Subject Name: **Source Code Management**

Subject Code: **22CS003**

Session: **2023-24**

Department: **DCSE**

|  |  |  |
| --- | --- | --- |
| **Submitted By:**  Ananya Aggarwal  2210990109-First Year  G2-A |  | **Submitted To:**  Dr. Amandeep Kaur |

****

**List of Programs**

|  |  |  |
| --- | --- | --- |
| **S. No** | **Program Title** | **Page No.** |
| 1 | Setting up of Git Client |  |
| 2 | Setting up GitHub Account |  |
| 3 | Generate logs |  |
| 4 | Create and visualize branches |  |
| 5 | Git life cycle description |  |
| 6 | Add collaborators on GitHub repository |  |
| 7 | Fork and commit |  |
| 8 | Merge and resolve conflicts |  |
| 9 | Git reset |  |

EXPERIMENT NO. 1:

**Aim: Setting up of Git Client**

*Theory:*

**What is Git?**

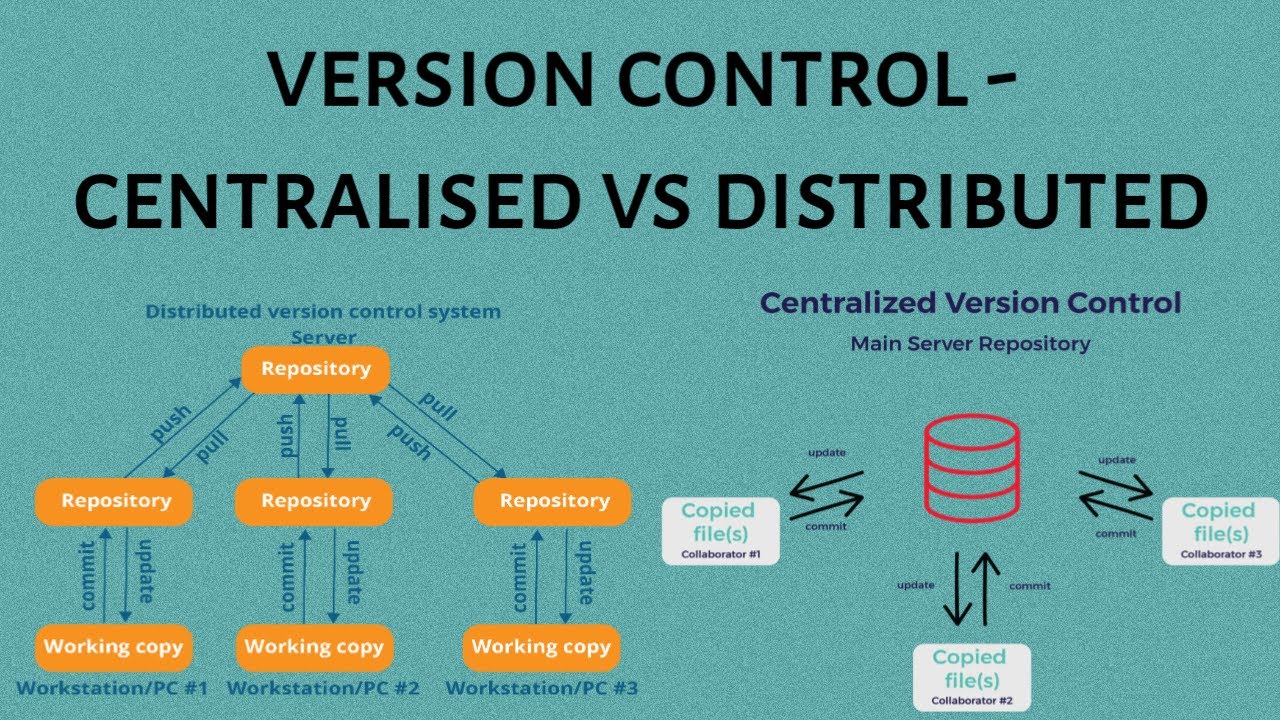
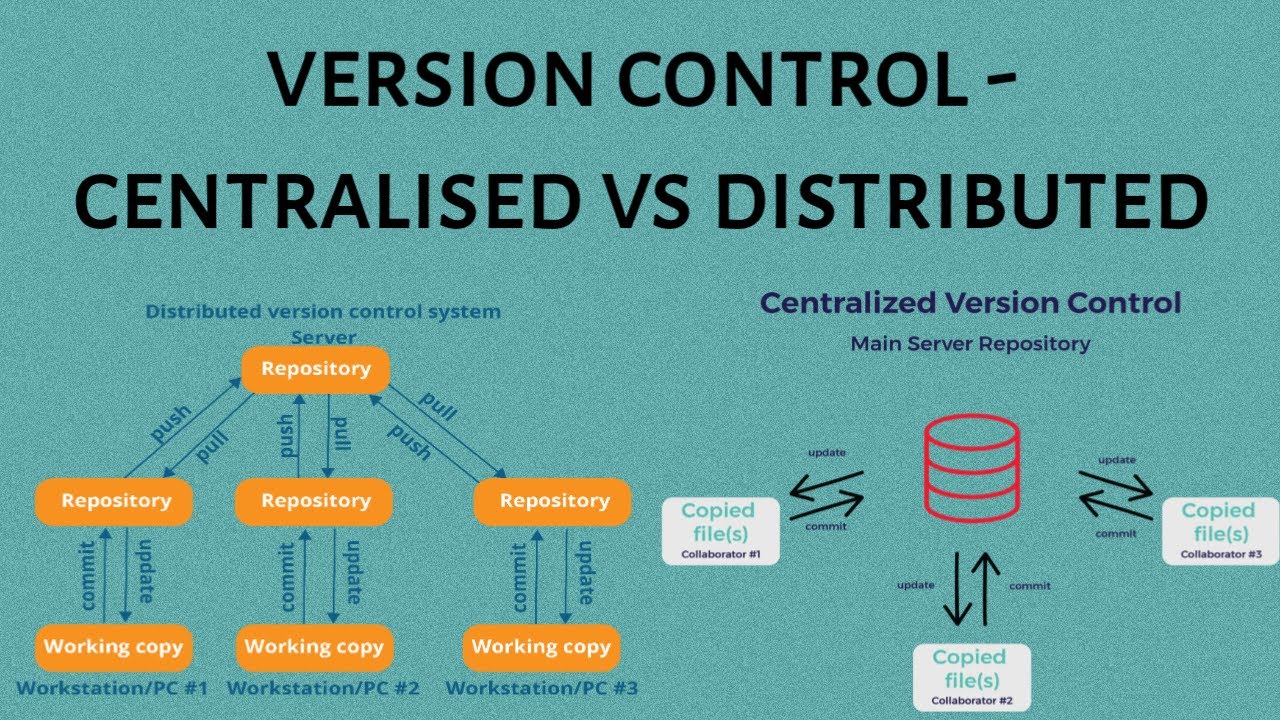
Git is a free and open-source version control system used to handle small to very large projects efficiently. This is also used for tracking changes in any set of files and usually helps in coordinating work among members of a team. Hence, enables multiple developers to work together on non-linear development.

**History of VCS:** The very first Version Control System was created in 1972 at Bell Labs where they also developed UNIX. The first one was called SCCS (Source Code Control System). It was available only for UNIX and only worked with Source Code files. Some types of Version Control Systems are:

• Local VCS: No internet is needed because it uses a database to keep and track of files.

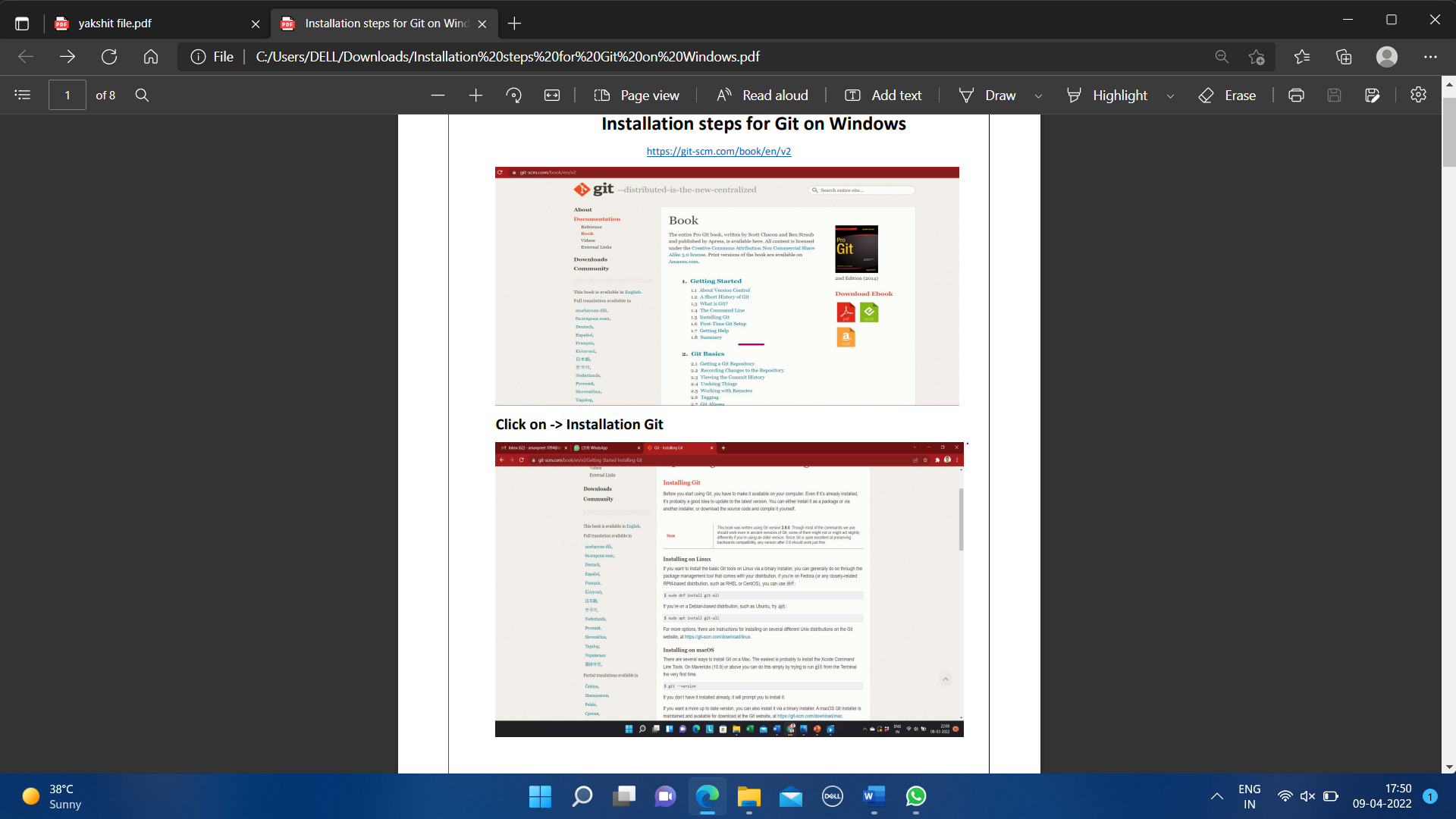
• Centralized VCS: Centralized version control systems are based on the idea that there is a single “central” copy of your project somewhere (probably on a server), and programmers will “commit” their changes to this central copy. This simply means recording the change in the central system (OS).

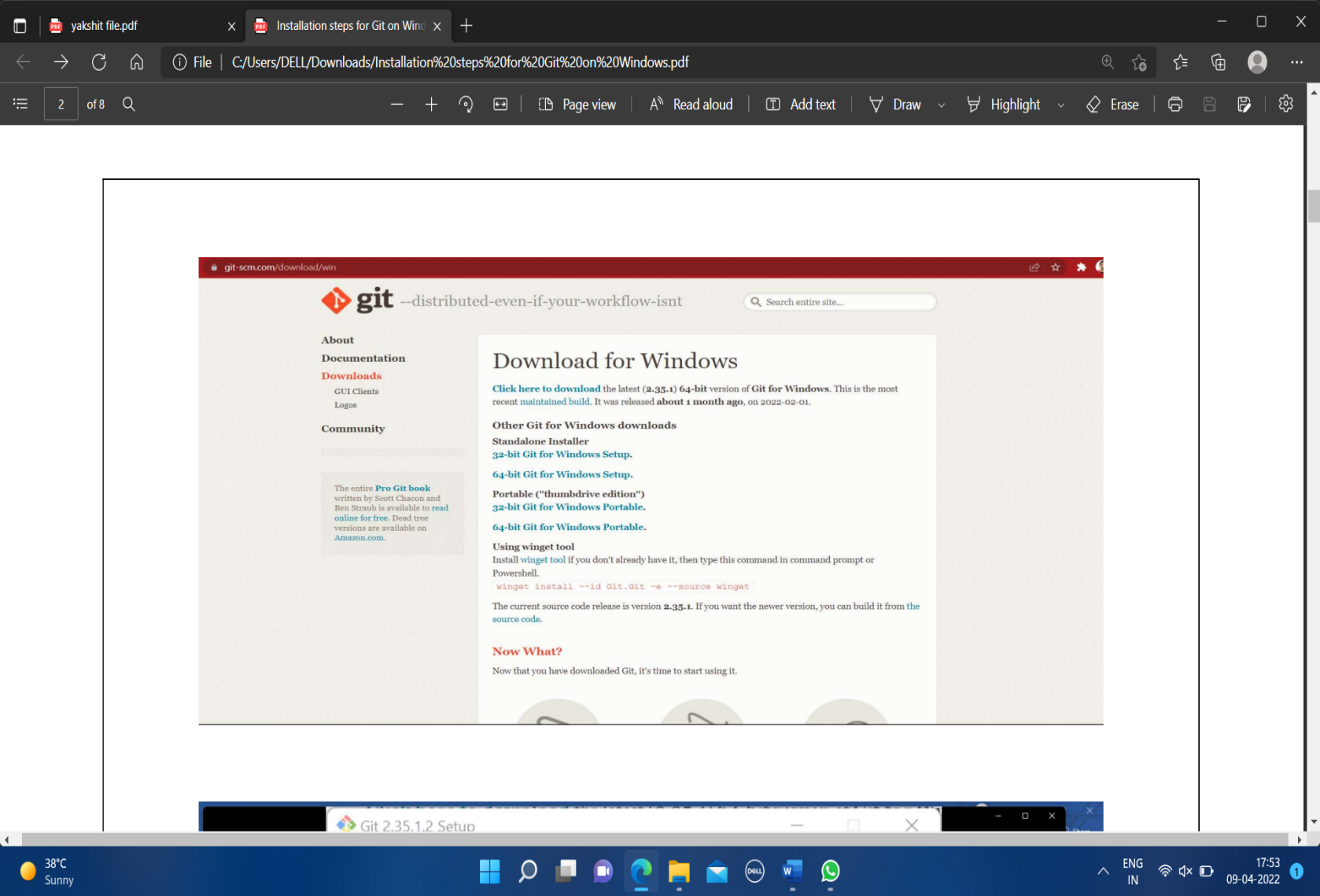
• Distributed VCS: A type of version control where the complete codebase including its full version history is mirrored on every developer's computer.

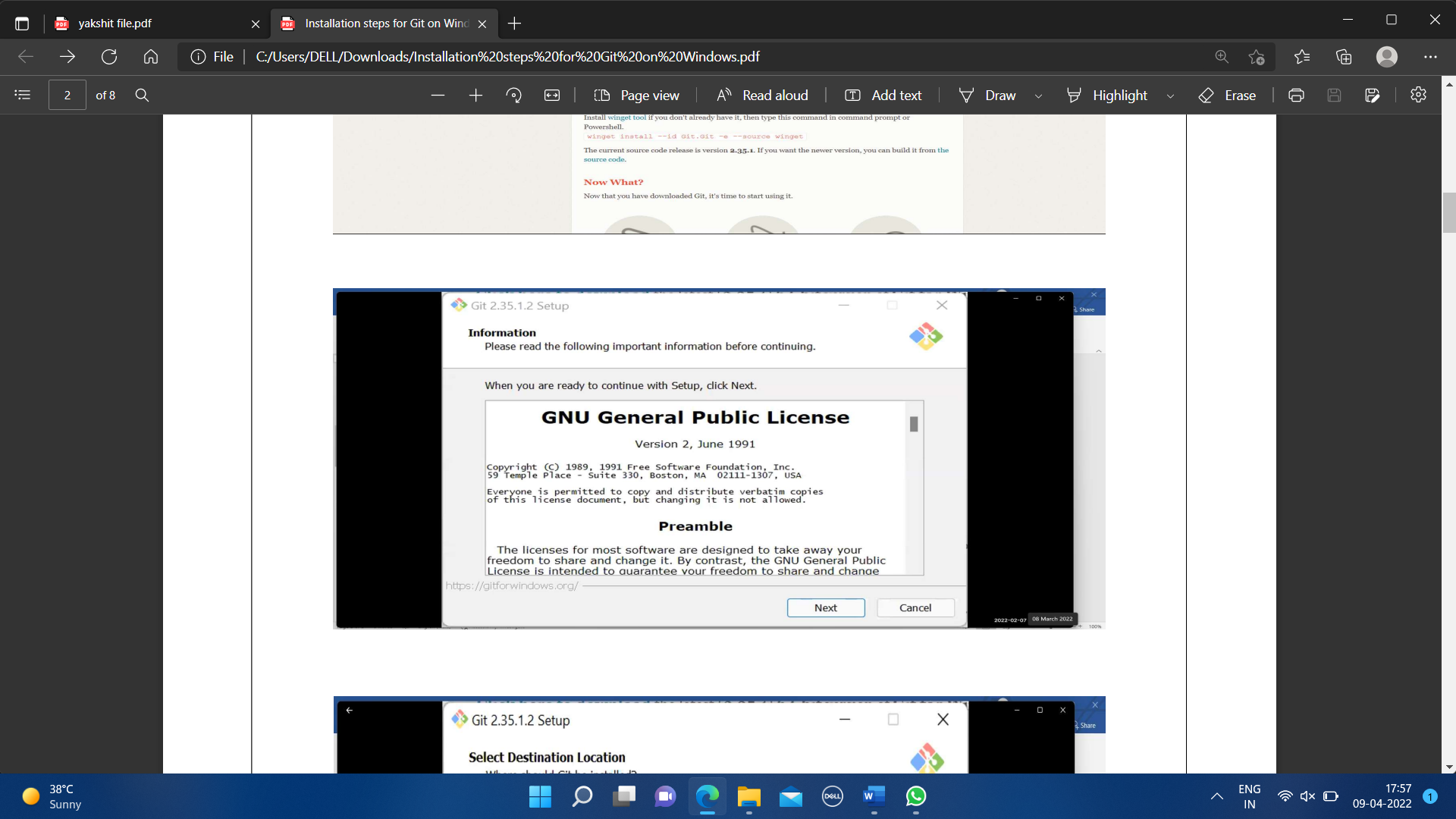
 

**How to install GIT on Windows?**

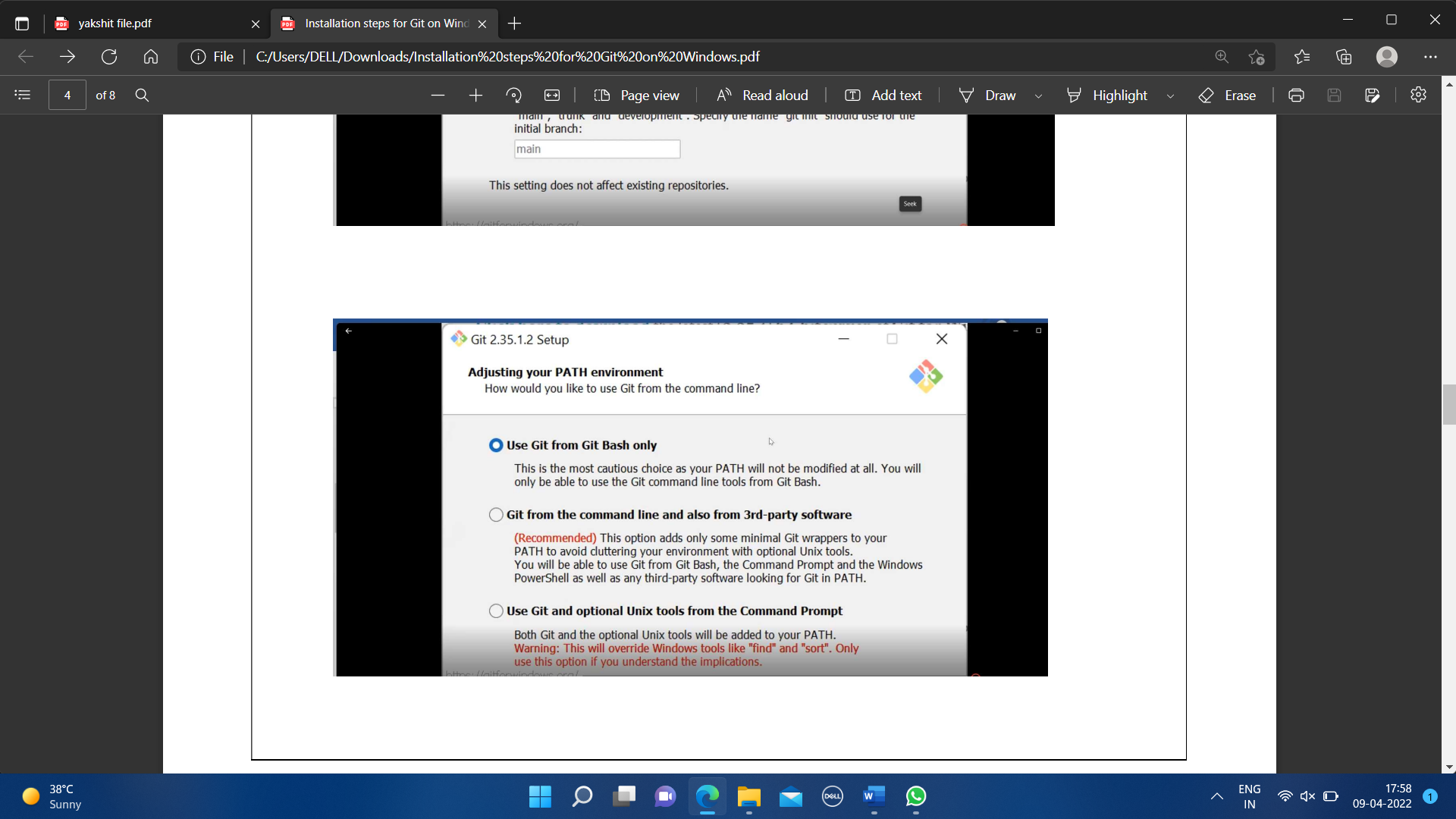
There are many ways to install Git on Windows. The most official build is available for download on the Git website. Go to https://gitscm.com/download/win and after a few settings the download will start automatically.

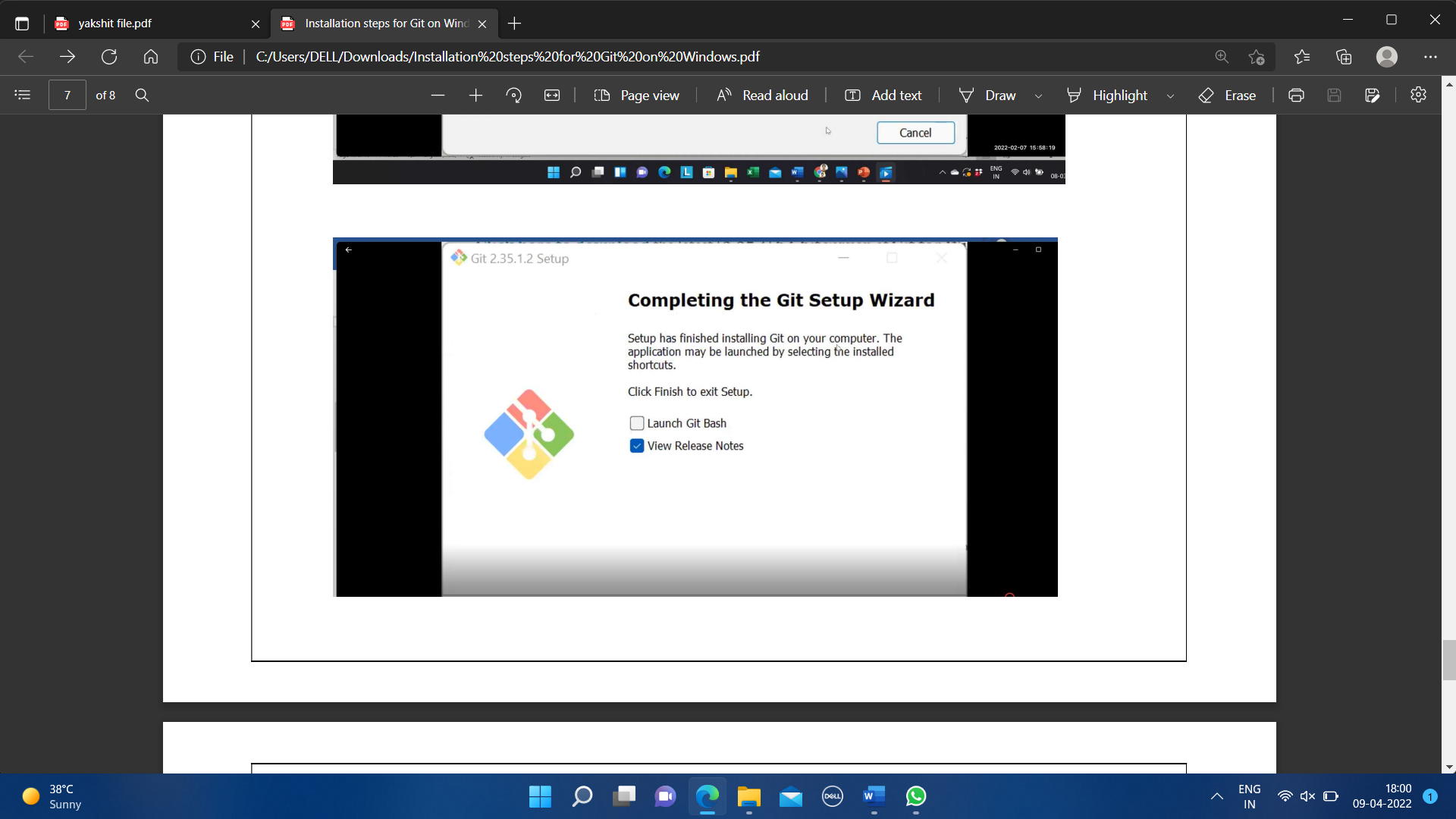
* Visit directly on git book page by <https://git-scm.com/book/en/v2> 
* Then click on Installation Git and click on whatever system you want, available are three- Windows, Apple and Linux.



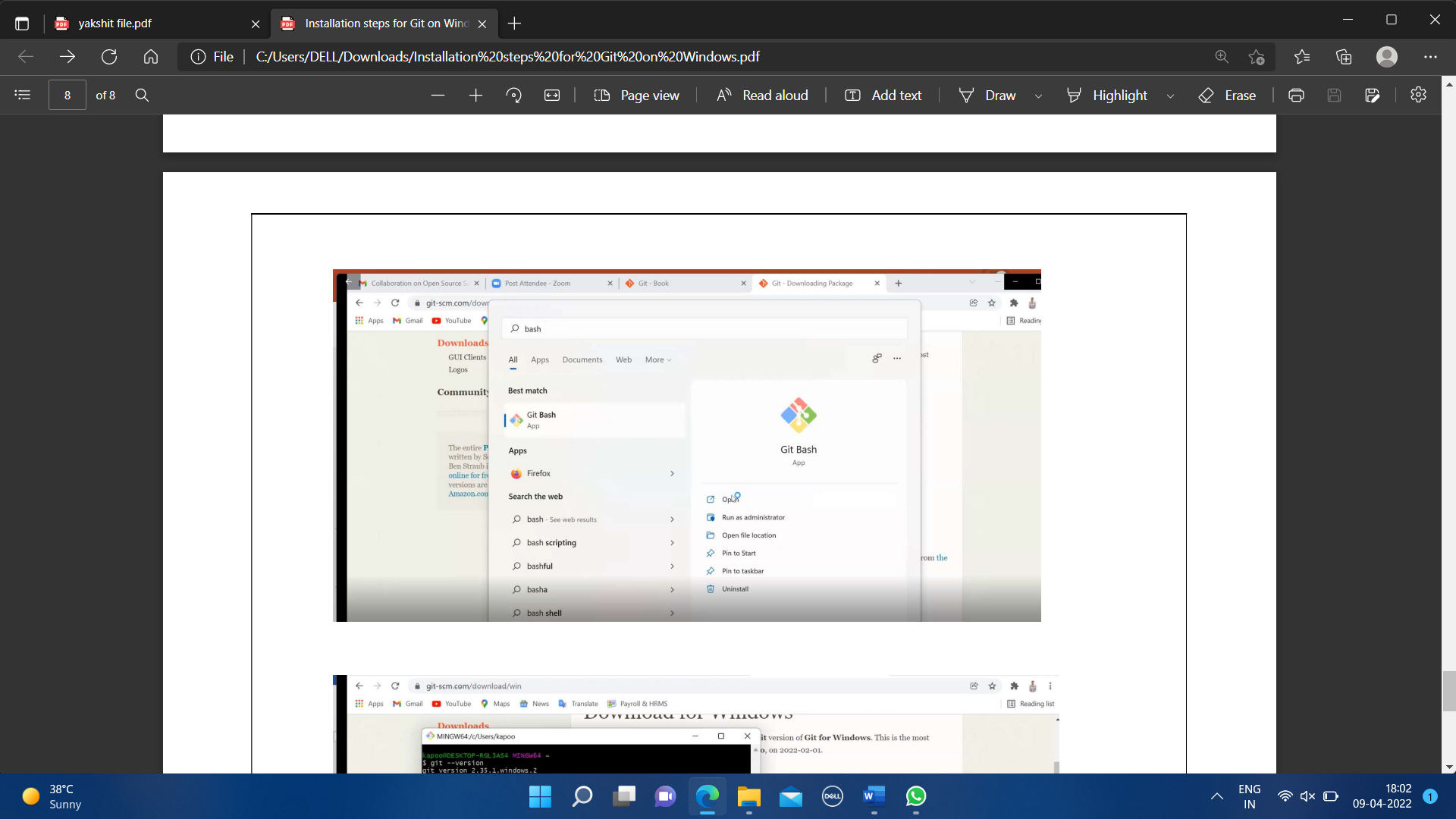


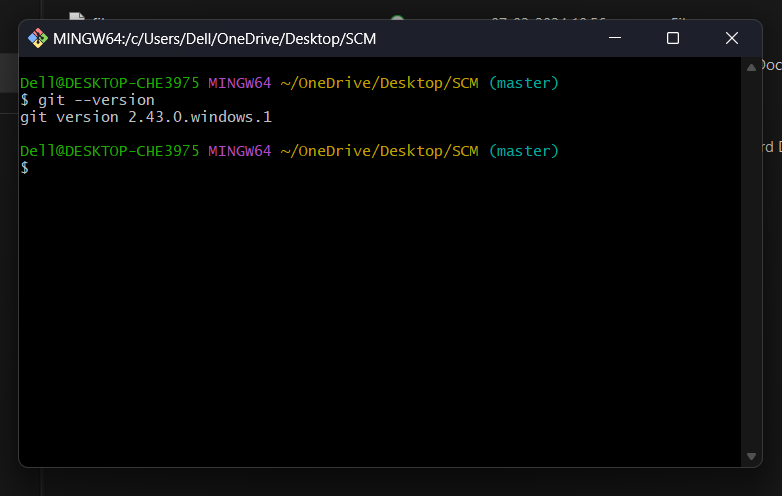
* After some more simple and easy settings and choosing your favourable environment and doing some SSH settings, it finally starts exporting the files in system and completes the Git hub wizard.





* Git bash got installed in system and seemed and opened on clicking seems of like:





You can also check the version of installed software by checking git version.

EXPERIMENT NO. 2:

**Aim: Setting up GitHub Account**

***Theory:***

**What is GitHub?**

GitHub is a code hosting platform for version control and collaboration. GitHub is a development platform inspired by the way you work. From open source to business, we can host and review code, manage projects, and build software alongside 36 million developers.

**Advantages:**

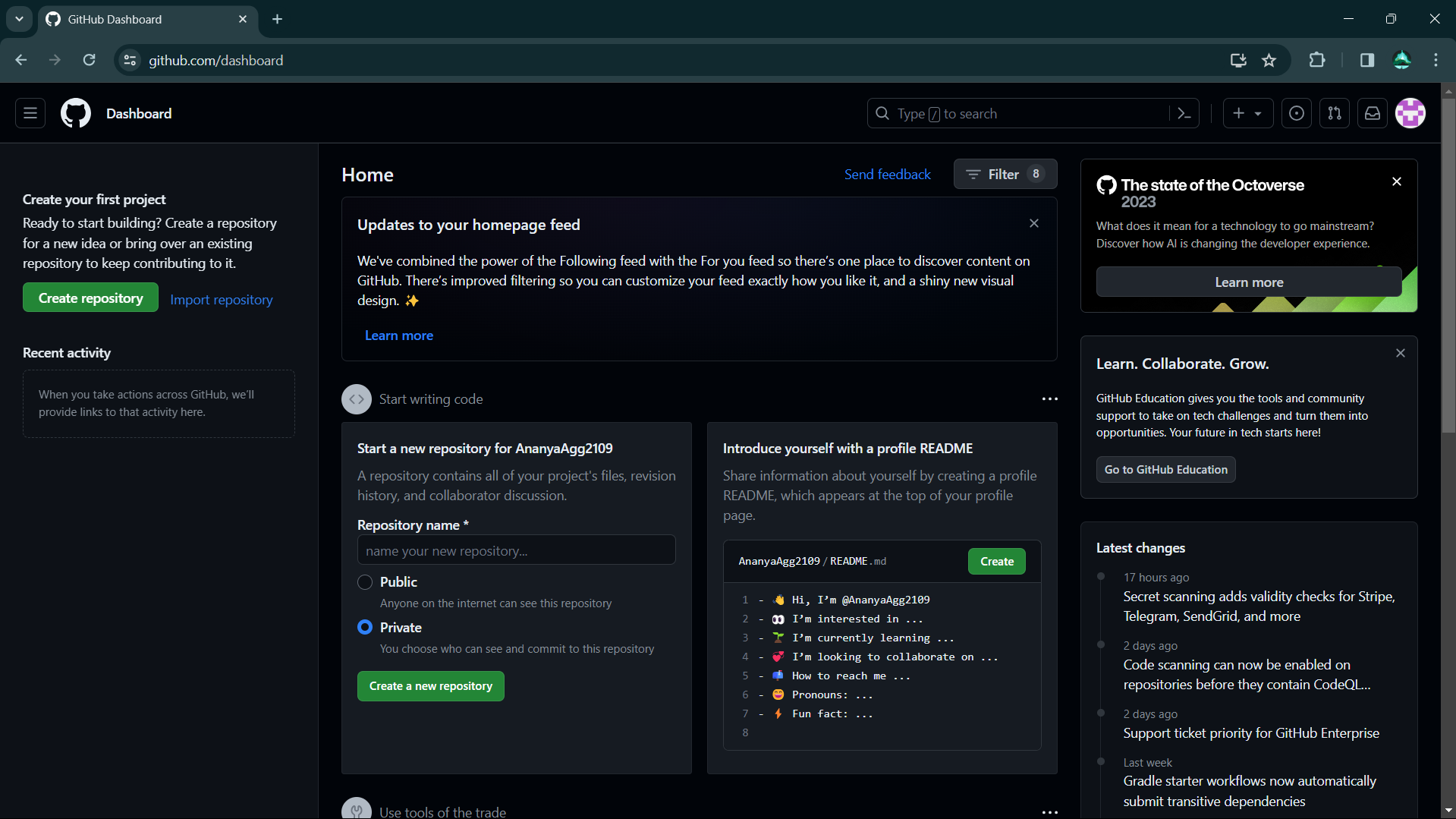
* Documentation.
* Showcase your work.
* Markdown.
* GitHub is a repository.
* Track changes in your code across versions.
* Integration options.

**Procedure:**

Search about GitHub: https://github.com/signup



By signing up for git you must remember your email and pass phases or password. For a new user, you must add your email and click on Sign up for GitHub. Otherwise click on Sign In at the top right corner.



**For linking Git Hub with Git Bash**:

Username-

git config --global user.name “username in github”

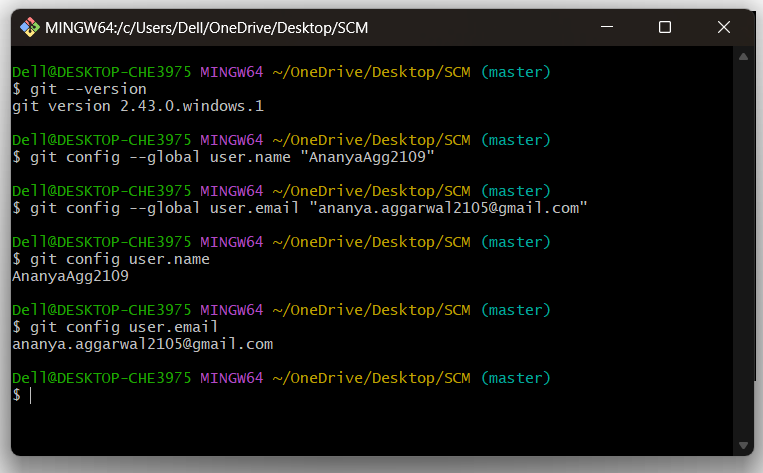
Email-

git config --global user.email “your email in github”

Check Username & Email:

git config user.name

git config user.email



**EXPERIMENT NO. 3:**

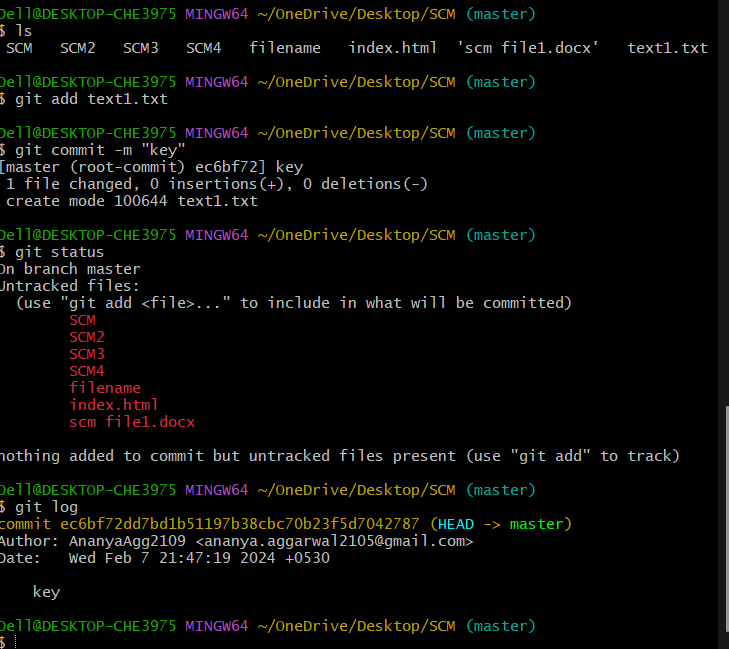
**Aim: Generate Logs on Git Hub**

***Theory:***

**Git Logs:** The git log command shows a list of all the commits made to a repository. You can see the hash of each Git commit, the message associated with each commit, and more metadata. This command is basically used for displaying the history of a repository.

**Why do we need logs?**

Git log is a utility tool to review and read a history of everything that happens to a repository. Anything we change at what time, by which log, everything is getting recorded in git logs.



You can use command **git log** to access logs(every change you make with time and date).

EXPERIMENT NO. 04

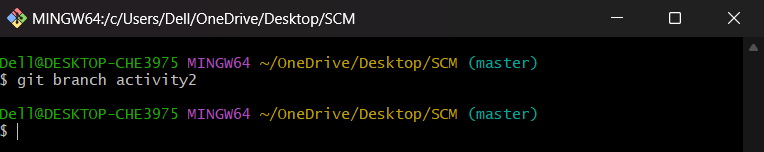
**Aim: Creating and Visualizing the Branches On Git Client**

***Theory:***

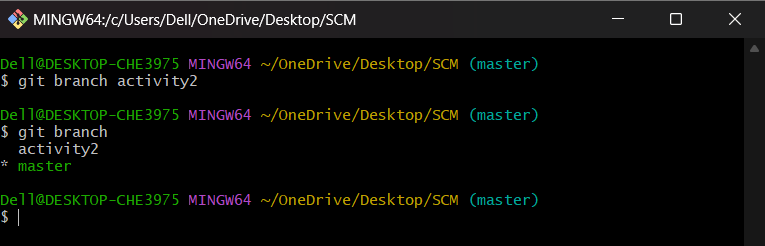
**How to create branches?**

The main branch in which we are working is master branch. you can use the “git branch” command with the branch name and the commit SHA for the new branch.

1. For creating a new branch: git branch “name of the branch”.

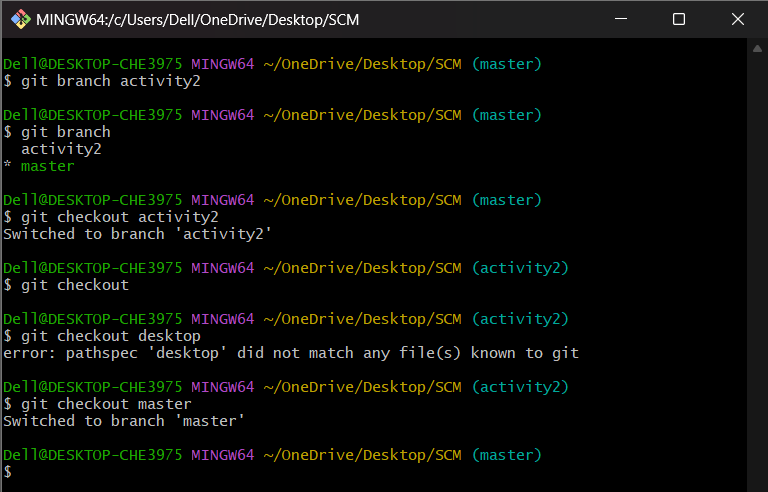


1. To check how many branches we have:



As you can see here two branches are showing that I create- aa and activity2.

1. To change the present working branch: git checkout “name of the branch” and command to go back to the master directory:



Here, you see by using checkout command we can switch branches and from branches to even master branch too.

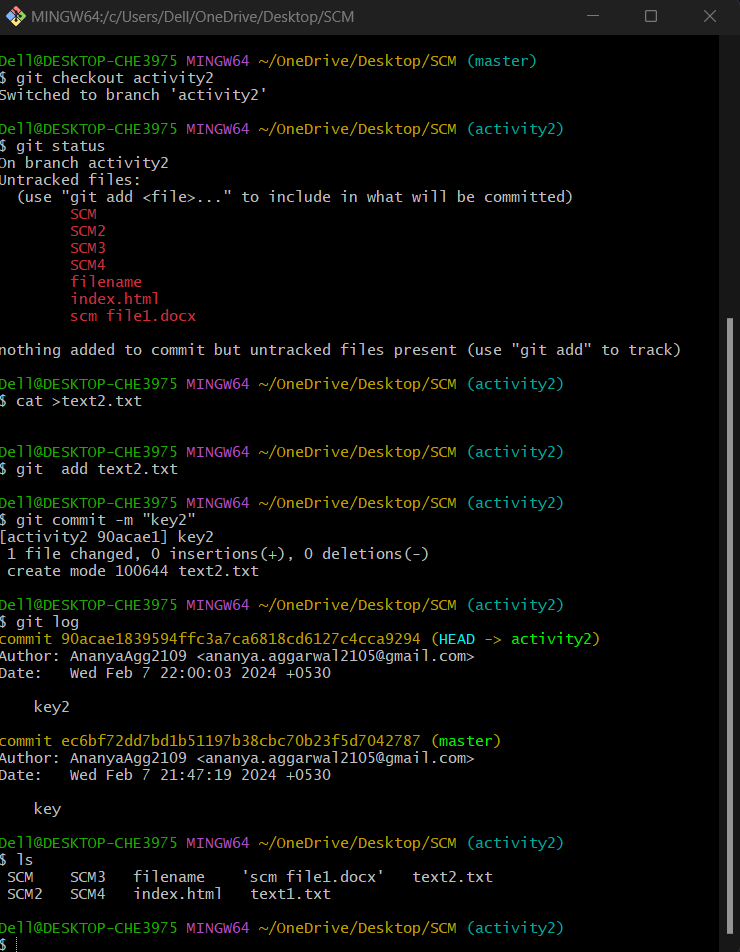
**Visualizing branches:**

for visualizing, we have to create a new file in the branch that we made “activity2” instead of the master branch. After this we have to do three step architecture that is working directory, staging area and git repository.

Firstly I’ve changed the branch from master to activity2 that I previously made and after that I check git status. Now I add text in text2 file (text2.txt) and use git add “file\_name”.

Then I use git commit -m “key\_name” command for the changes I made and insertions I do.

At last I check my activities with the help of git log command.



**EXPERIMENT NO. 05**

**Aim: Git lifecycle description**

*Theory:*

**Stages in GIT Life Cycle:** Files in a Git project have various stages like Creation, Modification,Refactoring, and Deletion and so on. Irrespective of whether this project istracked by Git or not, these phases are still prevalent. However, when a projectis under Git version control system, they are present in three major Git statesin addition to these basic ones. Here are the three Git states:

• Working directory

• Staging area

• Git directory

**Working Directory**:

When a project is residing in our local system we don’t know whether the project is tracked by Git or not. In any of the case, this project directory is called our Working directory.

**Staging Area:**

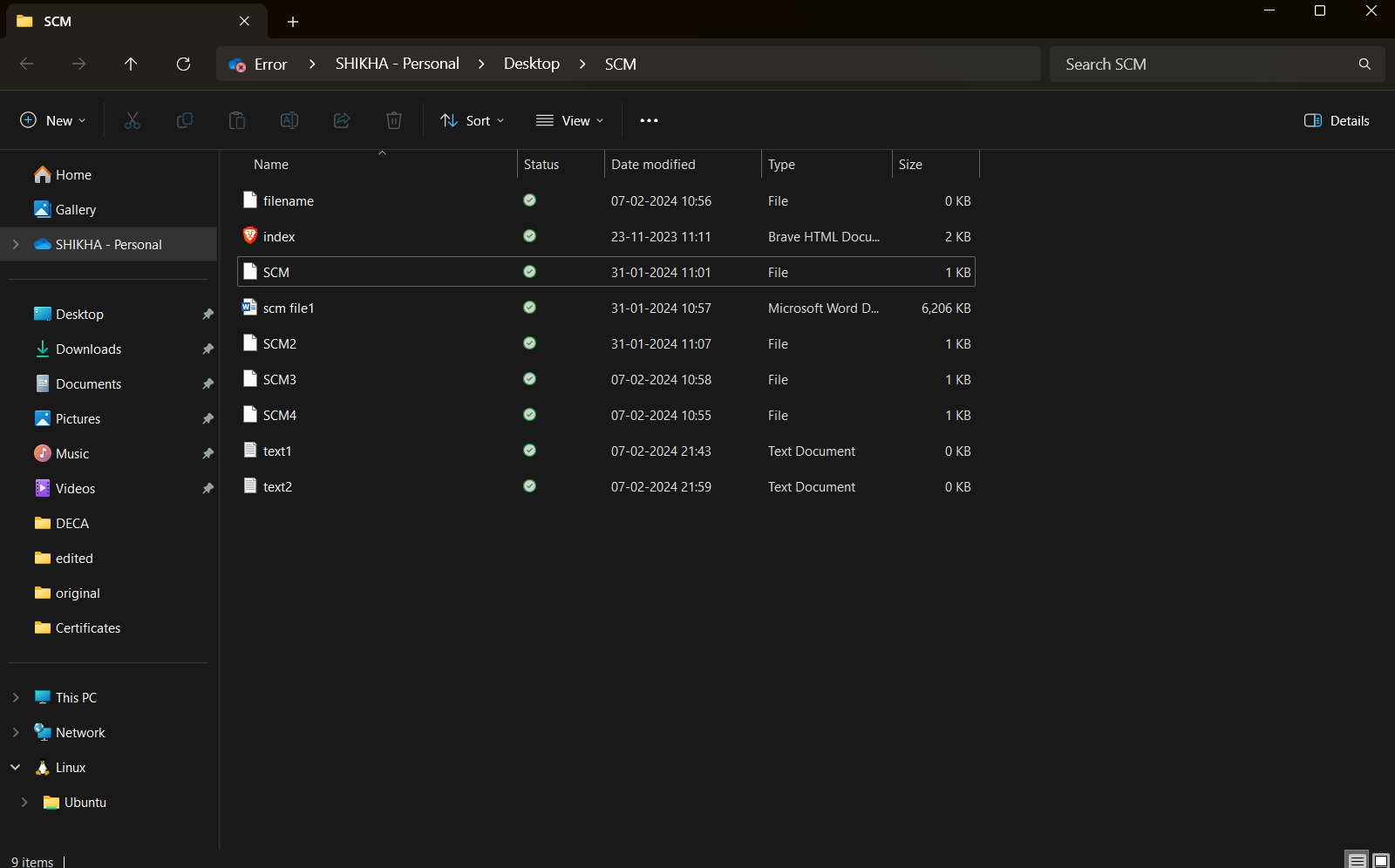
The staging area is like a rough draft space, it's where you can git add the version of a file or multiple files that you want to save in your next commit (in other words in the next version of your project)

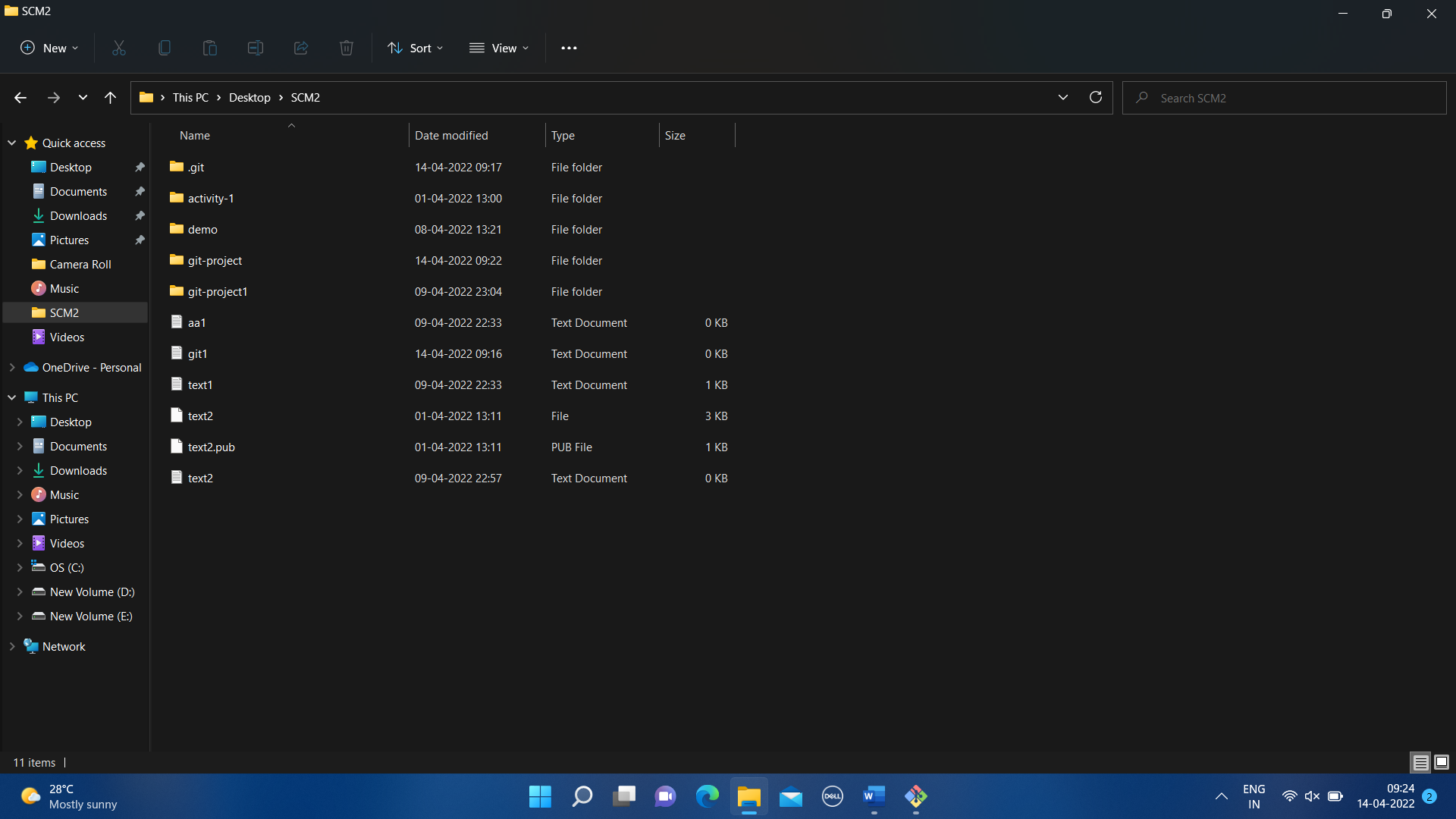
**Git Directory:**

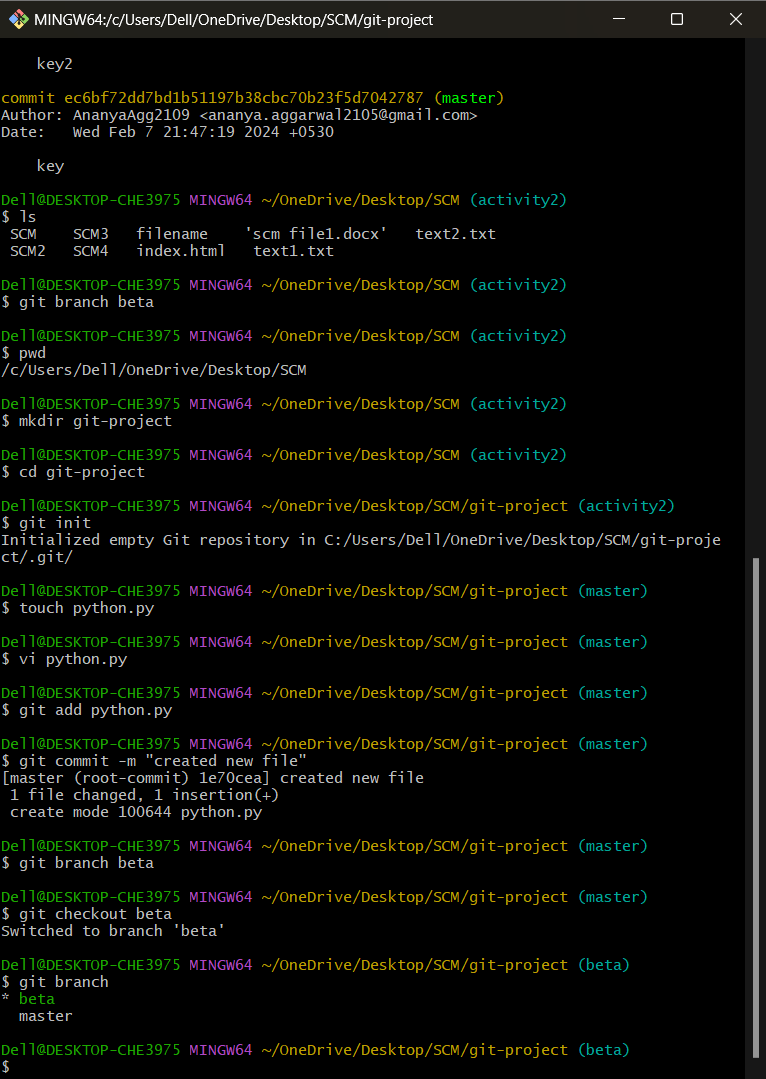
The . git folder contains all information that is necessary for the project and all information relating commits, remote repository address, etc. It also contains a log that stores the commit history. This log can help you to roll back to the desired version of the code

Remote Repository: Remote repositories are hosted on a server that is accessible for all team members - most likely on the internet or on a local network. Assessable and reachable by all.

Screenshot:







.

# EXPERIMENT 6: Add Collaborators on GitHub Repository

In GitHub, we can invite other GitHub users to become collaborators to our private repositories. We can pull the contents of the repository and push changes to the repository. You can add unlimited collaborators on public and private repositories. But, in a private repository, the owner of the repo can only grant writeaccess to the collaborators.

Inviting Collaborators to your personal repositories

Follow the steps below to invite collaborators to your own repository.

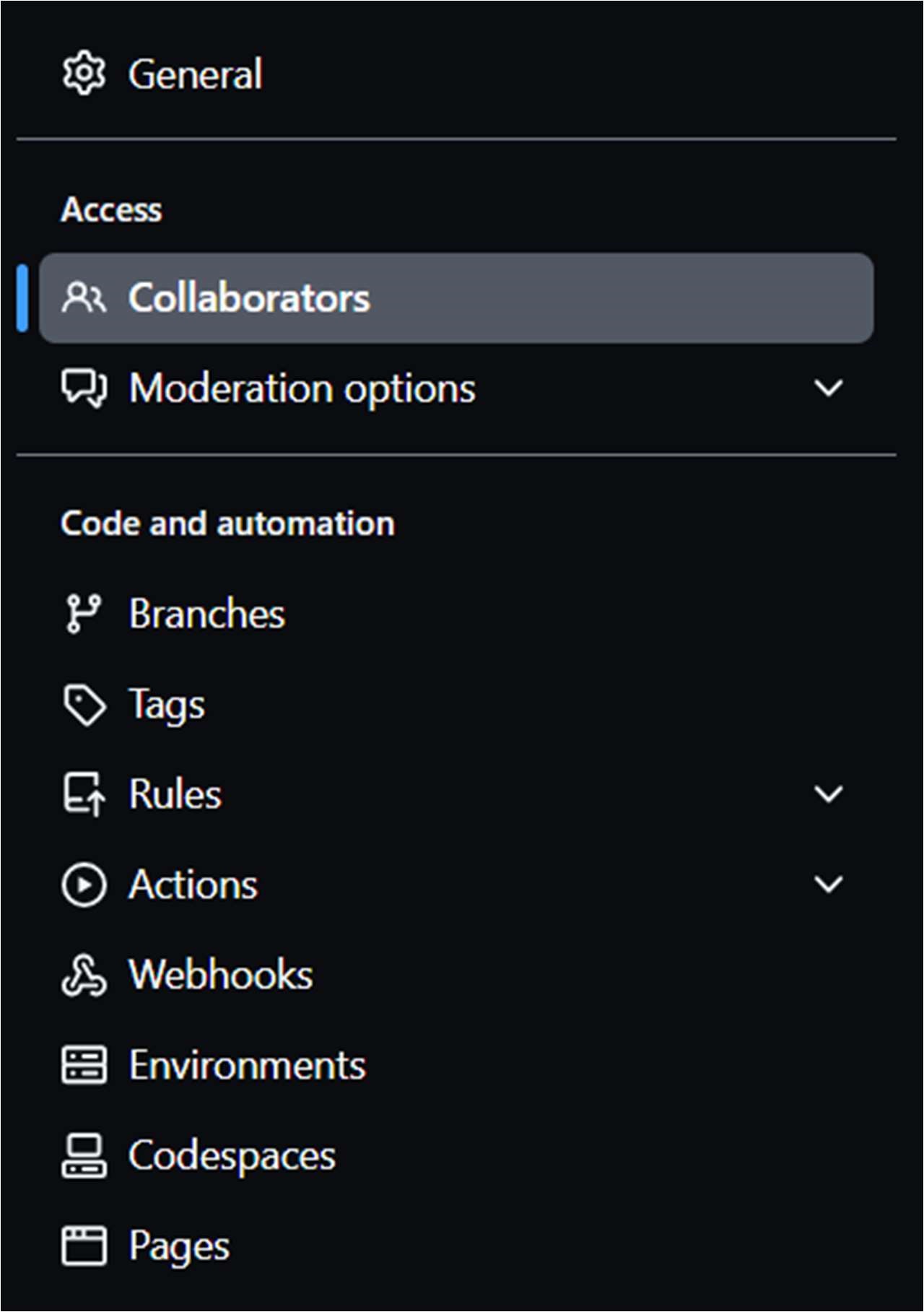
Step 1: Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

Step 2: Go to your repository.

Step 3: Click into the Settings.

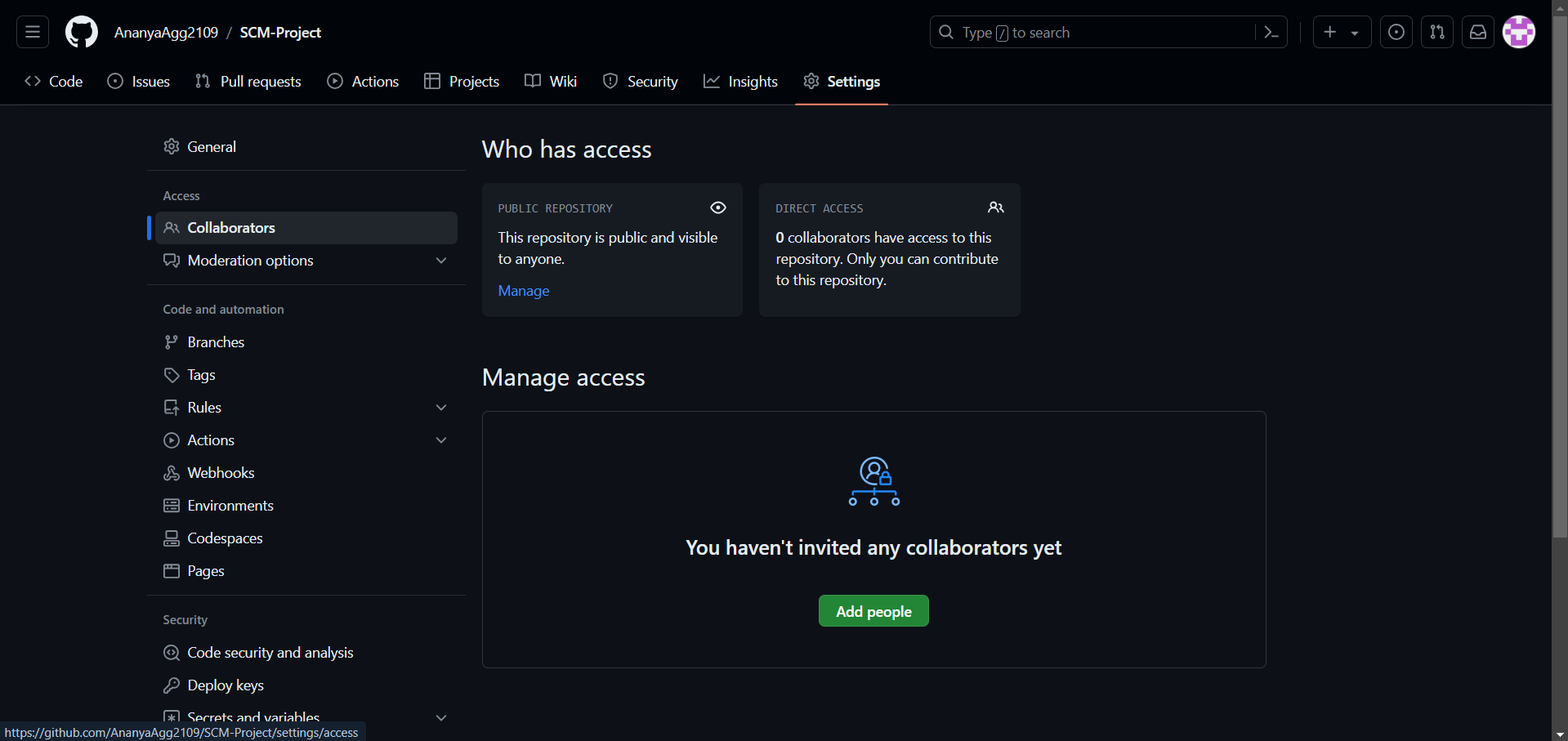


Step 4: A settings page will appear. Here, into the left-sidebar click into the Collaborators.



Step 5: Then a confirm password page may appear, enter your password for the confirmation.

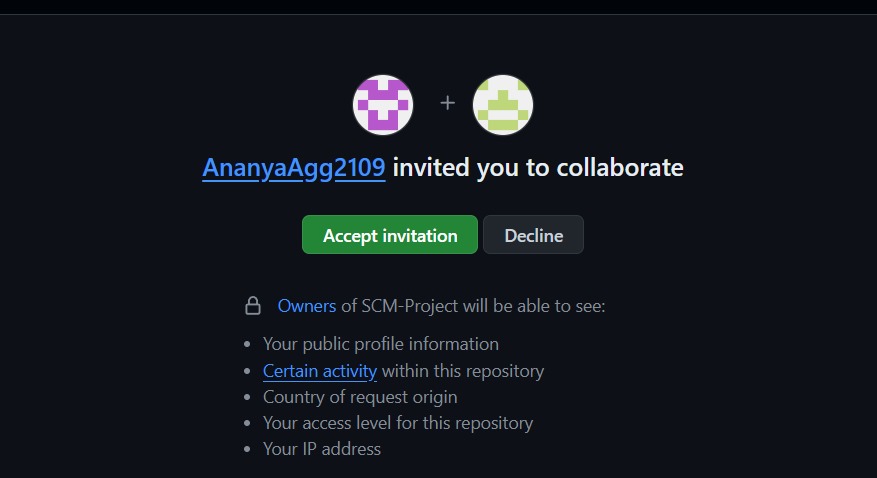
Step 6: Next, click into Add People.



Step 7: Then a search field will appear, where you can enter the username of the ones you want to add as collaborator.

Step 8: After selecting the people, add them as collaborator.

Step 9: After sending the request for collaboration to that person whom we wanted as our collaborator will get an email. He/she has to open the Email to view the invitation.



Step 10: After Clicking the View invitation, He/she will be redirected to the GitHub Page for accepting the invitation sent by the team leader.

Step 11: We are done adding a single collaborator. You can also add more collaborator and delete the existing one.

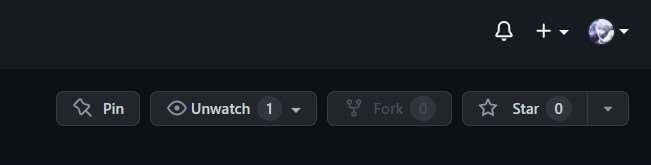
# EXPERIMENT 7: FORK AND COMMIT

What is a Fork ?

A GitHub fork is a copy of repository (repo) that sits in your account rather than the account from which you forked the data from. Once you have forked a repo, you own your forked copy. This means that you can edit the contents of your forked repository without impacting the parent repo.

Fork Workflow:

1. Fork this repo owned by Earth Lab into your GitHub account.
2. Clone the fork of your repo, so you can edit the contents locally.
3. Make edits to your local cloned copy of the repo on your computer.
4. Add, Commit and Push those edits back to your fork on GitHub.
5. Suggest the changes that you made, to be added to the Earth Lab central repo using a Pull Request.



What is COMMIT in GitHub?

Commit is like a snapshot of your repository. You should make new commits often, based around logical units of change.

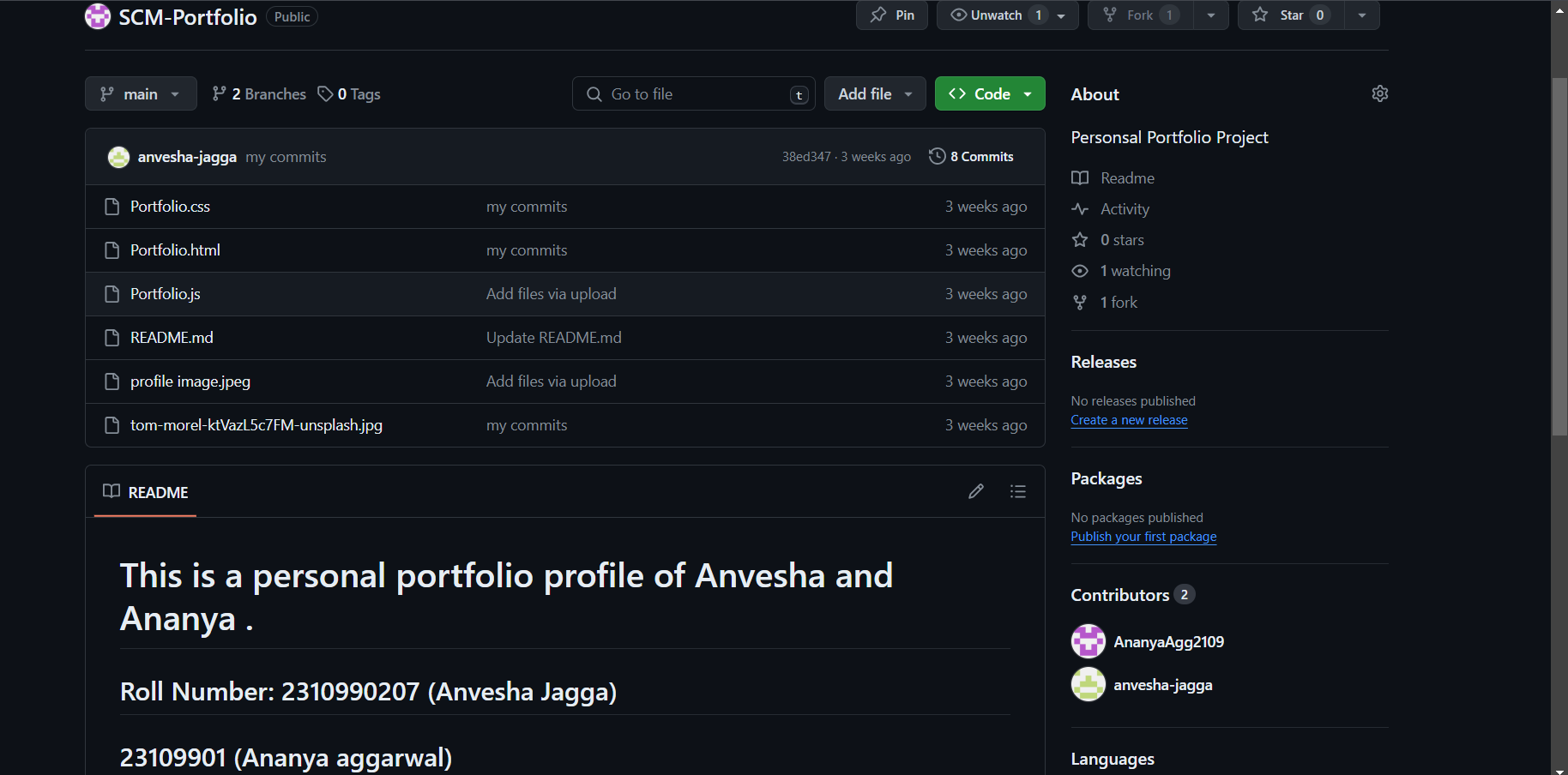
Over time, commits should tell a story of the history of your repository and how it came to be the way that it currently is.

Commits include lots of metadata in addition to the contents

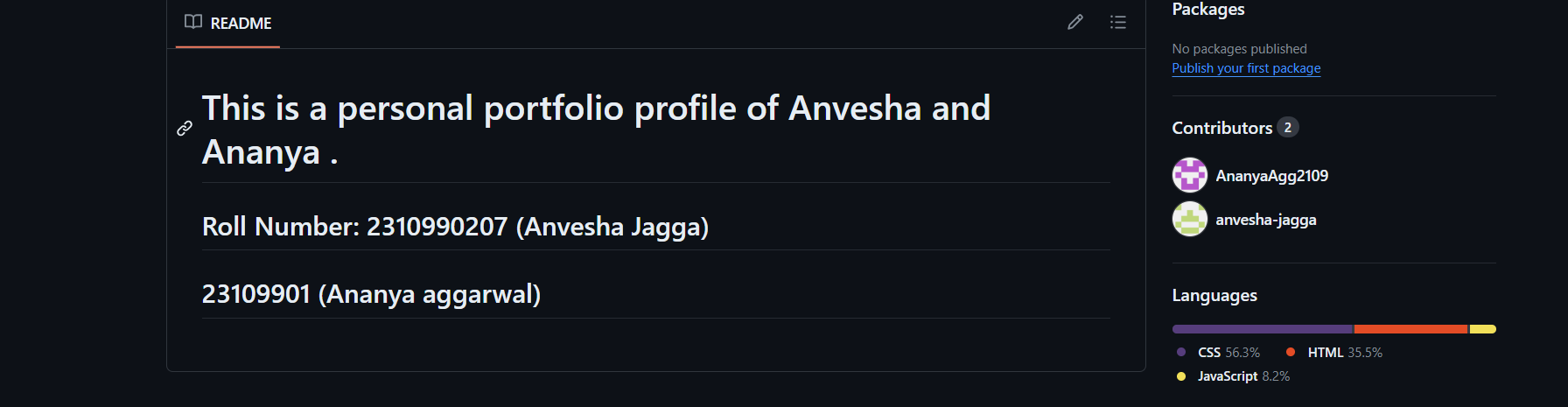
and message, like the author, timestamp and more.

Now, we will see the main repository content of the main Repository. We can see that there is a README.md file

The content in the README.md file in



the main repository, is given as below:



Now, we will see from the other collaborator’s repository. We know that this repository was forked from VanshikaArora//Task2, so the content of this repository should have same files, and we can see that it also has the README.md file and after opening it has the same content as that of the main repository README.md file.

* Now, when collaborator tries to change the content of the

README.md file in his forked repository

* We can see that the README.md file has been edited and now it will be committed in this forked repository.

* We can see the Commit history of the Repository

(Forked Repository)

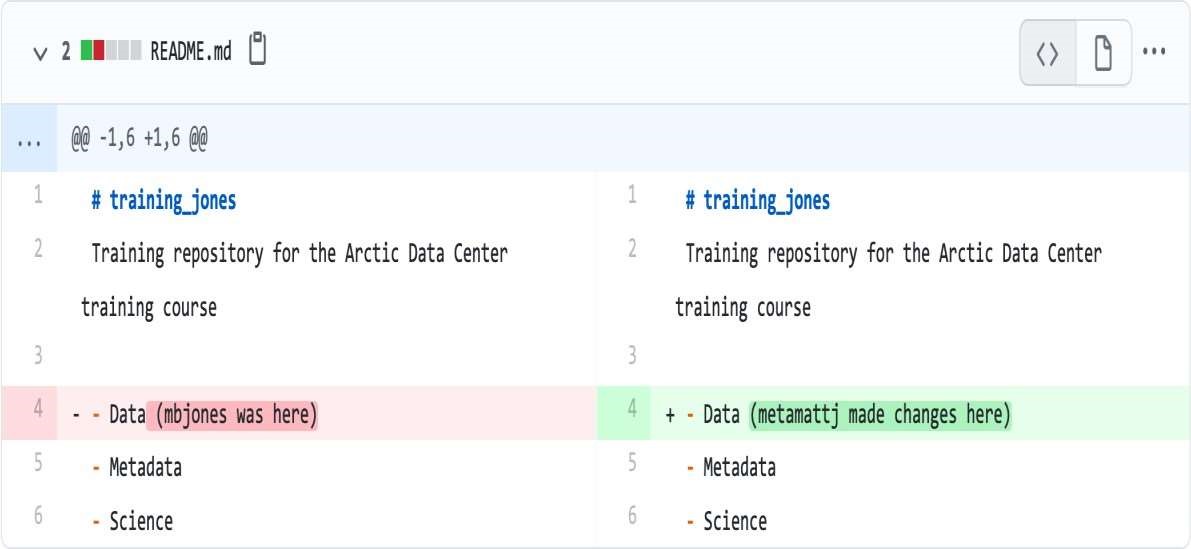
* After committing the README.md file in the main branch of the VanshikaArora//Task2, the new file will be changed.

# EXPERIMENT 8: MERGE AND RESOLVE CONFLICTS CREATED DUE TO OWN ACTIVITY

In this model, the collaborator will clone a copy of the owner’s repository from GitHub, and the owner will grant them collaborator status, enabling the collaborator to directly pull and push from the owner’s GitHub reposit Merge conflicts

So things can go wrong, which usually starts with a merge conflict, due to both collaborators making incompatible changes to a file. While the error messages from merge conflicts can be daunting, getting things back to a normal state can be straightforward once you’ve got an idea where the problem lies.

A merge conflict occurs when both the owner and collaborator change the same lines in the same file without first pulling the changes that the other has made. This is most easily avoided by good communication about who is working on various sections of each file, and trying to avoid overlaps. But sometimes it happens, and git is there to warn you about potential problems. And it will not allow overwriting.



The main problem in this is when the Owner and Collaborator both make changes to the same line of a file, git must know in whom it has to make.

How to resolve a conflict :

Sometimes you just made a mistake. When you get a merge conflict, the repository is placed in a ‘Merging’ state until you resolve it. There’s a command line command to abort doing the merge altogether:

git merge --abort

## Checkout

The simplest way to resolve a conflict, given that you know whose version of the file you want to keep, is to use the commandline git program to tell git to use either your changes (the person doing the merge), or their changes (the other collaborator).

* keep your collaborators file: git checkout --theirs conflicted file.Rmd
* keep your own file: git checkout --ours conflicted\_ file.Rmd Once you have run that command, then run add, commit, and push the changes as normal.

## Pull and edit the file

When you pulled the file with a conflict, git notices that there is a conflict and modifies the file to show both your own changes and your collaborator’s changes in the file. It also shows the file in the Git tab with an orange U icon, which indicates that the file is Unmerged, and therefore awaiting you help to resolve the conflict. It delimits these blocks with a series of less than and greater than signs, so they are easy to find:

To resolve the conflicts, simply find all of these blocks, and edit them so that the file looks how you want, and save. Be sure you removed the delimiter lines that started with

<<<<<<<, =======, and >>>>>>>.

Once you have made those changes, you simply add, commit, and push the files to resolve the conflict.

## Producing and resolving merge conflicts

To illustrate this process, we’re going to carefully create a merge conflict step by step, show how to resolve it, and show how to see the results of the successful merge after it is complete. First, we will walk through the exercise to demonstrate the issues.

## Owner and collaborator ensure all changes are updated

First, start the exercise by ensuring that both the Owner and Collaborator have all of the changes synced to their local copies of the Owner’s repository in R Studio. This includes doing a git pull to ensure that you have all changes local, and make sure that the Git tab in R Studio doesn’t show any changes needing to be committed.

## Owner makes a change and commits

From that clean slate, the Owner first modifies and commits a small change including their name on a specific line of the README.md file. Work to only change that one line, and add your username to the line in some form and commit the changes. We are now in the situation where the owner has not pushed changes that the collaborator cannot yet see.

## Collaborator makes a change and commits on same line

Now the collaborator also makes changes to the same of the README.md file in their R Studio copy of the project, adding their name to the line. They then commit. At this point, both the owner and collaborator have committed changes based on their shared version of the README.md file, but neither has tried to share their changes via GitHub.

## Collaborator pushes the file to GitHub

Sharing starts when the Collaborator pushes their changes to the GitHub repo, which updates GitHub to their version of the file. The owner is now one revision behind, but doesn’t yet know it.

## Owner pushes their changes and gets an error

At this point, the owner tries to push their change to the repository, which triggers an error from GitHub. While the error message is long, it basically tells you everything needed.



# EXPERIMENT 9: RESET AND REVERT

Reset a Git Commit

Let’s start with the Git command reset. Practically, you can think of it as a “rollback” – it points your local environment back to a previous commit. By “local environment,” we mean your local repository, staging area, and working directory.

If we look at what’s in our master branch now, we can see the chain of commits made so far.

$git log –one line b764644 File with three lines 7c709f0 File with two lines

9ef9173 File with one line

What happens if we want to roll back to a previous commit. Simple we can just move the branch pointer. Git supplies the reset command to do this for us. For example, if we want to reset master to point to the commit two back from the current commit, we could use either of the following methods:

$ git reset 9ef9173 (using an absolute commit SHA1 value 9ef9173)

Or

$ git reset current~2 (using a relative value -2 before the “current” tag)

Results of this operation. After this, if we execute a git log command on the current branch (master), we’ll see just the one commit.

$git log –one line

9ef9173 File with one line

The git reset command also includes options to update the other parts of your local environment with the contents of the commit where you end up. These options include: hard reset the commit being pointed to in the repository, populate the working directory with the contents of the commit, and reset the staging area; soft to only reset the pointer in the repository; and mixed (the default) to reset the pointer and the staging area.

Using these options can be useful in targeted circumstances such as git reset –hard <commit sha1 | reference>. This overwrites any local changes you haven’t committed. In effect, it resets (clears out) the staging area and overwrites content in the working directory with the content from the commit you reset to. Before you use the hard option, be sure that’s what you really want to do, since the command overwrites any uncommented changes.

Revert a Git Commit

The net effect of the git revert command is similar to reset, but its approach is different. Where the reset command moves the branch pointer back in the chain (typically) to “undo” changes, the revert command adds a new commit at the end of the chain to “cancel” changes. If we add a line to a file in each commit in the chain, one way to get back to the version with only two lines is to reset to that commit, i.e., git reset HEAD~1.

Another way to end up with the two-line version is to add a new commit hat has the third line removed—effectively cancelling out that change. This can be done with a git revert command, such as:

$ git revert HEAD

Because this adds a new commit, Git will prompt for the commit message:

Revert “File with three lines”

This reverts commit

b764644bad524b804577684bf74e7bca3117f554

# Please enter the commit message for your changes. Lines starting

# with ‘#’ will be ignored, and an empty message aborts the commit.

# On branch master

# Changes to be committed:

# modified: file1.txt

#

If we do a git log now, we’ll see a new commit that reflects the contents before the previous commit.

$ git log –oneline

11b7712 Revert “File with three lines” b764644 File with three lines 7c709f0 File with two lines

9ef9173 File with one line

Here are the current contents of the file in the working directory:

$ cat <filename>

Line 1

Line 2

Note: If you see the lines as we got after the git revert command, just visit your default text editor for git and commit the message from there, or it could directly take to you your default editor. We want a message here because when using git revert, it does not delete the commit instead makes a new commit that contains the removed changes from the commit. Difference Table

|  |  |  |
| --- | --- | --- |
| Git checkout | Git reset | Git revert |
| Discards the changes in the working repository. | Unstages a file and bring our changes back to the working directory | Removes the commits from the remote repository. |
| Used in the local repository. | Used in local repository | Used in the remote repository |
| Does not make any changes to the commit history. | Alters the existing commit history | Adds a new commit to the existing commit history. |
| Moves HEAD pointer to a specific commit. | Discards the uncommitted changes. | Rollbacks the changes which we have committed. |
| Can be used to manipulate commits or files. | Can be used to manipulate commits or files. | Does not manipulate your commits or files. |

