

**Subject Name : Source Code Management**

**Subject Code : CS181**

**Cluster : Beta**

**Department : CSE**



**Submitted By:**

Amy

2110990169

G8

**Submitted To:**

Mr. Monit Kapoor

## **INDEX**

<b>S. No.</b>	<b>Title</b>	<b>Page No.</b>
1	Setting up of Git Client	5-7
2	Setting up GitHub Account	8-9
3	Generate logs	10-11
4	Create and visualize branches	12-14
5	Git lifecycle description	15-16
6	Add collaborators on GitHub Repo	17-21
	Fork and Commit	21-25
8	Merge and Resolve conflicts created due to own activity and collaborators activity.	26-30
9	Reset and Revert	31-38
10	Task 2	38-51

### ❖ What is Git and Why it is used?

Git is a Free and open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. Git is easy to learn and has a tiny footprint with lightning fast performances.

It is used for:

- ✓ Tracking Code Changes
- ✓ Tracking who made Changes
- ✓ Coding Collaboration

### ❖ What is GITHUB?

GitHub is a code hosting platform for version control and collaboration. It lets you and others work together on Projects from anywhere.

### ❖ What is Repository?

A repository contains all of your project's files and each file's revision history. You can discuss and manage your project's work within the repository. A Git repository is the .git/ folder inside a project. This repository tracks all changes made to files in your project, building a history over time. Meaning, if you delete the .git/ folder, then you delete your project's history.

### ❖ What is Version Control System (VCS)?

Version Control Systems are the software tools for tracking/managing all the changes made to the source code during the project development. It keeps a record of every single change made to the code. It also allows us to turn back to the previous version of the code if any mistake is made in the current version. Without a VCS in place, it would not be possible to monitor the development of the project.

## ❖ Types of VCS

The three types of VCS are:

1. Local Version Control System
2. Centralized Version Control System
3. Distributed Version Control System

**1. Local Version Control System:** Local Version Control System is located in your local machine. If the local machine crashes, it would not be possible to retrieve the files, and all the information will be lost. If anything happens to a single version, all the versions made after that will be lost.

**2. Centralized Version Control System:** In the Centralized Version Control Systems, there will be a single central server that contains all the files related to the project, and many collaborators checkout files from this single server (you will only have a working copy). The problem with the Centralized Version Control Systems is if the central server crashes, almost everything related to the project will be lost.

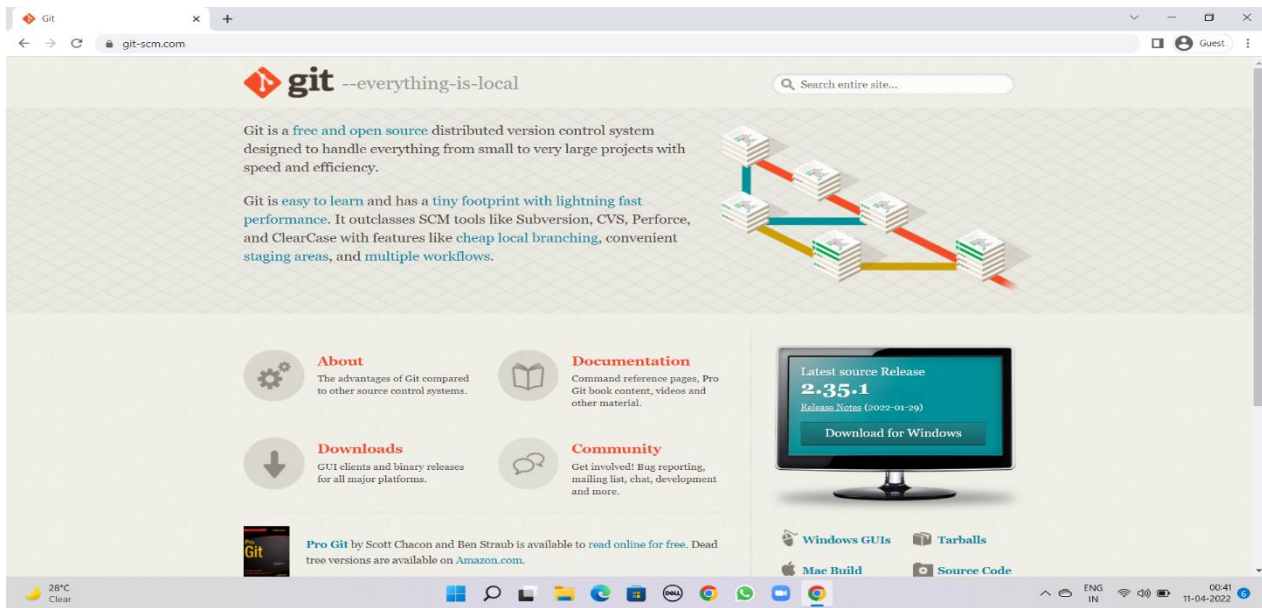
**3. Distributed Version Control System:** In a distributed version control system, there will be one or more servers and many collaborators similar to the centralized system. But the difference is, not only do they check out the latest version, but each collaborator will have an exact copy of the main repository on their local machines. Each user has their own repository and a working copy. This is very useful because even if the server crashes we would not lose everything as several copies are residing in several other computers.



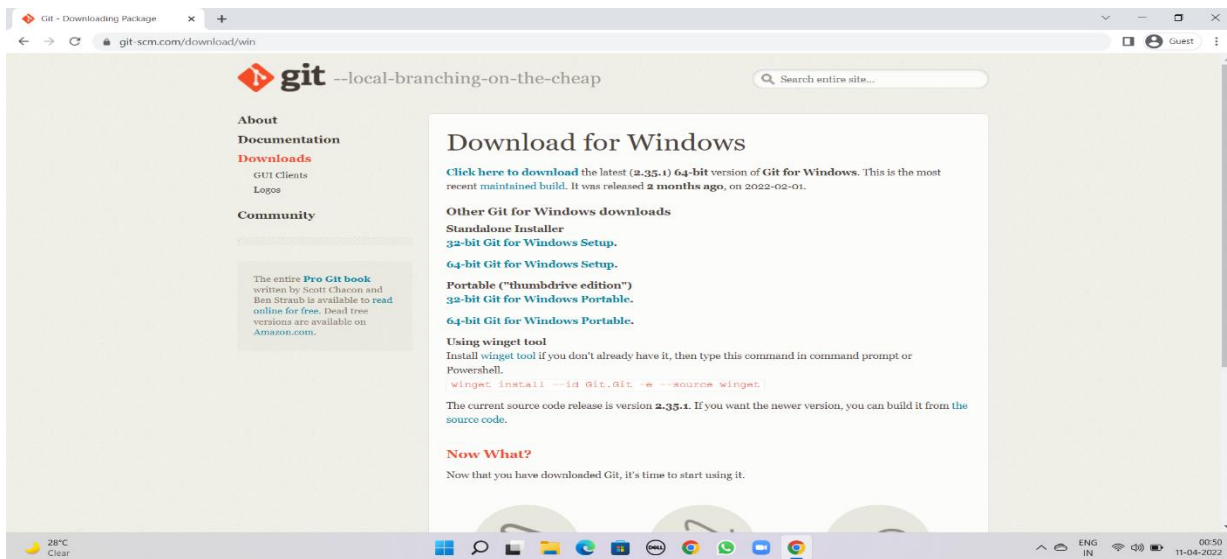
## Experiment – 1

**Aim:** Setting up the git client.

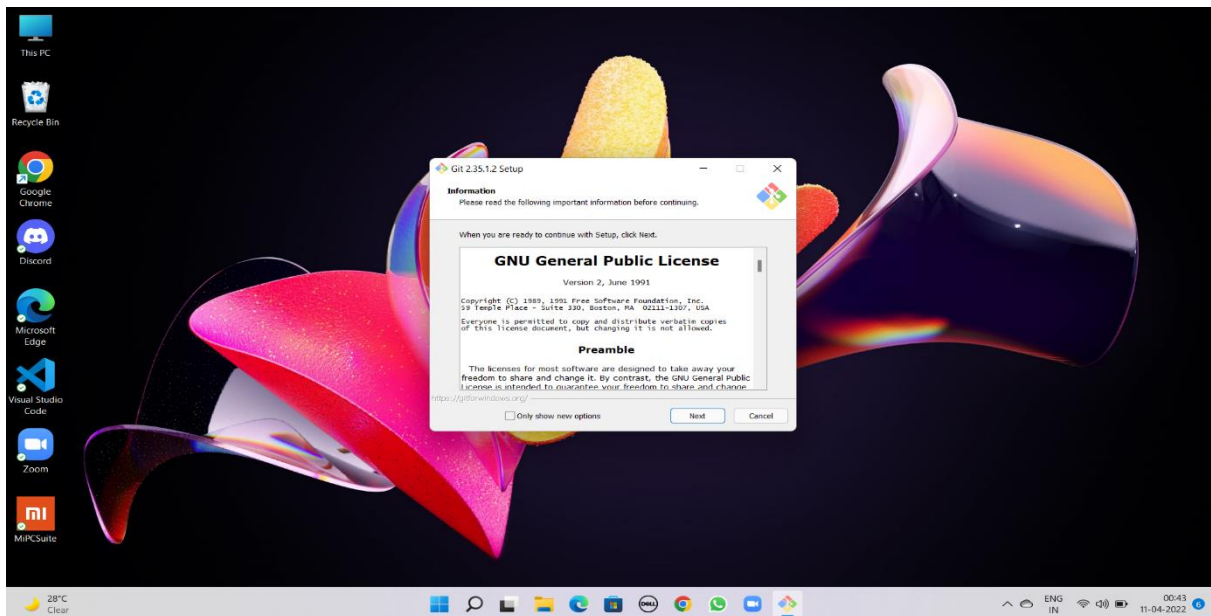
**Git Installation:** Download the Git installation program (Windows, Mac, or Linux) from [Git-Downloads\(git-scm.com\)](https://git-scm.com).



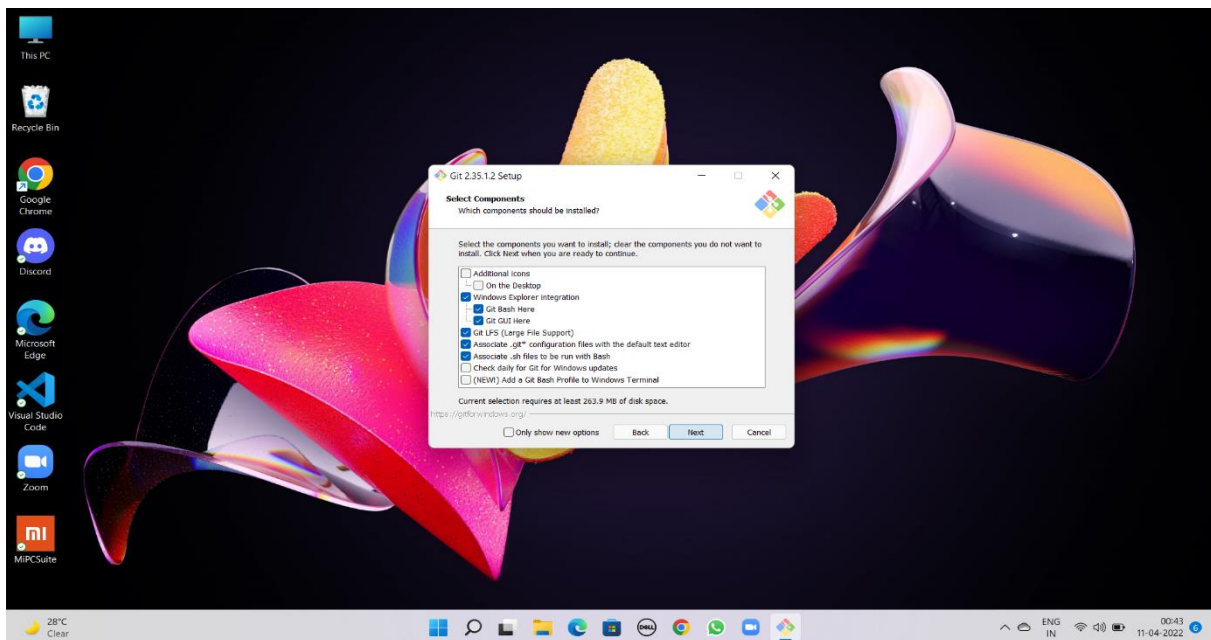
1. After opening this site, you have to select your operating system by clicking on it. Here I will show you the steps for the Windows operating system.



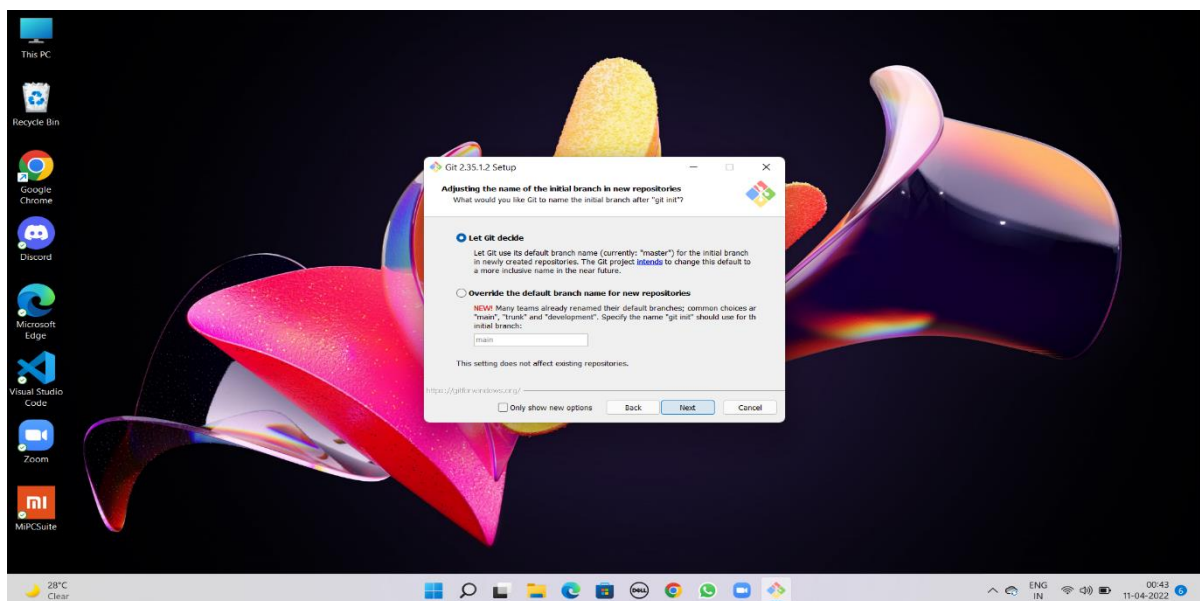
2. Now select the processor of the system you have. (Most of the system are now of 64-bits) After selecting the processor your download will start.
3. Now you have to open this folder.
4. After opening you will given a notification “Do you want to allow this app to make changes in your PC”
5. Click Next



6. Click Next

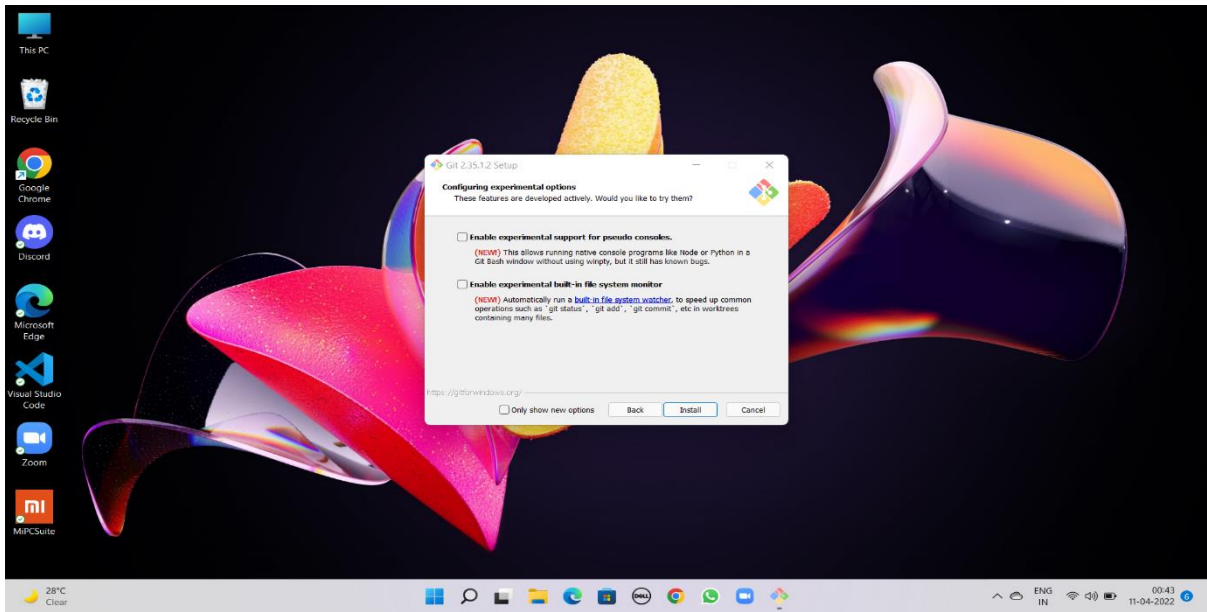


7. Click Next



8. Click on Install





9. Click on Finish after the installation is finished.
10. The installation of the git is finished and now we have to setup git client and GitHub account.

## Experiment – 2

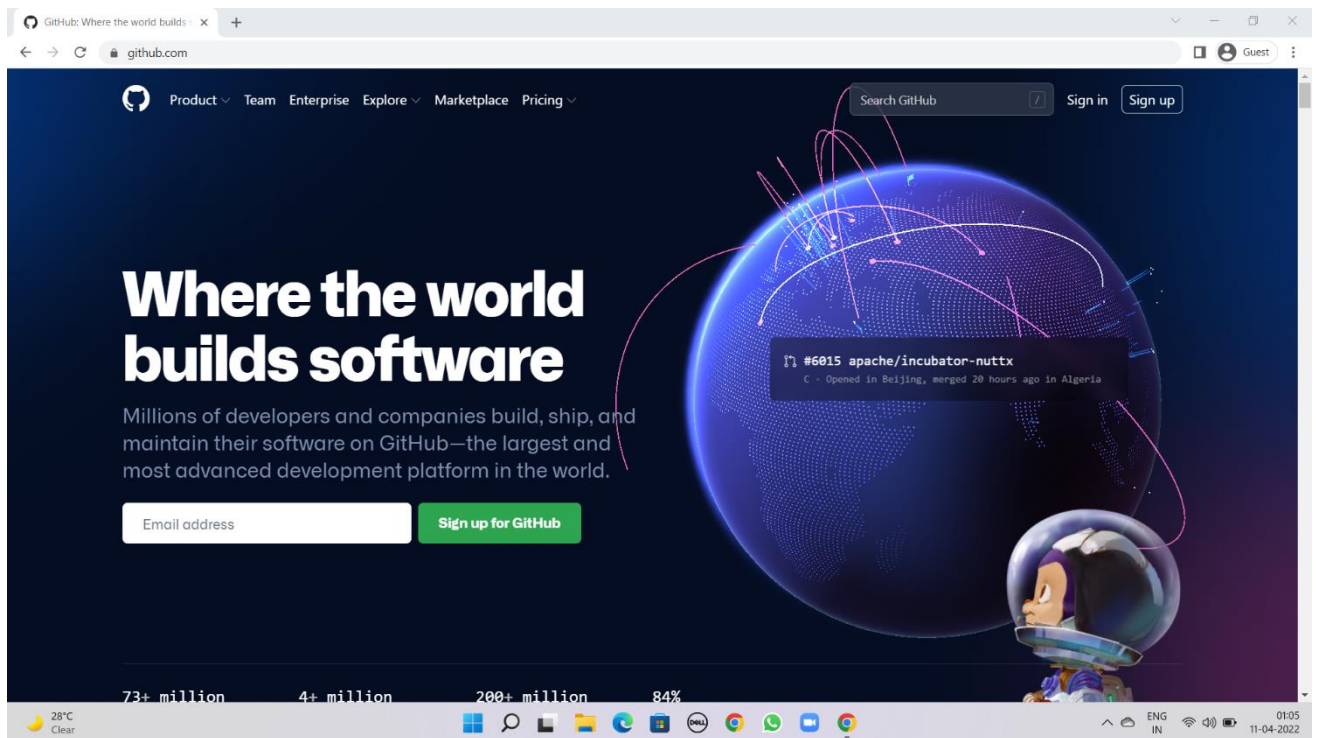
Aim: Setting up GitHub Account

The first steps in starting with GitHub are to create an account, choose a product that fits your needs best, verify your email, set up two-factor authentication, and view your profile.

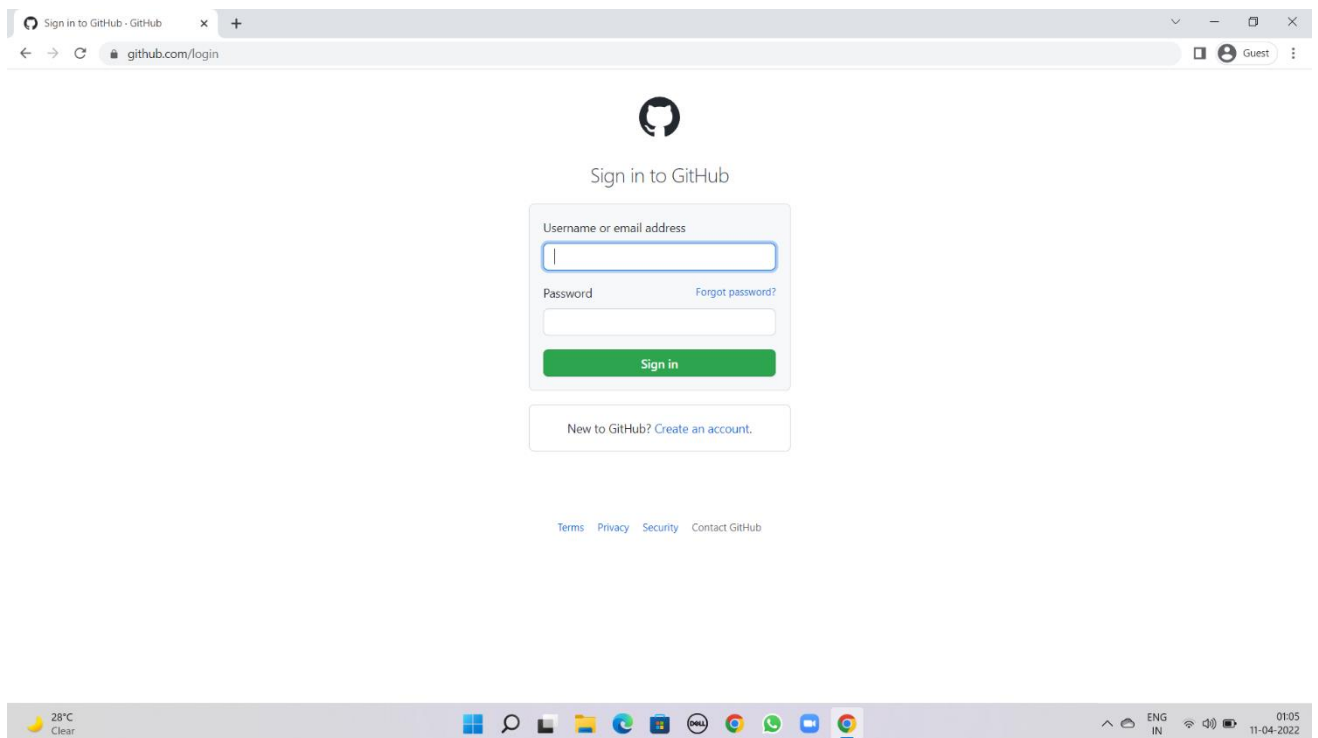
There are several types of accounts on GitHub. Every person who uses GitHub has their own user account, which can be part of multiple organisations and teams. Your user account is your identity on GitHub.com and represents you as an individual.

1. **Creating an account:** To sign up for an account on GitHub.com, navigate to <https://github.com/> and follow the prompts.





2. Click on Create an account if you are a new user or if you have already an account, please login.



3. After Clicking on create a new account you will be redirected to a new

page where you have to enter your email id which you want to use for your account. Now enter your password you want to create for your GitHub account. After that you will be asked to enter your username.

4. Now Click on Create Account.
5. Verify it from your email and you are all set to go.

## Experiment - 3

**Aim:** Program to generate logs

### **Git Status:**

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git. Status output does not show you any information regarding the committed project history.

### **Git Init:**

The git init is one way to start a new project with Git. To start a repository, use either git init or git clone - not both. To initialize a repository, Git creates a hidden directory called .git . That directory stores all of the objects and refs that Git uses and creates as a part of your project's history.

### **Git Add:**

`git add [filename]` selects that file, and moves it to the staging area, marking it for inclusion in the next commit. You can select all files, a directory, specific files, or even specific parts of a file for staging and commit.

### **Git commit:**

The `git commit` command captures a snapshot of the project's currently staged changes. Committed snapshots can be thought of as “safe” versions of a project—Git will never change them unless you explicitly ask it to.

### **Git Log:**

Git log is a utility tool to review and read a history of everything that happens to a repository. Multiple options can be used with a git log to make history more specific. Generally, the git log is a record of commits.



```

MINGW64/d/CSE/Git/git bash/second repo
91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   ../index.txt
        new file:   ../tut.txt

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git init
Initialized empty Git repository in D:/CSE/Git/git bash/second repo/.git/

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git add .

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git.txt

nothing added to commit but untracked files present (use "git add" to track)

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git add .

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   git.txt

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git commit -m "version 1"
Author identity unknown

*** Please tell me who you are.

Run

  git config --global user.email "you@example.com"
  git config --global user.name "Your Name"

to set your account's default identity.
Omit --global to set the identity only in this repository.

fatal: unable to auto-detect email address (got '91628@Amy.(none)')

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ git log
fatal: your current branch 'master' does not have any commits yet

91628@Amy MINGW64 /d/CSE/Git/git bash/second repo (master)
$ |
  
```

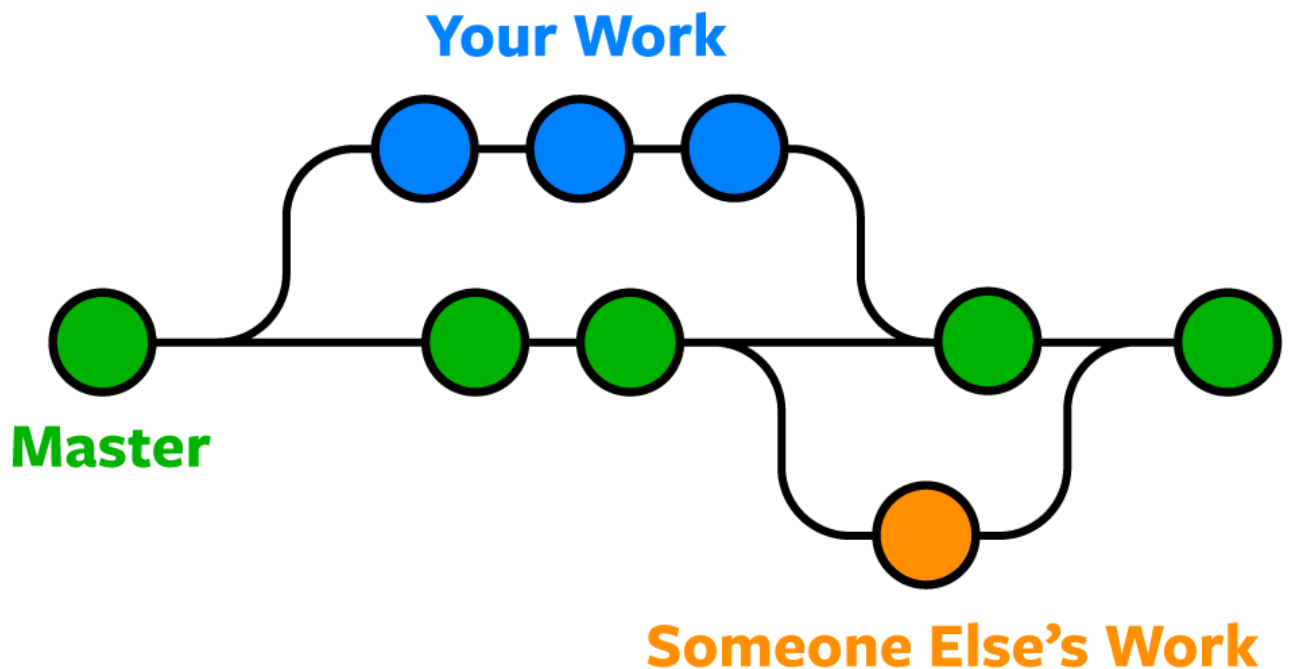


## Experiment – 4

**Aim:** Create and visualize branches in Git

### Git Branches:

A branch in Git is an independent line of work (a pointer to a specific commit). It allows users to create a branch from the original code (master branch) and isolate their work. Branches allow you to work on different parts of a project without impacting the main branch.



Let us see the command of it:

Firstly, add a new branch, let us suppose the branch name is activity .

For this use command →

**git branch name** [adding new branch]

**git branch** [use to see the branch's names]

**git checkout branch name** [use to switch to the given branch]



MINGW64:/d/CSE/Git/git bash

```
01628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git branch
* master

01628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git branch activity

01628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git branch
* activity
  master

01628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git checkout activity
Switched to branch 'activity'

01628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$ git branch
* activity
  master

01628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$ ls
index.txt  tut.txt

01628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$
```

In this you can see that firstly 'git branch' shows only one branch in green colour but when we add a new branch using 'git branch activity', it shows 2 branches but the green colour and star is on master. So, we have to switch to main by using 'git checkout main'. If we use 'git branch', now you can see that the green colour and star is on activity. It means you are in main branch and all the data of master branch is also on activity branch.

MINGW64:/d/CSE/Git/git bash

```
01628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$ ls
gitt.txt  index.txt  tut.txt

01628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$
```

Now add a new file in activity branch, do some changes in file and commit the file



MINGW64:/d/CSE/Git/git bash

```
91628@Amy MINGW64 /d/CSE/Git/git bash (activity)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git branch
  activity
* master

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ ls
index.txt  tut.txt

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ |
```

To add these files in master branch, we have to do merging. For this firstly switch to master branch and then use command:

MINGW64:/d/CSE/Git/git bash

```
91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ ls
index.txt  tut.txt

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git merge activity
Updating 32b80ea..3d37628
Fast-forward
 gitt.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 gitt.txt

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ git branch
  activity
* master

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ ls
gitt.txt  index.txt  tut.txt

91628@Amy MINGW64 /d/CSE/Git/git bash (master)
$ |
```

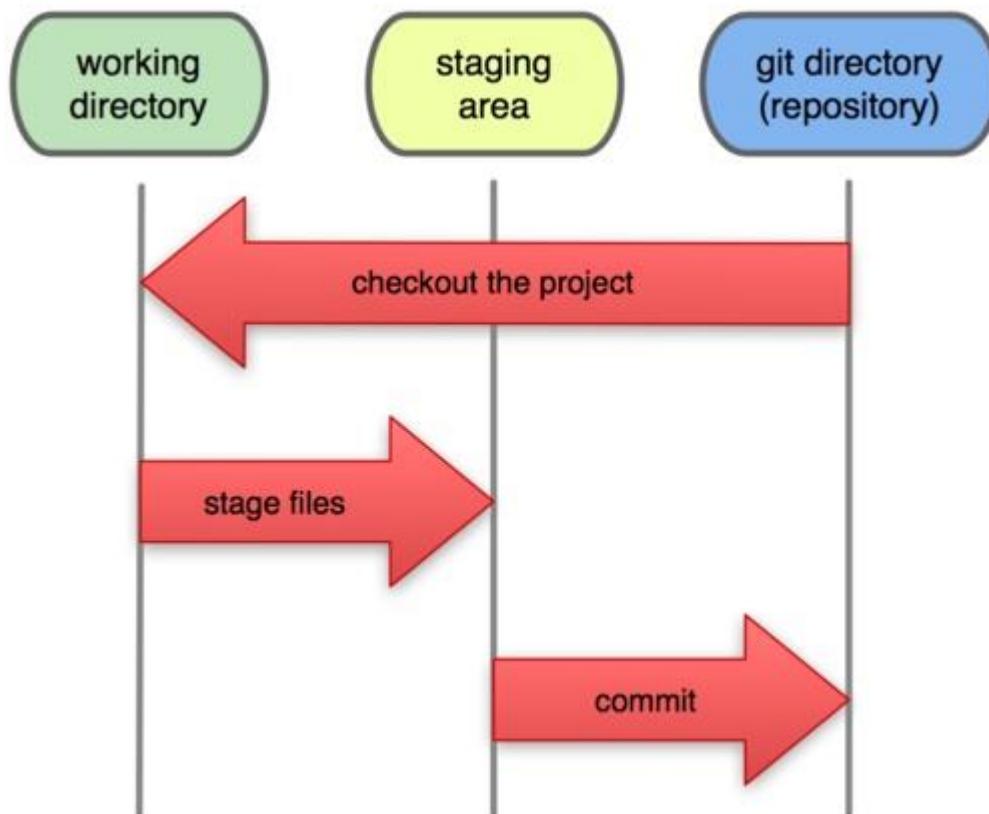




## Experiment – 5

**Aim:** Git Life Cycle Description

Now let's understand the three-stage architecture of Git:



**Working Directory:** This is the directory that we've initialized, and here all the changes are made to commit on GitHub.

**Staging Area:** This is where we first put out code or files of the working repository. The command that we use to stage code is, "git add --a", "git add File Name" or "git add -A". In simple terms, staging means telling Git what files we want to commit (new untracked files, modified files, or deleted files).

**Git directory(repository):** This is where all the commits are stored whenever we make a commit. We can revert to an older version of or

project using the “git checkout” command from this directory.

## Some Important Commands

### ➤ Git operations and commands:

First of all, Create a local repository using Git. For this, you have to make a folder in your device, right click and select “**Git Bash Here**”. This opens the Git terminal. To create a new local repository, use the command “**git init**” and it creates a folder **.git**.

➤ When we use GIT for the first time, we have to give the user name and email so that if I am going to change in project, it will be visible to all.

For this, we use command →

**“git config --global user.name Name”**

**“git config --global user.email email”**

For verifying the user’s name and email, we use →

**“git config --global user.name”**

**“git config --global user.email”**

- **ls** → It gives the file names in the folder.
- **ls -lart** → Gives the hidden files also.
- **git status** → Displays the state of the working directory and the staged snapshot.
- **touch filename** → This command creates a new file in the repository.
- **Clear** → It clears the terminal.
- **rm -rf .git** → It removes the repository.
- **git log** → displays all of the commits in a repository's history
- **git diff** → It compares my working tree to staging area.

## Experiment-6

**Aim:** Add collaborators on GitHub Repository.

### **Theory:**

In GitHub, We can invite other GitHub users to become collaborators to our private repositories(which expires after 7 days if not accepted, restoring any unclaimed licenses). Being a collaborator, of a personal repository you can pull (read) the contents of the repository and push (write) changes to the repository. You can add unlimited collaborators on public and private repositories(with some per day limit restrictions). But, in a private repository, the owner of the repo can only grant write-access to the collaborators, and they can't have the read-only access.

GitHub also restricts the number of collaborators we can invite within a period of 24 hours. If we exceed the limit, then either we have to wait for 24-hours or we can also create an organization to collaborate with more people.

### **Actions that can be Performed By Collaborators**

Collaborators can perform a number of actions into someone else's personal repositories, they have gained access to Some of them are,

- Create, merge, and close pull requests in the repository
- Publish, view, install the packages
- Fork the repositories
- Make the changes on the repositories as suggested by the Pull requests.
- Mark issues or pull requests as duplicate
- Create, edit, and delete any comments on commits, pull requests, and issues in the repository
- Removing themselves as collaborators on the repositories.
- Manage releases in the repositories.



Now, let's see how can we invite collaborators to our repositories.

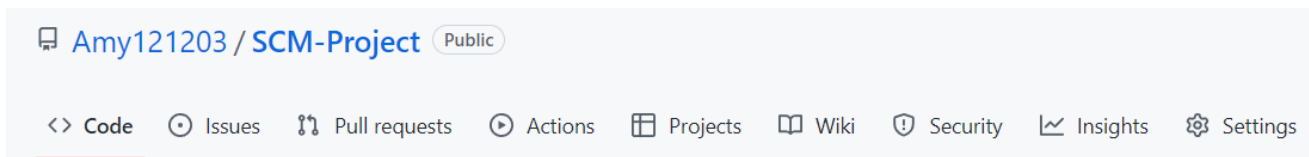
## Inviting Collaborators to your personal repositories

Follow the steps below to invite collaborators to your own repository (public or private).

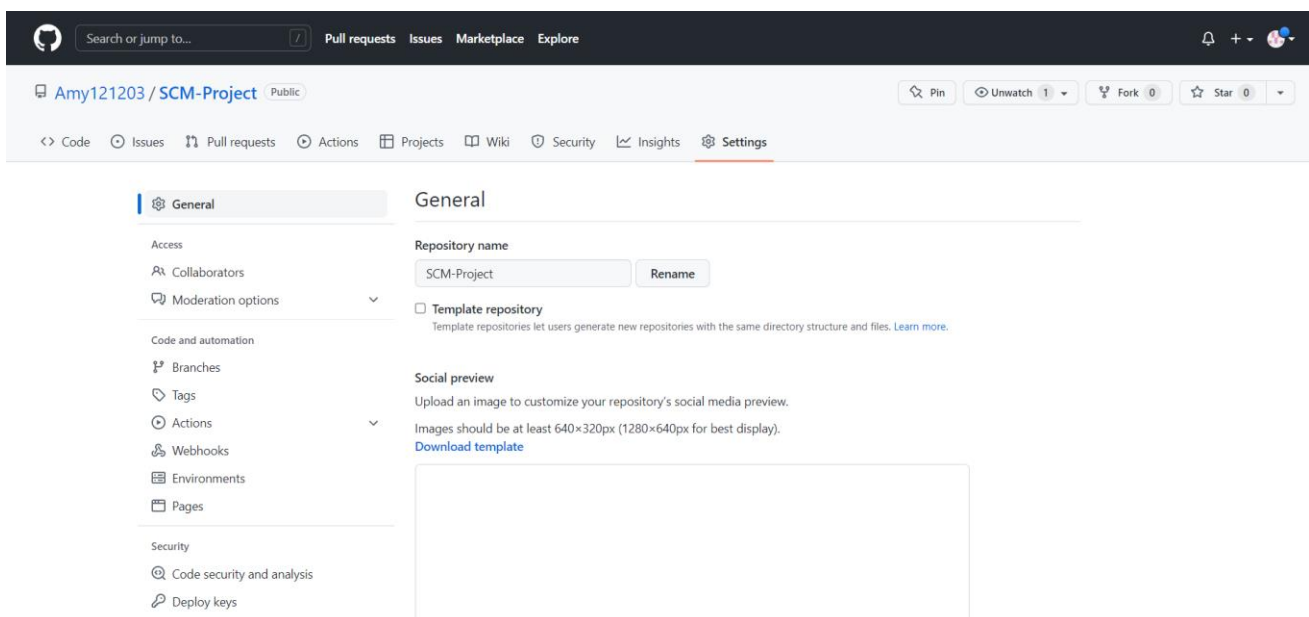
**Step 1:** Get the usernames of the GitHub users you will be adding as collaborators. In case, they are not on GitHub, ask them to sign in to GitHub.

**Step 2:** Go to your repository( intended to add collaborators)

**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.

Manage access



You haven't invited any collaborators yet

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.



X

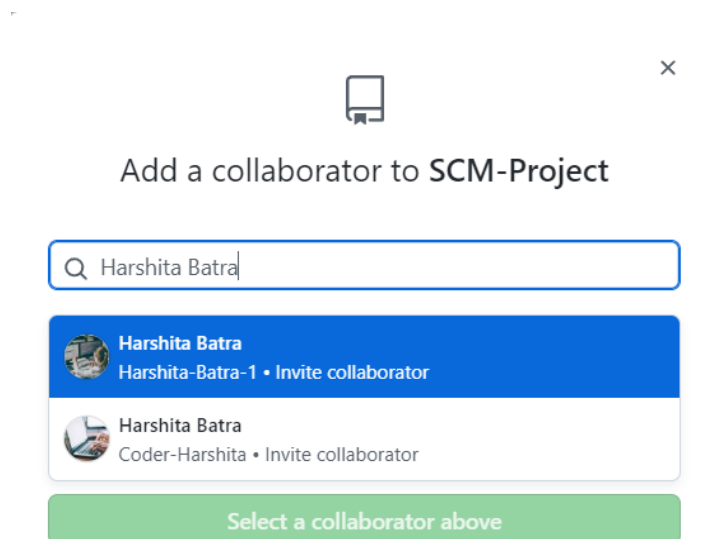
Add a collaborator to SCM-Project

Q Search by username, full name, or email

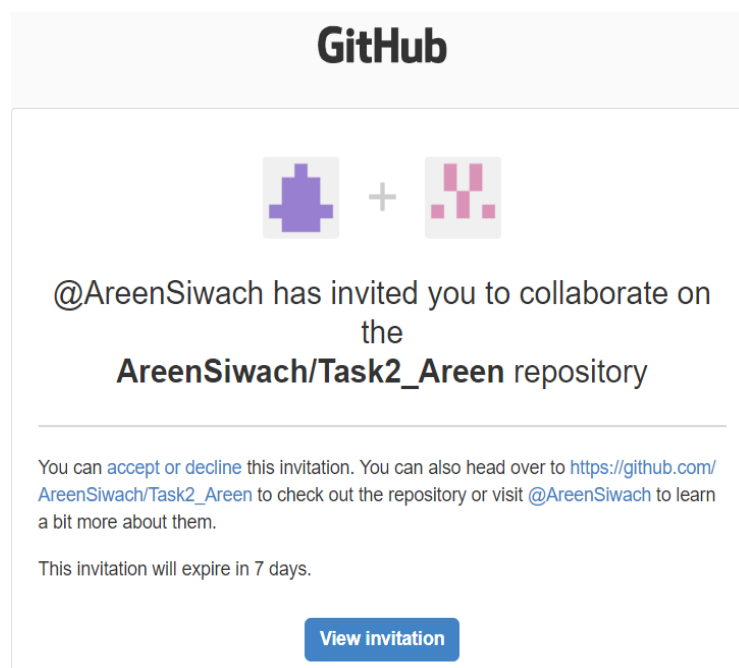
Select a collaborator above



**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 for our project will get a email from the Team leader (by GitHub). He/she has to open its 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. Now, they will get a mail regarding invitation to your repository. Once they accept, they will have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.

## Manage access

Add people

<input type="checkbox"/> Select all		Type ▾
<input type="text" value="Find a collaborator..."/>		
<input type="checkbox"/>	 <b>anushkam012</b> Collaborator	Remove
<input type="checkbox"/>	 <b>AreenSiwach</b> Collaborator	Remove
<input type="checkbox"/>	 <b>Harshita Batra</b> Harshita-Batra-1 • Collaborator	Remove



## Experiment-7

Aim: Fork and Commit

Forking a repository means creating a copy of the repo. When you fork a repo, you create your own copy of the repository on your GitHub account.

This is done for the following reasons:

1. You have your own copy of the project on which you may test your own changes without changing the original project.
2. This helps the maintainer of the project to better check the changes you made to the project and has the power to either accept, reject or suggest something
3. When you clone an Open Source project, which isn't yours, you don't have the right to push code directly into the project.

For these reasons, you are always suggested to FORK. Let's have a screenshot walkthrough of the whole process. When getting started with a contribution to Open Source Project, you have been advised to first FORK the repository(repo). But what is a fork?

You must have seen this icon on every repository in the top right corner. Now, this button is used to Fork the repo. But again, what is a fork or forking a repository in GitHub as shown in the below media as follows:

### Procedure:

**Step 1:** Go to Project SCM official repository.

**Step 2:** Find the Fork button on the top right corner.



Group08-Chitkara-University / 2110990169 Public

Edit Pins Watch 1 Fork 0 Star 0

Code Issues Pull requests Actions Projects Wiki Security Insights Settings

### Step 3: Click on Fork.

You will see this screen.

#### Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project.

Owner \*

Repository name \*



Amy121203



2110990169



By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

Description (optional)

Amy

*i* You are creating a fork in your personal account.

Create fork

After this, we will see how to work in the forked repositories which were forked by the collaborators. If the collaborator tries to change in his forked repository, it will not affect the main repository.

Group08-Chitkara-University / 2110990169 Public

Edit Pins Watch 1 Fork 1 Star 0

Code Issues Pull requests Actions Projects Wiki Security Insights Settings



## What is COMMIT in GitHub?

Commit is like a snapshot of your repository. These commits are snapshots of your entire repository at specific times. 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.

It is similar to saving a file that's been edited, a commit records changes to one or more files in your branch. Git assigns each commit a unique ID, called a SHA or hash, that identifies:

- The Specific Changes
- When the Changes were made
- Who created the changes

When you make a commit, you must include a commit message that briefly describes the changes, you can also add a co-author on any commits you collaborate on.

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

master 1 branch 0 tags

Go to file Add file Code

This branch is 1 commit ahead of Group08-Chitkara-University:master. Contribute Fetch upstream

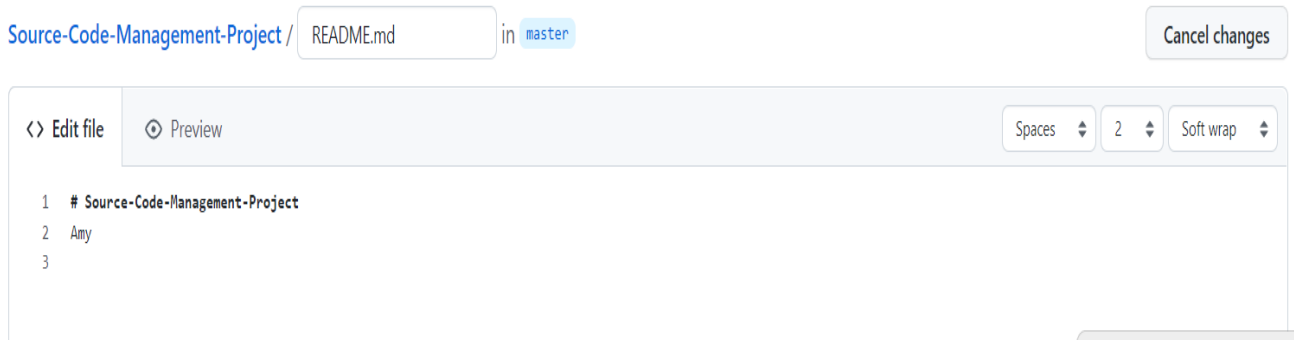
Amy121203 Create README.md 5dc791f in 24 seconds 2 commits

README.md Create README.md now

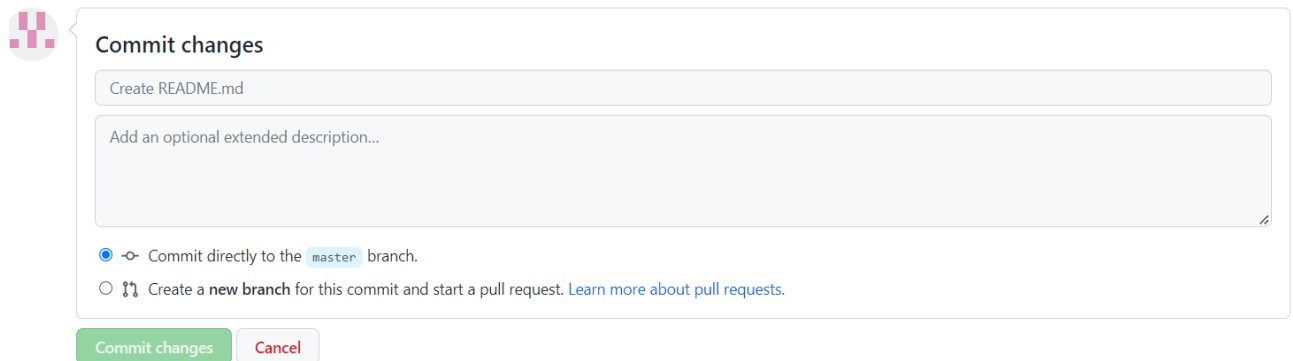


The content in the README.md file in the main Mascots repository, is given as below:

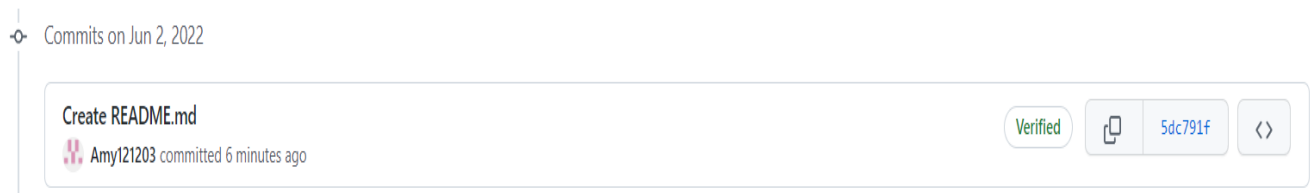
- Now, when Amy121203 (collaborator) tries to change the content of the README.md file in her forked repository i.e.,



- 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 Amy121203/Source-Code-Management-Project Repository.

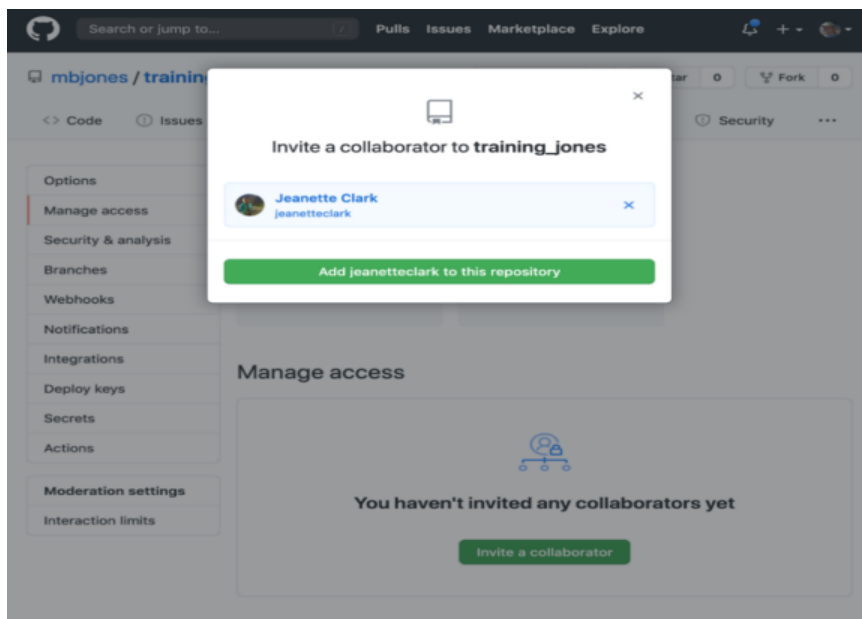




## Experiment-8

**Aim:** Merge and Resolve conflicts created due to own activity and collaborators activity.

### Step 1: Collaborator clone



To be able to contribute to a repository, the collaborator must clone the repository from the Owner's GitHub account. To do this, the Collaborator should visit the GitHub page for the Owner's repository, and then copy the clone URL. In R Studio, the Collaborator will create a new project from version control by pasting this clone URL into the appropriate dialog (see the earlier chapter introducing GitHub).

### Step 2: Collaborator Edits

With a clone copied locally, the Collaborator can now make changes to the index.Rmd file in the repository, adding a line or statement somewhere noticeable near the top. Save your changes.

### **Step 3: Collaborator commit and push**

To sync changes, the collaborator will need to add, commit, and push their changes to the Owner's repository. But before doing so, its good practice to pull immediately before committing to ensure you have the most recent changes from the owner. So, in R Studio's Git tab, first click the "Diff" button to open the git window, and then press the green "Pull" down arrow button. This will fetch any recent changes from the origin repository and merge them. Next, add the changed index.Rmd file to be committed by clicking the checkbox next to it, type in a commit message, and click 'Commit.' Once that finishes, then the collaborator can immediately click 'Push' to send the commits to the Owner's GitHub repository.

### **Step 4: Owner pull**

Now, the owner can open their local working copy of the code in RStudio, and pull those changes down to their local copy.

**Congrats, the owner now has your changes!**

### **Step 5: Owner edits, commit, and push**

Next, the owner should do the same. Make changes to a file in the repository, save it, pull to make sure no new changes have been made while editing, and then add, commit, and push the Owner changes to GitHub.

### **Step 6: Collaborator pull**

The collaborator can now pull down those owner changes, and all copies are once again fully synced. And you're off to collaborating.

## **Challenge**

Now that the instructors have demonstrated this conflict-free process, break into pairs and try the same with your partner. Start by designating one person as the Owner and one as the Collaborator, and then repeat the steps



described above:

- Step 0: Setup permissions for your collaborator
- Step 1: Collaborator clones the Owner repository
- Step 2: Collaborator Edits the README file
- Step 3: Collaborator commits and pushes the file to GitHub
- Step 4: Owner pulls the changes that the Collaborator made
- Step 5: Owner edits, commits, and pushes some new changes
- Step 6: Collaborator pulls the owners changes from GitHub

## Merge Conflicts

So things can go wrong, which usually starts with a merge conflict, due to both collaborators making 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 git will not allow you to overwrite one person's changes to a file with another's changes to the same file if they were based on the same version.

```
... @@ -1,6 +1,6 @@  
1 # training_jones  
2 Training repository for the Arctic Data Center  
  training course  
3  
4 - - Data (mbjones was here)  
5 - Metadata  
6 - Science
```



The main problem with merge conflicts is that, when the Owner and Collaborator both make changes to the same line of a file, git doesn't know whose changes take precedence. You have to tell git whose changes to use for that line.

## **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 RStudio. This includes doing a git pull to ensure that you have all changes local, and make sure that the Git tab in RStudio 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 (we will change line 4). Work to only change that one line, and add your username to the line in some form and commit the changes (but DO NOT push). We are now in the situation where the owner has unpushed changes that the collaborator cannot yet see.

### **Collaborator makes a change and commits on the same line**

Now the collaborator also makes changes to the same (line 4) of the README.md file in their RStudio 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 (that the owner's repository doesn't reflect the changes on GitHub, and that they need to pull before they can push).

A screenshot of a terminal window titled "Git Push" with a "Close" button in the top right corner. The terminal shows the command `>>> /usr/bin/git push origin HEAD:refs/heads/main` and the following output:

```
Warning: Permanently added the RSA host key for IP address '140.82.114.4' to the list of known hosts.
To github.com:mbjones/training_jones.git
 ! [rejected]        HEAD -> main (fetch first)
error: failed to push some refs to 'git@github.com:mbjones/training_jones.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

## Experiment-9

### Aim: Reset and Revert

**Theory:** While Working with Git in certain situations we want to undo changes in the working area or index area, sometimes remove commits locally or remotely and we need to reverse those changes. There are 3 different ways in which we can undo the changes in our repository, these are git reset, git checkout, and git revert. git checkout and git reset in fact can be used to manipulate commits or individual files. These commands can be confusing so it's important to find out the difference between them and to know which command should be used at a particular point of time

Let's make a sample git repository with a file constructor.cpp and Code written inside it.

```
91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:   24StarpatternTriangle1.cpp -> Pattern1.cpp
        renamed:   25squarepattern.cpp -> Pattern2.cpp
        renamed:   26StarPatternTriangle2.cpp -> Pattern3.cpp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Pattern2.cpp

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git add Pattern1.cpp

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git commit -m "Pattern"
[master 74f026b] Pattern
 3 files changed, 2 insertions(+), 2 deletions(-)
 rename 24StarpatternTriangle1.cpp => Pattern1.cpp (93%)
 rename 25squarepattern.cpp => Pattern2.cpp (100%)
 rename 26StarPatternTriangle2.cpp => Pattern3.cpp (93%)

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git log
commit 74f026bfde8e428dee8184e763c31a9b280d0983 (HEAD -> master)
Author: amy121203 <amy0169.be21@chitkara.edu.in>
Date: Thu Jun 2 22:28:09 2022 +0530

    Pattern

commit 58407f54eece7cd3edf9bb590db2e583dab4ff4b (origin/master)
Author: amy121203 <amy0169.be21@chitkara.edu.in>
Date: Thu Jun 2 17:01:34 2022 +0530

    first commit

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$
```

We can see that we have a single commit is done on the code file that has been committed with added new files in it. Now let's add some more comment or code to our code file. Let's add another line Hello World. Doing this change, our code file now needs to be added to the staging area for getting the commit done. This updates are currently in the working area and to see them we will see those using git status.

```
91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Pattern2.cpp
```

Now we have a change Hello World which is untracked in our working repository and we need to discard this change. So, the command that we should use here is –

## ➤ Git checkout

Git checkout is used to discard the changes in the working repository.

git checkout<filename>

When we write git checkout command and see the status of our git repository and also the code file we can see that our changes are being discarded from the working directory and we are again back to the code file that we had before. Now, what if we want to unstage a file. We stage our files before committing them and at a certain point, we might want to unstage a file. Let's add **Hello World** again to our text document and stage them using the **git add** command.

```
91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
(use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   Pattern2.cpp

no changes added to commit (use "git add" and/or "git commit -a")

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git checkout -- Pattern.cpp
```

## ➤ Git Reset

Git reset is used when we want to unstage a file and bring our changes back to the working directory. Git reset can also be used to remove commits from the local repository.

`git reset HEAD<filename>`

Whenever we unstage a file, all the changes are kept in the working area.

We are back to the working directory, where our changes are present but the file is now unstaged. Now there are also some commits that we don't want to get committed and we want to remove them from our local repository. To see how to remove the commit from our local repository let's stage and commit the changes that we just did and then remove that commit.

We have 2 commits now, with the latest being the Added Hello World commit which we are going to remove. The command that we would be using now is –

`git reset Head~1`

Point to be noted :-

- HEAD~1 here means that we are going to remove the topmost commit or the latest commit that we have done.
- We cannot remove a specific commit with the help of git reset , for ex : we cannot say that we want to remove the second commit or the third commit , we can only remove latest commit or latest 2 commits . latest N commits.(HEAD~n) [n here means n recent commits that needs to be deleted].

After using the above command we can see that our commit is being deleted and also our file is again unstaged and is back to the working directory. There are different ways in which git reset can actually keep your changes.

- git reset --soft HEAD~1 - This command will remove the commit but would not unstage a file. Our changes still would be in the staging area.
- git reset --mixed HEAD~1 or git reset HEAD~1 - This is the default command that we have used in the above example which removes the commit as well as unstage the file and our changes are stored in the working directory.
- git reset --hard HEAD~1 - This command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.

Points to keep in mind while using git reset command –

- If our commits are not published to remote repository, then we can use git reset.
- Use git reset only for removing commits that are present in our local directory and not in remote directory.



- We cannot remove a specific commit with the help of git reset, for ex: we cannot say that we want to remove the second commit or the third commit, we can only remove latest commit or latest 2 commits ... latest N commits. (HEAD~n) [n here means n recent commits that needs to be deleted].

We just discussed above that the git reset command cannot be used to delete commits from the remote repository, then how do we remove the unwanted commits from the remote repository.

## Git Revert

Git revert is used to remove the commits from the remote repository. Since now our changes are in the working directory, let's add those changes to the staging area and commit them.

git revert<commit id of the commit needs to be removed>

```
91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git log
commit 29f7d2354f8c8903eb7175df72366cae0e4ab780 (HEAD -> master)
Author: amy121203 <amy0169.be21@chitkara.edu.in>
Date: Thu Jun 2 23:03:18 2022 +0530

    Revert "pattern 2.1"

    This reverts commit f3c301ef51df7a72bdb15bc2d054d27fae8de264.

commit f3c301ef51df7a72bdb15bc2d054d27fae8de264
Author: amy121203 <amy0169.be21@chitkara.edu.in>
Date: Thu Jun 2 22:58:34 2022 +0530

    pattern 2.1

commit 3c37d9118ef1376102ede3179367903d7ce61f25
Author: amy121203 <amy0169.be21@chitkara.edu.in>
Date: Thu Jun 2 22:54:20 2022 +0530

    Pattern1

91628@Amy MINGW64 /d/CSE/Git/SCM Projects (master)
$ git revert 29f7d2354f8c8903eb7175df72366cae0e4ab780
```





```
MINGW64:/d/CSE/Git/SCM Projects
Revert "pattern 2.1"

This reverts commit f3c301ef51df7a72bdb15bc2d054d27fae8de264.

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Your branch is ahead of 'origin/master' by 3 commits.
#   (use "git push" to publish your local commits)
#
# Changes to be committed:
#   modified:   Pattern2.cpp
#
~
~
~
~
```

Now we want to delete the commit that we just added to the remote repository. We could have used the git reset command but that would have deleted the commit just from the local repository and not the remote repository. If we do this then we would get conflict that the remote commit is not present locally. So, we do not use git reset here. The best we can use here is git revert.

`git revert<commit id of the commit needs to be removed>`

Points to keep in mind –

- Using git revert we can undo any commit , not like git reset where we could just remove "n" recent commits.

Now let's first understand what git revert does, git revert removes the commit that we have done but adds one more commit which tells us that the revert has been done.

We can see that the new commit is being added. However since this commit is in local repository so we need to do **git push** so that our remote repository also notices that the change has been done.

And as we can see we have a new commit in our remote repository and **Hello World** which we added in our 2nd commit is being removed from the local as well as the remote repository.

### **Difference Table**

<b>git checkout</b>	<b>git reset</b>	<b>git revert</b>
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

# Project

## Problem statement

There is a problem as we have to code a C++ project but we have to make our part and send it to other participants again and again and then at end we have to combine all the part and attach those snaps which is a tough process. So we want an alternate solution for this to show changes conveniently and accurately. Also we want that if we make the changes it should reflect to all the other participants. Also the other participants can revert the changes and commit the changes.

## Objectives

**The objectives of the project are:**

- It is easy to contribute to opensource projects via GitHub.
  - It helps to create an excellent document.
  - You can attract the recruiter by showing off your work.
  - If you have a profile on GitHub, you will have a higher chance of being recruited.
- It allows your work to get out there in front of the public.

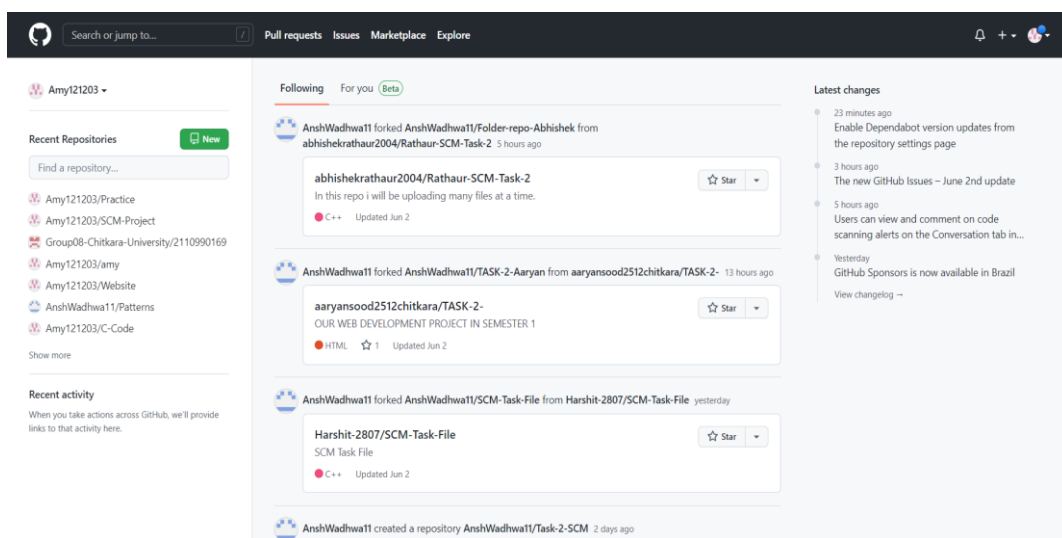


## Experiment - 8

**Aim:** Create a distributed Repository and add members in project team.

### Procedure:

- Open GitHub home page, in the right corner you can find your profile logo beside it is + icon which is used to create repositories. Create Repository from there.



- In creating repository add name and description of your choice and choose privacy of your repository and at bottom click on create repository.



Search or jump to... Pull requests Issues Marketplace Explore

### Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner <sup>\*</sup> Repository name <sup>\*</sup>

Amy121203 / SCM Projects ✓

Great repository names are [Your new repository will be created as SCM-Projects. bout miniature-goggles?](#)

Description (optional)

☒ **Public**  
Anyone on the internet can see this repository. You choose who can commit.

☐ **Private**  
You choose who can see and commit to this repository.

Initialize this repository with:

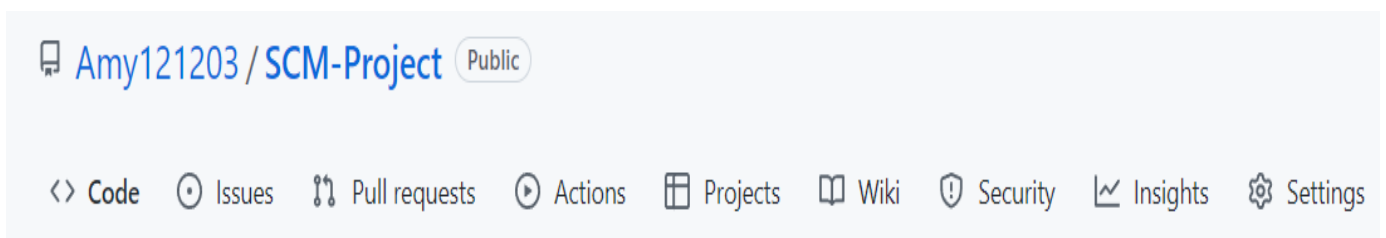
Skip this step if you're importing an existing repository.

☐ **Add a README file**  
This is where you can write a long description for your project. [Learn more.](#)

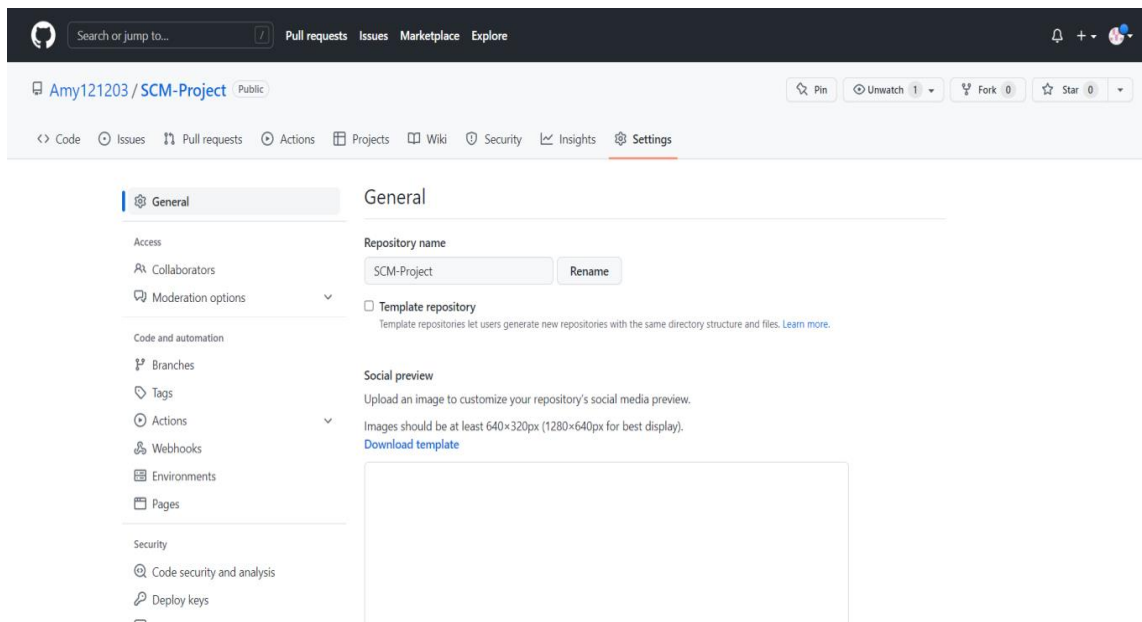
**Add .gitignore**  
Choose which files not to track from a list of templates. [Learn more.](#)

[Choose a template: More](#)

- Adding Members- To add members in your repo go to settings in your repo navigation bar.

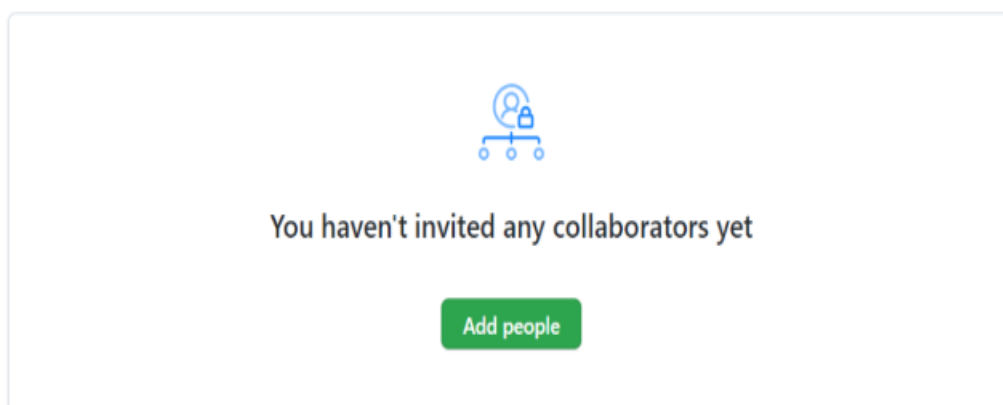


- A settings page will appear. Here, into the left-sidebar click into the Collaborators.

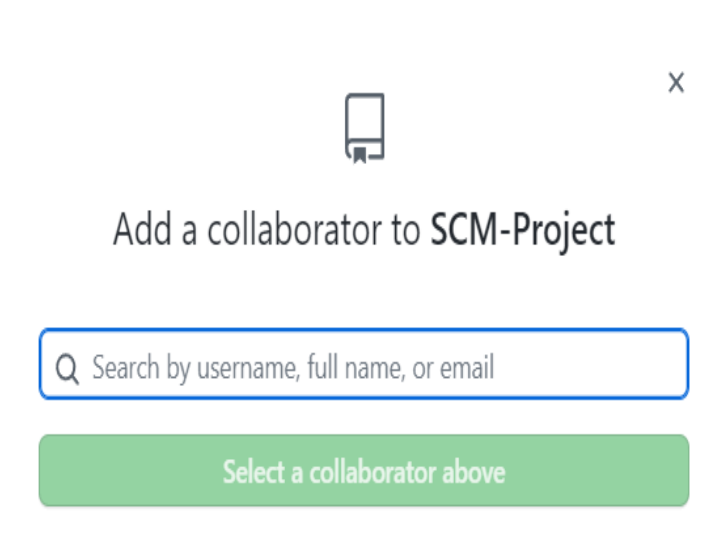


- Then a confirm password page may appear, enter your password for the confirmation.
- Next, click into Add People

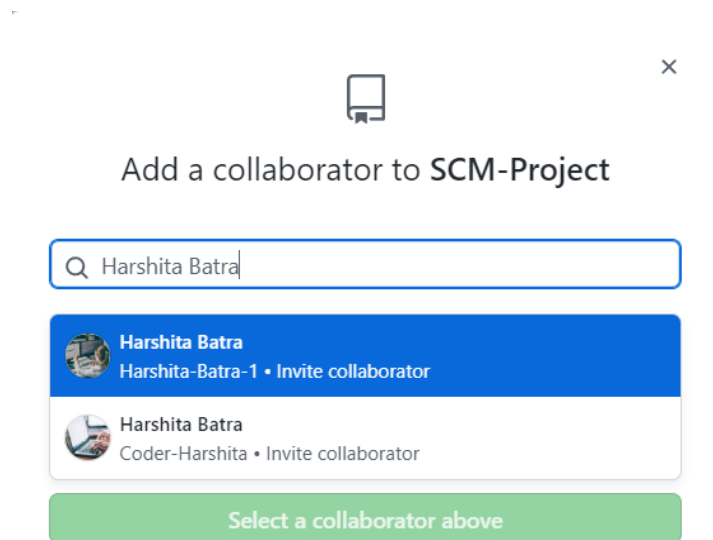
## Manage access



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



- After selecting the people, add them as collaborator.



- After sending the request for collaboration to that person whom we wanted as our collaborator for our project will get a email from the Team leader (by GitHub). He/she has to open its Email to view the invitation.
- We are done adding a multiple collaborator. Now, they will get a mail regarding invitation to your repository. Once they accept, they will



have collaborator access to your repository. Till then it will be in pending invitation state. You can also add more collaborator and delete the existing one as depicted below.

## Manage access

Add people

☐ Select all

Type ▼

Q Find a collaborator...

☐



**anushkam012**

Collaborator

Remove

☐



**AreenSiwach**

Collaborator

Remove

☐



**Harshita Batra**

Harshita-Batra-1 • Collaborator

Remove



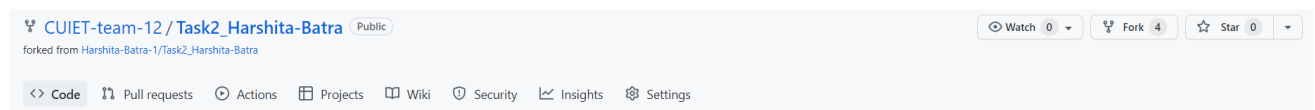


## Experiment - 11

**Aim:** Open and Close a Pull Request

### Forking Repositories:

- Open the repo which you want to fork then find fork button in right side of the navigation panel of repo.



- Click on fork.
- Then change the name if you wish to else you can continue with previous name and then click create fork.

### Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. [View existing forks.](#)

Owner \*

CUIET-team-12

Repository name \*

Task2\_Harshita-Batra

By default, forks are named the same as their parent repository. You can customize the name to distinguish it further.

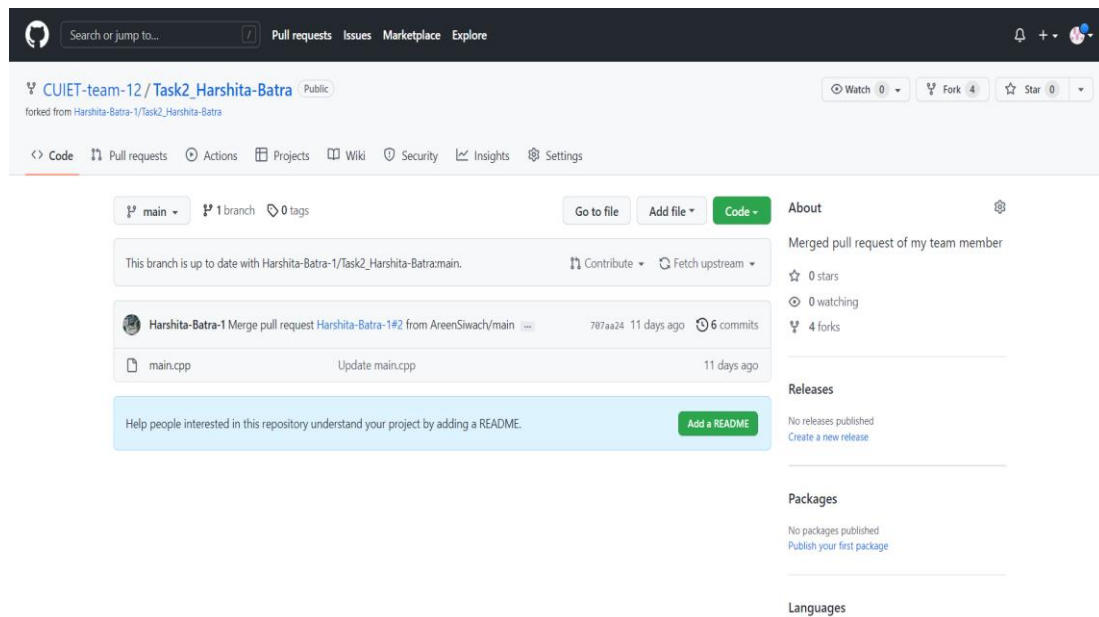
Description (optional)

Merged pull request of my team member

You are creating a fork in the CUIET-team-12 organization.

Create fork

- That's it, copy of the repository is created as your repository now make changes and commit them in your local copy of repository.



## Opening Pull Request:

To open a pull request we first have to make a new branch, by using git branch name.

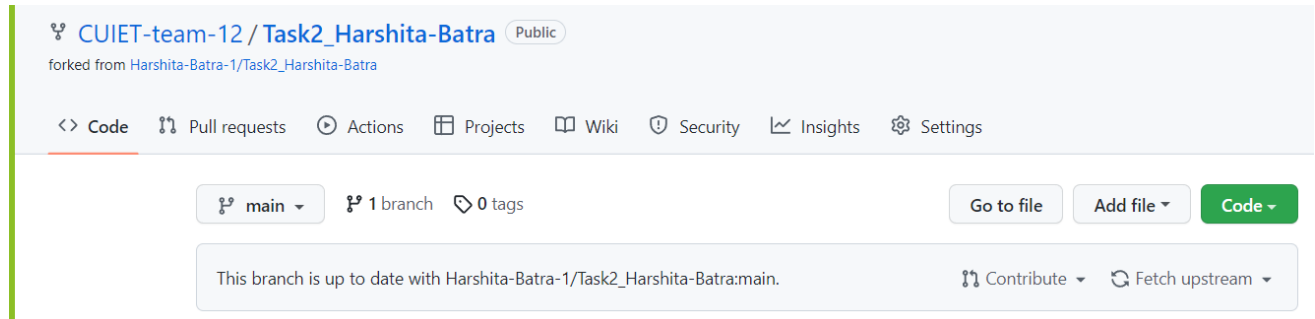
```
91628@Amy MINGW64 /d/CSE/Git/bash (master)
$ git branch main

91628@Amy MINGW64 /d/CSE/Git/bash (master)
$ git branch
main
* master

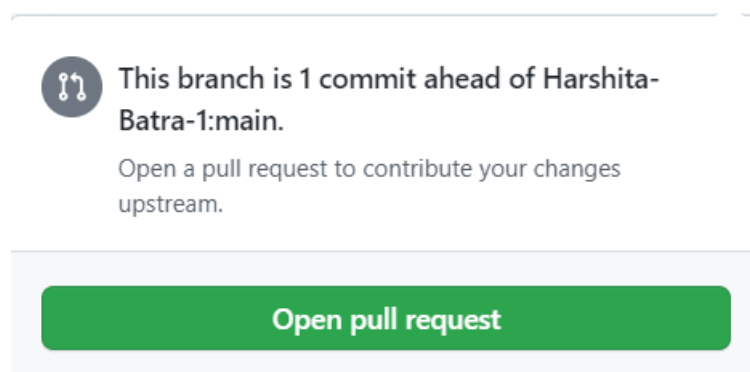
91628@Amy MINGW64 /d/CSE/Git/bash (master)
$ |
```

- After making new branch we add a file to the branch or make changes in the existing file.
- Add and commit the changes to the local repository.
- Use git push origin branch name option to push the new branch to the main repository.
- After you have committed the changes in your forked repository, now you are ready to open a pull request to send changes in the original base file.

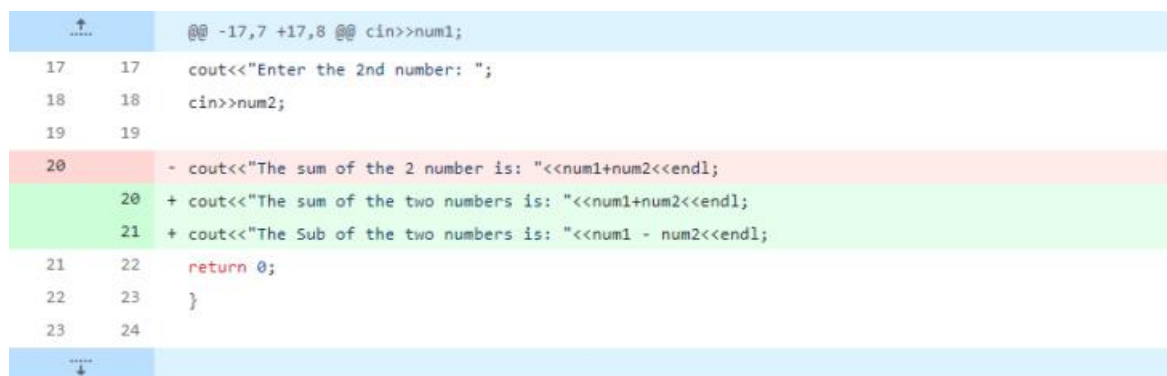
- You will be able to find a dialogue box above the list of files committed in your github.com page.

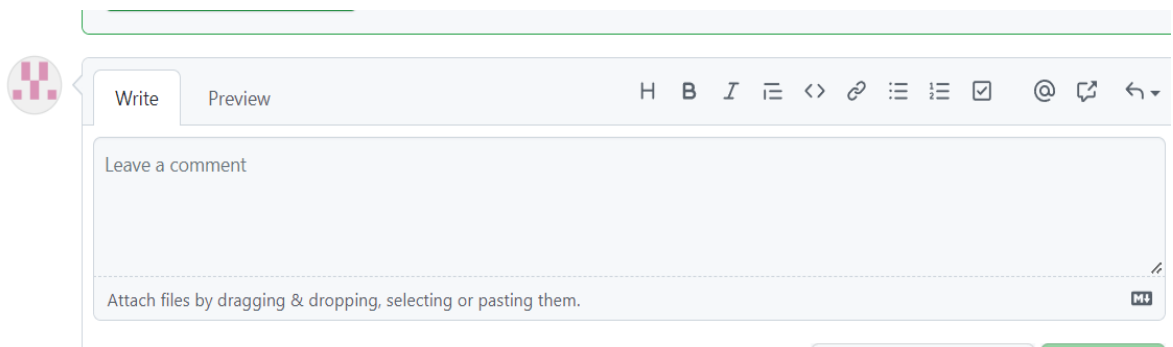


- After clicking on contribute dialogue box showing open pull request button will appear, click on open pull request.



- Click again on create pull request, drop down will appear asking for description of changes you made, type description there and click pull request at bottom.

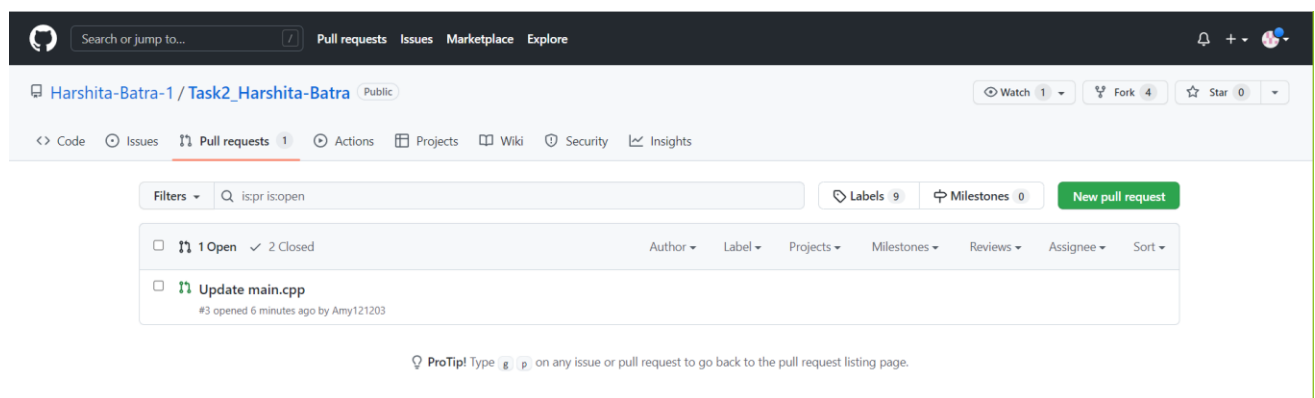




- Pull request is opened successfully, now the owner chooses either to merge or delete the request.

## Closing Pull Request:

- Open your repository and click on “Pull Requests” on the repo navigation bar.



- Click on the name of the Pull Requests to either merge or delete the request. There you can find Merge Pull Request button with drop down button beside it, using it you can if you wish to delete the request.



## Update main.cpp #3

Open Amy121203 wants to merge 1 commit into Harshita-Batra-1:main from CUIET-team-12:main

Conversation 0 Commits 1 Checks 0 Files changed 1



Amy121203 commented 7 minutes ago

Collaborator

I have added a commit.

Update main.cpp

Verified 8e4d4e9

Add more commits by pushing to the `main` branch on CUIET-team-12/Task2\_Harshita-Batra.



This branch has no conflicts with the base branch  
Merging can be performed automatically.

Merge pull request

You can also open this in [GitHub Desktop](#) or view [command line instructions](#).

- After clicking on Merge pull request button confirmation will be asked there you just need to confirm by clicking “Confirm Merge Button”.



## Experiment - 12

**Aim:** Create a pull request on a team member's repo and close pull requests generated by team members on own Repo as a maintainer.

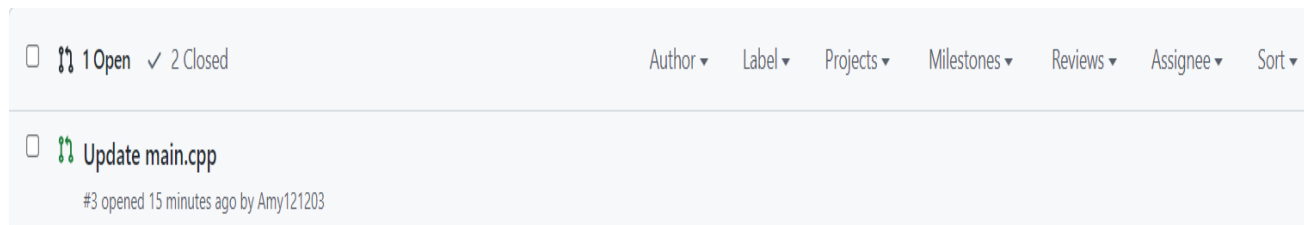
Do the required changes in the repository, add and commit these changes in the local repository in a new branch.

- Push the modified branch using git push origin branch name.
- Open a pull request by following the procedure from the above experiment.
- The pull request will be created and will be visible to all the team members.
- Ask your team member to login to his/her GitHub account.
- They will notice a new notification in the pull request menu.

To create a pull request on a team member's repository and close requests by any other team members as a maintainer follow the procedure given below:-

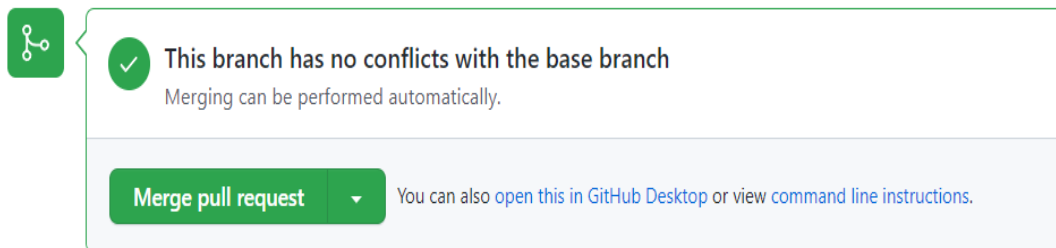


- Click on it. The pull request generated by you will be visible to them.

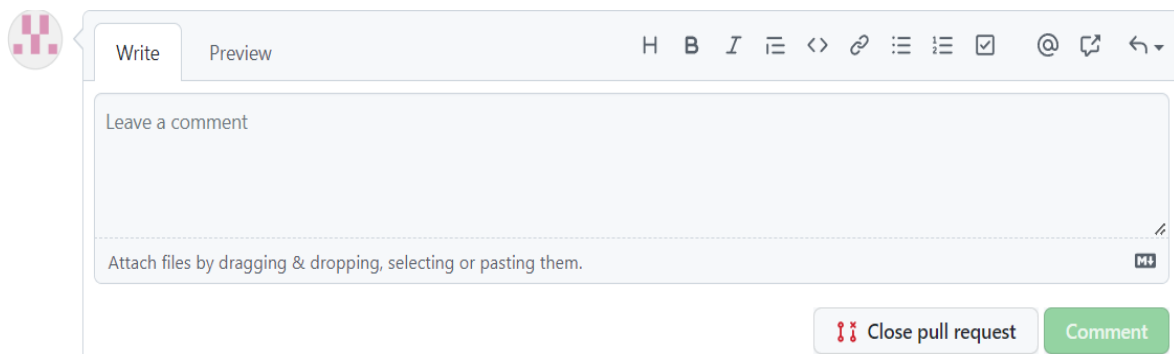


- Click on the pull request. Two options will be available, either to close the pull request or Merge the request with the main branch.

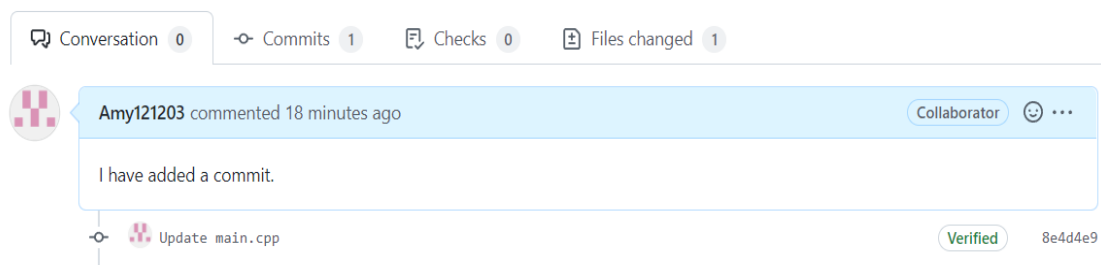
- By selecting the merge branch option the main branch will get updated for all the team members.



- By selecting close the pull request the pull request is not accepted and not merged with main branch.



- The process is similar to closing and merging the pull request by you. It simply includes an external party to execute.
- The result of merging the pull request is shown below



- Thus, we conclude opening and closing of pull request. We also conclude merging of the pull request to the main branch.

## Experiment – 13

**Aim:** Publish and print network graphs.

The network graph is one of the useful features for developers on GitHub. It is used to display the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.

A repository's graphs give you information on traffic, projects that depend on the repository, contributors and commits to the repository, and a repository's forks and network. If you maintain a repository, you can use this data to get a better understanding of who's using your repository and why they're using it.

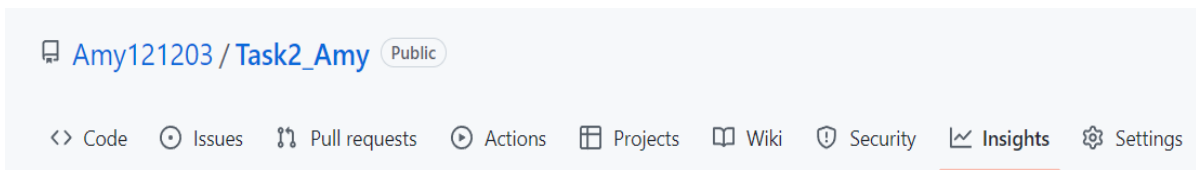
Some repository graphs are available only in public repositories with GitHub Free:

- Pulse
- Contributors
- Traffic
- Commits
- Code frequency
- Network

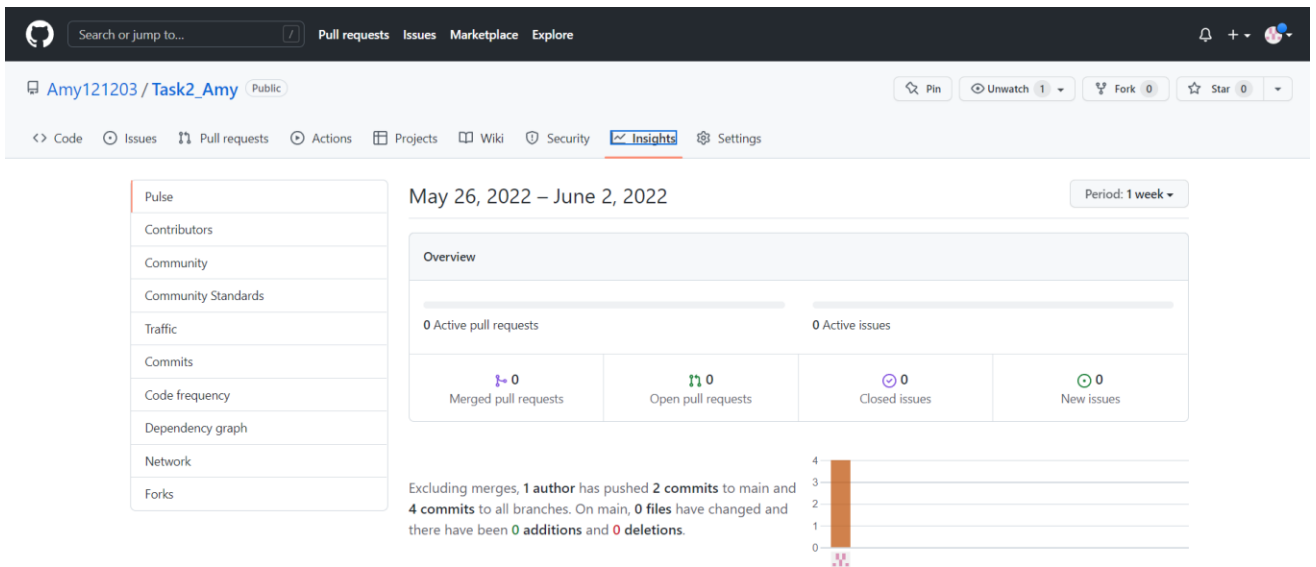
### **Steps to access network graphs of respective repository:**

1. On GitHub.com, navigate to the main page of the repository.
2. Under your repository name, click Insights.





3. At the left sidebar, click on Network.



4. You will get the network graph of your repository which displays the branch history of the entire repository network, including branches of the root repository and branches of forks that contain commits unique to the network.

