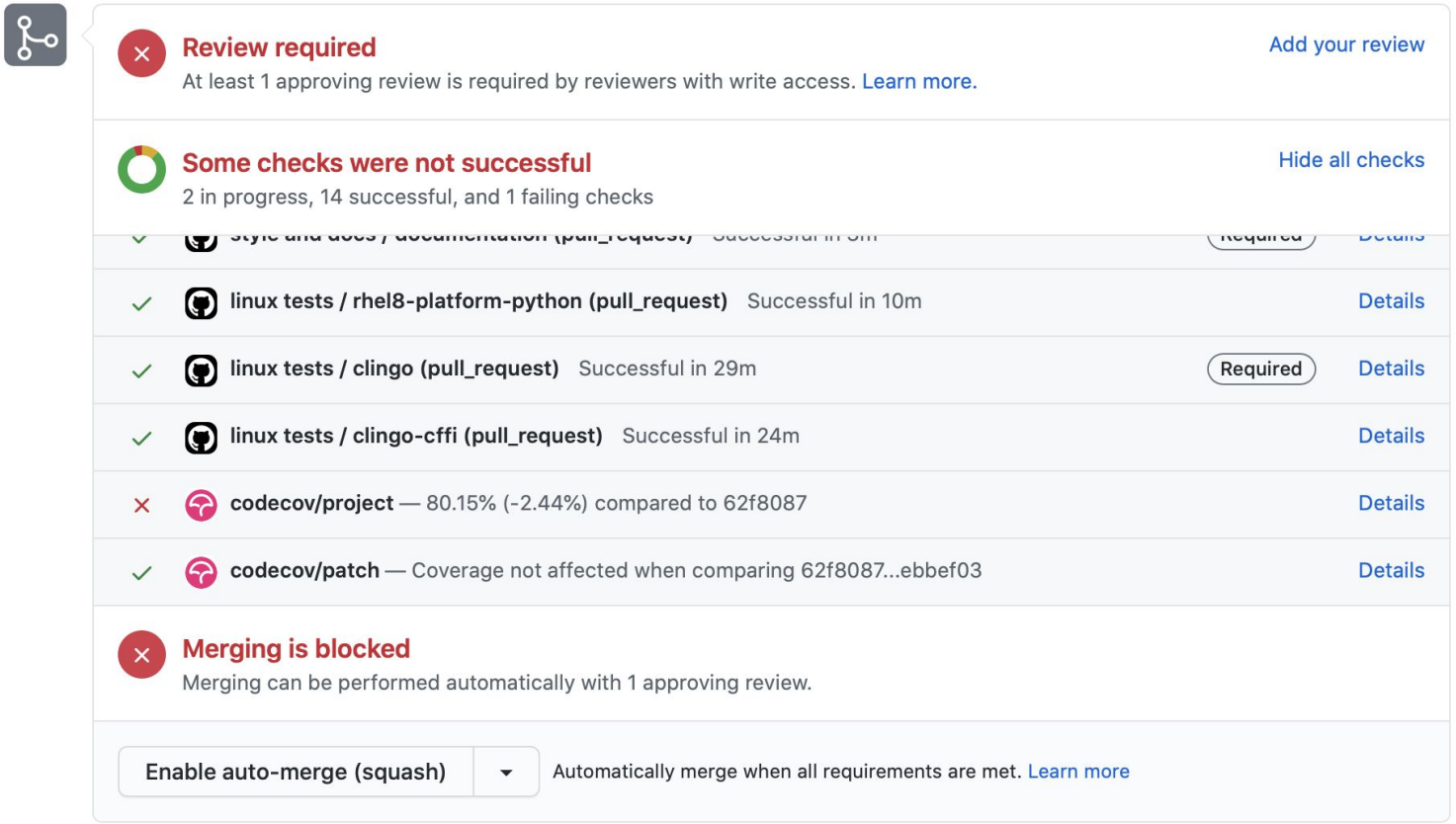
- **Like previous recommendations, don't overthink/engineer these**
  - *Each system does have slight differences in functionality*
- Even complex builds don't require complex scripting
- They are part of the project code, so develop and maintain them with the same care
  - *Including submitting changes through PRs - the CI can test CI!*

```yaml
  UbuntuMacWindows:
    strategy:
      fail-fast: false
      matrix:
        # As the matrix is sparse, write explicitly rather than x-product and excludes
        # Limit compilers AFAP to packages installable directly (no ppa)
        include:
          - { os: ubuntu-18.04, cc: gcc-7, cxx: g++-7 }
          - { os: ubuntu-18.04, cc: gcc-8, cxx: g++-8 }
          - { os: ubuntu-18.04, cc: clang-8, cxx: clang++-8 }
          - { os: ubuntu-20.04, cc: clang-9, cxx: clang++-9 }
          - { os: ubuntu-20.04, cc: gcc-9, cxx: g++-9 }
          - { os: ubuntu-20.04, cc: gcc-10, cxx: g++-10 }
          - { os: ubuntu-20.04, cc: clang-10, cxx: clang++-10 }
          - { os: macos-10.15 }
          - { os: windows-latest }
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v2
      - name: Setup build environment
        # Set CC/CXX only if they exist
        # See https://github.community/t/possible-to-use-conditional-in-the-env-section
        # for why this can be done in env: above
        shell: bash
        run: |
          [[ ! -z "${{ matrix.cc }}" ]] && echo "CC=${{ matrix.cc }}" >> $GITHUB_ENV ||
          [[ ! -z "${{ matrix.cxx }}" ]] && echo "CXX=${{ matrix.cxx }}" >> $GITHUB_ENV
      - name: Configure
        run: |
          cmake -S. -B ./build
      - name: Build
        run: |
          cmake --build ./build --config RelWithDebInfo
```

Reviewing Pull Requests

- **Use [PR reviews] as "peer review for code"**
  - *Discuss the implementation, suggest revisions, improvements*
  - *Ask for addition of missing items like tests, documentation!*
- Can be light touch "fine for me", to major revisions, even rejection
- **Collaborative approach is valuable in increasing the Bus Factor and reducing over-engineering**



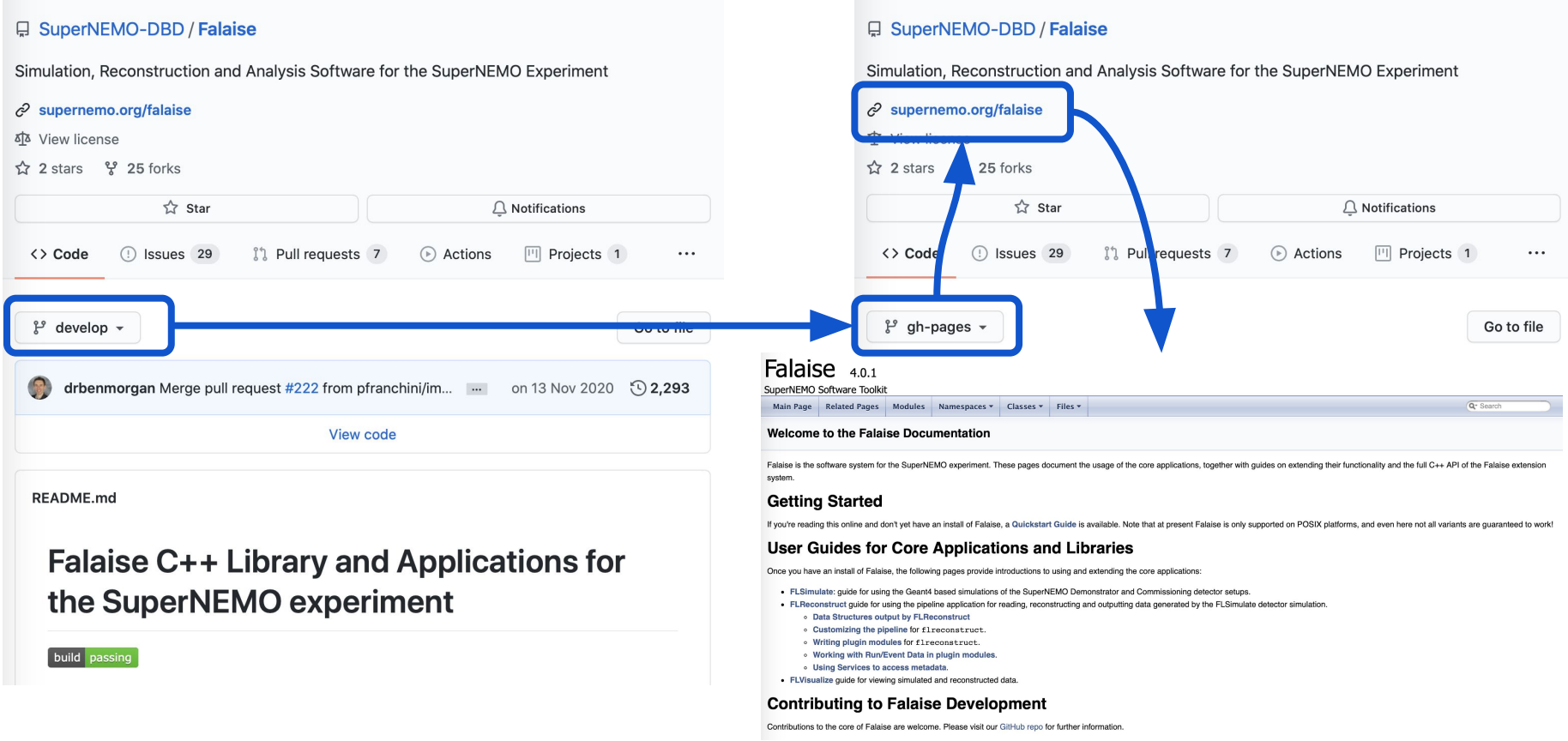


Merging Pull Requests

Use Actions and Reviews to build a “pre merge” checklist

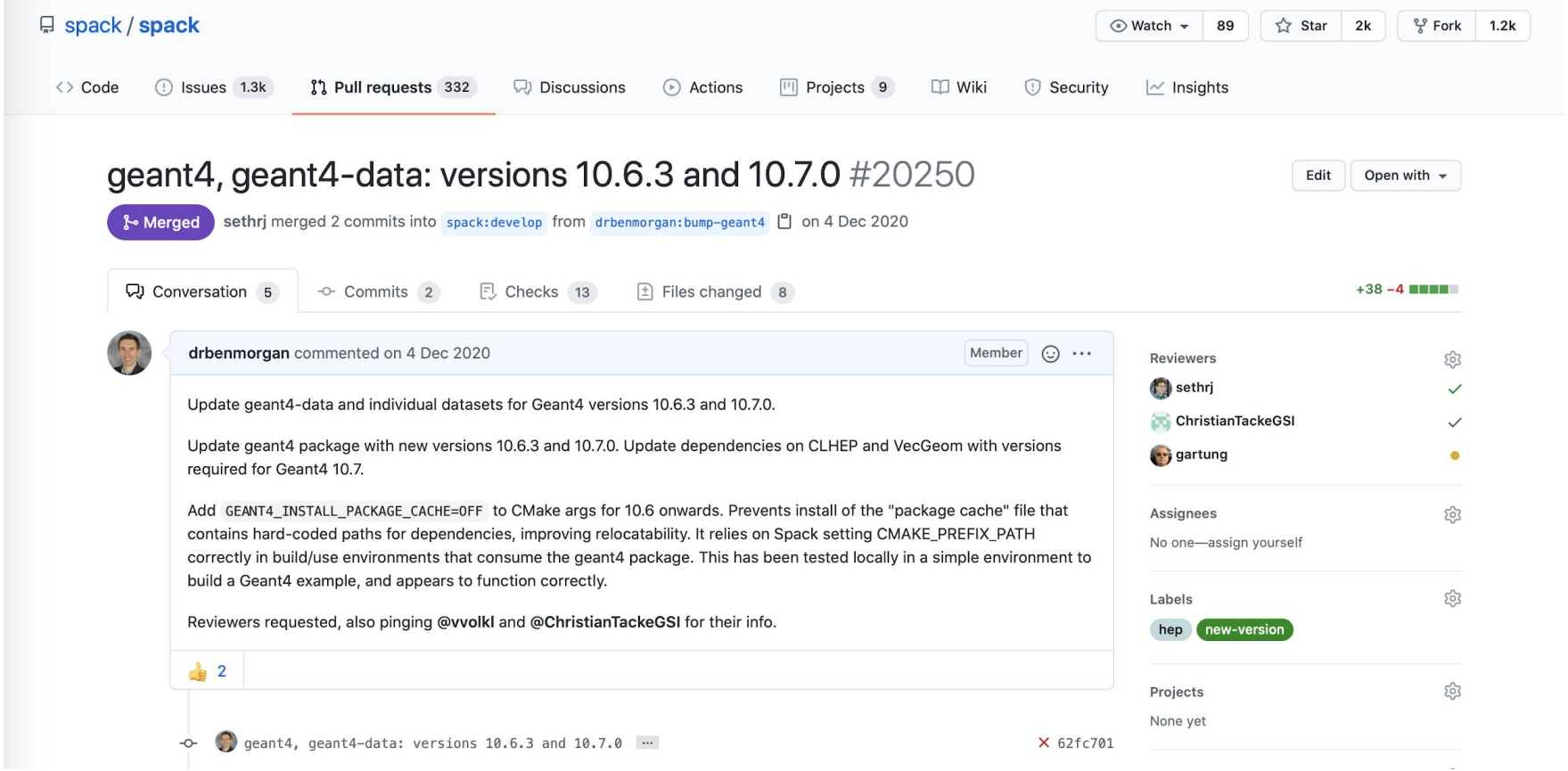
Release & Deployment

- By testing/reviewing PRs before merging to the `main` branch, **each PR merge provides a potential new release for the project**
  - *When/how often to make a release, and whether detailed testing (e.g. physics validation) is required before tagging is highly project dependent*
- **For each merge to `main` and new tag (release) of the project, a CI task should be triggered to:**
  - *Create and publish the documentation for the project*
  - *Create package(s)/environment for the project, e.g. conda, Docker, CVMFS*
- Last item invaluable if you need to run large validation jobs for releases!
- Might require extra resources (e.g. ReadTheDocs, DockerHub), **but earlier investment in standard tools simplifies integration!**
  - *And GitHub/Lab can often handle this for you (gh-pages, Packages)*



Publishing Documentation

An easy way to do this is through [GitHub Pages]



Example Packaging: Spack

Especially valuable for community wide projects to have a presence in mainline package managers

Finally: Don’t neglect Training until the last slide!

- “Documentation” should really be “Documentation **and Training**”!
  - ***Projects should point to existing Software Carpentry (and HSF!) lessons and others (e.g. GitHub Help) for the basics rather than write their own***
  - *Software Carpentry template good starting point for developing material for your project*
  - *Hands-on tutorials at, e.g. experiment collaboration meetings, can be valuable, though **you may have to fight for a slot**!*
  - *Link up remote/in-person material as later opportunities can be limited*
- Developers should **also** actively seek training/knowledge opportunities
  - *Engage with community meetings/courses like today*
  - *Institute/National **Research Software Engineer** groups/organizations*
  - *Conferences like **CHEP**, **SC**, though getting funding can be difficult*

In Summary

- Primary challenge for small software projects is **developer FTE**, so this must be used **effectively**
- **Invest time in simple and standard tools** and methods to reduce developer overhead and points of failure, and increase automation
- **Use GitHub/Lab's tools** to provide a simple yet highly capable platform for development that returns investment in standard tools
- ***In short****: Invest time in your tools, use them as designed, and let the computer take the strain!*