This document comprises the GIT demonstration steps covered in workshop.

PART 1: INITIALIZING A GIT REPOSITORY, STAGING AND COMMITTING CHANGES TO THE LOCAL REPOSITORY

1. Open command prompt
2. Navigate to the folder where you want your workspace directory
3. *Git help*: to get help with any command.
4. Configure your name and email ID to be associated with your GIT account:
    a. *Git config –global user.name "<your name>"*
    b. *Git config –global user.email "<your email>"*

    If this doesn't work use PAT as mentioned on Ed

5. Create a new folder
    a. *mkdir mygitproject*
6. Move into the created folder:
    a. *cd mygitproject*
7. *Git status* – error since git is not initialized in your workspace yet
8. *Git init*: Initialize a local repository [PAUSE]
9. Create a new file readme.txt with some content
10. *Git status*: There is an **untracked** file. A new file created in your working directory is always untracked. This would have been in the **modified** state if the file was not new. The file is not **staged** yet**.**
11. *Git add readme*: to stage the new file
12. *Git status*: new file is now staged and ready to be committed
13. *Git commit -m "added readme file":* file is committed and in your local repository
14. *Git status*: nothing to commit.
15. Make some changes to the *readme* file
16. *Git status*: file is now in the **modified** state
17. *git diff HEAD readme* : to get the difference between the modified file in the workspace and the commited file in the local repository
18. We can use *Git add* and *git commit* OR *git commit -a* to commit the changes to the local repository
    a. *Git commit -a -m "Added content to readme file"*

19. *Git log*: all different commits, version history, and commit messages are displayed. [PAUSE]

PART 2: CREATING A NEW REMOTE REPOSITORY ON GITHUB.COM AND CLONING THAT IN THE WORKSPACE, GOING BACK AND FORTH

1. Own a Github.com account, GitHub Pro account free → mandatory
2. Cd ..
3. Make sure the new repository in github.com is private, Add a readme, create repository
4. *Git clone https://github.com/Nimisha-Roy/<the remote repository>: myproject2*: to download remote repository to workspace with a new folder name using HTTPS protocol [pause]
5. *Cd myproject2*
6. *Echo created a newfile > newfile*: create a new file called newfile
7. *Git add newfile*: stage it
8. *Git commit -m "added new file" :* commit it to local repository
9. *Git push*: push it to the remote repository
10. If someone else made changes to remote and you wanted working directory to reflect those changes → Git pull

The typical user scenario for this will be that each user will have their local copy, work on some local file, commit them and push them to a remote repository where others can get changes, do further changes, push them, and so on and so forth.

PART 3: CREATING AND MERGING BRANCHES

Branching means making a copy of the current project so that we can work on that copy independently from the other copies, be it other branches or the main branch. Then we can decide whether we want to keep both branches or merge them at some point. This is particularly useful because if you think about how we generally develop software, we work with artifacts. For example, we might need to create a separate copy of your work space to do some experiments. You want to change something in the code; you are not sure it will work out, and you do not want to touch your main copy (main branch). So that is the perfect application for

branching. If you are happy with the changes, you will merge that branch with the original one; If you are not happy with the changes, you will delete that branch.

Git branch: to see which branches are present. (Until this point of the demonstration, we only had one main branch)

Git branch newBranch : to create a new branch

Git branch: We have two branches now, with the current branch (main) as star marked

Git checkout newBranch: To switch to *newBranch*

Git checkout -b testing: To create a new branch and switch to it

Create a new file called *testfile* in *testing* branch, and push it to the remote repository

Create testfile with testing code

Git add testFile – staged state

Git commit -m "test file added" – committed state [pause]

Move to the new branch and merge *testing* branch with *main* branch since we are happy with the changes made in the *testing* branch

Git checkout main

Git merge testing: merge testing branch with main [should not work this way – PR, may have merge conflict]

Let us delete the testing branch because it is no longer of any use

Git branch -d testing

Git push

PART 4: BRANCH CONFLICTS

So, something that might happen when you merge a branch is that you might have conflicts, such as changing the same file in two different branches. Let's see an example of that.

Move to the *main* branch and change *newfile* there

Git branch : It shows we have two branches, *main* and *newBranch*

Notepad newfile : Change newfile

Git commit -a -m "new file changed in main branch"

Move to the *newBranch* branch and change *newfile* there

Git checkout newBranch : Change *newfile* again

Git commit -a -m "new file changed in newBranch"

Now, newfile is modified independently in *newBranch* and *main* branch

Move to the *main* branch and merge newBranch

Git checkout main

Git merge newBranch

Conflict message displayed since both branches have independent copies of *newfile*

How to Resolve:

Open *newfile*

You will see annotations showing the different versions in both branches. You can edit that file, decide which version to keep and which to delete, delete the annotations and save the file.

Git commit -a -m "merged version of newfile" – Git already has merged the branches.

Git push and see that main has the merged version. Newbranch has the old version. You can see newbranch in github.

Git branch -d newBranch: We have now resolved the conflict and can delete the newBranch

PART 5: GIT TAG

Git tag sprint1

git push origin sprint1 : Tag created as a snapshot of the work done until that point in time

Create some changes. Create a file called codeaftertag. Add, commit and push.

Fetch tag in your sprint demo

> *git fetch --tags*
>
> *git checkout sprint1*

you cant see codeaftertag.

This will put you in a "detached HEAD" state, showing the repository exactly as it was at the time <tagname> was created.

PART 6: CODE REVIEWS

Pull requests and code reviews play a crucial role in the collaborative development process. A pull request is a mechanism for suggesting changes to a codebase, allowing contributors to propose modifications, bug fixes, or new features. Once a pull request is submitted, it undergoes a code review, where other team members carefully examine the proposed changes. Code reviews help maintain code quality, improve consistency, and catch potential bugs or issues early on. They foster collaboration, knowledge sharing, and ensure that the codebase aligns with established standards. Through pull requests and code reviews, teams can work together effectively, producing high-quality software with fewer errors and better overall maintainability.

Setup your branch protection Rule

Always have a branch protection rule enforced in your main GitHub repository branch.

Settings → Branches → Branch Protection Rule → Require a pull request before merging

Now when you push something to the main repository, you will be prompted to create a PR (image 1 below). You can assign someone or yourself to review the PR (image 2 below). If you enable "Require approvals" in your settings, you will not be able to merge your PR yourself. This is the ideal setting in workplace.

While reviewing a PR, you can compare changes in files (image 3 below), add comments within lines of code (+ sign in image 3 below), and make a final review, either by commenting, approving changes or requesting changes (image 4 below).

Look at cs3300-demo

Add a new branch, add a new file, push it, create PR, review.

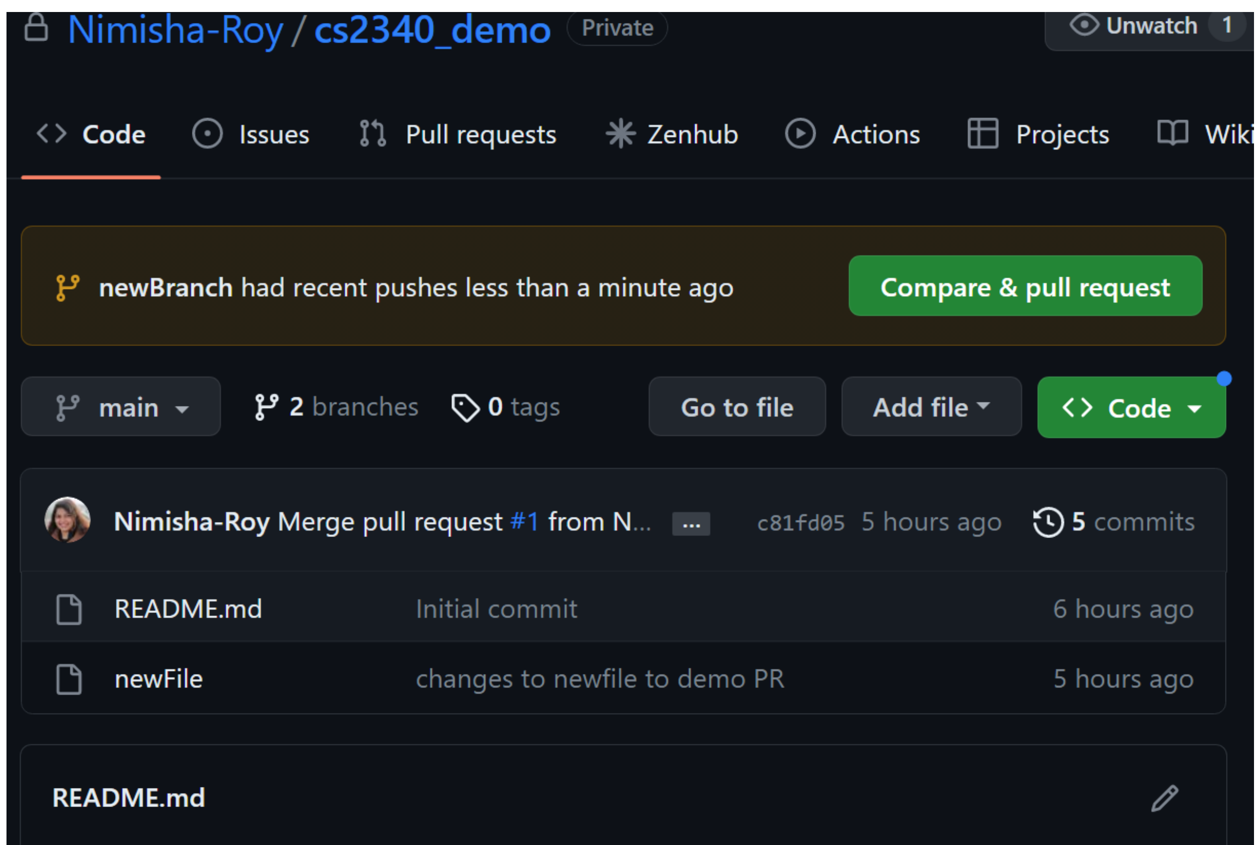

Image 1

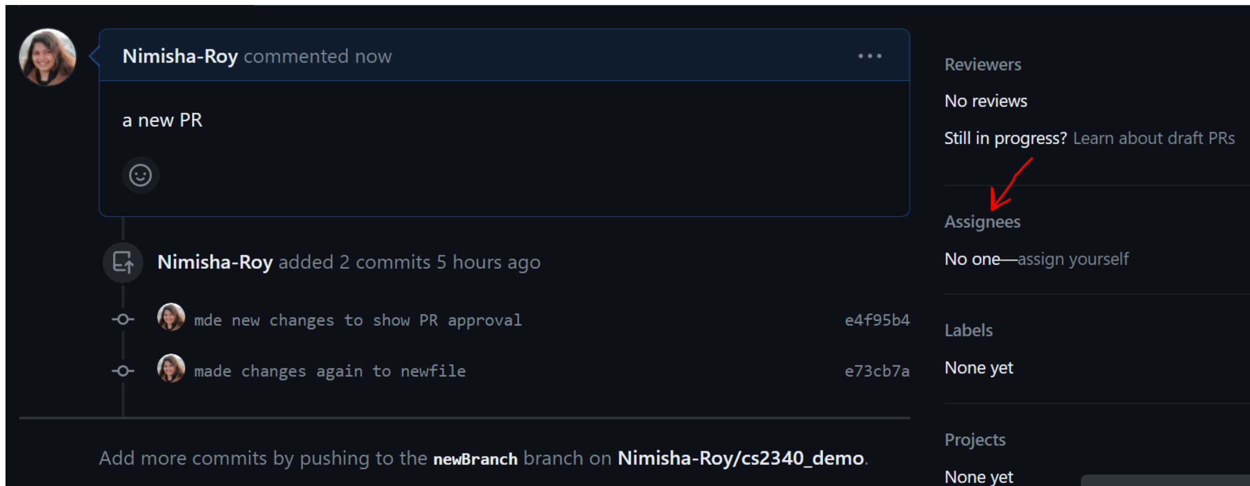

Image 2

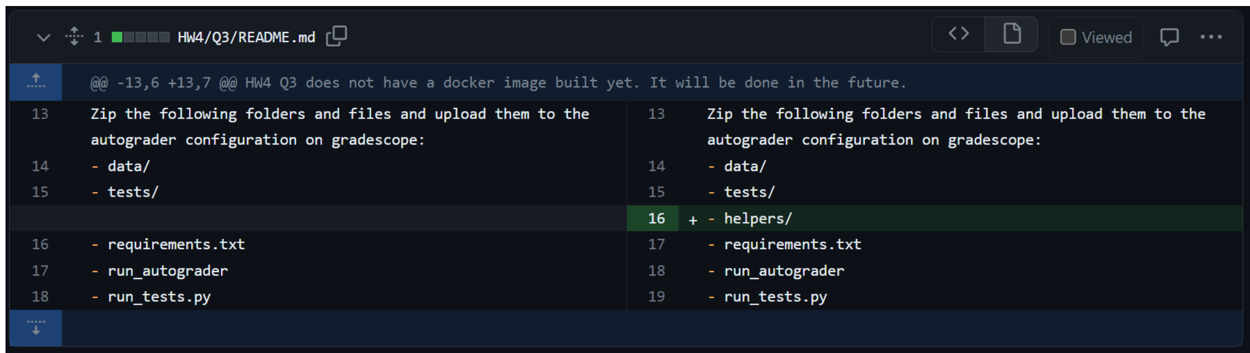

Image 3

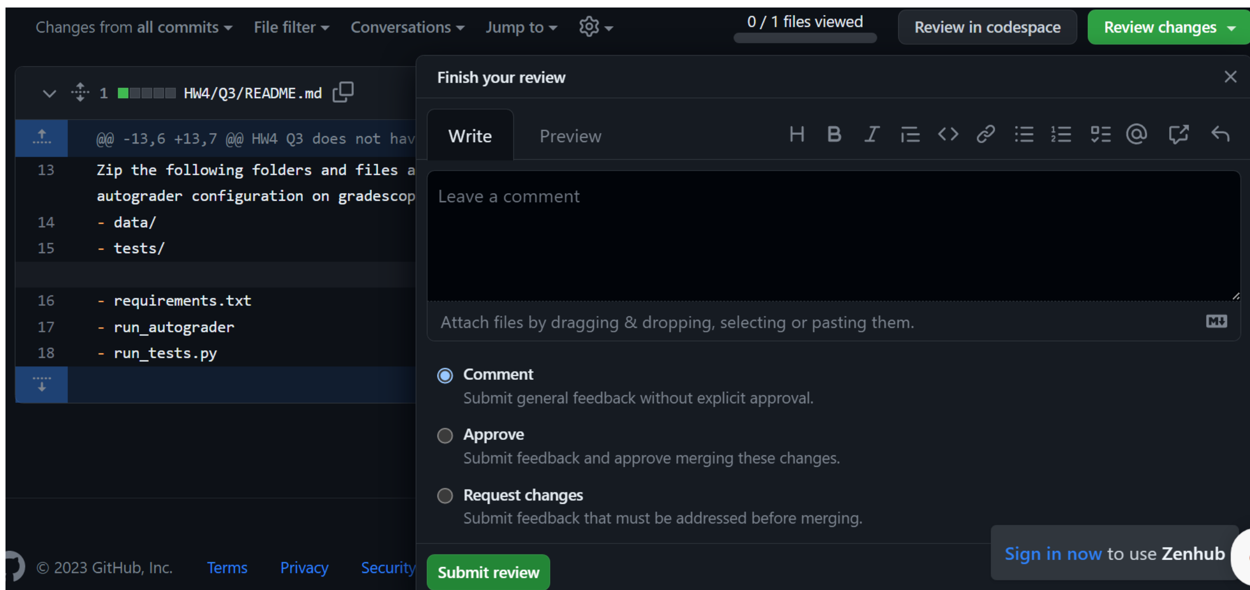

Image 4