NW-PaGe
Northwest Pathogen Genomics Center of Excellence

# Northwest Pathogen Genomics Center of Excellence

## Policies, standards, and guidelines

2025-02-25

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

# Table des matières

1 Introduction ..... 6
2 Get Started ..... 6
2.1 Checklist ..... 6
2.2 Access ..... 7
2.3 Github Security ..... 7
2.4 Building a Public Repo ..... 7
2.5 Licensing ..... 7
2.6 Repo tips/advice ..... 7
3 Security ..... 7
3.1 Objectives ..... 7
3.2 Protect Credentials with .gitignore ..... 8
3.2.a Environment Variables + .gitignore ..... 9
3.2.a.a .Renviron ..... 9
3.2.a.b yaml ..... 10
3.2.a.b.a Automating With Yaml Creds ..... 13
3.2.a.b.b yaml Template ..... 14
3.3 Security Guardrails ..... 15
3.4 Pre-commit Hooks ..... 15
3.4.a Windows ..... 16
3.4.b WSL/Linux ..... 18
3.5 Secret Scanning ..... 20
3.5.a git log for scanning ..... 21
3.5.b git secrets for scanning ..... 21
3.5.b.a Windows ..... 21
3.5.b.b WSL/Linux ..... 22
3.6 Pre-Recieve Hooks ..... 23
3.7 Pushing Private Code to Public Repos ..... 23
3.8 Code Reviewers/Github Operations Team ..... 23
4 GitHub Setup ..... 24
4.1 **Objectives** ..... 24
4.2 Git Basics ..... 24
4.3 Github Basics ..... 24
4.4 Contributing ..... 24
4.4.a Bug Report ..... 24
4.4.b Feature Request ..... 25
4.4.c Discussions ..... 25
4.4.d Contribute Code ..... 25
4.5 Create a Repo ..... 25
4.5.a.a Repository template ..... 25
4.5.a.b Owner and Repo name ..... 25
4.5.a.c Internal vs Private ..... 26
4.5.a.d README, .gitignore, license ..... 26
4.6 Cloning a Repo ..... 26

4.6.a HTTPS ... 26
4.6.a.a Make a PAT ... 28
4.6.b SSH ... 28
4.6.c GitHub CLI ... 30
4.6.c.a Windows ... 30
4.6.c.b WSL/Linux ... 31
4.6.d Open with GitHub Desktop ... 32
5 Collaboration Guide ... 33
5.1 Introduction ... 33
5.2 Workflow Overview ... 34
5.3 Steps for Collaborating on GitHub Using GitHub Desktop (GUI) ... 34
5.3.a 0. Install GitHub Desktop ... 34
5.3.b 1. Protecting the `main` Branch ... 34
5.3.b.a Step 1.1: Enable Branch Protection Rules ... 34
5.3.c 2. Initial SetUp: Cloning the Repository ... 35
5.3.d 3. Creating a Branch ... 37
5.3.e 5. Keeping Your Branch Up-to-Date ... 39
5.3.f 6. Pushing Changes to GitHub ... 40
5.3.g 7. Create a Pull Request ... 40
5.3.h 8. Reviewing and Addressing Feedback ... 41
5.3.i 9. Merging the Pull Request ... 42
5.3.j 10. Cleaning up Local Branches ... 42
5.3.k 11. Pulling Latest Changes to `main` ... 42
5.4 Steps for Collaborating on GitHub Using the Command Line ... 43
5.4.a 0. Sign-in ... 43
5.4.b 1. Protecting the `main` Branch ... 43
5.4.b.a Step 1.1: Enable Branch Protection Rules ... 43
5.4.c 2. Initial SetUp: Cloning the Repository ... 44
5.4.d 3. Branching Workflow ... 45
5.4.d.a Step 3.1: Create a branch ... 45
5.4.d.b Step 3.2: Develop on the branch ... 45
5.4.e 4. Making Pull Requests (PRs) ... 46
5.4.e.a Step 4.1 Update your Branch with `main` ... 47
5.4.e.b Step 4.2: Open a Pull Request ... 47
5.4.e.c Step 4.2: Resolve any Pull Request Feedback ... 48
5.4.f 5. Merging Pull Requests ... 48
5.4.f.a Step 5.1: Merge into `main` ... 48
5.4.g 6. Pruning Branches ... 48
5.4.g.a Step 6.1: Delete Local Branches ... 48
5.4.h 7. Update `main` with the merged PR ... 48
6 Licensing ... 49
6.1 **Summary** ... 49
6.2 General License Info ... 49
6.3 GNU GPL licenses ... 49
6.4 MIT license ... 49

7 Policies ..... 50
7.1 **Objectives** ..... 50
`README` ..... 50
`CODE_OF_CONDUCT` ..... 50
`CONTRIBUTING.md` ..... 50
`LICENSE` ..... 50
7.2 Set Policy Rules at Org Level ..... 51
7.2.a Document Requirements with `.github` Repos ..... 51
7.3 Set Templates at the Org Level ..... 53
7.3.a Commit Sign-Off Requirement - Github Apps ..... 56
7.4 IaC ..... 57
7.5 Branch Protections and GitHub Apps ..... 57
7.5.a 1. Protecting the `main` Branch ..... 57
7.5.a.a Step 1.1: Enable Branch Protection Rules ..... 57
7.5.b 2. GitHub App to allow bypass ..... 58
7.5.b.a Step 2.1 create an app ..... 58
7.5.b.b Step 2.2 generate secret ..... 58
7.5.b.c Step 2.3 install the app ..... 58
7.5.b.d Step 2.4 store secrets (for admins) ..... 58
7.5.b.e Step 2.5 (for all users) allow branch protection bypass ..... 59
7.5.b.f Step 2.6 set up a github action ..... 60
8 Reproducibility ..... 61
8.1 **Objectives** ..... 61
8.2 Data and Code Democratization ..... 61
8.3 Github Codespaces ..... 62
8.3.a Open a Codespace ..... 62
8.3.b Devcontainers ..... 63
8.4 Virtual Environments ..... 65
8.5 Github Releases ..... 65
8.6 Documentation ..... 65
9 Tutorials ..... 66
9.1 Release Cycles ..... 66
10 Github Releases ..... 66
11 Semantic Versioning ..... 69
12 Conventional Commits ..... 70
13 Automating The Release Cycle ..... 70
13.1 Github Action for auto releases ..... 71
..... 71
..... 72
14 Step by Step Instructions ..... 73
15 Resources ..... 76
15.1 Virtual Environments ..... 76
16 Introduction ..... 76
17 Python - Conda Environment Setup ..... 77
17.1 Step 1: Set Up Anaconda ..... 77

17.2 Step 2: Open Anaconda Prompt ... 77
17.3 Step 3: Change Directories ... 78
17.4 Step 4: Copy the repo env ... 79
17.5 Step 5: Activate the environment ... 80
18 Python - Programming Setup ... 80
18.1 IDE Setup - VS Code ... 81
18.1.a Step 1: Select a Python Interpreter ... 81
18.1.b Step 2: Write code ... 82
18.2 IDE Setup - PyCharm ... 84
18.2.a Step 1: Select a Python Interpreter ... 84
19 Python - Installing New Packages ... 84
19.1 Step 1: Install a new package ... 85
19.2 Step 2: Save the package to the repo ... 85
20 R Package Management - `renv` ... 86
20.1 Creating `renv` in a project ... 86
20.1.a Step 1: Open the `.Rproj` in your repo ... 86
20.1.b Step 2: Initialize `renv` for the repo ... 87
20.1.c Step 3: Push to Github ... 89
20.2 Using `renv` in a project ... 89
20.2.a Step 1: Open the `.Rproj` for your repo ... 89
20.2.b Step 2: Load `renv` packages ... 90
21 R Installing New Packages ... 90
22 Using Reticulate with Conda Env ... 91
22.1 Templates ... 92
22.2 R Github Template ... 92
22.3 Purpledoc Quarto Template ... 93
22.4 Documentation ... 94
23 Introduction ... 94
24 Create a Quarto Website ... 95
25 How to edit the website (add chapters and change the style) ... 96
25.1 Edit/Add Sections and Chapters ... 96
25.2 Website Style ... 97
26 How to edit a chapter ... 97
26.1 Open the R project ... 98
26.2 Open the files ... 98
26.3 Commit changes ... 99
27 How to link external code to the site ... 99
28 Publish the site/Github Actions ... 101
28.1 Example `YAML` Workflow ... 102
28.2 Using `renv` in the GH Action ... 104
28.3 Using a `_freeze` file ... 104
28.4 Troubleshooting ... 105
28.5 New Repo ... 105
28.6 Introduction ... 105
28.7 Steps ... 105

28.7.a Sign-in ..... 105
28.7.b Create a new repository ..... 105
28.7.c Fill out the new repository fields ..... 106
28.7.d Check out your new repo ..... 107
28.7.e Clone the new repo ..... 108
28.7.f Safeguard sensitive data ..... 109
28.7.g Populating the repo via commits ..... 109

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 Get Started

## 2.1 Checklist

If you need to make a public repo, this quick start guide is for you. **Please meet with Frank Aragona DOH.CDS.GenSeqSurvDQ@doh.wa.gov before creating a public repo.**

- ☐ Submit a request (service central ticket) to join a public GitHub organization (email[1] for help)
- ☐ Review items that are **prohibited** from being in a repo
- ☐ Add security guardrails
- ☐ Create global security hooks for all your repos
- ☐ Make a *private* staging repo to make sure everything is clean
- ☐ Perform secret scanning on your repo
- ☐ Add a license
- ☐ Make branch protections
- ☐ Copy over the *cleaned* main branch from private repo into new public repo
- ☐ Add a README and consider adding a Github page for documentation

[1] DOH.CDS.GenSeqSurvDQ@doh.wa.gov>

## 2.2 Access

Our organization has several public Github orgs. If you are internal to Washington State Department of Health, you will need to submit a Service Central Request for access. Please reach out to frank.aragona@doh.wa.gov for help.

## 2.3 Github Security

**Please read all of the security guidelines here**

The guide will go over

- what data/code is not allowed in a public repo
- security layers you need to prevent leaks
- converting private code to public repositories
- installation of pre-commit hooks to prevent leaks

## 2.4 Building a Public Repo

If you have an existing private repo that needs to be public, please follow these steps:

- repo cleaning guides
- code scanning

## 2.5 Licensing

- license rules

## 2.6 Repo tips/advice

- org policies
- make your repo reproducible
- repo documentation
- virtual environments
- git/github help, and how to collaborate on a repo

# 3 Security

## 3.1 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).

> **Caution**
>
> Err on the side of caution: If data isn't already public, it likely shouldn't be made public through a PR or a commit.

The following cannot be included in any repo **or any local commit!**:

Table 1 — Items prohibited from being in a Github repo

| Type | Examples |
|---|---|
| File Paths | • Network drives<br>• Shared internal drives |
| Server Names | • ODBC Connections |
| Credentials | • SSH Keys<br>• Tokens (REDCap, Azure, Github, etc)<br>• AWS paths<br>• Bucket connections<br>• Usernames<br>• Passwords<br>• Blob/bucket keys |
| Identifiable Information | • Case addresses<br>• Case names<br>• Any private health information (PHI)<br>• Geographic location (depending on circumstance)<br>• `WA` ids from PHL for genome sequencing (ids from public repos are okay) |
| Partners' Information | • Names of health facilities (i.e., facilities where healthcare-associated infection samples are collected)<br>• Geographic locations for mosquito traps (West Nile Virus surveillance) |

## 3.2 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.

### 3.2.a 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

#### 3.2.a.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

### 3.2.a.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:

```
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
)
```

```
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
```

#### 3.2.a.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
```

#### 3.2.a.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
```

## 3.3 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

## 3.4 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.

The instructions below work for a single user, but lacks automated security update capabilities. Please reach out to our email [DOH.CDS.GenSeqSurvDQ@doh.wa.gov] for access to our automated pre-commit hooks. We require this for all users in our org.

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

### 3.4.a 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

```
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 path/to/
secrets_key
```

10. Make sure the AWS providers are added:

```
git secrets --register-aws --global
```

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. Check your git history (Section 3.5)
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)

### 3.4.b 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. Add the secrets file to your provider list

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

11. Make sure the AWS providers are added:

```
git secrets --register-aws --global
```

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'
```

12. Check your git history (Section 3.5)
13. 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.
14. Test on one of my projects to see if rebasing is a sustainable option

15. Make repo public
16. Will automatically scan on every commit and won't let it commit unless it's clean - Create a few files to show it working

> **i** 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)

> **!** Important
>
> - 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

## 3.5 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/*`

I'll show 2 ways to scan for secrets, one with a native git command `git log` , and the other with `git secrets`

### 3.5.a git log for scanning

This is the easier way to scan *commit history*.

Navigate to your repo and execute this command:

```
git log -G"<regex here>" -p
```

Note that the regex pattern can be imported from the `secrets_key` file. You can also search for simple strings like this

```
git log -S "<string>" -p
```

There are a lot of options with the `git log` command and you can read more here and here

Basically, the `-G` option stands for `grep`, `-p` will show the diffs of the files, and `-S` is for string.

### 3.5.b git secrets for scanning

This isn't my favorite way to scan code, but it is an option if you want to double check things.

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.

#### 3.5.b.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
```

#### 3.5.b.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 {}
```

## 3.6 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

## 3.7 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)

## 3.8 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/1T4

HAPBFbb0?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

# 4 GitHub Setup

## 4.1 Objectives

- How to contribute to our GitHub repos
- Steps to create a GitHub repo
- Steps to clone a repo

**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

## 4.2 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

## 4.3 Github Basics

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

## 4.4 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.

### 4.4.a 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.

#### 4.4.b 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

#### 4.4.c Discussions

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

#### 4.4.d 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 and collaboration guide. There you will find our repo rules and instructions on how to set up pre-commit hooks and contributing how to contribute code.

Once granted access, follow the steps below to create a repo (Section 4.5) and/or collaborate on code (**?@sec-collab**).

### 4.5 Create a Repo

Please see our tutorial for more details.

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:

#### 4.5.a.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.

#### 4.5.a.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

#### 4.5.a.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.

#### 4.5.a.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

## 4.6 Cloning a Repo

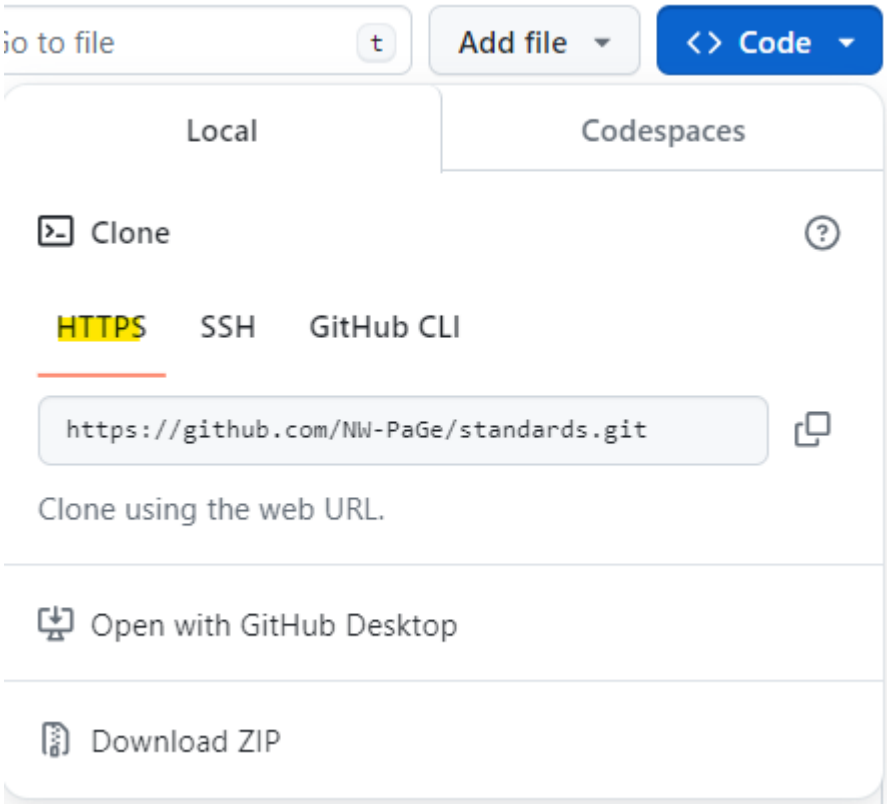
Whether you have created your own repo or want to contribute to someone else's repo, you will need to make a local clone of that repo on your personal machine.

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.

#### 4.6.a 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.

#### 4.6.a.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**

### 4.6.b 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
```

### 4.6.c GitHub CLI

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

#### 4.6.c.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
```

#### 4.6.c.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:

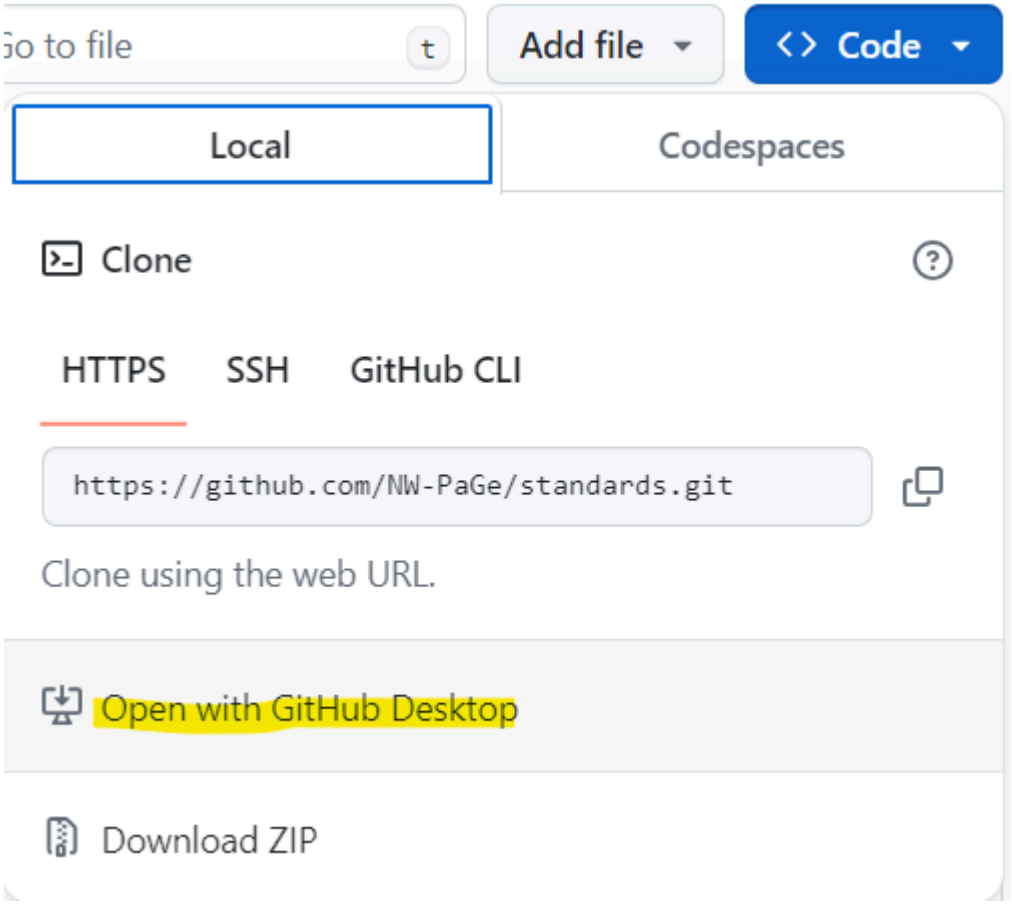
```
gh issue list
```

```
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
```

### 4.6.d 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>/`

# 5 Collaboration Guide

## 5.1 Introduction

We rely on GitHub for collaboration within and between teams. This tutorial goes over how to collaborate within the same GitHub repository if you have collaborator access to the repository. There are several ways in which you can interact with GitHub. We present two approaches using the command line and another using GitHub Desktop (a GUI).

## 5.2 Workflow Overview

The high-level workflow we recommend following for collaborators within a repository is:

1. Clone the repository.
2. Create a feature branch.
3. Make changes, commit, update branch with main, and push the branch.
4. Open a pull request from branch to main.
5. Conduct reviews and address comments.
6. Merge the pull request and delete the branch.
7. Update your local `main` repo with merged PR.
8. Repeat.

## 5.3 Steps for Collaborating on GitHub Using GitHub Desktop (GUI)

GitHub Desktop is a user-friendly application that simplifies version control and collaboration. This tutorial walks you through the process of collaborating on a GitHub repository using GitHub Desktop, including cloning a repository, creating branches, making changes, committing, pushing updates, and managing pull requests.

### 5.3.a 0. Install GitHub Desktop

GitHub Desktop is free and can be installed on your Windows or Mac machines. Follow the installation instructions here.

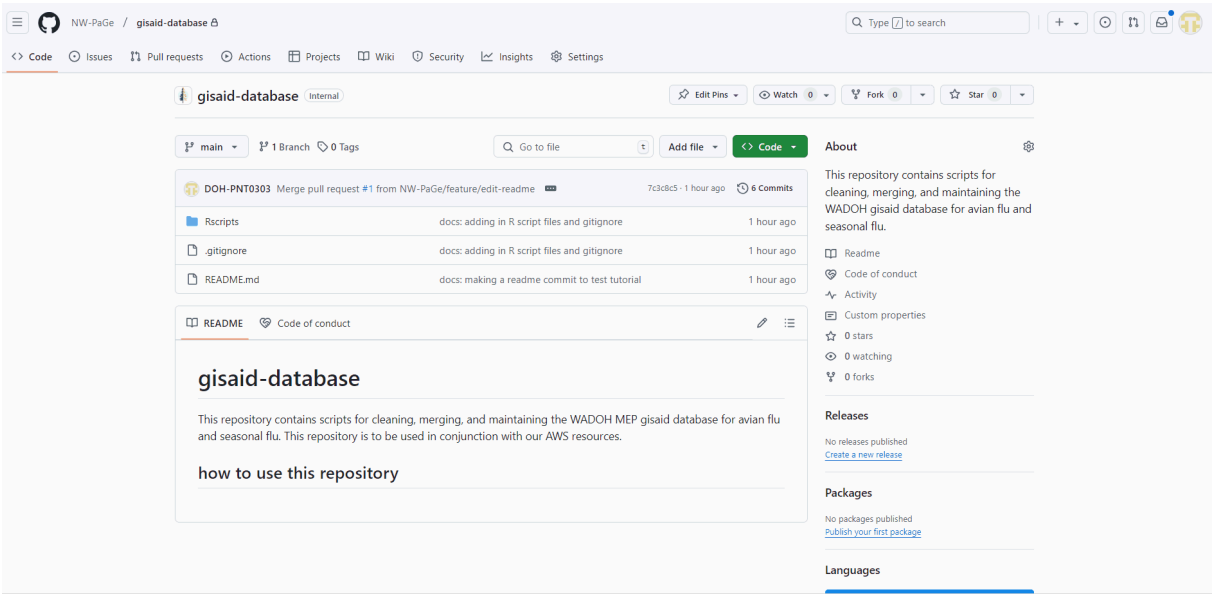
### 5.3.b 1. Protecting the `main` Branch

#### 5.3.b.a Step 1.1: Enable Branch Protection Rules

On your GitHub repository website page:

1. Navigate to **Settings** > **Branches** > **Branch Protection Rules** in the repository.
2. Click **Add branch ruleset**.
3. Enter a name for your ruleset.
4. Change **Enforcement status**: Active
5. Under **Targets**, click **Add target**:
    - Either click **Include by pattern** and type in `main` or if `main` is your default branch click **Include default branch**.

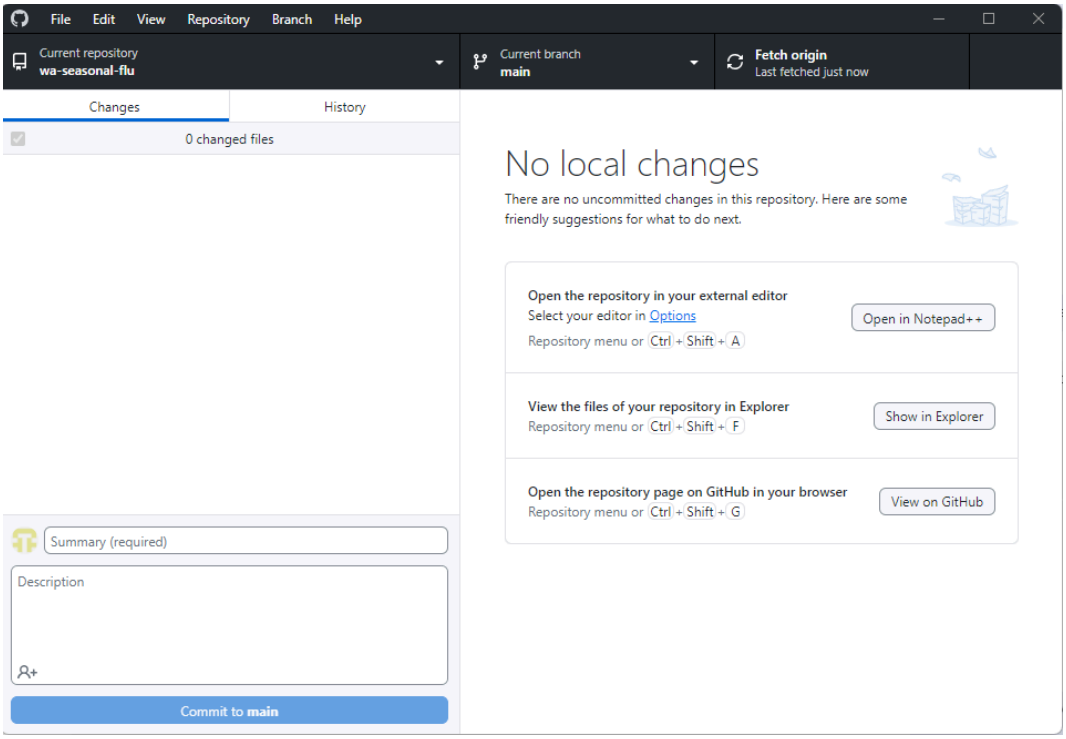
6. Under **Rules**:
    - Require a pull request before merging.
    - Restrict deletions.
    - Block force pushes.
    - Require status checks to pass before merging (optional but recommended for repos with CI/CD checks).
    - Under Require a pull request before merging: Enable Require approvals and specify the number of reviewers (optional)
7. Save changes.



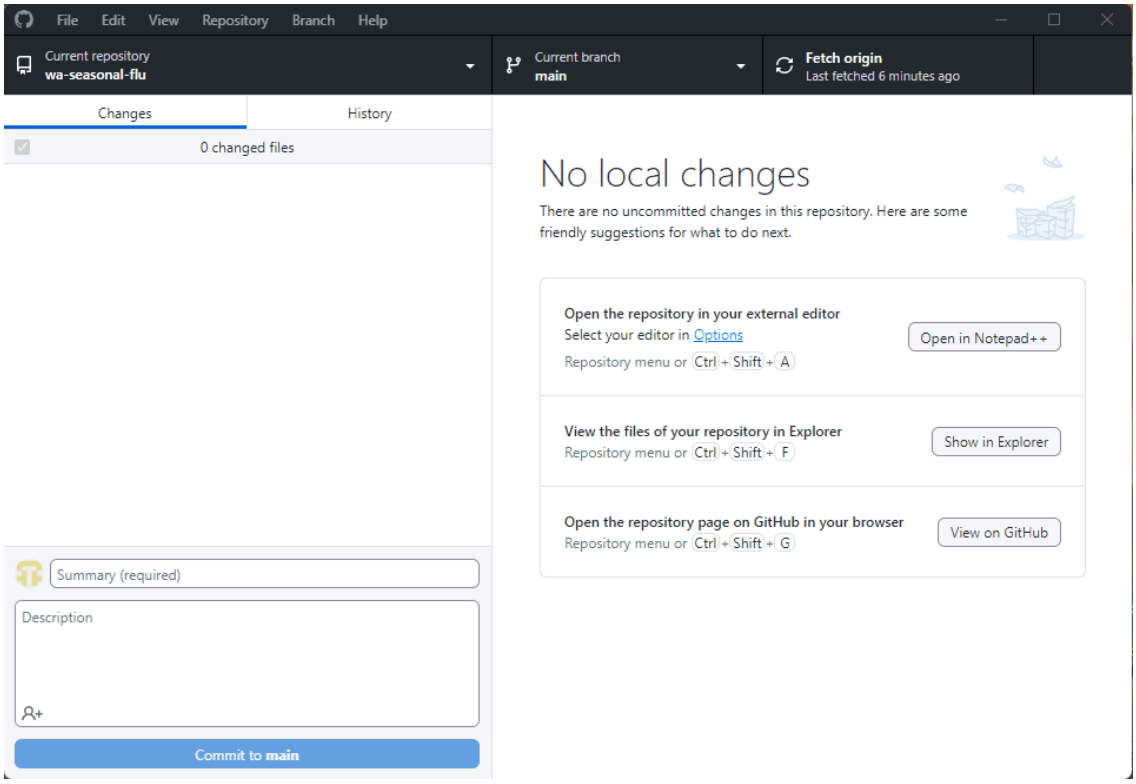

## 5.3.c 2. Initial SetUp: Cloning the Repository

1. Open GitHub Desktop

- Opening up GitHub Desktop should bring you to a UI similar to this where you see tabs for Current repository and Current branch.




2. Click the drop down arrow for **Current repository** > **Add** > **Clone repository** or you can click **File** > **Clone Repository**

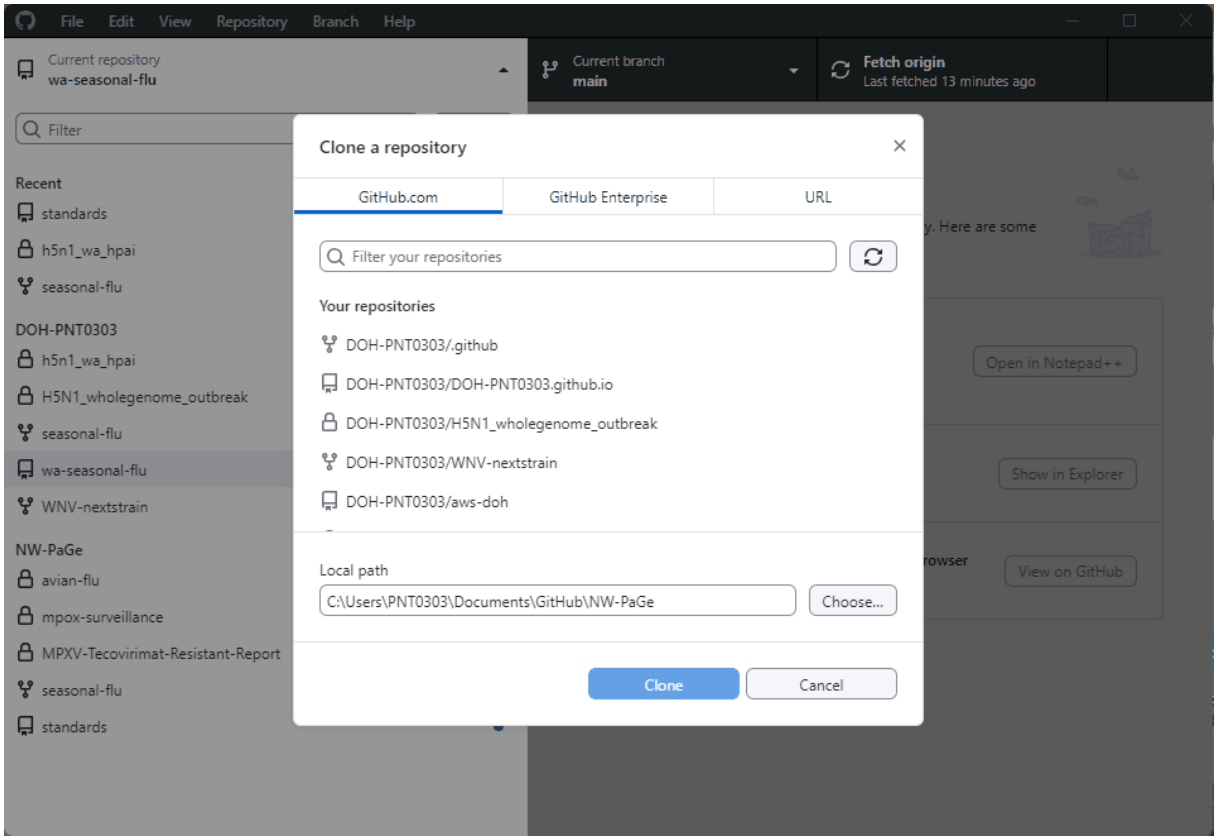



3. In the dialog box:

- Search for the repository that you want to clone
- Select the location you would like to clone the repository to on your local machine

4. Click **Clone**

Note: If you don't have collaborator access to a private repository you will not be able to clone the repository.

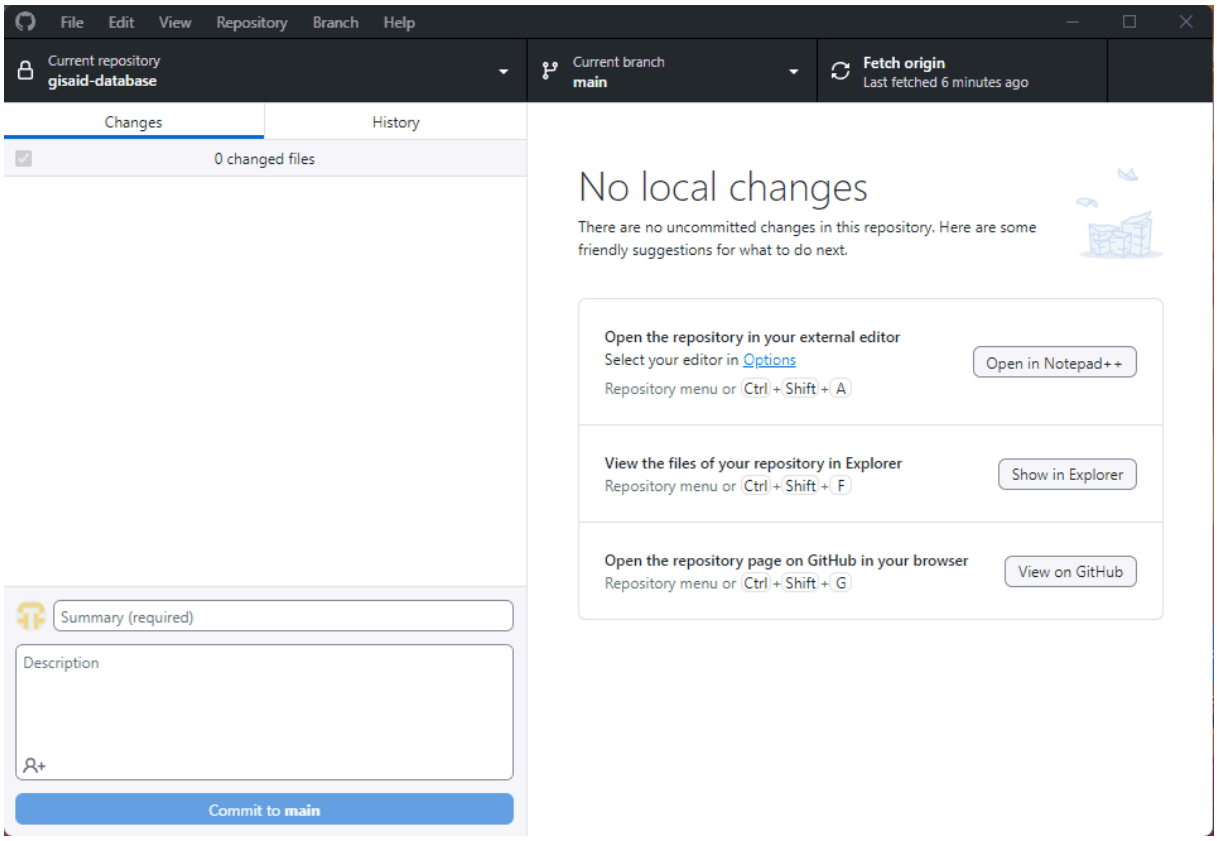



The repo and its contents will be located at the local path you've selected.

### 5.3.d 3. Creating a Branch

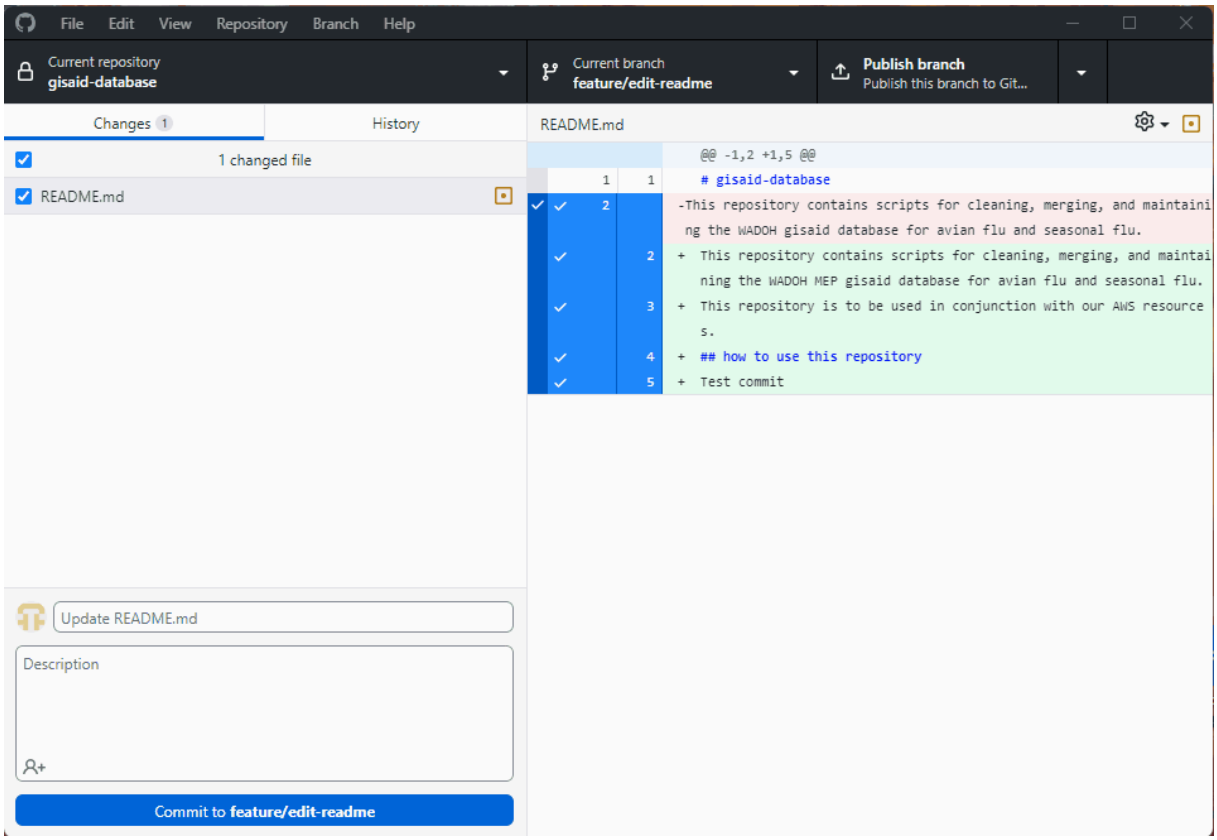
1. Ensure the `main` branch is selected in the current repository
   - if not, click the branch dropdown in the top bar and select `main`
2. Make sure that `main` is up to date by clicking `Fetch origin`
3. Click Branch > New Branch
4. Enter a descriptive name for the branch, such as `feature/add-readme` or `bugfix/fix-typo` .
5. Ensure that it says "Your new branch will be based on your currently checked out branch ( `main` ). `main` is the default branch for your repository", then click Create Branch.

6. The new branch will now be checked out automatically.




###

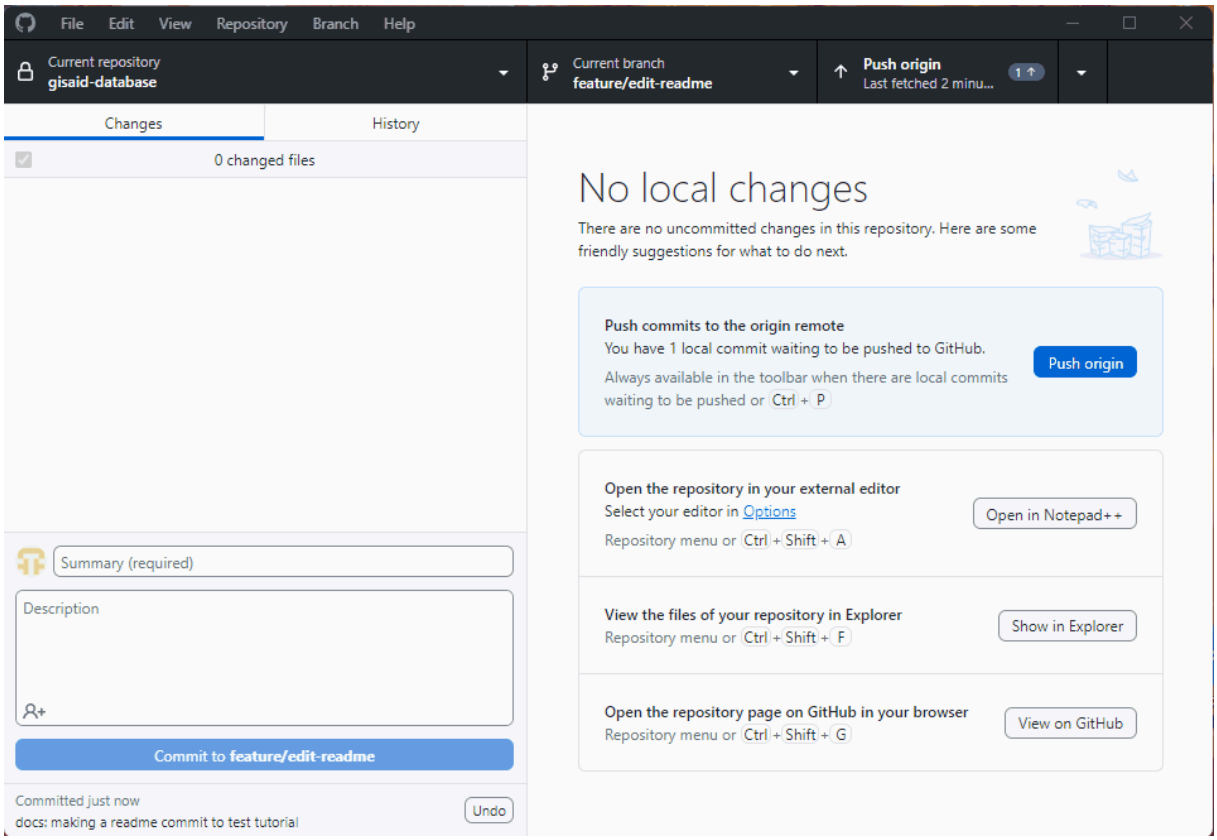
4. Make changes and commit those changes to the branch
5. Make changes to the files and code in your repository folder
6. After making changes:
    - Go back to GitHub Desktop
    - You'll see the list of changed files under the Changes tab
7. Stage and commit your changes:
    - Write a short, descriptive commit message in the Summary field (e.g. docs: added details to README)
    - Optionally, add a description for more details
    - Click **Commit to** `<branch_name>` to save your changes locally




### 5.3.e 5. Keeping Your Branch Up-to-Date

To avoid conflicts, ensure your branch is up-to-date with the latest changes from `main`

1. Switch to the `main` branch
    - Click the branch drop down and select `main`
    - Click **Fetch origin** to pull the latest changes
2. Switch back to your branch and merge `main`
    - Click **Current branch > Choose a branch to merge into** or you can click at the top **Branch > Merge into Current Branch**
    - Select `main` as the branch to merge.
    - If the branch is up to date with `main` then the "Create a merge commit" button will not be clickable.
    - Create a merge commit if there are differences between `main` and your branch.
3. Resolve any conflicts (if prompted).
4. Make any necessary commits after resolving conflicts

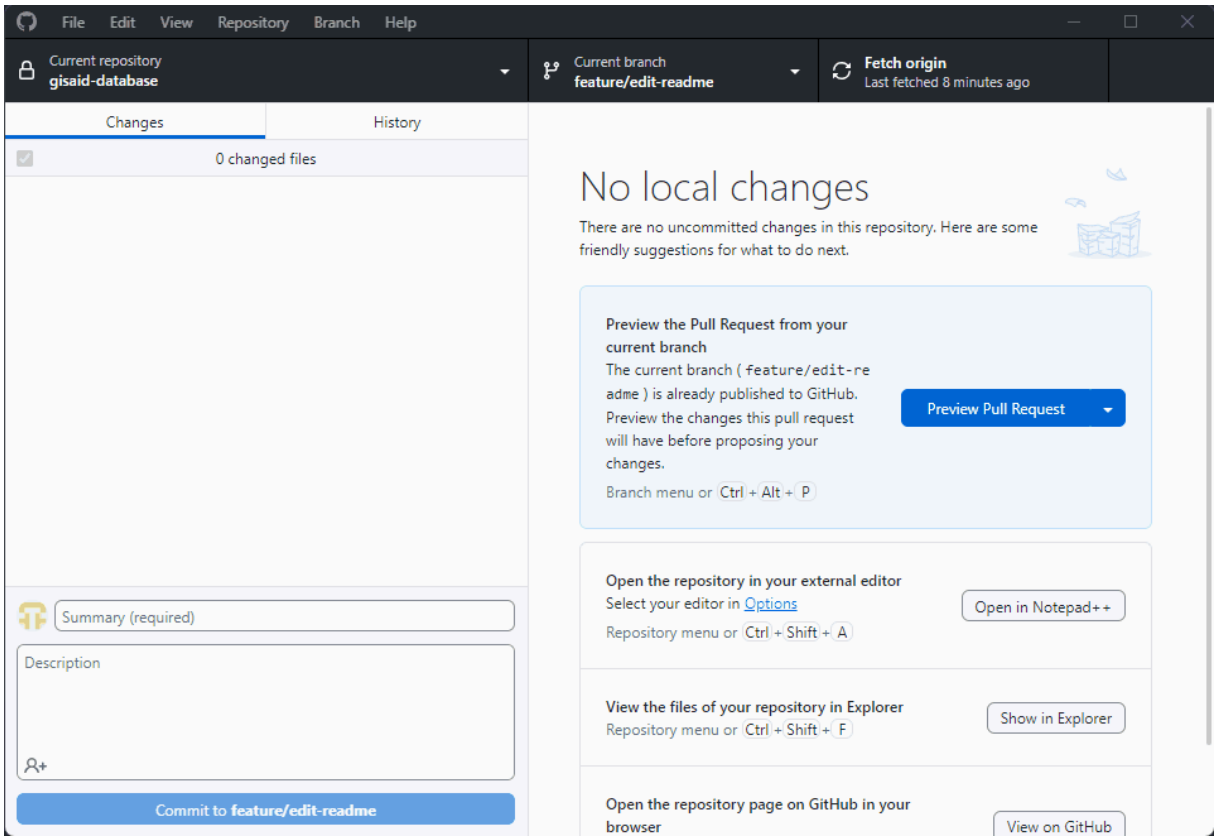


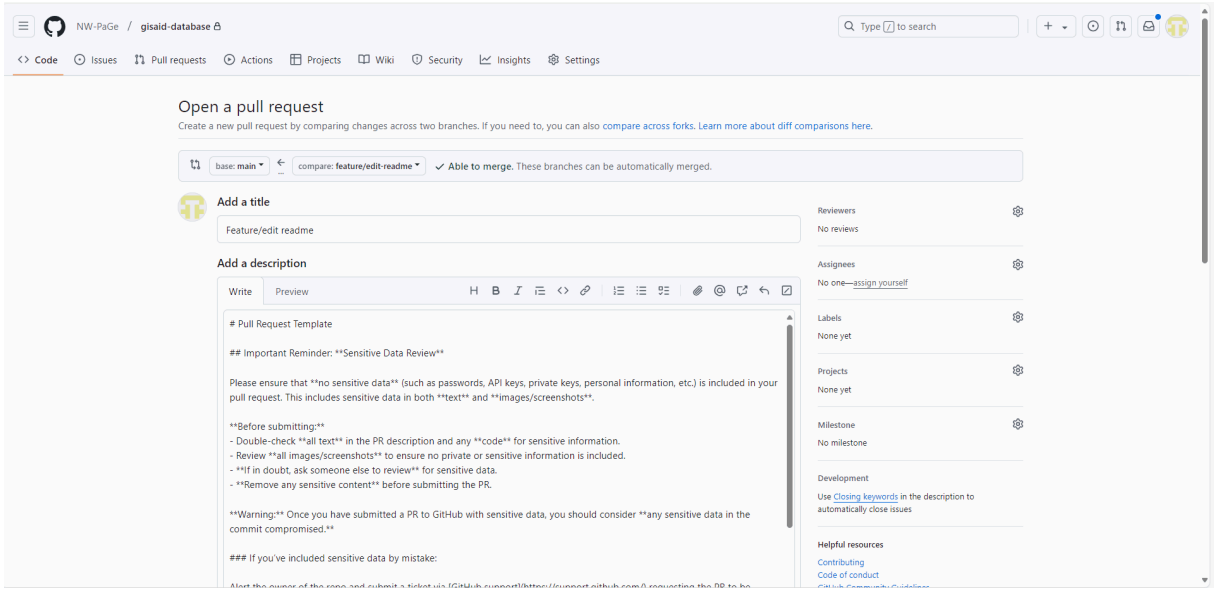

### 5.3.f 6. Pushing Changes to GitHub

1. After committing your changes, click **Publish branch** in GitHub Desktop to push the new branch to GitHub
2. If the branch is already published, click **Push origin** to sync your changes to the remote repository on GitHub.

### 5.3.g 7. Create a Pull Request

1. Open GitHub Desktop and click **Branch > Create Pull Request**
    - This opens the pull request (PR) page on GitHub in your web browser
2. Fill out the PR form:
    - Ensure the source branch is your feature branch and the target branch is `main` at the top of the PR where it should say `base:main <- compare:<branch_name>`
    - If your repository is in the NW-PaGe organization we have auto-populated Pull Request Templates to remind you to look for sensitive data that may be accidentally included in your commits or the pull request itself.
3. Submit the pull request



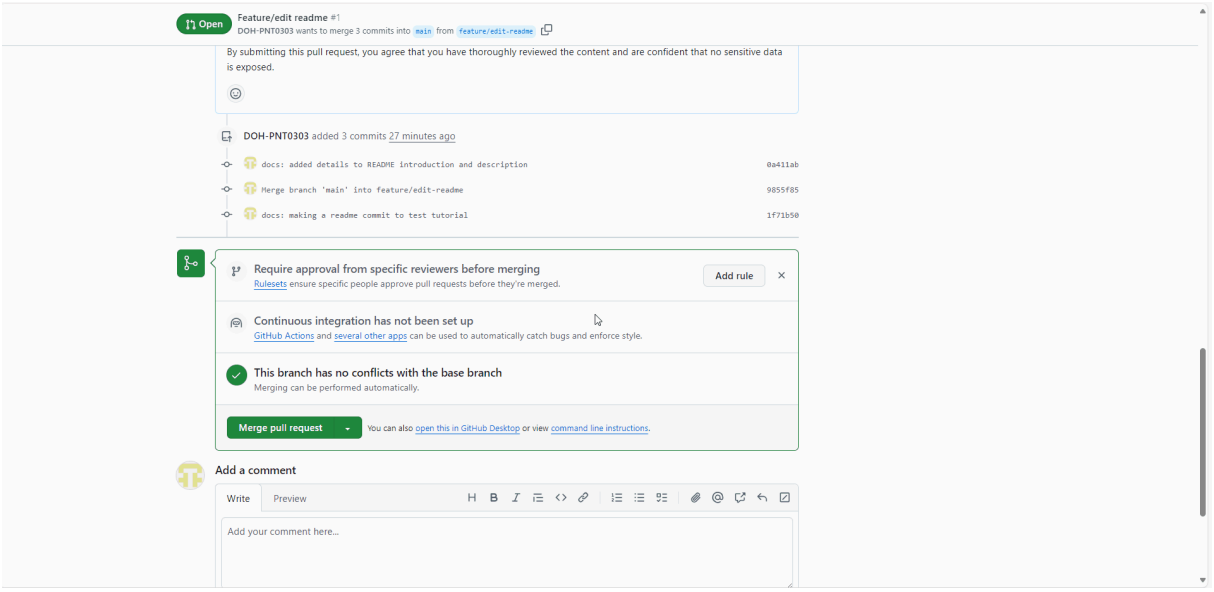




## 5.3.h 8. Reviewing and Addressing Feedback

1. Collaborators may review your pull request and suggest changes.
2. If changes are requested:
    - Make updates locally in your branch
    - Commit the changes in GitHub Desktop
    - Push the branch to update the PR automatically

### 5.3.i 9. Merging the Pull Request

1. Once the pull request is approved and all checks pass, click Merge PUll Request on GitHub
2. After merging, delete the branch on GitHub by clicking Delete Branch

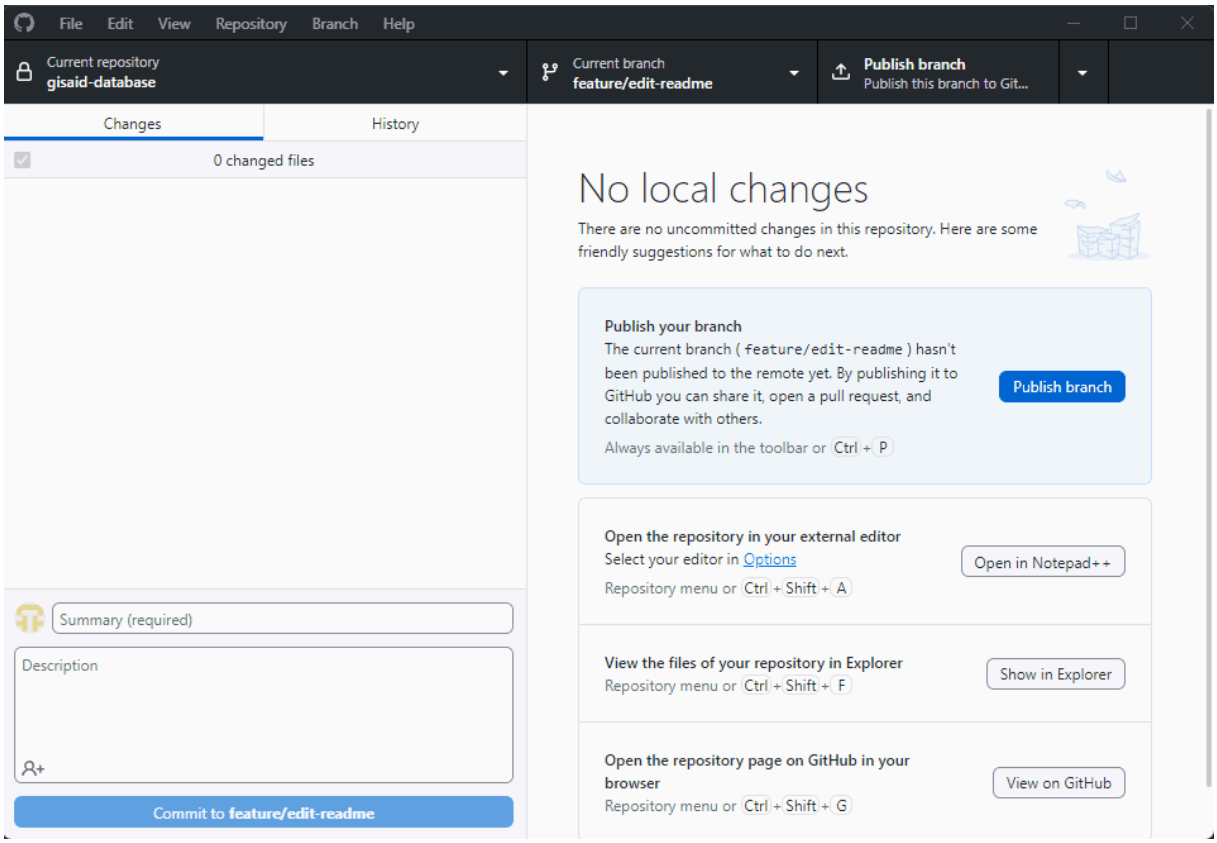



### 5.3.j 10. Cleaning up Local Branches

1. After merging the pull request, delete the local branch to keep your work space clean.
2. In GitHub Desktop:
    - go to the branch drop down
    - Select the branch you want to delete
    - Right-click and choose **Delete**

### 5.3.k 11. Pulling Latest Changes to `main`

1. Switch back to the `main` branch in GitHub Desktop
2. Click **Fetch origin** to pull the latest changes




## 5.4 Steps for Collaborating on GitHub Using the Command Line

### 5.4.a 0. Sign-in

Sign-in to GitHub using your GitHub credentials. If you are part of WA DOH make sure to use your WA DOH is compliant Git Hub account. Every WA DOH GitHub user should have 2-factor authentication enabled.

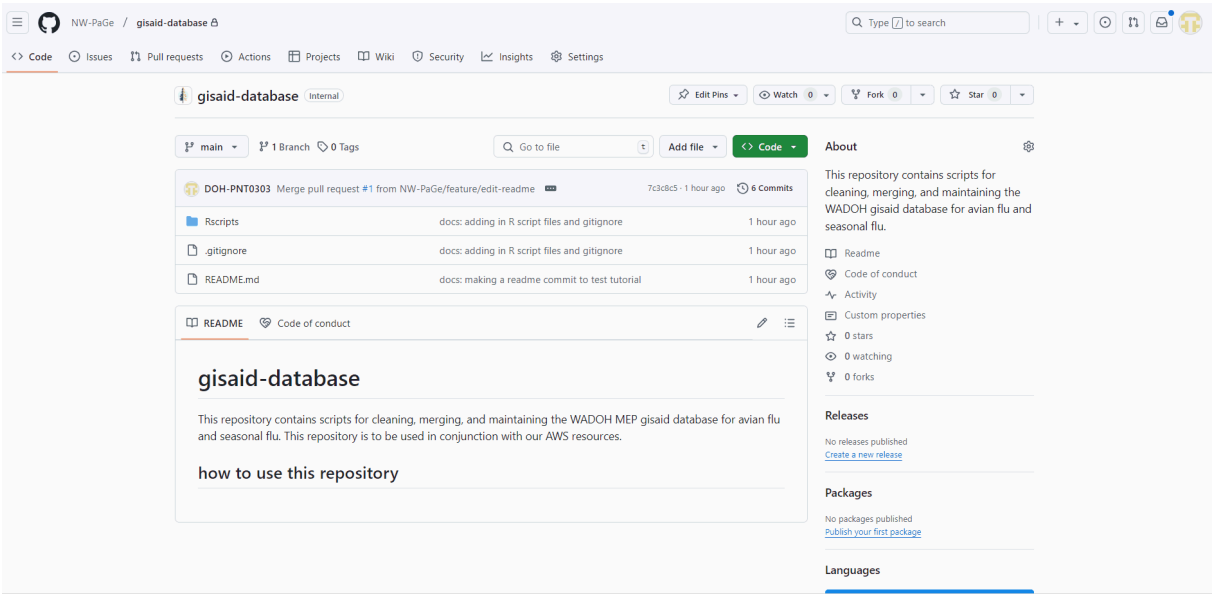
### 5.4.b 1. Protecting the `main` Branch

#### 5.4.b.a Step 1.1: Enable Branch Protection Rules

On your GitHub repository website page:

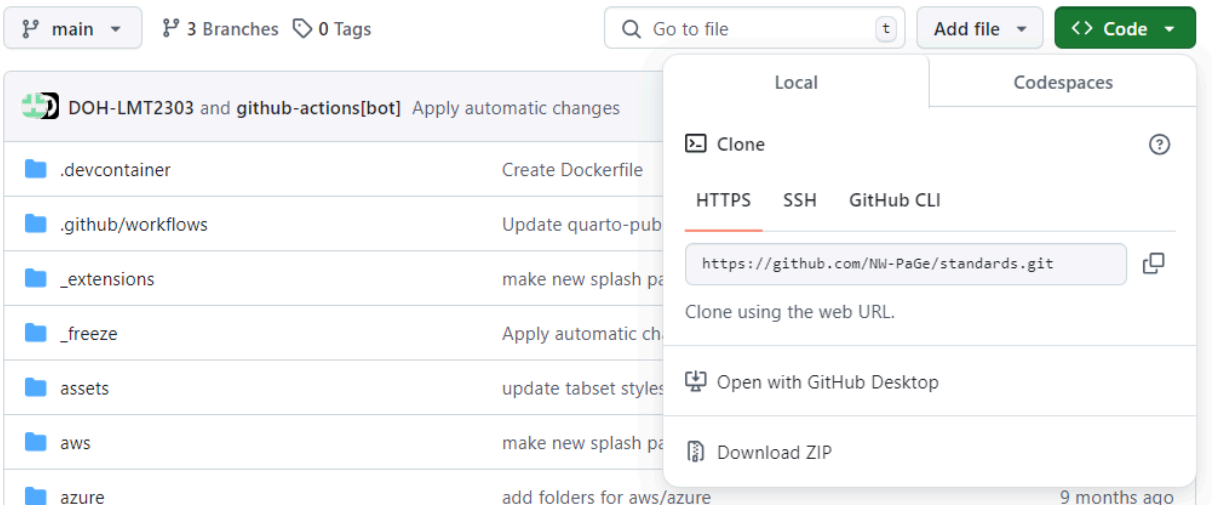
1. Navigate to **Settings** > **Branches** > **Branch Protection Rules** in the repository.
2. Click **Add branch ruleset**.
3. Enter a name for your ruleset.
4. Change **Enforcement status**: Active
5. Under **Targets**, click **Add target**:
    - Either click **Include by pattern** and type in `main` or if `main` is your default branch click **Include default branch**.

6. Under **Rules**:
    - Require a pull request before merging.
    - Restrict deletions.
    - Block force pushes.
    - Require status checks to pass before merging (optional but recommended for repos with CI/CD checks).
    - Under Require a pull request before merging: Enable Require approvals and specify the number of reviewers (optional).
7. Save changes.




## 5.4.c 2. Initial SetUp: Cloning the Repository

- Locate the repository on GitHub
- Copy the repository URL from the green Code button.




- Run the following command in your terminal to clone the repo:

```
git clone <repository_url>
```

- Navigate to the repository folder:

```
cd <repository_name>
```

### 5.4.d 3. Branching Workflow

#### 5.4.d.a Step 3.1: Create a branch

Branches are created to isolate development tasks. Always branch off the `main` branch.

1. Pull the latest change from the `main` branch:

```
git checkout main
git pull origin main
```

2. Create and switch to a new branch

```
git checkout -b <branch_name>
```

**Note: Branches can be called whatever you'd like. If you'd like to organize your name conventions you could consider using prefixes like `feature/` , `bugfix/` . This would look like: `git checkout -b feature/add-flu-lbis`**

3. Push the new branch to publish on GitHub:

```
git push -u origin <branch_name>
```

#### 5.4.d.b Step 3.2: Develop on the branch

1. Make changes to the code/repository.
2. Stage the changes:
    - First check that you have the changes you want to make by running `git status` . This will show all the files you've changed.

```
git status
```

- Then when you're sure you have the changes you want, you can stage the changes individually or all at once:

```
git add <path/to/changed/file> # stage individual file (conservative and secure approach)
```

```
git add . # stage all files at once (risky approach)
```

> ⚠ Warning
>
> `git add .` will stage *all* files with changes or deletions. This could be a security risk if you're unaware of all the changes you've made on a branch.

3. Commit the changes with a descriptive message:

```
git commit -m "docs: make changes to readme documentation to include instructions on logging in"
```

**Note: Use clear, description messages. You can follow the format of conventional commits such as `<type>:<subject>` for a commit message. Example: `fix: fix bug in merge.py script` .**

Please see the Release Cycle page for more info. In summary, conventional commits can trigger an action in GitHub. For example, whenever a commit title contains the word `fix:` , a GitHub Action will bump up the codebase's version number from something like 1.0.0 to 1.0.1 - We use the following key words in our commit messages:

| key word | when to use it |
|---|---|
| `fix:` | a commit of the *type* `fix` patches a bug in your codebase (this correlates with `PATCH` in Semantic Versioning). |
| `feat:` | a commit of the *type* `feat` introduces a new feature to the codebase (this correlates with `MINOR` in Semantic Versioning). |
| `docs:` | your commit is related to updating the documentation and *not* the codebase itself |
| `chore:` | your commit doesn't change what the code or documentation does, it just updates something like formatting, file structure, naming conventions, etc. |
| `test:` | your commit is just a test commit |
| | |

### 5.4.e 4. Making Pull Requests (PRs)

#### 5.4.e.a Step 4.1 Update your Branch with `main`

Before opening a PR, ensure your branch is up-to-date with the latest changes in `main` to ensure compatability. 1. Swith to the `main` branch and pull the latest changes:

```
git checkout main
git pull origin main
```

2. Switch back to your branch:

```
git checkout <branch_name>
```

3. Merge `main` into your branch

```
-   Merge (safe and retains all commit history):
```

```
git merge main
```

**Note: if you're comfortable with git and you need to keep a clean git history, consider using `git rebase main` . Here's an excellent article explaining the pros and cons of merge vs rebase**

4. Resolve conflicts, if any:

```
-   Edit conflicting files, then stage the changes:
```

```
git add <file_name>
```

```
    -   Commit the resolved conflicts:
```

```
git commit -m "chore: Resolve merge conflicts with main"
```

5. Push the updated branch:

```
git push
```

#### 5.4.e.b Step 4.2: Open a Pull Request

1. Push changes to the feature branch:

```
git push origin <branch_name>
```

2. On GitHub, click **Pull Requests** > **New Pull Request.**
3. Select your branch as the source and `main` as the target.
4. Add a title and descripton, request reviews, and submit
5. Submit the pull request.

#### 5.4.e.c Step 4.2: Resolve any Pull Request Feedback

1. Address feedback in your branch then commit and push back to the branch

```
git add .
git commit -m "Address PR feedback"
git push
```

### 5.4.f 5. Merging Pull Requests

#### 5.4.f.a Step 5.1: Merge into `main`

1. Ensure the PR passes all checks and is approved.
2. Click **Merge Pull Request**
3. Delete the branch on GitHub by clicking **Delete Branch**

### 5.4.g 6. Pruning Branches

#### 5.4.g.a Step 6.1: Delete Local Branches

1. List all local branches

```
git branch
```

2. Delete a branch:

```
git branch -d <branch_name>
```

**Note: Use `-D` to force delete if the branch isn't merged**

### 5.4.h 7. Update `main` with the merged PR

After merging your pull request, it's important to update your local `main` branch to reflect the latest changes from the remote repository. 1. Switch to the `main` branch:

```
git checkout main
```

2. Pull the latest changes from the remote:

```
git pull origin main
```

# 6 Licensing

## 6.1 Summary

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

## 6.2 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:

## 6.3 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."

## 6.4 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

# 7 Policies

## 7.1 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
- Set up bots and branch protections

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.

Below is 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`

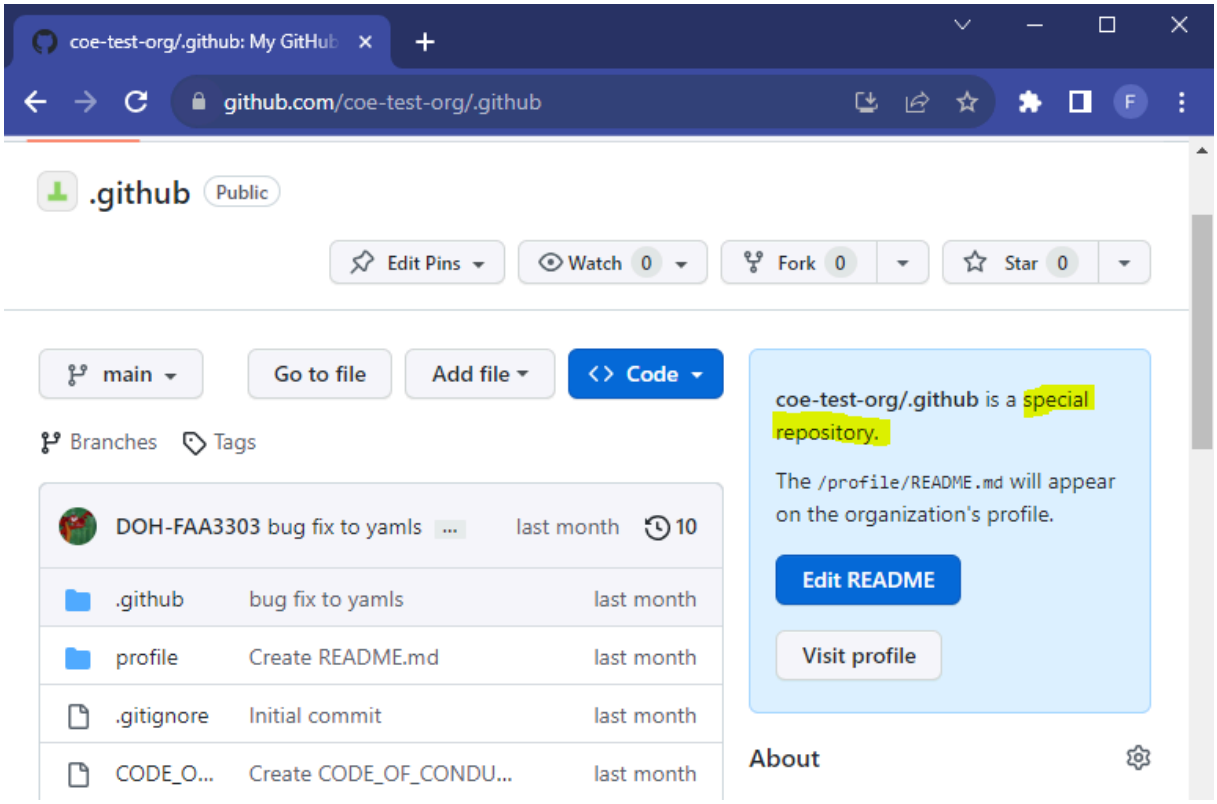
These should be and are set at the repo level. There will be many different licenses to choose from that will depend on the specific repo. More on that here.

## 7.2 Set Policy Rules at Org Level

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

### 7.2.a 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.

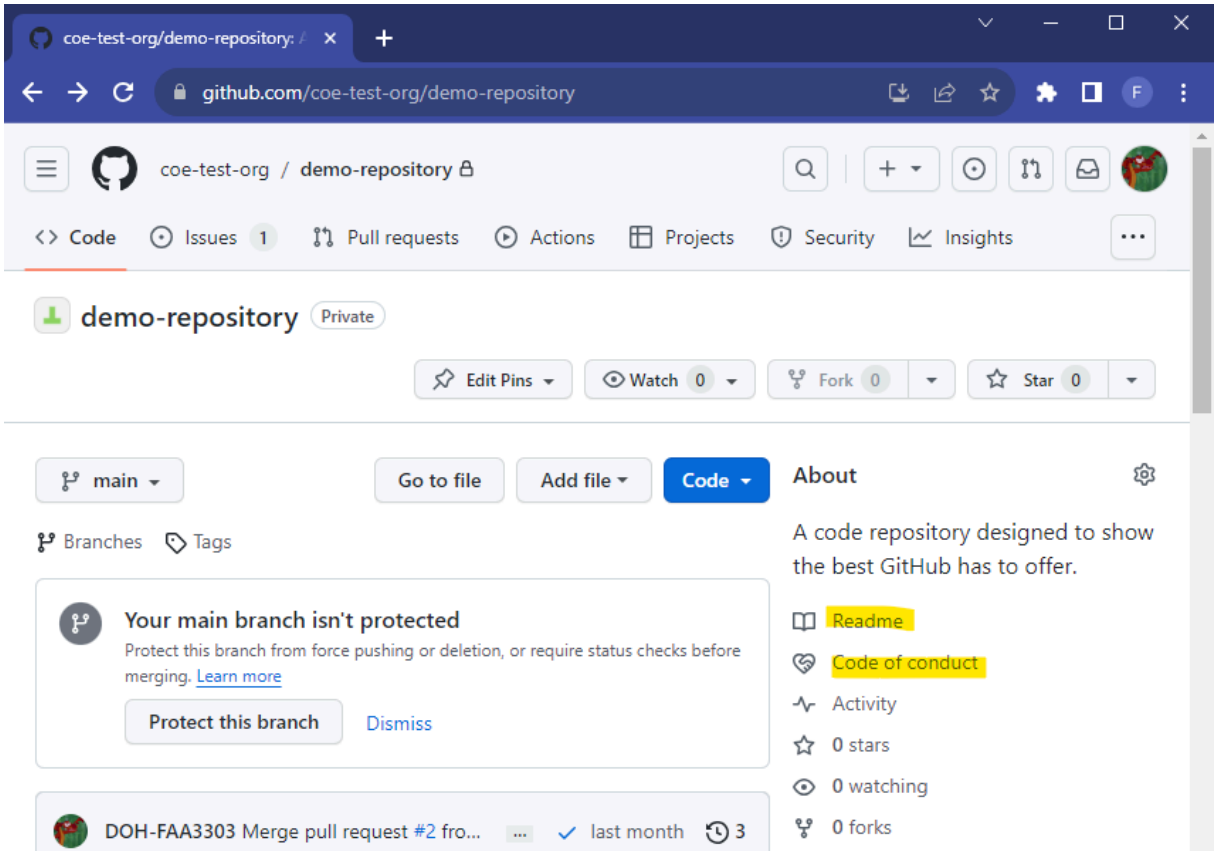



In order to write and set these policies at the organization level we can put them at the root of the `.github` repository and edit them there.

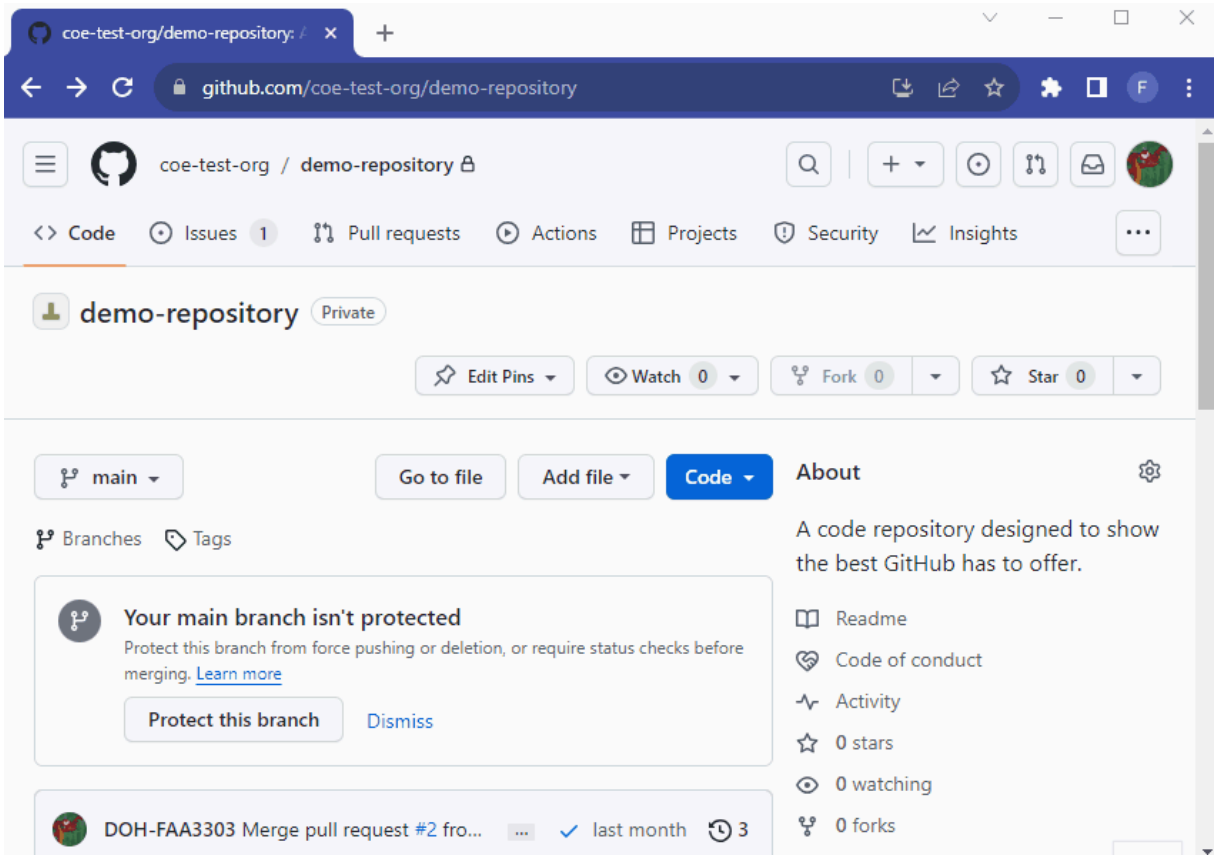
```
$ tree /f
C:.
|   .gitignore
|   CODE_OF_CONDUCT.md
|   CONTRIBUTING.md
|   LICENSE
```

```
│   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.

## 7.3 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
```

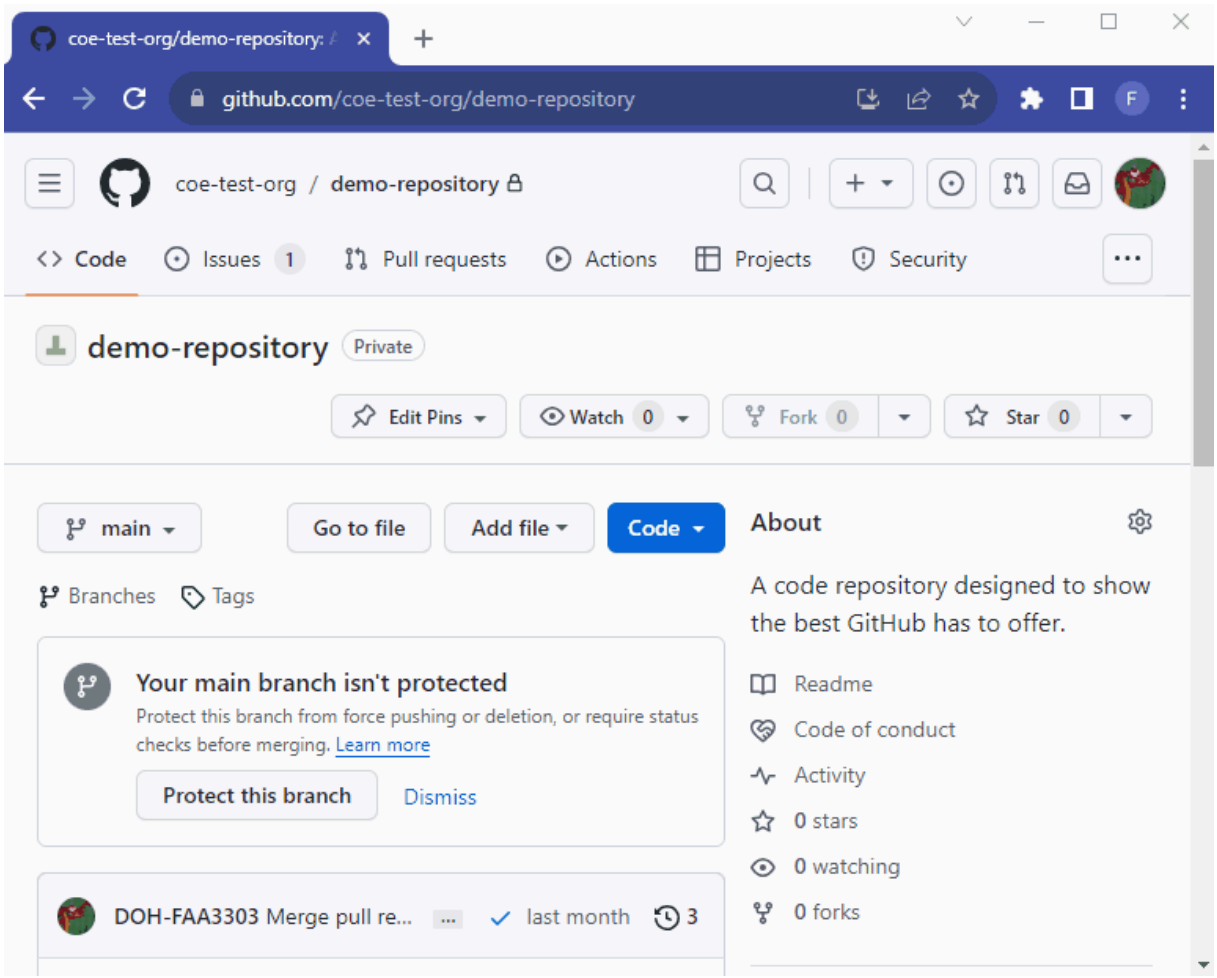
Each Folder has `.yml` files in it that are basically Github instructions on how to format issues and discussions.

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
```

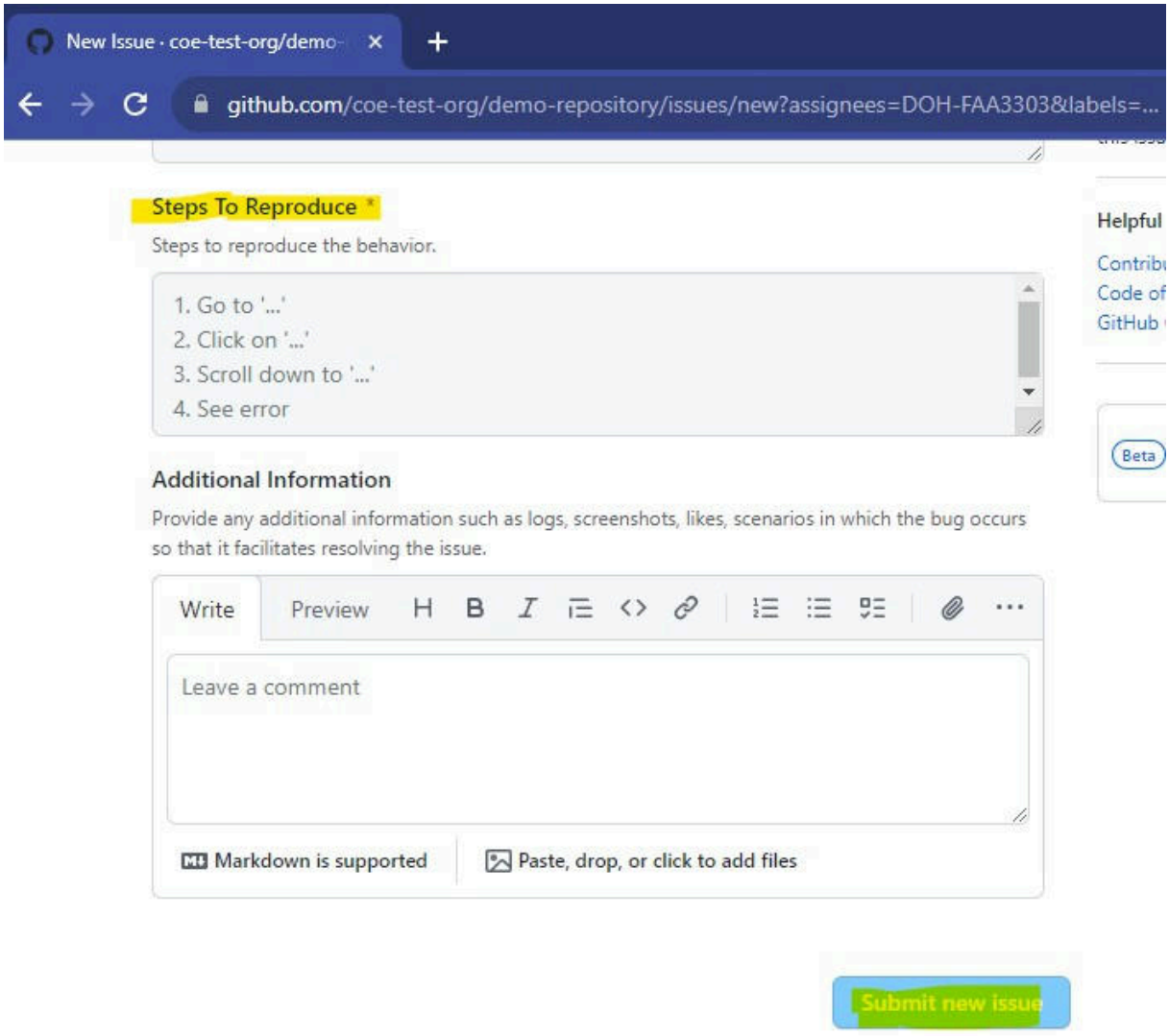
Now, when someone clicks on the `Issues` tab in a repo in this organization they will be met with the `Bug Report` template:




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
```




### 7.3.a 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.

## 7.4 IaC

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

## 7.5 Branch Protections and GitHub Apps

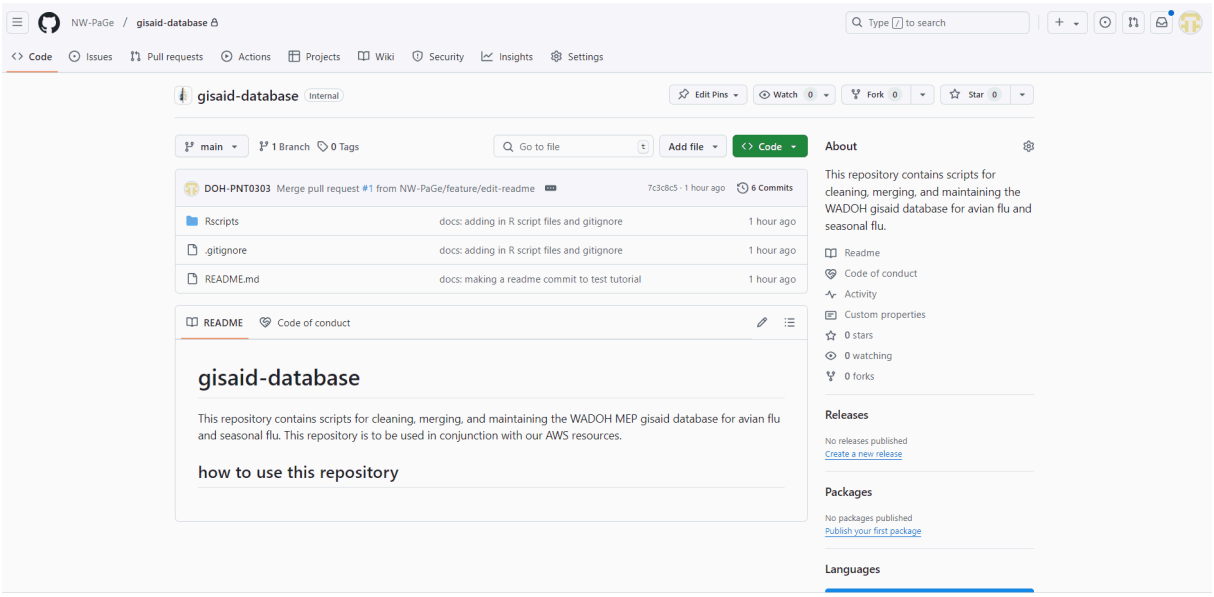
We want the main branch in our repos to be protected. We can require a pull request to the branch before merging to ensure that all commits are reviewed before being added to the main code base. Follow the steps below to protect your branch.

### 7.5.a 1. Protecting the `main` Branch

#### 7.5.a.a Step 1.1: Enable Branch Protection Rules

On your GitHub repository website page:

1. Navigate to **Settings** > **Branches** > **Branch Protection Rules** in the repository.
2. Click **Add branch ruleset**.
3. Enter a name for your ruleset.
4. Change **Enforcement status**: Active
5. Under **Targets**, click **Add target**:
    - Either click **Include by pattern** and type in `main` or if `main` is your default branch click **Include default branch**.
6. Under **Rules**:
    - Require a pull request before merging.
    - Restrict deletions.
    - Block force pushes.
    - Require status checks to pass before merging (optional but recommended for repos with CI/CD checks).
    - Under Require a pull request before merging: Enable Require approvals and specify the number of reviewers (optional)
7. Save changes.




### 7.5.b 2. GitHub App to allow bypass

Now that the main branch is protected and requires a pull request, our GitHub Actions will not be able to push code automatically because they cannot push to the main branch. You can have the Action make a pull request, but often we want our GitHub Actions to just automatically push code on a cadence or trigger. We need a bot to be able to bypass the branch protections for this because GitHub refuses to make this an easy option.

#### 7.5.b.a Step 2.1 create an app

Go to your Org Settings > scroll all the way down to Developer Settings > Click GitHub Apps > New GitHub App

#### 7.5.b.b Step 2.2 generate secret

Generate a client secret for the app. This will save a `.pem` file for the app to your local machine. SAVE THIS in a SECURE PLACE!!

#### 7.5.b.c Step 2.3 install the app

Go to Install App on the left toolbar > under Repository Access click All repositories > then save and Install

#### 7.5.b.d Step 2.4 store secrets (for admins)

Go to your Org main page > settings > Secrets and variables > New organization secret

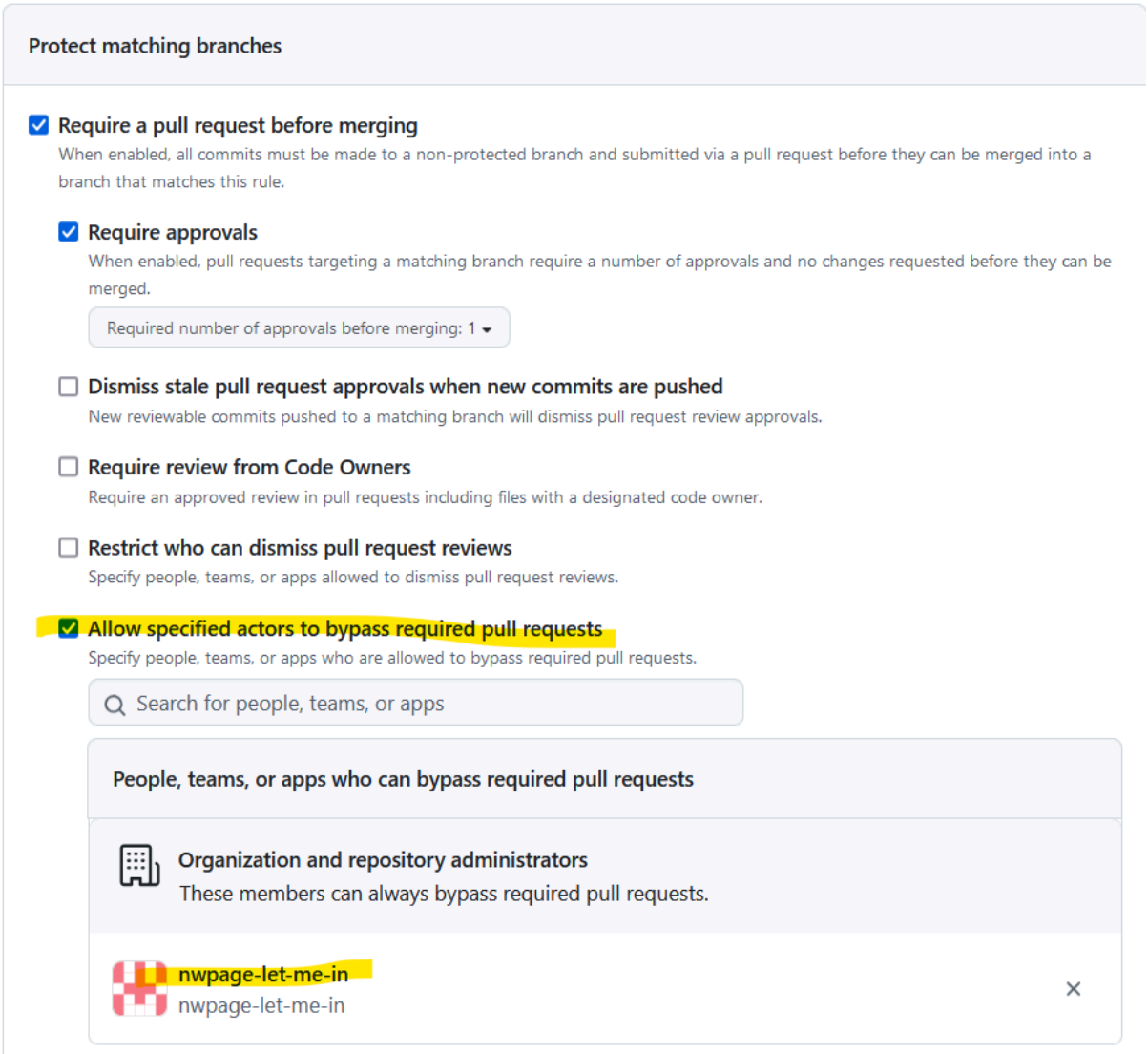
We're going to make two secrets:

1. `YOUR_GITHUB_APP_ID` - you can find this id when clicking on your app under `App ID:` , it should be like 110103202
2. `YOUR_GITHUB_APP_PRIVATE_KEY` - this is in the `.pem` file that you downloaded above. Copy everything including `-----BEGIN RSA PRIVATE KEY-----` and `-----END RSA PRIVATE KEY-----` . Then paste that into the secret

You won't be able to see these secrets again but you can overwrite them if needed.

### 7.5.b.e Step 2.5 (for all users) allow branch protection bypass

Go to Settings > Branches > select your main branch protection (click Edit) > make sure 'Require a pull request before merging' is selected > below it, check the option called 'Allow specified actors to bypass required pull requests' > add `nwpage-let-me-in`



#### 7.5.b.f Step 2.6 set up a github action

Now we need to use the github app in our github action. Add this step to you GitHub Action workflow:

```
steps:
  # Step 1: Create GitHub App Token
  - name: Create GitHub App Token
    id: app-token
    uses: actions/create-github-app-token@v1
    with:
      app-id: ${{ secrets.YOUR_GITHUB_APP_ID }}
      private-key: ${{ secrets.YOUR_GITHUB_APP_PRIVATE_KEY }}

  # Step 2: Checkout repository code using GitHub App Token
  - name: Check out repository code
    uses: actions/checkout@v4
    with:
      token: ${{ steps.app-token.outputs.token }}
  - run: echo "💡 The ${{ github.repository }} repository has been
cloned to the runner."
```

> ⚠️ Warning
>
> I had tons of problems with the github action making infinite loops and triggering over and over on push to main. **Make sure you don't have multiple steps that make commits!**

The full yaml will look something like this:

```
on:
  workflow_dispatch:
  push:
    branches:
      - main

name: Render and Publish

jobs:
  build-deploy:
    runs-on: ubuntu-latest
     if: github.actor != 'github-actions[bot]'   # Prevent action
loop
```

```
    env:
      RENV_PATHS_ROOT: ~/.cache/R/renv

    steps:
      # Step 1: Create GitHub App Token
      - name: Create GitHub App Token
        id: app-token
        uses: actions/create-github-app-token@v1
        with:
          app-id: ${{ secrets.YOUR_GITHUB_APP_ID }}
          private-key: ${{ secrets.YOUR_GITHUB_APP_PRIVATE_KEY }}

      # Step 2: Checkout repository code using GitHub App Token
      - name: Check out repository code
        uses: actions/checkout@v4
        with:
          token: ${{ steps.app-token.outputs.token }}
      - run: echo "💡 The ${{ github.repository }} repository has
been cloned to the runner."

other-steps-here:
```

# 8 Reproducibility

## 8.1 Objectives

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

## 8.2 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 possi-

ble in every scenario, but there are tools available in Github to make this possible for the majority of our repos.

## 8.3 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 (Integrated 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.

### 8.3.a 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

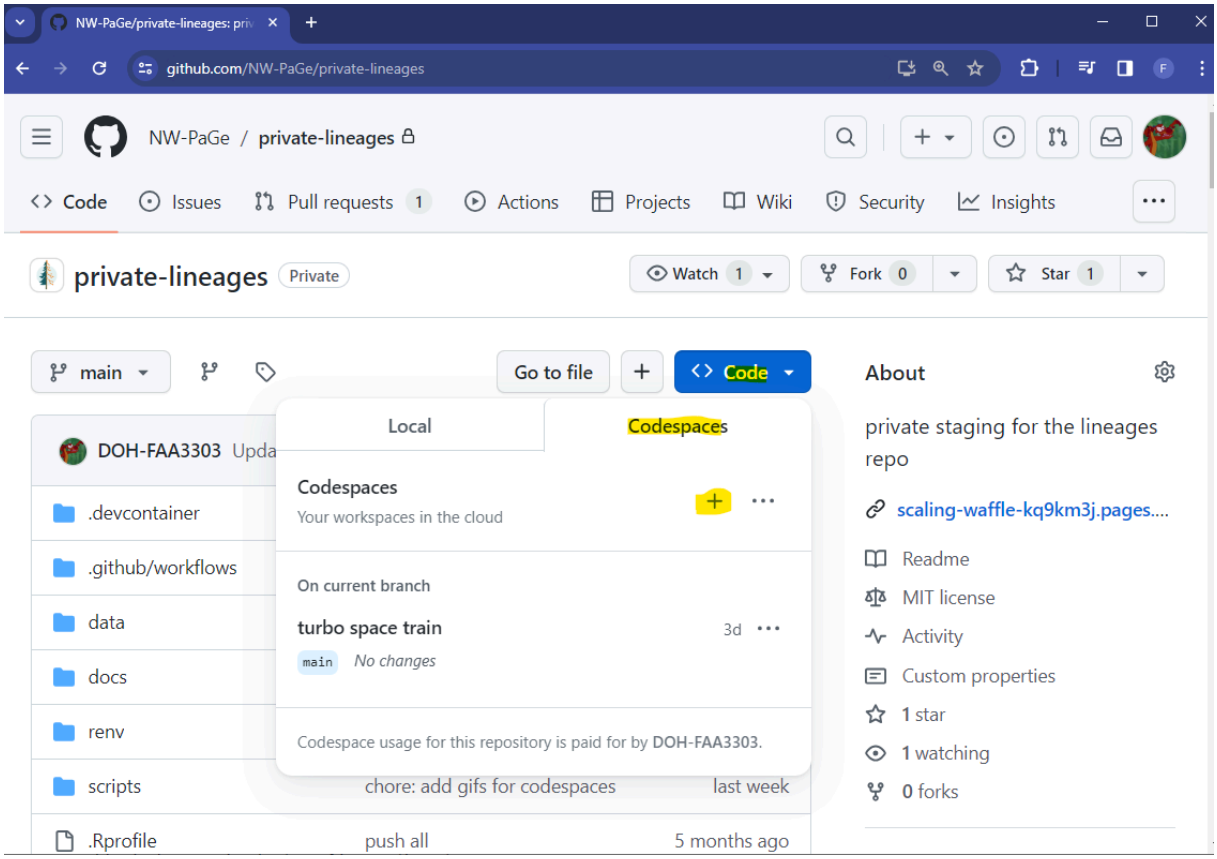



Figure 1 — open up a codespace

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:

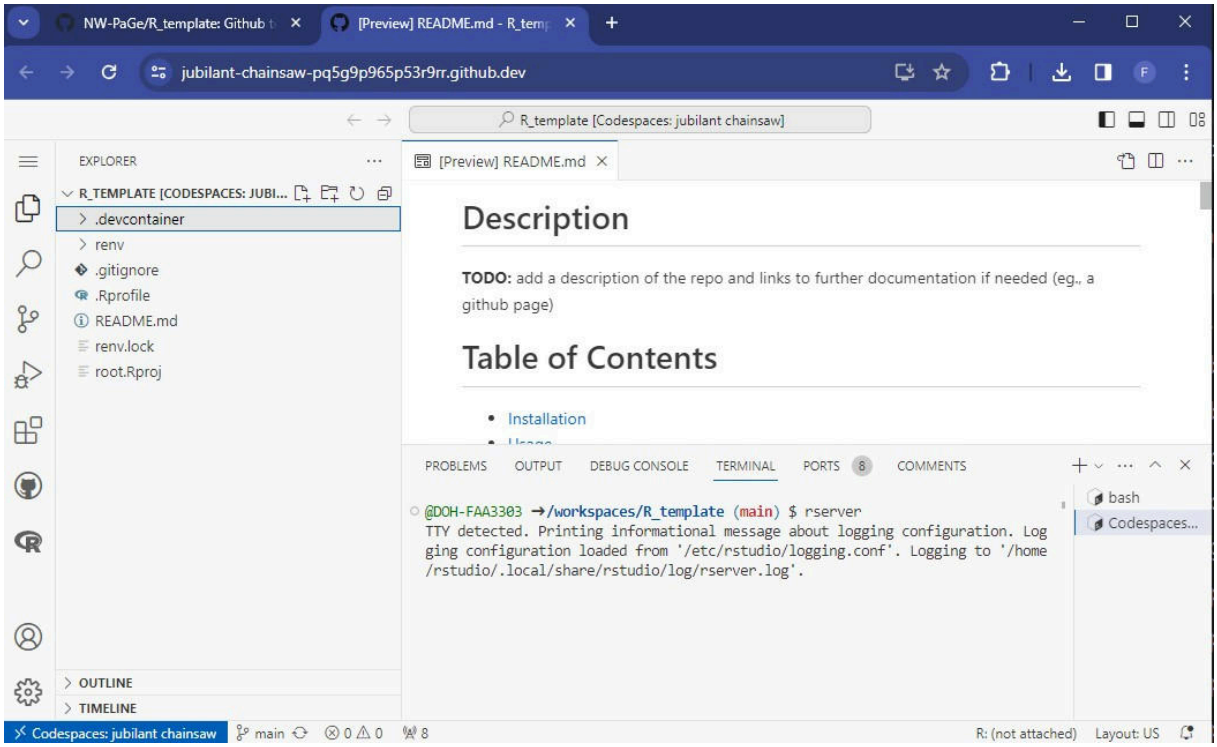



Figure 2 — VS Code IDE in Github Codespaces

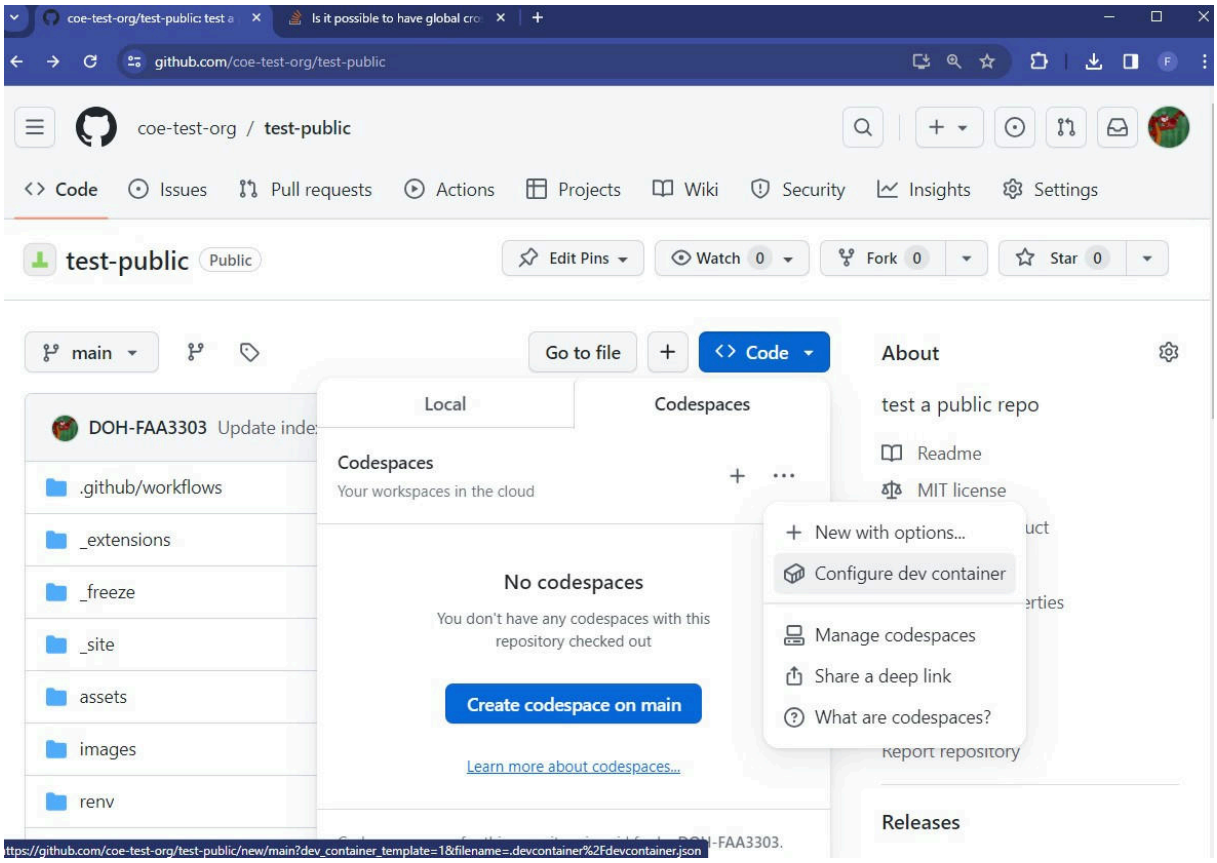
Here you can install most software. You can also customize the Codespace so that whenever someone opens one in your repo it will come with software pre-installed. More on that in the devcontainers section

### 8.3.b Devcontainers

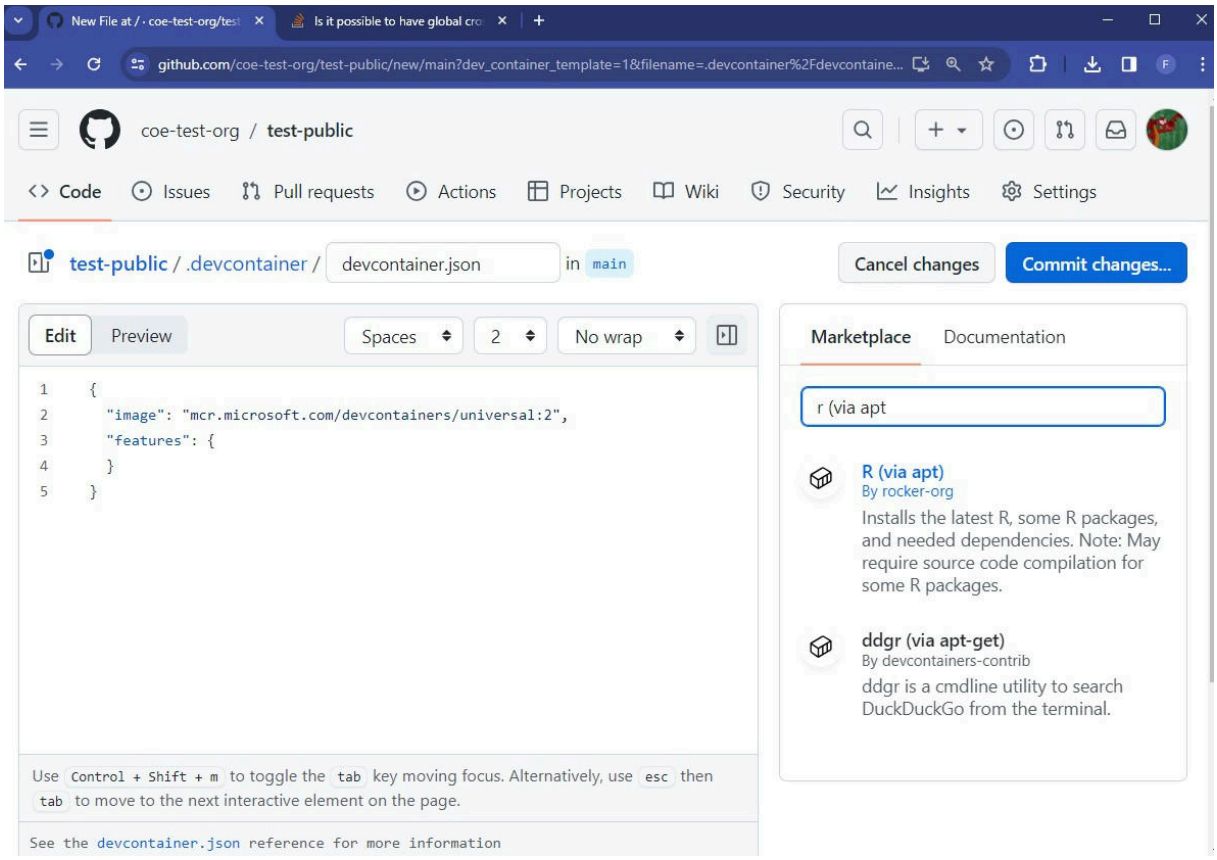
Devcontainers are a way to install software into a Codespace so that whenever a user opens up the Codespace they won't need to install anything themselves. Making a container can be a little tricky, so we've made Github templates that have devcontainers already made. See templates. There are R, Python,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

## 8.4 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.

## 8.5 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.

## 8.6 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.

# 9 Tutorials

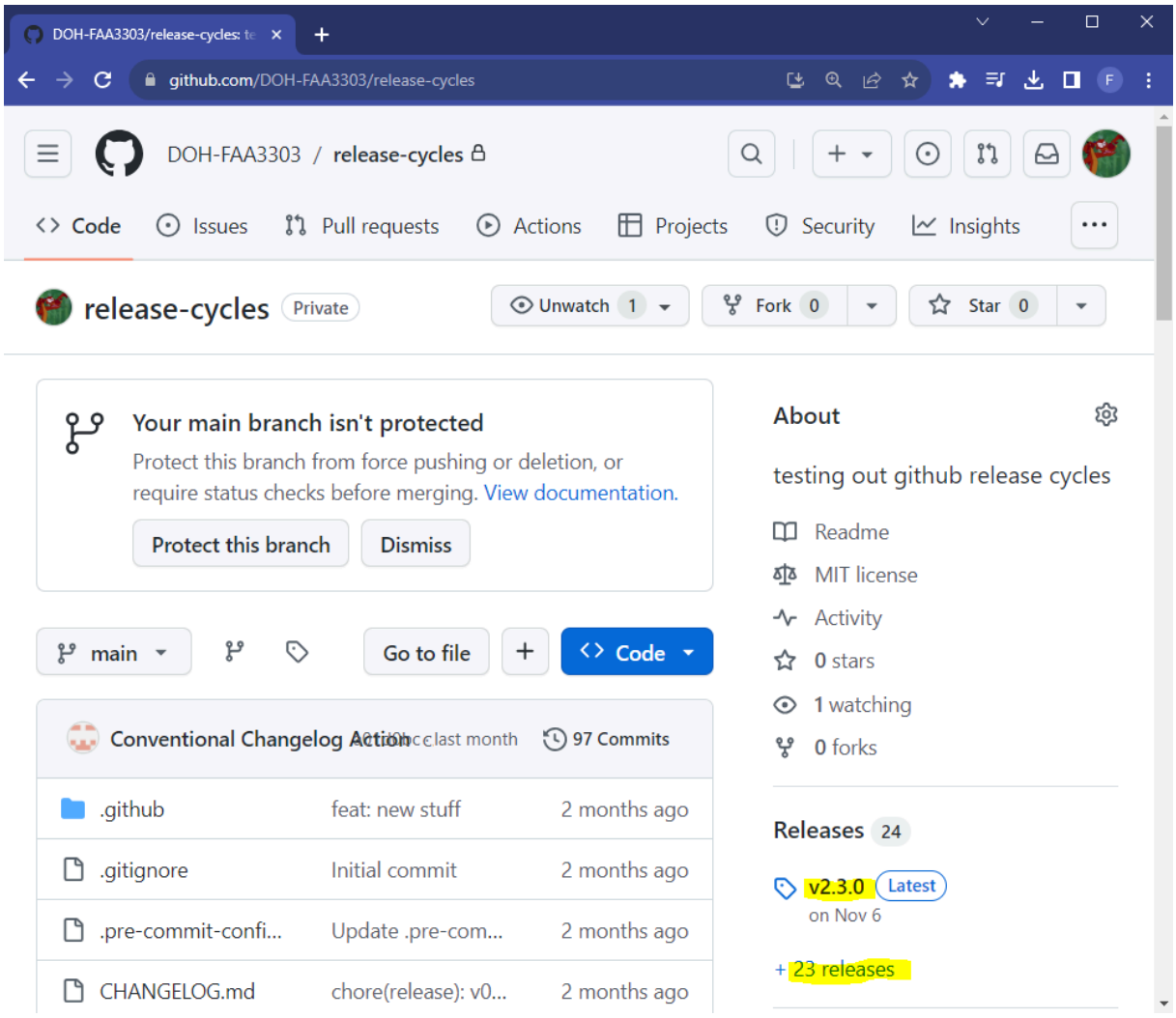
## 9.1 Release Cycles

**Summary**

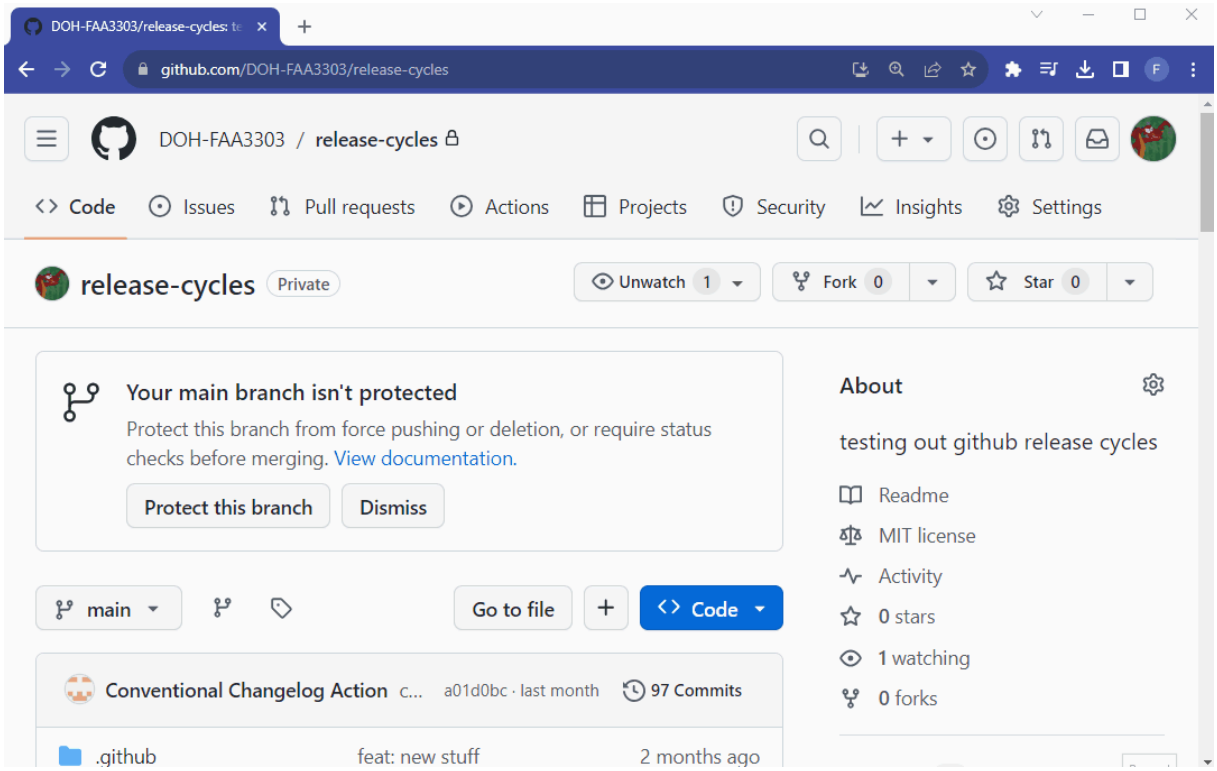
- Github Releases
- Helps devs and end users
- Changelogs and semantic versioning
- Automate the release process

# 10 Github Releases

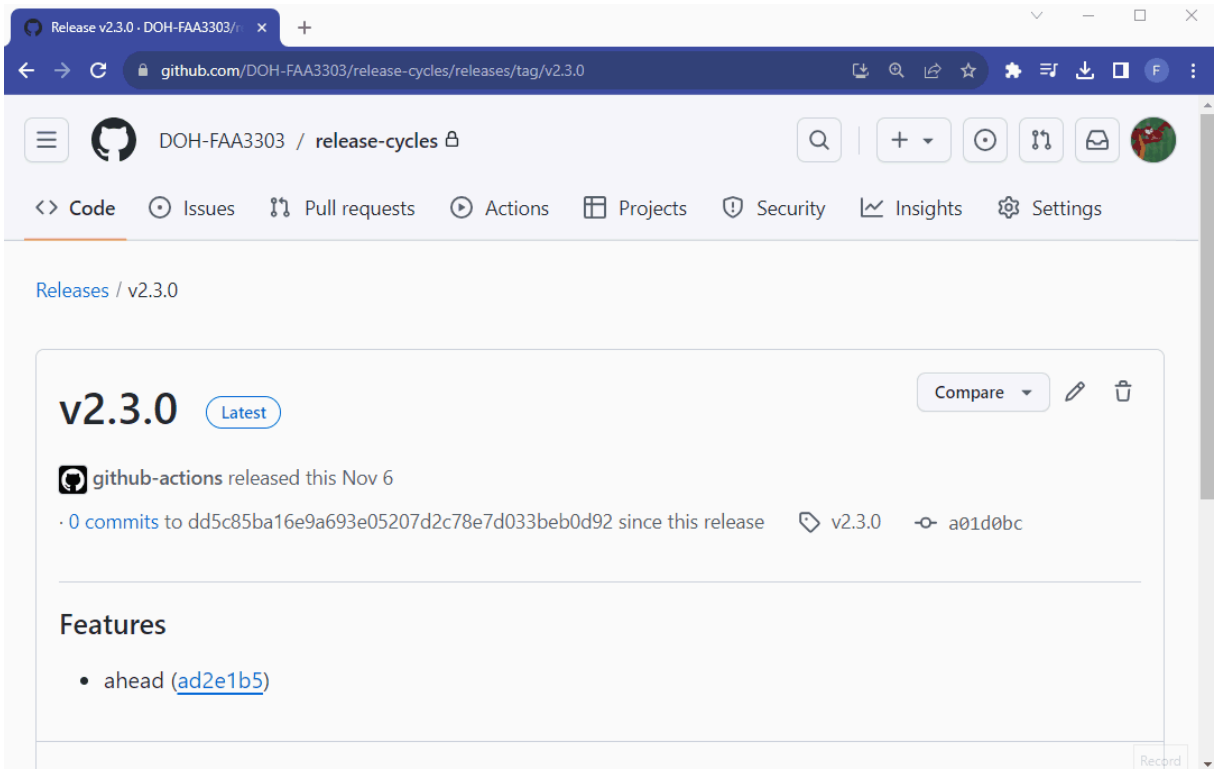
In the right panel of your Github repo there is a section labeled Releases. Here you can create or find a version of your repo's code base. Each version comes with a changelog, tags, and downloadable source code. Developers and end-users may find this helpful to navigate to what the repo contained at specific release versions and have the source code available for download at the specific version.



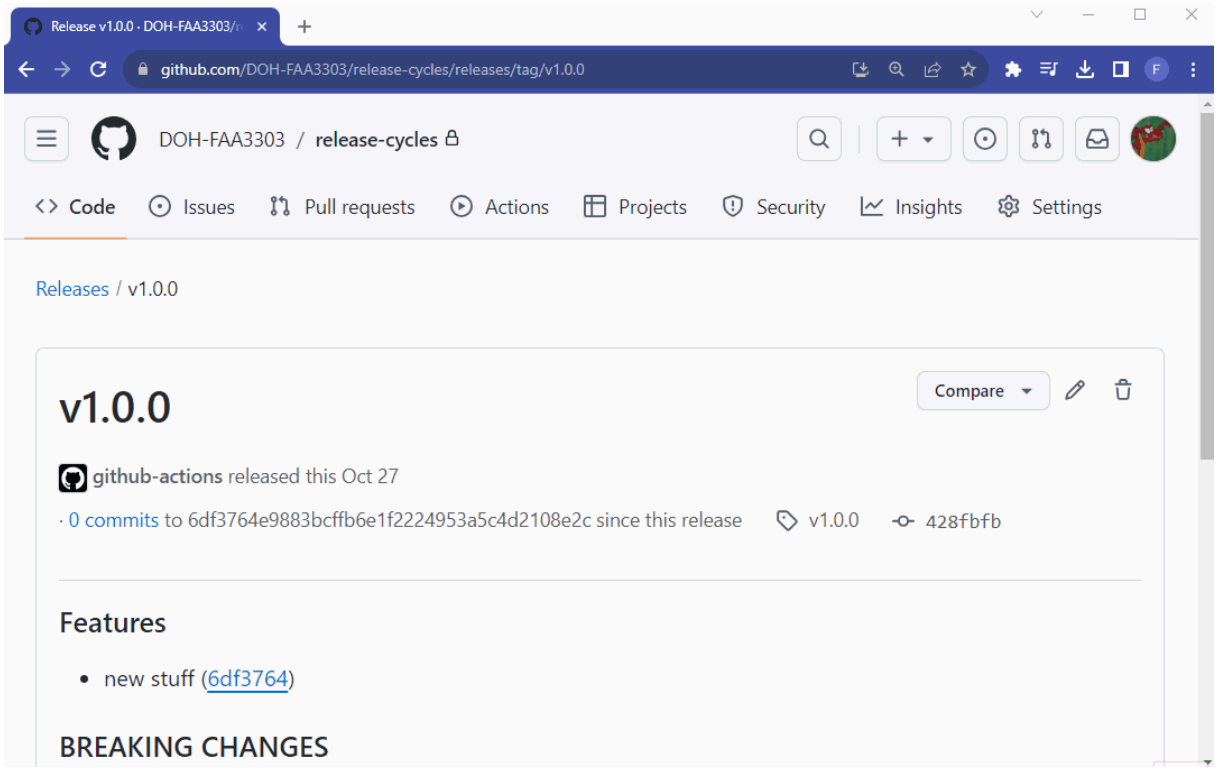

If you click on the releases you can see different release tags/versions. Each comes with a changelog, tag, git hash number, and zip files to download the repo *at the time the specific version was released*. This means you can automatically save repo snapshots and backups whenever your project cycle is released.




you can flip through different releases and tags here




You can click on a tag and it will take you to the repo *at the time the specific version was released*




# 11 Semantic Versioning

Software projects often label their releases using semantic versioning. It looks like this, where the software version numbers all have a definition:

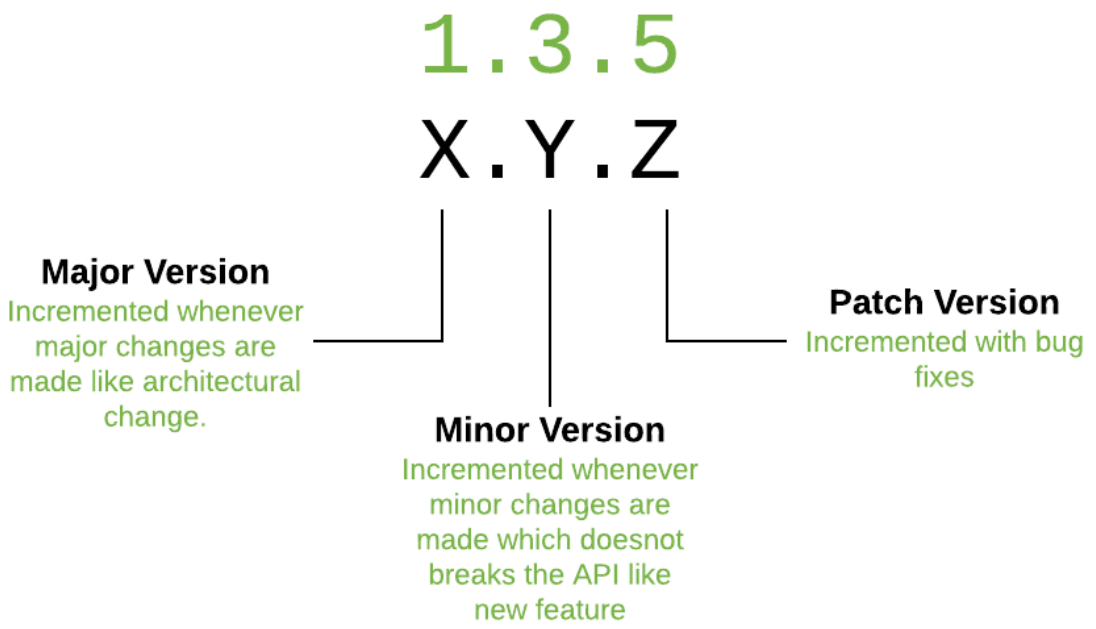



Figure 3 — https://www.softwarecraftsperson.com/2020/12/06/semantic-versioning-semver-introduction/

## 12 Conventional Commits

To create the release cycle in your repo you may want to use Conventional Commits.

Conventional Commits are a way to format and standardize your commit messages, which can be used to then automate the repo's release cycle. For example, one conventional naming method is to label any commit associated with a new feature as `feat:` plus a commit message.

- The word `feat:` can trigger a Github Action to add that commit to your changelog under the **Features** header,
- and it will up-version the minor release version number.
- So if you are on release 1.0.0, a new `feat` will up-version the cycle to 1.1.0
- Commit titles that start with the word `fix:` as in a bug fix will up-version the patch number of the, i.e. 1.0.0 to 1.0.1

## 13 Automating The Release Cycle

You should consider automating your release cycle so that your project cycle is consistent and predictable. There are many different ways to approach this.

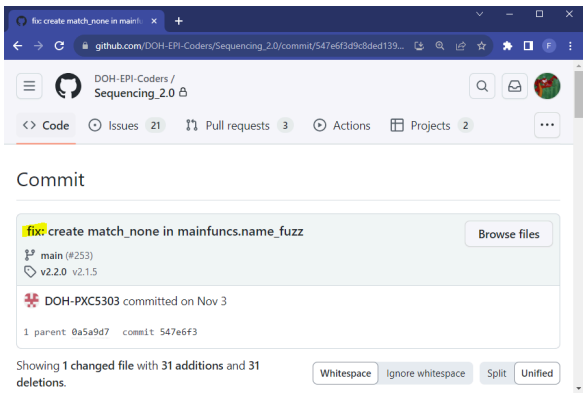
Some repos have semi-automatic cycles where there is some manual component of releasing their software, whereas others are fully automated. Manual releases can work too for some scenarios.

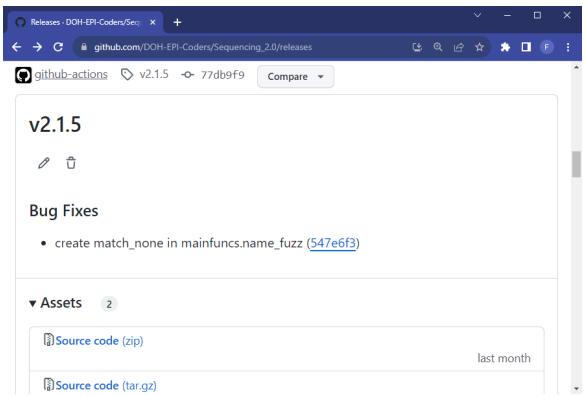
## 13.1 Github Action for auto releases

I recommend first creating a test repo for this. In the repo, create a Github Action workflow called `changelog.yml` . You can copy the full file below:

```yaml
name: Changelog
on:
  push:
    branches:
      - main

jobs:
  changelog:
    runs-on: ubuntu-latest

    permissions:
      # write permission is required to create a github release
      contents: write

    steps:
      - uses: actions/checkout@v2

      - name: Conventional Changelog Action
        id: changelog
        uses: TriPSs/conventional-changelog-action@v3
        with:
          github-token: ${{ secrets.github_token }}
          create-summary: true

      - name: Create Release
        uses: actions/create-release@v1
        if: ${{ steps.changelog.outputs.skipped == 'false' }}
        env:
          GITHUB_TOKEN: ${{ secrets.github_token }}
        with:
          prerelease: false
          tag_name: ${{ steps.changelog.outputs.tag }}
          release_name: ${{ steps.changelog.outputs.tag }}
          body: ${{ steps.changelog.outputs.clean_changelog }}
```

This workflow will be triggered everytime a branch is merged to main. If that branch has conventinal commit messages the commits will be summarized in the changelog. See an example workflow below:




If I make a branch off of main, I can add features, bug fixes, and more. If I used conventional commit messages in the title (i.e. `feat: message`, `fix: message`) the Github Action workflow will detect the trigger word in the title and divide the commit accordingly in the changelog. Notice how the commit title message gets output automatically into the changelog under the header **Bug Fixes** and the commit + commit hash number are generated.

A new version will be released, and since this was just a bug fix the version number went from `v2.1.4` to `v2.1.5` since bug fixes only up-version the patch numbers

The first step uses the Github Action `TriPSs/conventional-changelog-action@v3` which will scan your

```yaml
steps:
  - uses: actions/checkout@v2

  - name: Conventional Changelog Action
    id: changelog
    uses: TriPSs/conventional-changelog-action@v3
    with:
      github-token: ${{ secrets.github_token }}
      create-summary: true
```

The second step uses the Github Action `actions/create-release@v1` which will create git tags with the version number and a changelog with downloadable source code

```
- name: Create Release
  uses: actions/create-release@v1
  if: ${{ steps.changelog.outputs.skipped == 'false' }}
  env:
    GITHUB_TOKEN: ${{ secrets.github_token }}
  with:
    prerelease: false
    tag_name: ${{ steps.changelog.outputs.tag }}
    release_name: ${{ steps.changelog.outputs.tag }}
    body: ${{ steps.changelog.outputs.clean_changelog }}
```

# 14 Step by Step Instructions

1. Make a test repo in Github, then clone it locally
2. Make a folder in the root of the repo named `.github`
3. In the `.github` folder make another folder named `workflows` . the `.github/workflows` folder needs to be spelled the same and in the in exact same place for a github action to work.
4. In the `workflows` folder make a file called `changelog.yml` and paste code below in it:

```
name: Changelog
on:
  push:
    branches:
      - main

jobs:
  changelog:
    runs-on: ubuntu-latest

    permissions:
      # write permission is required to create a github release
      contents: write

    steps:
      - uses: actions/checkout@v2
```

```
      - name: Conventional Changelog Action
        id: changelog
        uses: TriPSs/conventional-changelog-action@v3
        with:
          github-token: ${{ secrets.github_token }}
          create-summary: true

      - name: Create Release
        uses: actions/create-release@v1
        if: ${{ steps.changelog.outputs.skipped == 'false' }}
        env:
          GITHUB_TOKEN: ${{ secrets.github_token }}
        with:
          prerelease: false
          tag_name: ${{ steps.changelog.outputs.tag }}
          release_name: ${{ steps.changelog.outputs.tag }}
          body: ${{ steps.changelog.outputs.clean_changelog }}
```

> **Note**
>
> Now when you commit something to the repo and have a convential commit message it will trigger the Github Action to make a changelog.
>
> For example, if you make a commit with a bug fix and in the commit title you write the word `fix:` like this - `fix: this commit is a bugfix` - the action will get triggered to make a new changelog with the commit and bump up the minor version (like from v1.0.0 to v1.0.1)

**This is great, but if you do not write a conventional commit message it will not work**. Step 5 has instructions on how to make sure everyone on your team writes conventional commits. It uses a pre-commit hook, which will prevent any commit that does not have a conventional commit message from being pushed to the repo.

For example, this commit message will error out and not be pushed to the repo - `this commit is a feature` . But this commit message will work - `feat: this commit is a feature`

5. (Optional, but recommended) Make a pre-commit hook
    - Open a Windows Command Prompt
    - Install pre-commit with the code below

```
pip install pre-commit
```

- Check that pre-commit is working by running the code below. If you get an error message with this you may need to install `pip`

```
pre-commit --version
```

- In the **root** of you repo, make a file called `.pre-commit-config.yaml` and put the following code in it (note the `.` is necessary in the file name):

```
repos:
- repo: https://github.com/compilerla/conventional-pre-commit
  rev: v2.4.0
  hooks:
    - id: conventional-pre-commit
      stages: [commit-msg]
      args: []
  # args is optional: list of Conventional Commits types to
allow e.g. [feat, fix, ci, chore, test]
```

- Last, in the Windows Command Prompt, run this:

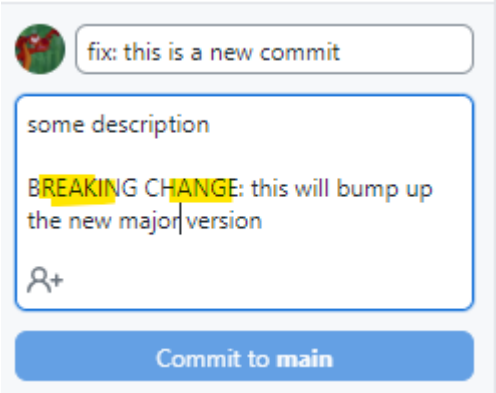
```
pre-commit install --hook-type commit-msg
```

Now you will be required to use conventional commit messages when pushing new code to the repo.

Here's a common guide:

| conventional commit keyword | expected outcome |
|---|---|
| feat: | new changelog, bump up the minor release version (i.e, 1.0.0 to 1.1.0) |
| fix: | new changelog, bump up the patch release version (i.e, 1.0.0 to 1.0.1) |
| chore:<br>docs:<br>test: | no changes |
| any keyword plus `BREAKING CHANGE:` in the commit message | new changelog, bump up the MAJOR release version (i.e, 1.0.0 to 2.0.0) |

**NOTE**: breaking changes need to be written like this:




# 15 Resources

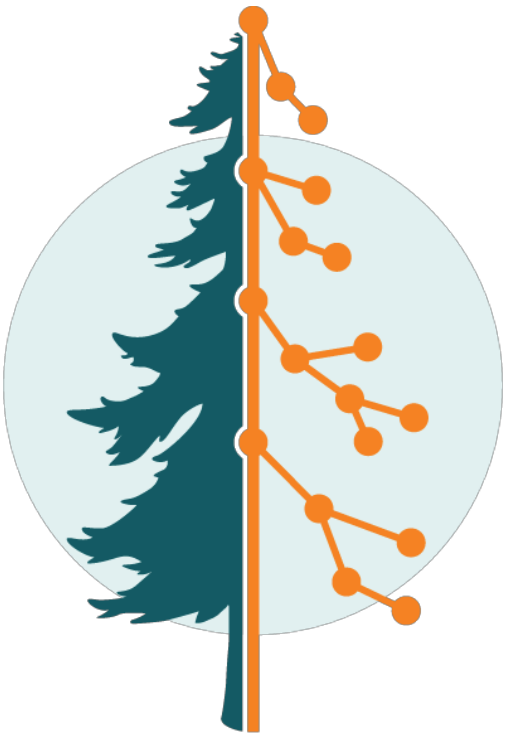
These videos are excellent summaries of how to use Github Releases and semantic versioning

https://www.youtube.com/embed/fcHJZ4pMzBs?si=jb-1lgaYevGqOUU9

https://www.youtube.com/embed/q3qE2nJRuYM?si=7nQ7itJFt3oUtgaL

## 15.1 Virtual Environments

# 16 Introduction



Virtual environments allow us to execute code while accounting for software/package version differences we have on our local machines. This repo uses virtual environments to configure a user's R and Python software and packages to the repo's specific package versions.

For example, say you have dplyr version 2.0 but this repo uses dplyr version 1.1, you may not be able to run the scripts as intended by the author since the functions in dplyr 1.1 may be different than in 2.0. The virtual environment will allow a given user to use only this repo's version of dplyr so they can run the code as intended.

> ! Important
>
> There are two different virtual environments for this repo, one for R and another for Python. **Your workflows for opening R and Python and how you install packages needs to utilize the virtual environments so that all machines are able to use your code.**

# 17 Python - Conda Environment Setup

This repo has a file named `environment.yml` . It contains a list of all packages and versions used for Python software. This file can be used as a set of instructions for your local machine when configuring your local environment.

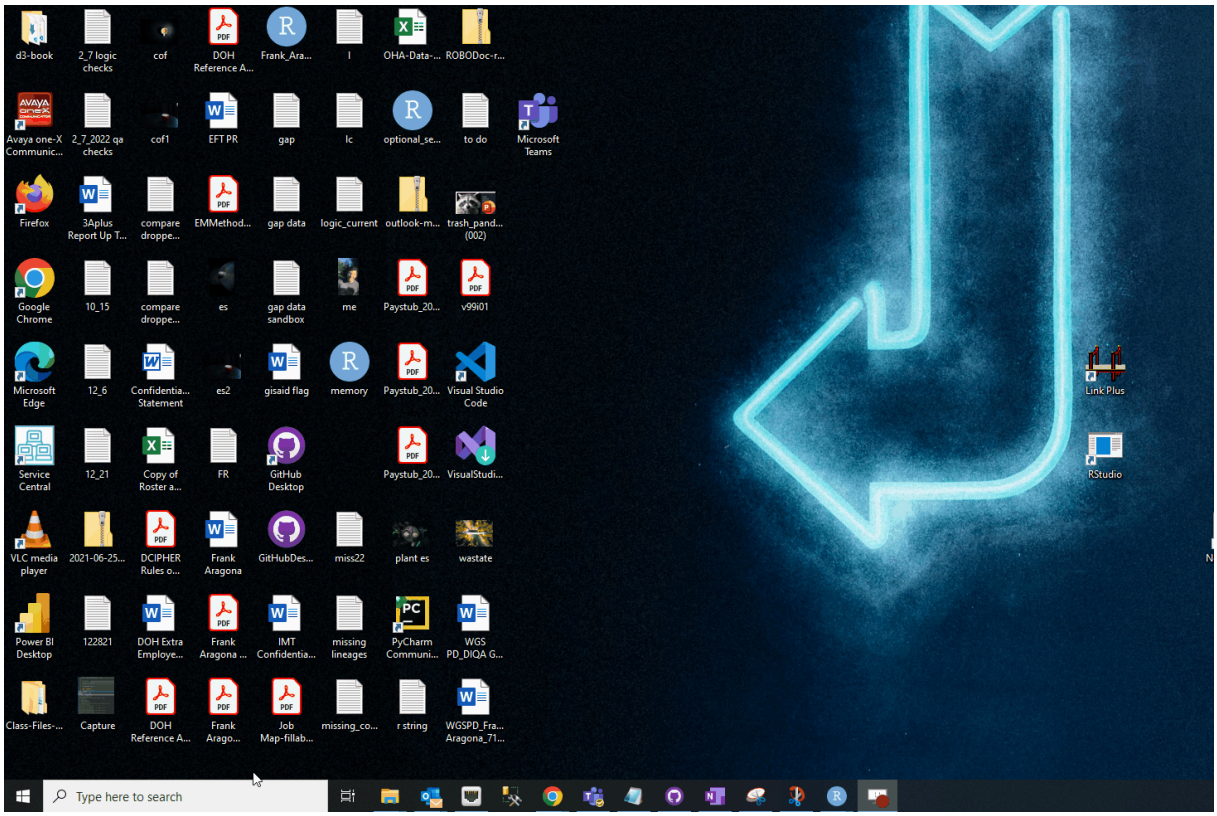
## 17.1 Step 1: Set Up Anaconda

You should already have Anaconda installed on your machine if not..

under construction

## 17.2 Step 2: Open Anaconda Prompt

You may have different Anaconda prompts (prompts aligned with different shells, like PowerShell, bash, etc). There should be a generic Anaconda prompt. Open that one:




If the first line in the prompt doesn't start with `(base)` , write:

```
conda deactivate
```

and it will bring you back to your base environment.

## 17.3 Step 3: Change Directories

Change the directory of the prompt to the repo's directory. The code is

```
cd C:/Users/XXXXXXX/Projects/Sequencing_2.0
```

If you are already in your user directory, you can just type

```
cd projects/sequencing_2.0
```

capitalization doesn't matter


Notice that the folder path is now changed to the sequencing repo folder.

## 17.4 Step 4: Copy the repo env

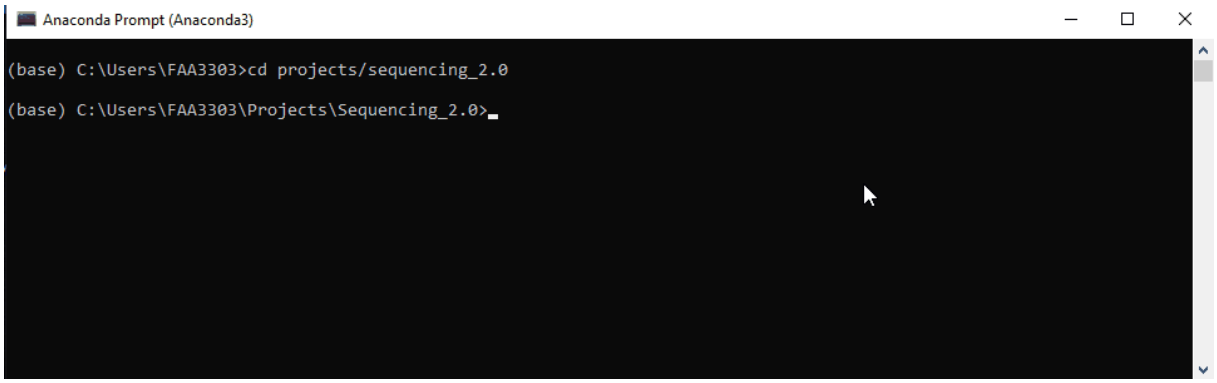
Now we're ready to create a new environment based on the repo's environment.

Type: `conda env create --name seq_env --file=environment.yml`

Note that how you name your environment doesn't really matter, but name it something that resembles the repo. This will save the headache of having random environments for random repos that you can't remember..

- `conda env create` will create a new environment in your `C:/Users/XXXXX/Anaconda3/envs` file path
- `--name` or `-n` will name that environment, in this case `seq_env`
- `--file=environment.yml` this code will take the file in the sequencing 2.0 repo and use it to create this environment. It is essentially a copy of the software versions in the file.

Note: I sped up the gif below. The whole process may take a few minutes



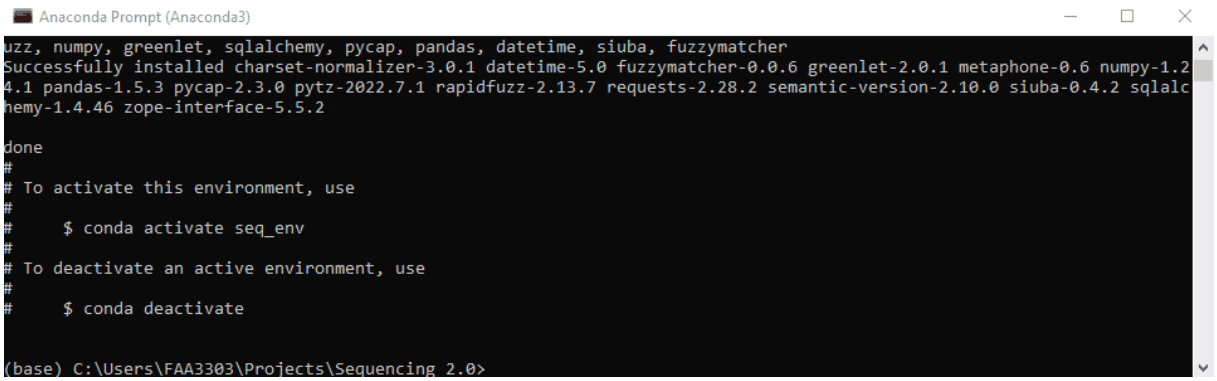

## 17.5 Step 5: Activate the environment

You can switch between environments in the conda prompt or in a programming IDE (or both? idk). To activate and switch the env, write:

`conda activate <env_name>`

in this case

`conda activate seq_env`




> **Note**
>
> The environment your in will show on the left of the prompt message. In this case it says `(seq_env)` instead of `(base)`. That way you know what env your working in

# 18 Python - Programming Setup

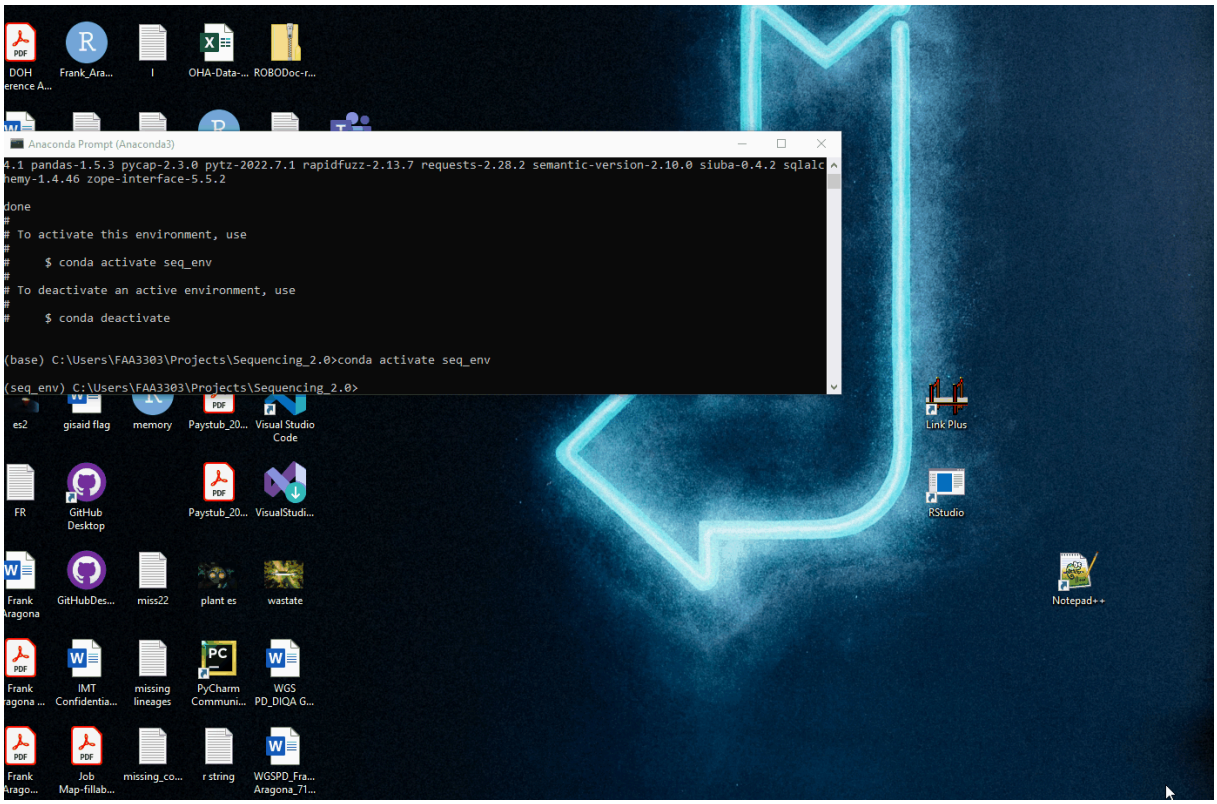
Now that your virtual conda environment is set up, let's set up your programming IDE to be able to recognize your environment.

## 18.1 IDE Setup - VS Code

VS Code has a lot built in to use a conda environment. Since your env is already activated, if you have VS Code installed, you can type

```
code
```

into the anaconda prompt and it will open a VS Code window



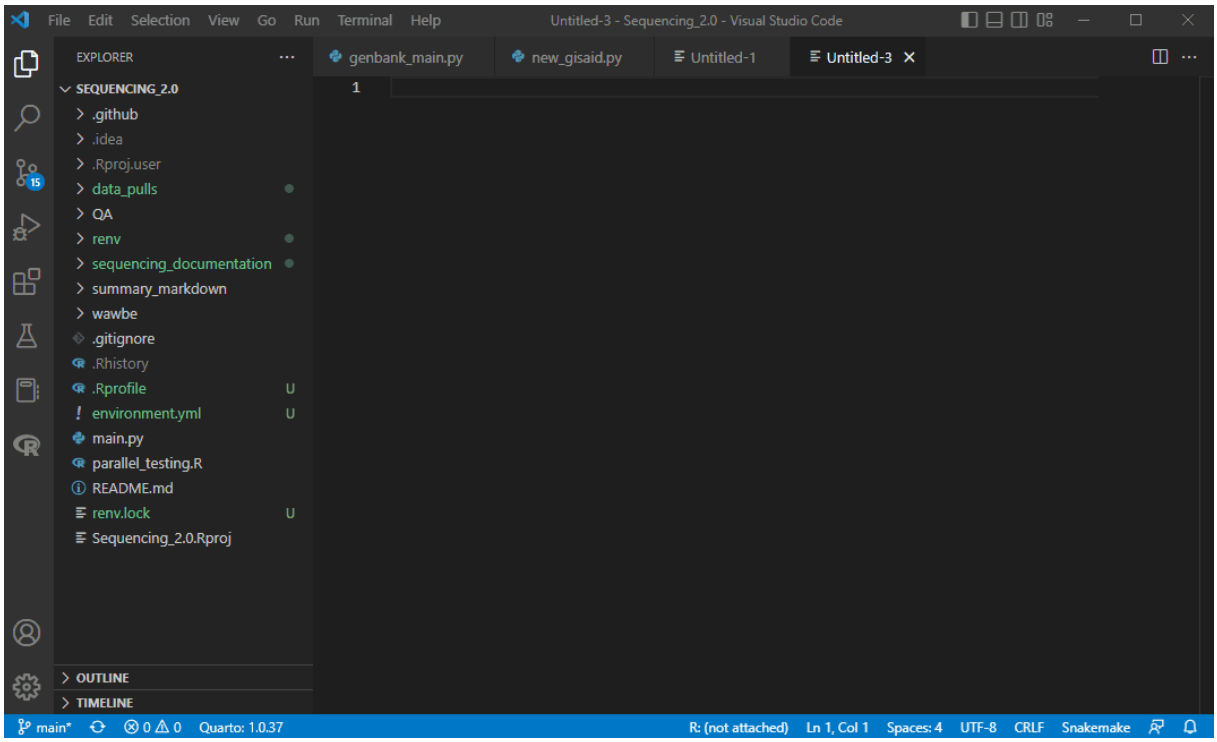

### 18.1.a Step 1: Select a Python Interpreter

First we need to select a python interpreter, which is in our env. On your keyboard, press

```
CTRL+SHIFT+P
```

This should bring up a window with an option that says `Python: Select Interpreter` . You may need to search for it.

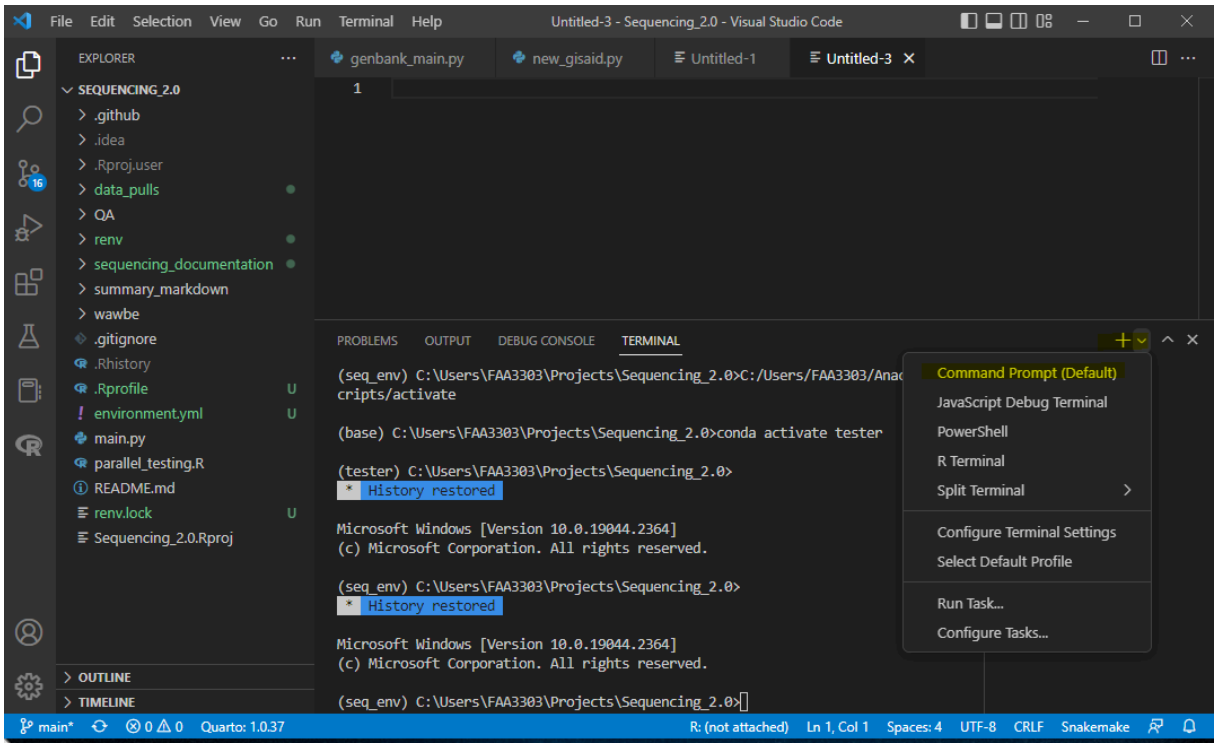
Click it and you should see your new environment `seq_env` in the list. Click it



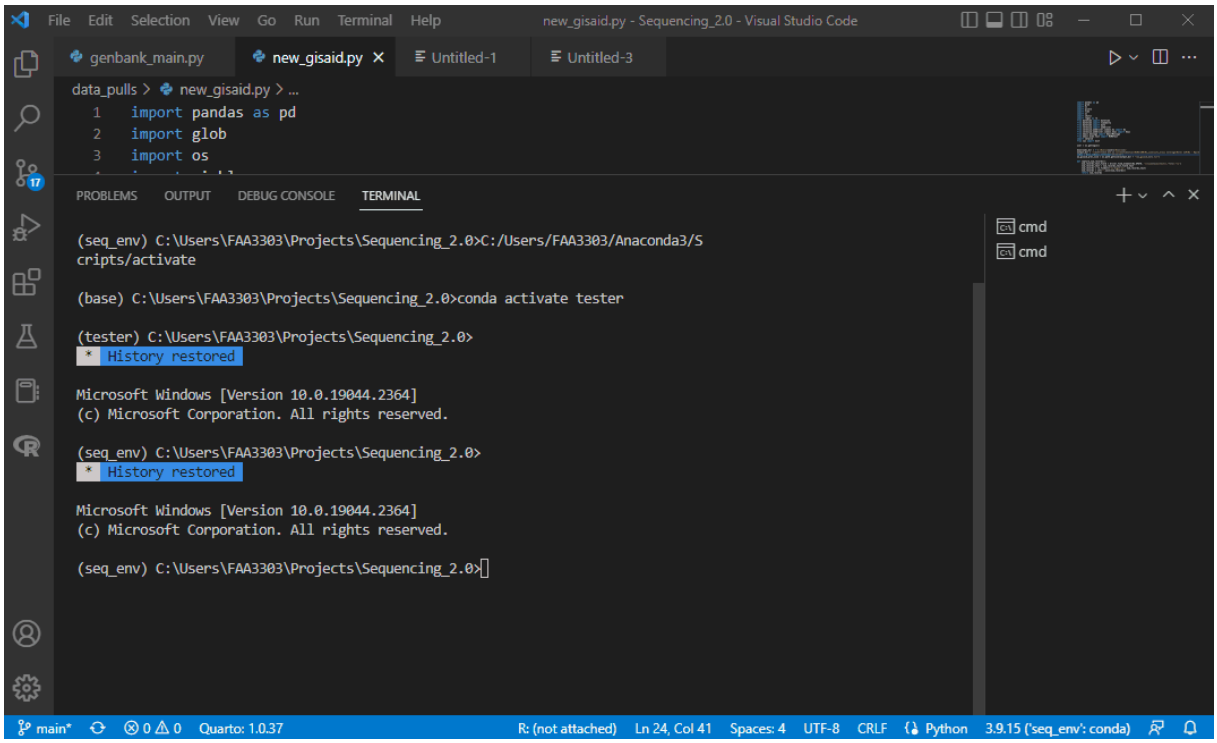

### 18.1.b Step 2: Write code

Now your VS Code is using your environment and the python version/packages in that environment. Check to see that your terminal is using the correct env.

Open the terminal (terminal > new terminal) and confirm that you are in a cmd prompt in the terminal. On the right side of the terminal it should say `cmd` . If it says `powershell` or something else, let's change it.. See the pic below. There's a drop down that gives you shell types. Change your default to Command Prompt




Also notice in the picture that my environment now switches to `seq_env` . Yours should do the same. You should now be able to run code in a python script. Notice that your terminal will change to run python. If you get an error, write `python` in the terminal and hit enter. It will change your terminal a little. Now you can run python code and it will output to this terminal.



## 18.2 IDE Setup - PyCharm

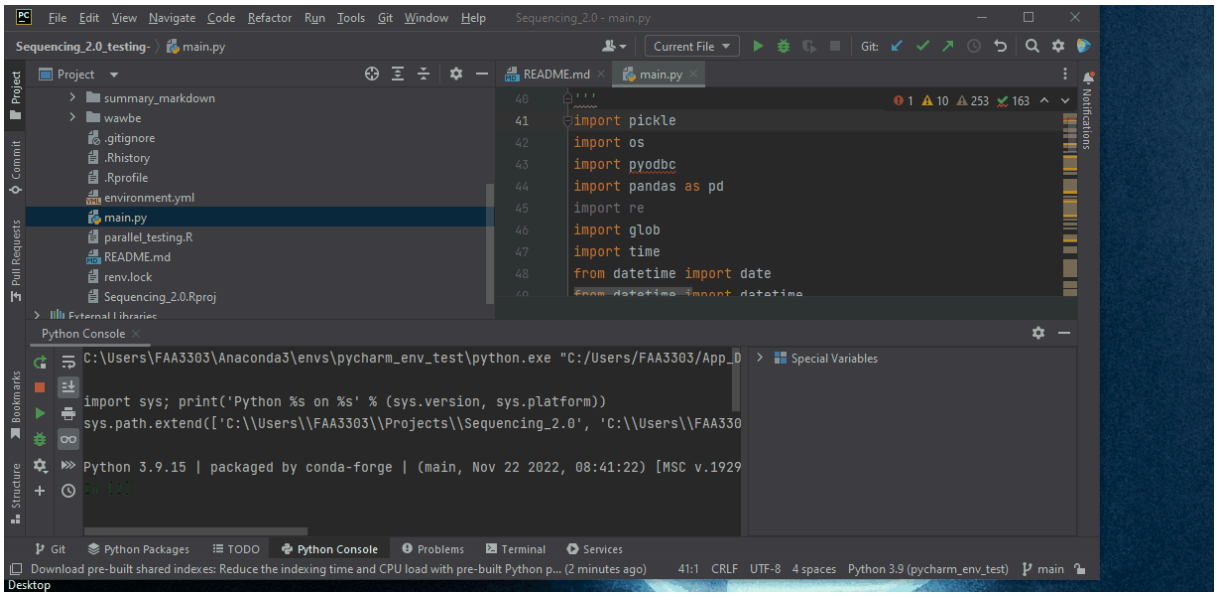
PyCharm also works great with a conda environment.

### 18.2.a Step 1: Select a Python Interpreter

You may also be able to open a PyCharm window from an Anaconda prompt like with VS Code (if it's installed in your env). To do so, write `pycharm` in the prompt and it should open a new window with the env activated.

If that doesn't work, open PyCharm and on the bottom right there is a python version and interpreter selected. Click it and open "Add New Interpreter" > "Add local interpreter". This opens a new window. Click "Conda Environment" and under "Interpreter" click the dropdown. You should be able to see your new environment there. If not, click away and click the dropdown again. It's weird sometimes.

Then click okay. Close and reopen the Python Console window and it should have your environment path for the python.exe. Also, the Python Libraries window should have all of the libraries in your environment now.




Notice that now there are a list of interpreters for you to use. You can now switch back and forth between environments. This is great if you have other repos to use or want to test out new packages that aren't in the main environment.

# 19 Python - Installing New Packages

Let's say you want to add a new python package to the repo. I recommend doing this in an Anaconda prompt and then saving it over the environment.yml. Then

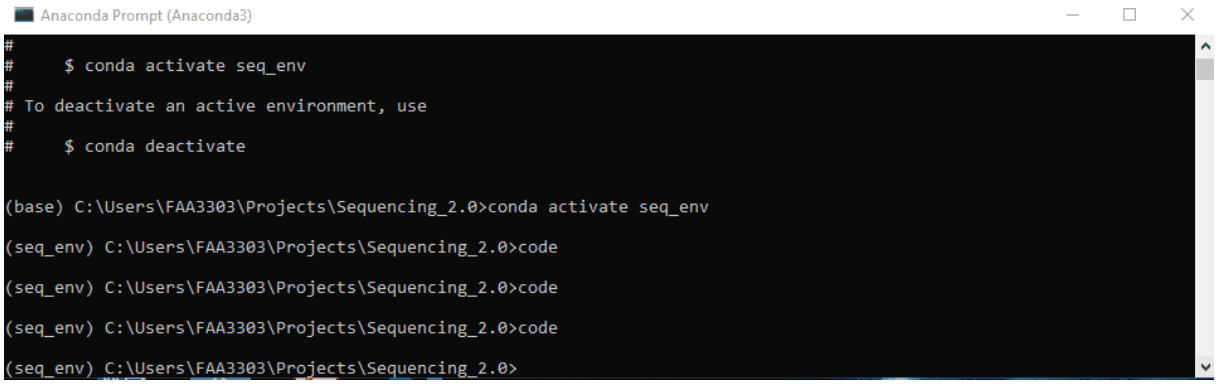
you can push the new environment.yml with the new changes to github. Use these steps:

## 19.1 Step 1: Install a new package

Go to the Anaconda prompt, make sure you're in the repo file path ( `cd projects/sequencing_2.0` ) and make sure you're in the right conda env ( `conda activate seq_env` ).

Now, install the package. Usually packages can be installed with `pip install` or `conda install` or `conda install -c conda forge <package>` . This depends on the packages. Some need pip, others need conda. Google it to find out. Here i'm going to download a package from NCBI to demonstrate. The package is called `ncbi-datasets-cli` .

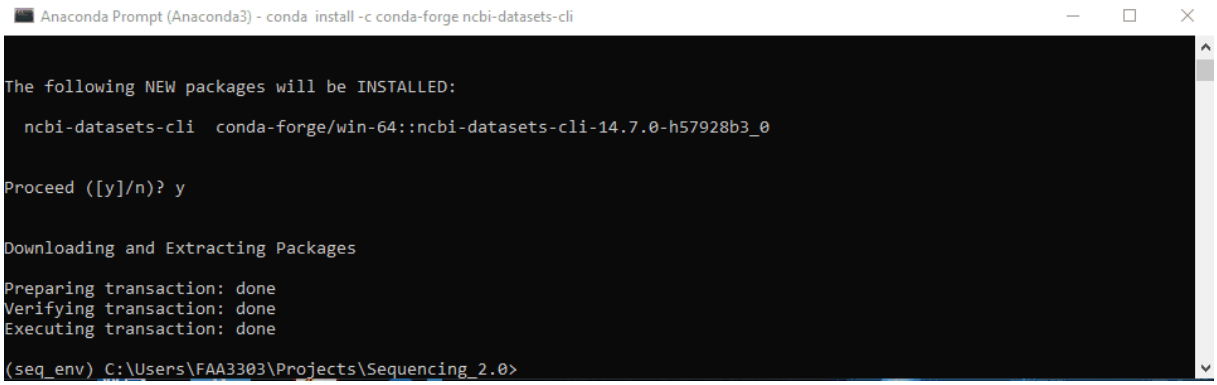
- This package uses `conda-forge` to install. Type in `conda install -c conda-forge ncbi-datasets-cli`
- It will give you a message Y/N to confirm. Type "y" and enter




## 19.2 Step 2: Save the package to the repo

Now we need to save this package to the repo's `environment.yml`

- Type `conda env export > environment.yml`
- Since the package is in your environment, this code is exporting your new environment to repo's one.
- Now push to github




# 20 R Package Management - `renv`

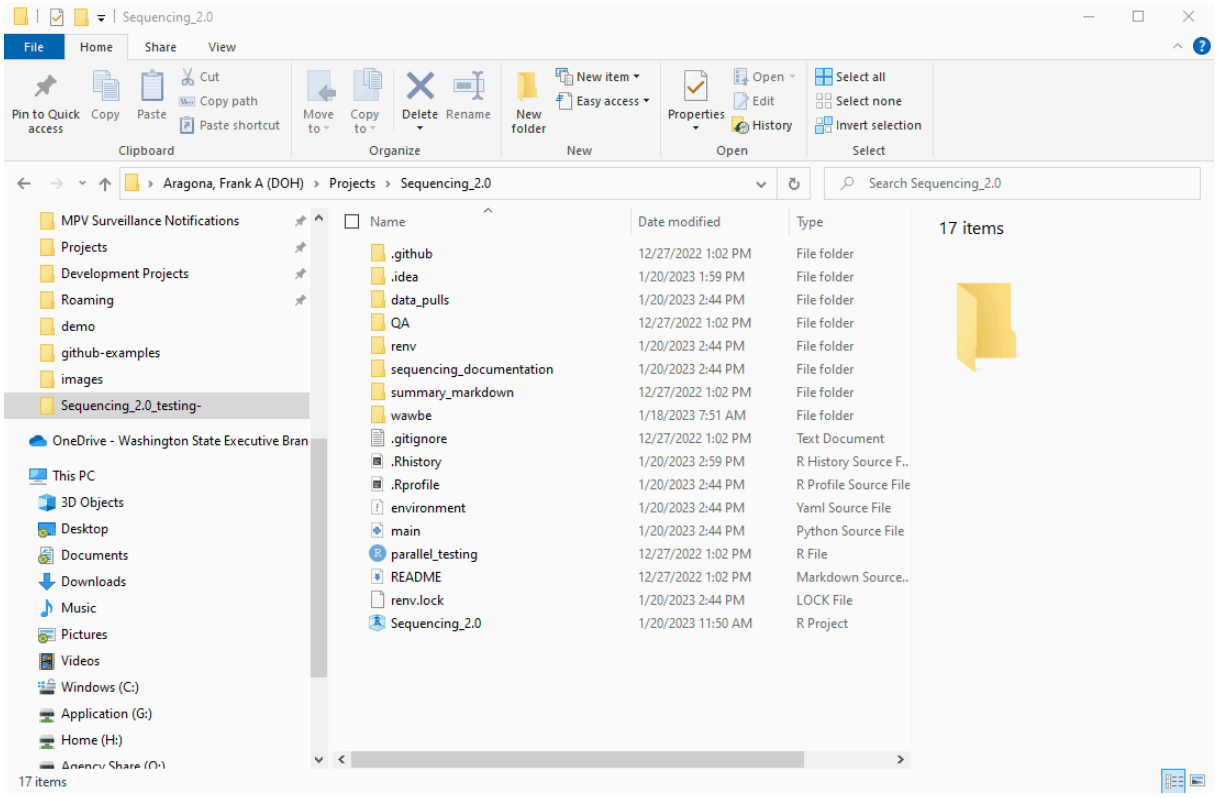
Managing R packages is much easier than managing Python packages. The R environment uses a package called `renv`. It will save a list of packages to a lock file, similar to `environment.yml`. Then, every time you or another teammate opens the R project in your repo the `renv` package will activate in the background and determine if any packages are not aligned with the repo's lock file. **It will ensure that everyone is using the same package versions**

> ! Important
>
> Your repo should have a `.Rproj` file at the *root* of the directory. If it doesn't you can create it by opening `Rstudio > File > New Project... > Existing Directory (or New Directory)`
>
> **Make sure `.Rproj` files are NOT in your `.gitignore`**

## 20.1 Creating `renv` in a project

### 20.1.a Step 1: Open the `.Rproj` in your repo

The R project will open up Rstudio at the root of you directory path.

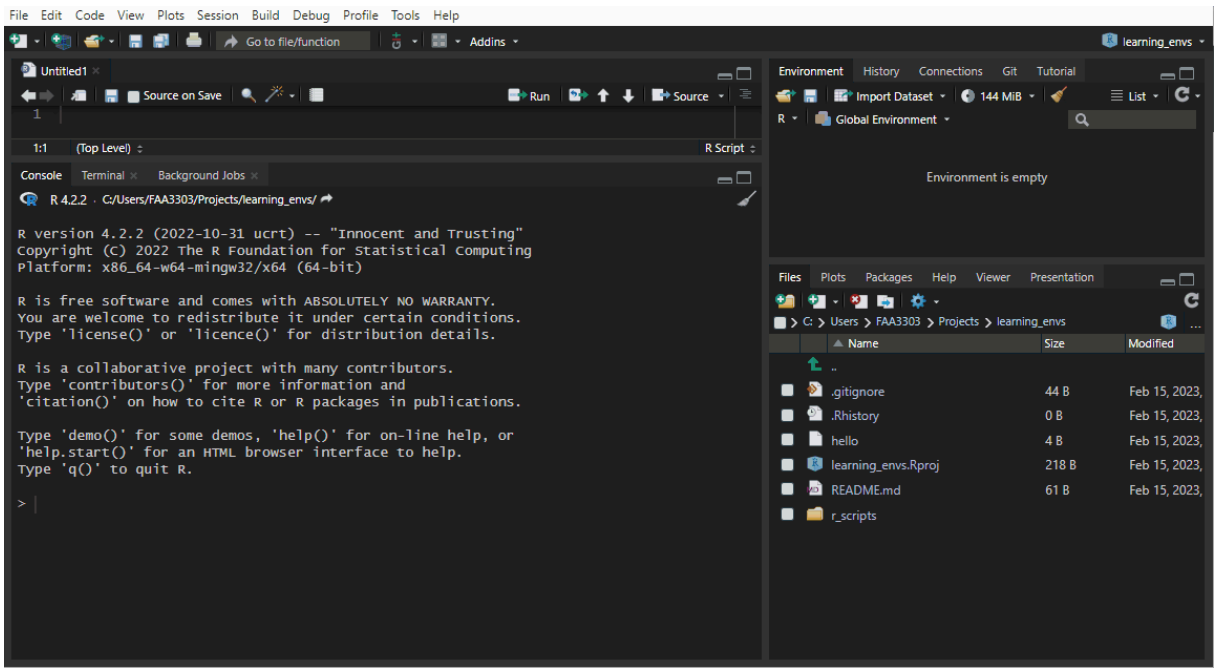



## 20.1.b Step 2: Initialize `renv` for the repo

Now that we're in the root of your repo directory, let's initialize `renv` .

First install `renv` - `install.packages("renv")`

Then in your console write `renv::init()` and run it.

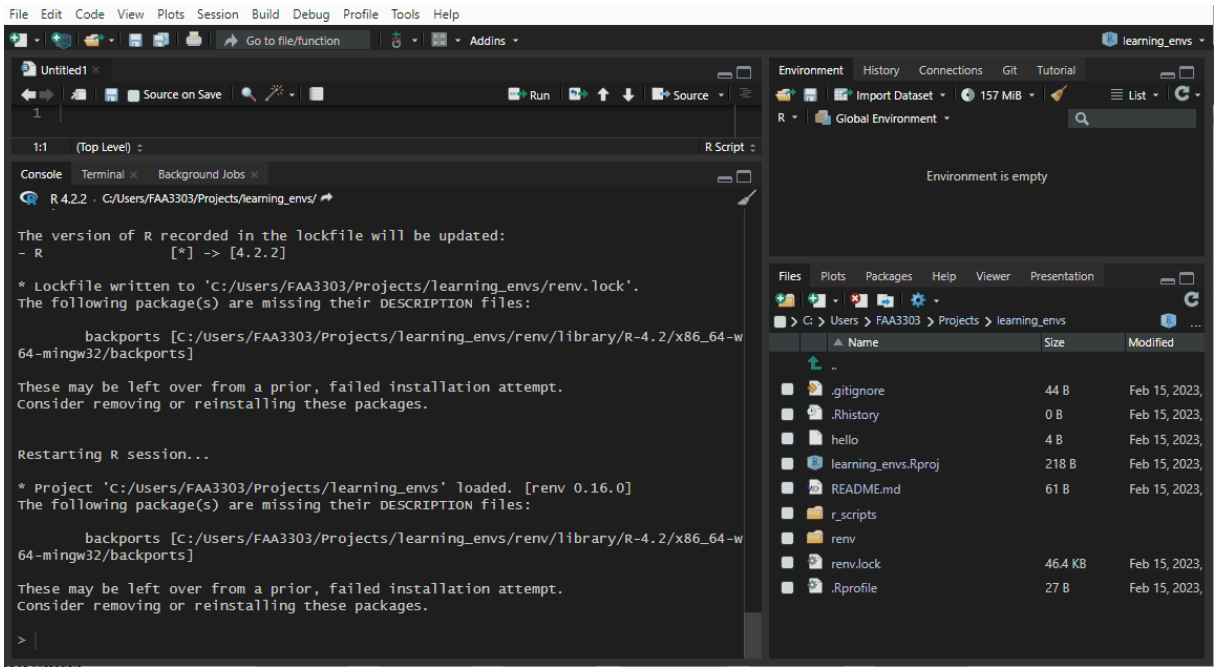


If you already have an existing repo, you will probably see warnings and errors in the renv::init like I did in the gif above. Not to worry! Read the warnings and follow the instructions. Usually you will need to re-install a package. If you get this warning:

```
These may be left over from a prior, failed installation attempt.
Consider removing or reinstalling these packages.
```

- Then run `renv::install("THAT PACKAGE")`. It will install the package again,
- and then you need to update the lock file (more on that later) by running `renv::snapshot()`.

Now the package will be installed correctly




`renv::init()` will:

1. Search through all R scripts in your repo and find all packages used
2. Create a snapshot of those packages
3. Save all packages in the repo in a new `renv` libraries path (similar to your C drive R libraries paths)
4. Create a `.gitignore` within the `renv` libraries path so that you don't get spammed with thousands of libraries in your git commit
5. Create a lock file - this is like the `environment.yml` for conda. Think of it as instructions for which packages your repo is using

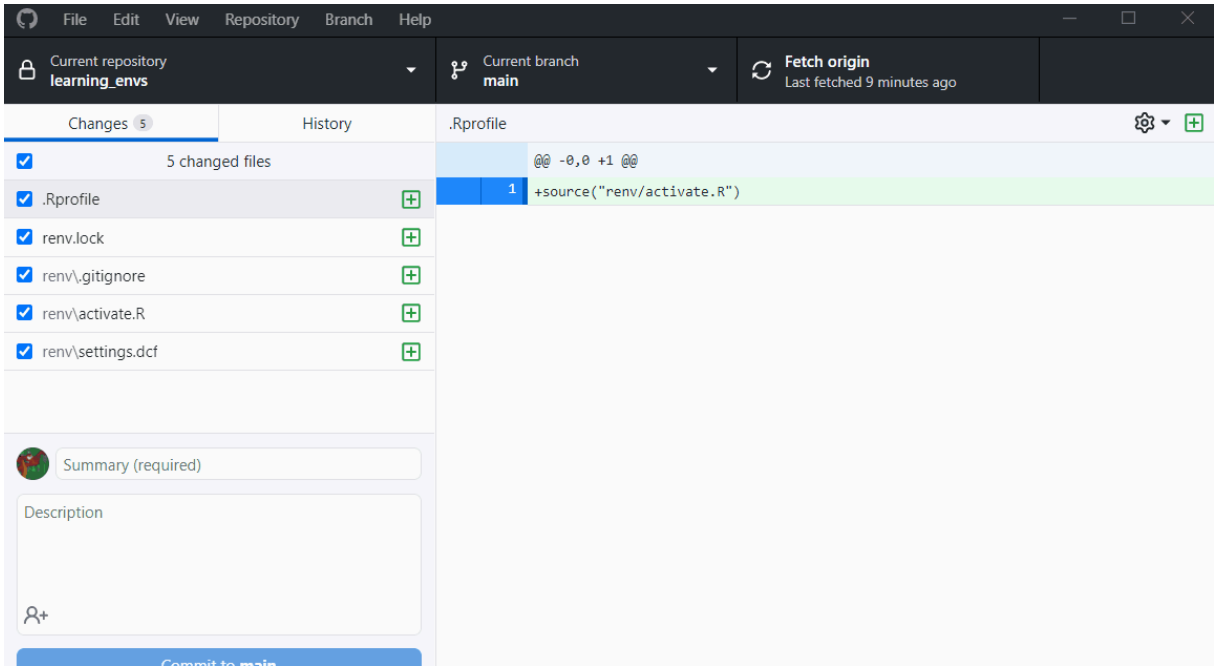
6. It also saved things like an `activate` R script which will activate that `renv` every time the repo is opened from the `.Rproj`

### 20.1.c Step 3: Push to Github

Now look at your git stage and you will see all the files `renv` created.

We have

1. `.Rprofile` that contains an `renv activate.R` script - this will activate the repo's `renv` every time the project is opened
2. The `renv.lock` file shows information on each package used in the repo and is used to update collaborator's environments to match the lock file.
3. `renv/.gitignore` I don't feel like explaining this one right now - i'll write more later
4. `renv/activate.R` this will activate the env whenever the R project is opened
5. `renv/settings.dcf` I have no clue what this is

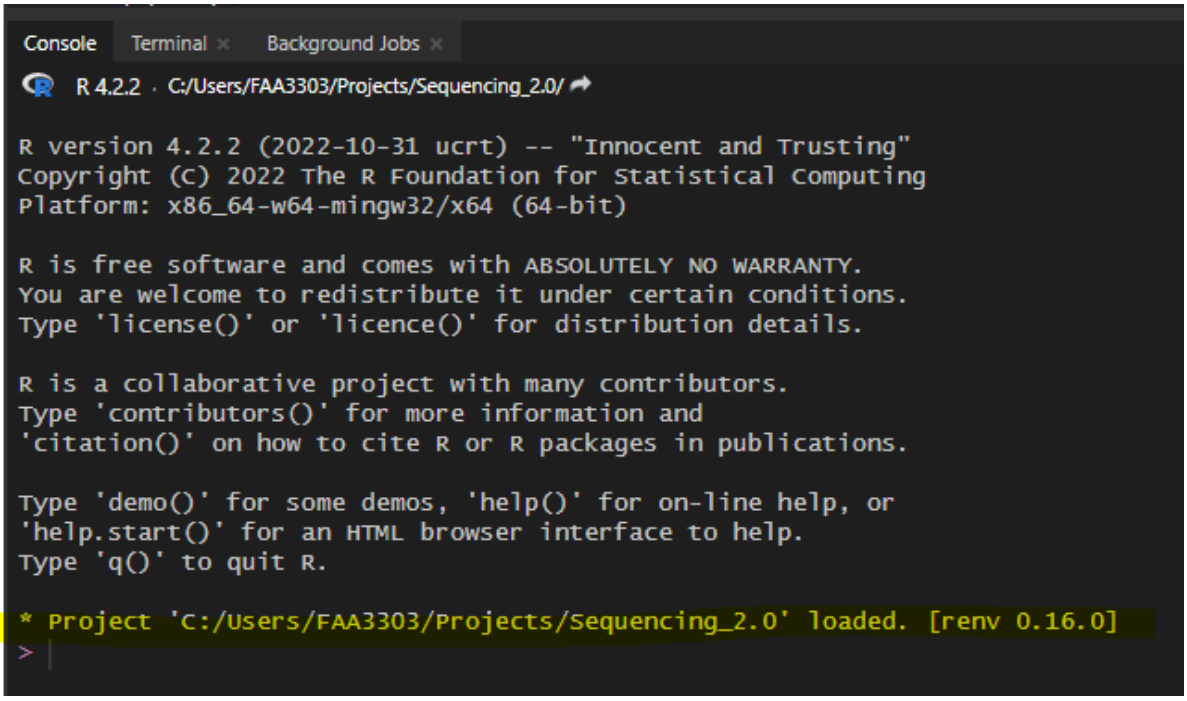



## 20.2 Using `renv` in a project

### 20.2.a Step 1: Open the `.Rproj` for your repo

Any time you need to code or run code from the repo, open up the `.Rproj` file that contains the sequencing 2.0 project. In your file explorer, go to the repo and open Sequencing_2.0 `.Rproj`

This will open up an R window with the repo file path as a root directory. It will also utilize the `renv`. Your console should say something about `renv`, like this




### 20.2.b Step 2: Load `renv` packages

The first time you use `renv` you will need to configure it to your local machine. To do this, type:

- `renv::restore()` in your console.
- This will create a new environment for your R in your local machine using the lock file packages.

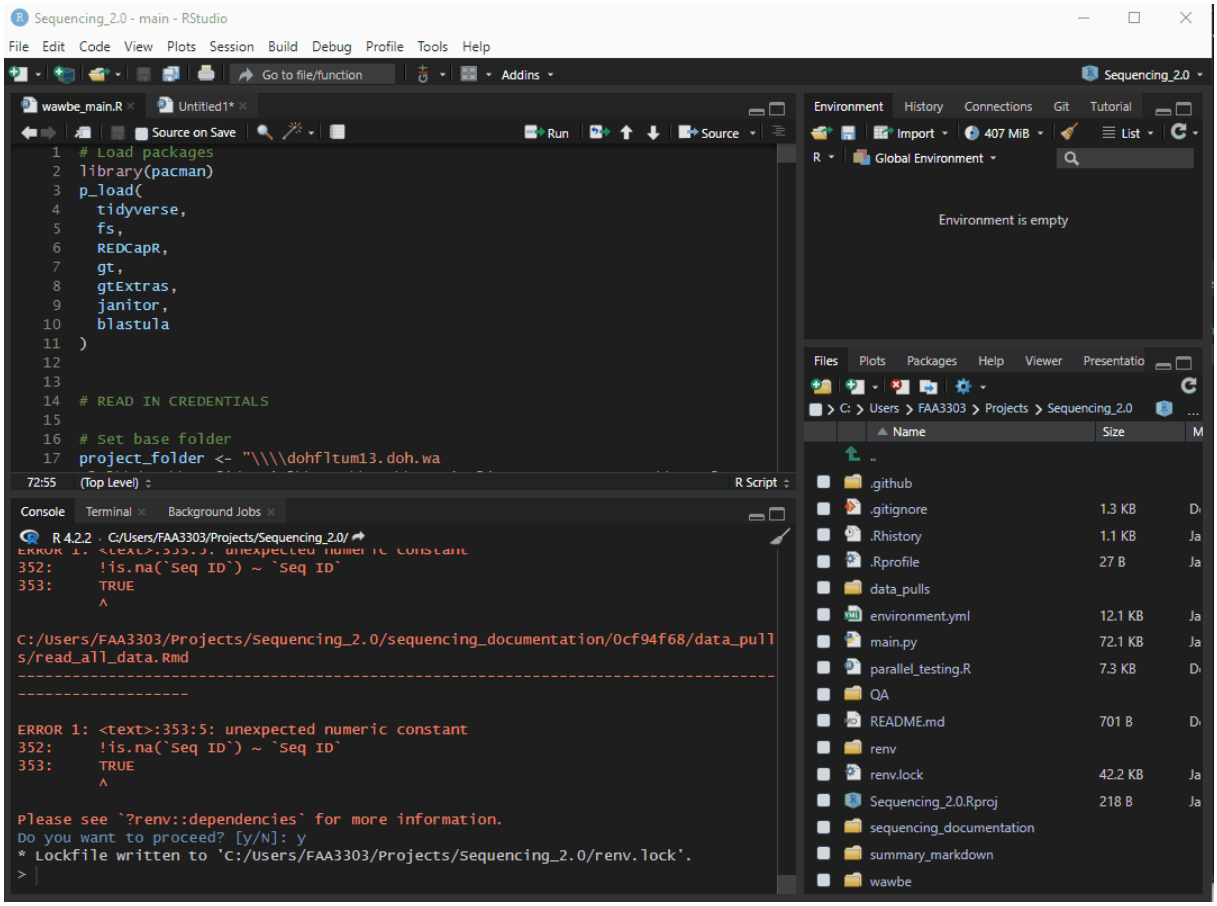
Now you're ready to use the scripts! Way less complicated than conda

# 21 R Installing New Packages

If you need to install a new package and want to put it in the repo, you will need to update the lock file. To do this:

- `renv::install()`, `pak::pkg_install("<PACKAGE_NAME>")` (the safe way) or even `install.packages()`
- `renv::snapshot()` this will overwrite the lock file with the packages you added
- Then push to github

> ⚠ Warning
>
> There may be version dependency issues when installing a package and running a script. You may need to use `renv::history()` to see the previous hash of the lock file and use `renv::revert()` to revert the lock file back to its previous, stable state. More on this here https://solutions.posit.co/envs-pkgs/environments/upgrades/




The video at the bottom of this page explains in detail `renv` and its capabilities https://solutions.posit.co/envs-pkgs/environments/upgrades/

# 22 Using Reticulate with Conda Env

Now that your R and Python environments are set up, if you have code that uses the `reticulate` package in R, it might still be pointing to your base Python environment. So, if you need to write python code in R (using reticulate), the code may break. Here's what you need to do to make sure your `reticulate` python path is pointing towards your conda environment:

1. Open an anaconda prompt

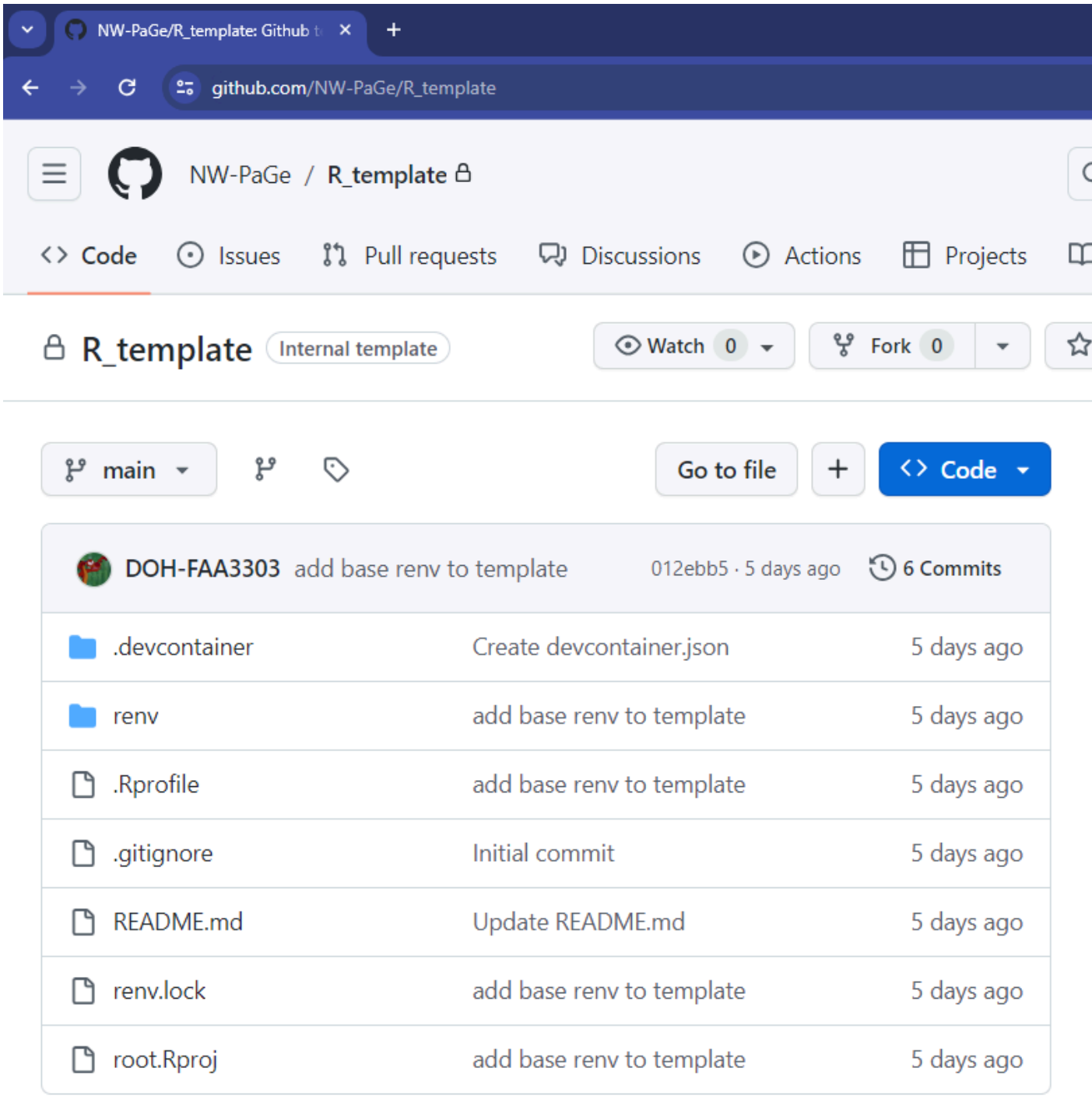
2. Activate your env and then write `where python` and it will provide you with a python.exe for that particular env. Copy that path
3. Open the `.Rprofile` file in your repo and add this code: `Sys.setenv(RETICULATE_PYTHON = PATH_TO_ENV_PYTHON))`
    - Note, if you have other team members, make this code flexible to their usernames. It could look something like this `Sys.setenv(RETICULATE_PYTHON = file.path(Sys.getenv("USERPROFILE"),"Anaconda3/envs/seq_env/python.exe"))`
    where `Sys.getenv("USERPROFILE")` will add `C/Users/XXXX/` and it will automatically add the user's name
4. Now restart your R
5. Open back up R and **without running any other code or loading any libraries** run `reticulate::py_config()` in your console. This should now show your conda environment path being used for reticulate

## 22.1 Templates

## 22.2 R Github Template

This is a Github repo template for R projects. It has `renv` set up and a `.devcontainer` for reproducibility

Select `Use Template > NW-PaGe/R_template` when creating a new repo in the NW-PaGe org. The template comes with a pre-filled `README.md` file that you can edit to your own needs. It's recommended to leave the `.devconainter` folder in this repo. It serves as instructions to install all the software and R packages needed to reproduce the repos code using a Github Codespace. See the `README.md` for more information.




## 22.3 Purpledoc Quarto Template

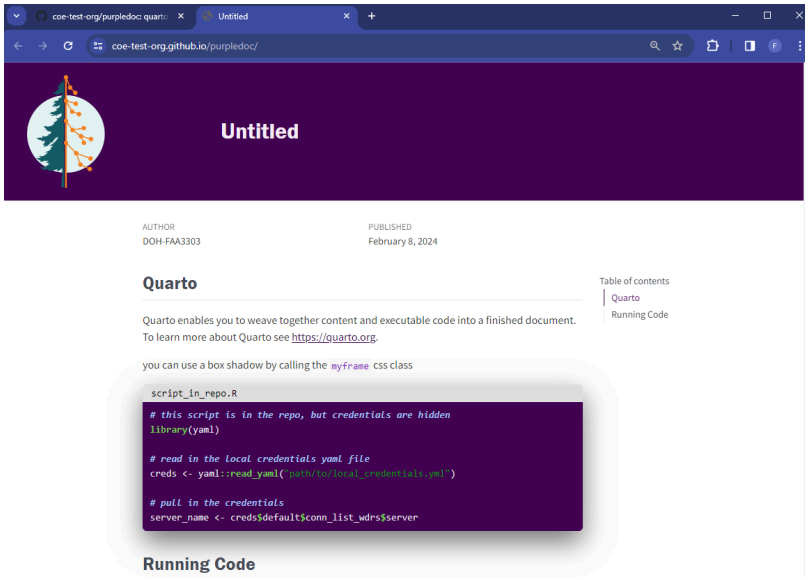
This is a Quarto template that contains the style and images used in this site

```
quarto use template coe-test-org/purpledoc
```

This will install the extension and create an example qmd file that you can use as a starting place for your report.

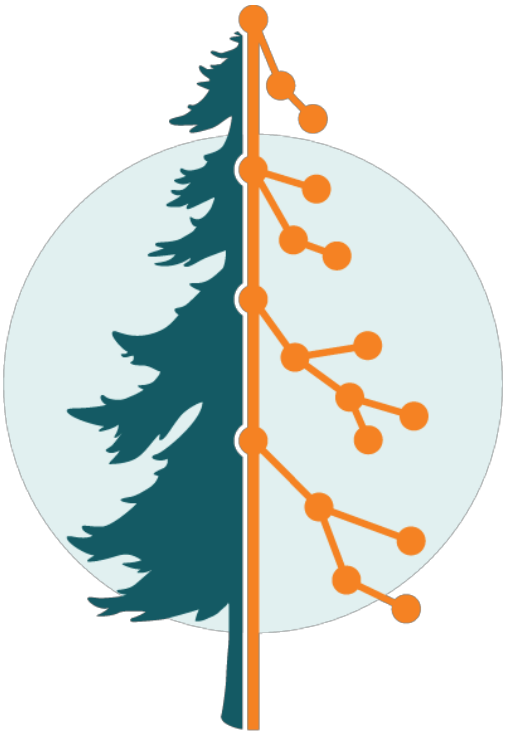
You may also use this format with an existing Quarto project or document. From the quarto project or document directory, run the following command to install this format:

```
quarto add coe-test-org/purpledoc
```




## 22.4 Documentation

# 23 Introduction



This site was created using Quarto, Github, and uses a Github Action to automatically render when a commit is pushed to the main branch of this repository.

Quarto is a framework for creating documentation, slideshows, articles, blogs, books and websites using markdown. It can execute R, Python and other programming languages within the document.

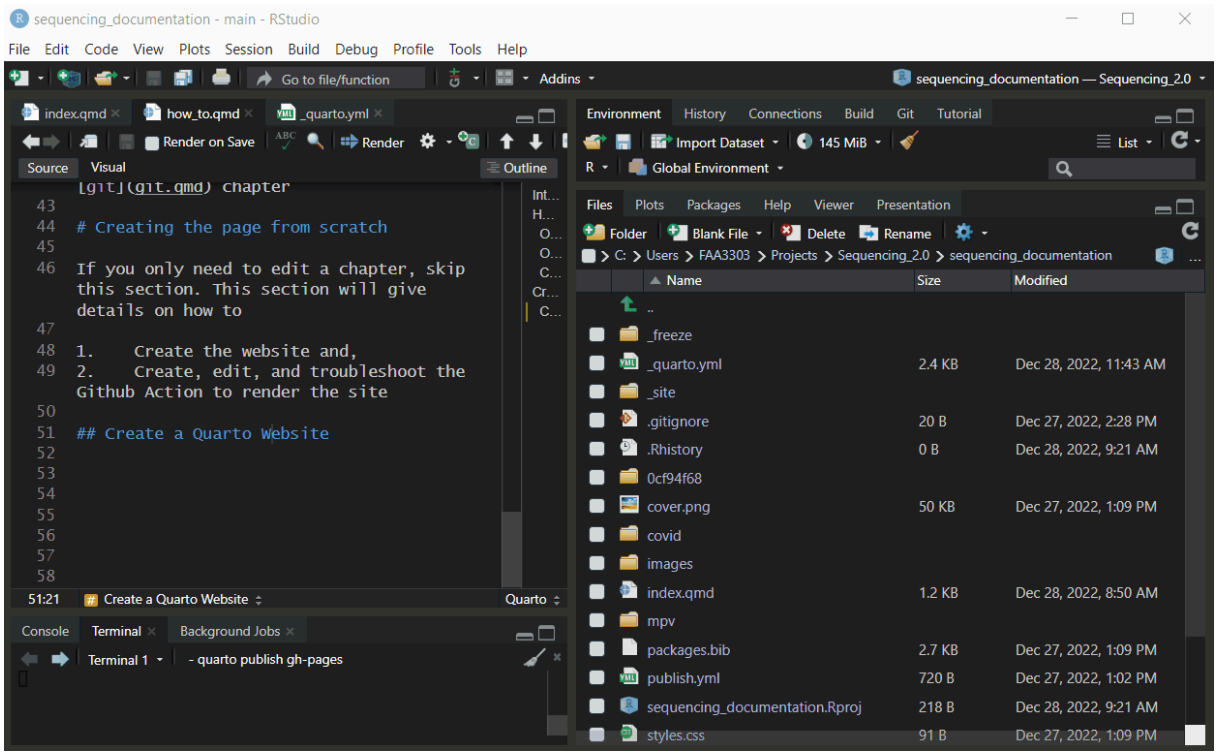
Github Actions uses a `.yml` file in the repository to trigger an action based on a certain event. In this case, when a commit is pushed to the main branch the `.yml` will trigger this Quarto website to render to the `gh-pages` branch of the repository and publish the github page. This section will give details on how to

1. Create the website
2. Create, edit, and troubleshoot the Github Action to render the site

# 24 Create a Quarto Website

Go to: `File > New Project… > New Directory > Quarto Website > Create Project`

There is an option to use `renv` which is a virtual environment for the R project. This is helpful, but if you're unfamiliar with `renv` you can un-check it.




The Quarto project will come with a `_quarto.yml` file. This is similar to an rmarkdown `yml` header where you can specify how you want your document to be style and what format to output it to.

# 25 How to edit the website (add chapters and change the style)

Most websites have a main file that sources all the htmls, css, javascript and other files into one. Quarto is the same - it sources all the markdown files and css files into one `.yml` file that dictates the output appearance and functionality of the site as a whole. Think of `.yml` files or headers as instructions for a document's style, output, and functions. It is the same thing as the `yaml` header in rmarkdown files.

## 25.1 Edit/Add Sections and Chapters

To add a section, open up the `_quarto.yml` file and scroll to the `navbar` section

```
project:
  type: website
website:
  title: "COE Github Standards"
  search: true

  navbar:
    background: primary
    left:
      - text: Home
        href: index.qmd
      - text: Github Organization Standards
        menu:
          - href: std/security.qmd
            text: "0: Security Standards"
          - href: std/lic.qmd
            text: "1: Choosing a License"
          - href: std/templates.qmd
            text: "2: Org Policy Setting"
```

This is where all of the `qmd` files are sourced and the instructions on how to format and style the navigation bar in the website.

Currently, the project is set up to have each section have it's own drop down menu in the navbar. In a section, use `- href:` to specify a file and `text:` to give the file a custom name in the website.

Each chapter exists within a sub-folder, so to add a chapter make sure create the qmd in its sub-folder and then reference the sub-folder and chapter in the `.yml` . For example, if you make a new chapter called `new-chapter.qmd` and it exists in the `covid` section/sub-folder, you need to reference it in the `.yml` file like: `covid/new-chapter.qmd`

## 25.2 Website Style

You can customize many aspects of the website in the `.yml` file itself with the `format:` function. There are a ton of themes included in Quarto here and you can also add a custom `css` and/or `scss` file to your project. I *think* you can even go super in depth and customize the javascript components of the site, but I'm not entirely sure how to do that yet. This website has a ton of custom `css` components with Quarto, and possibly uses custom javascript, so it could be a place to start if you're interested. Basically, you need to embed the `css` file into your `_quarto.yml` file

```
format:
  html:
    theme:
      - cosmo
      - assets/styles.scss
    scss: assets/styles.scss
    # css: styles.css
    toc: true
    highlight-style: assets/custom.theme
```

# 26 How to edit a chapter

First you must clone our github repository. For more information on cloning the repo, follow these instructions

Once you have a local clone you can start editing these files and push updates.

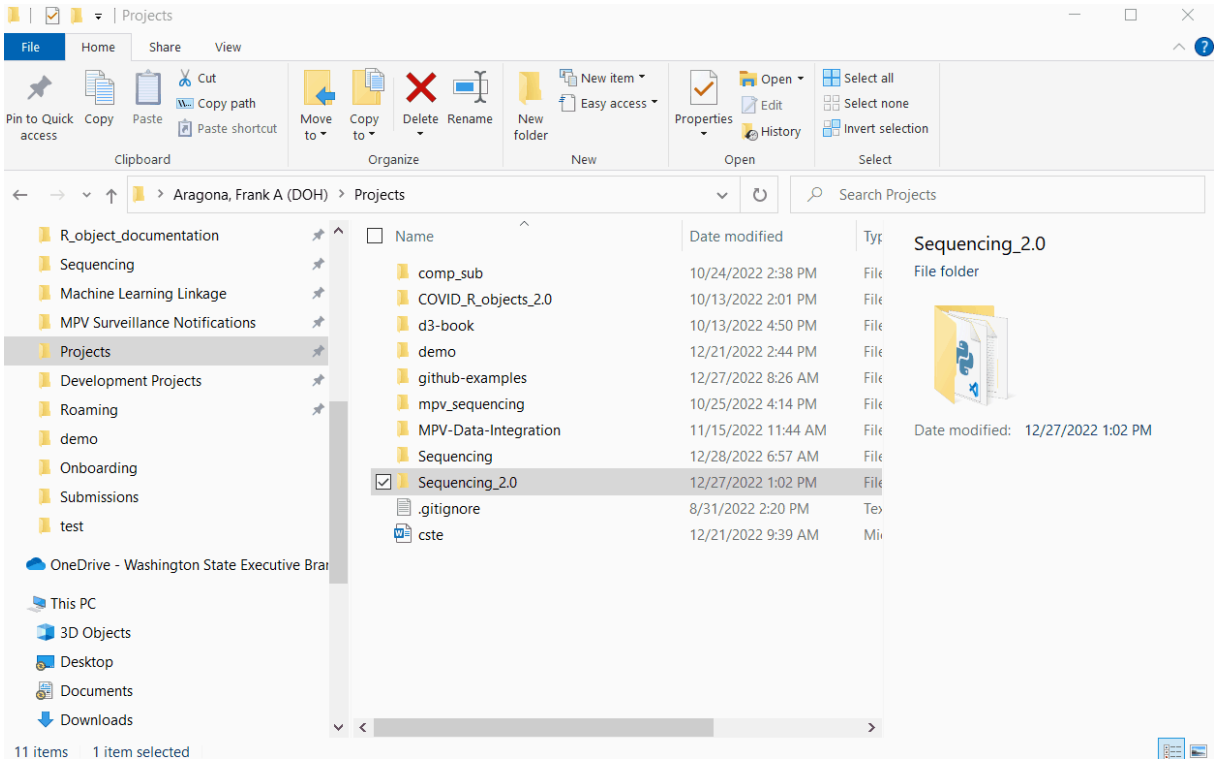
Basic steps:

1. Open the R Project
2. Open the folder/section you want to edit
3. Open the specific chapter in that section you want to edit (each chapter is a `.qmd` file)
4. Save the change and push to the main branch (or make a pull request to the main branch)

Once a commit is push to the main branch, it will trigger a Github Action to re-render the website and publish it to the main github repository

## 26.1 Open the R project

This is a Quarto website that is contained in a `.rproj` file path. The R project contains all the documents used to create this website. Begin by opening the R project when should be in your local clone under `C:\Users\XXXXXXX\Projects\Sequencing_`




## 26.2 Open the files

This project has `.qmd` files (Quarto Markdown files) that each represent a chapter in the website. All of the `.qmd` files are knitted together (using R `knitr` ) which compiles all of the files to be sourced into htmls.

This website is set up to have each major section contain multiple chapters. To open a chapter, the bottom right pane in your R Studio window should contain folders for each section, highlighted below

```
$ tree /f
C:.
|   .gitignore
|   about.qmd
|   index.qmd
|   standards.Rproj
```

```
|   _quarto.yml
|
├──assets
|       custom.theme
|       styles.css
|       styles.scss
|
├──std
|   |   creds.qmd
|   |   lic.qmd
|   |   public_code.qmd
|   |   security.qmd
|   |   templates.qmd
|   |
|   └──images
|
├──tools
|   |   how_to.qmd
|   |   iac.qmd
|   |   link_code.qmd
|   |   readme.qmd
|   |   release.qmd
|   |   renv.qmd
|   |   teams.qmd
|   |
|   └──images
```

The `.qmd` files are inside of these folders. Select one to edit.

## 26.3 Commit changes

Once you're done editing, push the change to the main branch (or make a new branch, and then a pull request for the main branch). More one this in the git chapter

# 27 How to link external code to the site

The code in this website is automatically linked to the code in the repository. This lets us update the code and not need to copy and paste any new changes to the documentation/github page

To do this, open your *external* R script in your repo. In this case we're using `external_script.R` . Now wrap the chunks of code you want to link with comments like this, `## ---- libraries` and `## ---- stop` :

```
## ---- libraries
library(pacman)
p_load(
  reticulate,
  fs,
  lubridate,
  dplyr,
  stringr,
  magrittr,
  readr,
  httr
)
## ---- stop
```

The `## ---- libraries` signals the beginning of a chunk. the `## ---- stop` signals the end of a chunk.

Now you can call this chunk in your *github page/quarto document* like this:

Scan the external R script for code chunks:

````
```{r setup}
#| echo: false
knitr::read_chunk(file.path(getwd(),"external_script.R"))
```
````

Call the code chunk you want in the `{r}` header within the chunk. like this `{r libraries}` :

````
```{r libraries}

```
````

And now the document will output any code in that code chunk and can also execute that code chunk if you want. Here's what the output will look like in this case:

▼ Code

```
library(pacman)
p_load(
  reticulate,
  fs,
  lubridate,
  dplyr,
  stringr,
  magrittr,
  readr,
  httr
)
```

# 28 Publish the site/Github Actions

Github Actions allow you to automate tasks in your repository. Quarto has functions for Github Actions that allow you to automatically render your Quarto document and publish it to Github Pages. Github Pages is a free service from Github to host a website. Documentation for R and Python code is usually found in a Github Page within the package repository

To create this process, I followed the Quarto dev documentation:

> Add the GitHub Actions workflow to your project

1. Copy quarto-publish-example.yml to `.github/workflows/quarto-publish.yml`. Uncomment the "Publish to GitHub Pages (and render)" action. No further changes are needed to the action (in particular, do not edit the line below to add a secret to this file. This file has the same permissions as your repository, and might be publicly readable)

2. run `quarto publish gh-pages` locally, once

3. Quarto needs to configure the repository for publishing through GitHub Actions. To do this, run `quarto publish gh-pages` locally once.

4. Now, add and commit the workflow file you have just created, and push the result to GitHub. This should trigger a new action from

GitHub that will automatically render and publish your website through GitHub pages.

Note that GitHub Pages uses a gh-pages branch in your repository, which will be automatically created if one doesn't exist.

## 28.1 Example `YAML` Workflow

The `.yml` workflow for this project looks something like this:

`on:` is a tag indicating when the action will run. Right now it will run when any code gets pushed to the main branch in the documentation folder or `lineages_public_repo.R` script

```yaml
on:
  push:
    branches:
      - main
    paths:
      - documentation/**
      - lineages_public_repo.R
```

`jobs:` is a tag that tells a Github virtual machine what to run and what operating system to run it on. In this case `ubuntu` with the `latest` version. This can be `windows` , `linux` or `macOS` .

```yaml
name: Render and Publish

jobs:
  build-deploy:
    runs-on: ubuntu-latest
```

Now we have the steps:

- `env` will find the renv folder
- `uses: actions/checkout@v3` will refresh the repo and pull the latest changes
- `uses: quarto-dev/quarto-actions/setup@v2` will install quarto
- `uses: actions/cache@v1` and the code below it will set up `renv` and use the cached packages to install them onto the Github virtual machine

```
    env:
      RENV_PATHS_ROOT: ~/.cache/R/renv

    steps:
      - name: Check out repository
        uses: actions/checkout@v3

      - name: Set up Quarto
        uses: quarto-dev/quarto-actions/setup@v2

      - name: Prep CURL install
        run: sudo apt-get update

      - name: Install CURL Headers
        run: sudo apt-get install libcurl4-openssl-dev

      # - name: Setup Renv
      #   uses: r-lib/actions/setup-renv@v2

      - name: Cache packages
        uses: actions/cache@v1
        with:
          path: ${{ env.RENV_PATHS_ROOT }}
       key: ${{ runner.os }}-renv-${{ hashFiles('**/renv.lock') }}
          restore-keys: |
            ${{ runner.os }}-renv-

      - name: Restore packages
        shell: Rscript {0}
        run: |
                if (!requireNamespace("renv", quietly = TRUE))
install.packages("renv")
          renv::restore()
```

And finally,

- `uses: quarto-dev/quarto-actions/publish@v2` will render the site by running `quarto render`
- `with: target: gh-pages path: documentation/_site` lets you know which branch and path to render the site to

```
      - name: Publish to GitHub Pages (and render)
        uses: quarto-dev/quarto-actions/publish@v2
```

```
        with:
          target: gh-pages
          path: documentation/_site
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }} # this secret
is always available for github actions
```

## 28.2 Using `renv` in the GH Action

If you need to constantly update your website with code chunk, this is the best way to do it. It is also probably safer and better than the `_freeze` way, but it requires a better understanding of Github Actions and virtual environments.

`renv` is an R package for creating a project level virtual environment. In other words, `renv` will create project specific folders that contain the specific R package versions you use in an project. More on virtual environments here

To use `renv` in a Github Action, you can put

```
- name: Setup Renv
  uses: r-lib/actions/setup-renv@v2
```

or use the renv cache code in the `yaml` section above

## 28.3 Using a `_freeze` file

If you only need to execute the code once or just need to render a non-executable code chunk once, make sure you have this code in your `_quarto.yml` file:

```
execute:
  error: true
  freeze: true
```

and then run this in your terminal window:

`quarto render name-of-specific-document-or-chapter.qmd`

This will render that specific document in the website, execute code chunks if they are set to execute ( `eval: true` ) and then it will create a `_freeze` file. The `_freeze` file will save a snapshot of that specific document and *not* re-render it in the Github Action. This means you can render other parts of the website, but any files in the `_freeze` folder will stay the same as they are in the freeze. If you need to make changes to a freeze document, run the quarto render code again after making changes.

This is also documented in the Quarto dev documentation

## 28.4 Troubleshooting

So you did these steps:

1. Create the `quarto-publish.yml`
2. Run `quarto publish gh-pages` in the **terminal**
3. Push all the files in your git to the main branch

If this works on your first try then the universe is taking extra special care of you.

If not, you are like the rest of us poor souls:/

The first thing I would check is the error in your Github repo's Action tab.

**If the error is something like `jsonlite not installed` or `some package not installed`** then it most likely means your are trying to commit a chunk of code in the documentation. *Even if you are not executing the code*, Github Actions will punish you. There are a couple options to fix this, depending on your priorities.

1. If you don't care about executing your code and/or only need to push that part of the script once, consider using the `_freeze` option
2. If you need to execute code *or* need to programmtically render the document with code chunks often, consider using `renv` or a similar package installation method

## 28.5 New Repo

## 28.6 Introduction

We are often writing scripts that could be useful to others. This tutorial goes over how to make your R scripts accessible to others by making them available in the NW-PAGE GitHub repo.

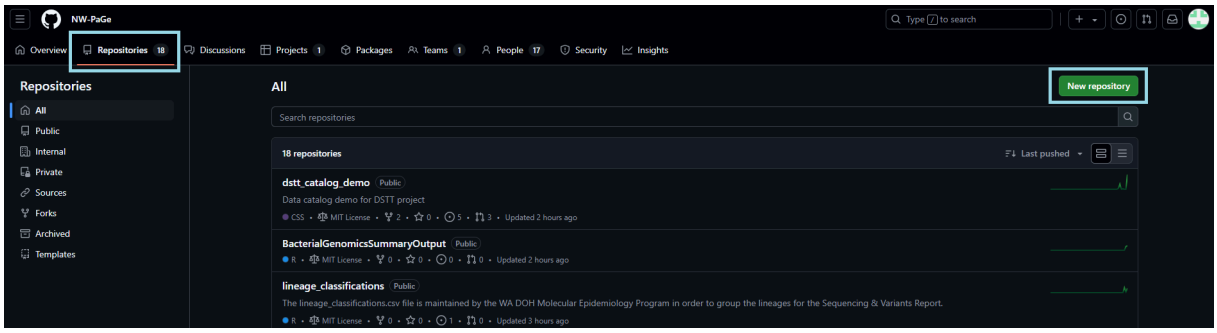
## 28.7 Steps

### 28.7.a Sign-in

Sign-in to GitHub using your GitHub credentials. If you are part of WA DOH make sure to use your WA DOH compliant Git Hub account.

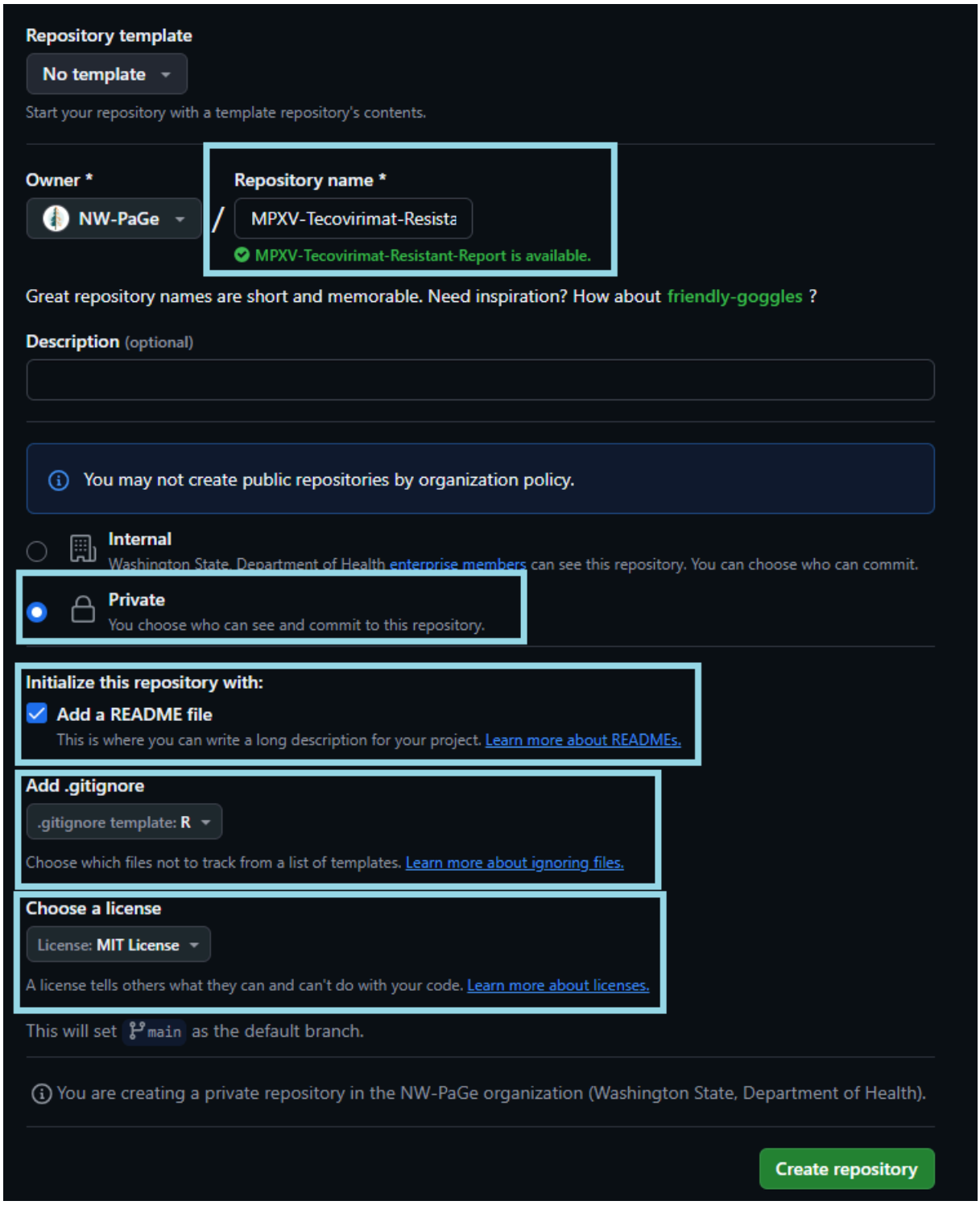
### 28.7.b Create a new repository

In the browser, navigate to `Repositories` and click `New repository`



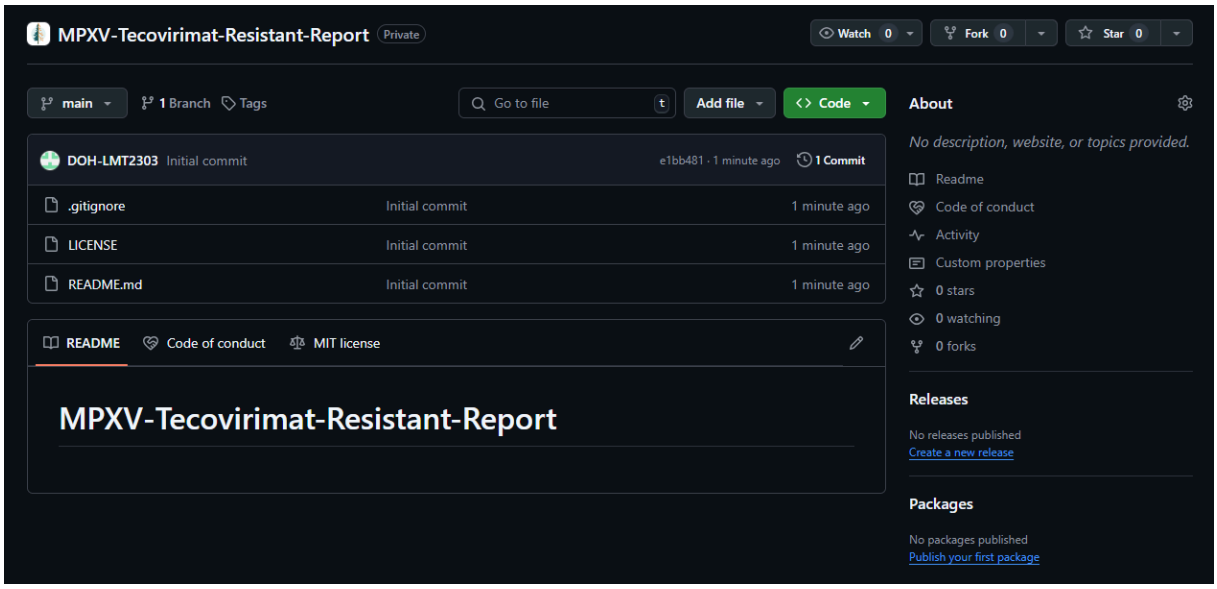

## 28.7.c Fill out the new repository fields

1. Name your repo
2. Select your repo to be private, you can change this later
3. Select the option to add a README file
4. Select a .gitignore template (R or Python are good options)
5. Select the MIT license
6. Click `Create repository`



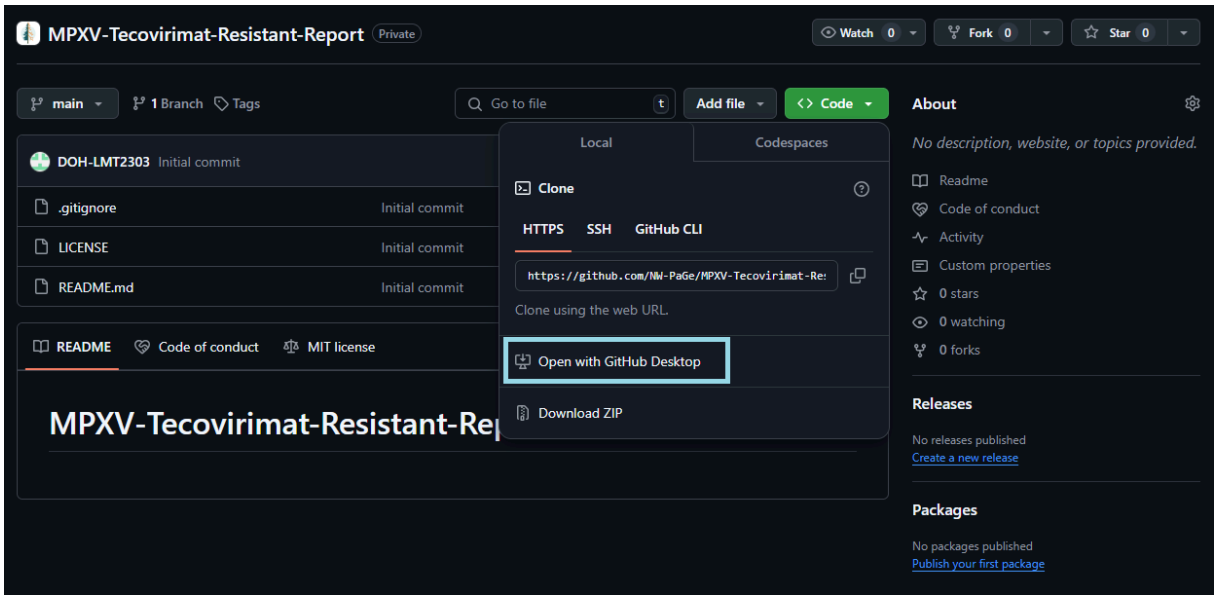

## 28.7.d Check out your new repo

Congratulations you have created a new GitHub repo. Now you have to populate it with information!

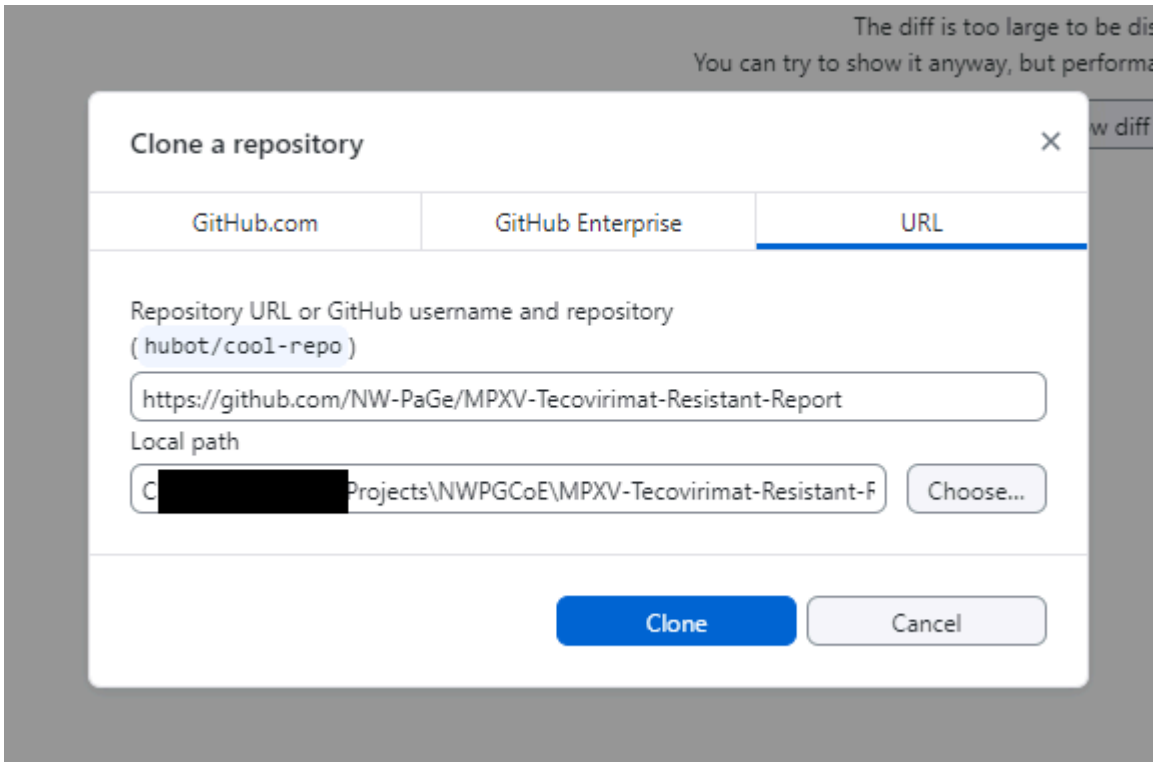



## 28.7.e Clone the new repo

Now you would want to clone your repo in your machine to start uploading content via commits. To do this click `Code` and either clone your repo using terminal by copying the URL, or click `Open with GitHub Desktop`. I will demonstrate using the latter option.




GitHub Desktop will open. Double check the location where you want to close your repo. Click `Clone` and double check that a folder with the repo name has been created in the local path you provided.




### 28.7.f Safeguard sensitive data

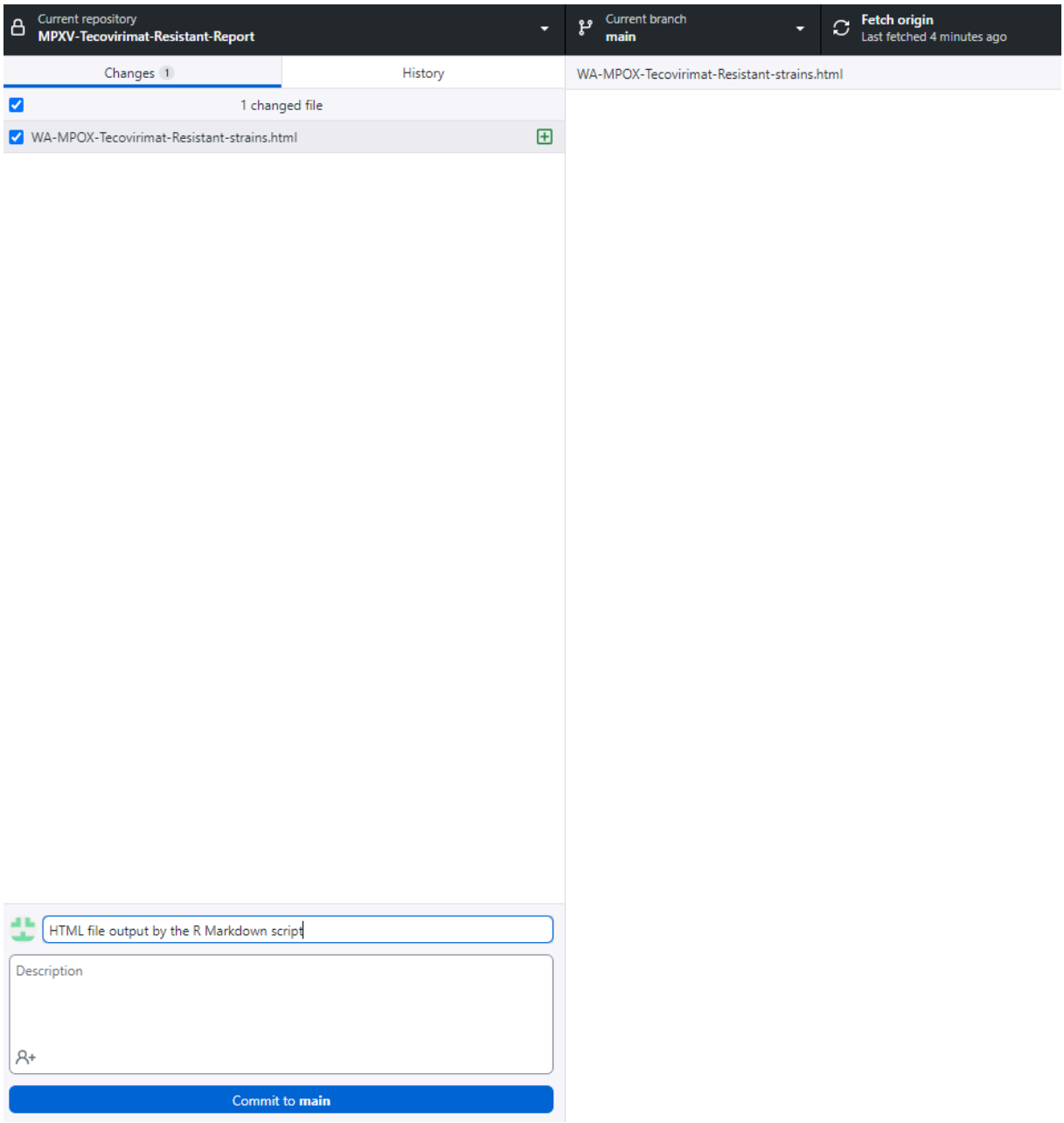
Before making your first commit you would want to make sure to do your due diligence to safe guard important information.

1. Check that you are using git secrets which will block commits that contain file paths and server names from being uploaded to your repo
2. Add file names that you don't want to accidentally upload to the repo to the gitignore file

### 28.7.g Populating the repo via commits

If your information is saved elsewhere and you need to upload it to the repo simply copy and paste the files inside the folder where you cloned the repo. In this example, I pasted an HTML file inside the folder and then I navigated to GitHub Desktop to make the commit which will upload that file to the repo.

Remember to select the file, add a title to the commit, click `Commit to main`, and then click `Push origin`.


Current repository
MPXV-Tecovirimat-Resistant-Report
Current branch
main
Fetch origin
Last fetched 4 minutes ago
Changes 1
History
WA-MPOX-Tecovirimat-Resistant-strains.html
1 changed file
WA-MPOX-Tecovirimat-Resistant-strains.html
HTML file output by the R Markdown script
Description
Commit to main