How you
Yes YOU!
Can become a JEDI Developer too

GitHub, Git-flow, documentation,
pull requests, code reviews…

Outline

I) The way of a JEDI

- ✦ Agile project management
- ✦ git and GitHub
- ✦ git-flow

II) Preparing to contribute

- ✦ Work from a fork
- ✦ Make sure your branch is up to date with develop
- ✦ Make sure your code is adequately tested
- ✦ Make sure your code is adequately documented

III) Contributing code

- ✦ Pull requests
- ✦ Code Reviews

The Way of a JEDI

- *Collaborative*
  - *A Joint Center (JCSDA)*
    - *Partners, collaborators, stakeholders, community*
  - *A Joint Effort (JEDI)*
    - *Distributed team of software developers, with varying objectives and time commitments*
- *Agile*
  - *Innovative*
  - *Flexible (future-proof)*
  - *Responsive to users and developers*
  - *Continuous delivery of functional software*

Agile Software Development

- ***12 Agile Principles***

Early and continuous delivery of valuable software

1

Welcome changing requirements even late in development

2

Deliver working software frequently

3

Business people and developers working together daily

4

Build projects around motivated individuals and trust them to get the job done

5

The most effective method of conveying information is face-to-face conversation

6

Working software is the primary measure of progress

7

Sustainable development: maintain a constant pace indefinitely

8

Continuous attention to technical excellence

9

Simplicity: maximize the amount of work not done

10

Teams self-organize

11

Teams regularly reflect and adjust behaviour

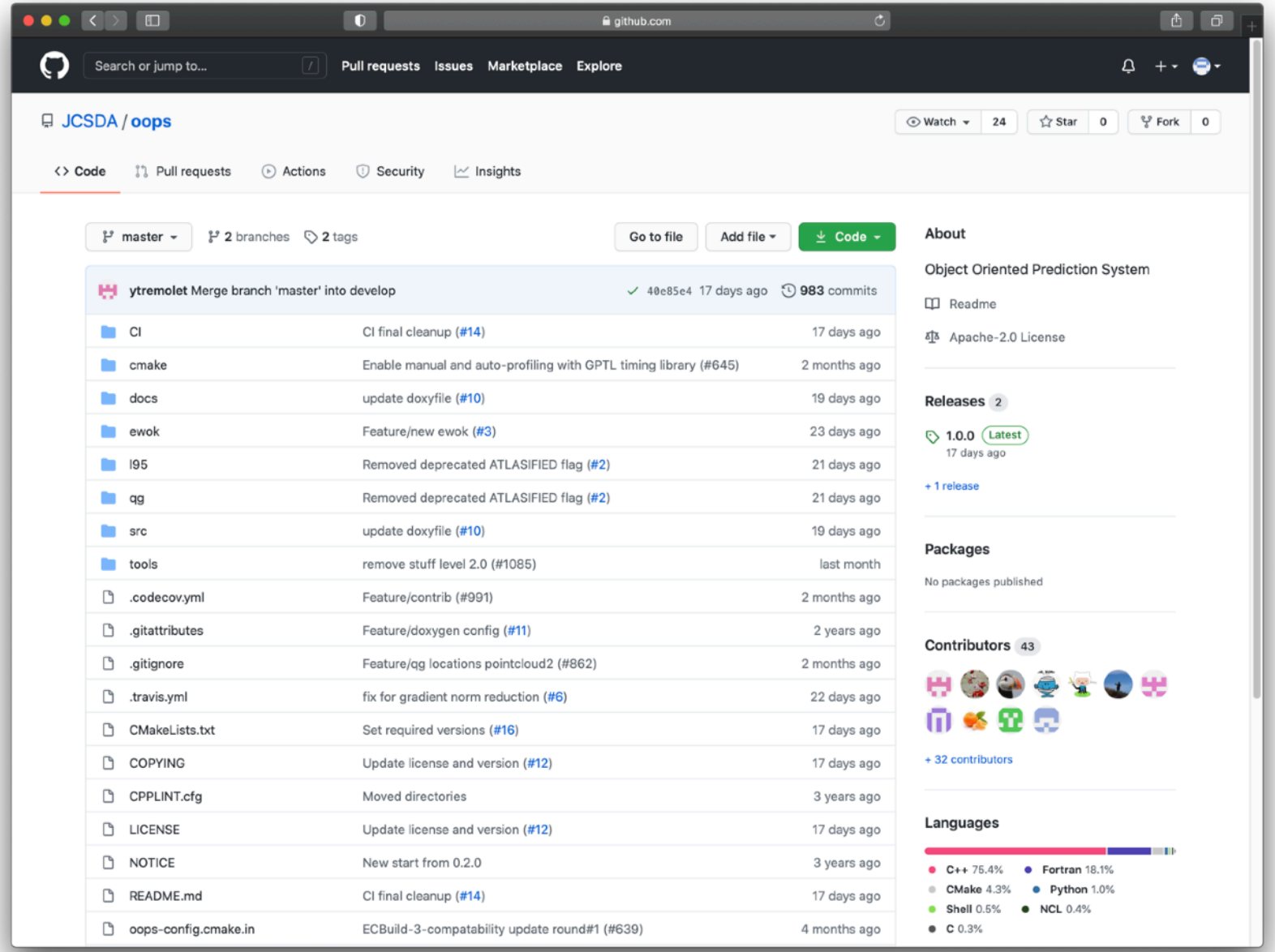
12

https://nomad8.com/

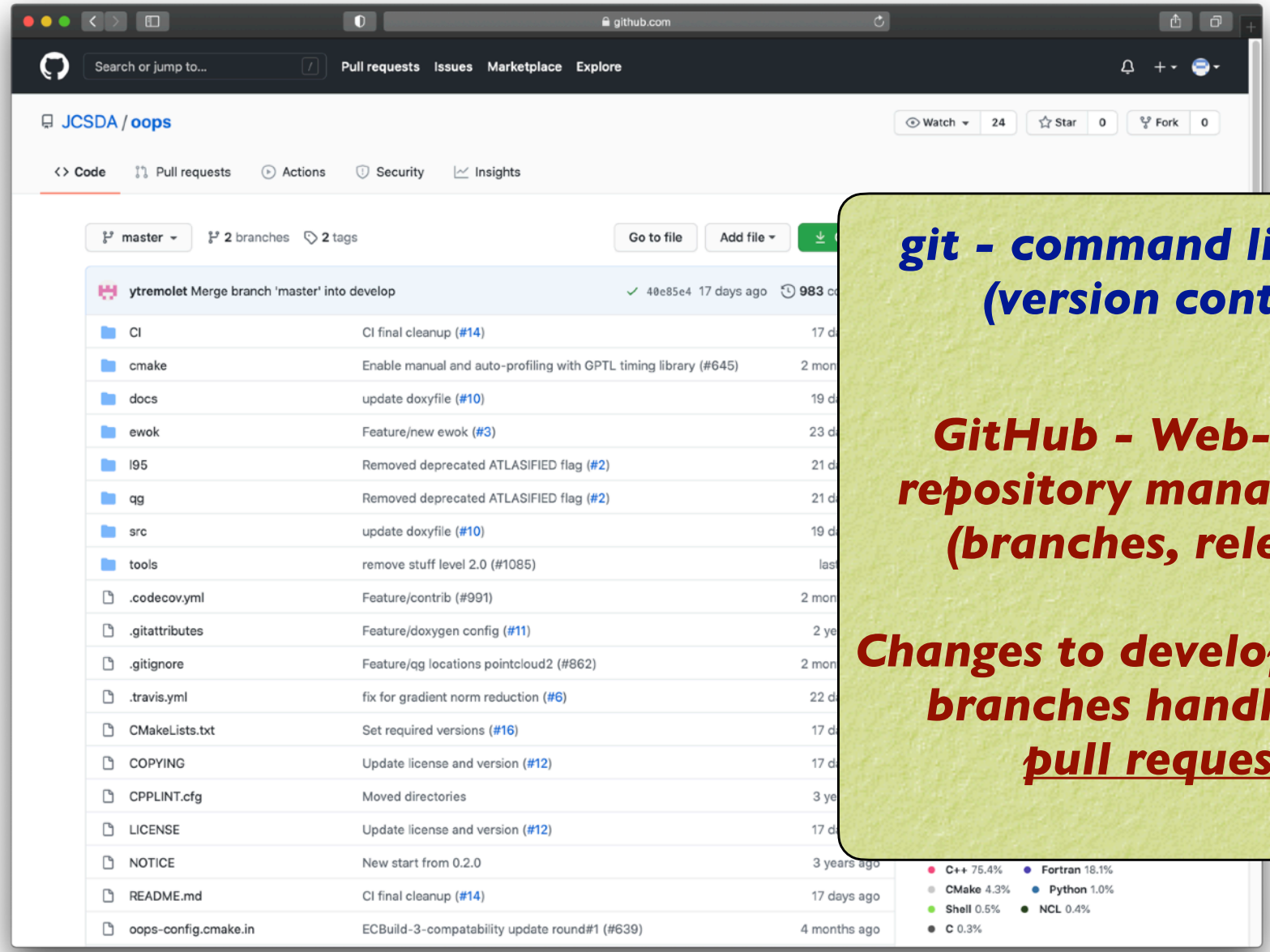
Agile Tools

- *git/GitHub*
  - *Version control and Release distribution*
  - *Pull requests, Code reviews*
  - *Coordination of distributed community of developers*
- *Git-Flow*
  - *Innovation*
  - *Continuous Delivery*
- *ZenHub*
  - *Agile project management*
  - *Issue tracking, enhanced code review*
- *Forums: https://forums.jcsda.org*
  - *User support, stakeholder feedback*

git/GitHub


github.com
Search or jump to...
Pull requests
Issues
Marketplace
Explore
JCSDA / oops
Watch 24
Star 0
Fork 0
Code
Pull requests
Actions
Security
Insights
master
2 branches
2 tags
Go to file
Add file
Code
ytremolet Merge branch 'master' into develop
40e85e4 17 days ago
983 commits
CI CI final cleanup (#14) 17 days ago
cmake Enable manual and auto-profiling with GPTL timing library (#645) 2 months ago
docs update doxyfile (#10) 19 days ago
ewok Feature/new ewok (#3) 23 days ago
l95 Removed deprecated ATLASIFIED flag (#2) 21 days ago
qg Removed deprecated ATLASIFIED flag (#2) 21 days ago
src update doxyfile (#10) 19 days ago
tools remove stuff level 2.0 (#1085) last month
.codecov.yml Feature/contrib (#991) 2 months ago
.gitattributes Feature/doxygen config (#11) 2 years ago
.gitignore Feature/qg locations pointcloud2 (#862) 2 months ago
.travis.yml fix for gradient norm reduction (#6) 22 days ago
CMakeLists.txt Set required versions (#16) 17 days ago
COPYING Update license and version (#12) 17 days ago
CPPLINT.cfg Moved directories 3 years ago
LICENSE Update license and version (#12) 17 days ago
NOTICE New start from 0.2.0 3 years ago
README.md CI final cleanup (#14) 17 days ago
oops-config.cmake.in ECBuild-3-compatability update round#1 (#639) 4 months ago
About
Object Oriented Prediction System
Readme
Apache-2.0 License
Releases 2
1.0.0 Latest
17 days ago
+ 1 release
Packages
No packages published
Contributors 43
+ 32 contributors
Languages
C++ 75.4%
Fortran 18.1%
CMake 4.3%
Python 1.0%
Shell 0.5%
NCL 0.4%
C 0.3%

git/GitHub



git - command line tool (version control)

GitHub - Web-based repository management (branches, releases)

Changes to develop, master branches handled via pull requests

Git-Flow

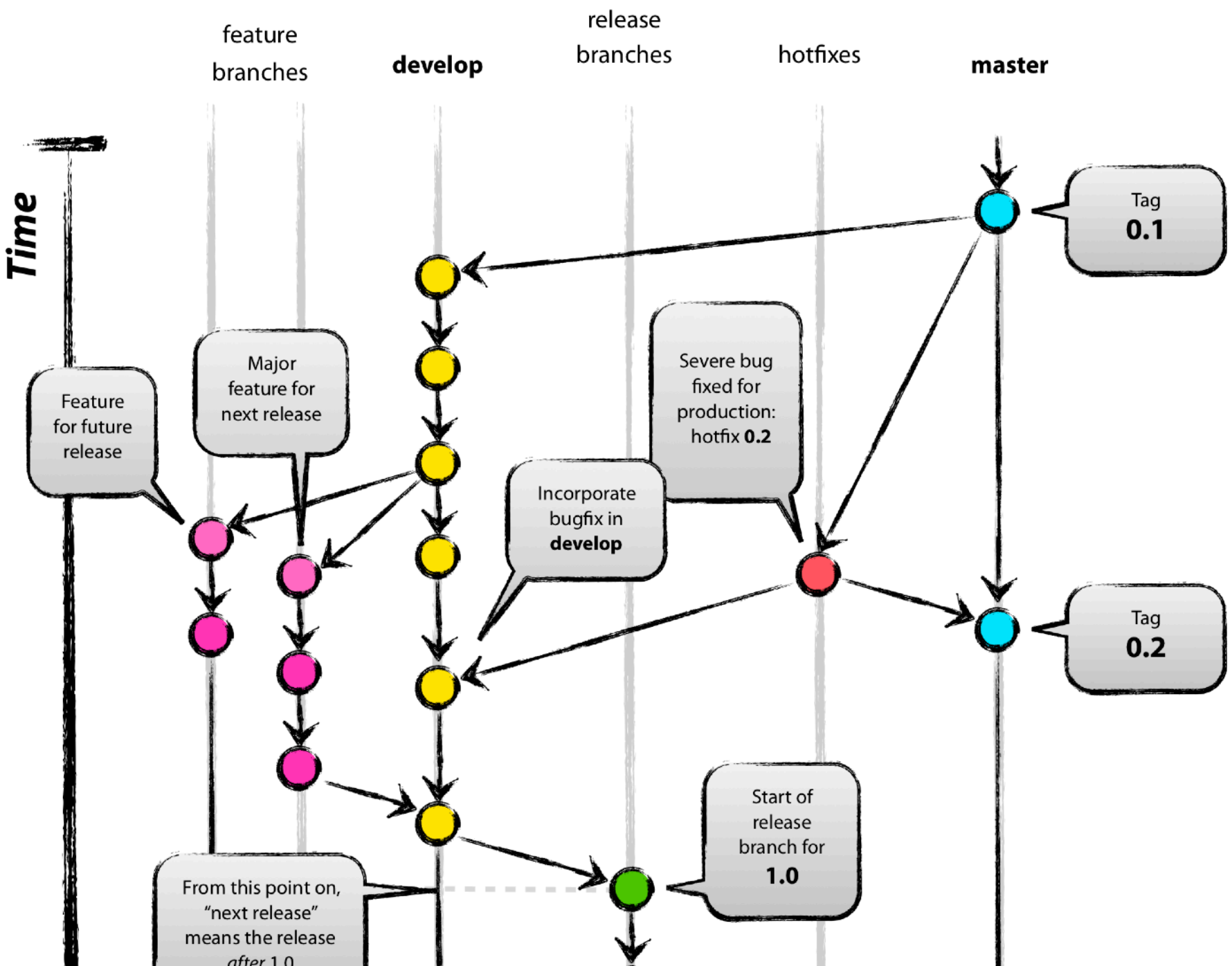
Git Flow is:

- ***A Philosophy***
  - ✦ ***Optimal for Agile Software Development***
    - ***Innovation***
    - ***Continuous Delivery***
- ***A Working Principle***
  - ✦ ***Enforcement of branch naming conventions***
- ***An Application (extension to git)***
  - ✦ ***Already installed Singularity Container***

Vincent Driessen (2010)

Git-flow manifesto

http://nvie.com/posts/a-successful-git-branching-model/


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 0.2
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after 1.0

The Git-Flow Manifesto: Takaways

- *master is for releases only*
- *develop*
  - *Not ready for public consumption but compiles and passes all tests*
- *Feature branches*
  - *Where most development happens*
  - *Branch off of develop*
  - *Merge into develop*
- *Release branches*
  - *Branch off of develop*
  - *Merge into master and develop*
- *Hotfix*
  - *Branch off of master*
  - *Merge into master and develop*
- *Bugfix*
  - *Branch off of develop*
  - *Merge into develop*

Feature branches should be focused and short, with a specific goal

They should exists for days or weeks, not months

I) The way of a JEDI

- ✦ Agile project management
- ✦ git and GitHub
- ✦ git-flow

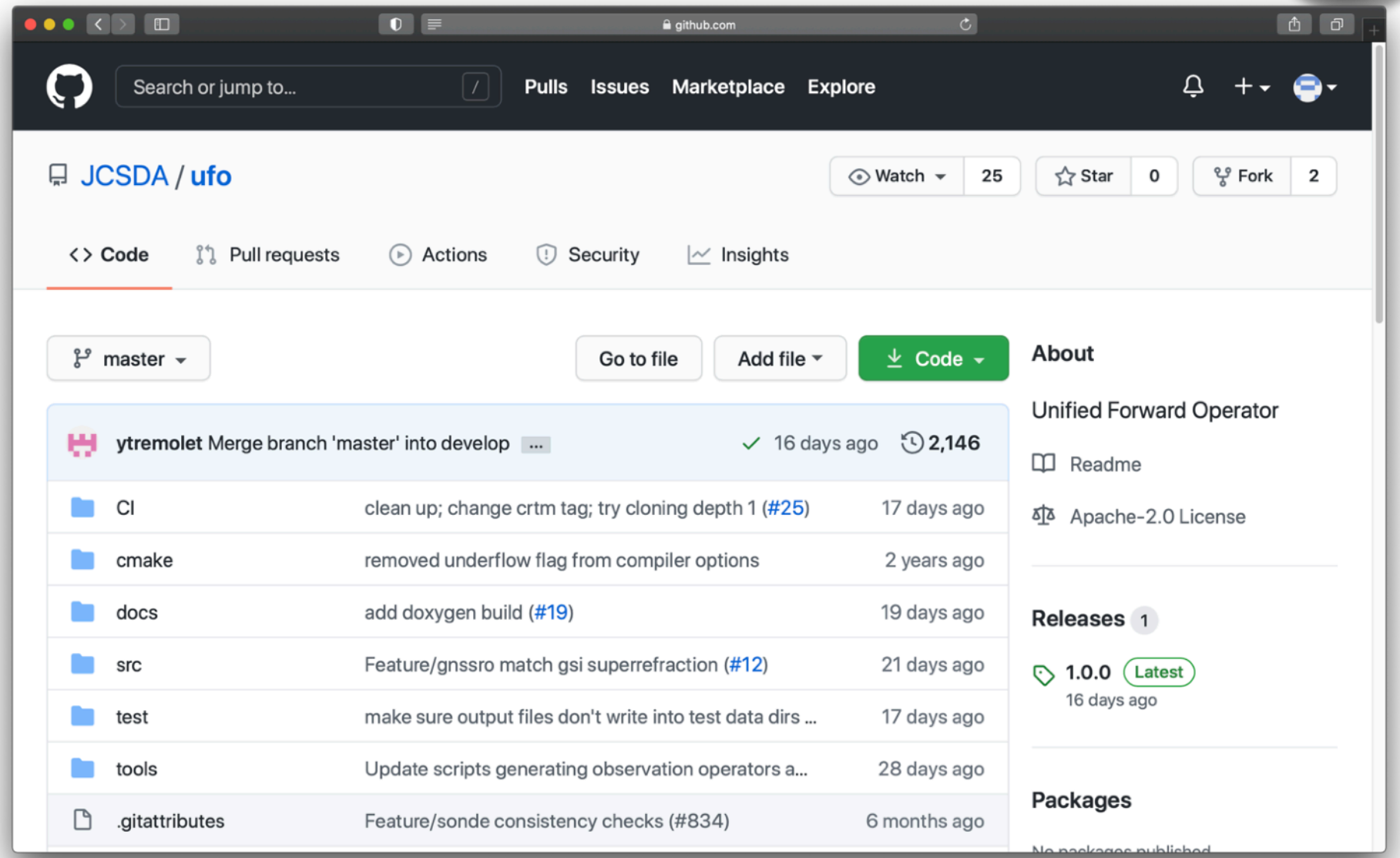
II) Preparing to contribute

- ✦ **Work from a fork**
- ✦ **Make sure your branch is up to date with develop**
- ✦ **Make sure your code is adequately tested**
- ✦ **Make sure your code is adequately documented**

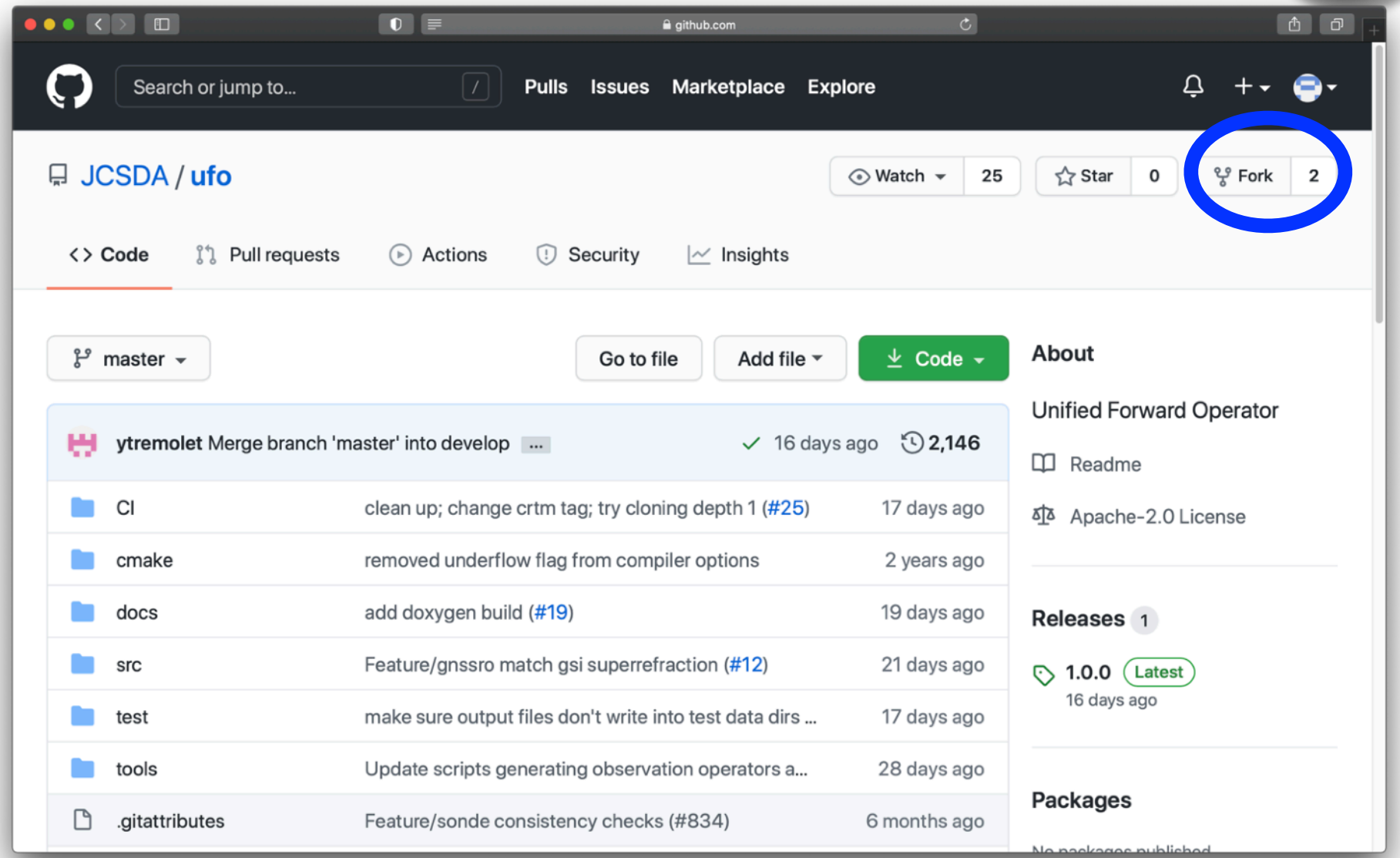
III) Contributing code

- ✦ Pull requests
- ✦ Code Reviews

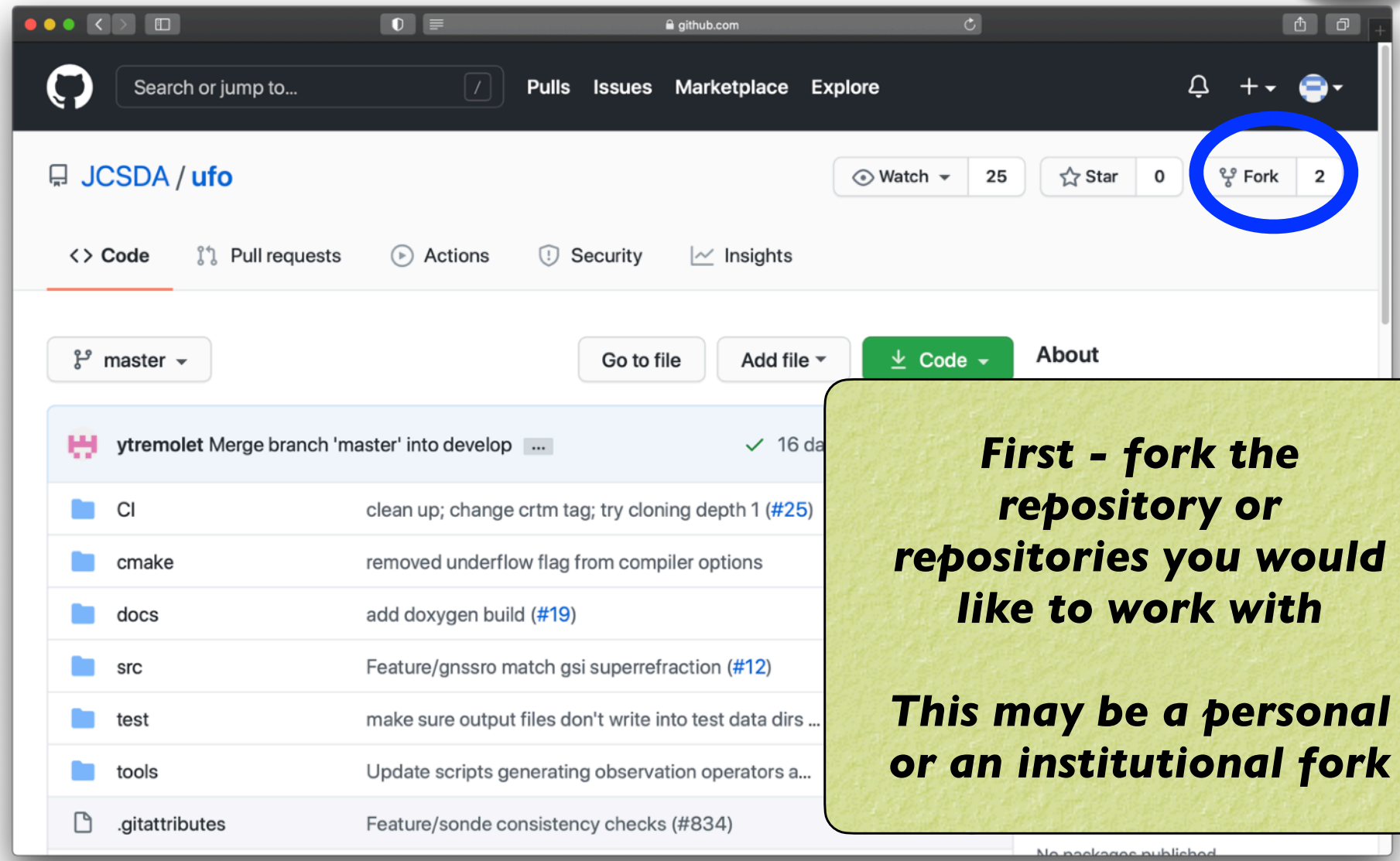
Part II: Preparing to contribute



Part II: Preparing to contribute



Part II: Preparing to contribute



First - fork the repository or repositories you would like to work with

This may be a personal or an institutional fork

Create a feature branch

Set up JCSDA as the develop branch

```
git clone https://github.com/<myaccount>/ufo.git
cd ufo
git remote add upstream https://github.com/JCSDA/ufo.git
git fetch --tags upstream
git checkout --track upstream/develop
```

Create feature branch from JCSDA develop

```
git checkout -b feature/<mybranch> develop
```

Implement code changes

Edit the code in the feature branch, commit changes, and push it to your fork

```
git add *
git status
git commit
git push origin --set-upstream feature/<mybranch>
```

Make sure you're committing the files you intend to commit

Continue to make changes, commit them, test them, and push to your fork. Periodically synchronize with JCSDA develop and resolve any merge conflicts that may arise

```
git checkout develop
git pull upstream develop
git checkout feature/<mybranch>
git merge develop
```

Add Tests and Documentation

Be sure to add tests that execute the code you added or modified
(For instructions, see Maryam's lecture)
If you do not, then your code will not pass our CI (CodeCov) testing and it will not be merged

Also add documentation explaining the purpose of the code, what it does, how to use it, when to use it, scientific and/or mathematical background, and known limitations or bugs

- ***Doxygen***
  - ***Low-level descriptions of functions, classes, subroutines, etc, embedded directly in the code***
- ***Sphinx:*** http://jedi-docs.jcsda.org
  - ***Repository:*** https://github.com/JCSDA/jedi-docs.git
  - ***High-level documentation (context, use cases, theory…)***

Documenting Fortran Source Code

```
! ! ------------------------------------------------------------------------
!> \brief Example function
!!
!! \details **myfunction()** takes a and b as arguments and miraculously creates c.
!! I could add many more details here if I chose to do so.  I can even make a list:
!! * item 1
!! * item 2
!! * item 3
!!
!! \date A long, long, time ago: Created by L. Skywalker (JCSDA)
!!
!! \warning This isn't a real function!
!!
subroutine myfunction(a, b, c)
   integer, intent(in)              :: a !< this is one input parameter
   integer, intent(in)              :: b !< this is another
   real(kind=kind_rea), intent(out) :: c !< and this is the output
   [...]
```

Note

Doxygen has known problems with object-oriented Fortran and Fortran/C++ bindings

Documenting C++ Source Code

```
// ----------------------------------------------------------------------------
/*! \brief Example function
*
* \details **myfunction()** takes a and b as arguments and miraculously creates c.
* I could add many more details here if I chose to do so.  I can even make a list:
* * item 1
* * item 2
* * item 3
*
* \param[in] a this is one input parameter
* \param[in] b this is another
* \param[out] c and this is the output
*
* \date A long, long, time ago: Created by L. Skywalker (JCSDA)
*
* \warning This isn't a real function!
*
*/
void myfunction(int& a, int& b, double& c) {
   [...]
```

Useful Doxygen Commands

- \brief
- \details
- \param
- \return
- \author
- \date
- \note
- \attention
- \warning
- \bug
- \class <name> [<header-file>]
- \mainpage
- \f$ … \f$ (inline formula)
- \f[… \f] (formula block)
- \em (or * … *)
- \sa (see also)
- \typedef
- \todo
- \version
- \namespace
- […](…) (url)
- \image
- \var
- \throws (exception description)

Many more described here:

https://www.stack.nl/~dimitri/doxygen/manual/commands.html

Sample output: “man page”

◆ testStateInterpolation()

template<typename MODEL >

void test::testStateInterpolation ( )

Interpolation test.

testStateInterpolation() tests the interpolation for a given model. The conceptual steps are as follows:

1. Initialize the JEDI **State** object based on idealized analytic formulae
2. Interpolate the **State** variables onto selected "observation" locations using the getValues() method of the **State** object. The result is placed in a JEDI **GeoVaLs** object
3. Compute the correct solution by applying the analytic formulae directly at the observation locations.
4. Assess the accuracy of the interpolation by comparing the interpolated values from Step 2 with the exact values from Step 3

The interpolated state values are compared to the analytic solution for a series of **locations** which includes values optionally specified by the user in the "StateTest" section of the config f a randomly-generated list of **Nrandom** random locations. Nrandom is also specified by the user in the "StateTest" section of the config file, as is the (nondimensional) tolerence level (**inte** to be used for the tests.

This is an equation:

$$\zeta = \left(\frac{x - x_0}{\lambda} \right)^{2/3}$$

Relevant parameters in the **State* section of the config file include

- **norm-gen** Normalization test for the generated **State**
- **interp_tolerance** tolerance for the interpolation test

Date

April, 2018: M. Miesch (JCSDA) adapted a preliminary version in the feature/interp branch

Warning

Since this model compares the interpolated state values to an exact analytic solution, it requires that the "analytic_init" option be implemented in the model and selected in the "State.StateGenerate" section of the config file.

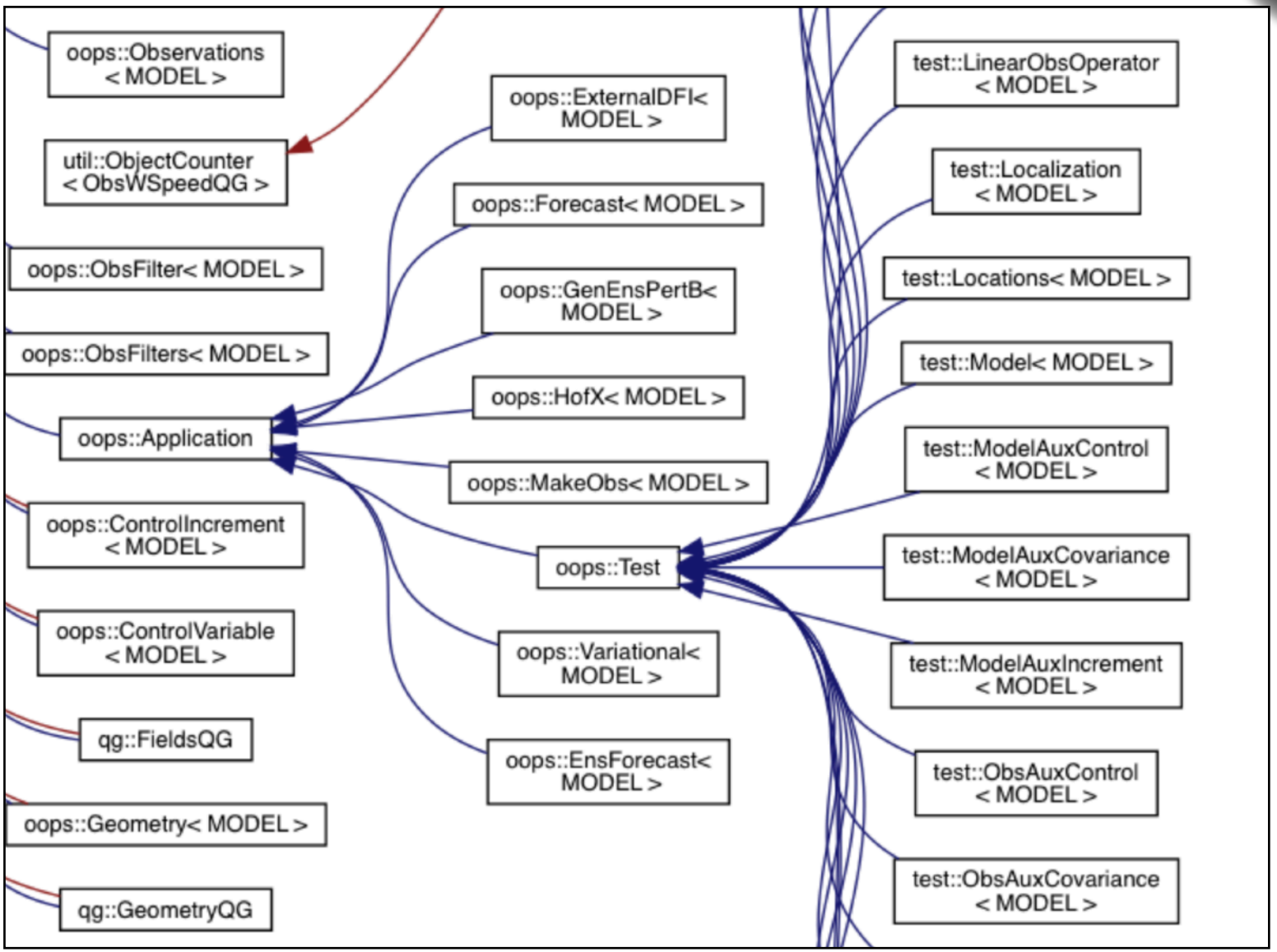
Corresponding code

```
// -----------------------------------------------------------------------------
/*! \brief Interpolation test
 *
 * \details **testStateInterpolation()** tests the interpolation for a given
 * model.  The conceptual steps are as follows:
 * 1. Initialize the JEDI State object based on idealized analytic formulae
 * 2. Interpolate the State variables onto selected "observation" locations
 *    using the getValues() method of the State object.  The result is
 *    placed in a JEDI GeoVaLs object
 * 3. Compute the correct solution by applying the analytic formulae directly
 *    at the observation locations.
 * 4. Assess the accuracy of the interpolation by comparing the interpolated
 *    values from Step 2 with the exact values from Step 3
 *
 * The interpolated state values are compared to the analytic solution for
 * a series of **locations** which includes values optionally specified by the
 * user in the "StateTest" section of the config file in addition to a
 * randomly-generated list of **Nrandom** random locations.  Nrandom is also
 * specified by the user in the "StateTest" section of the config file, as is the
 * (nondimensional) tolerence level (**interp_tolerance**) to be used for the tests.
[…]
```

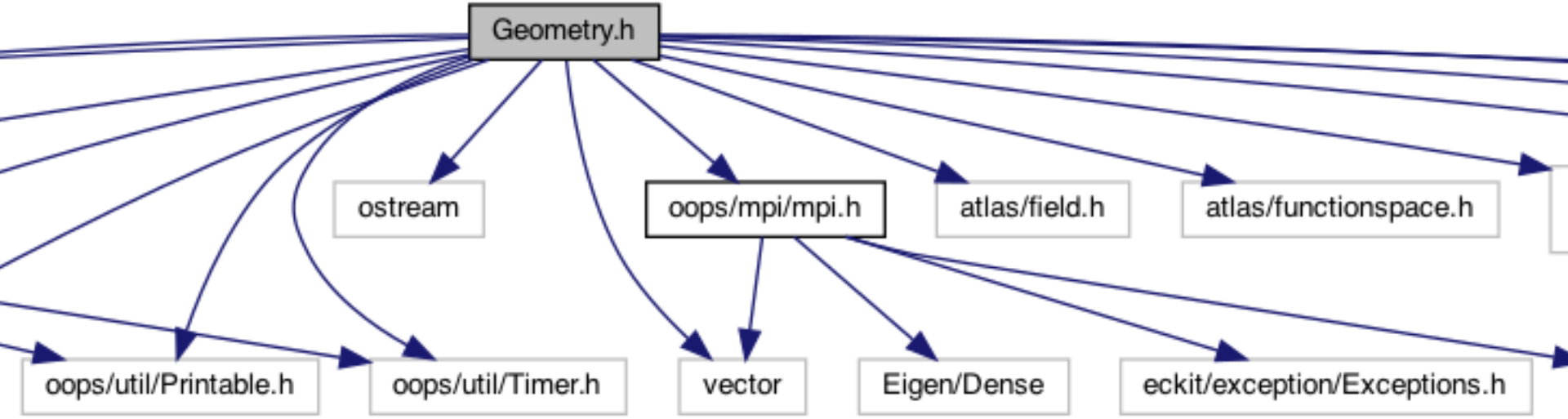
Corresponding code (cont.)

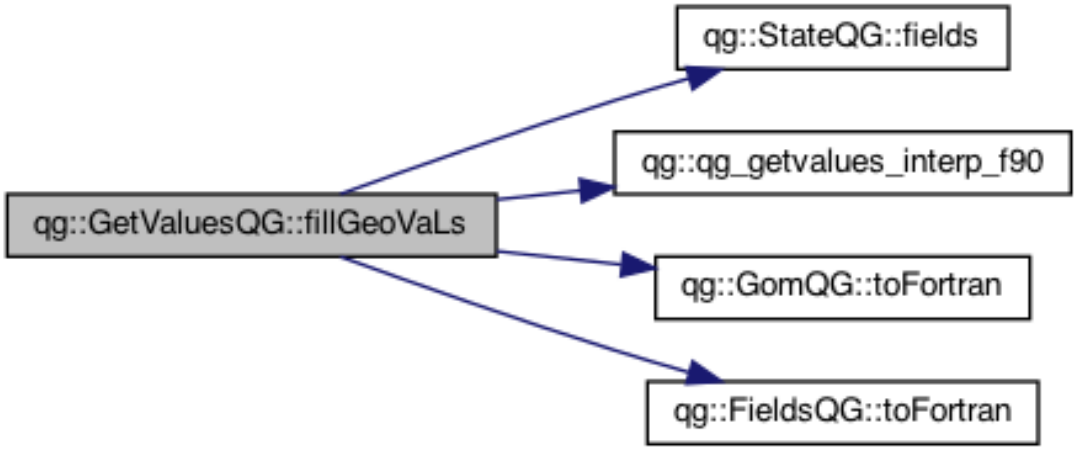
```
[…]
 *
 * This is an equation:
 * \f[ \zeta = \left(\frac{x-x_0}{\lambda}\right)^{2/3} \f]
 *
 * Relevant parameters in the **State* section of the config file include
 *
 * * **norm-gen** Normalization test for the generated State
 * * **interp_tolerance** tolerance for the interpolation test
 *
 * \date April, 2018: M. Miesch (JCSDA) adapted a preliminary version in the
 * feature/interp branch
 *
 * \warning Since this model compares the interpolated state values to an exact analytic
 * solution, it requires that the "analytic_init" option be implemented in the model and
 * selected in the "State.StateGenerate" section of the config file.
 */
```

Sample output: class hierarchy



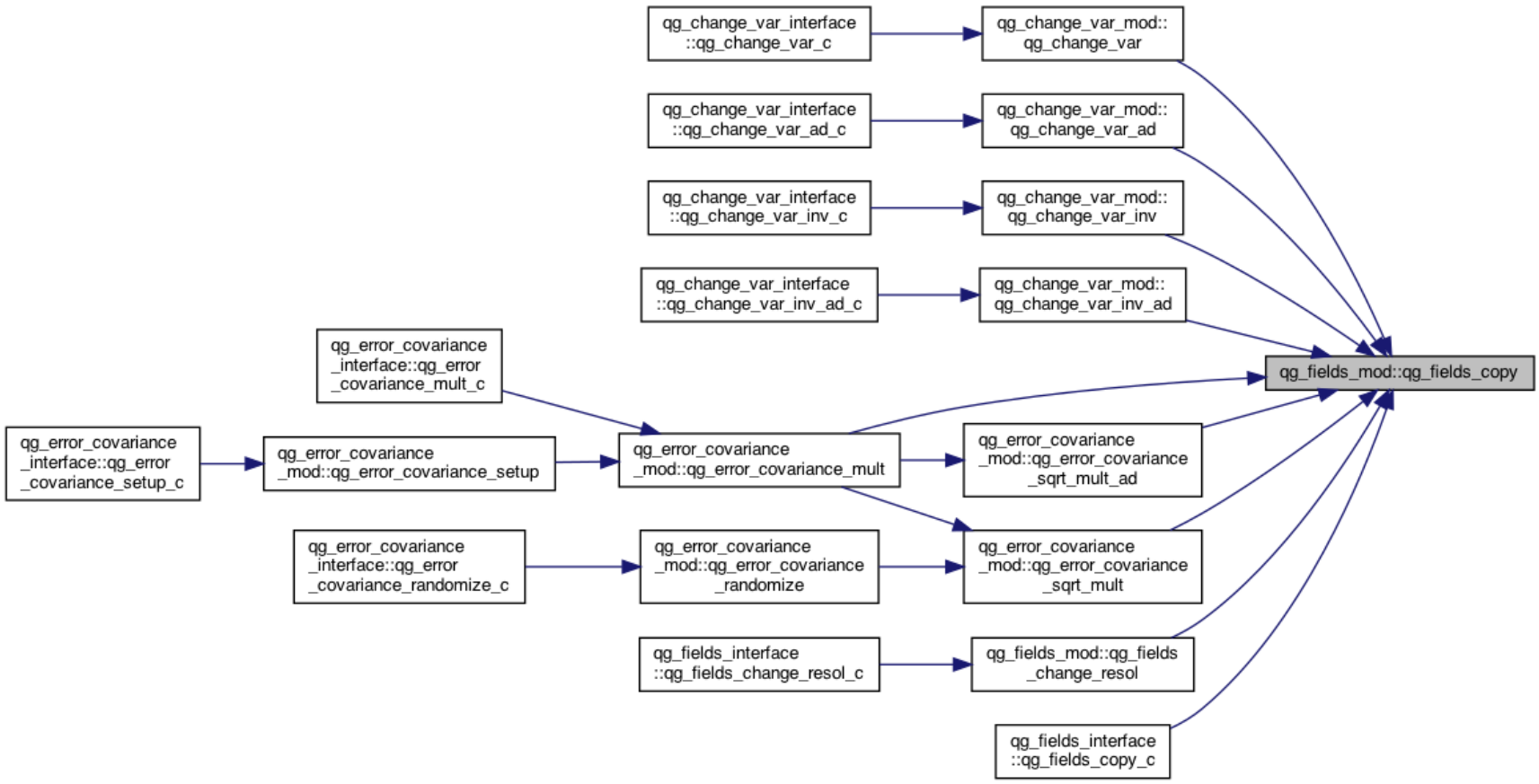
Sample output: include, call graphs





Clickable boxes!

Sample output: caller graphs



Note that these traces end in _c (this is a Fortran routine)
Doxygen has trouble with C++ / Fortran binding
Look for corresponding _f90 routine to follow further

Doxygen in JEDI

After you have added doxygen documentation to the source code, you can generate html doxygen output for a particular repository by enabling the documentation with ecbuild.

Be sure you have doxygen and graphviz installed (can install with homebrew, apt, yum, etc)

```
ecbuild -DENABLE_OOPS_DOC=ON ../fv3-bundle
make -j4
```

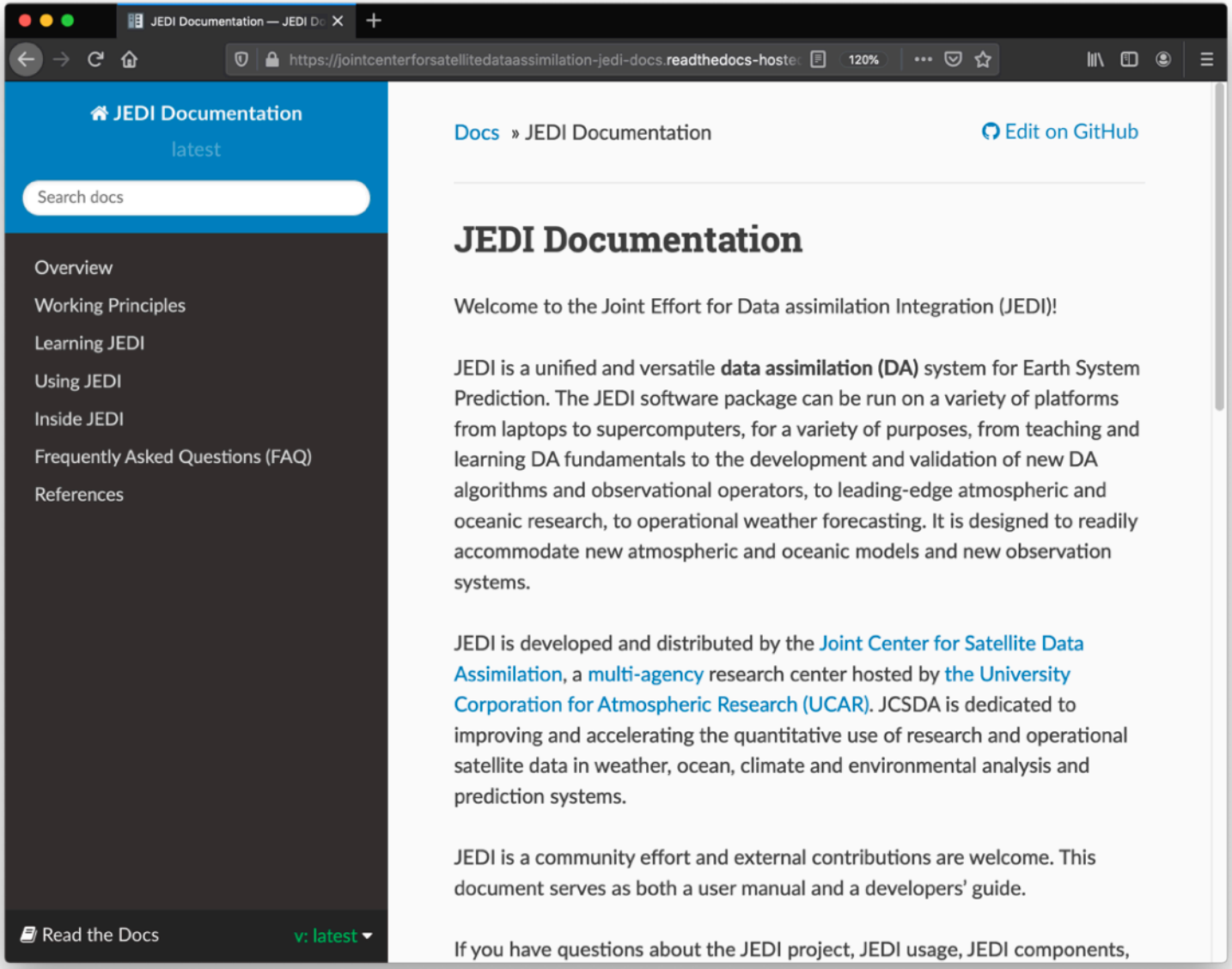
You can find the results in the <build>/<repo>/docs/html directory

Doxygen documentation for JEDI components is available on the academy and JEDI documentation web sites

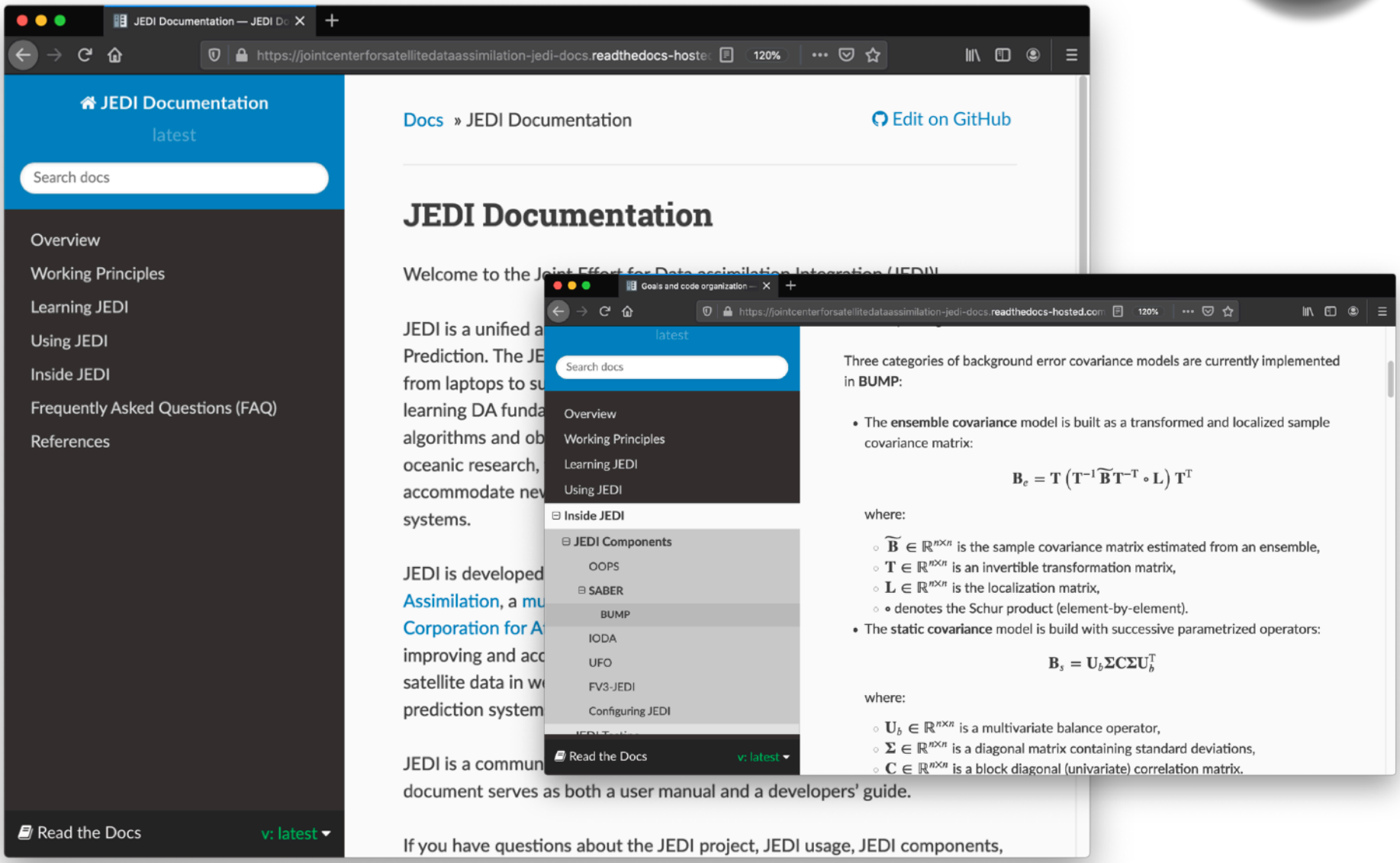
http://academy.jcsda.org/nov2020/pages/doxygen.html

http://jedi-docs.jcsda.org

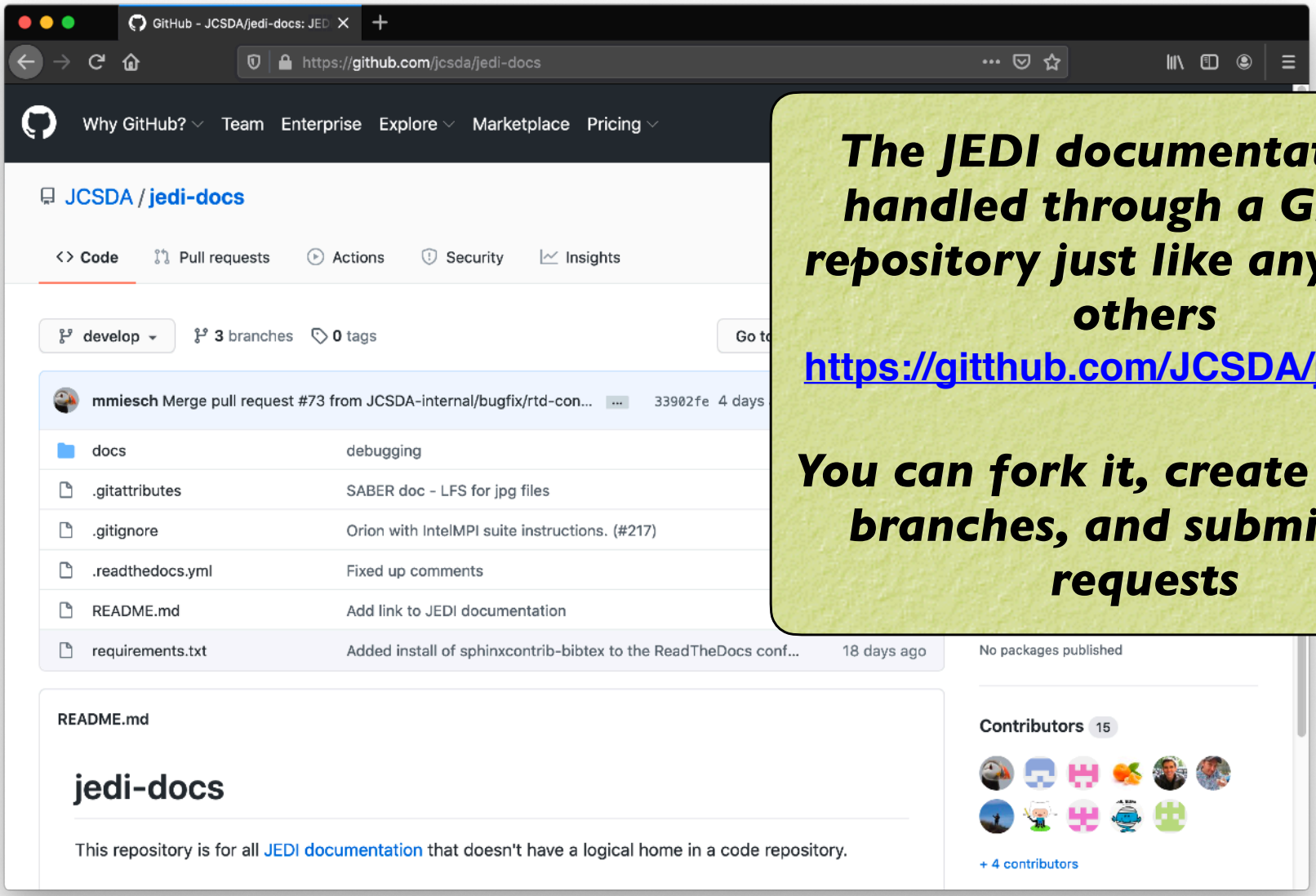
JEDI User/Developer Manual



JEDI User/Developer Manual



jedi-docs GitHub Repository



The JEDI documentation is handled through a GitHub repository just like any of the others

https://gitthub.com/JCSDA/jedi-docs

You can fork it, create feature branches, and submit pull requests

jedi-docs GitHub Repository

Documentation is written as reStructuredText (rst) files which are converted to html by the

Sphinx

Python documentation generator

https://www.sphinx-doc.org

SPHINX
Python Documentation Generator

I) The way of a JEDI

- ✦ Agile project management
- ✦ git and GitHub
- ✦ git-flow

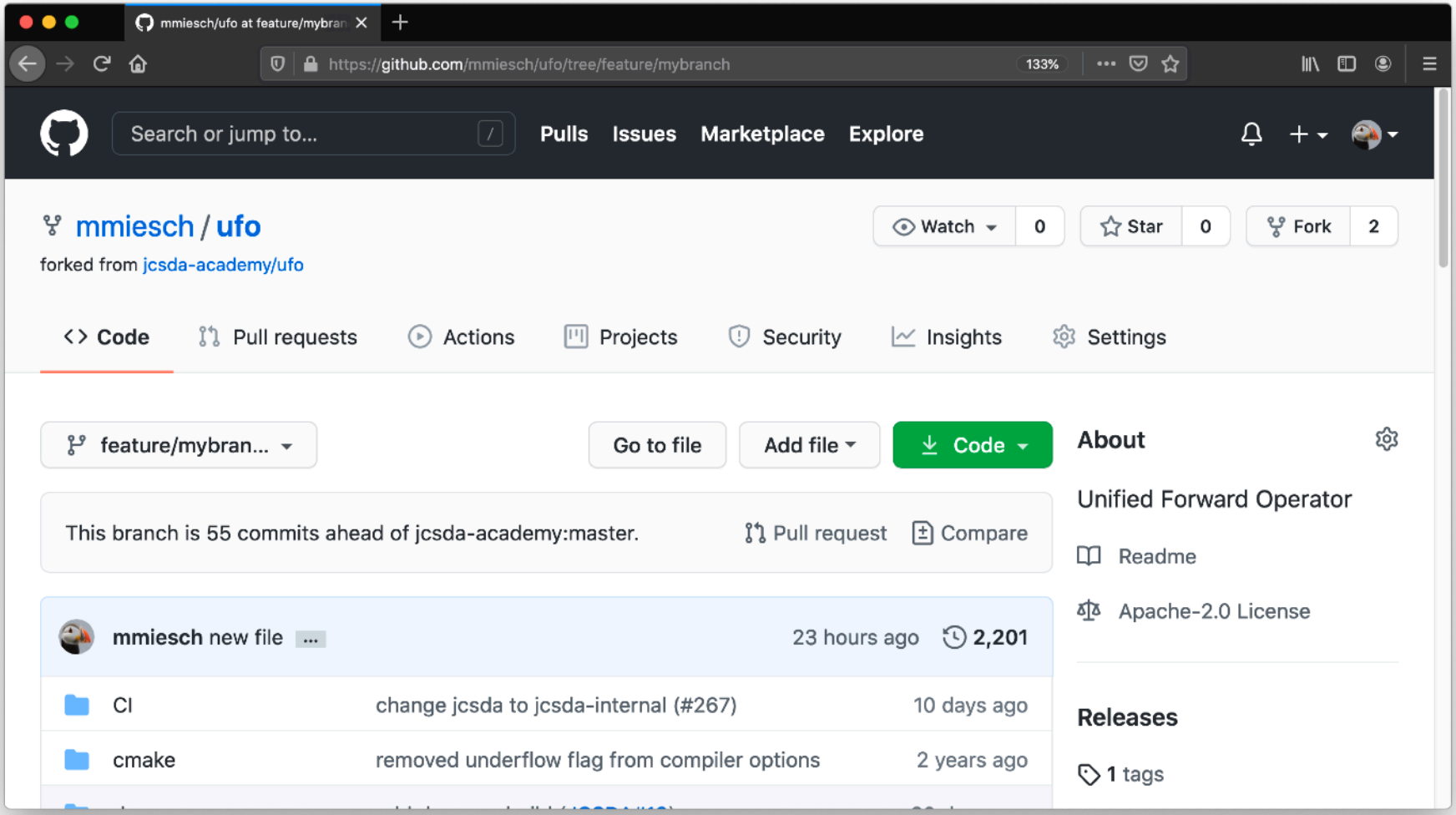
II) Preparing to contribute

- ✦ Work from a fork
- ✦ Make sure your branch is up to date with develop
- ✦ Make sure your code is adequately tested
- ✦ Make sure your code is adequately documented

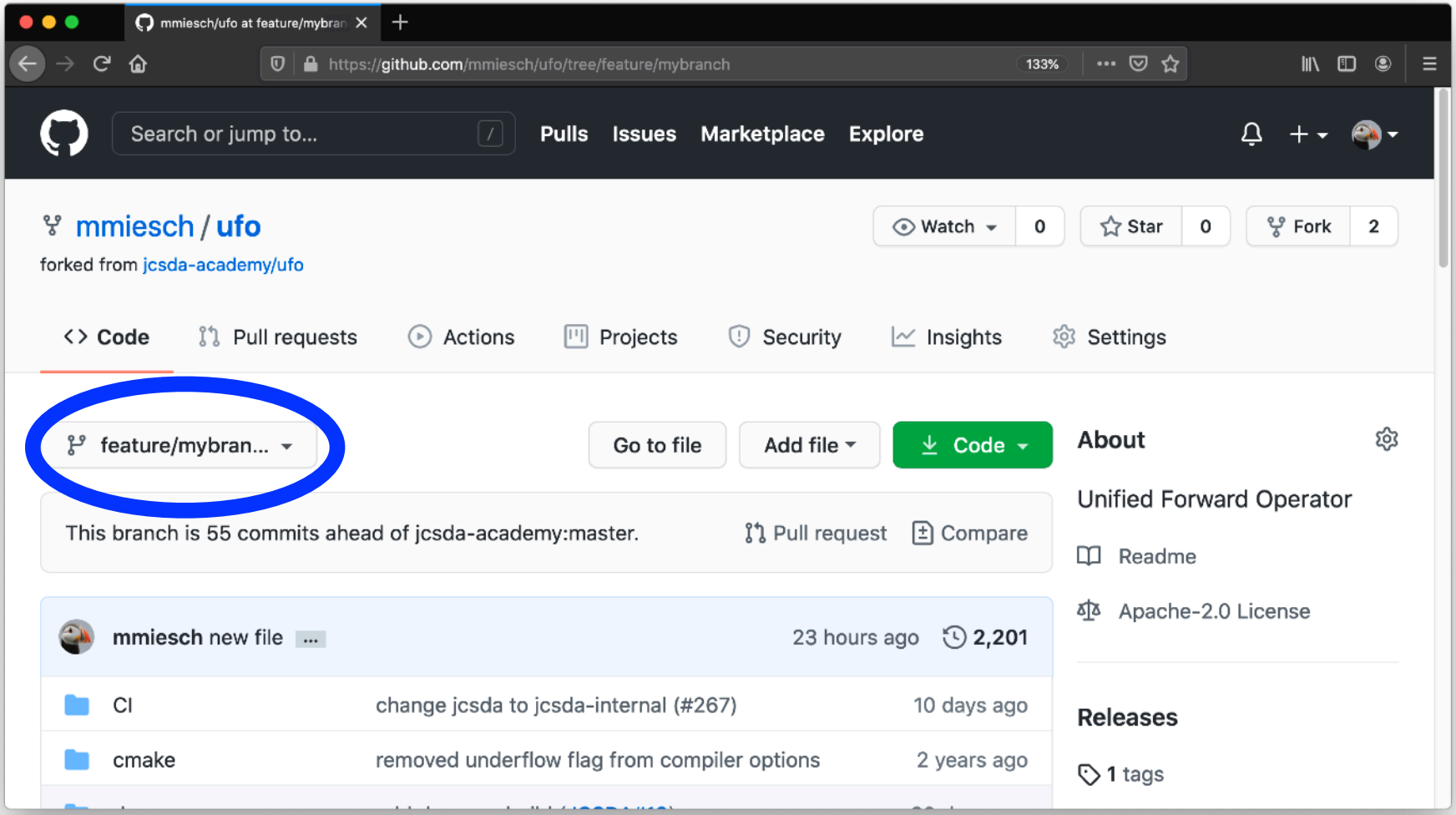
III) Contributing code

- ✦ **Pull requests**
- ✦ **Code Reviews**

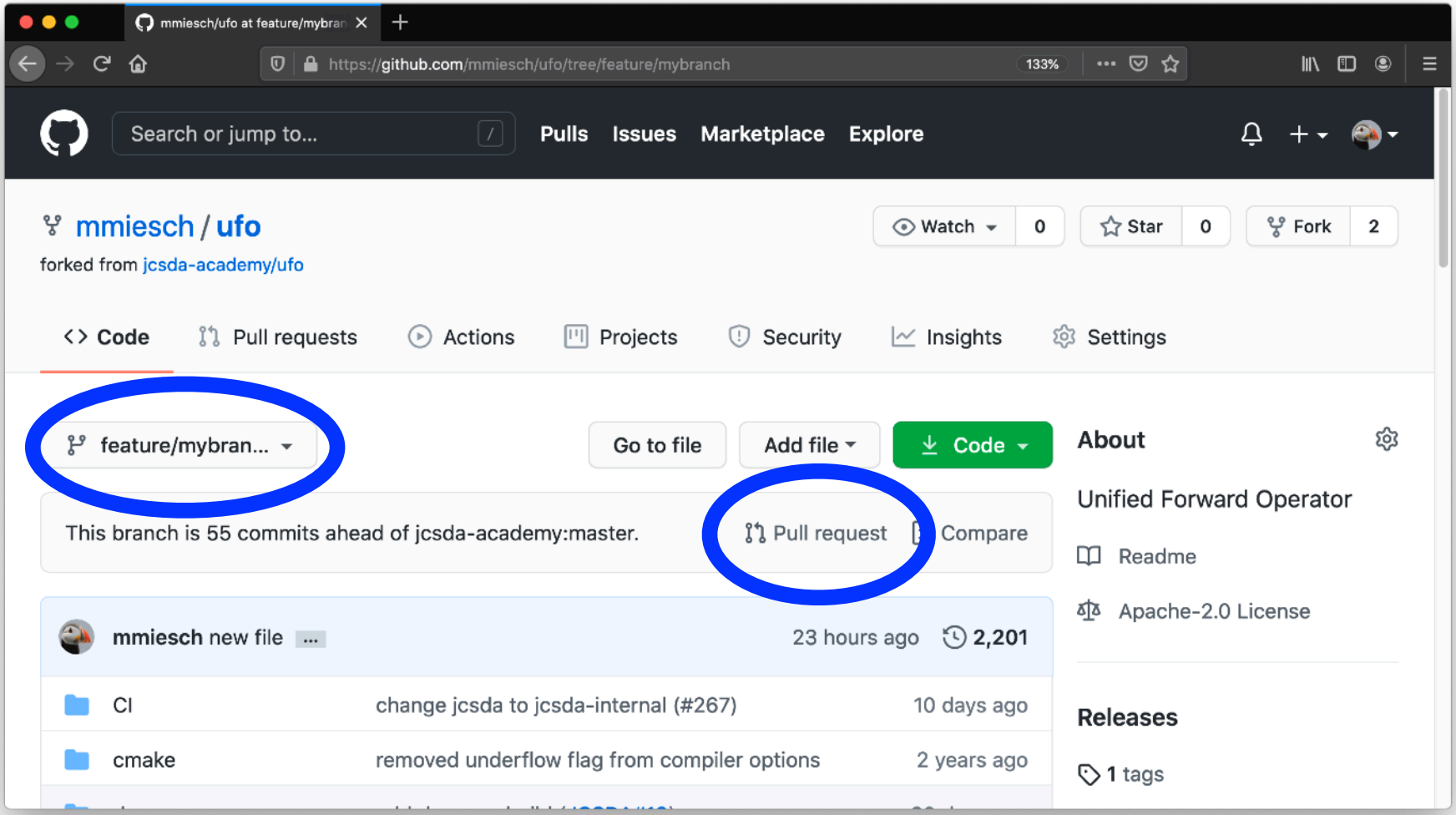
Pull Request



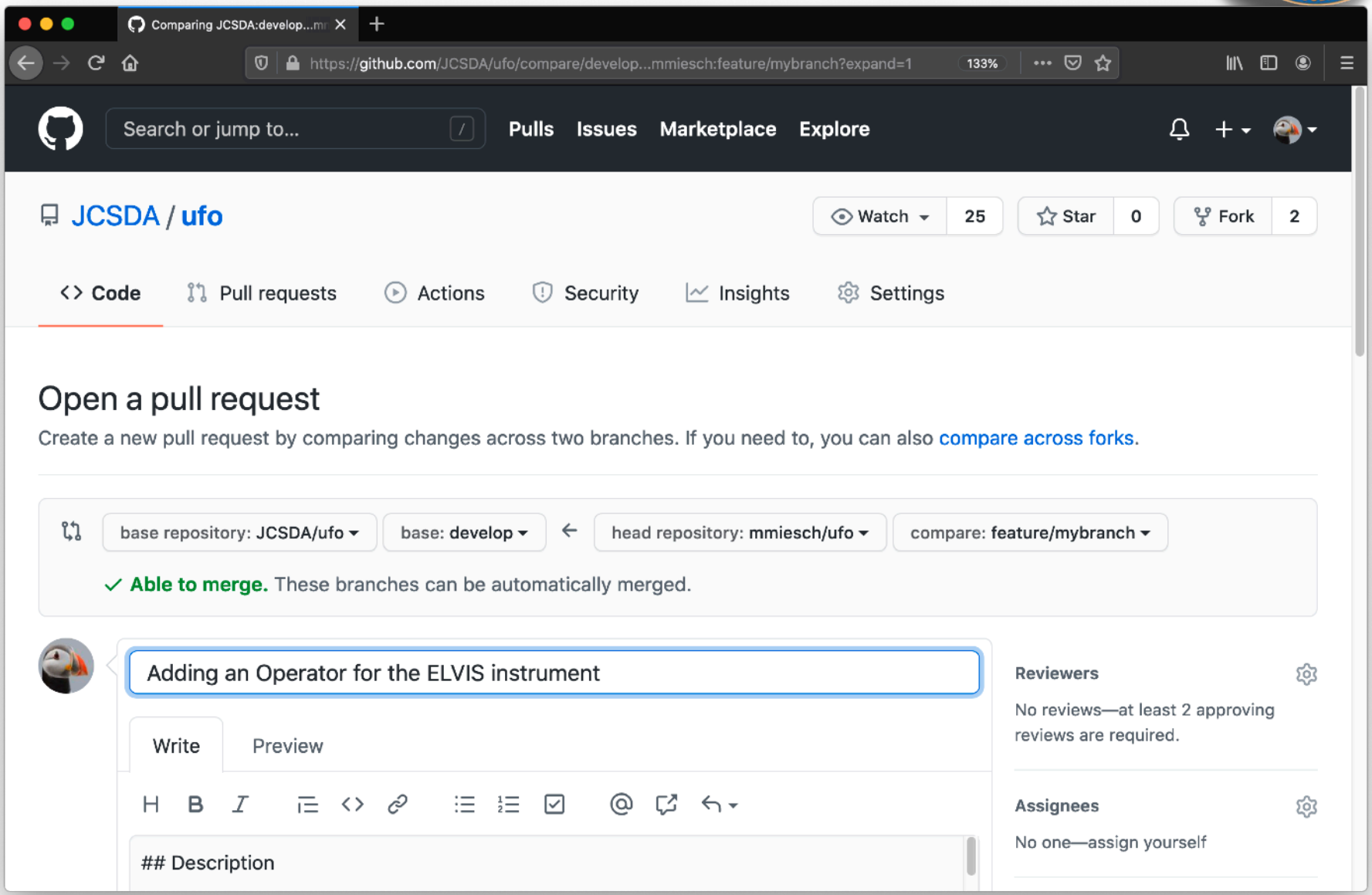
Pull Request



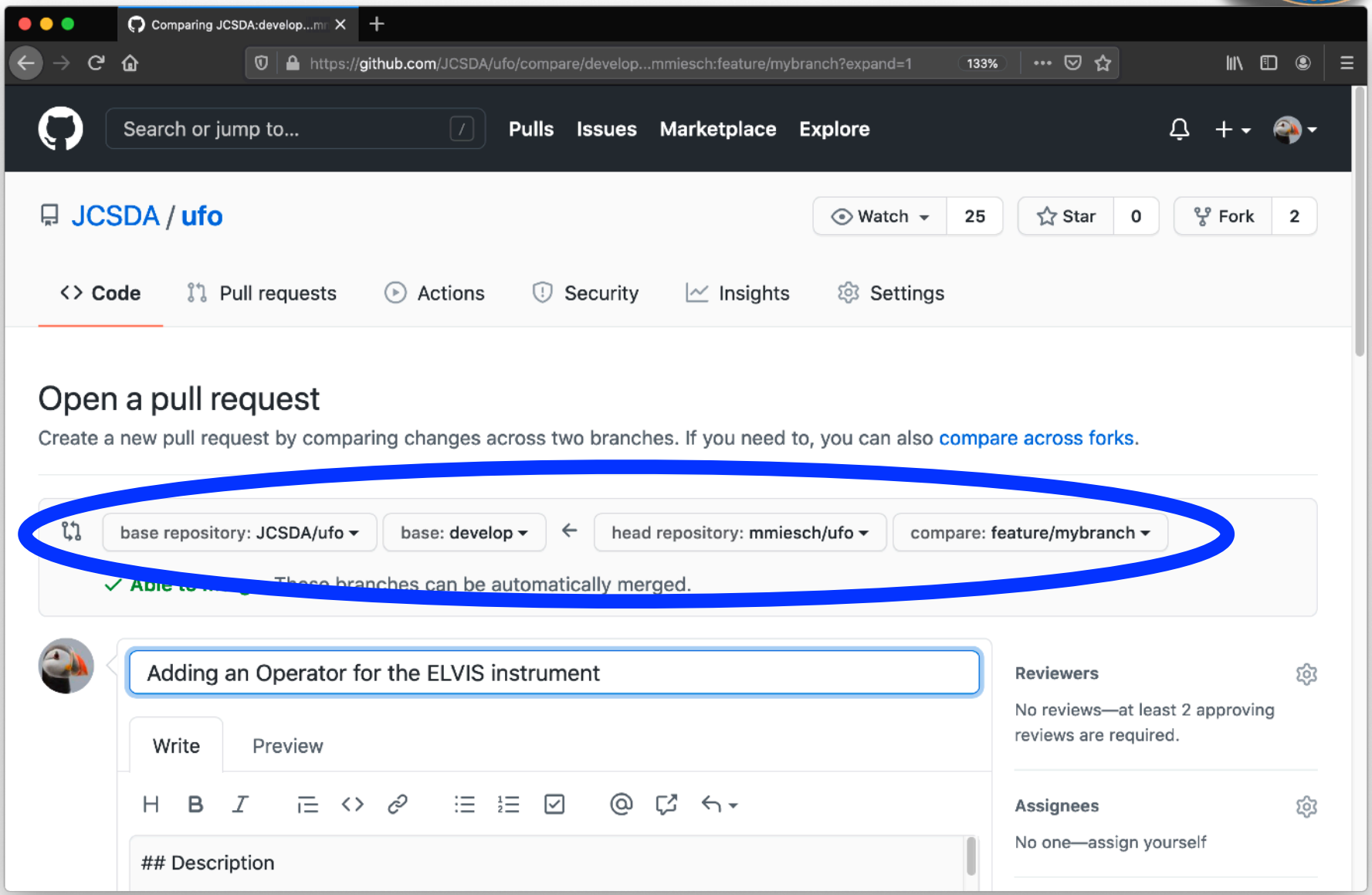
Pull Request



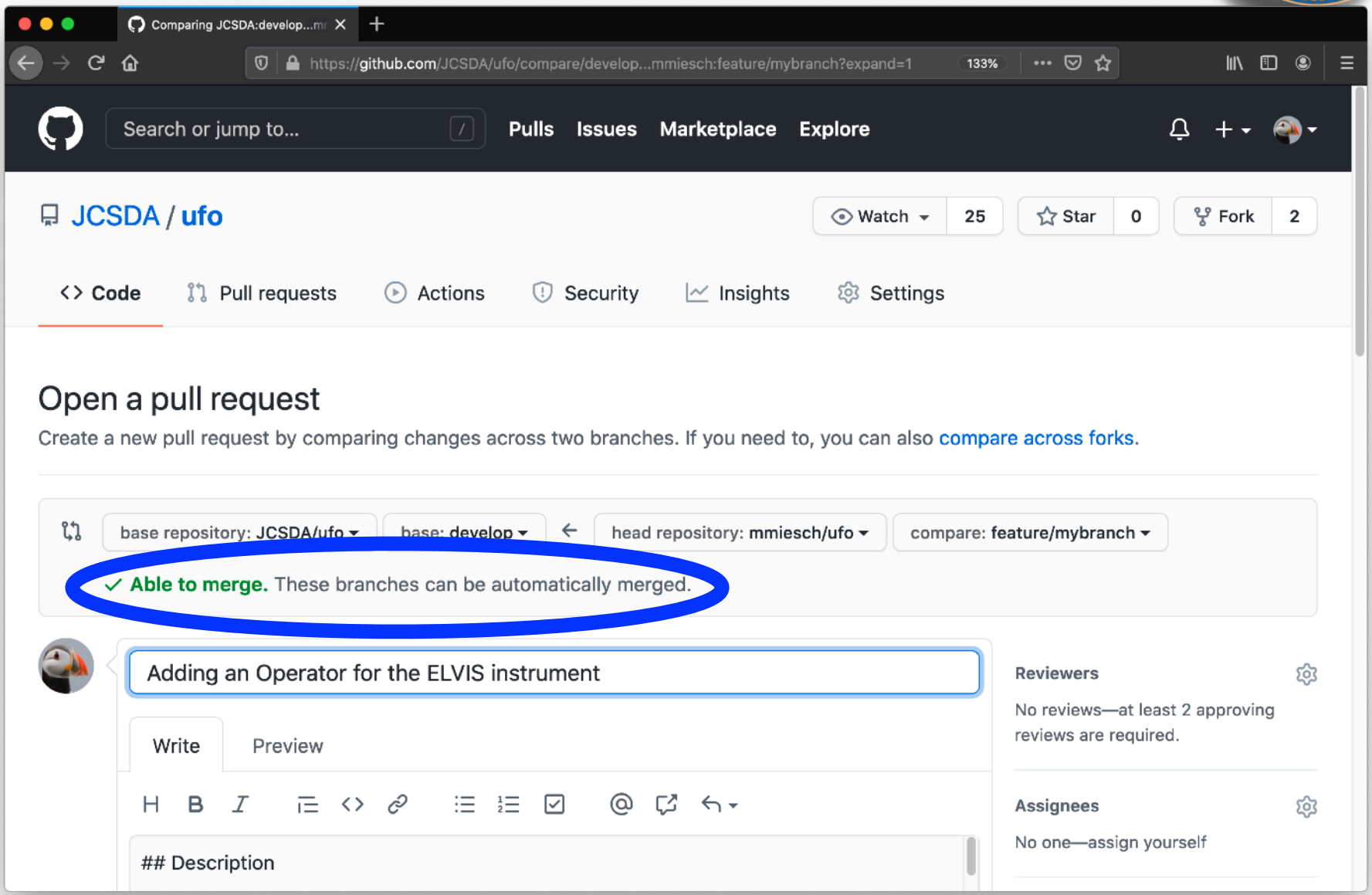
Pull Request



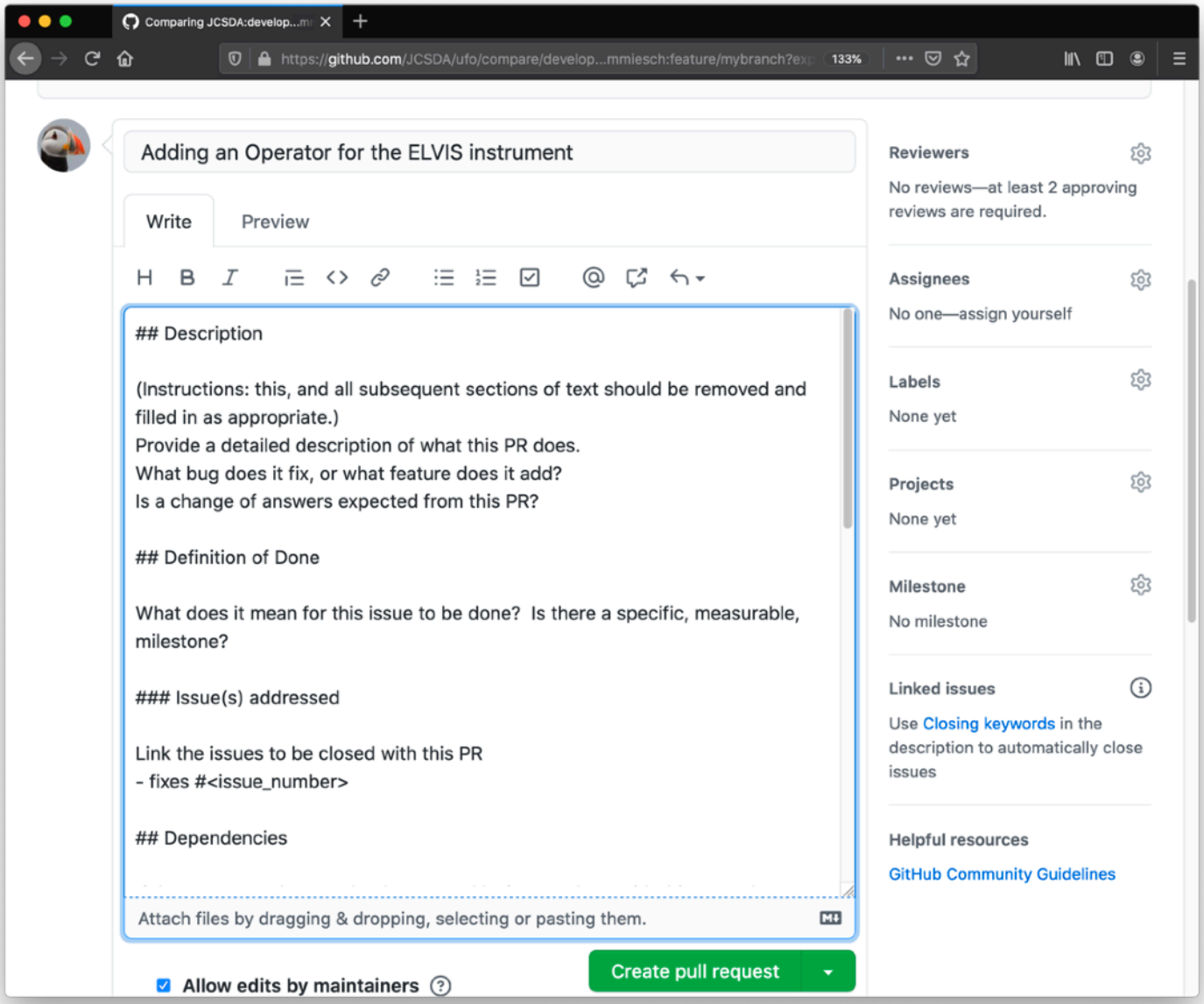
Pull Request



Pull Request



Pull Request

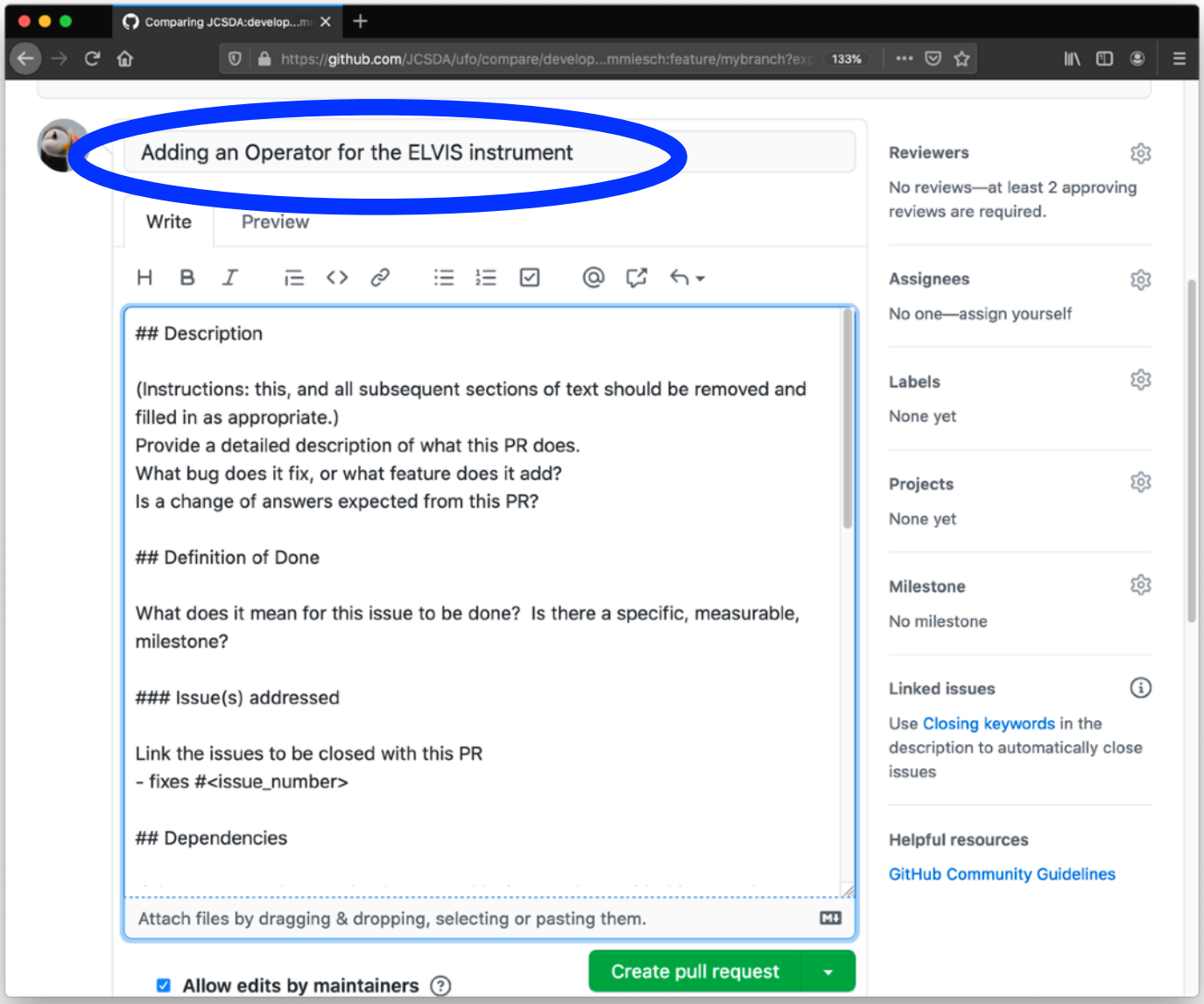


Make it clear what was done and why

Refer to forum discussions if applicable

JEDI team will assign reviewers for external pull requests

Pull Request

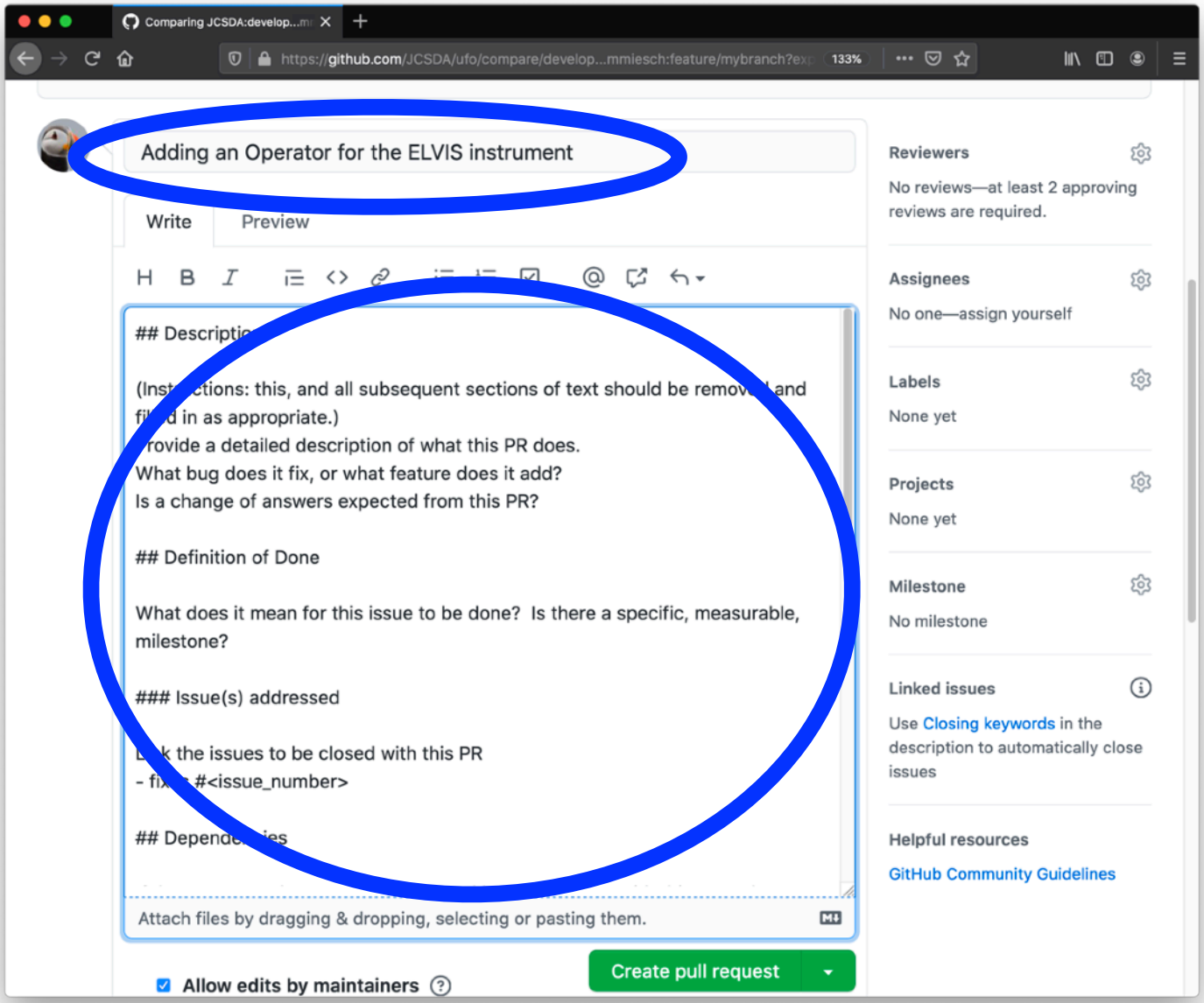


Make it clear what was done and why

Refer to forum discussions if applicable

JEDI team will assign reviewers for external pull requests

Pull Request

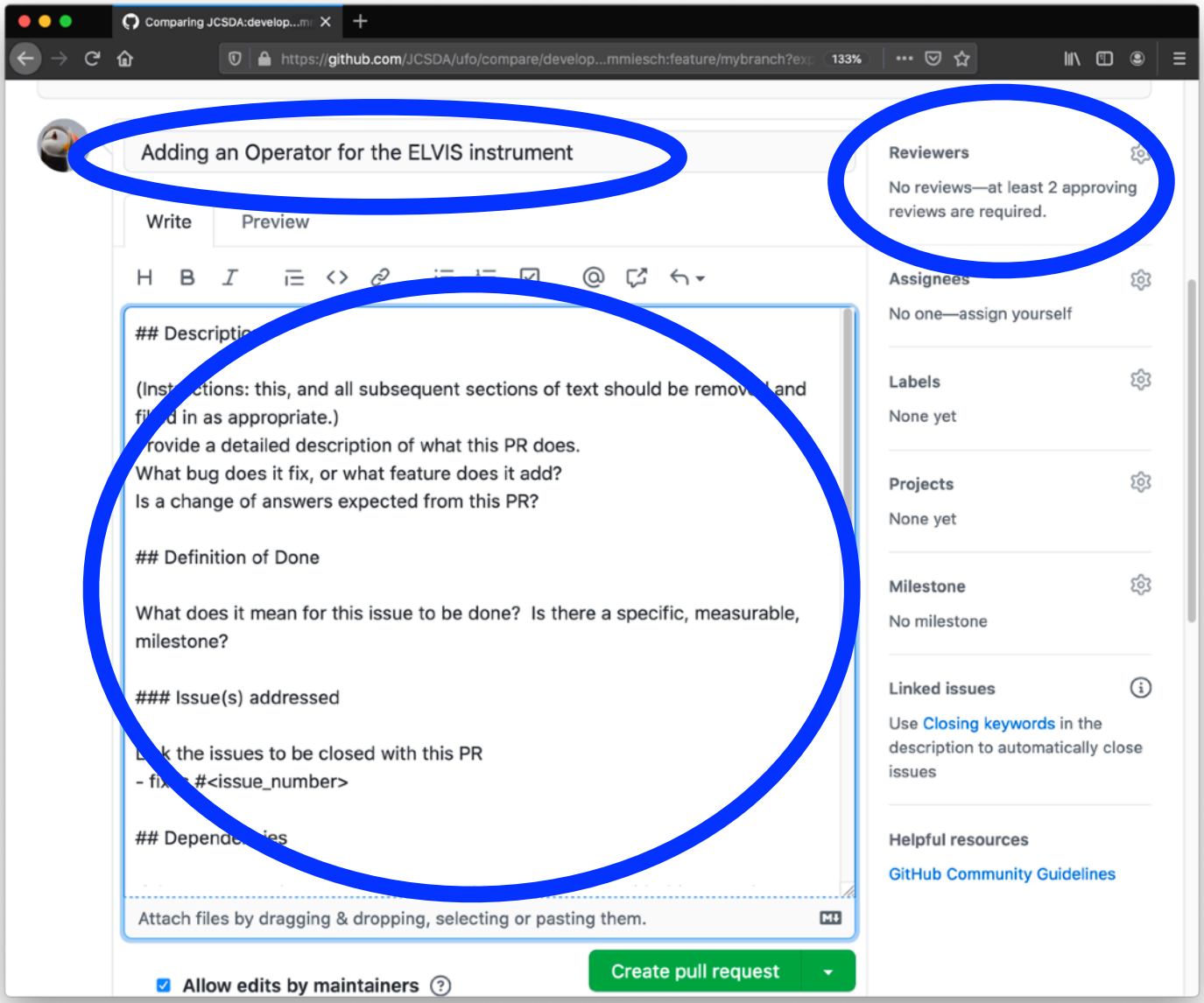


Make it clear what was done and why

Refer to forum discussions if applicable

JEDI team will assign reviewers for external pull requests

Pull Request

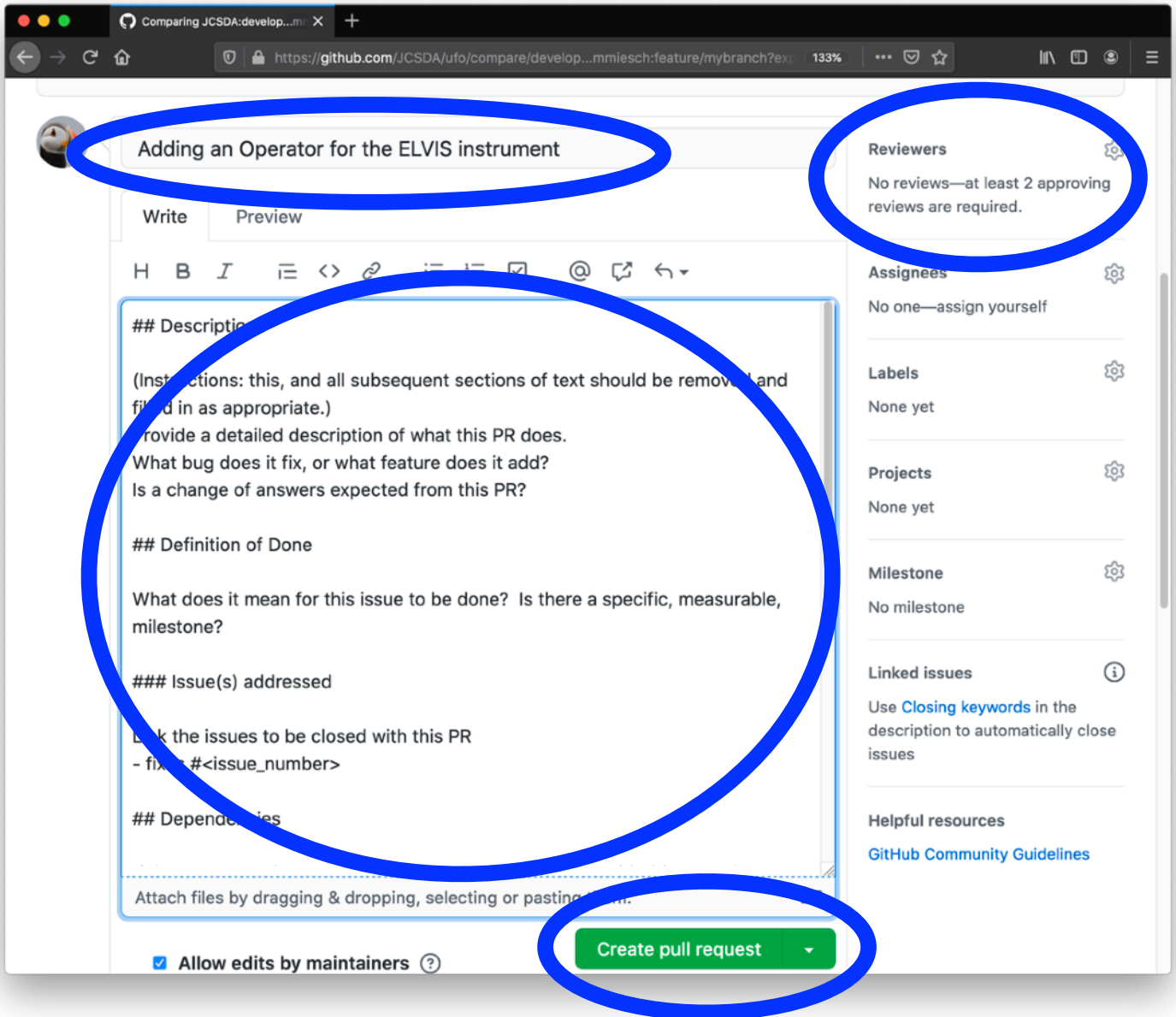


Make it clear what was done and why

Refer to forum discussions if applicable

JEDI team will assign reviewers for external pull requests

Pull Request



Make it clear what was done and why

Refer to forum discussions if applicable

JEDI team will assign reviewers for external pull requests

Pull Requests

- *Make feature branches short and focused*
- *Fill in the requested information in the template*
- *Explain what was done and why*
- *What does it mean for this modification to be finished?*
- *Refer to relevant conversations (forum threads, issues, etc)*
- *Identify appropriate reviewers*
- *Make sure new/modified code is tested*
- *Make sure new/modified code is documented*
- *Be willing to change your code in response to reviews*
- *Read the Working Principles and Best Practices for Developers sections of the JEDI Documentation*

Code Reviews

Purpose

To ensure that the overall health of the code
(scope, functionality, clarity, efficiency, reliability)
improves over time

Requirements

To be useful, they must be
timely, courteous, informative, constructive, and reasonable
(there is no perfect code, only better code)

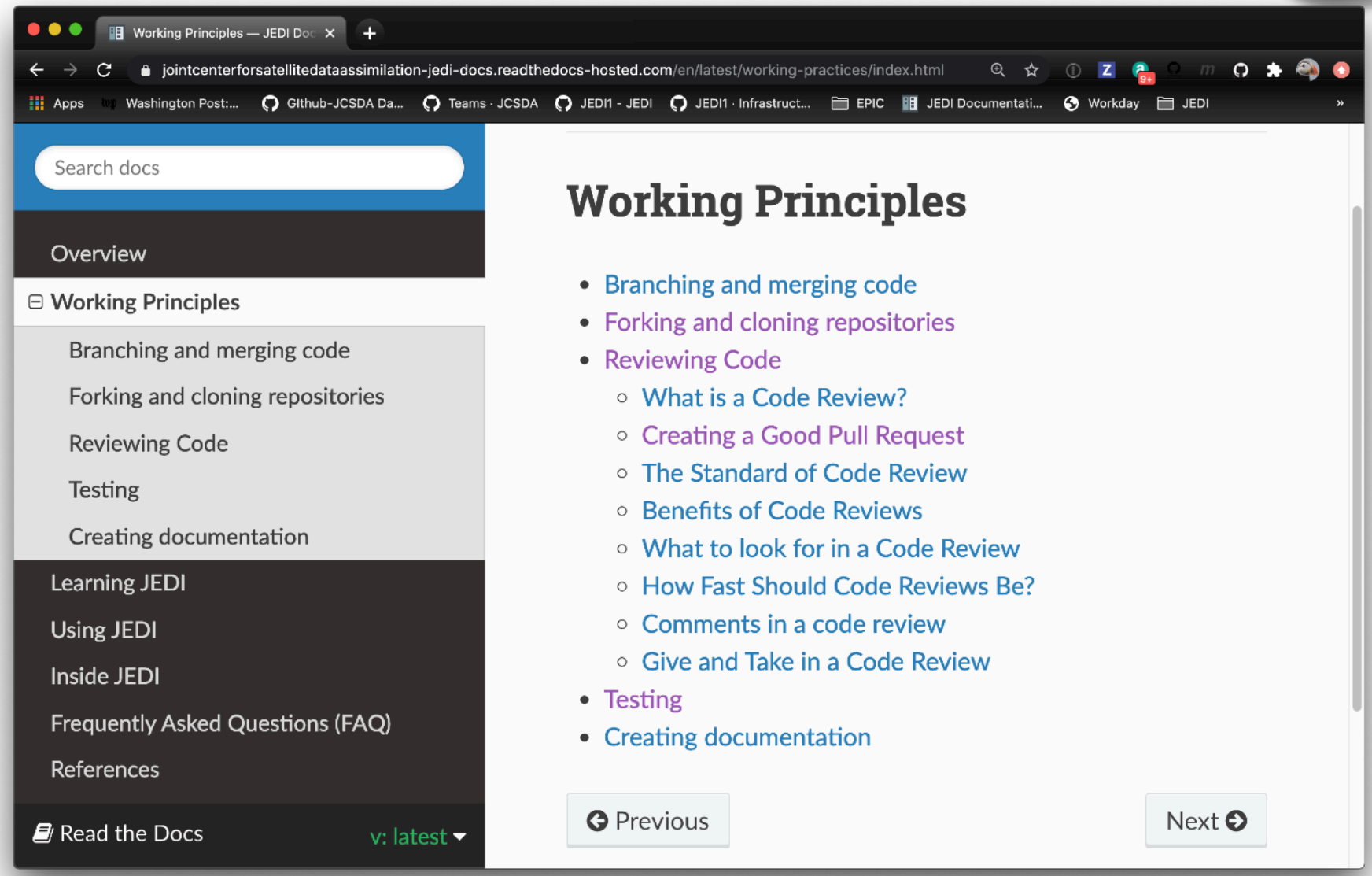
Additional Benefits

Sharing knowledge, team building and mentoring,
improving the development process,
imposing a consistent style & coding norms

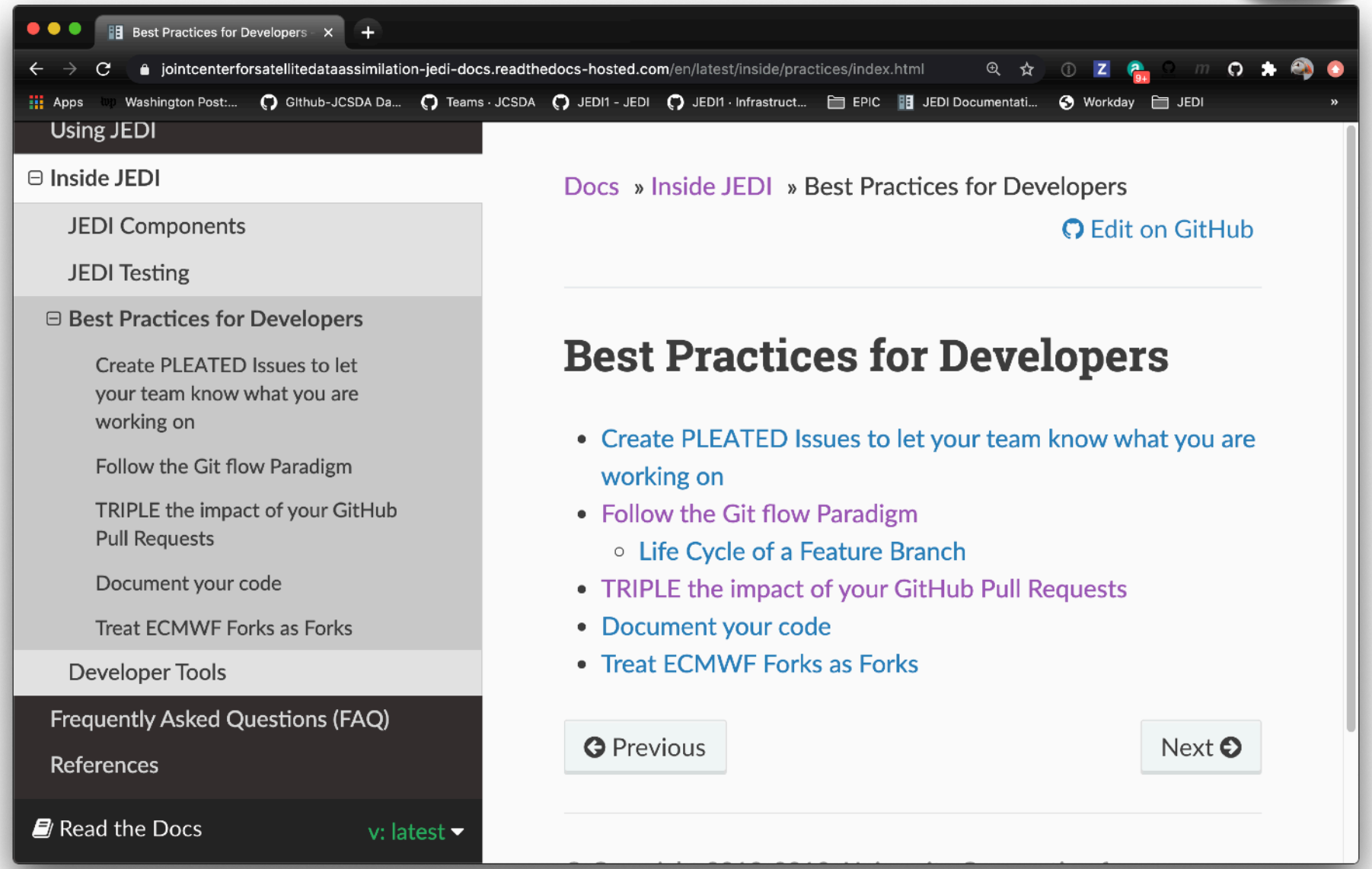
Questions to ask yourself as a reviewer

- ***Does this improve the overall health of the code?***
- ***Is it clear from the title and description what is being done and why? Does it achieve what it says it does?***
- ***Can the desired goal be achieved in a different way that is more readable, more efficient, or more generic?***
- ***Is there extraneous code that should be removed (e.g. debug print statements, unnecessary include statements…)?***
- ***Is the new code adequately tested? Does it pass all tests?***
- ***Is the new code adequately documented?***
- ***Does this belong in the code base or elsewhere (e.g. library)***
- ***Have I read the Working Principles and Best Practices for Developers sections of the JEDI Documentation?***

jedi-docs.jcsda.org



jedi-docs.jcsda.org



Summary

Work from forks, follow git-flow principles

Make sure any code you contribute is well tested and documented

Submit code through pull requests on GitHub and anticipate that each PR will be subject to code reviews and CI testing

Realize that you make be asked to do code reviews as well

Questions Welcome