NW-PaGe
Northwest Pathogen Genomics Center of Excellence

# Northwest Pathogen Genomics Center of Excellence

## Policies, standards, and guidelines

2024-08-16

**Frank Aragona**
Washington State Department of Health
2024
*Data Integration/Quality Assurance*

# Table des matières

Frank Aragona
Washington Department of Health, Data Integration/Quality Assurance
frank.aragona@doh.wa.gov

# 1 Introduction

This document details the policies and guidelines for the Northwest Pathogen Genomics Center of Excellence (NW-PaGe) Github Organization.

For more information, tutorials and code examples, please see the policies website here https://nw-page.github.io/standards/.

# 2 Contribute

**Summary**

- The Northwest Pathogen Genomics Center of Excellence (NW-PaGe) uses a public Github organization to host our code.
- If you want to contribute to the organization, please read this guide and our security guidelines.

**You will need Git and Github to make code contributions:**

- Git is a version control software.
- Github is a platform for developers that utilizes Git
- In order to contribute to this organization you must have Git installed and a Github account

# 3 Git Basics

- You need to install Git on your machine follow here for help.
- For a tutorial on how Git works, follow our Git page here

# 4 Github Basics

- Go to the Github website to create an account.
- Bookmark the NW-PaGe Github Org

# 5 Contributing

There are multiple ways to contribute to a Github repo, whether it is to report a bug, request a feature, or actively contribute to the code base.

## 5.1 Bug Report

To report a bug,

1. click on a repo and click on the `Issues` tab.
2. click the `New issue` button
3. click on the `Bug Report` tab

From here you will need to fill out the bug report along with steps to reproduce the behavior you're seeing.

## 5.2 Feature Request

Do you have a feature that you want included in the code base?

1. click on a repo and click on the `Issues` tab.

2. click the `New issue` button
3. click on the `Feature Request` tab

From here you will need to fill out the feature request along with details

## 5.3 Discussions

There is a discussions tab in our Github org. You can start discussions, ask questions, and share ideas here.

## 5.4 Contribute Code

To contribute to a public repo in our Github org, please contact the repo owner to request read/write access. If you want to create a repo in the org, please contact `frank.aragona@doh.wa.gov` .

Before contributing any code, please read our security policies. There you will find our repo rules and instructions on how to set up pre-commit hooks.

Once granted access, follow the steps below to create a repo (Section 6) and/or collaborate on code (Section 8).

# 6 Create a Repo

Once granted access to create a repo, you can go to our org and click `Repositories > New repository` or click here

This will take you to the `Create a new repository` screen. Please follow these instructions when filling it out:

### 6.1.a Repository template

Consider using a template unless you want to develop a repo from scratch. We have pre-built R, Python, and base templates that have Github Codespaces set up as well as `.gitignore` files and virtual environments.

### 6.1.b Owner and Repo name

Make sure the `Owner` name is NW-PaGe. Name your repository something descriptive and easy to type out. Avoid spaces and capital letters unless necessary.

The repo description can be filled out at any time after creating the repo

### 6.1.c Internal vs Private

We don't allow you to create a public repo initially. Please create a Private repo first, and then once you are ready to make it public you can.

### 6.1.d README, .gitignore, license

- Check the `Add a README file` box.
- Add a `.gitignore` if the option is available (choose either R or Python)
- Choose an `MIT` license unless you know you want a different license more info here

# 7 Cloning a Repo

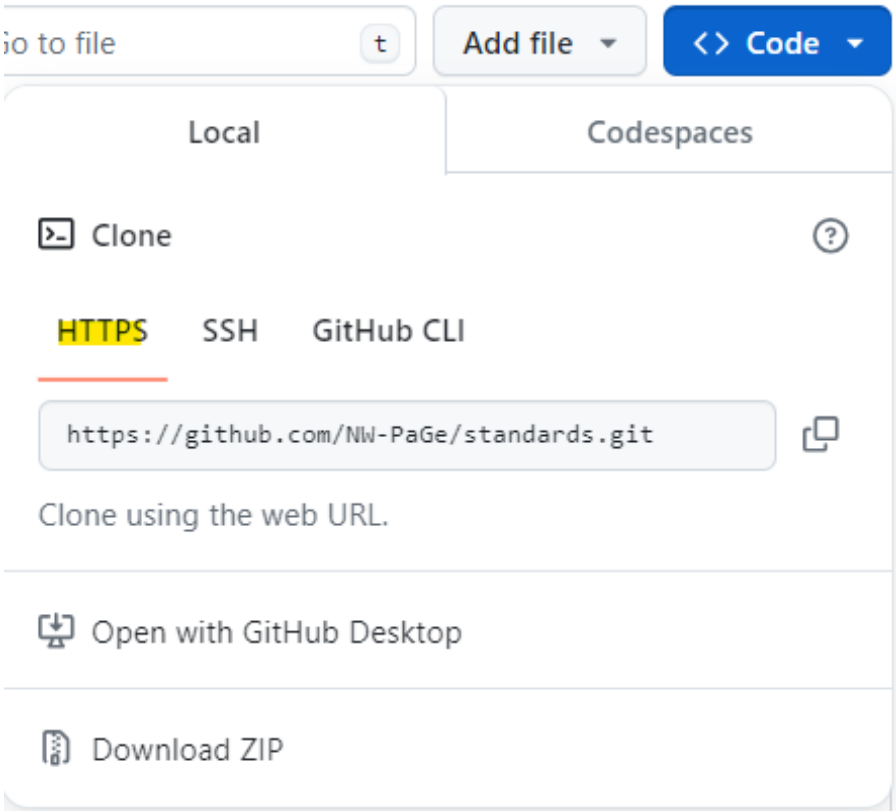
Whether you have created your own repo or want to contribute to someone else's repo, you will need to make a local clone of that repo on your personal machine.

To make a local clone of a repo, click on the green `Code` button when you're in the main repo's web page. In the local tab there are multiple ways to clone. For most of our work, I suggest creating an SSH key. If you are new to git/Github and on a Windows machine, I recommend installing the Github Desktop app and following the instructions below.

## 7.1 HTTPS

Cloning via HTTPS is a relatively quick process.

1. Start by navigating to the repo in Github and selecting the `Code` button:



2. Copy the path that starts with `https://` , in this case it's `https://github.com/NW-PaGe/standards.git`
3. In a terminal/command prompt, navigate to a folder of your choice (in windows I would make a folder called Projects here: `C:/Users/<username>/Projects` )

```
cd C:/Users/<your_username>/Projects
```

4. Use git clone and replace the `https://github.com/NW-PaGe/standards.git` with your path:

```
git clone https://github.com/NW-PaGe/standards.git
```

5. Check if things ran by executing this code:

```
git status
```

**NOTE:** the HTTPS method is good but it will require you to enter your username and a token every time you push a commit to the remote repo in Github. You will need to create a Personal Access Token (PAT) whenever you want to make a commit. If this is annoying to you, use the SSH or Github Desktop App methods.

### 7.1.a Make a PAT

Here's a guide on making a PAT

1. Click on you Github profile icon in the upper right
2. Click Settings
3. Scroll down to `Developer Settings`
4. Select Personal access tokens (classic) and then Generate new token
5. When you make a commit you will need to input this personal access token when it asks for your password.

**Do not store this token anywhere! Especially make sure it is not stored in your repo. This has tons of security risks and needs to be for singular use only**

## 7.2 SSH

SSH is an excellent option for cloning a repo. It is similar to using an identifier to tell Github that you are, in fact, you. This video below is a great resource on how to set up the key. I will also write out the steps in the video below. Also, see the Github documentation for more information.

https://www.youtube.com/embed/8X4u9sca3Io?si=bHKQHA28VBz2PXUP

1. In a terminal, write the following and replace the email with your email:

```
ssh-keygen -t ed25519 -C your@email.com
```

2. It should then ask if you want to make a passphrase. I recommend doing this
3. Get the pid

```
eval "$(ssh-agent -s)"
```

4. Make a config file

```
touch ~/.ssh/config
```

5. If the file doesn't open, you can open it like this

```
nano ~/.ssh/config
```

6. Add this to the config file. it will use your passkey and recognize you

```
Host *
  IgnoreUnknown AddKeysToAgent,UseKeychain
  AddKeysToAgent yes
  IdentityFile ~/.ssh/id_ed25519
  UseKeychain yes
```

To save this file in nano, on your keyboard write `CRTL+O` then `ENTER` to save the file. Then `CTRL+X` to exit back to the terminal. You can also open this file through a notepad or other software. You could also search for the file in your file explorer and edit it in notepad if that is easier.

7. Add the identity

```
```

```
ssh-add ~/.ssh/id_ed25519
```

8. In Github, go to your profile and the `SSH + GPG Keys` section
9. Click SSH Keys, add a title, and in the key location write your key. You can find your key in your terminal by writing:

```
cat ~/.ssh/id_ed25519.pub
```

**Copy the whole output including your email and paste it into the Github key location**

10. Test it by writing this:

```
ssh -T git@github.com
```

11. Use the key to clone a repo.

Now you can clone a repo using the SSH key. Copy the SSH path and write this (replace the string after clone with your repo of choice):

```
git clone git@github.com:org/reponame.git
```

## 7.3 GitHub CLI

The GitHub CLI is an excellent tool for not just cloning your repo, but for managing repositories and organizations from a terminal.

### 7.3.a Windows

To install the CLI in Windos, I follwed the instructions provided in the Github CLI repo.

I normally install commands using Scoop, but you have many options here.

1. Paste this code into a powershell window and execute it

```
winget install --id GitHub.cli
```

2. Now update the package

```
winget upgrade --id GitHub.cli
```

3. You will need to authorize your github account like this:

```
gh auth login
```

4. It will ask you to authorize in a browser or with a personal access token

I created a personal access token.

5. Now you can clone a repo like this:

```
gh repo clone org/repo-name
```

You can also now do some cool things with your org/repo like searching for strings, creating issues, and more. For example, here are the issues in this repo:

```
gh issue list
```

```
Showing 3 of 3 open issues in NW-PaGe/standards

ID  TITLE                                  LABELS         UPDATED
#7  add .gitignore documentation                          about
2 months ago
#3  Make sure all references are added to ...  documentation  about
5 months ago
#2  Fix cross reference links                  documentation  about
5 months ago
```

### 7.3.b WSL/Linux

To install in a linux terminal, I'm following the instructions provided in the Github CLI repo.

1. Paste this code into your bash terminal and execute it.

```
(type -p wget >/dev/null || (sudo apt update && sudo apt-get install
wget -y)) \
&& sudo mkdir -p -m 755 /etc/apt/keyrings \
&& wget -qO- https://cli.github.com/packages/githubcli-archive-
keyring.gpg | sudo tee /etc/apt/keyrings/githubcli-archive-
keyring.gpg > /dev/null \
&& sudo chmod go+r /etc/apt/keyrings/githubcli-archive-keyring.gpg
\
&& echo "deb [arch=$(dpkg --print-architecture) signed-by=/etc/
apt/keyrings/githubcli-archive-keyring.gpg] https://cli.github.
com/packages stable main" | sudo tee /etc/apt/sources.list.d/
```

```
github-cli.list > /dev/null \
&& sudo apt update \
&& sudo apt install gh -y
```

2. Then upgrate the command with the code below

```
sudo apt update
sudo apt install gh
```

3. You now need to authorize yourself as a user.

```
gh auth login
```

4. It will ask you to authorize in a browser or with a personal access token

I created a personal access token. In linux there are some issues with the command and using a browser fyi.

5. Now you can clone a repo like this:

```
gh repo clone org/repo-name
```

You can also now do some cool things with your org/repo like searching for strings, creating issues, and more. For example, here are the issues in this repo:

```
gh issue list
```

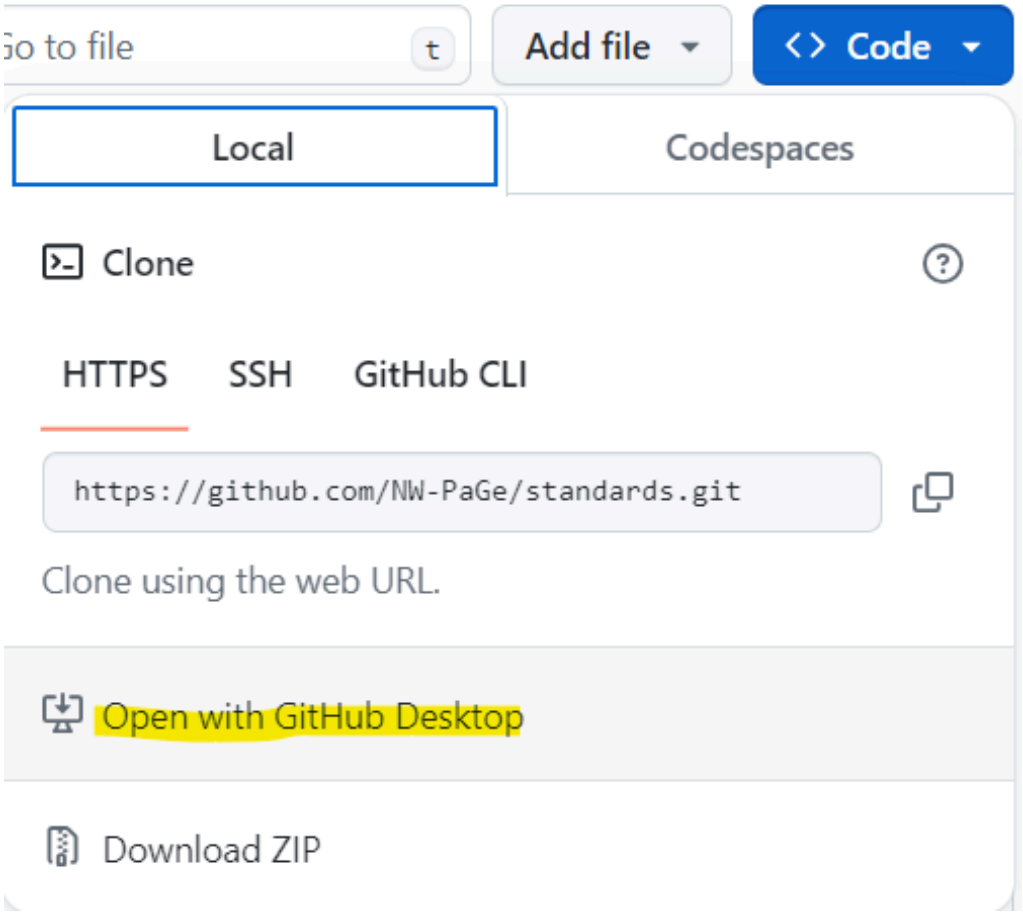
```
Showing 3 of 3 open issues in NW-PaGe/standards

ID  TITLE                                    LABELS         UPDATED
#7  add .gitignore documentation                             about
2 months ago
#3  Make sure all references are added to ...  documentation  about
5 months ago
#2  Fix cross reference links                  documentation  about
5 months ago
```

## 7.4 Open with GitHub Desktop

If you're new to Git or Github and are using a Windows machine, the GitHub Desktop app is a great option for managing git workflows.

1. Install the app

2. You will need to authenticate your account
3. Now you should be able to clone repos through the app. In Github, when you click on the Code tab you will see the option to open in Github Desktop:




This will open up the desktop app and let you choose a file path for your Github repos. I recommend putting your repos into a Github or Projects folder in your local C drive, like this

- `C:/Users/yourname/Projects/<your-repo>/`

If you're cloning many repos you should put the repos into folders separated by the Github org

- `C:/Users/yourname/Projects/<gh-org-name>/<repo-in-org>/`

# 8 Code Collaboration

Code collaboration can vary widely depending on the team and owner of the repo. Below I will describe how our team collaborates in Github. To summarize,

in our system we prioritize preserving the main branch of a repo for production level code, and when we want to make changes we do it locally in a separate branch. We have rules for our commit messages, making pull requests (merging changes into the main branch), taging issues, release cycles, and documenting changes.

## 8.1 Before you make code changes

After creating a repo (Section 6) and/or cloning the repo (Section 7) into a your local machine, you can start writing and contributing code to the remote code base in Github.

1. Check that you have your local clone linked to the remote repo by running git status. It should tell you that you're on the main branch

```
git status
```

2. We need to refresh the repo and check for any code changes to make sure our local clone is up to date. Use `git fetch` to find changes and `git pull` to pull those changes into your local clone.

```
git fetch
git pull
```

3. Make a new branch so we can isolate your changes and prevent accidentally pushing code changes to the main branch

```
git branch <insert your branch name here>
```

4. Switch to that branch

```
git switch <your branch name here>
```

Now you are working in the local branch that you created and you can begin writing code or making updates.

## 8.2 Committing your changes

Once you make changes you can commit them to the local branch you created. This is like saving your work to the branch. The branch can be pushed to the remote repo in Github, so you can continually make changes and push them to the remote where they will be stored safely.

To make a commit, save your work and then in the terminal write `git commit -m "<your message here"` , like this:

```
git commit -m "feat: this is a new feature!"
```

Note that the `-m` is a parameter to let you write a commit message. Commit messages are important so that other collaborators can understand what changes you made. You can write a description like this

```
git commit -m "feat: this is a new feature!" "this is a description.
I made this feature in the code"
```

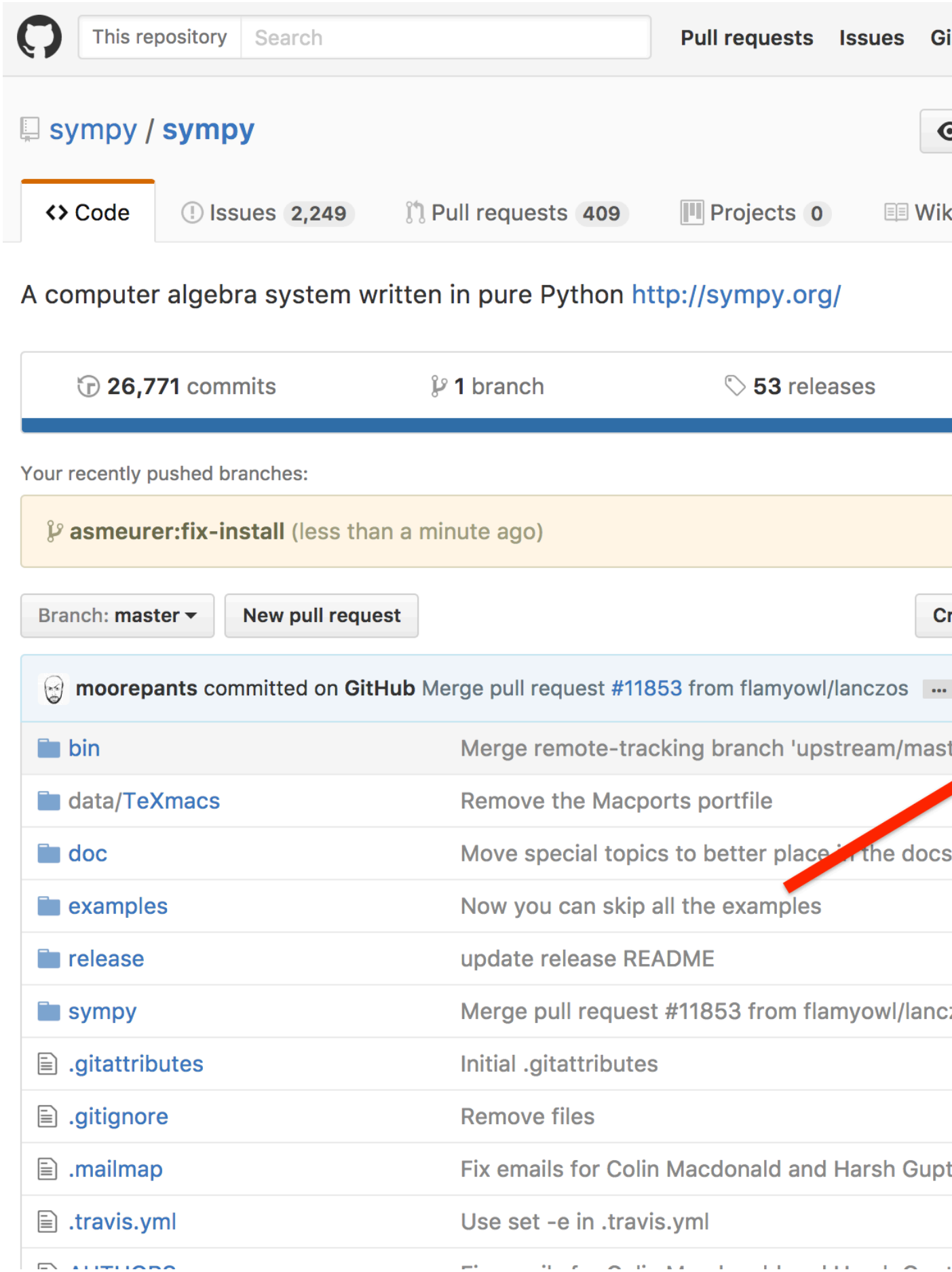
Also note that I am using the word `feat` in the commit message. This is important word that can trigger a github action. We'll cover it below in Section 8.5

## 8.3 Make a pull request

Now you have committed your changes, but your code has only been committed to a *branch*. In order to have your changes implemented in the main codebase you need to merge your branch into the main branch.

When working in a collaborative team setting it is important to have your team review the changes you made before implementing them into the main branch. Everyone makes mistakes, and this is an opportunity to vet your code and have everyone sign off on the changes you want.

After you make your commits, go back to Github in your browser and go to your repo. There should be a box that appears showing your commit and a button that says `Compare & pull request`

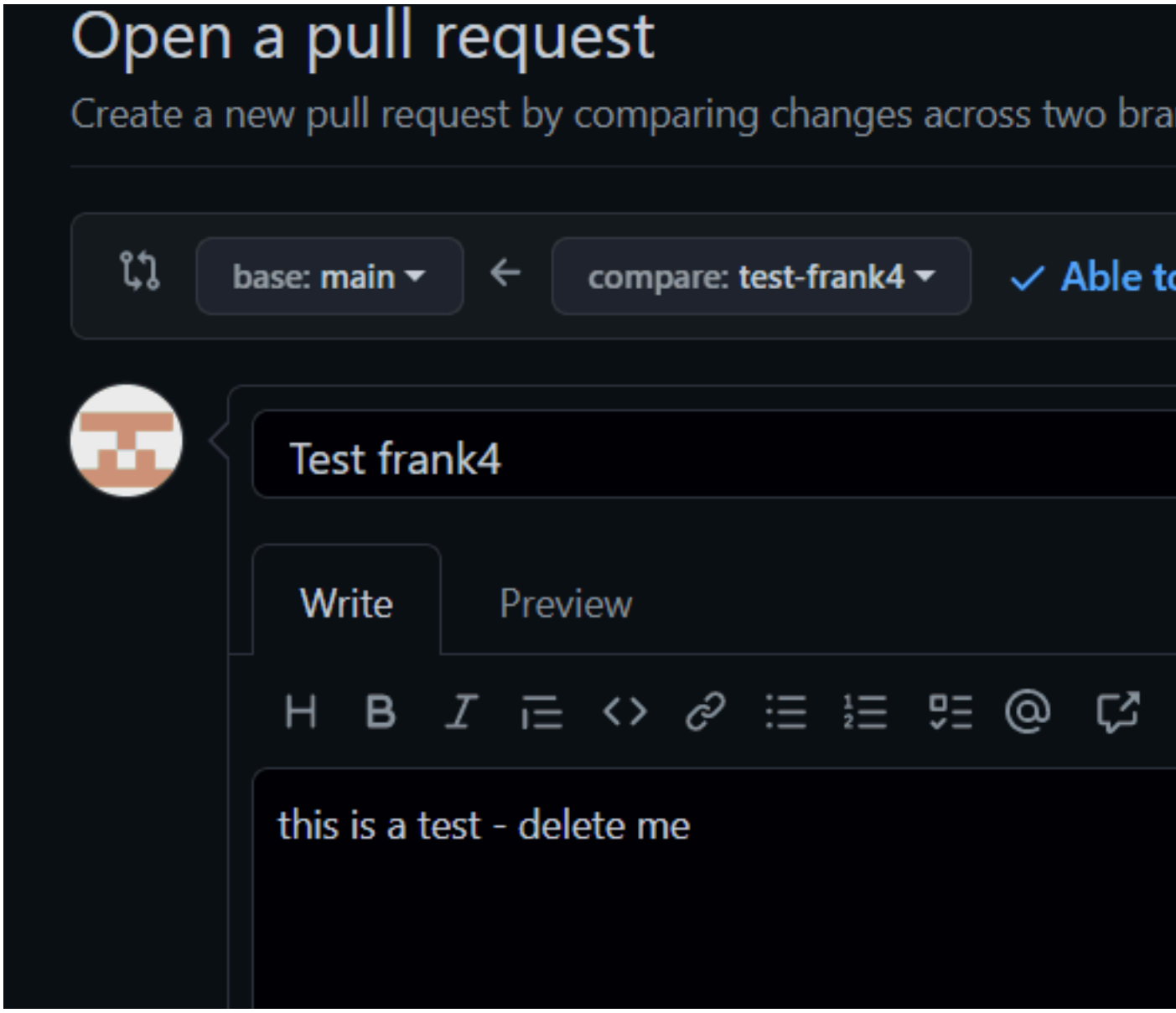

This repository
Search
Pull requests
Issues
sympy / sympy
Code
Issues 2,249
Pull requests 409
Projects 0
A computer algebra system written in pure Python http://sympy.org/
26,771 commits
1 branch
53 releases
Your recently pushed branches:
asmeurer:fix-install (less than a minute ago)
Branch: master
New pull request
moorepants committed on GitHub Merge pull request #11853 from flamyowl/lanczos
bin
Merge remote-tracking branch 'upstream/mast
data/TeXmacs
Remove the Macports portfile
doc
examples
Now you can skip all the examples
release
update release README
sympy
Merge pull request #11853 from flamyowl/lanc
.gitattributes
Initial .gitattributes
.gitignore
Remove files
.mailmap
Fix emails for Colin Macdonald and Harsh Gupt
.travis.yml
Use set -e in .travis.yml

Click that button and it will bring you to the `Open a pull request` page.

1. Select who you want to review your code and assign yourself.
2. Use labels to tag what this pull request refers to (very helpful in search for changes when managing the project) and
3. Add a milestone if it applies.

Note that labels, milestones, and projects are a way to keep track of changes and issues in your project. I highly recommend setting them up, more below.

This should automatically send an email to the reviewers that there is code needed to be merged to a branch.



## 8.4 Merging a branch to main

Typically your teammates and the repo admin will review your code and merge your branch into the main branch.

In Github, click on the `Pull requests` tab. Here you will see open pull requests and you can click on the one you want to merge.

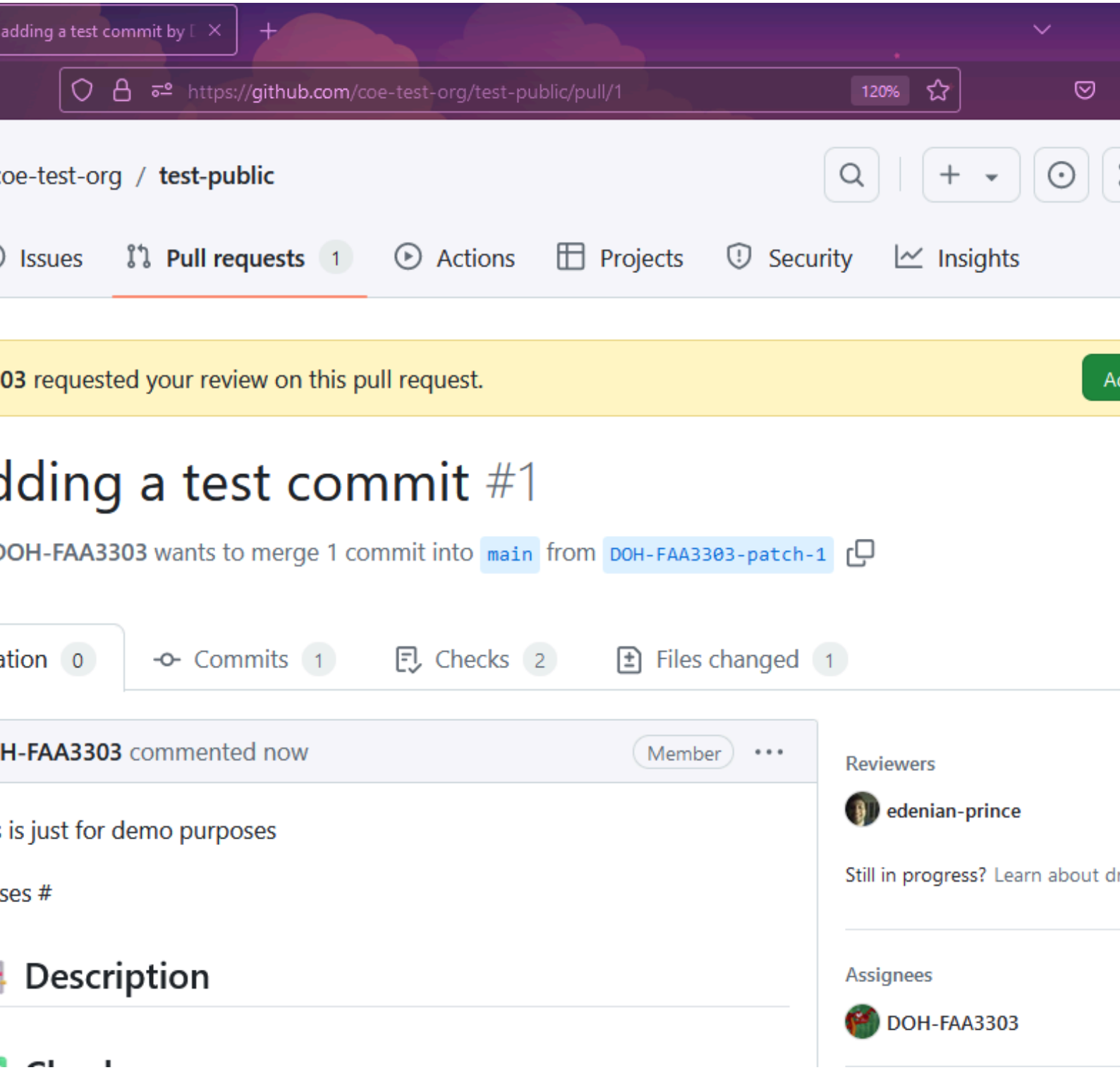



Figure 1 — Review a pull request

In the pull request you will see 4 tabs;

1. Conversation tab that shows all the comments, descriptions, tags, and more.
2. Commits tab that contains a list of all the commits made in this request
3. Checks - if you have automated testing or apps in the repo you can trigger them with a pull request and see them here. For example, you can set up automated unit tests to run whenever a pull request is made. A github action will run the unit test and pass (or fail) here before it is merged to main.
4. Files changed - I personally always flip through this tab because it shows all the differences (diffs) between the old codebase and the new commits that were made.

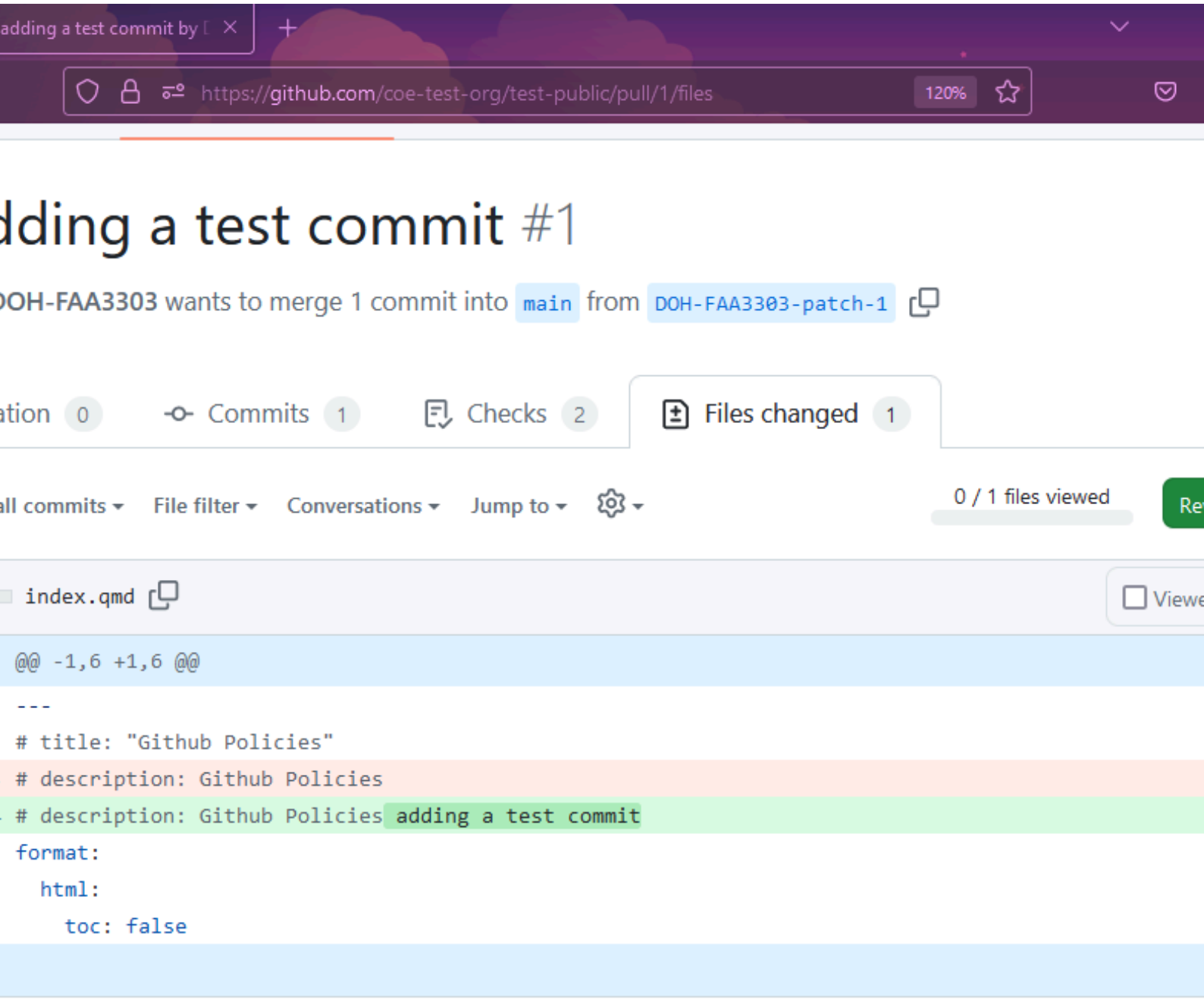



Figure 2 — Files Changed

When you are comfortable with merging these changes, you can either leave a comment, approve, or request further changes by clicking on the `Review changes` dropdown menu.

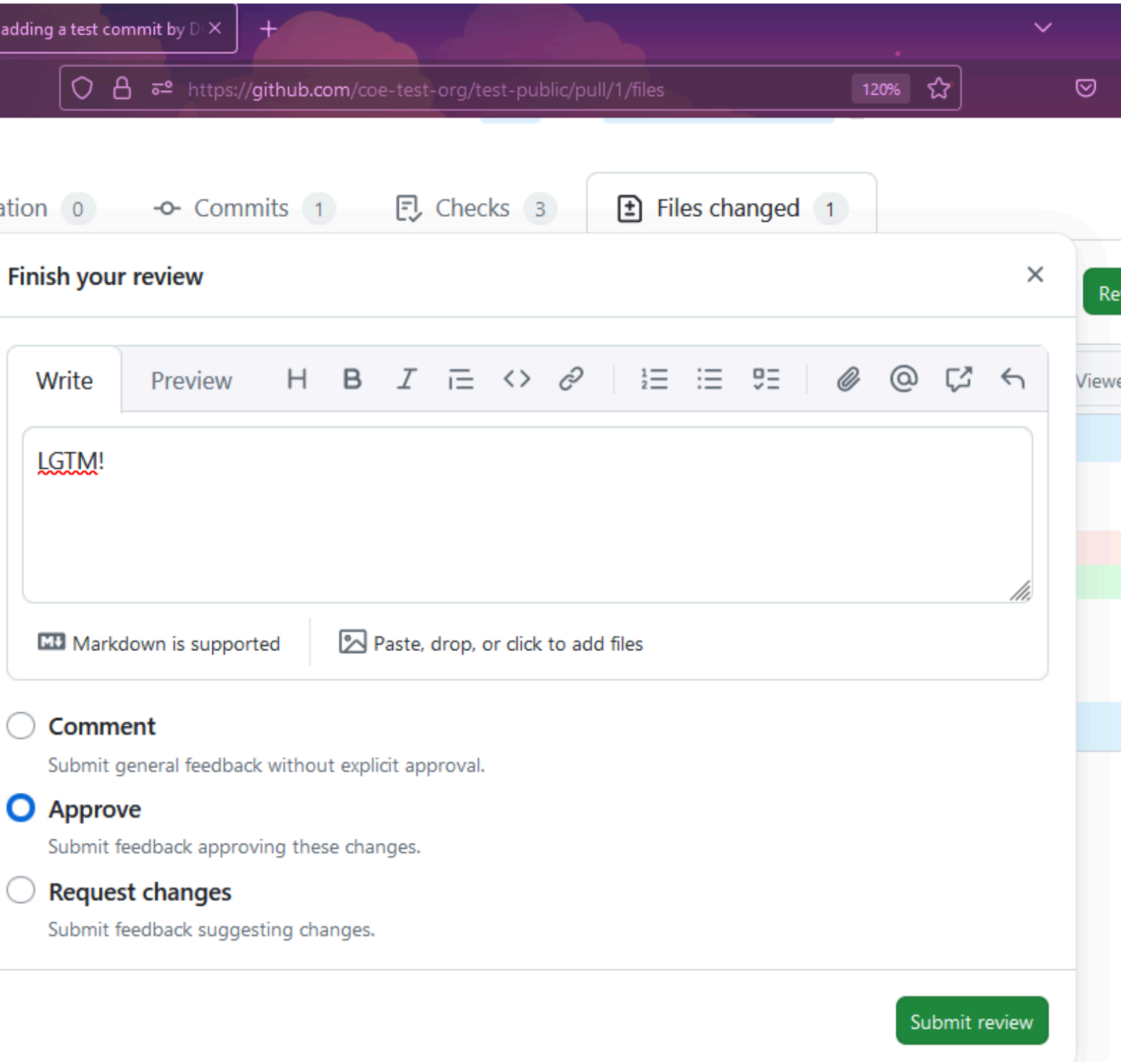



Figure 3 — Review changes

You can leave inline comments in the commits by viewing the file of choice, and then hovering over the line of interest and clicking the `+` sign:

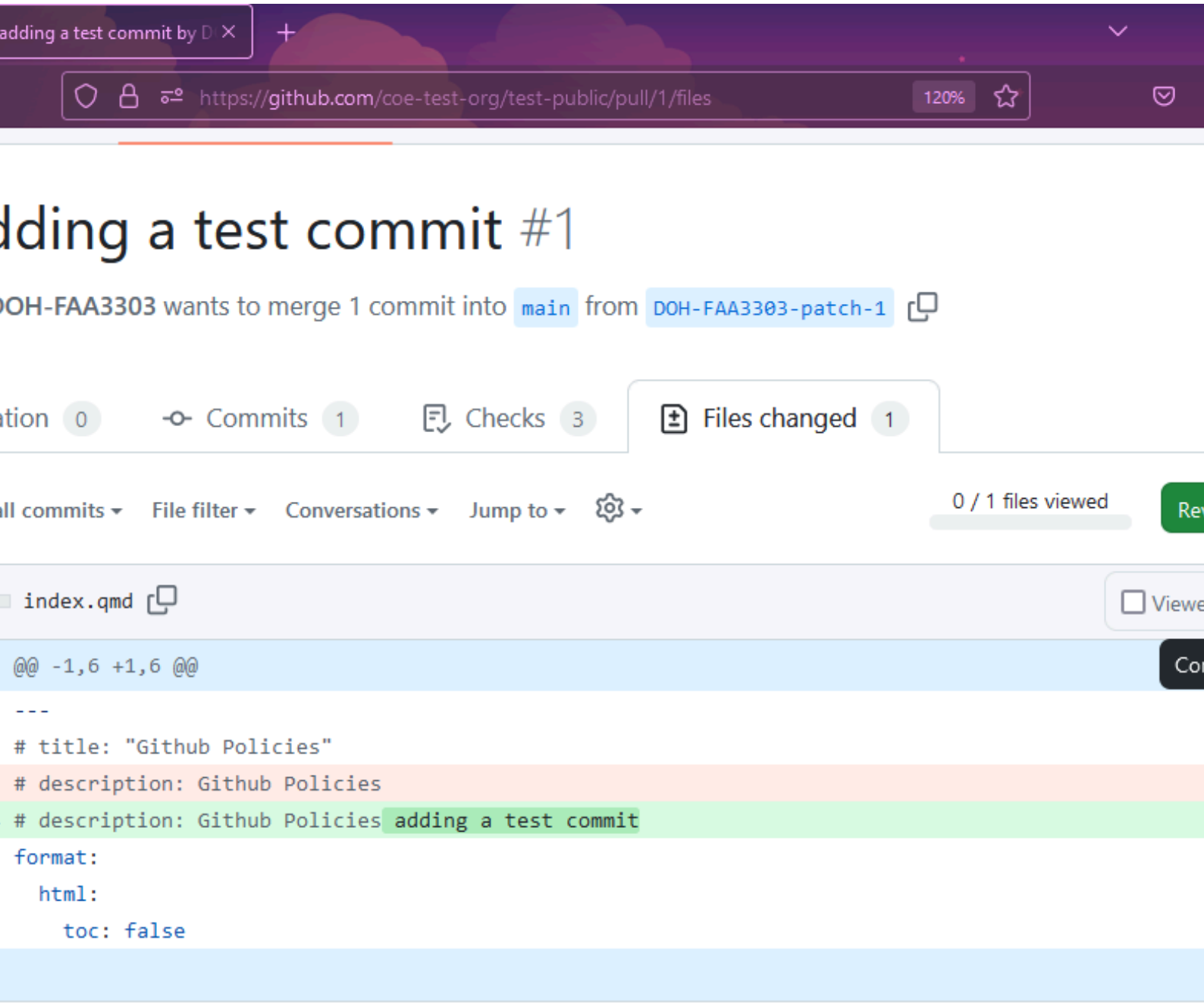



Figure 4 — view file

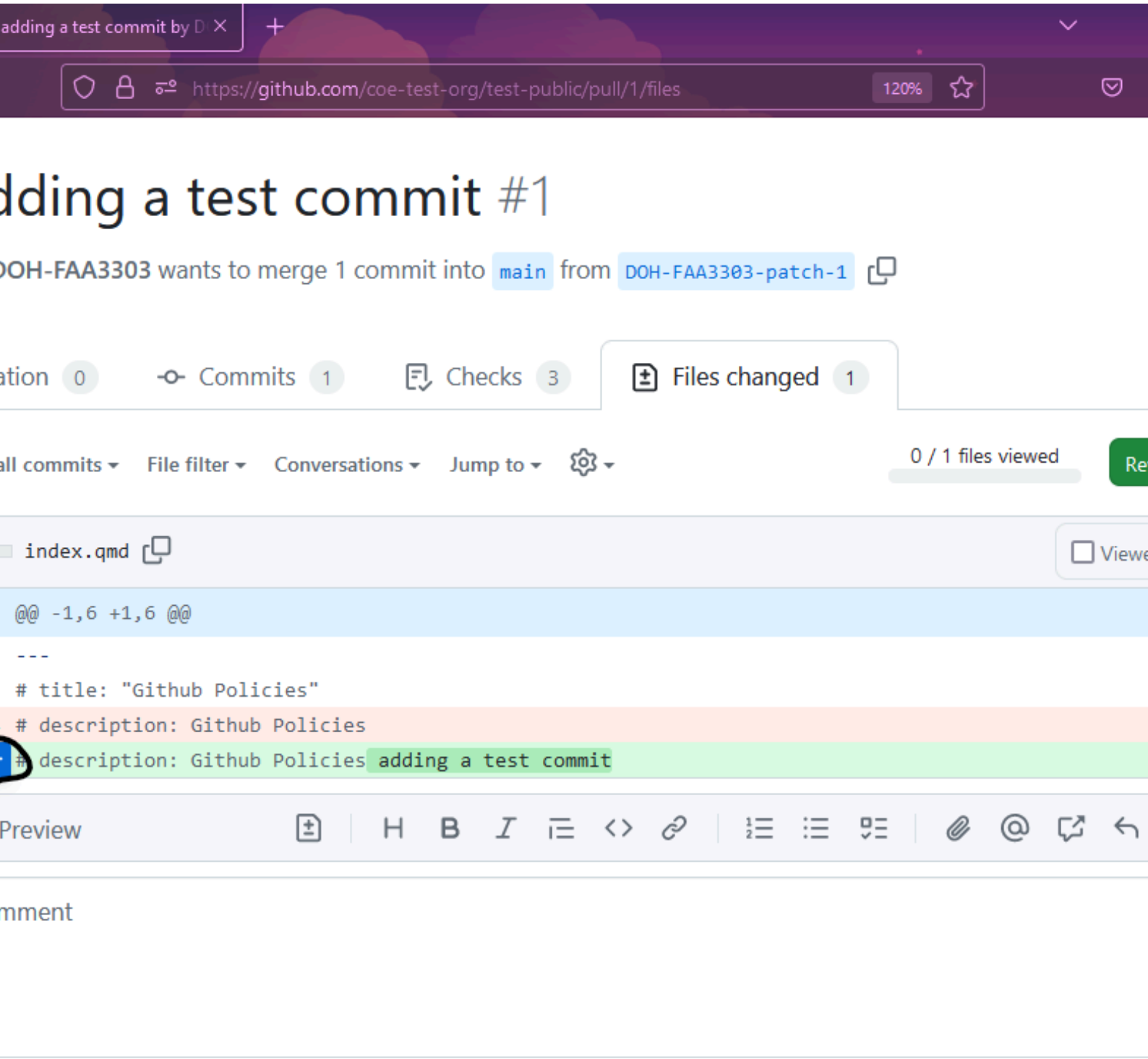



Figure 5 — inline comment

These comments will be tagged in the pull request and will need to be resolved by the person making the request before the code can get merged into the main branch.

Once the pull request has the approvals needed, you can merge it. Note, admins can customize how approvals work. We normally just have one admin or person

required to approve a pull request for it to be mergeable. To state again, in the `Review changes` dropdown there is an option to approve the request - that is what is required.

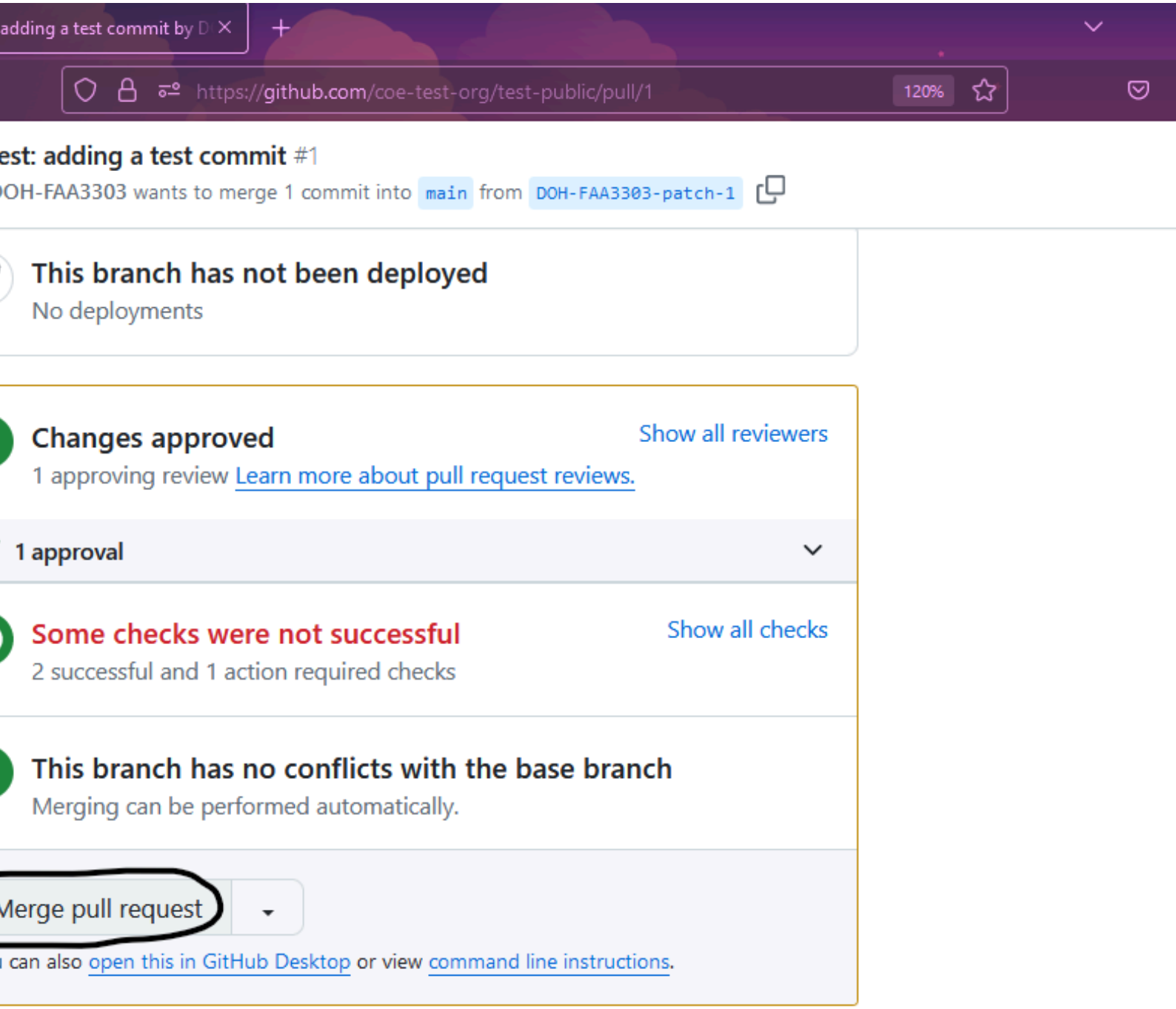



Figure 6 — merge

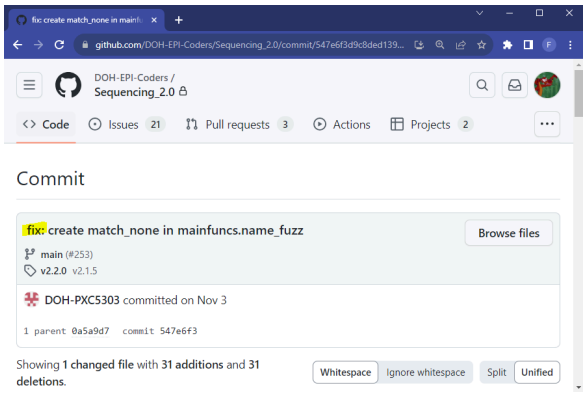
After approval, hopefully you have a message that says *This branch has no conflicts with the base branch*. If there are conflicts it will prevent you from merging. We require the user who made the commit to resolve merge conflicts. The conflict typically happens when your branch is out of date and it is not lined up with the current main branch. You sometimes need to merge the main branch into your

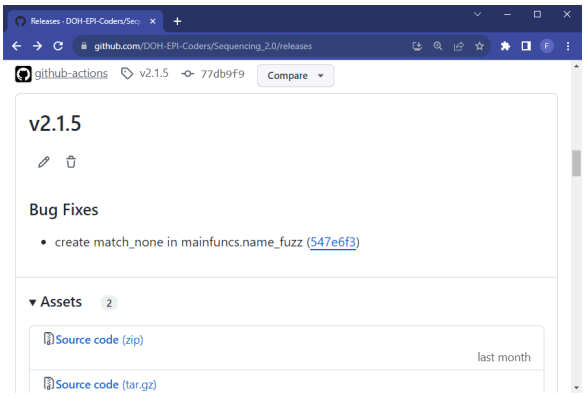
local branch and then commit those changes. This is very scenario dependent and requires some googling. Please reach out to us for help if this happens!

## 8.5 Release Cycles and Changelogs

As I mentioned before, I used special trigger words like `feat` or `fix` in commit messages. These words indicate that a commit contains a new code feature or a bug fix. They trigger a github action that will produce a changelog, documentation, and a version change in the code based when the branch gets merged to main. Please read more about this topic in release cycles. In summary, there are key words you can add to your commit message that will trigger certain actions.

For example, the word `bug` will trigger the version *patch* number to increase, meaning if your current codebase is on version v1.0.0 it will increase the version to v1.0.1 . Here's what it looks like in Github:



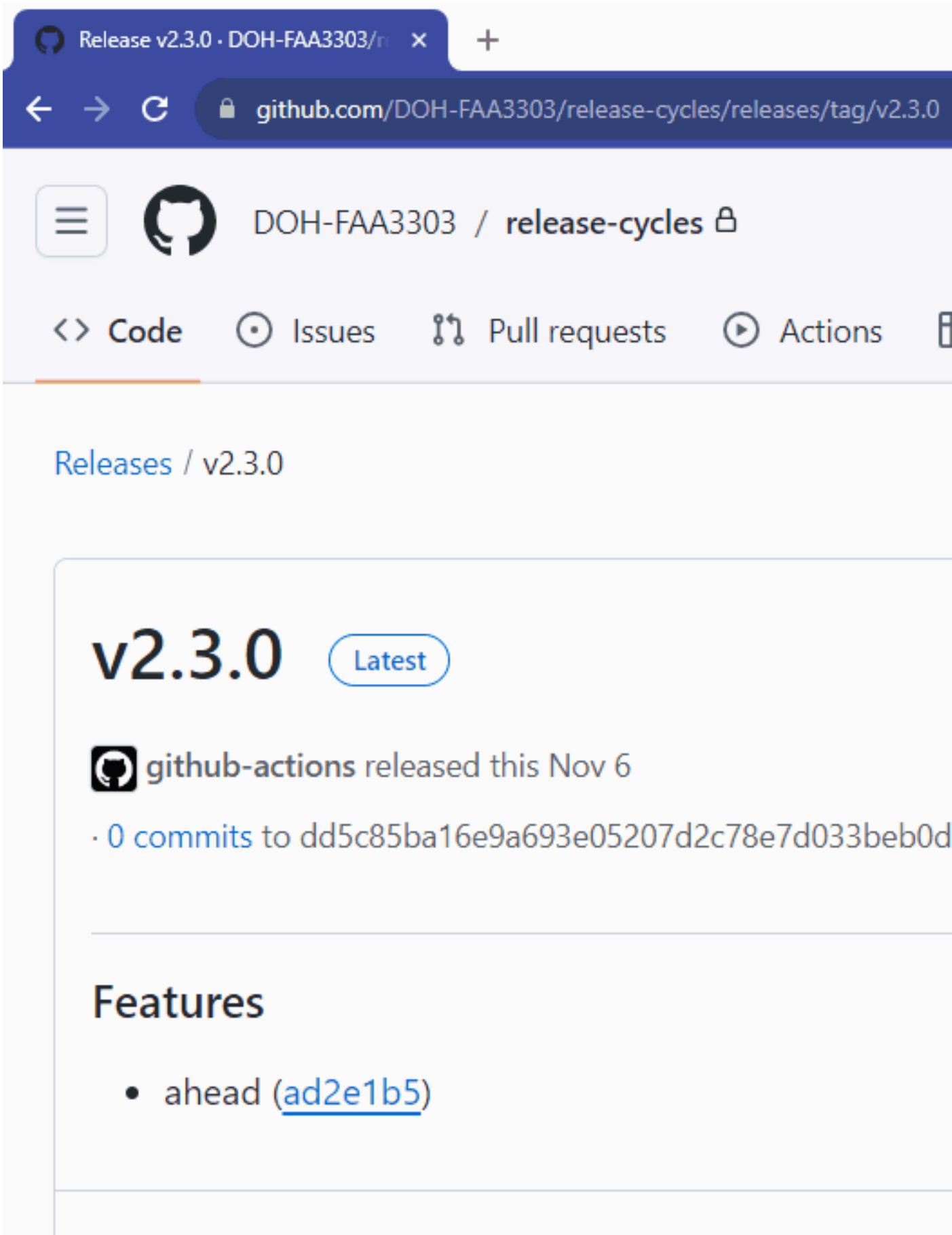

The word `feat` will trigger a change in the minor version, so it will bump v1.0.0 to v1.1.0. There are many other words that can trigger actions and you can customize them to do what you want.

I strongly recommend implementing this in your repo and working in a release cycle. To give an example, our team has a 1 month release cycle:

- week 1: planning out code changes and fixes
- week 2: writing out code and making pull requests
- week 3: reviewing pull requests and testing them
- week 4: preparing communications about the new changes and merging the code to main

Once the code is merged to main, we have a github action that automatically creates our versioning, change logs, documentation, and saves a snapshot of our codebase. The action looks for trigger words (like `fix`, `feat` or

`BREAKING CHANGE` ) and it will divide the commits that were merged into the main branch and write out all the documentation for the change log. It looks like this:


Release v2.3.0 · DOH-FAA3303/
github.com/DOH-FAA3303/release-cycles/releases/tag/v2.3.0
DOH-FAA3303 / release-cycles
Code
Issues
Pull requests
Actions
Releases / v2.3.0
v2.3.0
Latest
github-actions released this Nov 6
· 0 commits to dd5c85ba16e9a693e05207d2c78e7d033beb0d
Features
ahead (ad2e1b5)

Notice that I can:

- Use the version numbers as a tag and switch versions of my entire code base. Very useful if something breaks in production and you want to revert to the old code base, and it gives someone the option of downloading a specific version of your package
- I have a summary of all the commits (with links) that were included in that version
- I don't need to manually do anything
- I can send that change log to leadership and show a high level view of the changes made, with the option to see granular details if wanted
- It documents all of you conversations and changes (+1 for transparency! )

For more details, follow the release cycles guide

## 8.6 Github project management

Milestones, projects, labels, etc.

# 9 Security

**Objectives**

- Prevent sensitive information leaks to Github
- Set up guardrails, `.gitignore` , hooks
- Scrub private repos before they go public

If sensitive information is leaked and commited to the remote repo, then they will stay in the git history (and will require a lot of effort to remove them from the history). The following cannot be included in any repo **or any local commit!**:

| Type | Examples |
|---|---|
| File Paths | • Network drives<br>• Shared internal drives |
| Server Names | • ODBC Connections |
| Credentials | • SSH Keys<br>• Tokens (REDCap, Azure, Github, etc)<br>• Usernames<br>• Passwords<br>• Blob/bucket keys |
| Identifiable Information | • Addresses<br>• Names<br>• Any PHI |

# 10 Protect Credentials with .gitignore

It is bad practice and a security risk to add private credentials to a script. If your script contains things like passwords, server names, or network drives, be aware that that information will be publicly visible when you push it to a remote git/ github repo. Many of our scripts must call passwords and server names in order for them to work properly, so we need a way to hide that information from the public but still be able to run the scripts locally. `.gitignore` can help achieve this.

In the root of your repo there should be a file called `.gitignore`. This files contains a list of file types that we don't want to be added to the remote git repo. Here's an example:

consider this `.gitignore`:

```
# excel files
*.xlsx
*.xls

# logs
*.log

# text files
*.txt
```

```
# RDS Objects
*.RDS
```

It contains anything with an excel, log, txt or RDS extension. This means that any file with those extensions that you create in your local clone of the repo will exist for you, but the file cannot and will not ever be pushed to the remote repo.

## 10.1 Environment Variables + .gitignore

We can store private information in local files and make sure that the do not get pushed to the public remote repo by using `.gitignore`. There are a number of ways to do this. We typically use a yaml file that can be filled out with personal credentials locally. The file will not be committed to the remote repo.

There are many ways to achieve this. If you have a more simple workflow that uses R, consider the `.Renviron` approach. If you have a more complex workflow that has multiple languages and many credentials, consider the `yaml` approach

### 10.1.a .Renviron

If you're using just R in your repo and have just a few things you want private, consider using a `.Renviron` file in addition to `.gitignore`.

1. In the `.gitignore`, add `.Renviron`:

   ```
   # R Environment Variables
   .Renviron
   ```

2. Create a `.Renviron` file at the root of you local repo
3. Add the things you want to be kept private

   ```
   my_password="thisismypassword123"
   ```

4. Now in an R script you can call that password *and* hide the credentials instead of writting the password in the script for everyone to see:

```
my_password <- Sys.getenv('my_password')
```

```
my_password <- "thisismypassword123"
```

`Sys.getenv()` looks for the `.Renviron` file and the variables inside of it. This means you can get all your credentials from the `.Renviron` but also keep that information

### 10.1.b yaml

Here's another way to add credentials that may be more robust.

Many of our scripts use a `.yml` file that contains a list of API tokens, server names, and usernames/passwords specific to each individual user. There are two `.yml` files. One is a template (containing no actual passwords..) that exists in the repo and serves as a template so every individual user can keep up to date with new credential additions. The other is the individual `creds.yml` that is in the repo's `.gitignore`. This file will never exist in the repo and only exist locally (in the user's C drive).

The `.yml` file can work with multiple programming languages including R and Python. They are read in the same way and can be easily adjusted when adding new passwords or using them as configuration files. It can work like this:

1. In your `.gitignore`, add a new line that says `creds.yml`.

```
# creds files
creds.yml
```

2. In the root of you local git clone, make a file called `creds.yml`.
3. In the yaml file you can nest values. For example, under `conn_list_wdrs` I have all the parameters needed to make a SQL server connection string in R/Python:

```
# Default is needed to distinguish values.
# Leave a blank line (NO SPACES) as the last line in this file or
things will break
# Quotes aren't necessary, but can be used.
default:
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server: "someservername"
    Database: "db"
    Trusted_connection: "yes"
    ApplicationIntent: "readonly"

  fulgent:
```

```
    username: <USERNAME>
    password: <PASSWORD>
```

4. To call these credentials in R or Python it will look like this:

```r
library(yaml)

# read in the local credentials yaml file
creds <- yaml::read_yaml("creds.yml")$default

# call in the variables

connection <- DBI::dbConnect(
  odbc::odbc(),
  Driver = creds$conn_list_wdrs$Driver,
  Server = creds$conn_list_wdrs$Server,
  Database = creds$conn_list_wdrs$Database,
  Trusted_connection = creds$conn_list_wdrs$Trusted_connection,
  ApplicationIntent = creds$conn_list_wdrs$ApplicationIntent
)
```

```python
import yaml

# read credentials
with open(f"creds.yml") as f:
    creds = yaml.safe_load(f)['default']

conn = pyodbc.connect(
  DRIVER=creds['conn_list_wdrs']['Driver'],
  SERVER=creds['conn_list_wdrs']['Server'],
  DATABASE=creds['conn_list_wdrs']['Database'],
                      Trusted_Connection=creds['conn_list_wdrs']
['Trusted_connection'],
  ApplicationIntent=creds['conn_list_wdrs']['ApplicationIntent']
  )
```

5. You can add more nested sections besides default, like this, where I added a `test` parameter:

```
# Default is needed to distinguish values.
# Leave a blank line (NO SPACES) as the last line in this file or
things will break
```

```
# Quotes aren't necessary, but can be used.
default:
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server: "someservername"
    Database: "db"
    Trusted_connection: "yes"
    ApplicationIntent: "readonly"

  fulgent:
    username: <USERNAME>
    password: <PASSWORD>

test:
  conn_list_wdrs:
    Driver: "SQL Server Native Client 11.0"
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:
```

This is useful to organized and automatically call different parameters. Now there is a `test` list with its own variables. This lets us switch a set of variables within our scripts. `default` applies to the main credentials where `test` can distinguish which variables should be test or dev scripts specific. Notice below that you can now call the credentials from a `.yml` file into an R or Python script and the actual credentials will never exist in the code pushed to the repo.

```
# this script is in the repo, but credentials are hidden
library(yaml)

# read in the local credentials yaml file
creds <- yaml::read_yaml("path/to/local-credentials.yml")

# pull in the credentials
server_name <- creds$default$conn_list_wdrs$server
```

### 10.1.b.a Automating With Yaml Creds

We can even get more specific and add an `if-else` statement to specify which credential we want to select. This can be helpful if we have a CI/CD pipeline and have a script automatically run on a task scheduler or cron job. We can call the credentials we want in the command line and have the command line code

run in my task scheduler. That way we can use multiple different versions of the same script and have all of it be automated.

For example,

- the R script on the left uses the `commandArgs()` to pull any arguments passed to the script in a shell/command line script.
- on the right, the shell script has `production` and `test` as second arguments.
- these are passed to the R script as `arg[2]` .
- now we can use `arg[2]` in the if-else statement to conditionally select credentials and do it automatically in a pipeline.

```
args <- commandArgs(TRUE)

# this script is in the repo, but credentials are hidden
library(yaml)

# read in the local credentials yaml file
creds <- yaml::read_yaml("path/to/local-credentials.yml")

# pull in the credentials
if(args[2] == "production"){
  server_name <- creds$default$conn_list_wdrs$server
} else if(args[2] == "test"){
  server_name <- creds$test$conn_list_wdrs$server
}
```

```
# Run the production code
$ Rscript -e "source('path/script_in_repo.R');" production

# Run the test/dev code
$ Rscript -e "source('path/script_in_repo.R');" test
```

### 10.1.b.b yaml Template

You can put a template creds.yml file in your repo so that others can see what credentials they need in order for the code to run.

**This is a *template* file, so it will not have any passwords/secrets in it**. Its only purpose is to provide an example copy of what a user's `creds.yml` file needs to look like.

1. Make a template called `creds_TEMPLATE.yml`

2. Remove any passwords, usernames, secrets, etc to have it be a file that looks like this:

```
# Default is needed to distinguish values.
# Leave a blank line (NO SPACES) as the last line in this file or
things will break
# Quotes aren't necessary, but can be used.
default:
  conn_list_wdrs:
    Driver:
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:

  fulgent:
    username:
    password:

test:
  conn_list_wdrs:
    Driver:
    Server:
    Database:
    Trusted_connection:
    ApplicationIntent:
```

3. Once you have the `creds_TEMPLATE.yml` template in your repo, make sure that nobody on your team (or anyone with write access..) is able to accidentally push changes to the template. We don't want someone's passwords or API tokens to exist in GitHub.

This link shows how to skip any changes made to the specific file https://stackoverflow.com/a/39776107. If someone makes local changes to the template, the changes will not show in their commit. It is a safe guard.

4. For all individual users, run this code:

```
git update-index --skip-worktree creds_TEMPLATE.yml
```

This will tell your local git to ignore any changes made to `creds_TEMPLATE.yml`, but also allow it to exist in the repo (since `.gitignore` will prevent it from being in the repo)

5. If you need to update the template file run this:

```
git update-index --no-skip-worktree creds_TEMPLATE.yml
```

This will allow changes to the template. **So when you need to update the template, use this code**

And to get a list of files that are "skipped", use this code:

```
git ls-files -v . | grep ^S
```

# 11 Security Guardrails

Using a `.gitignore` file for environmental variables/credentials is an excellent guardrail and promotes good coding habits, but we may also want additional guardrails such as hooks.

Hooks are processes that run in the background and can prevent code from being pushed if there is a security flaw. There are two hooks we could use for security; pre-commit hooks and pre-receive hooks

# 12 Pre-commit Hooks

Pre-commit hooks run a process locally when the user attempts to commit code to a git branch. Hooks have many uses. Here we can use them as a security guardrail to prevent accidental credential leaks in committed code. For example, if someone accidentally pushes a server name to the public repo, the hook will prevent that code from ever getting into the remote repo and will give the user a local error.

Follow the instructions below to set up pre-commit hooks *for all of your repos*. You can set up different hooks for individual repos, but I recommend setting this up globally so that all of your local clones are covered by the security hook.

This document goes through setting up git hooks in more detail

## 12.1 Windows

1. Clone or download the zip from the AWS Git Secrets repo
2. Extract zip or `cd` to the repo
3. Open folder and right click install.ps1.

a. Run in Power Shell
b. Type Y to give permission

Alternatively, in the powershel terminal you can change directories `cd` to the repo and `.\install.ps1`

4. Make a directory for a global hook template

```
mkdir ~/.git-template
```

5. Run git secrets –install

```
git secrets --install ~/.git-template
```

6. Configure git to use that template for all your repos

::: smallframe ::: {.cell filename='PowerShell'}

```
git config --global init.templateDir '~/.git-template'
```

::: :::

7. Make or copy the regex file called `secrets_key` containing the secret patterns **outside of your git repos**.
   - This file should be given to you by the Github admins. It contains a regex of potential secrets. Contact `frank.aragona@doh.wa.gov` for more information.
8. Make sure the file `secrets_key` is in your `.gitignore` . We can't push that to the remote repo.
9. Run `git secrets --add-provider -- cat ./secrets_key`

```
git secrets --add-provider --global -- cat ./secrets_key
```

You can also add prohibited patterns like this

```
# add a pattern
git secrets --add '[A-Z0-9]{20}'

# add a literal string, the + is escaped
git secrets --add --literal 'foo+bar'
```

```
# add an allowed pattern
git secrets --add -a 'allowed pattern'
```

10. Test Git history by running

````
::: smallframe
::: {.cell filename='PowerShell'}

```{.bash .cell-code}
git secrets --scan-history
```

:::
:::
````

11. If something gets flagged and you don't care about your history anymore: Delete .git folder and reinitialize repository
    - I would take caution about this point. There might be better ways to clean your git history if you don't want to get rid of everything.
12. Test on one of my projects to see if rebasing is a sustainable option
13. Make repo public
14. Will automatically scan on every commit and won't let it commit unless it's clean - Create a few files to show it working

> **Note**
>
> We can't use the "Non capture group" feature of regex. Meaning we can't use patterns like this in our regex: (?:abc) – see https://regexr.com IMPORTANT: Tab separate your regex expressions. Making new lines caused a bit of chaos and took really long to figure out. (you can use multiple tabs to separate them more visually)

## 12.2 WSL/Linux

1. Clone the AWS Git Secrets repo
2. In the terminal, `cd` to the repo
3. Install the command:

```
sudo make install
```

4. You may need to add this file to your $PATH variables.
    - run `nano .bashrc` to get your bash profile:

      ```
      nano .bashrc
      ```

    - then down arrow key to get to the last line in the file
    - add the path like this:

      ```
      export PATH=$PATH:/user/local/bin/git-secrets\
      ```

    - hit `CTRL + O` then `ENTER` to save
    - hit `CTRL + X` to exit
    - start a new terminal and write this to see your path variables.
    - `git-secrets` should be in there somewhere now

      ```
      echo $PATH
      ```

5. Make a git template directory

   ```
   mkdir ~/.git-template
   ```

6. Install the hooks globally into that template

   ```
   git secrets --install ~/.git-template
   ```

7. Configure git to use that template globally

   ```
   git config --global init.templateDir '~/git-template'
   ```

8. Make or copy the regex file called `secrets_key` containing the secret patterns into your folder.
    - This file should be given to you by the Github admins. It contains a regex of potential secrets. Contact `frank.aragona@doh.wa.gov` for more information.
9. Make sure the file `secrets_key` is in your `.gitignore`. We can't push that to the remote repo.
10. Run `git secrets --add-provider -- cat ./secrets_key`

````
::: smallframe
::: {.cell filename='terminal'}

```{.bash .cell-code}
git secrets --add-provider --global -- cat ./secrets_key
```
:::
:::
````

You can also add prohibited patterns like this

```
# add a pattern
git secrets --add '[A-Z0-9]{20}'

# add a literal string, the + is escaped
git secrets --add --literal 'foo+bar'

# add an allowed pattern
git secrets --add -a 'allowed pattern'
```

11. Test Git history by running

    ```
    git secrets --scan-history
    ```

12. If something gets flagged and you don't care about your history anymore: Delete .git folder and reinitialize repository
    - I would take caution about this point. There might be better ways to clean your git history if you don't want to get rid of everything.
13. Test on one of my projects to see if rebasing is a sustainable option
14. Make repo public
15. Will automatically scan on every commit and won't let it commit unless it's clean - Create a few files to show it working

> **Note**
>
> We can't use the "Non capture group" feature of regex. Meaning we can't use patterns like this in our regex: (?:abc) – see https://regexr.com IMPORTANT: Tab separate your regex expressions. Making new lines caused a bit of chaos and took really long to figure out. (you can use multiple tabs to separate them more visually)

**NOTE!!**

- The REGEX strings used in the `secrets_key` file may be decieving
- Make sure to test that the regex flags what you want it to
- `git secrets --scan-history` may take a very long time to run
- Follow the Secret Scanning instructions below for more help

# 13 Secret Scanning

Now that the pre-commit hook is set up, any future commits to your repo will be scanned for secrets. If you are pushing a pre-existing repo to a public repo for the first time, you should scan the existing code in the repo because the pre-commit hook will not automatically do that. They are really set up to prevent any *future* secrets from being pushed to the repo, not to scan what is *currently* in the repo.

There are a few ways to scan the history of your repo for secrets. The `git secrets` command comes with a few options to scan the history, but I have found that it is a bit broken.

- the `git secrets --scan-history` command will run forever if you have a large repo (especially if you have html files in it)
- globs have not worked for me (specifying the file types you want to scan for `git secrets --scan *glob` )
- likewise, scanning specific folders have not worked for me like this `git secrets --scan directory/*`

## 13.1 Instructions

1. Check that the `secrets_key` regex is working by running the process on a repo that you know has secrets in it. For example, in a different folder, run all the pre-commit hook steps above and add a known "bad" string into the regex. For example, in the regex put `bad_string` and in a file in that folder put `bad_string` . When you scan it should get flagged.

2. If secret scanning is taking too long, you might want to check certain files first. I've found that HTML files take a *very* long time to scan for secrets.

Follow the instructions below to scan for specific files. The script will scan for all the file types that you select. For example, if you want to only scan R files, it will only scan R files.

### 13.1.a Windows

1. In PowerShell, navigate to your repo and paste this code:

```
# Example Usage
# write this in the powershell terminal, adjust for the file type(s)
you want to scan - can be multiple types: $fileExtensions = @(".R",
".py")
# then execute this in the terminal: ScanFiles -FileExtensions
$fileExtensions

# It will give you an output of any secrets that are contained in
those files

Function ScanFiles{
  param (
      [string]$filePath = (Get-Location).Path,
      [string[]]$fileExtensions
)
  Get-ChildItem $filePath -recurse | Where-Object {$_.extension -
in $fileExtensions} |
  Foreach-Object {

      git secrets --scan $_.FullName

  }
}
```

2. Write the file extensions you want to scan for in a PowerShell Terminal window like this:

```
$fileExtensions = @(".R",".py",".Rmd",".qmd")
```

3. Now, you can scan your secrets by copying and pasting this code into PowerShell:

```
ScanFiles -FileExtensions $fileExtensions
```

### 13.1.b WSL/Linux

1. In a bash/Ubuntu terminal, navigate to your repo and paste this code:

```
find . -type f \( -name "*.R" -o -name "*.py" -o -name "*.qmd" -
o -name "*.rmd" -o -name "*.md" \) -print0 | xargs -0 -I {} git
secrets --scan {}
```

2. This is set to scan all R, Python, QMD, RMD, or MD files. If you want to add another file type, do it like this where you add `-o -name "*.NEW_TYPE"` to the `find` command args:

```
find . -type f \( -name "*.R" -o -name "*.py" -o -name "*.qmd" -o
-name "*.rmd" -o -name "*.md" -o -name "*.NEW_TYPE"  \) -print0 |
xargs -0 -I {} git secrets --scan {}
```

# 14 Pre-Recieve Hooks

These are still being investigated. They are remote hooks (not local like pre-commit hooks) that can be deployed throughout the Github organization. They can block certain commits from ever being pushed to the remote repo. They may make things unnecessarily complicated

# 15 Pushing Private Code to Public Repos

We may wish to take private codes and push them to a public repo. We need to make sure that the public code doesn't not contain sensitive or forbidden data/code, so cleaning up the private repo is important before pushing.

There are a few ways to do this, but the easiest way is to copy the clean private code to the public repo, that is, copy all the files you want to add publicly but **do not copy the `.git` folder**. If the private repo has a dirty git history we will not want that history in the public repo because the sensitive data will then be publicly available.

The private repository on the left still contains sensitive information in the git history. The public repository on the right has a clean git history because we copied only the current clean files from the private repo and did not attach its git history (which lives in the hidden `.git` folder)

# 16 Code Reviewers/Github Operations Team

With the guardrails above in place there should be few chances that credentials get pushed to a repo. However accidents may still happen. We want to make sure that anyone who opens up a repo in the Github organization adheres to the rules, has the proper credential/coding set-up, and installs their local pre-commit hooks properly.

It may be useful to have a team within the organization that helps with repo set-up. The team would help avoid a scenario where a person opens up a repo without reading this documentation and understanding the rules (and thus potentially breaking security rules).

This Github Operations Team could also be helpful in managing permissions for members in the organization. See the video below on how the company Qualcomm manages their Github organization https://www.youtube.com/embed/1T4HAPBFbb0?si=YRsUYXIxLPhdr41T and how they use a Github Operations Team to guide new members access/repo development

https://www.youtube.com/embed/1T4HAPBFbb0?si=YRsUYXIxLPhdr41T

# 17 Licensing

**Summary**

- Licenses prevent code theft and inappropriate redistribution of code.
- Review common open-source licenses
- License types vary depending on repo goals

# 18 General License Info

Below is a list of common open-source licenses.

There isn't a one size fits all license, so thankfully there are a variety of options. Here are two common ones:

# 19 GNU GPL licenses

a. These are the strong licenses
b. Prevents someone from taking our code and privatizing it (and making money off of it)

c. Someone can still use our code, they just need to ensure that what they're doing with it is open-source
d. "Copyright and license notices must be preserved."
e. "Contributors provide an express grant of patent rights. When a modified version is used to provide a service over a network, the complete source code of the modified version must be made available."

# 20 MIT license

a. I think this is the most commonly used one
b. "short and simple permissive license… only requiring preservation of copyright and license notices"
c. "Licensed works, modifications, and larger works may be distributed under different terms and without source code."
d. Someone could basically do whatever they want with the code.
e. Nextstain/ncov repo is currently using this

And here are a couple of youtube videos that were helping in explaining licensing

https://www.youtube.com/embed/rbQg9DY_4y0?si=OvU9vLBHX43dTIcA

https://www.youtube.com/embed/ndORMSnb2nw?si=tkUzjwZYWKfrLTEU

# 21 Policies

**Objectives**

- Ensure that all repos in the org have the required documents
- Set policy rules at the Organization level
- Repos need to have reproducible code
- Repos need to have documentation

In the Github Organization we may require all repositories to contain certain documents. For example, we want to make sure that every repo has a `CODE OF CONDUCT` document that is a general policy applied throughout the organization.

Here's a list of required documents:

`README`

`README` files are instructions or documentation on how to use your software. It should give a quick introduction to the repo and instructions on how to install or run the code.

### `CODE_OF_CONDUCT`

A Code of Conduct can let a user know what the rules of the organization are and how any wrongful behavior will be addressed. The document will provide the "standards for how to engage in a community"

### `CONTRIBUTING.md`

This file should appear in the issue tab in a repo. It lets a user know how they can contribute to the project and if they need to sign any forms before contributing. Some larger organizations require that a person knows what they are contributing to and they must sign a form acknowledging that any software/code contributions to the project will be used and cannot be retracted by the user. The code submitted may also be used to develop processes but the organization will not pay the individual contributor (since this is open-source, we only look for open-source contributions)

### `LICENSE`

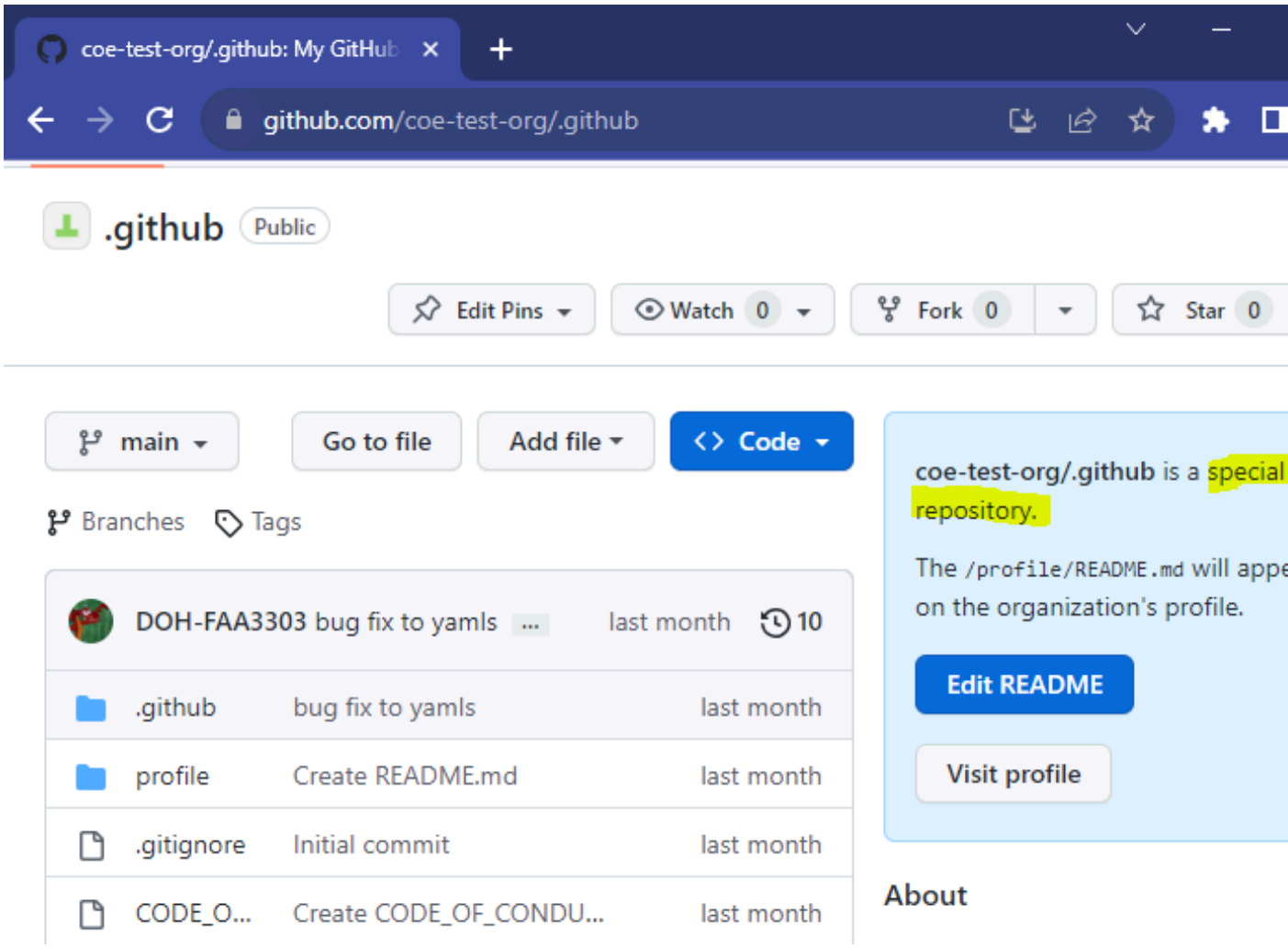
These should be and are set at the repo level. There will be many different licenses to choose from that will depend on the specific repo. More on that here.

# 22 Set Policy Rules at Org Level

Policy rules may include requiring certain documents in each repo or requiring that a person sign every commit.

## 22.1 Document Requirements with `.github` Repos

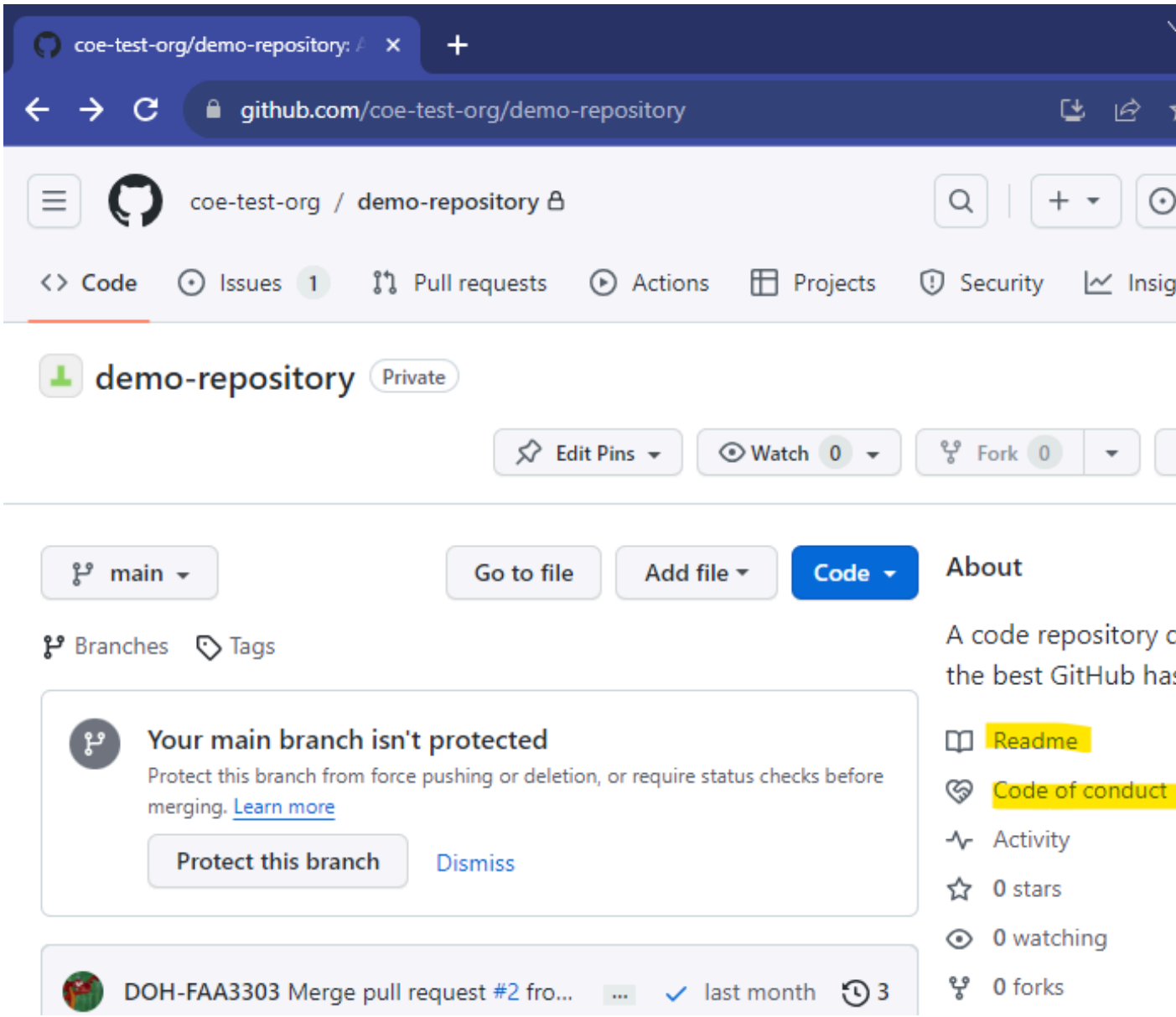
You can set most policy rules and create documents for each repo at the organization level by using a special `.github` repo. Dot files and dot folders have special functionality in some software. For Github, the `.github` folder defines workflows for things like Github Actions in a repo. A `.github` *repository* on the other hand defines *organization level* rules and templates.



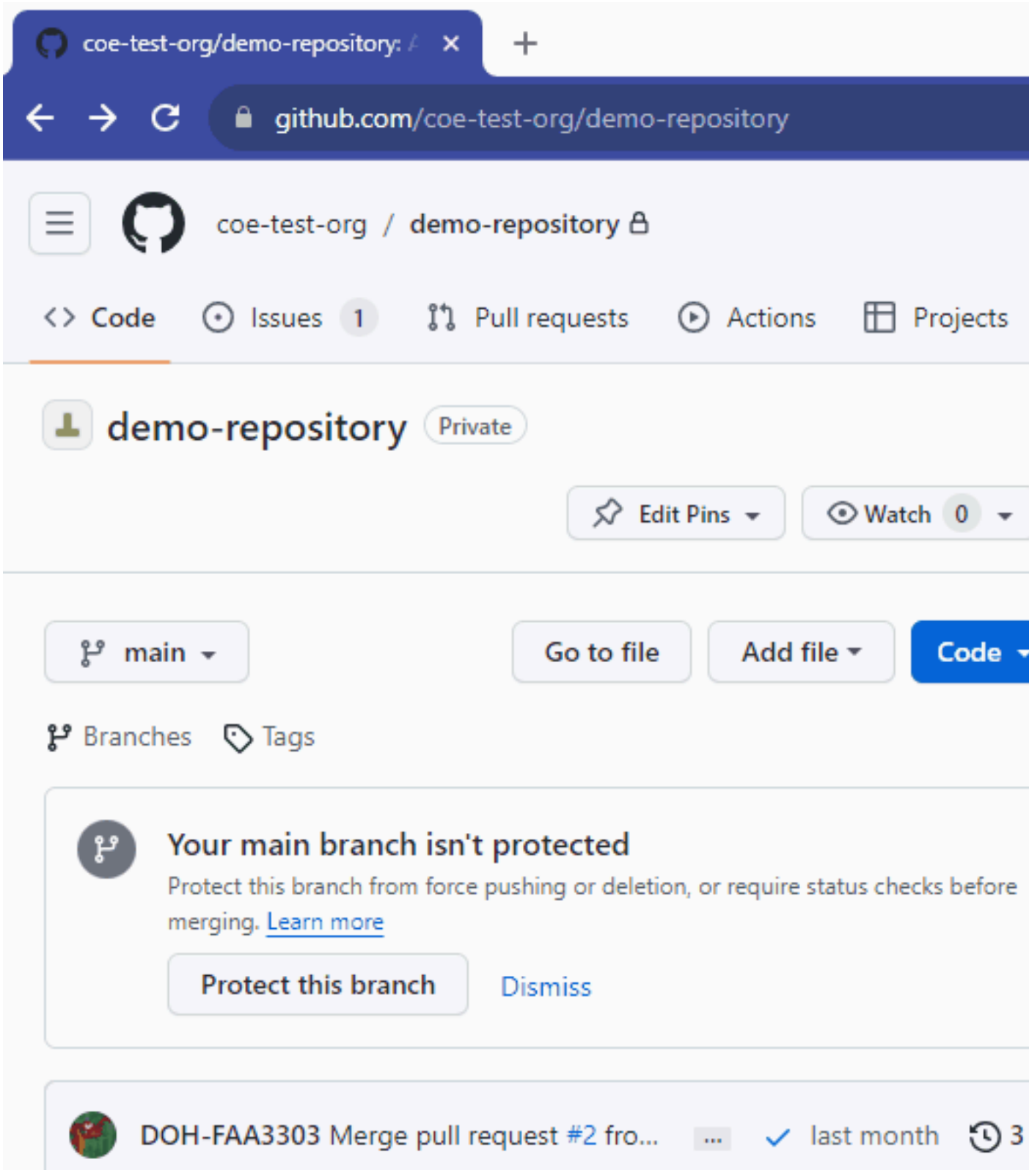

In order to write and set these policies at the organization level we can put them at the root of the `.github` repository and edit them there.

```
$ tree /f
C:.
|   .gitignore
|   CODE_OF_CONDUCT.md
|   CONTRIBUTING.md
|   LICENSE
|   README.md
|
├───.github
└───profile
        README.md
```

Take a look above. I have the required documents/policies at the root of the `.github` repo directory. Now if I open up any given repo in the organization I will find a link to those files:




If you click on the `CODE_OF_CONDUCT` link it will take you right to the `.github` repo and open the `CODE_OF_CONDUCT.md` file there:




Now you can set organization level policies from the `.github` repo and they will automatically populate in *all* existing and new repositories *unless there are repo specific policies in place*. If a repo already has its own policies they will not be overwritten.

# 23 Set Templates at the Org Level

Aside from policy documents, you can make templates at the organization level. Two commonly used templates are issue templates and discussion templates.

In the public repos there may be end users that may have limited experience using Github. If they want to submit an issue or ask a question they get lost. Templates can help them form a question or idea. Templates can also help standardize how issues and discussions are maintained throughout the organization.

Structuring the format of issues and discussions can make the author and the end-user's lives easier.

In the `.github` repo I made a *folder* called `.github`. This is a special folder that can hold Github Action workflows and more, as mentioned above.

In the `.github` *folder* I have a folder called `DISCUSSION_TEMPLATE` and another called `ISSUE_TEMPLATE`. These are special folders that Github recognizes as discussion and issue folders that will set templates at the repo (or in this case the org) level.

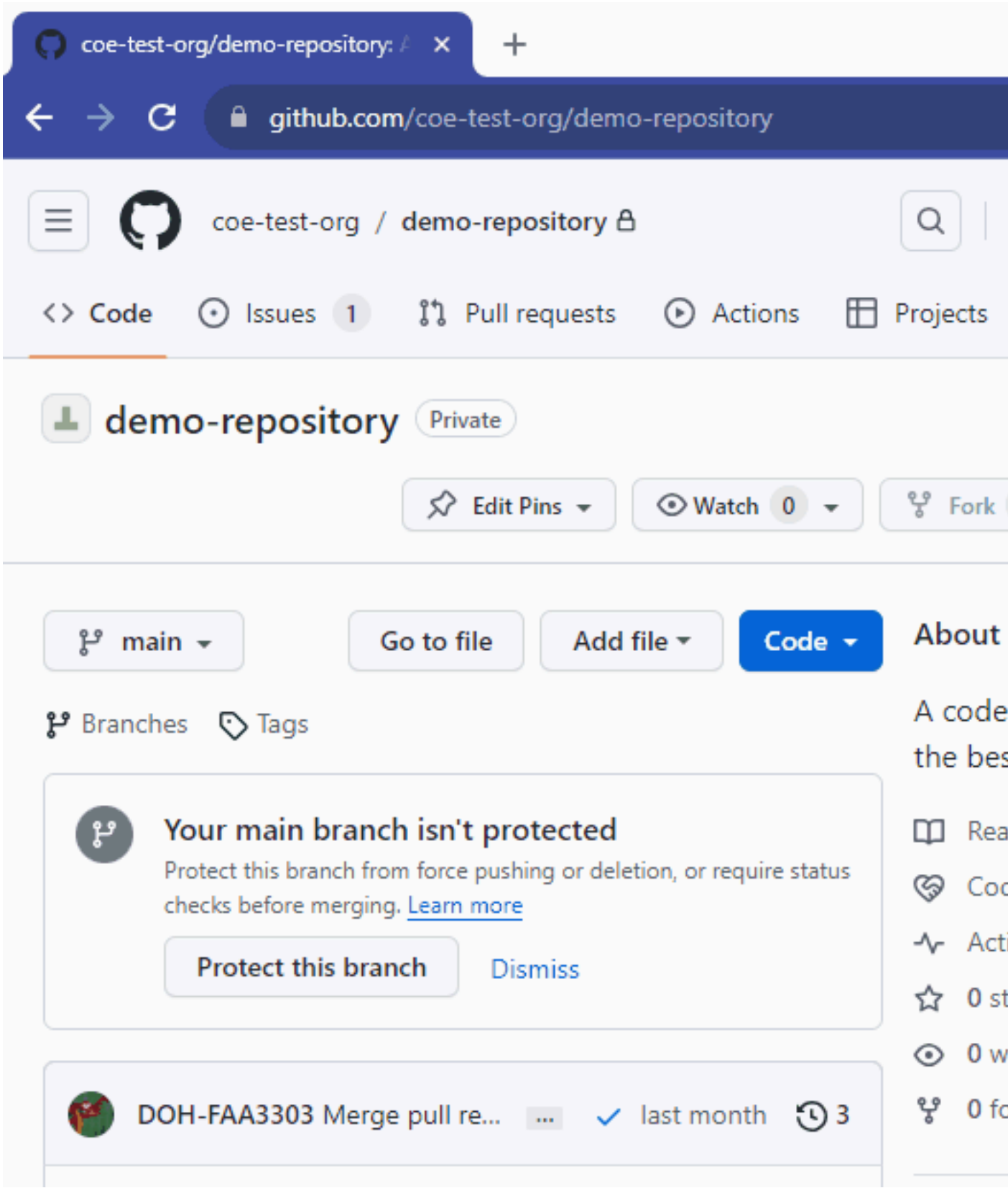
```
$ tree /f
C:.
│   .gitignore
│   CODE_OF_CONDUCT.md
│   CONTRIBUTING.md
│   LICENSE
│   README.md
│
├───.github
│   │   pull_request_template.md
│   │
│   ├───DISCUSSION_TEMPLATE
│   │       feature-requests.yml
│   │       q-a.yml
│   │       show-and-tell.yml
│   │
│   └───ISSUE_TEMPLATE
│           bug_report.yml
│           config.yml
│           feature_request.yml
│
└───profile
        README.md
```

Each Folder has `.yml` files in it that are basically Github instructions on how to format issues and discussions.

For example, in the `ISSUE_TEMPLATE` folder I have a `.yml` file called `bug_report.yml`. This file contains the structure for how someone can report a bug.

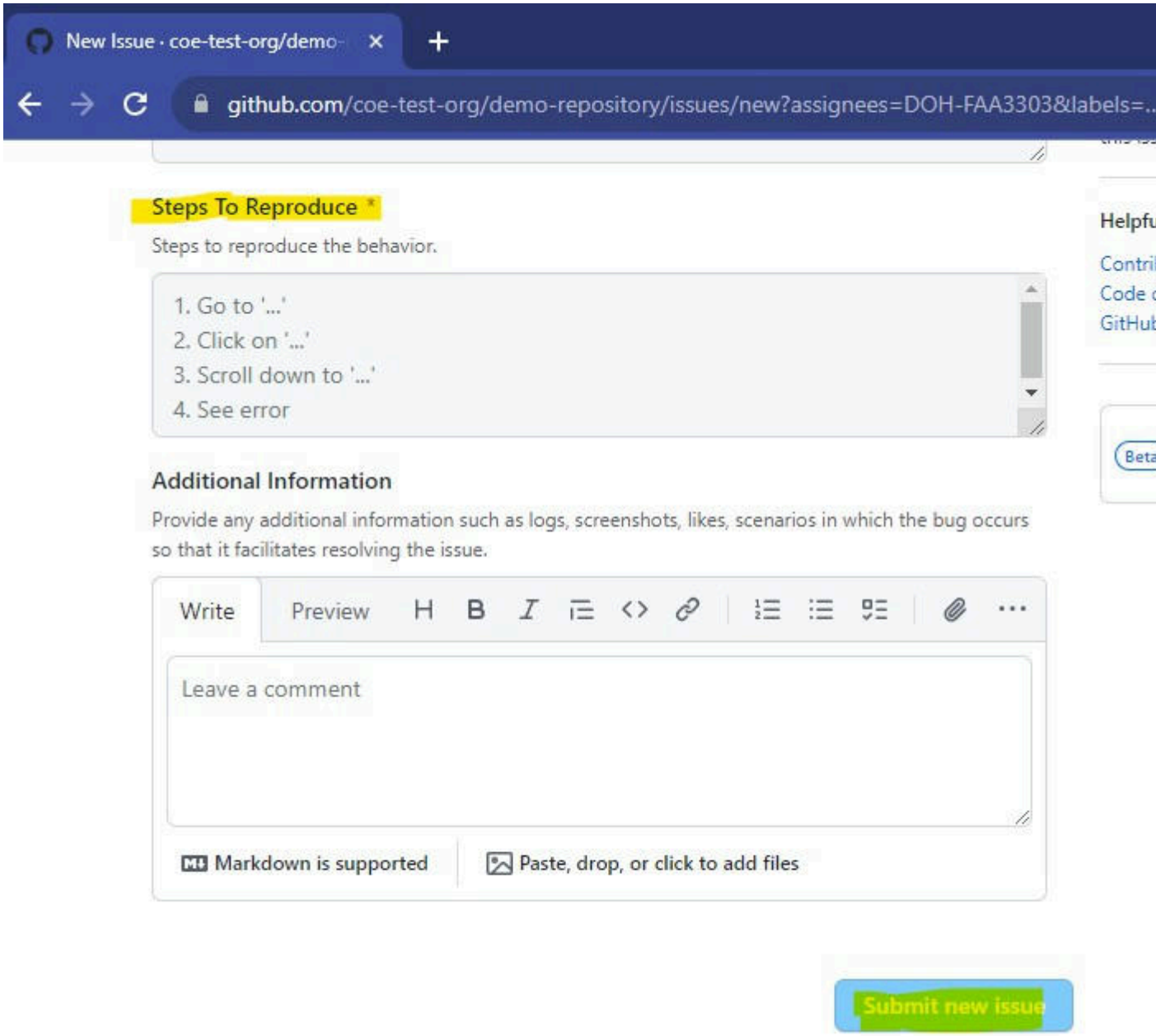
```yaml
name: Bug Report
description: File a bug report here
title: "[BUG]: "
labels: ["bug"]
assignees: ["DOH-FAA3303"]
body:
  type: markdown
    attributes:
      value: |
        Thanks for taking the time to fill out this bug report
       Make sure there aren't any open/closed issues for this topic
```

Now, when someone clicks on the `Issues` tab in a repo in this organization they will be met with the `Bug Report` template:




Notice that in the template you can create text areas and pre-fill those areas with suggestions. You can even require that someone fills out those areas before they can submit the issue:

```
- type: textarea
    id: steps-to-reproduce
    attributes:
      label: Steps To Reproduce
      description: Steps to reproduce the behavior.
      placeholder: |
        1. Go to '...'
        2. Click on '...'
        3. Scroll down to '...'
        4. See error
    validations:
      required: true
```



### 23.1 Commit Sign-Off Requirement - Github Apps

We may want to require authors or reviewers to sign-off on commits to a repo. This is sometimes established in projects to "ensure that copyrighted code not released under an appropriate free software (open source) license is not included in the kernel."

You can install a Github App in the organization and it will be applied to all repos. The DCO App (Developer Certificate of Origin) is popular and lightweight. To install it in the organization, click on Configure and it will give you the option to configure it with the organization of choice.

## 24 IaC

Infrastructure as Code (IaC) can be helpful when managing administration tasks or writing hooks at the org level.

## 25 Reproducibility

**Objectives**

- Data and Code Democratization
- Github Codespaces
- Package reproducibility with virtual environments
- Github Releases
- Documentation

## 26 Data and Code Democratization

Data and code in our repositories need to be accessible to end users and developers. There should be no bottlenecks or difficulties with installing software, executing code, finding documentation, and using test datasets.

The goal is for any user to run code without needing to install anything on their personal machine and run your code with minimal set up. This may not be possible in every scenario, but there are tools available in Github to make this possible for the majority of our repos.

## 27 Github Codespaces

Github Codespaces are virtual machines (VMs) owned by Github that are connected to each repository. They let a user open the repo in a browser IDE (Inte-

grated Development Environment) and execute the code in that environment. There is no set up or installation necessary for them.

The VMs are free for up to 60 hours a month of use and there are more hours added for Github users with paid memberships. 60 hours/month should be plenty for our purposes. Users are responsible for their own Codespace, so if they go over the limit they will be responsible for adding more hours and paying for the service.

## 27.1 Open a Codespace

At the root of the repo, click on the **Code** drop down button

1. On the right there is a tab called Codespaces.
2. Click the + sign and a Codespace will launch

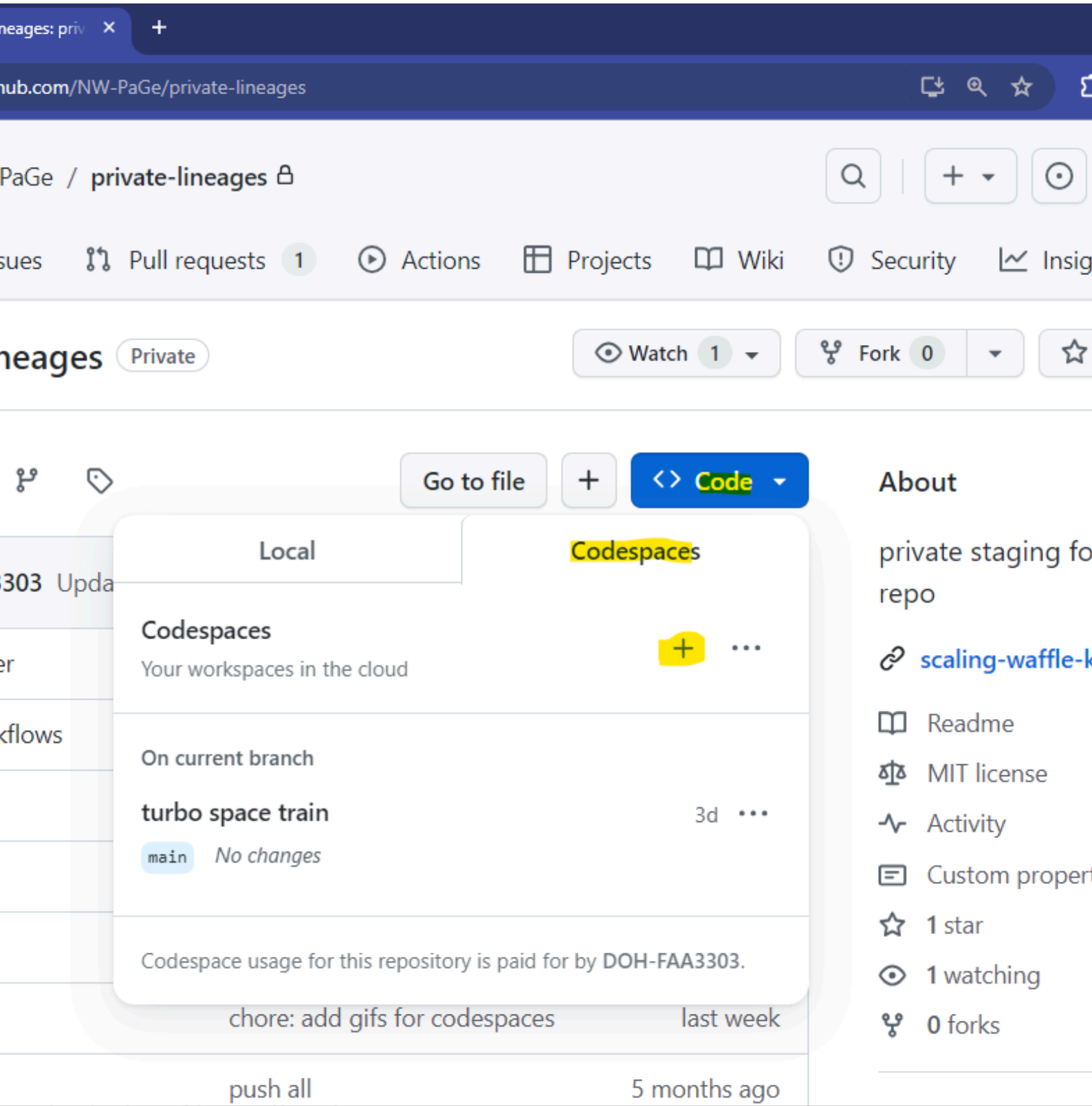



Figure 7 — open up a codespace

This will open up a VS Code window in your browser. There are also options to open up a Jupyter Notebook or Jetbrains IDE (Pycharm). You can also install an

Rstudio IDE into the codespace. It will look something like this - note that the repository is already linked and checked out into the codespace:

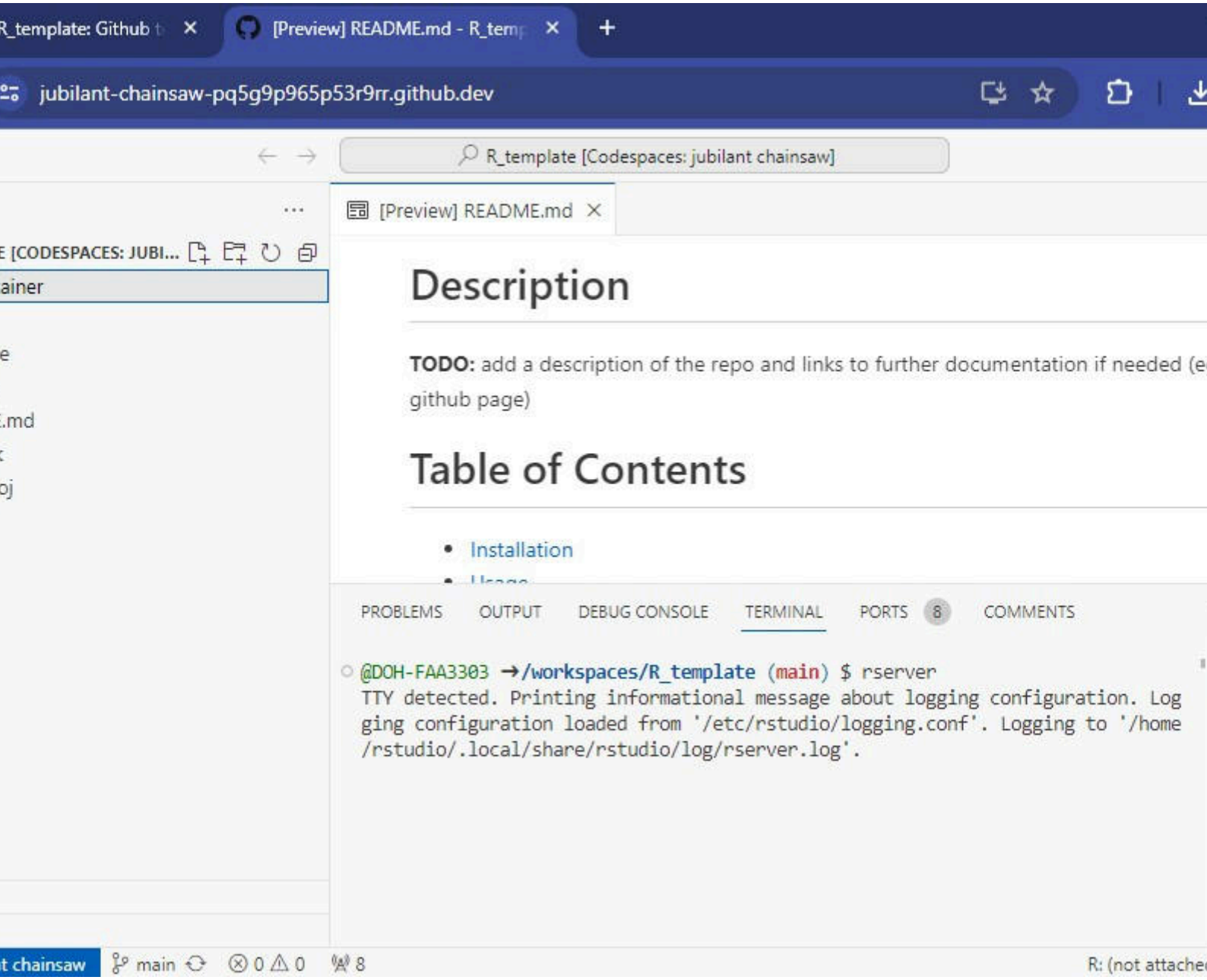



Figure 8 — VS Code IDE in Github Codespaces

Here you can install most software. You can also customize the Codespace so that whenever someone opens one in your repo it will come with software pre-installed. More on that in the devcontainers section

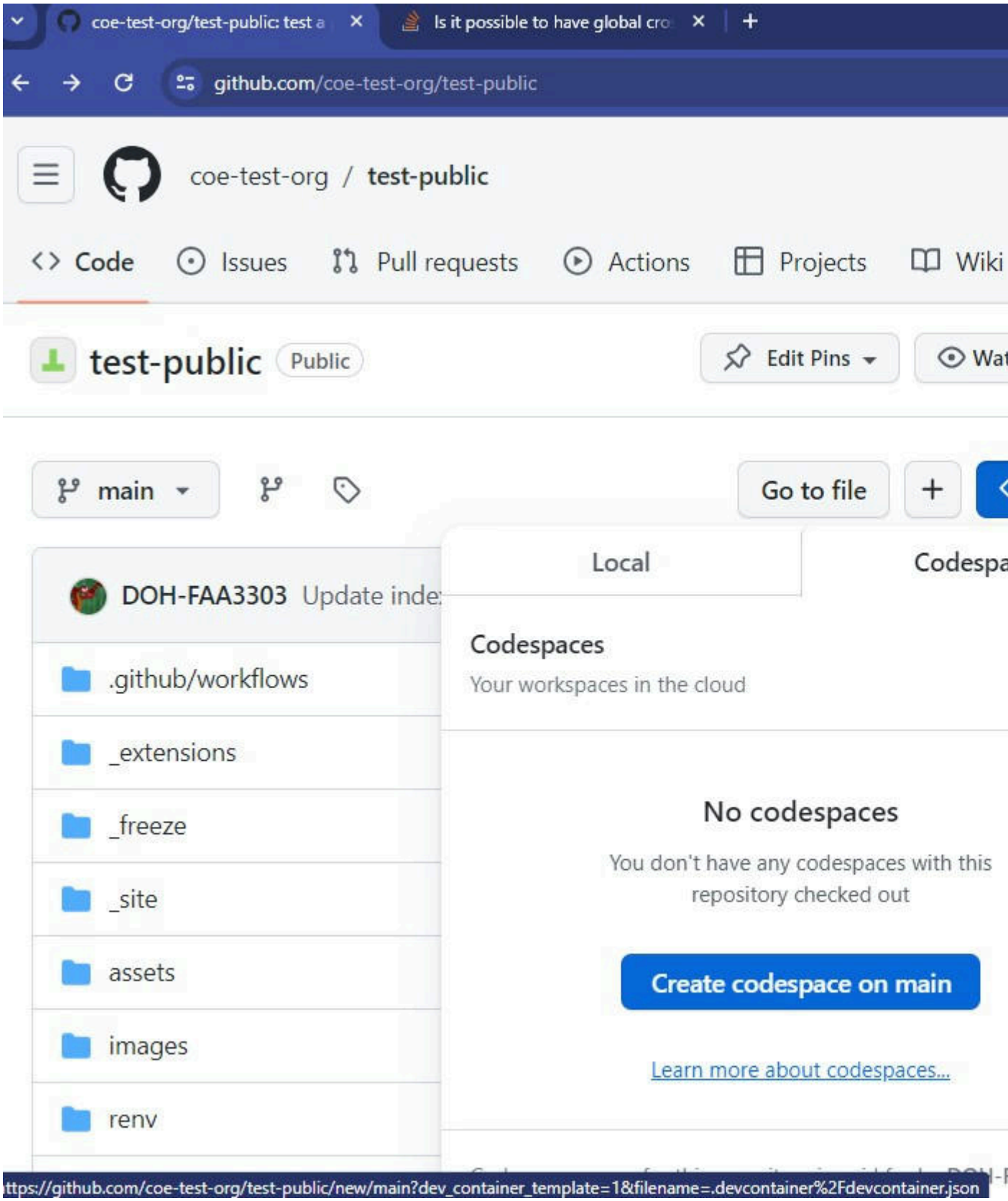
## 27.2 Devcontainers

Devcontainers are a way to install software into a Codespace so that whenever a user opens up the Codespace they won't need to install anything themselves. Making a container can be a little tricky, so we've made Github templates that

have devcontainers already made. See templates. There are R, Python, and general default templates. These containers will install R, Rstudio, Python, and all the packages in the repo's virtual environments (venv, conda, pip, renv, etc) so that the user can run all the code in your repo within a couple minutes.

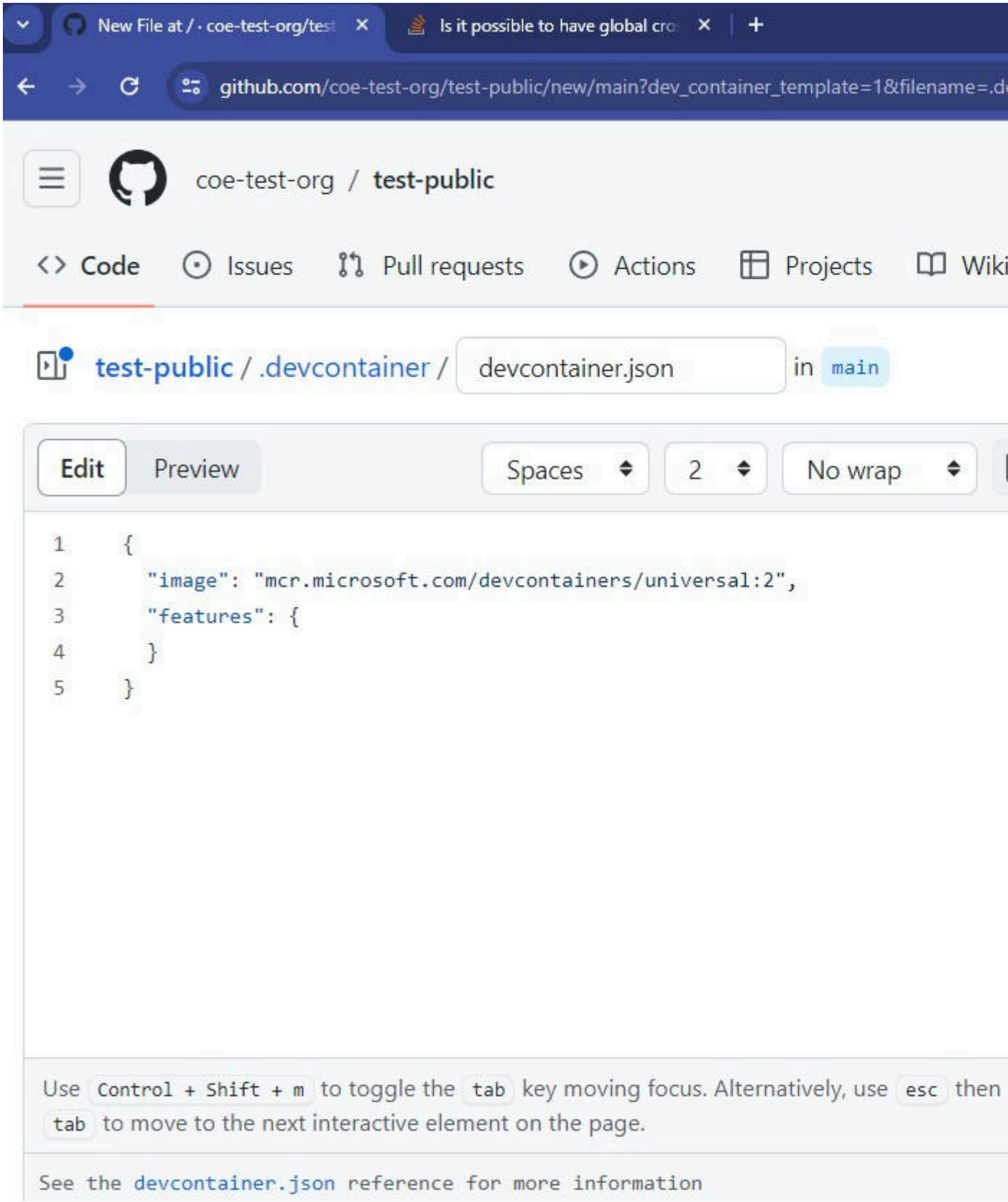
To set up a devcontainer for yourself;

1. Click on `Code > Codespaces > Configure dev container




2. This will make a folder named `.devcontainer` at the root of your repo
3. In that folder it will make a file named `devcontainer.json`

4. On the right there is a searchable marketplace for software to add to your container



5. Each one comes with instructions on how to add the software to the `.devcontainer.json`

For more information about Codespaces, see the guides here

# 28 Virtual Environments

Virtual Environments are another great way to make sure aspects of your repo are reproducible. They are commonly used to record package versions that the code/project uses. For more on virtual environments, please see the venv guide.

# 29 Github Releases

Github Releases save code snapshots, versions, and changelogs of your repo. They are a great way for end users and developers to use different versions of their code and visualize changes that happened with each version. Please see the Github Releases guide for more information.

# 30 Documentation

Your code should be well documented so that end users (and developers) can understand what code is doing, how to install the software, and the utility of the project.

In general, you should have a README.md file in your repo that explains at least a high level summary of the code in the repo and what it does, how to install the code, outputs, and how to contribute to the repo. In addition, it may be a good idea to make a Github Page (a static website hosted in your repo) that explains the code in more detail. See the documentation guides here.

Having a Github Page is necessary if you have a package. Consider using software like `pkgdown` for R or `quartodoc` for Python (or other related software that helps link code to your documentation automatically). See more about package documentation here.