VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"Jnana Sangama", Machhe, Belagavi, Karnataka-590018

Lab Experiment Record

Project Management with Git [BCSL58C]

Submitted in partial fulfillment towards AEC of 3rd semester of

Bachelor of Engineering
in
Computer Science and Engineering
(Artificial Intelligence & Machine Learning)

Submitted by

B.Tanmayi
4GW24CI005

DEPARTMENT OF CSE (Artificial Intelligence & Machine Learning)

GSSS INSTITUTE OF ENGINEERING & TECHNOLOGY FOR WOMEN

(Affiliated to VTU, Belagavi, Approved by AICTE, New Delhi & Govt. of Karnataka)

K.R.S ROAD, METAGALLI, MYSURU-570016, KARNATAKA

(Accredited by

NAAC)

2025-2026

BASICS

1.git config

This command is used to customize the Git environment, typically defining the user's identity for associating commits. The --global flag ensures the setting applies to all local repositories.

2.git init

Initializes a new local Git repository.

This provides an overview of which files are modified, staged, or untracked, helping the user manage their changes effectively.

3.Tracking and Committing Changes

These commands manage the movement of files between the working directory, the staging area, and the local repository history.

*git add <file> :- Moves changes from the working directory to the staging area.

*git commit -m "msg" -Records staged changes as a permanent snapshot in the repository history

This creates a new, unique commit object with a descriptive message, effectively creating a specific version of the project at that point in time

4.git status →how the state of the working directory and staging area.

This provides an overview of which files are modified, staged, or untracked, helping the user manage their changes effectively.

5.git clone <URL> →Creates a complete local copy of an existing remote repository.

6. git push → Uploads local commits to a remote repository.

This shares local changes with collaborators and updates the remote branch with the latest local progress .

7. git pull → Fetches changes from a remote repository and immediately merges them into the current local branch.

Experiment 1 :
Setting Up and Basic Commands

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Solution:

To initialize a new Git repository in a directory, create a new file, add it to the staging area, and commit the changes with an appropriate commit message, follow these steps:

1.Open your Git Bash and navigate to the directory where you want to create a git repository.(You can create a new folder andopen that folder and right click and select a option open git Bash here).

2.You can set the global configuration with these git commands:

$ git config --global user.name "Your Name"

$ git config --global user.email "your.email@gmail.com"

```
B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$ git config --global user.name "BTanmayi30"

B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$ git config --global user.email "btanmayi.1234@gmail.com"
```

3. Initialize a new Git repository in that directory:

$ git init

4. Next,create a new file in the directory. For example, let's create a file named "git_lab.docx" You can use any text editor or command-line tools to create the file.

5. Add the newly created file to the staging area ,with the following git command.

```
B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$ git add git_lab.docx
```

This command stages the file for the upcoming commit.

6. Commit the changes with an appropriate commit message.

$ git commit -m "Your commit message here"

```
B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$ git commit -m "practice for git_lab"
[master (root-commit) a8cf20b] practice for git_lab
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git_lab.docx

B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$
```

Your commit message should briefly describe the purpose or nature of the changes you made.For example: $ git commit -m "Add a new file called git_lab.docx " After these steps, your changes will be committed to the Git repository with the provided commit message. You now have a version of the repository with the new file and its history stored in Git.

4.Make your changes in the "feature-branch" by adding, modifying, or deleting files as needed.

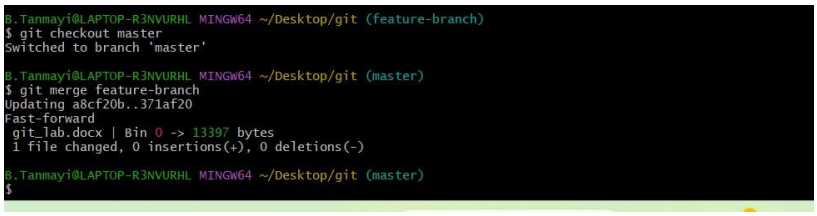
Stage and commit your changes in the "feature-branch": Replace "Your commit message for feature-branch" with a descriptive commit message for the changes you made in the "feature-branch."

5. Switch back to the "master" branch:

   $ git checkout master

6. Merge the "feature-branch" into the "master" branch:

   $ git merge feature-branch



This command will incorporate the changes from the "feature-branch" into the "master" branch. Now, your changes from the "feature-branch" have been merged into the "master" branch. Your project's history will reflect the changes made in both branches .

This experiment helps the learner understand the complete lifecycle of initializing a Git repository, from configuring user identity to committing the first version of a project. It establishes the foundation for version control by demonstrating how Git tracks changes, maintains history, and enables structured development. This experiment is crucial as it introduces best practices such as meaningful commit messages and proper file staging, which are essential for collaboration and long-term project maintenance.

Experiment 2.

Creating and Managing Branches:

Solution:

To create a new branch named "feature-branch," switch to the "master" branch, and merge the "feature-branch" into "master" in Git, follow these steps:

1. Make sure you are in the "master" branch by switching to it:
   $ git checkout master

2. Create a new branch named "feature-branch" and switch to it:
   $ git checkout -b feature-branch
   This command will create a new branch called "feature-branch" and switch to it.
   $ git add .

   $ git commit -m "Your commit message for feature-branch"

3. Make your changes in the "feature-branch" by adding, modifying, or deleting files as needed.

4. Stage and commit your changes in the "feature-branch": Replace "Your commit message for feature-branch" with a descriptive commit message for the changes you made in the "feature-branch."

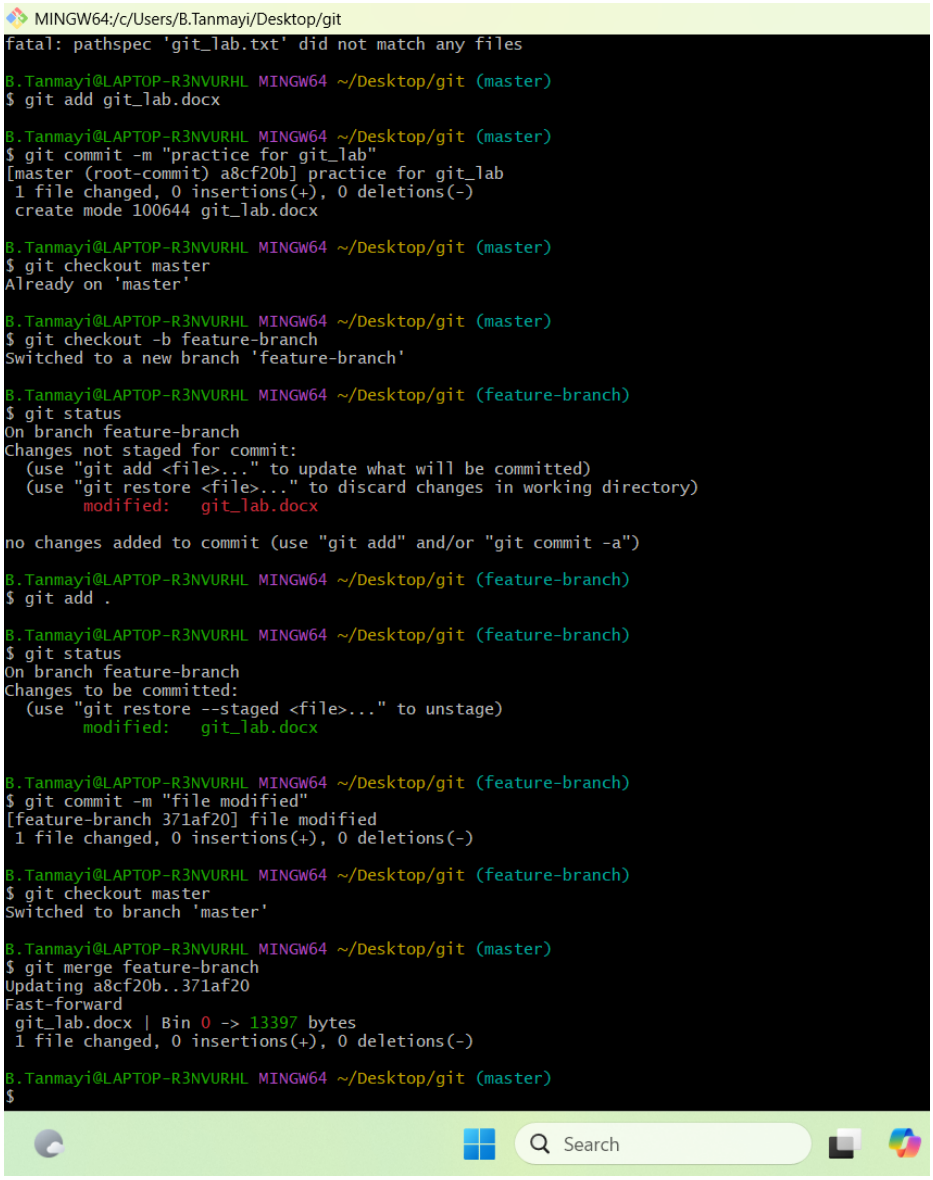
5. Switch back to the "master" branch:

$ git checkout master

Merge the "feature-branch" into the "master" branch:
$ git merge feature-branch
This command will incorporate the changes from the "feature-branch" into the "master" branch. Now, your changes from the "feature-branch" have been merged into the "master" branch. Your project's history will reflect the changes made in both branches.

This experiment emphasizes the importance of branching in Git for parallel development. By creating and merging branches, users learn how teams can work on new features independently without affecting the main codebase. It also highlights how Git preserves commit history during merges, ensuring that development

remains organized, traceable, and conflict-free when managed correctly.



Experiment 3.

Creating and Managing Branches:

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

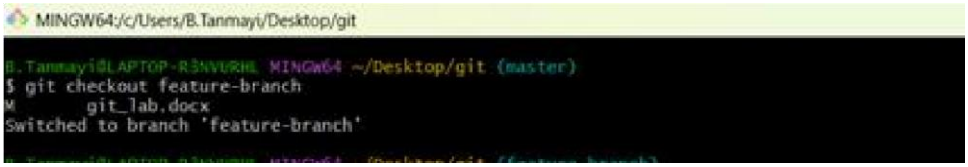
Solution: To stash your changes, switch branches, and then apply the stashed changes in Git, you can use the following commands:

1. Stash your changes:
   $ git stash save "Your stash message"

This command will save your changes in a stash, which acts like a temporary storage for changes that are not ready to be committed.

2. Switch to the desired branch:

   $ git checkout target-branch



Replace "target-branch" with the name of the branch you want to switch to.

3. Apply the stashed changes: $ git stash apply

This command will apply the most recent stash to your current working branch.

If you have multiple stashes, you can specify a stash by name or reference (e.g., git stash apply stash@{2}) if needed. If you want to remove the stash after applying it, you can use git stash pop instead of git stash apply. Remember to replace "Your stash message" and "target-branch" with the actual message you want for your stash and the name of the branch you want to switch to. Through this experiment, learners gain insight into temporary change management using Git stash. It is especially useful in real-world scenarios where urgent context switching is required. This experiment teaches how to safely store uncommitted changes, switch tasks efficiently, and restore work without losing progress, thereby improving workflow flexibility.

Experiment 4

Collaboration and Remote Repositories:

Clone a remote Git repository to your local machine.

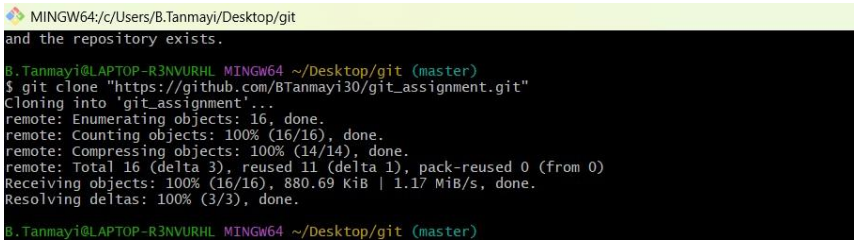
Solution:

To clone a remote Git repository to your local machine, follow these steps:

1. Open your terminal or command prompt.

2. Navigate to the directory where you want to clone the remote Git repository. You can use the cd command to change your working directory.

3. Use the git clone command to clone the remote repository. Replace with the URL of the remote Git repository you want to clone. For example, if you were cloning a repository from GitHub, the URL might look like this: $ git clone



Here's a full example: $ git clone https://github.com/username/repo-name.git Replace https://github.com/username/repo-name.git with the actual URL of the repository you want to clone.

4.Git will clone the repository to your local machine. Once the process is complete, you will have a local copy of the remote repository in your chosen directory. You can now work with the cloned repository on your local machine, make changes, and push those changes back to the remote repository as needed.

This experiment introduces the concept of distributed version control by cloning a remote repository. It explains how developers can obtain a complete project history on their local systems, enabling offline development and experimentation. Understanding repository cloning is essential for contributing to open-source projects and team-based software developm

Experiment 5

Collaboration and Remote Repositories:

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Solution:

To fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch in Git, follow these steps:

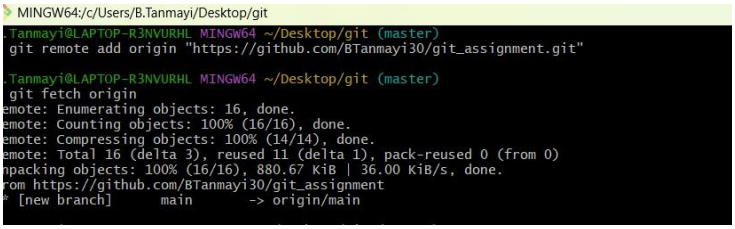
1. Open your terminal or command prompt.

2. Make sure you are in the local branch that you want to rebase. You can switch to the branch using the following command, replacing with your actual branch name:

$ git checkout

3. Fetch the latest changes from the remote repository. This will update your local repository with the changes from the remote without merging them into your local branch:

$ git fetch origin



$ git rebase origin/

Here, origin is the default name for the remote repository.

If you have multiple remotes, replace origin with the name of the specific remote you want to fetch from.

4. Once you have fetched the latest changes, rebase your local branch onto the updated remote branch: Replace with the name of the remote branch you want to rebase onto.This command will reapply your local commits on top of the latest changes from the remote branch, effectively incorporating the remote changes into your branch history.

5. Resolve any conflicts that may arise during the rebase process. Git will stop and notify you if there are conflicts that need to be resolved. Use a text editor to edit the conflicting files, save the changes, and then continue the rebase with: $ git rebase --continue

6. After resolving any conflicts and completing the rebase, you have successfully updated your local branch with the latest changes from the remote branch.

7. If you want to push your rebased changes to the remote repository, use the git push command. However, be cautious when pushing to a shared remote branch, as it can potentially overwrite other developers' changes: $ git push origin

Replace with the name of your local branch.

By following these steps, you can keep your local branch up to date with the latest changes from the remote repository and maintain a clean and linear history through rebasing.

CONCLUSION:

This experiment focuses on synchronizing local and remote repositories using fetch and rebase operations. It highlights the advantage of maintaining a clean and linear commit history while integrating upstream changes. Learners also understand conflict resolution during rebasing, which is a critical skill in collaborative environments.

Experiment 6.

Collaboration and Remote Repositories:

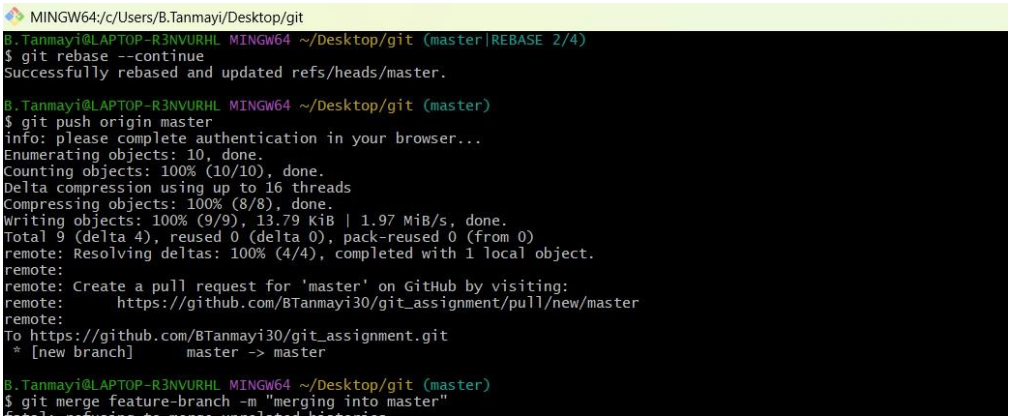
Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.

Solution:

To merge the "feature-branch" into "master" in Git while providing a custom commit message for the merge, you can use the following command:

$ git checkout master

$ git merge feature-branch -m "Your custom commit message here"



Replace "Your custom commit message here" with a meaningful and descriptive commit message for the merge. This message will be associated with the merge commit that is created when you merge "feature-branch" into "master."

Experiment 7.

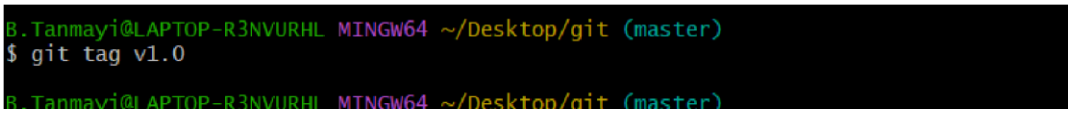
Git Tags and Releases:

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

Solution:

To create a lightweight Git tag named "v1.0" for a commit in your local repository, you can use the following command:

$ git tag v1.0



This command will create a lightweight tag called "v1.0" for the most recent commit in your current branch. If you want to tag a specific commit other than the most recent one, you can specify the commit's SHA-1 hash after the tag name. For example:

$ git tag v1.0 <commit-Id>

This experiment introduces Git tags as a method of marking significant points in a project's history. By creating lightweight tags, learners understand how releases and versions are identified, making it easier to manage deployments, rollbacks, and long-term maintenance of software projects.

Experiment 8.

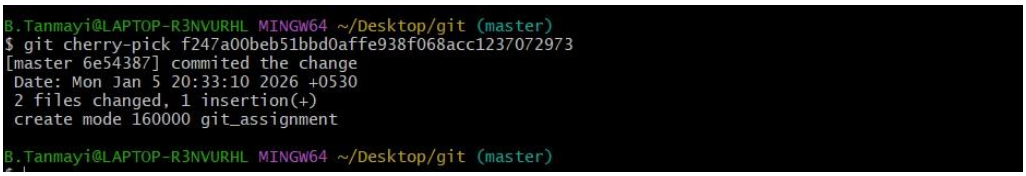
Advanced Git Operations:

Write the command to cherry-pick a range of commits from "source-branch" to the current branch.

Solution:

To cherry-pick a range of commits from "source-branch" to the current branch, you can use the following command:

$ git cherry-pick <start-commit>^..<end-commit>



Replace with the commit at the beginning of the range, and with the commit at the end of the range. The ^ symbol is used to exclude the itself and include all commits after it up to and including . This will apply the changes from the specified range of commits to your current branch. For example, if you want to cherry-pick a range of commits from "source-branch" starting from commit ABC123 and ending at commit DEF456, you would use:

$ git cherry-pick ABC123^..DEF456

Make sure you are on the branch where you want to apply these changes before

Cherry-picking allows selective integration of commits across branches. This experiment demonstrates how specific features or fixes can be applied without merging entire branches. It is especially valuable in bug-fixing and release management scenarios where precision and control are required.

Experiment 9.

Analysing and Changing Git History:

Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message? Solution:

To view the details of a specific commit, including the author, date, and commit message, you can use the git show or git log command with the commit ID. Here are both options:

1. Using git show:
   bash
   git show <commit-id>

Replace with the actual commit ID you want to view. This command will display detailed information about the specified commit, including the commit message, author, date, and the changes introduced by that commit. For example:
$ git show abc123 2.
Using git log:
$ git log -n 1

For example: The -n 1 option tells Git to show only one commit. Replace with the actual commit ID.

This command will display a condensed view of the specified commit, including its commit message, author, date, and commit ID. $ git log-n 1 abc123 Both of these commands will provide you with the necessary information about the specific commit you're interested in.

Experiment 10.

Analysing and Changing Git History

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."

Solution: To list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31" in Git, you can use the git log command with the --author and --since and --until options. Here's the command:

$ git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"

```
B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
$ git log --author="BTanmayi30" --since="2025-12-01" --until="2026-01-05"
commit f247a00beb51bbd0affe938f068acc1237072973 (HEAD -> master, tag: v1.0, origin/master)
Author: BTanmayi30 <btanmayi.1234@gmail.com>
Date:   Mon Jan 5 20:33:10 2026 +0530

    commited the change

commit 3017ff9ed1e5eb3ce2595aca445e5fa12f5cd95e
Author: BTanmayi30 <btanmayi.1234@gmail.com>
Date:   Mon Jan 5 18:32:15 2026 +0530

    file modified

commit baab472836ba448ecbfc3ec0a11bea2221fb110e
Author: BTanmayi30 <btanmayi.1234@gmail.com>
Date:   Mon Jan 5 18:27:25 2026 +0530

    practice for git_lab

B.Tanmayi@LAPTOP-R3NVURHL MINGW64 ~/Desktop/git (master)
```

This command will display a list of commits made by the author "JohnDoe" that fall within the specified date range, from January 1, 2023, to December 31, 2023. Make sure to adjust the author name and date range as needed for your specific use case.

Experiment 11.

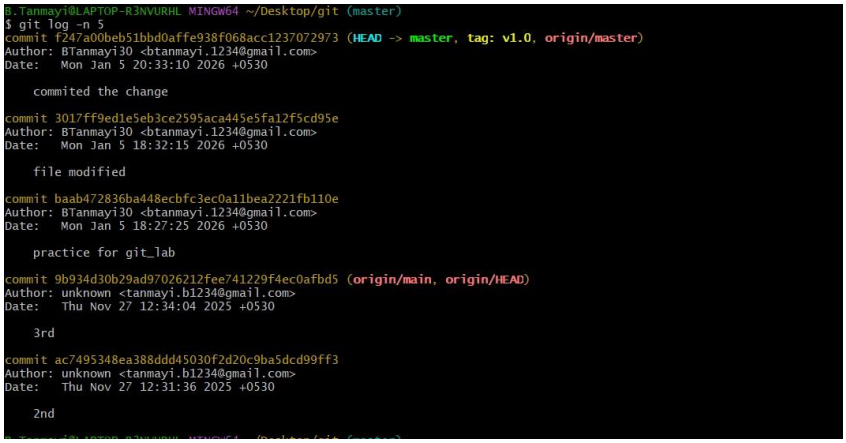
Analysing and Changing Git History

Write the command to display the last five commits in the repository's history.

Solution:

To display the last five commits in a Git repository's history, you can use the git log command with the -n option, which limits the number of displayed commits. Here's the command:

$ git log -n 5



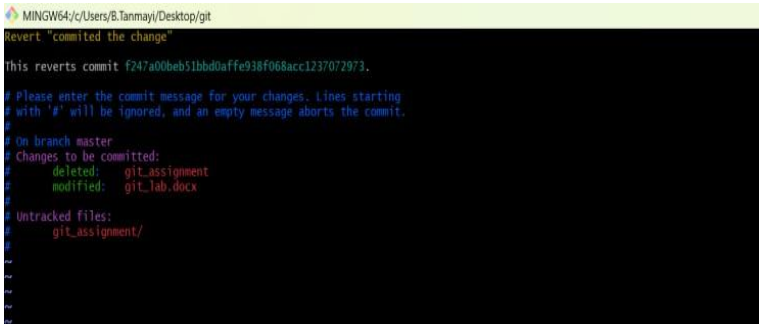
This command will show the last five commits in the repository's history. You can adjust the number after -n to display a different number of commits if needed

Experiment 12.

Analysing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".
Solution: To undo the changes introduced by a specific commit with the ID "abc123" in Git, you can use the git revert command. The git revert command creates a new commit that undoes the changes made by the specified commit, effectively "reverting" the commit. Here's the command:

$ git revert abc123



Replace "abc123" with the actual commit ID that you want to revert. After running this command, Git will create a new commit that negates the changes introduced by the specified commit. This is a safe way to undo changes in Git because it preserves the commit history and creates a new commit to record the reversal of the changes.

This experiment explains safe methods for undoing changes using Git revert. It reinforces the importance of preserving commit history while correcting mistakes. Learners understand why revert is preferred in shared repositories, as it ensures stability, traceability, and collaboration safety.

REPO LINK: “https://github.com/BTanmayi30/4GW24CI005.git”