Table of Contents

[What is GitHub? 3](#_Toc34135548)

[GitHub Introduction 3](#_Toc34135549)

[Version Control Introduction 3](#_Toc34135550)

[Access to GitHub 3](#_Toc34135551)

[**What is a Repository**? 4](#_Toc34135552)

[Best repository practices: 4](#_Toc34135553)

[**Creating a New Repository** 4](#_Toc34135554)

[What does the term local/machine mean? 6](#_Toc34135555)

[**What is Git?** 6](#_Toc34135556)

[Some advantages of using Git: 6](#_Toc34135557)

[Video: How to integrate Git and RStudio 6](#_Toc34135558)

[Best practices to use both: Git and GitHub 6](#_Toc34135559)

[**Repository Cloning** 8](#_Toc34135560)

[**What are the roles of the collaborators in the project repository?** 9](#_Toc34135561)

[**Admin** 9](#_Toc34135562)

[**Developer** 9](#_Toc34135563)

[**What is a branch in a repository?** 10](#_Toc34135564)

[**Branching Pros and Cons:** 10](#_Toc34135565)

[Pros: 10](#_Toc34135566)

[Cons: 11](#_Toc34135567)

[**What is the difference between master branch, and developer development branches?** 11](#_Toc34135568)

[The master branch: 11](#_Toc34135569)

[The developer development branch: 12](#_Toc34135570)

[ADMIN- Creating a new developer branch 12](#_Toc34135571)

[**ADMIN- Managing `pull request` and `merges`** 13](#_Toc34135572)

[What is a Merge Conflict? 13](#_Toc34135573)

[Merge Conflict Example 13](#_Toc34135574)

[**Sync Developer and Master Branch** 16](#_Toc34135575)

[**ADMIN- Permissions Set Up** 19](#_Toc34135576)

[Minimum Branch Protection Settings 19](#_Toc34135577)

[All Branch Permission Settings 19](#_Toc34135578)

[**Developer- Notifications about submitted `pull requests`** 20](#_Toc34135579)

[**Reasons why pull requests are not approved** 21](#_Toc34135580)

[Naming Conventions for files 21](#_Toc34135581)

[Naming Conventions for folders 21](#_Toc34135582)

[**Developer- Git Bash commands for specific branches** 21](#_Toc34135583)

[**Developer- R Studio and branches** 22](#_Toc34135584)

[**Admin- Creating an R Project** 23](#_Toc34135585)

[**Lone Cabbage Reef Workflow Theory** 30](#_Toc34135586)

[Branching figure- Staging area workflow theory using *git add* and *git commit* commands. 30](#_Toc34135587)

[Local Machine Workflow Theory 31](#_Toc34135588)

[GitHub Online Portal Workflow Theory 33](#_Toc34135589)

[**R Studio (Git tab) Workflow Theory** 39](#_Toc34135590)

[Changing Windows Line endings 41](#_Toc34135591)

[References & Resources 47](#_Toc34135592)

# What is GitHub?

GitHub is an online portal that allows for developers to store code, documents, images, and videos that enables a version control for any change to the repository. Version control means that for every change or “version”, there is a record of who submitted the change, when, and what exactly they changed.

Version control systems keep these revisions straight, storing the modifications in a central repository. This allows developers to easily collaborate, as they can download a new version of the software, make changes, and upload the newest revision. Every developer can see these new changes, download them, and contribute.

GitHub Introduction

Beginners guide to GitHub:

<https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>

Version Control Introduction

Beginners guide to version control:

<https://www.atlassian.com/git/tutorials>

Access to GitHub

There is no need to download GitHub, it is an online website (https://github.com/), and can be accessed by any computer that connects to the internet.

Sign up for your own free GitHub account here, (no need to sign up for the upgraded GitHub account type):

<https://github.com/join>

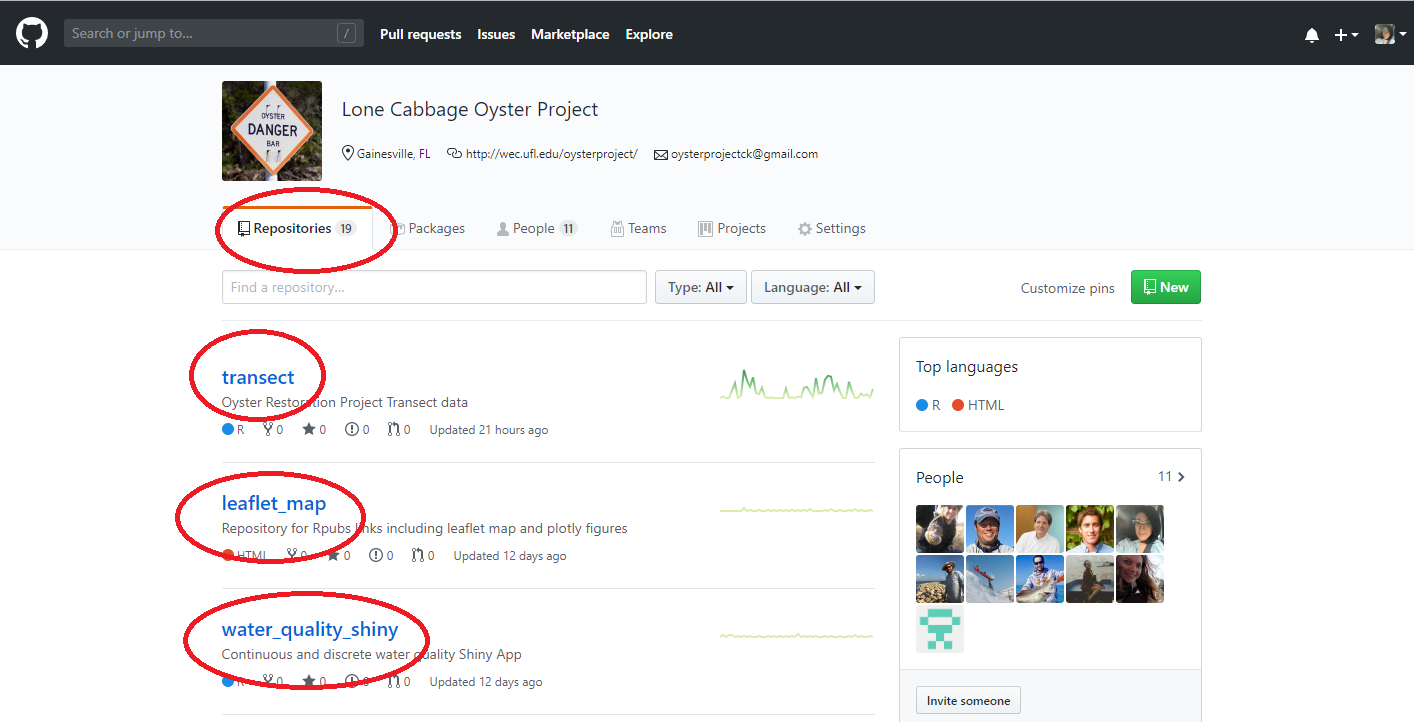
However, you can download a GitHub Desktop App if it is more convenient for you. The App will allow you to make file changes and to push and pull into branches and forked repositories (<https://desktop.github.com/>).

**What is a Repository**?

A repository (usually abbreviated to “repo”) is a location where all the files for a particular project are stored. Each project has its own repo, and you can access it with a unique URL.

Best repository practices:

* create a new repository for each project
* avoid confusing names for folders, be specific



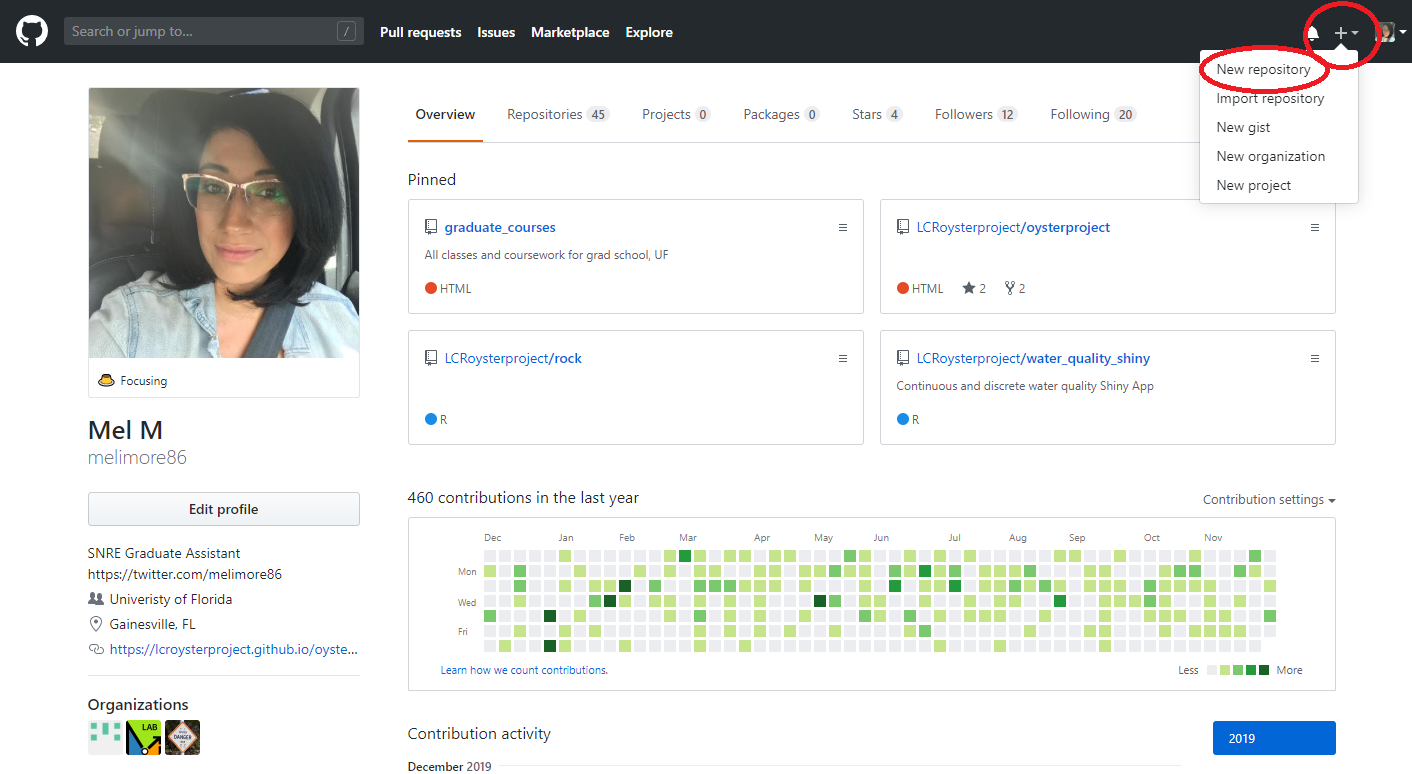
Repository Screenshot 1- Image showing project repositories and the Repository tab.

**Creating a New Repository**

To create a new repository click on the “+” in your account or on the project page.

**NOTE: For the oyster project there is a repository already in place:**

The link for the Oyster project Data page, <https://github.com/LCRoysterproject>



Profile Screenshot 1- Basic profile of a developer and admin, where they can create a repository at the top right“+”. The admin will always create the new repository for the project, but a developer can create their own repository for their own personal use.



Profile Screenshot 2- Profile of the project, where a developer or admin can create a repository at the top right under “New”. Only admins can create a new repository for

What does the term local/machine mean?

* “Local” is the term used to describe files, or folders on your local computer.
* Machine is another term for your personal/work laptop, computer.
* Files from GitHub can be directly downloaded into your computer either through the GitHub online portal, GitHub Desktop App, or through the Git Bash command line.

**What is Git?**

Git Bash is a command prompt which allows you to write out code to submit changes from your local machine to the GitHub online portal. All commands that can be executed in the command prompt, can also be completed in the GitHub portal and through RStudio.

Some advantages of using Git:

* Allows submission of changes through RStudio directly into your GitHub online portal for the repository.
* Allows for status checks on development branches to the `master` branch.

Video: How to integrate Git and RStudio

<https://www.youtube.com/watch?v=E2d91v1Twcc>

Additional resources for Git and RStudio

<https://happygitwithr.com/>

Best practices to use both: Git and GitHub

It is normal practice to use both Git Bash and GitHub portal to:

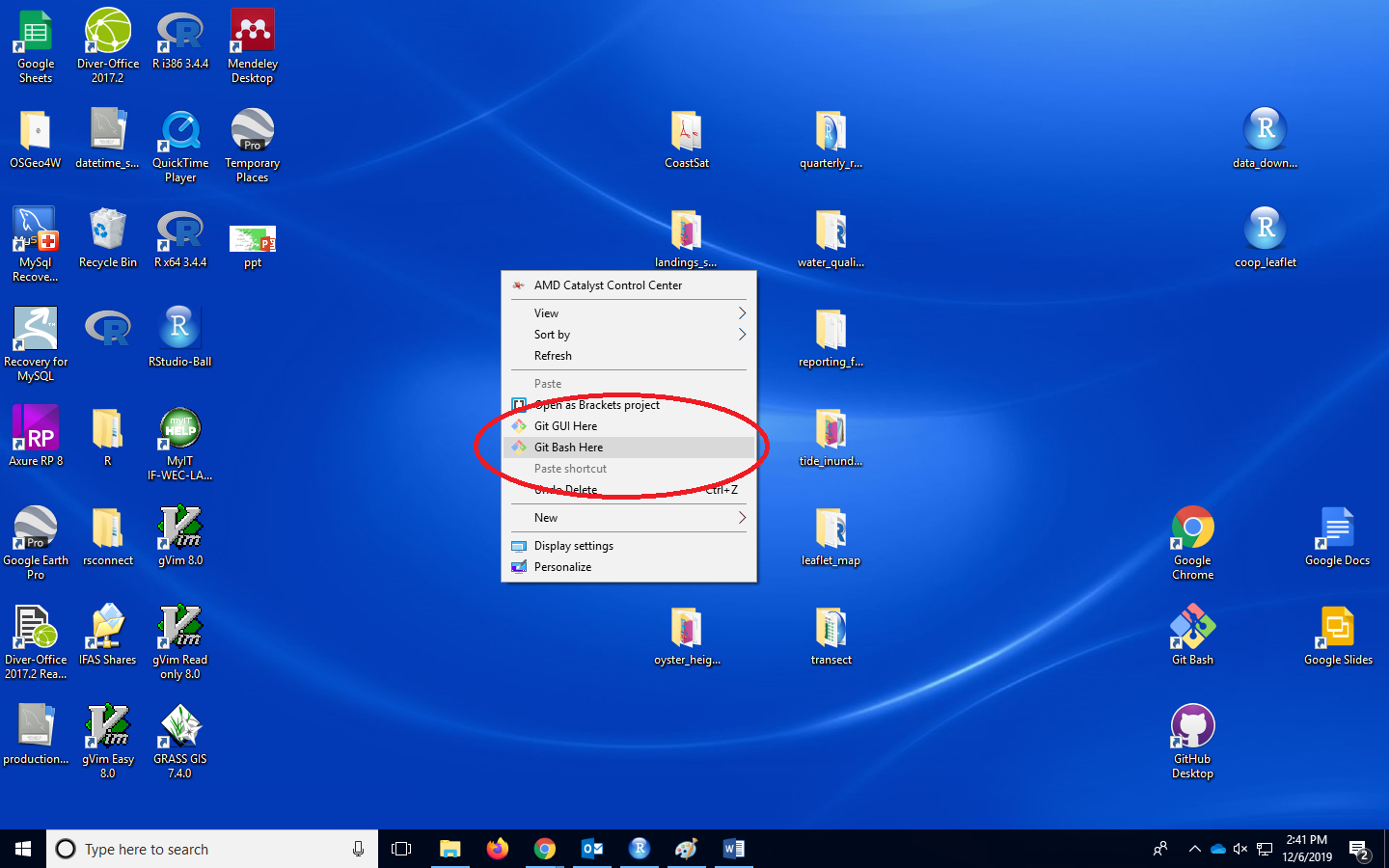
* clone repositories to your local machine
* track your status updates
* change files
* commit pushes
* perform pull requests
* configure RStudio with Git to access your GitHub online portal

It is up to each individual developer to decide on preferences both practices are outlined in this workflow documentation.

**Access to Git Bash**

You can download Git Bash here, <https://git-scm.com/downloads>.

Instructions below will be for Microsoft PCs. There are additional steps and considerations for Mac users, <https://github.com/fabriziocucci/git-bash-for-mac>.

Once you download Git Bash, you can right-click on your mouse, to open the Git Bash terminal. If you do not right-click on an already existing cloned folder, you will need to navigate to the folder, in terminal, through a series of commands. These instructions are for Microsoft computers.

Git Bash Figure 1- Right click on the mouse to access the Git Bash Here option.



Git Bash Figure 2- . This is the command prompt of the Git Bash terminal.

Git Bash command reference guide and additional resources

<https://sklise.com/2012/09/22/introduction-to-git/>

<https://www.atlassian.com/git/tutorials/git-bash> (try the interactive Git Bash tutorial)

https://edav.info/index.html

Setting up your credentials in Git

Additionally, it is important to add your global username and email to Git. You can follow the instructions located here, <https://git-scm.com/book/en/v2/Getting-Started-First-Time-Git-Setup>, but the code will be listed below. You can copy and paste these commands into your Git Bash.

##Update your username and email (use the same email and username when you setup your GitHub profile)

*git config --global user.name "John Doe"*

*git config --global user.email johndoe@example.com*

##Check your user settings

*git config --list --show-origin*

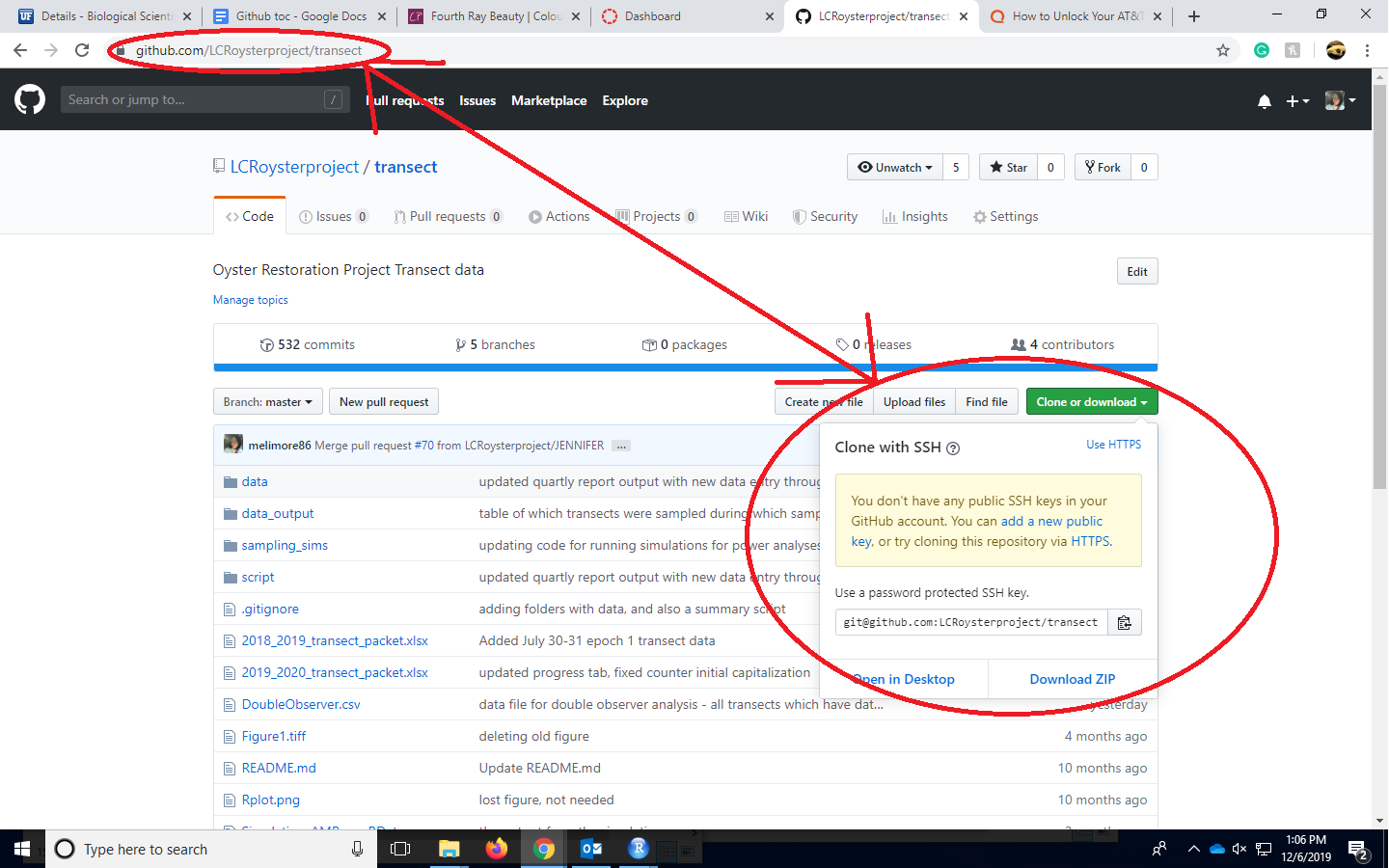
**Repository Cloning**

When a repository is created in the GitHub online portal, there will be a set of instructions to follow to clone the repository into your local machine. You can use Git Bash to connect to the project GitHub directly.

The command for cloning the repository is:

*git clone* [*https://GitHub.com/LCRoysterproject/*](https://github.com/LCRoysterproject/)*(repository name).git*

Each repository will have its own link used in the command line, located in the areas below:



Cloning Figure 1- Click on Clone or Download to locate the link used for cloning. You can also use the link in the URL bar.

**What are the roles of the collaborators in the project repository?**

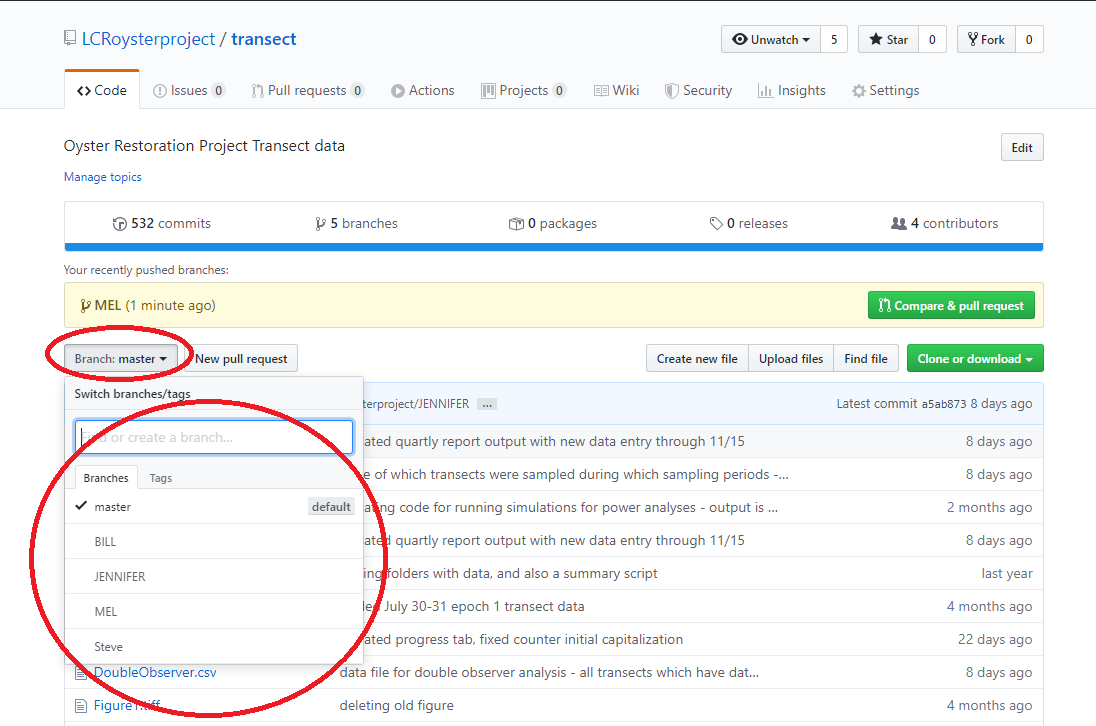
**Admin**

* Monitors the `master` branch, usually 1-2 people
* Creates development branch for each developer, and deletes branches when developer is no longer working on project
* Reviews all `pull requests` from each developer, chooses whether to approve, change, or deny the `pull request`
* Creates permissions for the repository
* Resolves merge conflicts

**Developer**

* Utilizes developer specific branch to modify, delete, or change code/ text
* Works independently of other branches including the `master` branch
* Submits `pull requests` to the `master` branch to merge changes
* Checks emails frequently once a `pull request` is submitted

**What is a branch in a repository?**

Branches are separate “clones” of the `master` branch, that can `pull` changes from the `master` or  submit `pull requests` to the `master` branches. Branches are located within one repository and can be found in the “Branch:” button.

Through a series of developer specific branches, each developer will submit their changes/ modifications to the `master` branch that will be approved by the admin (Bill Pine, Jennifer M, Mel M). Pull requests can be approved, commented on, or denied. All pull requests will have thorough documentation for each submission. Branches will not be deleted if the developer is still active. Each branch must submit a pull request from the `master` -> branch, prior to any pull requests. Each step will be explained in further detail.

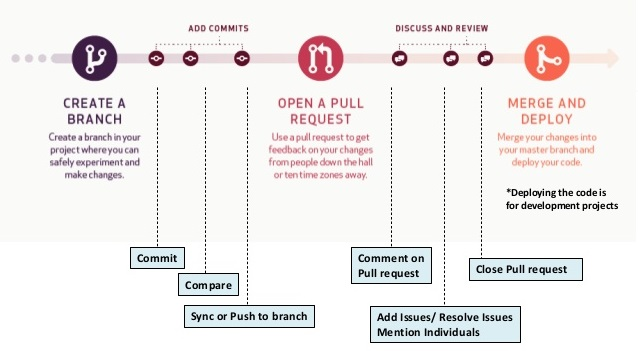
**Branching Pros and Cons:**

Pros:

* Keeps all of the work being done around a project in one place
  + No confusion with forks, or forking
* All collaborators can push to the same branch to collaborate on it
  + All collaborators will be able to view the master branch pull request after they are approved
* There's only one Git remote to deal with

Cons:

* Branches that get abandoned can pile up more easily
  + Not in our case, since each developer will have their own branch, and developers that are not on the project any more, will have their branches deleted
* Your team contribution process doesn't match the external contributor process
  + Both processes are indeed different
* You need to add team members as contributors before they can branch
  + This will be the responsibility of the Admin

****

Branch Figure 1- A basic workflow working with branches. Admin will create the branch for the developer, where the developer will commit their changes, and submit a pull request from the developer branch -> master branch. Pull requests will have comments from the admin, where the issues can be resolved (if any), and the pull request will be approved, and the pull request will be closed. ([http://www.smlcodes.com/tutorials/GitHub-tutorial/](http://www.smlcodes.com/tutorials/github-tutorial/))

**What is the difference between master branch, and developer development branches?**

The master branch:

* Considered the production/live branch.
* The production branch is where all reviewed code and documents submitted to for review, and all approved changes will be live.
* The master branch is protected, and should only be handled/reviewed by an Admin
* Developers will not work directly in this branch.



The developer development branch:

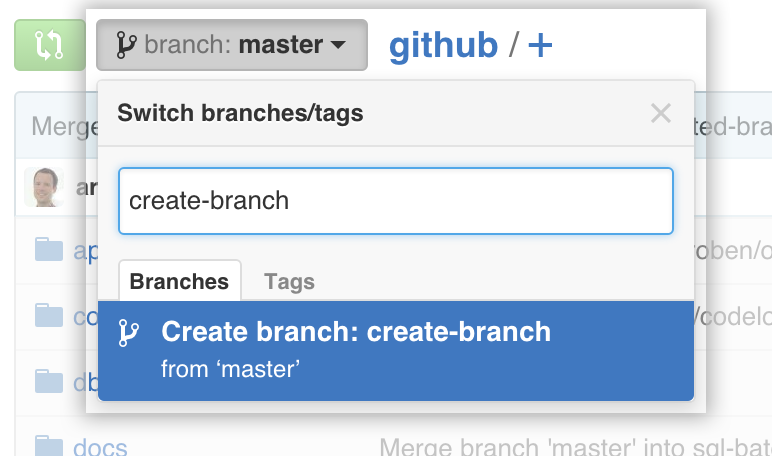
* For testing and development of code, or edits of documents.
* **Each developer will have their own development branch, in which they will work on solely on.**
* Developers can work on their own branch, with no Admin monitoring.
* Developers are free to make and changes/modifications/edits to their own branch.
* Admin will create a new branch for each collaborator.

ADMIN- Creating a new developer branch

Setting up the developer involves creating a new branch for each developer. The Admin is the only allowed personnel to create/delete branches.

Each developer will need a new branch, in which they will work on solely.

Changes that are made on a development branch doesn’t affect the `master` branch, so the developer is free to make and commit changes to their own branch. The `master` branch is safe in the knowledge that a development branch won't be merged until it's ready to be reviewed by the ADMIN. ([https://guides.GitHub.com/introduction/flow/](https://guides.github.com/introduction/flow/))



Creating branch Figure 1- The branches will be created from the `master` branch. At the time the developer branch is created, the new branch will be an exact clone of the `master` branch.

<USERNAME> - DEVELOPMENT

**ADMIN- Managing `pull request` and `merges`**

The master branch will receive the `pull request` from the developer. An Admin will be selected to review the changes prior to merging to the master branch. The repository will check to automatically correct any `merge conflicts`

What is a Merge Conflict?

A `merge conflict` occurs when the branch that is submitting a `pull request` to the master branch was not originally up to date with the `master branch. This will cause GitHub to say: 

Merge Conflict Example

GitHub will not know which code/ text to change if both files started off fundamentally different. Here is an example.

Imagine a text file in the master repository, starting like this:

Master branch:

----------

**This is a brown cat.**

----------

Now imagine, that a development branch has an older version of the text file, like this:

Development branch:

----------

**This is a cat.**

----------

Now let’s say that the development branch starts working off this file, without the “brown”, and makes changes:

Development branch:

----------

**This is a cat. This cat is also fluffy.**

----------

Now this development branch wants to send a `pull request` from the development branch -> master branch. There will be a `merge conflict`. The development branch must have the most up to date version of the file they are trying to change, before making any changes.

Now let’s theoretically, make these changes again, but with the development branch using the most up to do date version of the text file.

Master branch:

----------

**This is a brown cat.**

----------

Now imagine, that a development branch has the most up to date version of the file, like this:

Development branch:

----------

**This is a brown cat.**

----------

Now let’s say that the development branch starts working off the most current file version:

Development branch:

----------

**This is a cat. This cat is also fluffy.**

----------

This `pull request` can now be merged because the development branch was using the most up to date version of the text file prior to the `pull request`.

Any `merge conflicts` will have to be review by the Admin, and resolved by the Admin.

**What is a pull request?**

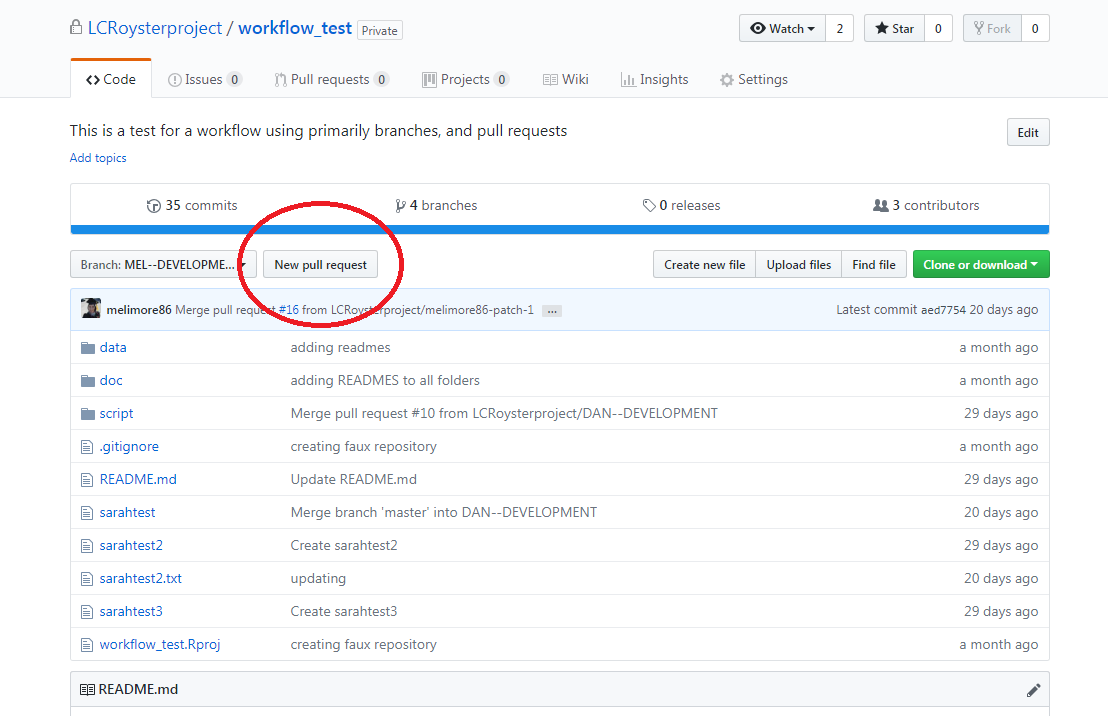
Pull requests let you tell others about changes you've pushed to a branch in a repository on GitHub. Once a `pull request` is opened, you can discuss and review the potential changes with collaborators and add follow-up commits before your changes are merged into the base branch.

Note: When working with pull requests, keep the following in mind:

* If you're working in the [shared repository model](https://help.github.com/articles/about-collaborative-development-models), we recommend that you use a topic branch for your pull request. While you can send pull requests from any branch or commit, with a topic branch you can push follow-up commits if you need to update your proposed changes.
* When pushing commits to a pull request, don't force push. Force pushing can corrupt your pull request.

After initializing a pull request, you'll see a review page that shows a high-level overview of the changes between your branch (the compare branch) and the repository's base branch. You can add a summary of the proposed changes, review the changes made by commits, add labels, milestones, and assignees, and @mention individual contributors or teams. ([https://help.GitHub.com/articles/about-pull-requests/](https://help.github.com/articles/about-pull-requests/))



**Each developer will have their own branch, in which they will be the only developer to work in it. Any change, update/change/modification**, will be sent to the master branch through a pull request from the developer branch -> master branch.   
Click on “New pull request”.

Pull Request Figure 1- Screenshot of where to start a New Pull Request.



Pull Request Figure 2- Screenshot showing how to compare the developer branch to the master branch. The developer branch is `Able to merge` because the developer branch was originally up to date with the master branch.

If there are any merge conflicts, contact your admin.

It is imperative that the `pull request`, to submit changes to the production master branch, be from the **development branch-> master**. This means that the developer is trying to merge their changes to the master branch.

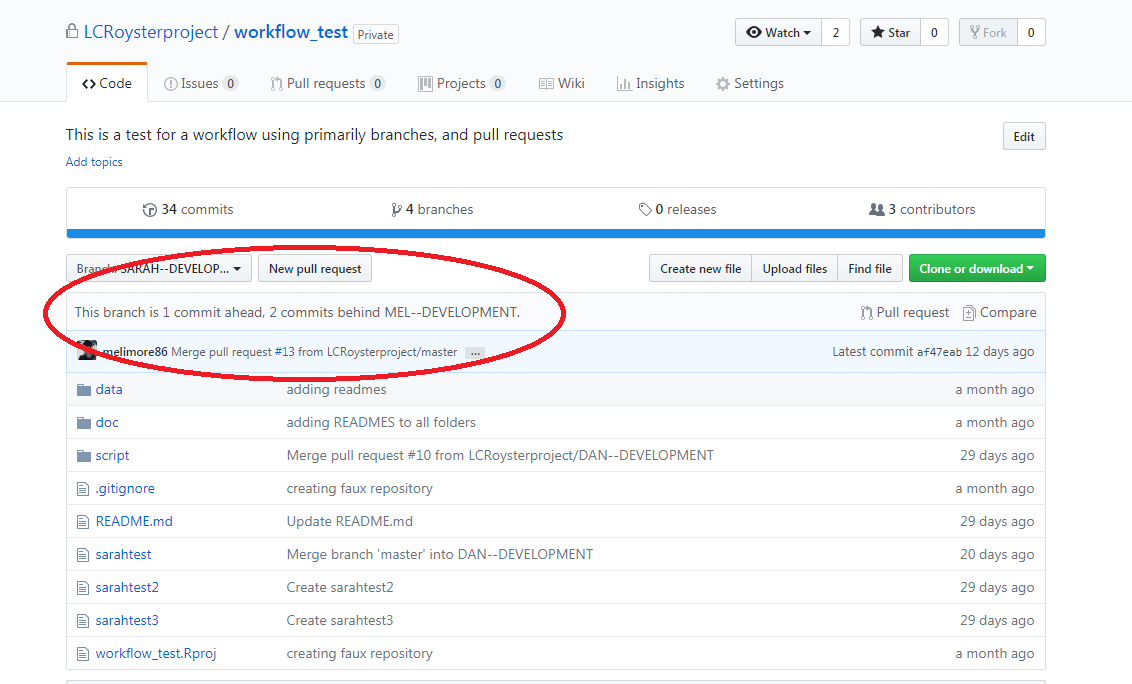
If your developer branch has no merge conflicts, there will be a message “Able to merge”. This means that the changes you are submitting were created using the latest version of the master branch, prior to creating the changes.

**Sync Developer and Master Branch**

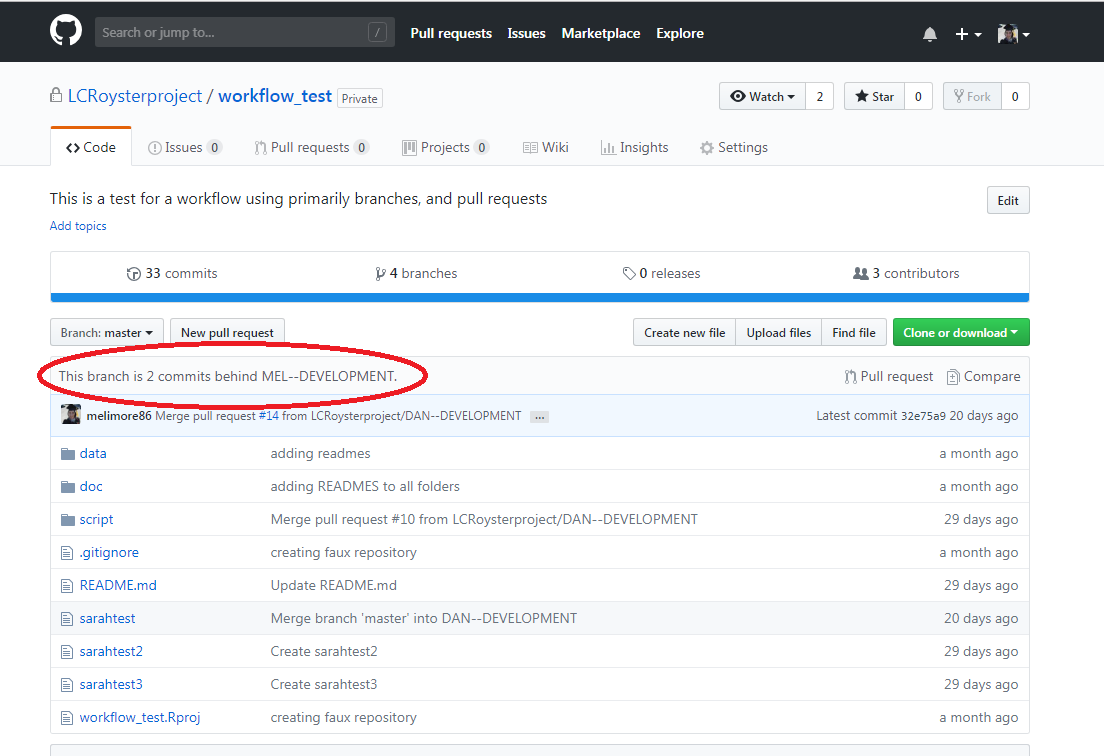


Here are a few ways to check if the developer branch is up to date with the master branch.

In the front of the developer branch, there will be a message on how many commits any branch is ahead, including developers and master branch.



Pull Request Figure 3- A developer can view if they are ahead or behind any of the branches in the repository.



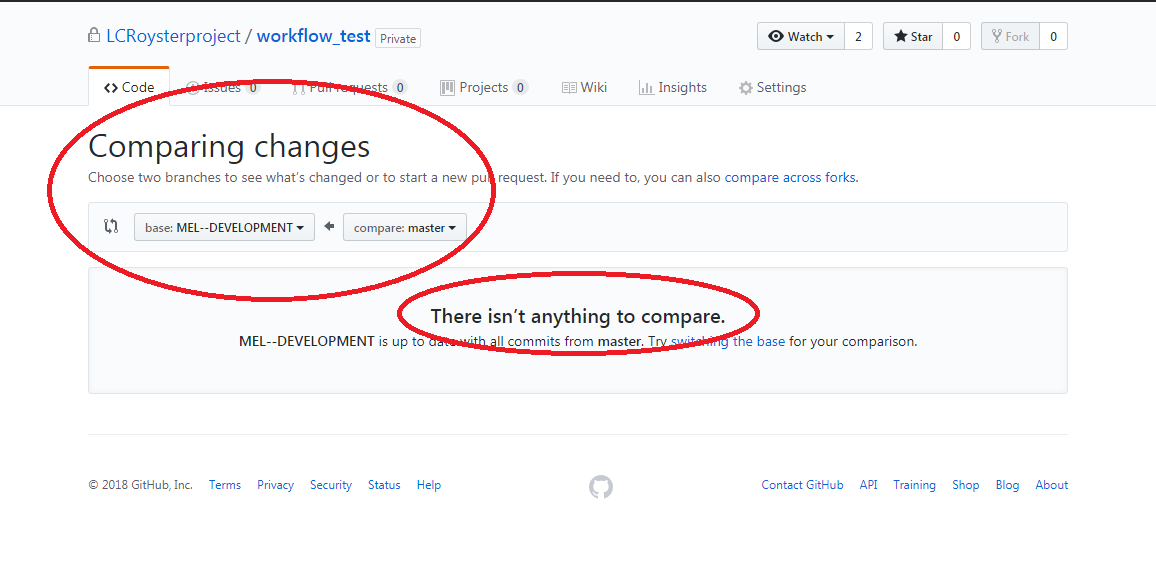
Pull Request Figure 4- The developer will not be able to make any changes to the master branch, but a developer can click on the master branch, to view if the master branch is ahead or behind of any of the branches in the repository.



A `pull request` can also be set from the **master-> development branch**, so that the development branch will have no merge conflicts when they submit a pull request (development branch -> master).

This does not need Admin approval, and can be done at any time, **BUT MUST** be done **EVERY TIME** that the developer wants to continue working on their branch, that is not up to date with the master. Use the previous steps to check if the developer branch is the same as the master branch.

If you see a message “There isn’t anything to compare”, then your developer branch is up to date with the master branch, and you can start making changes to your developer branch.



Pull Request Figure 5- Screenshot of the commit status of the master branch to the developer branch.



**ADMIN- Permissions Set Up**

Permissions for the repository are set up by the Admin. Permissions can be set up in Settings -> Branches -> Protected branches. From there the Admin will choose to protect the master branch, by clicking the dropdown box “Choose a branch” and selecting master. There will be many settings for the repository, but the minimum that should be selected are as below.

Minimum Branch Protection Settings

* Protect this branch
  + Require pull request reviews before merging
    - Required approving reviews: (at least 1)
* 2)  Included administrators

Of course more protection options can be placed, but these options are enough to protect the branch. With each protection option, there will be a description of what the protection option does.

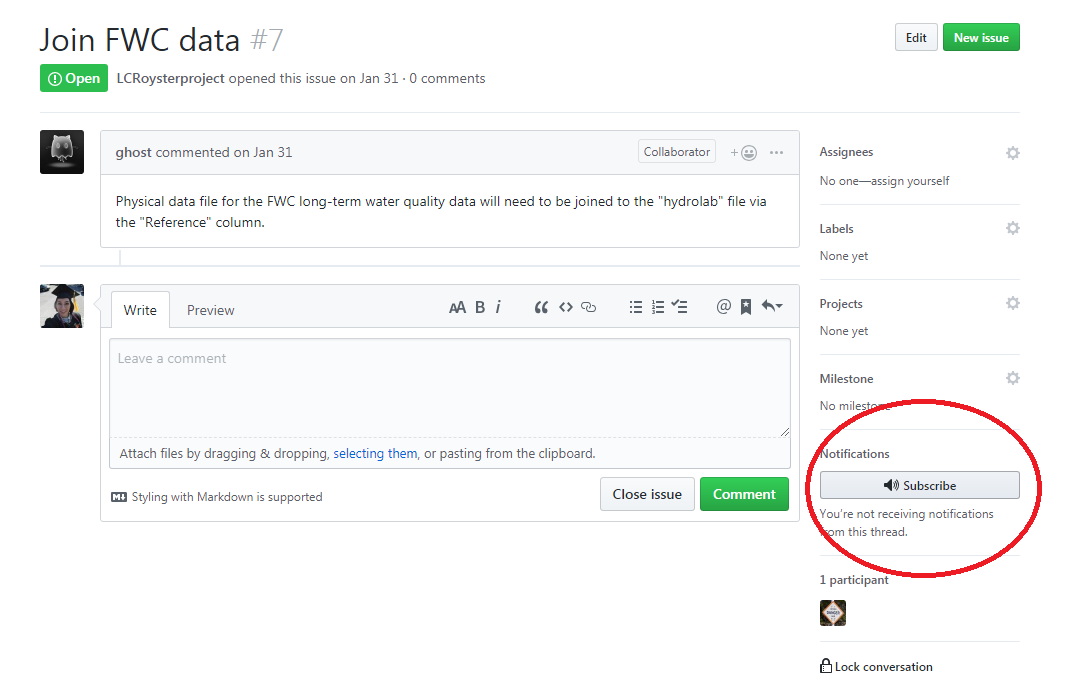
Below are all of the permissions that are in the “Branch Protection”, and the highlighted ones are the minimum permissions for the branch.

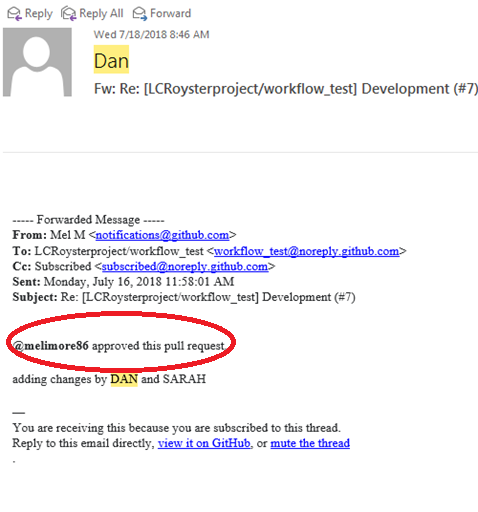
All Branch Permission Settings

Protect this branch. (Disables force-pushes to this branch and prevents it from being deleted.)

* Require pull request reviews before merging (When enabled, all commits must be made to a non-protected branch and submitted via a pull request with the required number of approving reviews and no changes requested before it can be merged into master.)
* Required approving reviews: 1 (at least one)
* Dismiss stale pull request approvals when new commits are pushed. (New reviewable commits pushed to a branch will dismiss pull request review approvals.)
* Require review from Code Owners (Require an approved review in pull requests that include files with a [designated code owner](https://help.github.com/articles/about-codeowners).)
* Restrict who can dismiss pull request reviews. (Specify people or teams allowed to dismiss pull request reviews.)
* Require status checks to pass before merging (Choose which [status checks](https://developer.github.com/v3/repos/statuses/) must pass before branches can be merged into master. When enabled, commits must first be pushed to another branch, then merged or pushed directly to master after status checks have passed.)
* Require branches to be up to date before merging (This ensures the branch has been tested with the latest code on master. This setting will not take effect unless at least one status check is enabled.)
* Require signed commits.(Commits pushed to this branch must have verified signatures.)
* Include administrators. (Enforce all configured restrictions for administrators.)
* Restrict who can push to this branch.( Specify people or teams allowed to push to this branch. Required status checks will still prevent these people from merging if the checks fail.)

**Developer- Notifications about submitted `pull requests`**

Developers will need to be have their email available to check on `pull request` activities. An email will be sent to the developer that indicates the status of their pull request to the `master` branch. Whenever the Admin submits a change to the developers’ ‘pull request`, the developer will receive an email with the `pull request` outcome. Only the branch that submits the `pull request` will receive any notifications about its outcome, so other developer will not receive an email. Another option is also subscribe to notifications about the pending `pull request`.

Notification Figure 1- Clicking on the Subscribe button will send a notification about the status of the pull request.

Notification Figure 2- Email sent from the ‘pull request’ update

**Reasons why pull requests are not approved**

* 1. Naming conventions are not appropriate( see Naming convention guidelines below)
  2. Wrong branch submitting pull request i.e. developer submitting pull request from another developer’s branch
  3. Ineffective code or text i.e code or text tested and not approved

Naming Conventions for files

-No spaces in file names ALWAYS use \_ (underscore)

- date first yyyymmdd, and then category/gear type ( i.e transect, quadrat, biomass, tong), following site if applicable

-i.e 20180802\_transect.csv, text file

-i.e 20180809\_salinity\_allsites.png, figure file, can use “allsites” for sites 1-9, but otherwise label the sites, 20180809\_salinity\_123.png

- No camels (upper and lower, ALWAYS lowercase)

- Invalid characters, i.e $,#,!, &,(),@,%,^,\*,”

Naming Conventions for folders

fig-figures

doc- documents, text files

data- collected data, usually in .csv or .xlsx

script- for R code, ‘data’ folder must be located inside for a R project  
compare- folder where script and files will be stored for future comparison

reconcile- folder for cleaning scripts and files can be stored, separate from compared folder

* Admin will always reject the files or folders unless they are in proper format.

**Developer- Git Bash commands for specific branches**

To make sure that developer is editing their specific branch, a series of commands needs to be entered to make the changes are submitted to the correct branch.

Clone repository to your clone machine

*git clone url*

Check branches of the repository (mandatory)

*git branch –a*

Checkout on USER DEVELOPMENT branch (mandatory)

*git checkout -b USER DEVELOPMENT*

Add changes to be pushed (mandatory)

*git add nameoffile*

or

Add changes to the whole branch

*git add -A*

Add a message for the commit (mandatory)

*git commit –m “message you are adding”*

Now push your changes

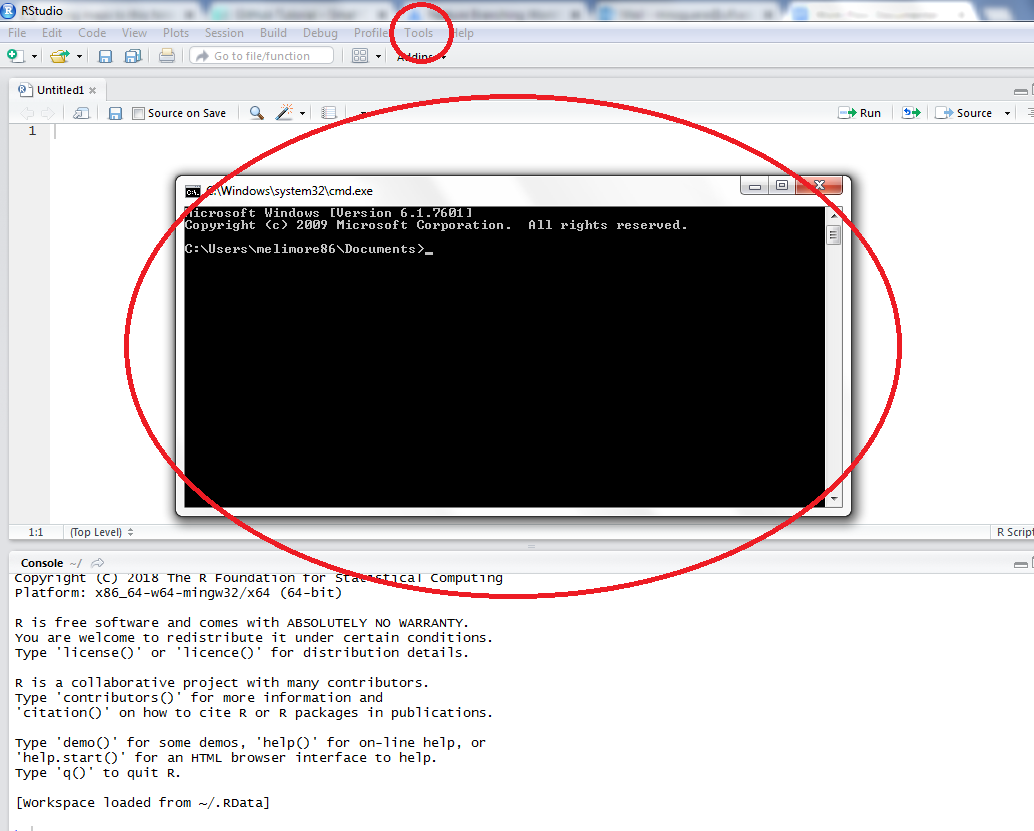
*git push origin  USER DEVELOPMENT*

Now the branch USER DEVELOPMENT is updated. Tests can be done on this new code in the DEVELOPMENT branch, and once it is verified, a pull request can be used to compare across branches before they are merged.

**Developer- R Studio and branches**

RStudio can’t create branches directly, so you need to either:

* create them in GitHub and pull the changes in your repository;
* create them from the Shell (Tools > Shell) and type:
  + git checkout -b USER DEVELOPMENT



Developer- Updating developer branch on local machine

R Studio branches Figure 1- Shell (Tools > Shell) and type: git checkout -b USER DEVELOPMENT, a prompt will appear where you can select the branch and then follow the steps in “**Developer- Git Bash commands for specific branches”**

**Admin- Creating an R Project**

Each repository will have an R project associated with it.

1.       Set up an R project in R studio

a.       File -> new project -> existing directory

b.       Find the location of your Data folder: for me it’s Desktop/git/Data

2.       If you already have an Rproject and you want to connect it to a GitHub repository:

a.       In Rstudio go to Tools -> Version Control -> Project Setup

b.       I have not actually done this before but this website may be helpful:<https://support.rstudio.com/hc/en-us/articles/200532077?version=1.0.143&mode=desktop>

**Data Validation in Excel**

The LCR project standards include a double entry system in Excel when entering discrete observations. Double entry system refers to entering data twice, by different users, to ensure data integrity. This is a standard practice in many data collection efforts. Currently, we are entering in transect, spat plate, and oyster height observations through an Excel worksheet packet. This packet needs to include the following.

Sheet 1- Field data sheet including all parameters needed for the data collection. Some examples include:

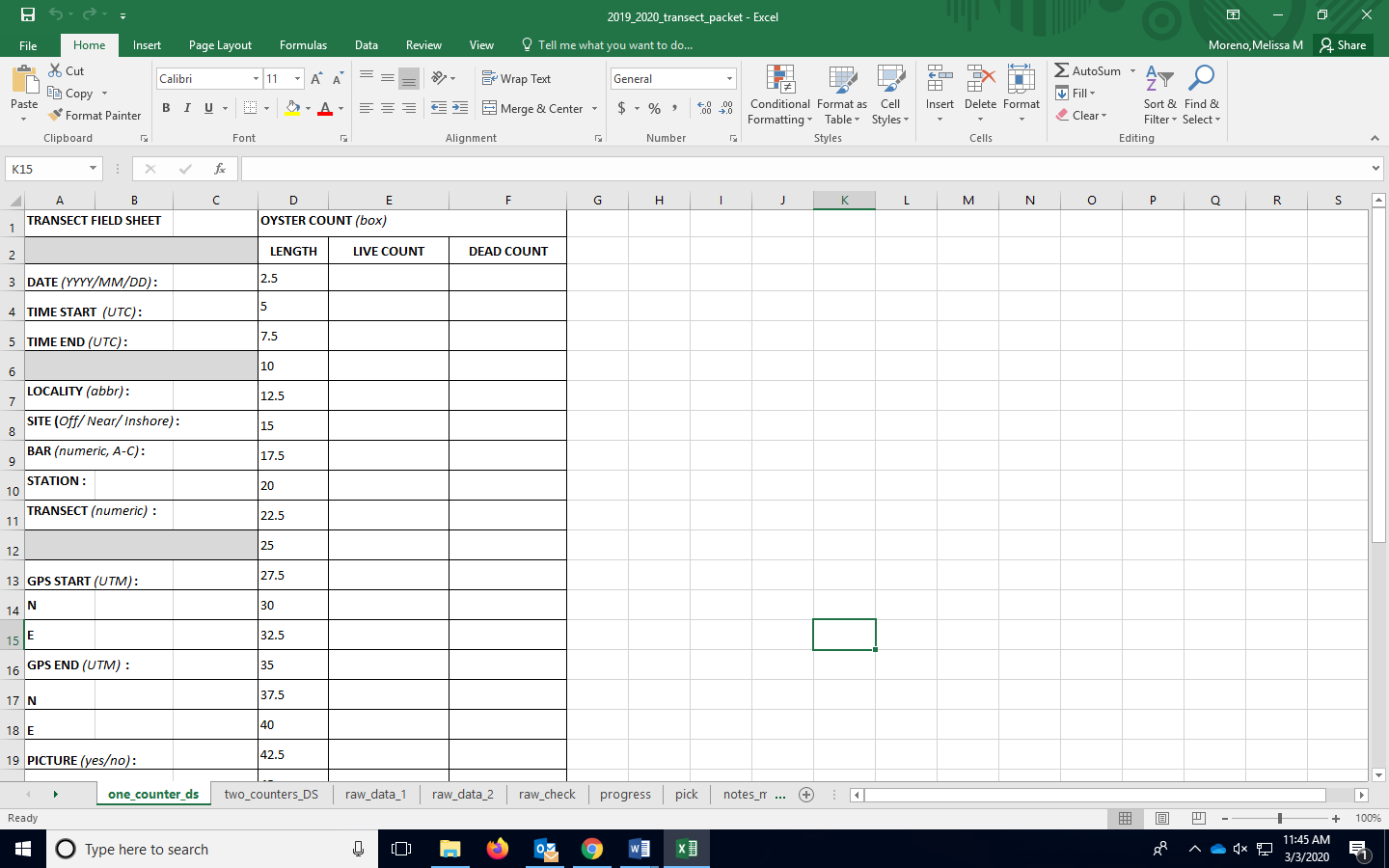
- Date- YYYMMDD in UTC

- Location (GPS) in UTMs

Your data sheet should also be:

- Clearly marked areas for observational counts/ measurements

- Formatted in a way to include units with all fields as well as format type



- It is advised to keep the physical data sheet in the packet, to ensure that all data needed is represented in this sheet.

Sheet 2- This is the first data entry sheet, which includes concise names for the columns. Columns must have no spaces in-between, preferably using “\_”. Columns should be in all on letter case, preferably lowercase.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **obs** | **date** | **year** | **month** | **day** | **start\_time** | **end\_time** | **locality** | **site** | **bar** |

-Units are not needed in the column names, because the units will be specified in the meta data sheet (Sheet 7).

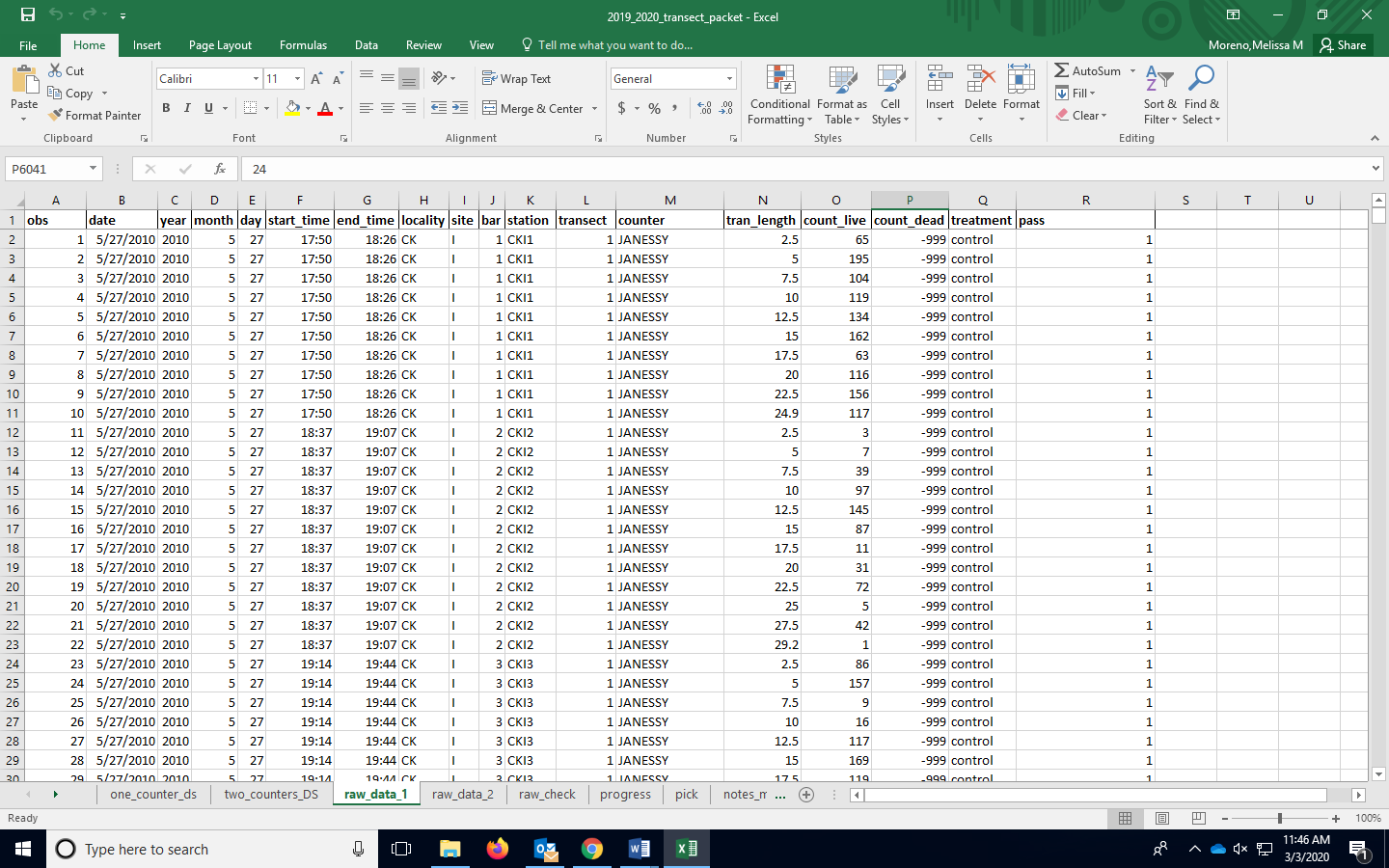
**- This sheet will have data validation parameters that will be set up from the data validation pick list, (Sheet 6).**

Sheet 3- This is the second attempt at data entry, which should have the exact same columns in the same order, as the first data entry sheet.

- All of the same parameters in Sheet 2 will be applied to Sheet 3.

- Sheet 3 format should be exactly the same as Sheet 2. The columns need to be in the same order, and have the same names.

**- This sheet will have data validation parameters that will be set up from the data validation pick list, (Sheet 6).**

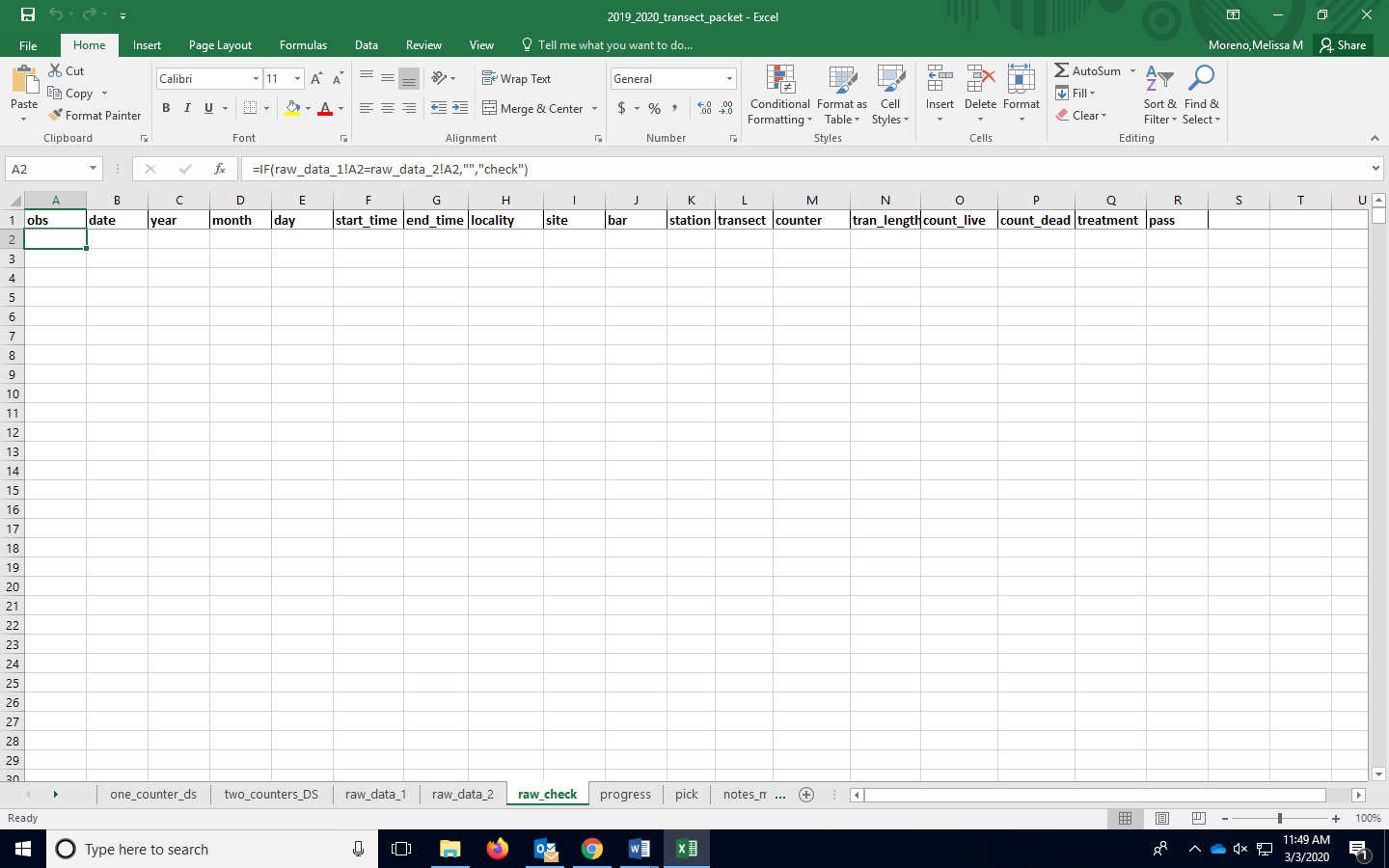


Sheet 4- This sheet should be solely for checking data integrity. This sheet will include individual cells that will need to be “checked”. You can use the equation for each cell in the data checking tab:

*=IF(raw\_data\_1!A2=raw\_data\_2!A2,"","check")*

-Make sure to apply this equation to all cells that will correspond to the double entry sheets. This worksheet needs to include all column names, in the same order, as the double entry sheets.

- If a “check” appears on the cell, it is up to the data manager, or third party individual from the two users that entered the data, to check the discrepancy. The data validation

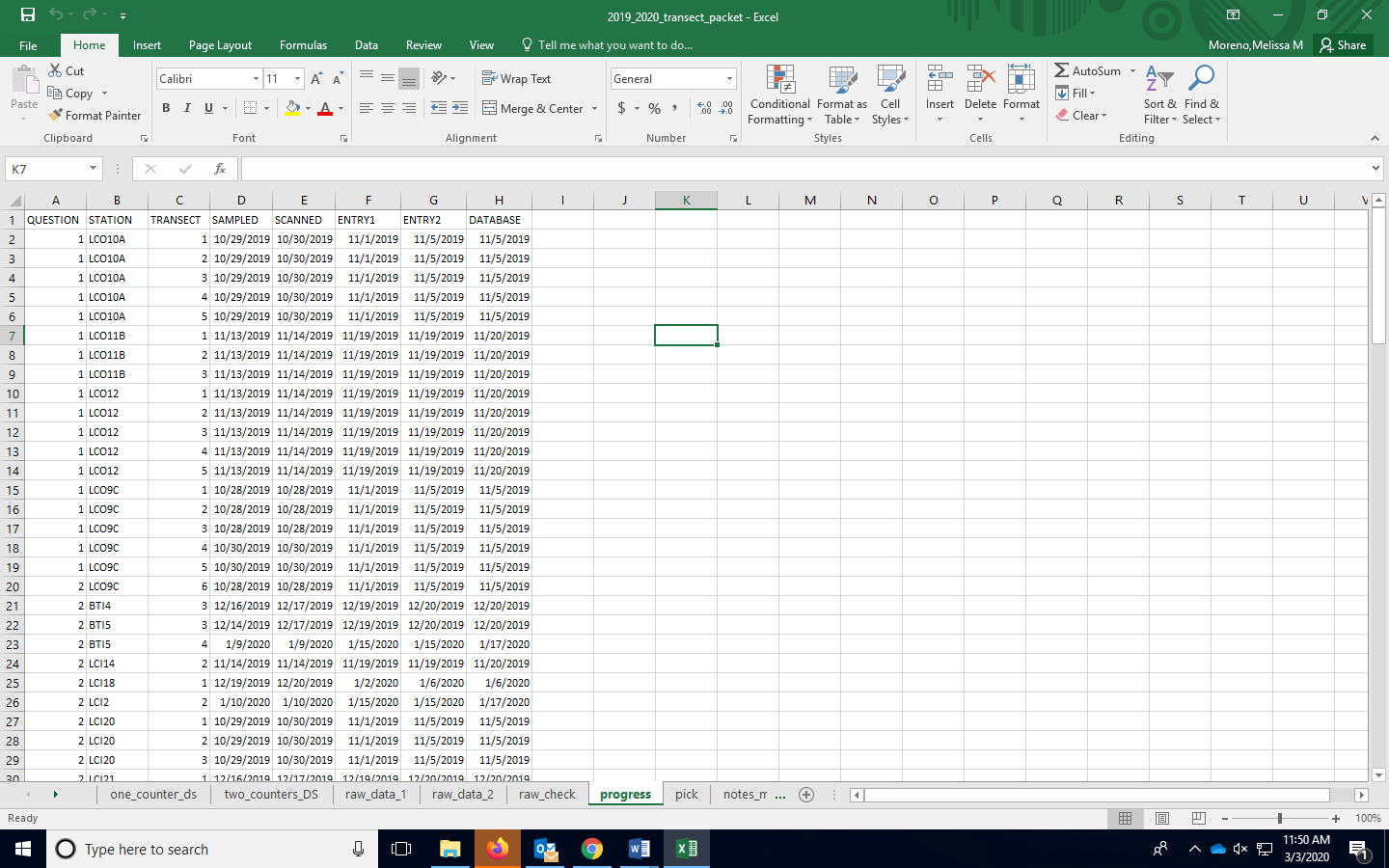


Sheet 5- Progress of the data collection, which is basically a summary of sampling events. Include similar columns (not all will be applicable):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **QUESTION** | **STATION** | **TRANSECT** | **SAMPLED** | **SCANNED** | **ENTRY1** | **ENTRY2** | **DATABASE** |

- This sheet will be for internal purposes and not usually used for analysis.

- Include fields that are necessary for the data collection. This will need to be discussed beforehand.

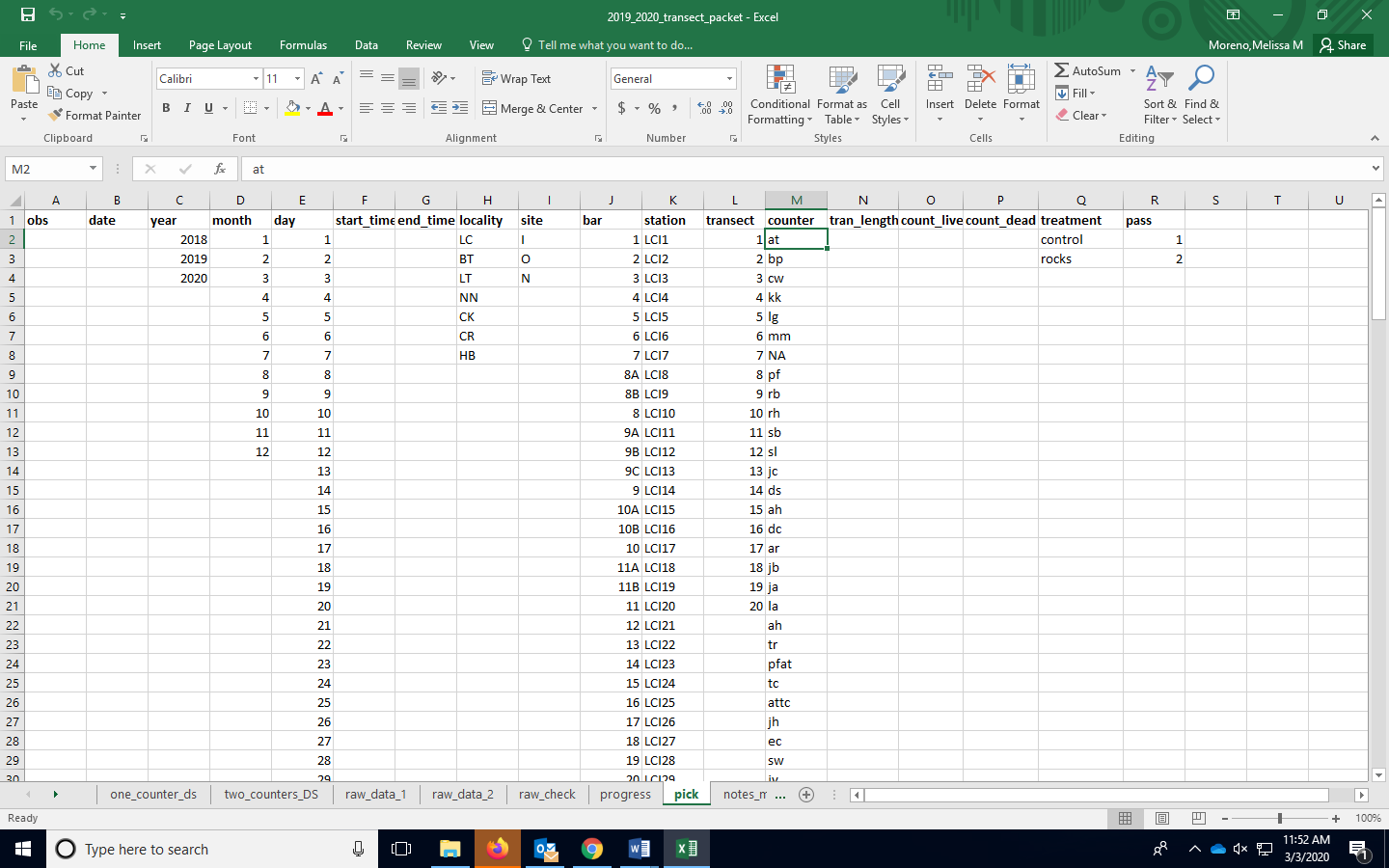


Sheet 6- A pick list that will have validation and govern over Sheets 2 and 3. Data validation ensures that individual columns will only have specific options that can be selected, and not entered by the user. When the use click on a cell in Sheets 2 and 3, they will be prompted to select one of the variables mentioned in this pick list sheet, to choose in the cell.

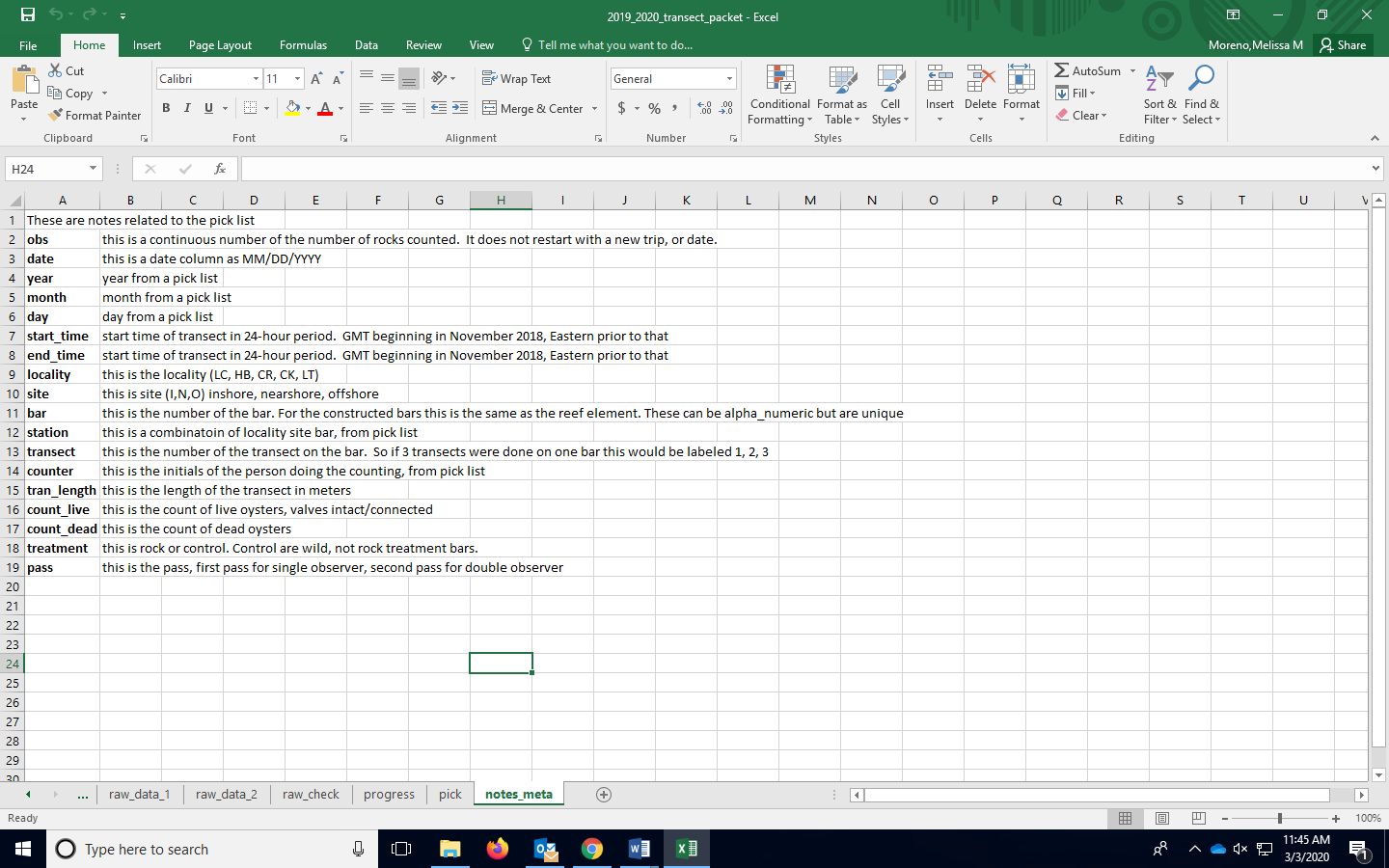
- The columns of this sheet need to be in the same order and same names as the double entry sheets, Sheets 2 and 3.

- Each column needs to have listed all of the possible variables that can be selected by the user. For example for month we only have the options 1 through 12, because there are only 12 months in a year. Without this data validation it could be possible for the user to enter 13, but having a pick list with data validation steps, will ensure that no selection outside of the allowed possibilities can be chosen by the user.

More information set up a data validation pick list can be found here: <https://www.officetooltips.com/excel_2016/tips/check_data_entry_for_invalid_entries.html>



Sheet 7- This meta data worksheet include the data entry (Sheet 2 and 3) column names and their parameters explained. All columns in Sheets 2 and 3 need to be represented in the meta data worksheet. The compilation of this sheet is also very important and highly advised.

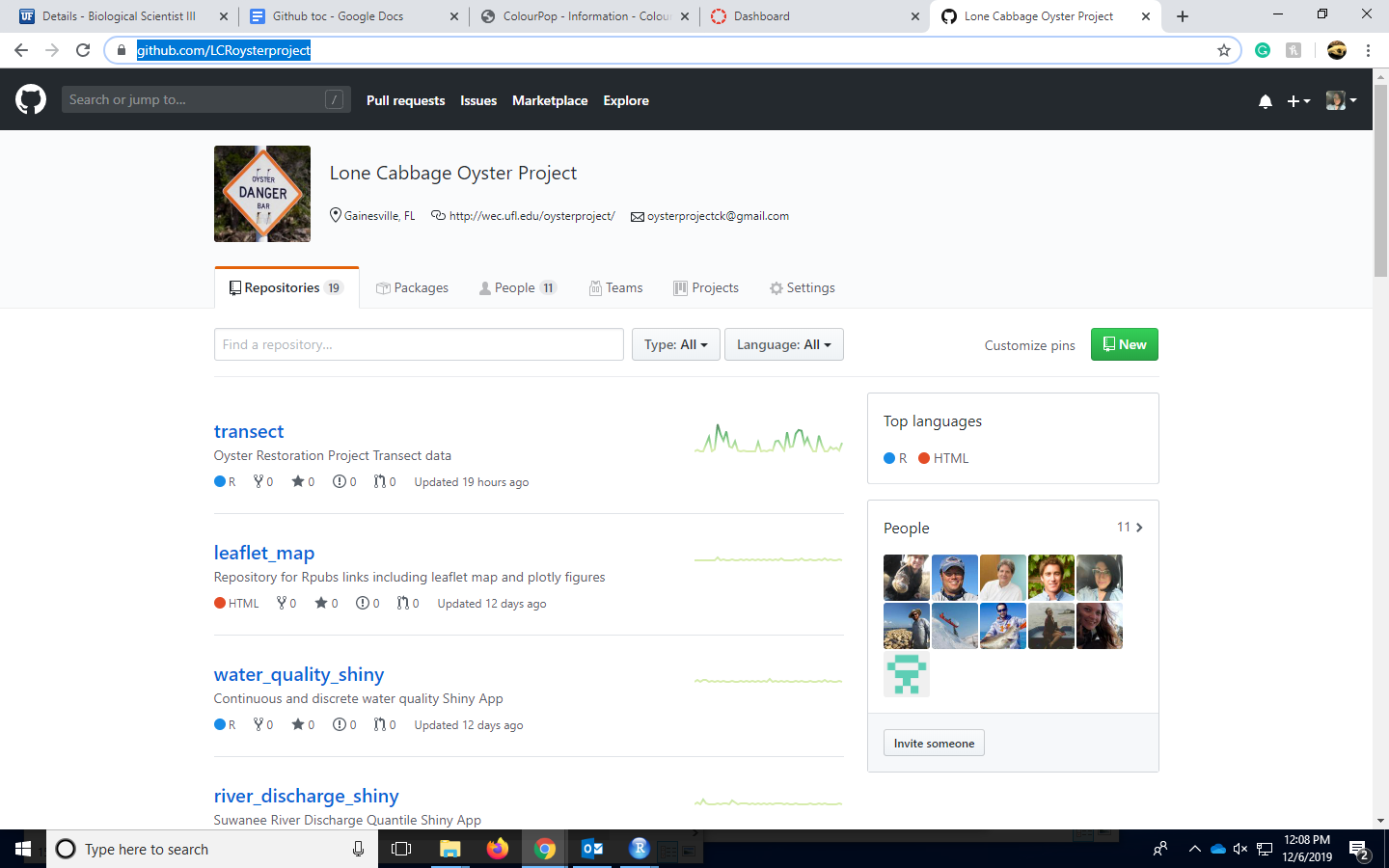


**Resources for Data Validation:**

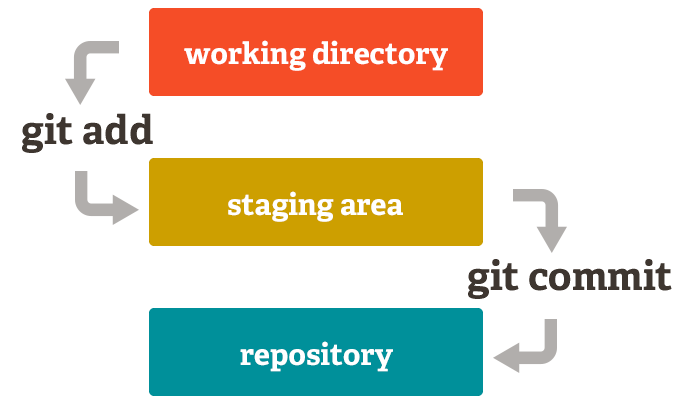
<https://support.office.com/en-us/article/apply-data-validation-to-cells-29fecbcc-d1b9-42c1-9d76-eff3ce5f7249>

<https://www.officetooltips.com/excel_2016/tips/check_data_entry_for_invalid_entries.html>

# Lone Cabbage Reef Workflow Theory

****

([https://GitHub.com/LCRoysterproject/](https://github.com/LCRoysterproject/))

Now that you are aware of GitHub, Git Bash, and RStudio Git, we will discuss the workflow for all the different ways you can use these tools either separately or in conjunction with one another. The idea of a workflow is to have a streamlined process on adding, editing, and finalizing documents and coding scripts. Workflows are used to create a systematic way for users to collaborate on documents/coding scripts. For the Lone Cabbage Reef project we are using a branching workflow. There will be several workflows laid out in this section that are used for the project. Examples and screenshots will be given.

Branching figure- Staging area workflow theory using *git add* and *git commit* commands.

(<https://vickysteeves.gitlab.io/repro-papers/git.html>)

# Local Machine Workflow Theory



As mentioned before, local machine refers to a user’s personal computer or laptop, essentially a local copy of the repository that the user is using. Below are the steps for local machine workflow theory.

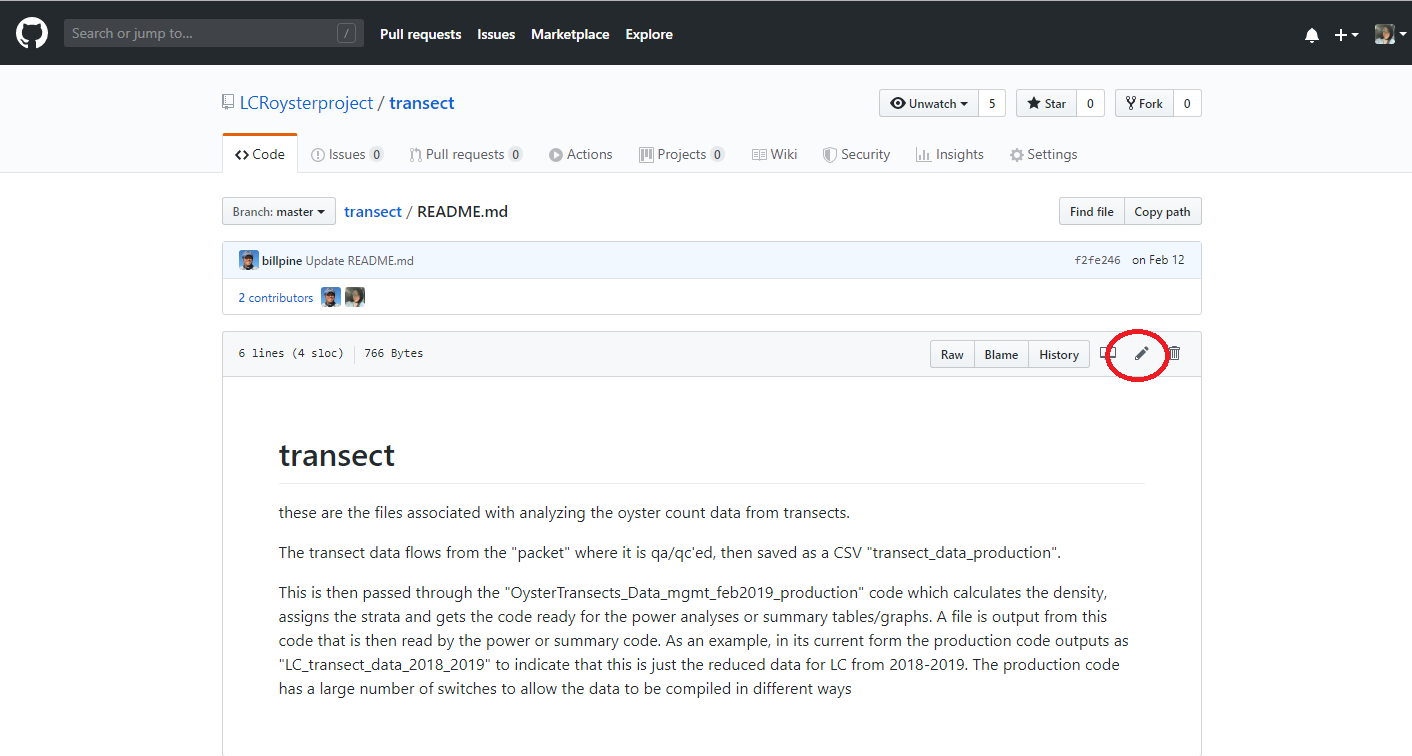
1. When a user modifies or edits a file/folder on their local machine, their changes are saved into their local disk space.
2. Using Git Bash, a user can “add” the change they want to be considered to be updated in the GitHub online repository. The user will use the git command, *git add filename* or *git add –A* (to add all changes files). It might be easier to use *git add –A* instead of *git add filename* if you are adding the changes to several files.
3. Once files are added, it is essential to commit these changes into a staging area. It is beneficial to add and commit multiple changes when a user is working on a document because Git Bash will save these commit messages, even if a computer shuts down suddenly. Git Bash will store all unpublished commits for up to 2 weeks, so it is important to make sure to add and commit changes regularly (<https://sethrobertson.github.io/GitBestPractices/>). To commit your changes type in Git Bash, *git commit –m “your message”*. The user should write a detailed commit message where the changes are described, any added or deleted files are also described.
4. Committed changes need to be pushed to GitHub to have the changes completely saved and recorded within GitHub.
5. Once the user has submitted the changes they have made successfully to their GitHub branch, it is important to communicate with the team about whether your updates should be submitted into the `master` branch. It would be safe to assume that not every branch update requires a pull request to the `master` branch,
6. If it is agreed that the work is appropriate to be submitted to the `master` branch, the user will create a pull request within their branch to the `master` branch.
7. Once a pull request is created, GitHub will direct you to enter a message on why you are creating the pull request.
8. In the following screen, an Admin will review the changes and choose to approve, deny, or request additional edits. The original user that created the pull request will receive e-mail updates about the status of the pull request. It is also important to communicate with the Admin reviewing the pull request for additional information about the status.
9. Once the pull request is approved and the changes implemented into the `master` branch, the user will want to make sure all of their changes or any additional changes that the `master` branch might have are applied to the user’s personal branch.
10. A user can create a pull request from the `master` branch to their own branch without any approval by an Admin. This can be done any time to ensure that the user’s personal branch is up to date with the `master`’s before the user starts to make changes on the files of their branch.
11. Without submitting a pull request from the `master` branch to the user branch, it is very possible that a merge conflict will arise in the future when the user tries to submit their own edits to files that are not up to date with the `master` branch. Merge conflicts will be discussed in detail in another section.

# GitHub Online Portal Workflow Theory

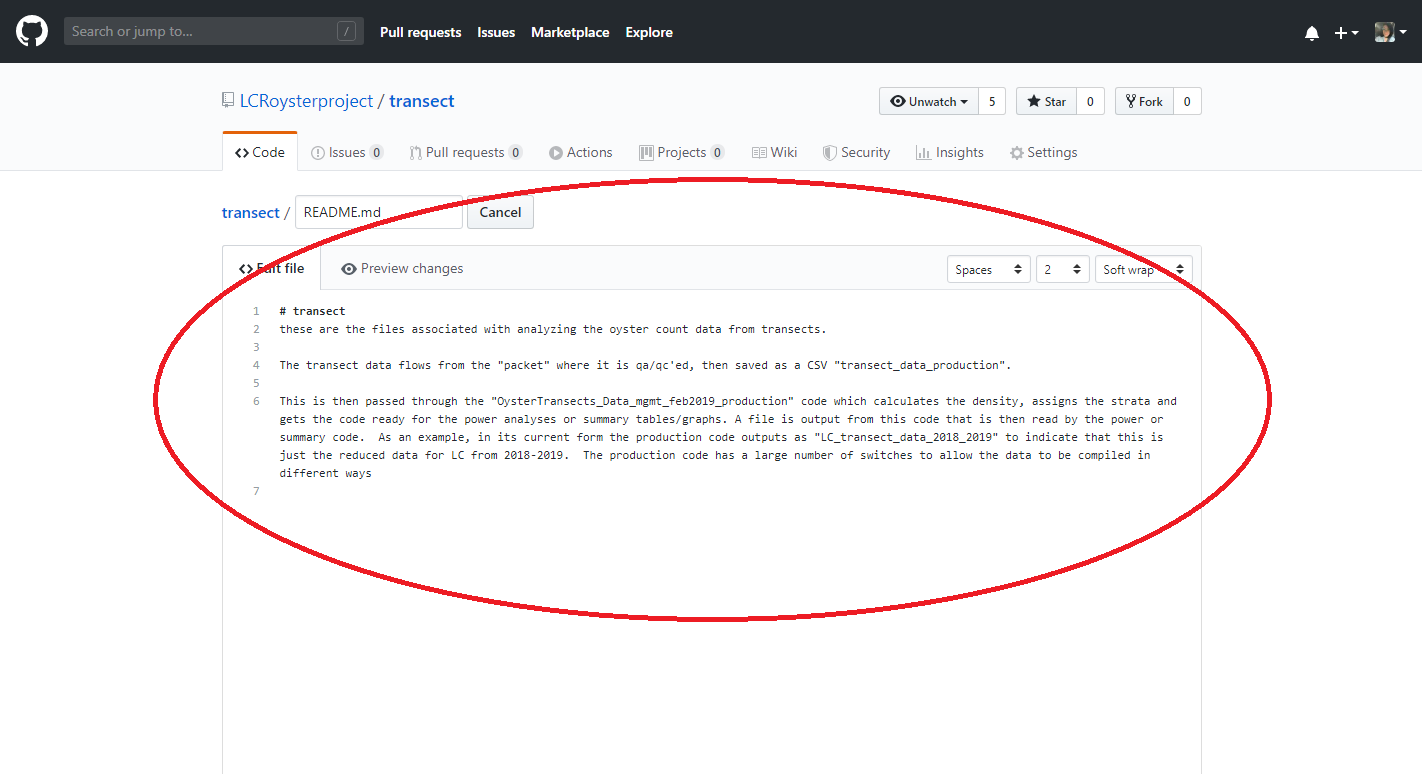


Working within the GitHub online portal can be easy and initiative. The GitHub online portal allows the user to edit, add, and delete files/folders within the repository. Protected branches, such as the ‘master’ branch will not allow for any changes without approval. Unprotected user branches are able to be modified without prior approval. Below are the steps for the GitHub online portal Workflow.

1. Only some file types are allowed to be edited directly in GitHub. Some file types such as .docx will need to be downloaded and opened to be edited, and then dragged into the repository for the new changes. Some file types that you can edit directly are R files and .txt files. Select the file that you would like to edit. Click on the pencil icon. The screen will change into a new editable screen. You can edit the text in the document directly.

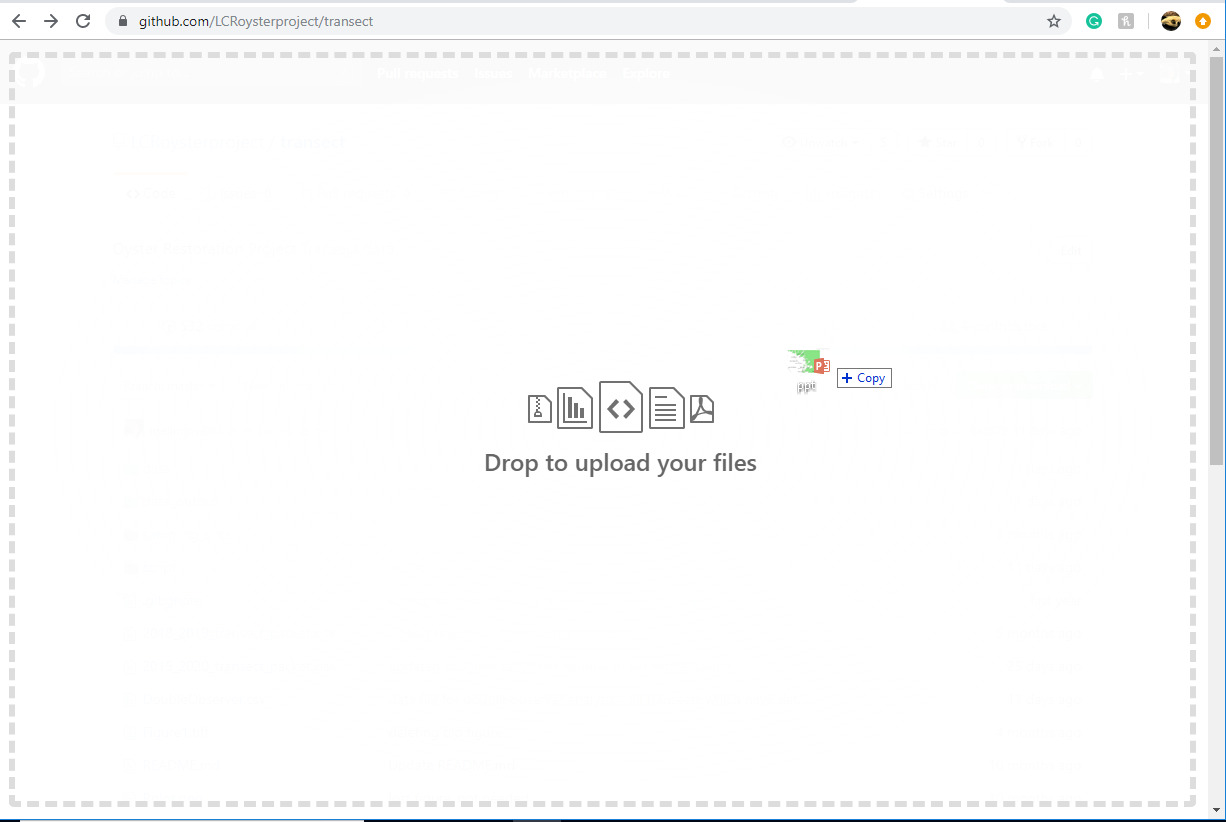


Screen shot- Pencil icon on top right of selected file.



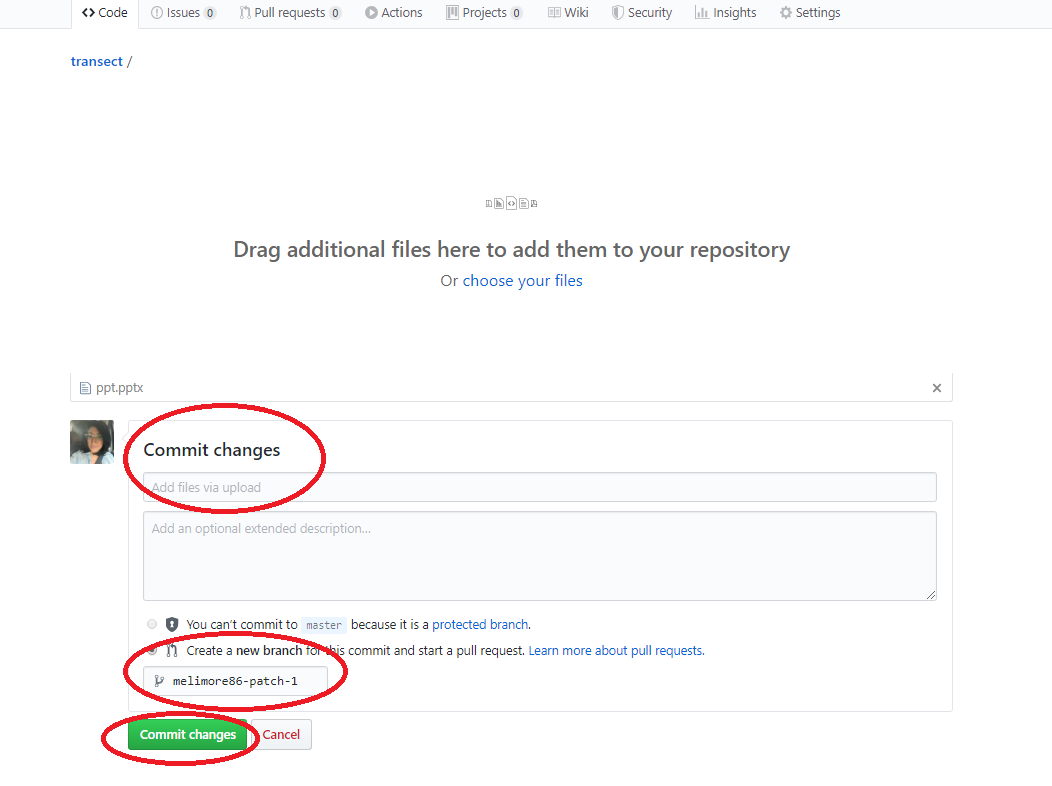
Screen shot- Editable screen in GitHub.

1.2. A user can also choose to drag changes into their folder or where a file is located



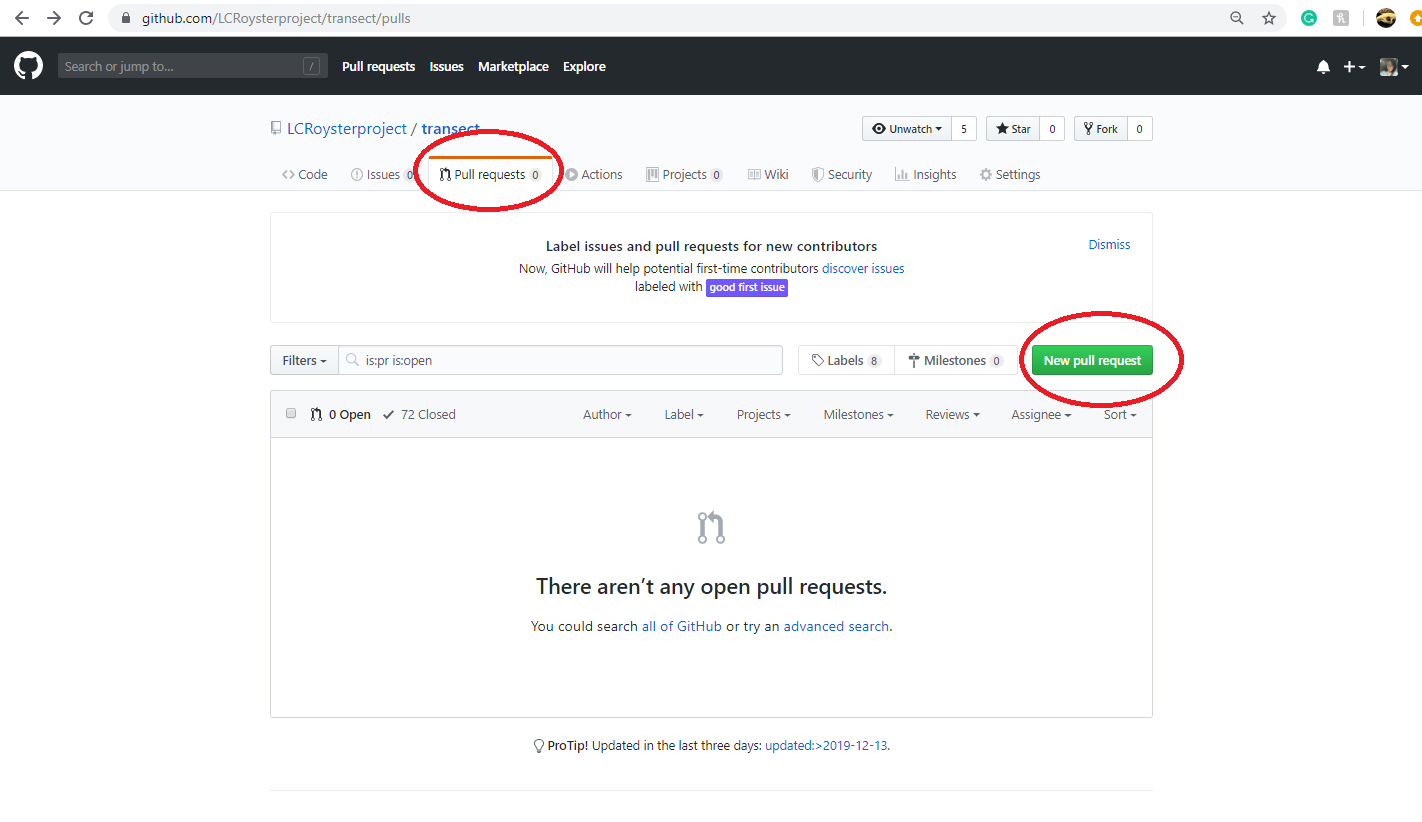
Screen shot- GitHub online portal drag method to update or add a new file.

branch, git iconStep 2. Add a commit message to your changes, make sure to be descriptive and detailed. Double check which branch will be updated by viewing name next to the branch icon.

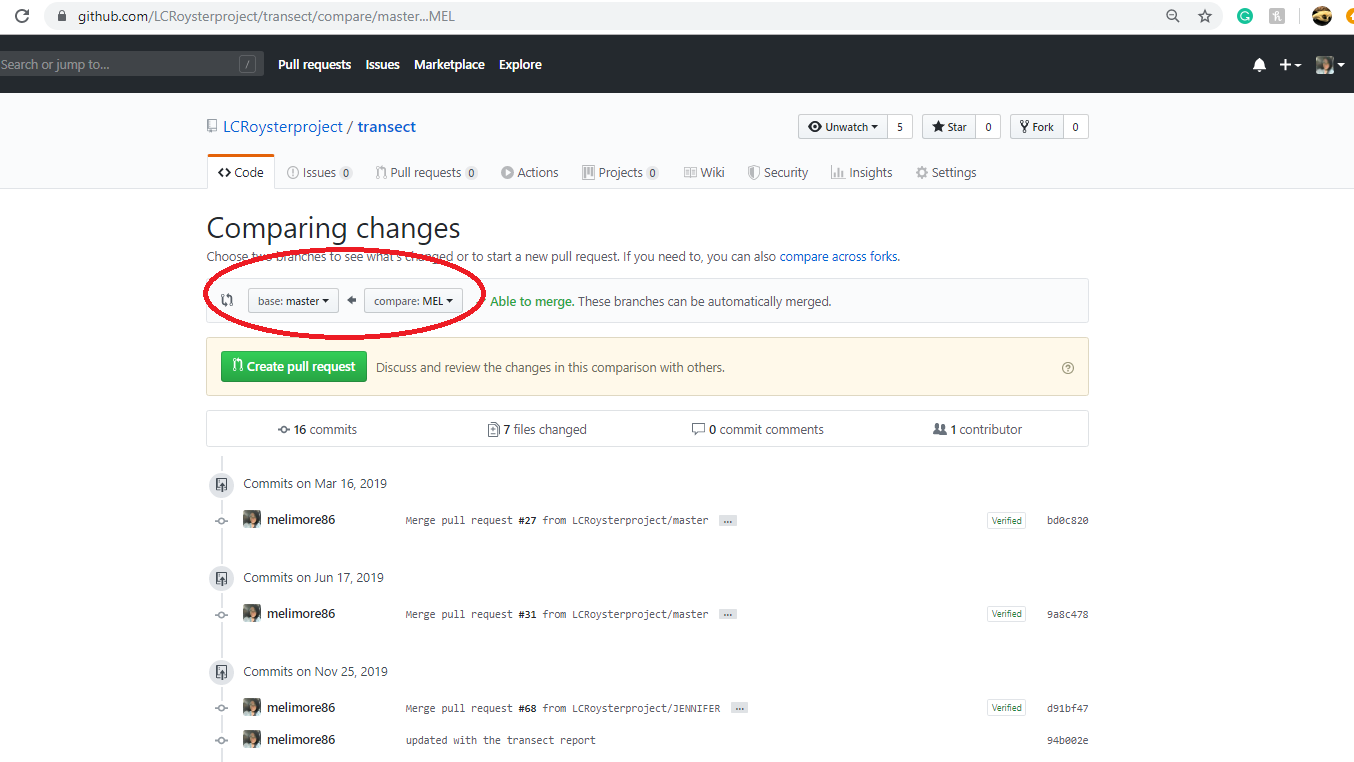
Step 3. Commit the changes.

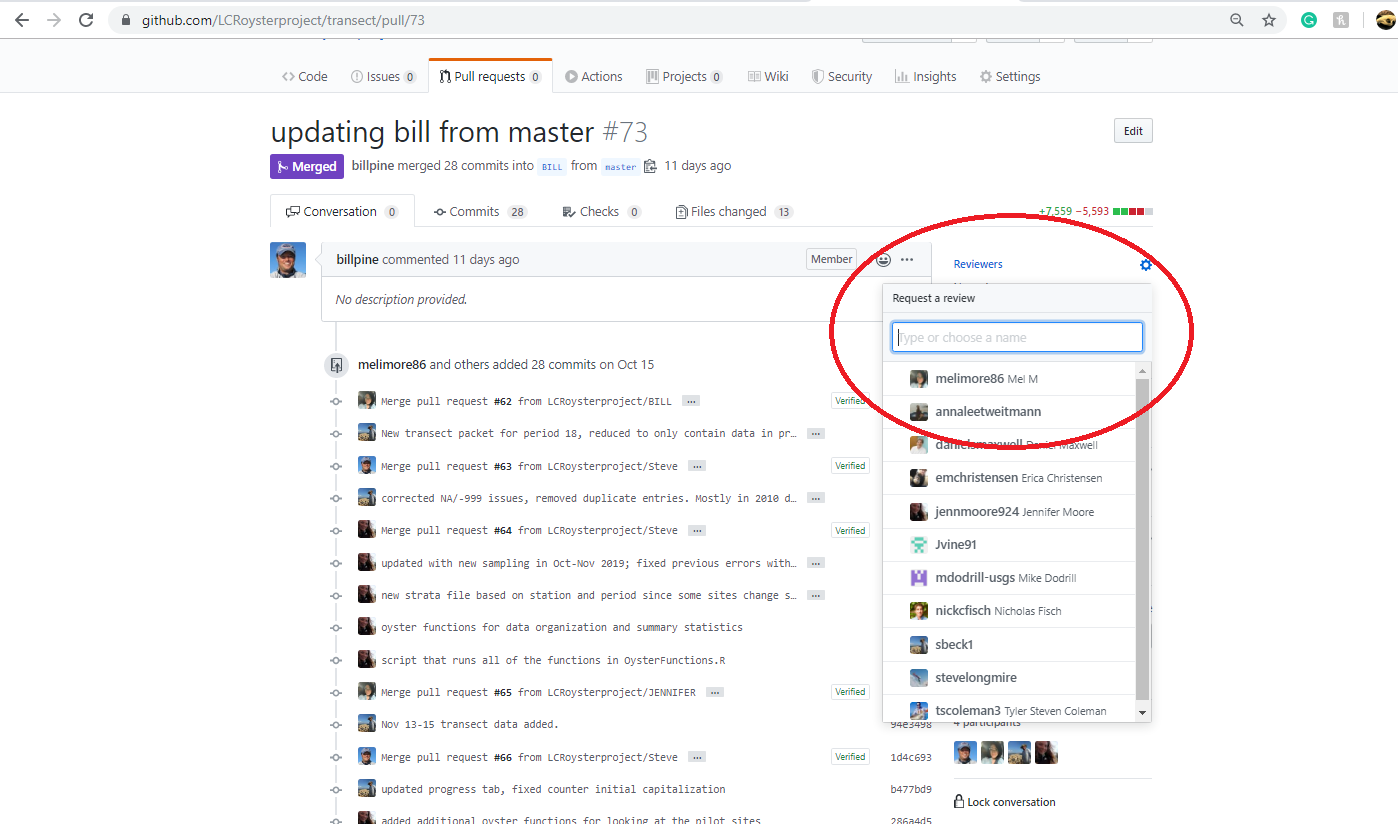
Screen shot- Steps 2 and 3.

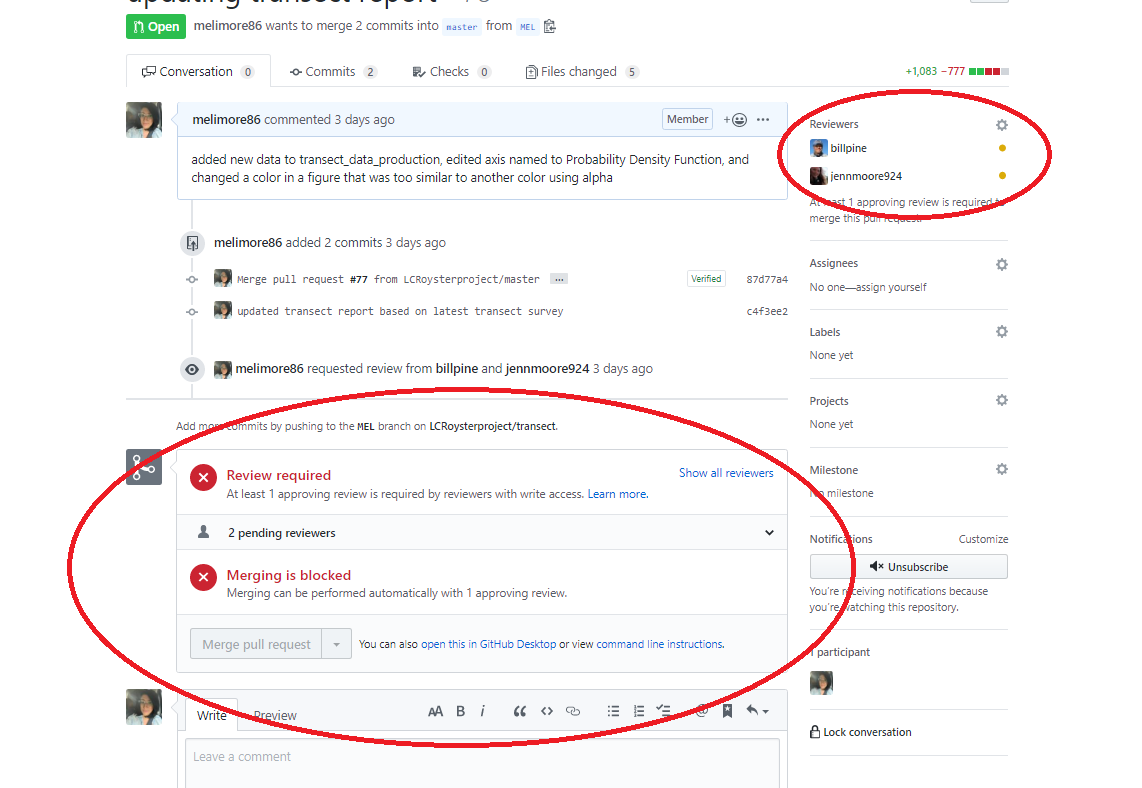
Step 4. Communicating with the team is essential for the user to know if their work should update the `master` branch. To submit a pull request from the user branch to the `master` branch, click on the **Pull Request** tab and click on the button **New pull request**.



Screen shot – Location of New pull request button.



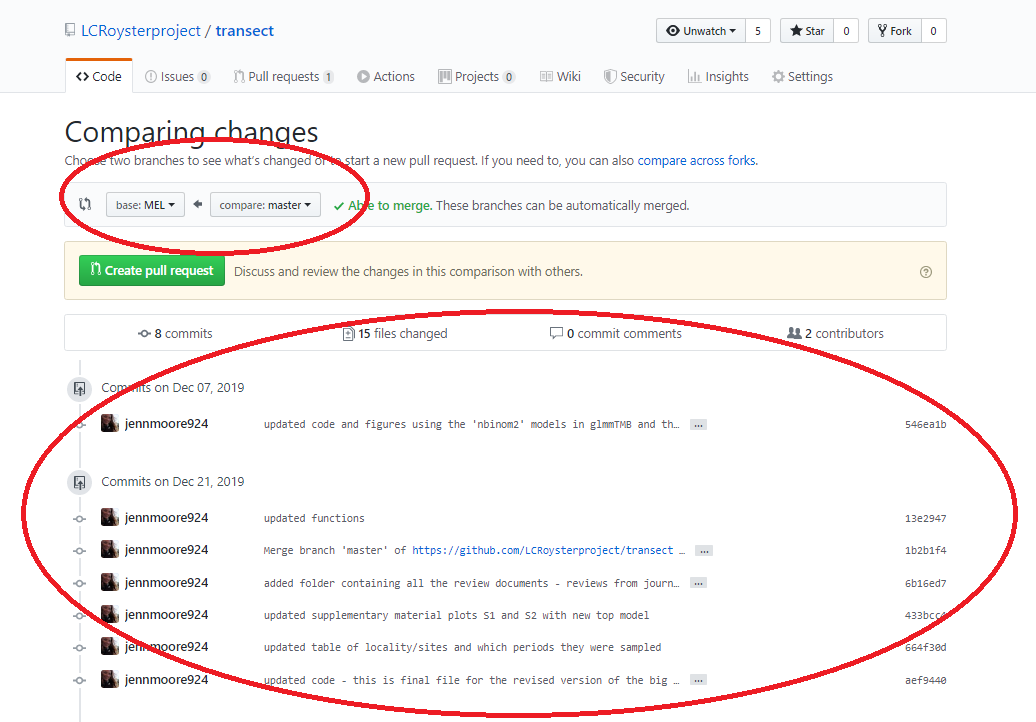
Step 5. Admins, selected in the Reviewer drop down box, will approve or deny all pull requests. The Admin can choose to approve the updates, deny the pull request, or request for additional changes before the pull request is approved.

Screen shot- Select the reviewers for the pull request.

Screen shot- Once selected a similar screen will appear with the Reviewers identified and a block to merging

Step 6. All users are required to update their branch prior to starting any work. **A pull request from the master -> user branch is needed.** Note that commits that have not been applied to the individual user branch are listed below the pull request. There are no blocks or reviewers needed for this kind of pull request.

\* Note that if the pull request from master -> user branch isn’t either checked, or applied to the user branch, it is possible that merge conflicts will arise.



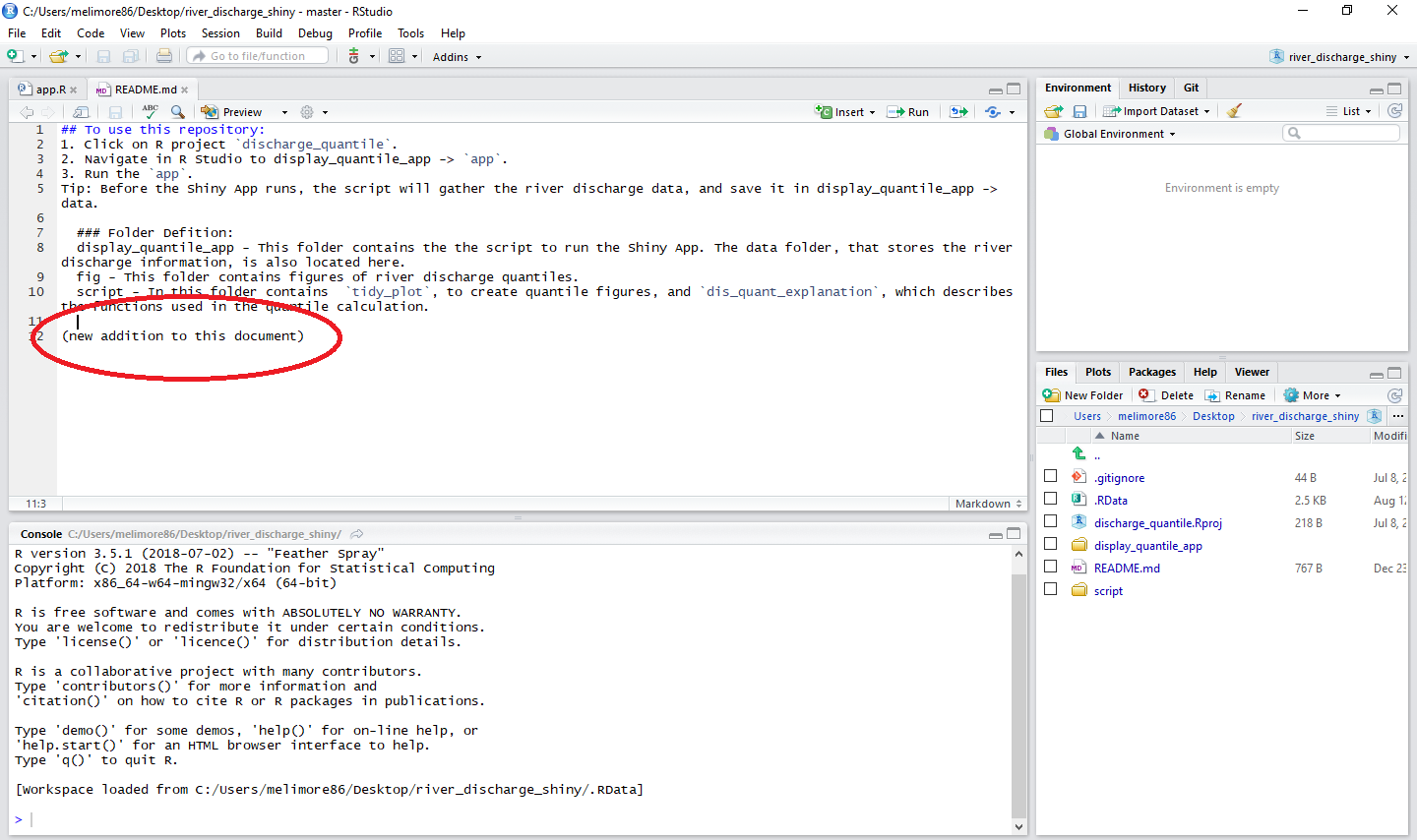
Things to consider:

- Ensure that there are restrictions on the ‘master’ branch before starting this workflow.

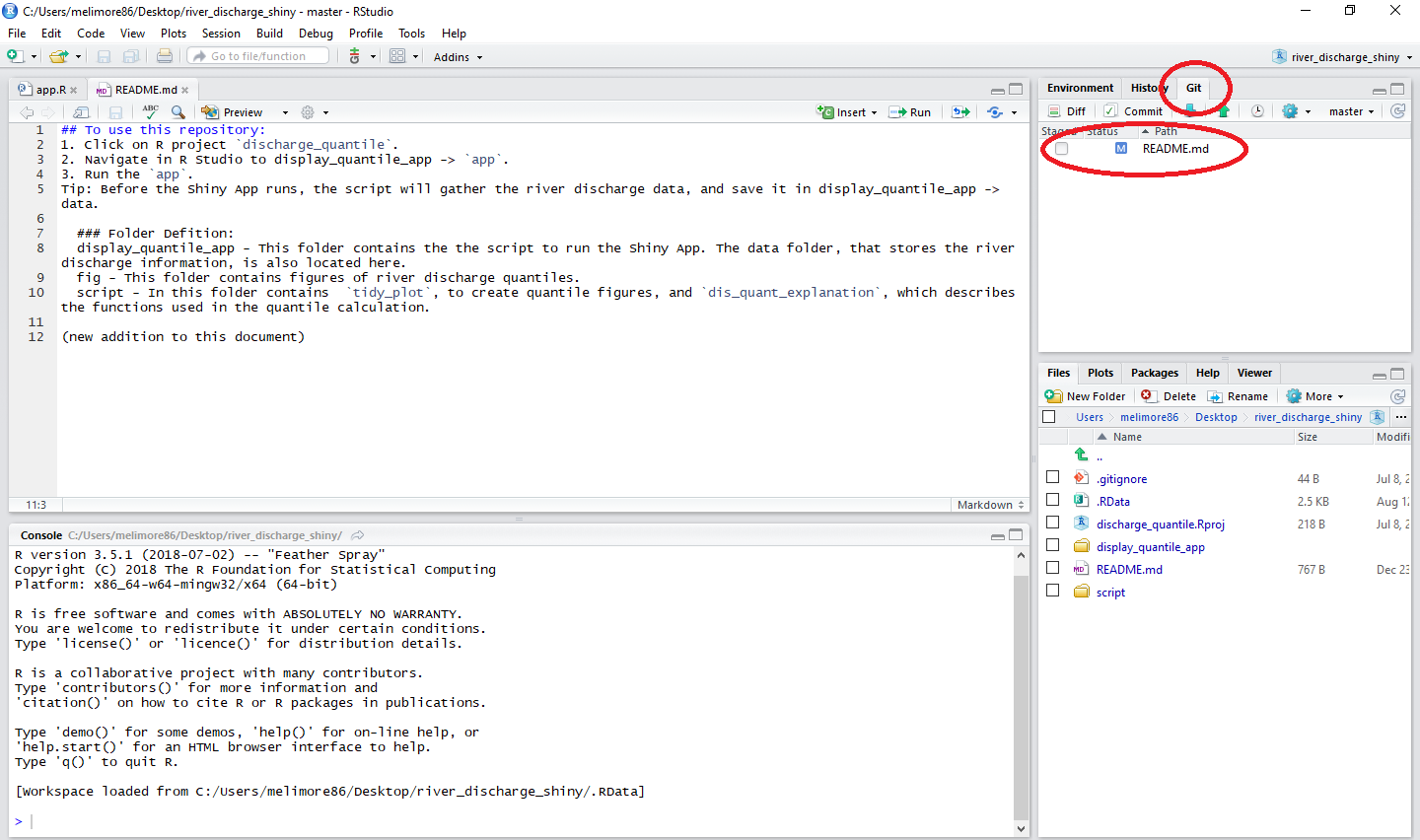
- Users will usually have to be added to the LCR Oyster Project, or individual GitHub repository as collaborator. Their permissions can also be edited any time during the life span of the project (<https://help.github.com/en/github/setting-up-and-managing-your-github-user-account/permission-levels-for-a-user-account-repository> ).

# R Studio (Git tab) Workflow Theory

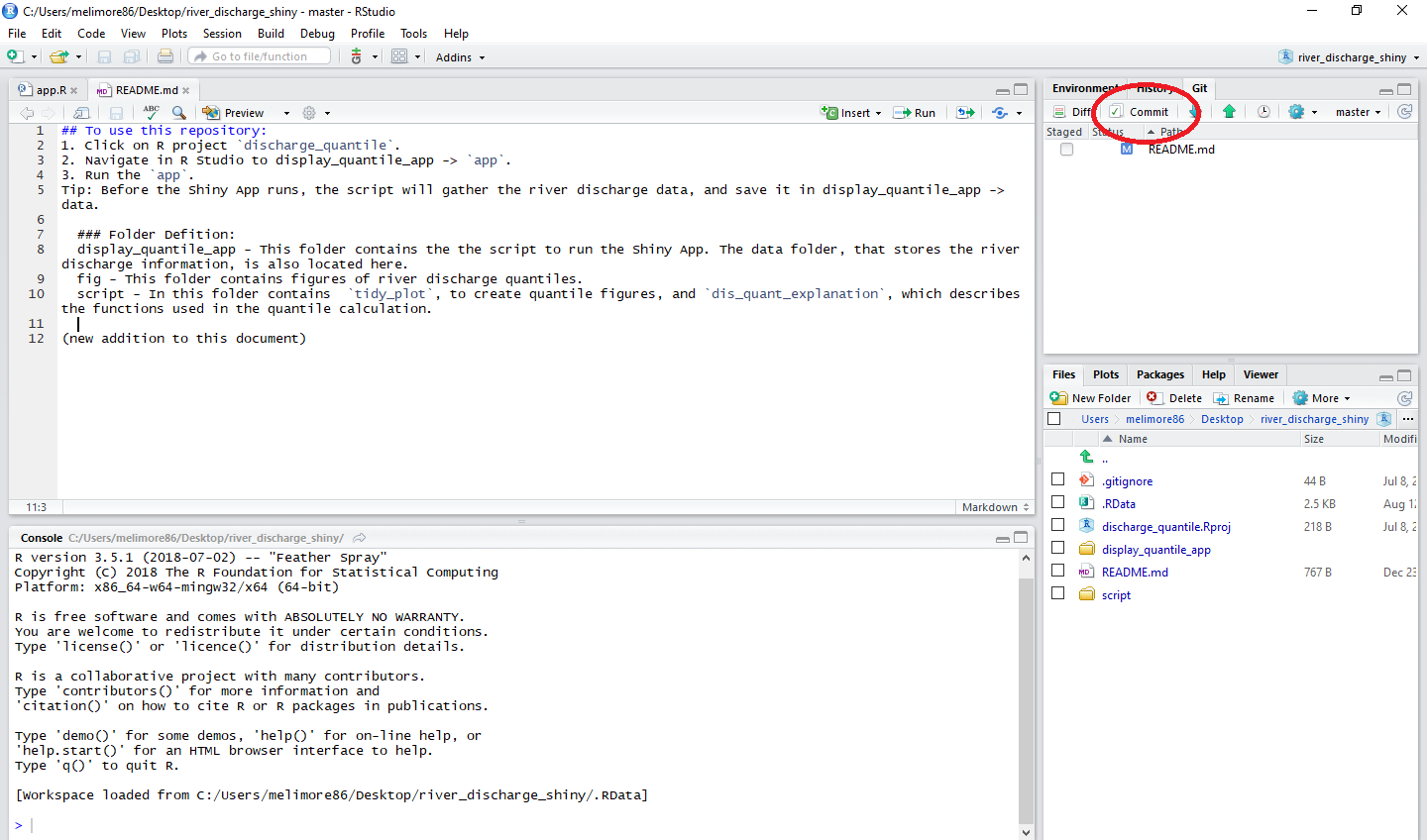


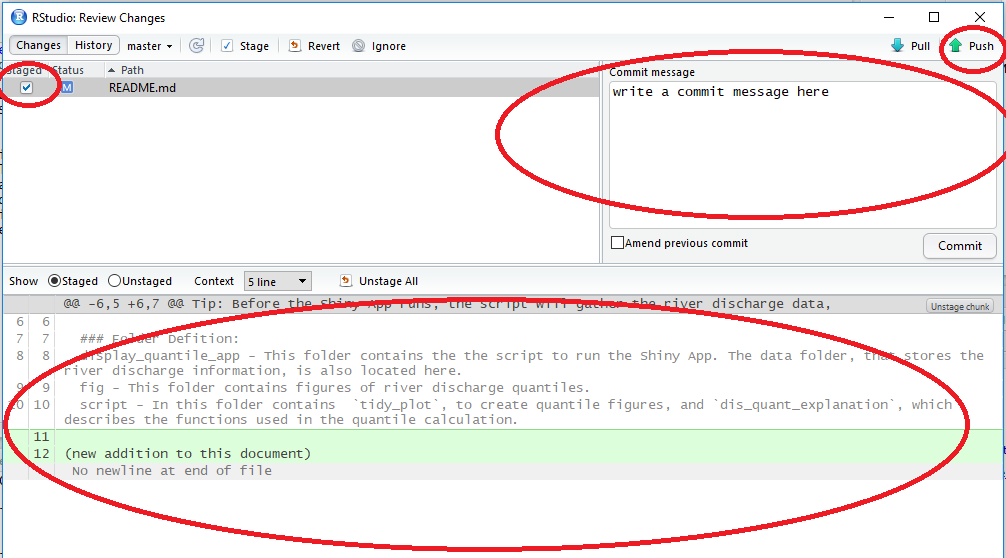
Step 1. Use R Studio to make changes to scripts or text.

Step 2. Click on the Git tab, near the top right. There will be a list of any Additions (A, green), Deletions (D, red) and Modifications (M, blue). Click on the change you want to commit, small clear square on the side of the change.



Step 3. Click on the Commit button. A new screen will appear.

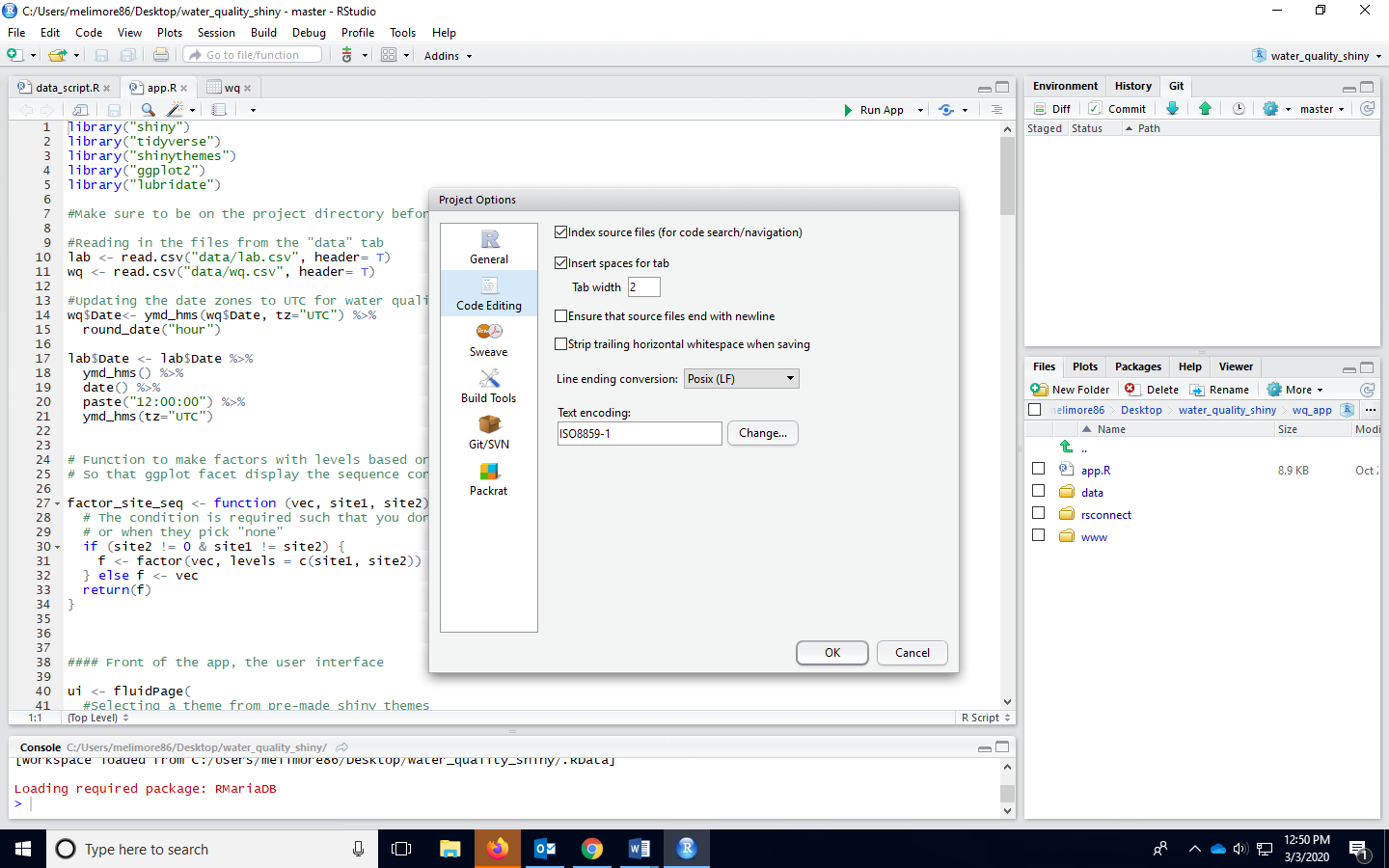


Step 3.1. The new screen will allow the user to select the staged elements, if not selected before, and to write a commit message. The commit message should be detailed according to the change that are being submitted. In section below, additions (green) and deletions (red) will be highlighted. Once everything is selected and written, click on the Push arrow.

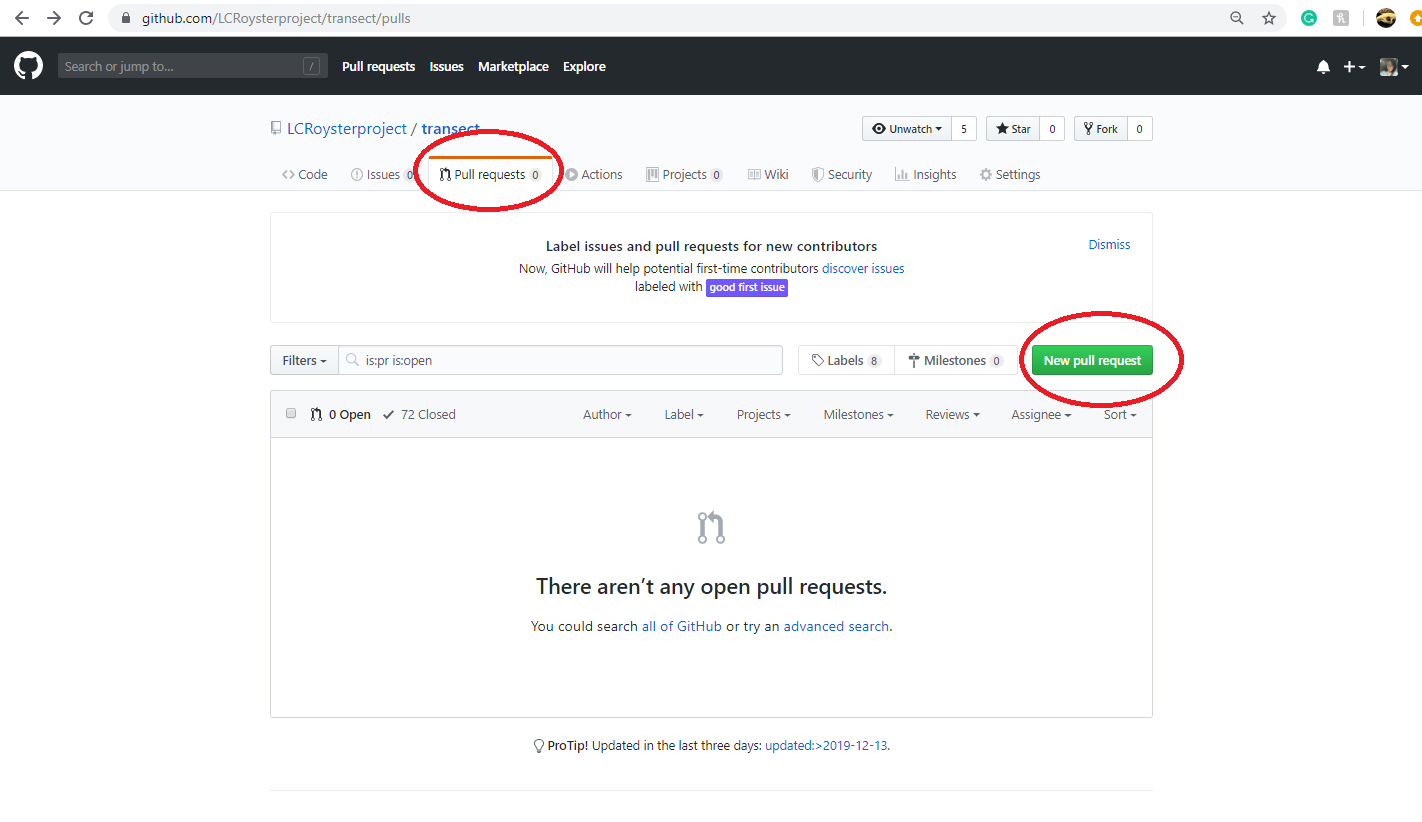
\* Note- If you are using Windows please make the following changes to your project set up.

## Changing Windows Line endings

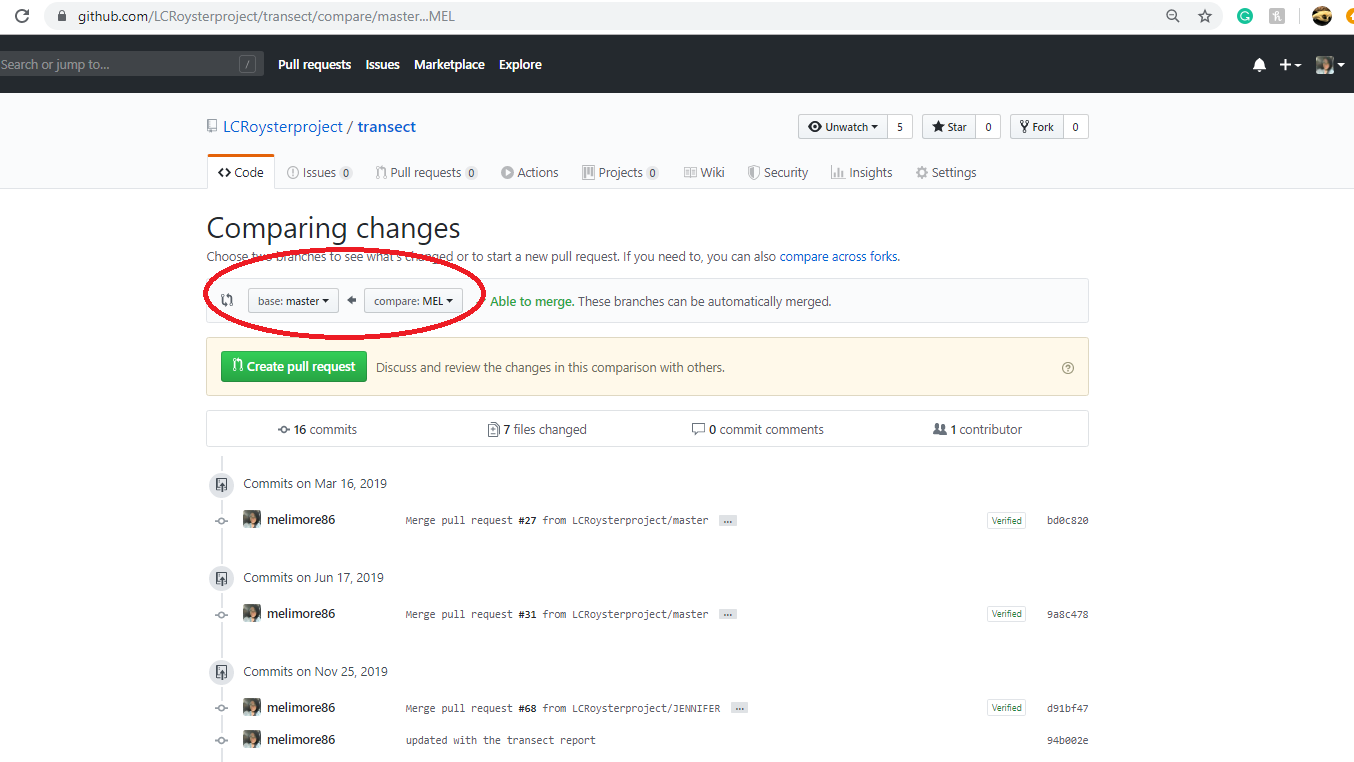
Changing the Windows OS line endings will help to correctly add and commit in the Git Tab in R Studio. Without changing the line endings, R Studio will register all of the changes as one large when submitting a pull request. It is easy to notice, especially if only one line is edited, but the whole document is “green” in the commit stage. In R Studio go to Tools - > Project Options -> Line ending conversion and select Posix (LF).

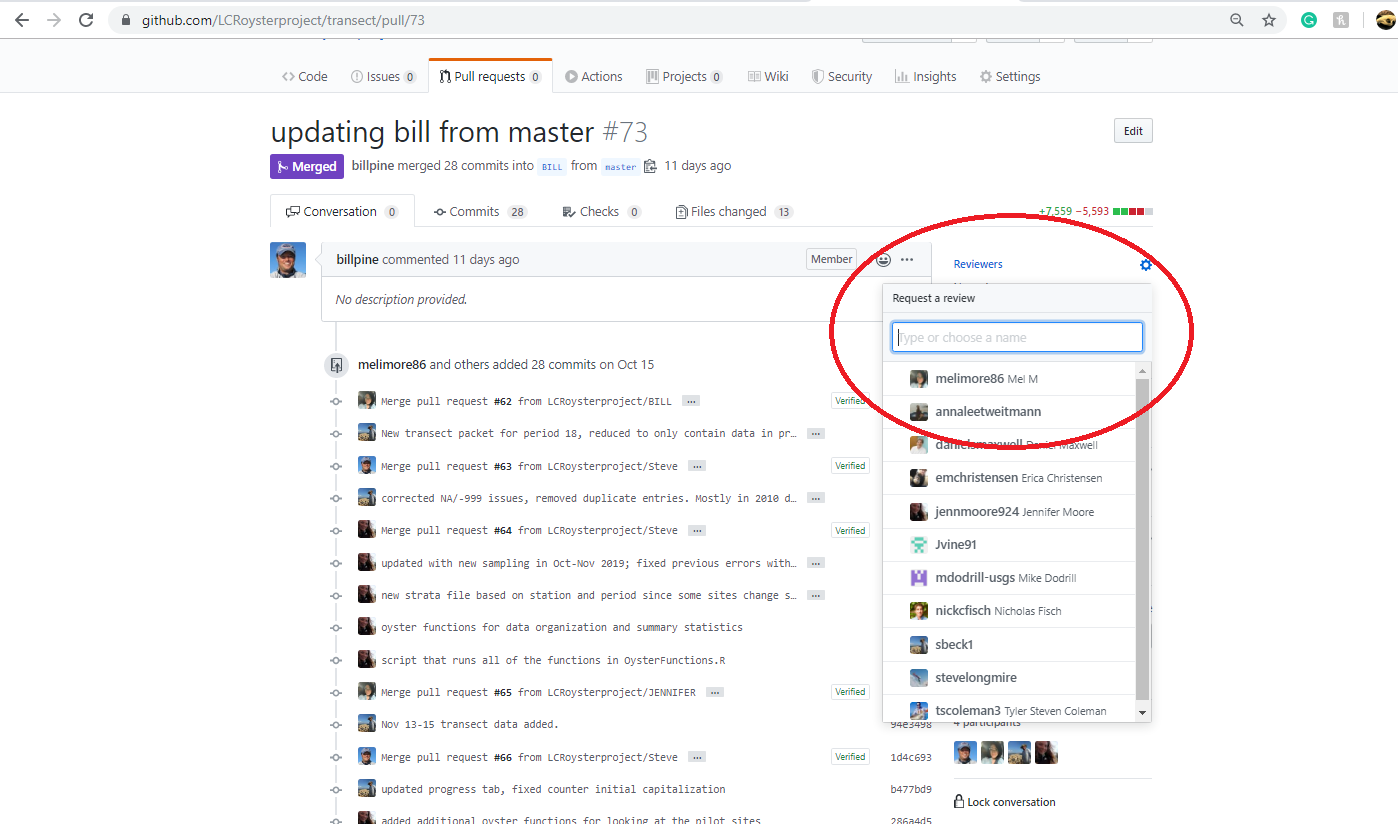


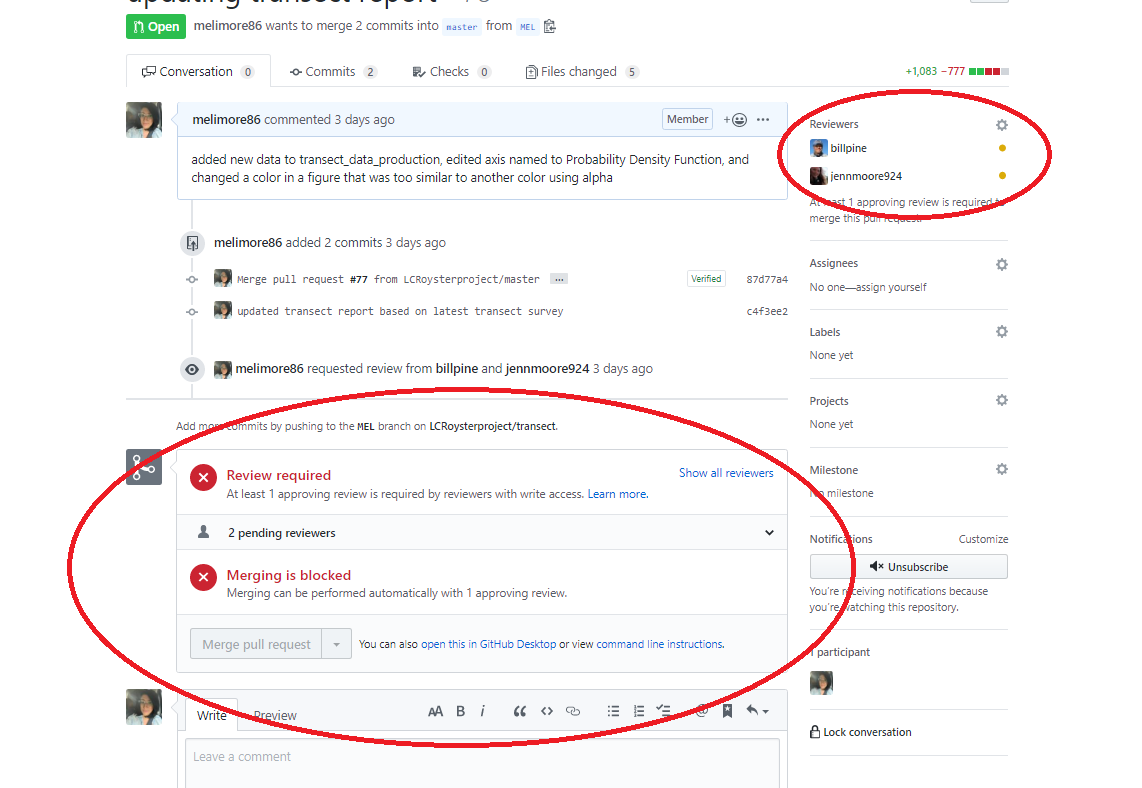
Step 4. Communicating with the team is essential for the user to know if their work should update the `master` branch. To submit a pull request from the user branch to the `master` branch, click on the **Pull Request** tab and click on the button **New pull request**.



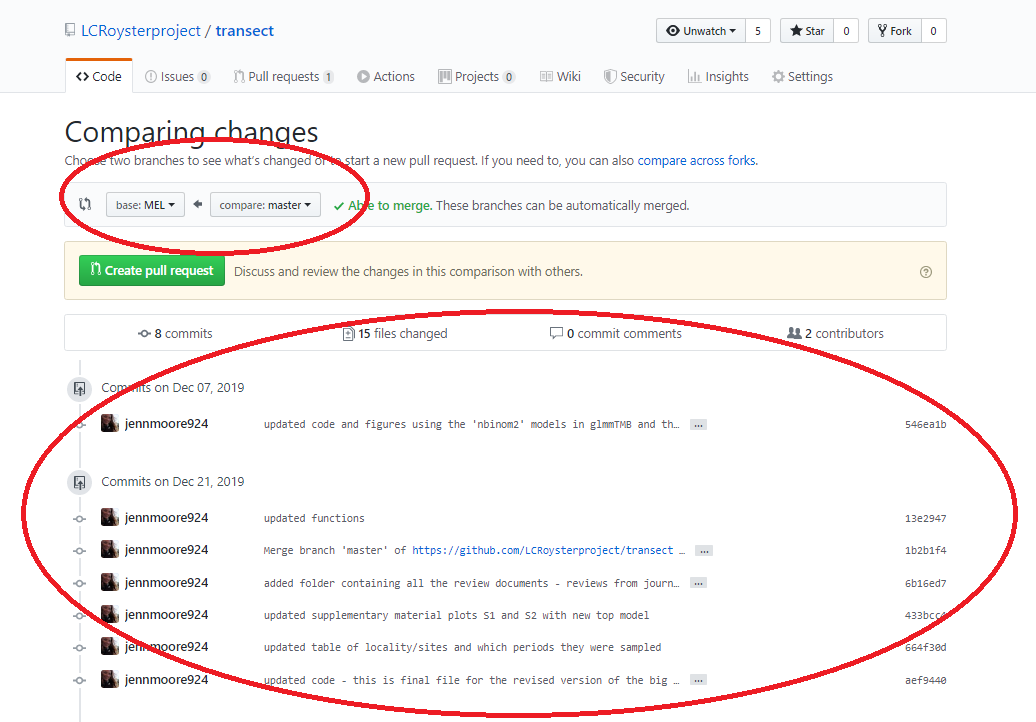
Screen shot – Location of New pull request button.



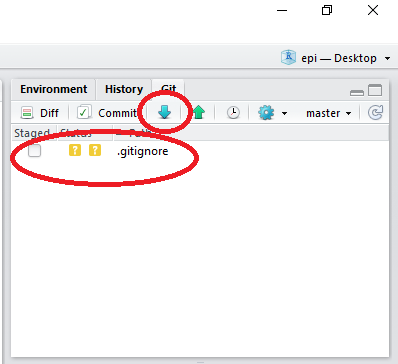
Step 5. Admins, selected in the Reviewer drop down box, will approve or deny all pull requests. The Admin can choose to approve the updates, deny the pull request, or request for additional changes before the pull request is approved.

Screen shot- Select the reviewers for the pull request.

Screen shot- Once selected a similar screen will appear with the Reviewers identified and a block to merging

Step 6. All users are required to update their branch prior to starting any work. A pull request from the master -> user branch is needed. Note that commits that have not been applied to the individual user branch are listed below the pull request. There are no blocks or reviewers needed for this kind of pull request.

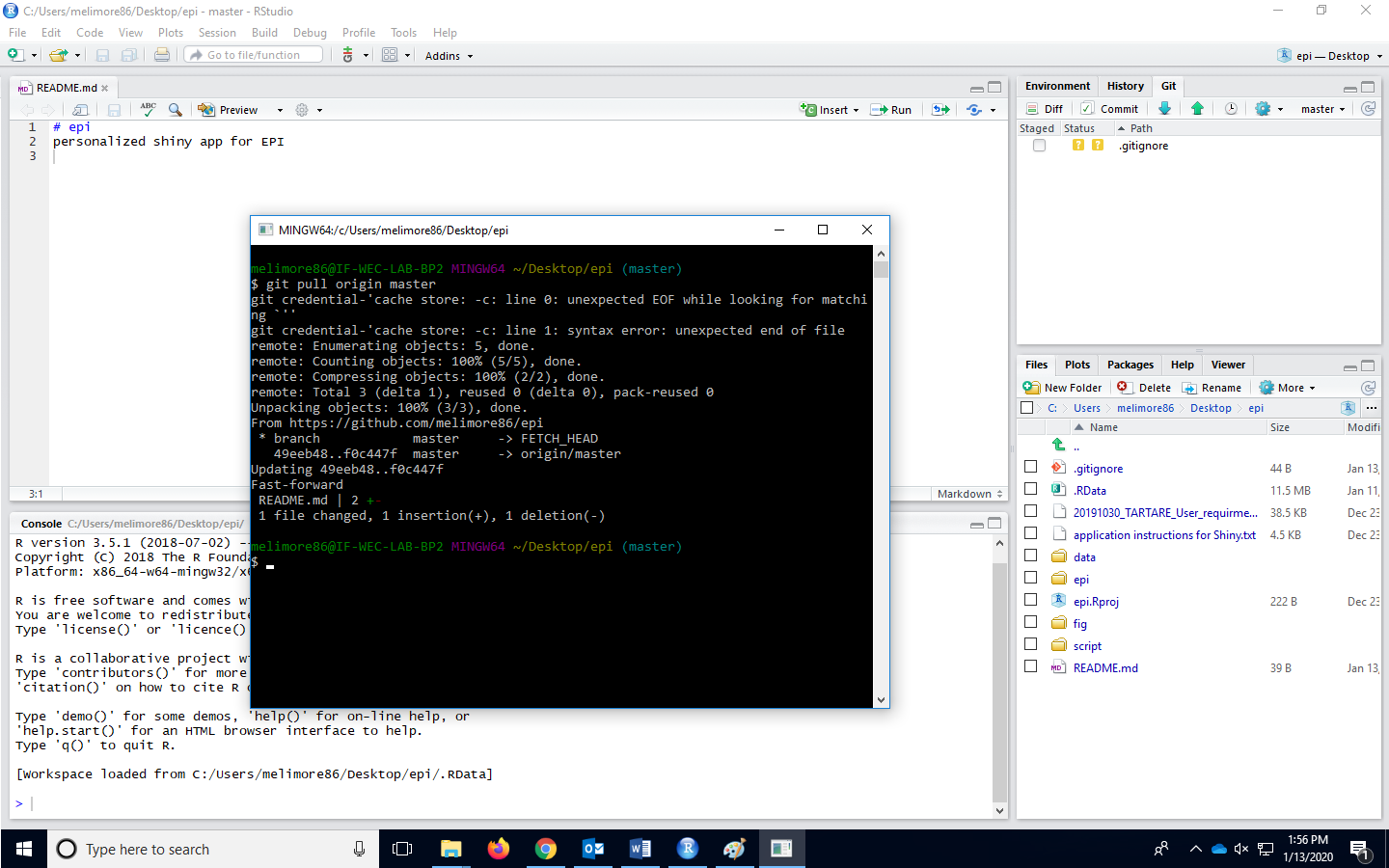
Step 6. R Studio is able to retrieve the updated files with a pull request in the Git tab. When files on the GitHub Online portal are updated with the newest version, these changes should appear on the Git tab of a folder that is connected to GitHub. Normally, with an R project, a user can switch back and forth between R projects to check if there is a need to do a pull request in the Git tab (example below).



A user can also select to do a pull request from the Shell terminal inside R Studio. It will depend on the user on which method works for them.

Click on Tools -> Shell, and a bash terminal will appear. The command for a pull request is:

*git pull origin branch.name*





1. Simba has recently made some edits to his Research Proposal on the effects of eating too many grubs. He is comfortable using GitHub and R Studio so he is using a combination of workflows.

2. Simba drag his document into the GitHub portal, where GitHub will automatically know if the file exists within the repository of not. If it does exist it will add any new additions to the document and if it does not exists it will create a new file.

3. Simba struggles to keep on a human keyboard but manages to type a descriptive commit message, and update his branch.

4. Simba goes for a hunt and comes back to make some edits on R Studio to his document. Before he make his edits he will check in the status of the files in R Studio to make sure everything is up to date. He checks this in the Tools-> Shell terminal using the command:

*git status*

5. He notices that his files might not all be up to date because the status comes back as being behind the master branch. He makes a pull request from the master -> Simba (branch).

6. The cycle repeats when Simba decides to drag the file to update the GitHub portal.

# References & Resources

What is GitHub? <https://www.howtogeek.com/180167/htg-explains-what-is-GitHub-and-what-do-geeks-use-it-for/>)

Beginners guide to GitHub:

<https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>

Beginners guide to version control:

<https://www.atlassian.com/git/tutorials>

What is a repository?  ([https://www.howtogeek.com/180167/htg-explains-what-is-GitHub-and-what-do-geeks-use-it-for/](https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/) )

Git Bash Commands:

<https://sklise.com/2012/09/22/introduction-to-git/>

GitHub Workflow:

([https://guides.GitHub.com/introduction/flow/](https://guides.github.com/introduction/flow/))

Merge Conflicts:

<https://www.git-tower.com/learn/git/ebook/en/command-line/advanced-topics/merge-conflicts> )

R Studio and GitHub:

[http://r-bio.GitHub.io/intro-git-rstudio/](http://r-bio.github.io/intro-git-rstudio/)

Video: How to integrate GitHub and R Studio <https://www.youtube.com/watch?v=E2d91v1Twcc>

Using git from RStudio:

<https://nceas.github.io/oss-lessons/version-control/4-getting-started-with-git-in-RStudio.html>

GitHub, Git, and Math 335:

<https://byuistats.github.io/M335/git_335.html>