VISVESVARAYATECHNOLOGICALUNIVERSITY

“JnanaSangama”,Machhe,Belagavi,Karnataka-590018

Lab Experiment Record

Project Management with Git [BCSL358C]

SubmittedinpartialfulfillmenttowardsAECof3rd semesterof

Bachelorof Engineering
in
ComputerScienceandEngineering
(ArtificialIntelligence&MachineLearning)

Submittedby

ACHALA C
4GW24CI001

DEPARTMENTOF CSE(ArtificialIntelligence&MachineLearning)

GSSSINSTITUTEOFENGINEERING&TECHNOLOGYFORWOMEN

(AffiliatedtoVTU,Belagavi,ApprovedbyAICTE,NewDelhi&Govt.ofKarnataka)

K.R.SROAD,METAGALLI,MYSURU-570016,KARNATAKA

(AccreditedbyNAAC)

2025-2026

INDEX

Project Management with Git		Semester	3
Course Code	BCS358C	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	0: 0 : 2: 0	SEE Marks	50
Credits	01	Exam Marks	100
Examination type (SEE)	Practical		

Course objectives:

- .To familiar with basic command of Git
- To create and manage branches
- To understand how to collaborate and work with Remote Repositories
- To familiar with virion controlling commands

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.

2. Creating and Managing Branches

Create a new branch named "feature-branch." Switch to the "master" branch. Merge the"feature-branch" into "master."

3. Creating and Managing Branches

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

4. Collaboration and Remote Repositories

Clone a remote Git repository to your local machine.

5. Collaboration and Remote Repositories

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

6. Collaboration and Remote Repositories

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

7. Git Tags and Releases

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

8. Advanced Git Operations

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

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?

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."

11. Analysing and Changing Git History

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

12. Analysing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".

Git Basics

What is Git?

Git is a distributed version control system (VCS) that is widely used for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control in the software development industry. Git allows multiple developers to collaborate on a project by providing a history of changes, facilitating the tracking of who made what changes and when. Here are some key concepts and features of Git:

1. Repository (Repo):
   A Git repository is a directory or storage location where your project's files and version history are stored. There can be a local repository on your computer and remote repositories on servers.
2. Commits:
   In Git, a commit is a snapshot of your project at a particular point in time. Each commit includes a unique identifier, a message describing the changes, and a reference to the previous commit.
3. Branches:
   Branches in Git allow you to work on different features or parts of your project simultaneously without affecting the main development line (usually called the "master" branch). Branches make it easy to experiment, develop new features, and merge changes back into the main branch when they are ready.
4. Pull Requests (PRs):
   In Git-based collaboration workflows, such as GitHub or GitLab, pull requests are a way for developers to propose changes and have them reviewed by their peers. This is a common practice for open-source and team-based projects.
5. Merging:
   Merging involves combining changes from one branch (or multiple branches) into another. When a branch's changes are ready to be incorporated into the main branch, you can merge them.
6. Remote Repositories:
   Remote repositories are copies of your project stored on a different server. Developers can collaborate by pushing their changes to a remote repository and pulling changes from it. Common remote repository hosting services include GitHub, GitLab, and Bitbucket.
7. Cloning:
   Cloning is the process of creating a copy of a remote repository on your local machine. This allows you to work on the project and make changes locally.
8. Forking:
   Forking is a way to create your copy of a repository, typically on a hosting platform like GitHub. You can make changes to your fork without affecting the original project and later create pull requests to contribute your changes back to the original repository.

Git is known for its efficiency, flexibility, and ability to handle both small and large-scale software projects. It is used not only for software development but also for managing and tracking

changes in various types of text-based files, including documentation and configuration files. Learning Git is essential for modern software development and collaboration.

Why do we need git?

Git is an essential tool in software development and for many other collaborative and versioncontrolled tasks. Here are some key reasons why Git is crucial:

1. **Version Control**: Git allows you to track changes in your project's files over time. It provides a complete history of all changes, making it easy to understand what was done, when it was done, and who made the changes. This is invaluable for debugging, auditing, and collaboration.
2. **Collaboration**: Git enables multiple developers to work on the same project simultaneously without interfering with each other's work. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.
3. **Branching**: Git supports branching, which allows developers to create isolated environments for developing new features or fixing bugs. This is essential for managing complex software projects and experimenting with new ideas without affecting the main codebase.
4. **Distributed Development**: Git is a distributed version control system, meaning that every developer has a complete copy of the project's history on their local machine. This provides redundancy, facilitates offline work, and reduces the reliance on a central server.
5. **Backup and Recovery**: With Git, your project's history is distributed across multiple locations, including local and remote repositories. This provides redundancy and makes it easy to recover from accidental data loss or system failures.
6. **Code Review**: Git-based platforms like GitHub, GitLab, and Bitbucket provide tools for code review and collaboration. Developers can propose changes, comment on code, and discuss improvements, making it easier to maintain code quality.
7. **Open Source and Community Development**: Git has become the standard for open source software development. It allows anyone to fork a project, make contributions, and create pull requests, which makes it easy for communities of developers to collaborate on a single codebase.
8. **Efficiency**: Git is designed to be fast and efficient. It only stores the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.
9. **History and Documentation**: Git's commit history and commit messages serve as a form of documentation. It's easier to understand the context and reasoning behind a change by looking at the commit history and associated messages.
10. **Customizability**: Git is highly configurable and extensible. You can set up hooks and scripts to automate workflows, enforce coding standards, and integrate with various tools.

In summary, Git is essential for tracking changes in your projects, facilitating collaboration among developers, and ensuring the integrity and version history of your code. Whether you're working on a personal project or as part of a large team, Git is a fundamental tool for modern software development and version control.

What is Version Control System (VCS)?

A Version Control System (VCS), also commonly referred to as a Source Code Management (SCM) system, is a software tool or system that helps manage and track changes to files and

directories over time. The primary purpose of a VCS is to keep a historical record of all changes made to a set of files, allowing multiple people to collaborate on a project while maintaining the integrity of the codebase. There are two main types of VCS: centralized and distributed.

Centralized Version Control Systems (CVCS): In a CVCS, there is a single central repository that stores all the project files and their version history. Developers check out files from this central repository, make changes, and then commit those changes back to the central repository. Examples of CVCS include CVS (Concurrent Versions System) and Subversion (SVN).

Distributed Version Control Systems (DVCS): In a DVCS, every developer has a complete copy of the project's repository, including its full history, on their local machine. This allows developers to work independently, create branches for experimentation, and synchronize their changes with remote repositories. Git is the most well-known and widely used DVCS, but other DVCS options include Mercurial and Bazaar.

Key features and benefits of Version Control Systems include:
1. History Tracking: VCS systems maintain a complete history of changes, including who made the change, what was changed, and when it was changed. This makes it easy to review and understand the evolution of a project.
2. Collaboration: VCS allows multiple developers to work on the same project simultaneously. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.
3. Branching and Isolation: VCS systems support branching, allowing developers to create isolated environments for new features or bug fixes. This isolates changes and helps manage complex development tasks.
4. Revert and Rollback: If a mistake is made, it is possible to revert changes to a previous state or commit. This is essential for error correction and maintaining code quality.
5. Backup and Recovery: Project data is stored in multiple locations, providing redundancy and facilitating data recovery in case of accidental data loss or system failures.
6. Documentation: Commit messages and history serve as a form of documentation, explaining why a change was made, who made it, and when it was made.
7. Efficiency: VCS systems are designed to be fast and efficient. They typically store only the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.

VCS is a fundamental tool in software development and is used not only for source code but also for tracking changes in documentation, configuration files, and other types of text-based files. It is especially crucial for collaborative projects, allowing teams of developers to work together on the same codebase with confidence.

Git Life Cycle

The Git lifecycle refers to the typical sequence of actions and steps you take when using Git to manage your source code and collaborate with others. Here's an overview of the Git lifecycle:

1. Initializing a Repository:

To start using Git, you typically initialize a new repository (or repo) in your project directory. This is done with the command git init.

2. Working Directory:

Your project files exist in the working directory. These are the files you are actively working on.

3. Staging:

Before you commit changes, you need to stage them. Staging allows you to select which changes you want to include in the next commit. You use the git add command to stage changes selectively or all at once with git add ..

4. Committing:

After you've staged your changes, you commit them with a message explaining what you've done. Commits create snapshots of your project at that point in time. You use the git commit command to make commits, like git commit -m "Add new feature".

5. Local Repository:

Commits are stored in your local repository. Your project's version history is preserved there.

6. Branching:

Git encourages branching for development. You can create branches to work on new features, bug fixes, or experiments without affecting the main codebase. Use the git branch and git checkout commands for branching.

7. Merging:

After you've completed work in a branch and want to integrate it into the main codebase, you perform a merge. Merging combines the changes from one branch into another. Use the git merge command.

8. Remote Repository:

For collaboration, you can work with remote repositories hosted on servers like GitHub, GitLab, or Bitbucket. These repositories serve as a central hub for sharing code.

9. Pushing:

To share your local commits with a remote repository, you push them using the git push command. This updates the remote repository with your changes.

10. Pulling:

To get changes made by others in the remote repository, you pull them to your local repository with the git pull command. This ensures that your local copy is up to date.

11. Conflict Resolution:

Conflicts can occur when multiple people make changes to the same part of a file. Git will inform you of conflicts, and you must resolve them by editing the affected files manually.

12. Collaboration:

Developers can collaborate by pushing, pulling, and making pull requests in a shared remote repository. Collaboration tools like pull requests are commonly used on platforms like GitHub and GitLab.

13. Tagging and Releases:

You can create tags to mark specific points in the project's history, such as version releases. Tags are useful for identifying significant milestones.

14. Continuous Cycle:

The Git lifecycle continues as you repeat these steps over time to manage the ongoing development and evolution of your project. This cycle supports collaborative and agile software development.

The Git lifecycle allows for effective version control, collaboration, and the management of complex software projects. It provides a structured approach to tracking and sharing changes, enabling multiple developers to work together on a project with minimal conflicts and a clear history of changes.

Git Installation

To install Git on your computer, you can follow the steps for your specific operating system:

1. Installing Git on Windows:

a. Using Git for Windows (Git Bash):

- Go to the official Git for Windows website: https://gitforwindows.org/
- Download the latest version of Git for Windows.
- Run the installer and follow the installation steps. You can choose the default settings for most options.

b. Using GitHub Desktop (Optional):

- If you prefer a graphical user interface (GUI) for Git, you can also install GitHub Desktop, which includes Git. Download it from https://desktop.github.com/ and follow the installation instructions.

2. Installing Git from Source (Advanced):

- If you prefer to compile Git from source, you can download the source code from the official Git website (https://git-scm.com/downloads) and follow the compilation instructions provided there. This is usually only necessary for advanced users.

After installation, you can open a terminal or command prompt and verify that Git is correctly installed by running the following command:

```
$ git --version
```

If Git is installed successfully, you will see the Git version displayed in the terminal. You can now start using Git for version control and collaborate on software development projects.

How to Configure the Git?

Configuring Git involves setting up your identity (your name and email), customizing Git options, and configuring your remote repositories. Git has three levels of configuration: system, global, and repository-specific. Here's how you can configure Git at each level:

1. System Configuration:

- System-level configuration affects all users on your computer. It is typically used for site-specific configurations and is stored in the /etc/gitconfig file.

To set system-level configuration, you can use the git config command with the --system flag (usually requires administrator privileges). For example:

$ git config --system user.name "Your Name"
$ git config --system user.email "your.email@example.com"

1. Global Configuration:

• Global configuration is specific to your user account and applies to all Git repositories on your computer. This is where you usually set your name and email.

To set global configuration, you can use the git config command with the --global flag. For example:

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

You can also view your global Git configuration by using:
$ git config --global --list

Git Commands List

Git is a popular version control system used for tracking changes in software development projects. Here's a list of common Git commands along with brief explanations:

1. **git init**: Initializes a new Git repository in the current directory.
2. **git clone<repository URL>** : Creates a copy of a remote repository on your local machine.
3. **git add<file>** : Stages a file to be committed, marking it for tracking in the next commit.
4. **git commit -m "message"**: Records the changes you've staged with a descriptive commit message.
5. **git status**: Shows the status of your working directory and the files that have been modified or staged.
6. **git log**: Displays a log of all previous commits, including commit hashes, authors, dates, and commit messages.
7. **git diff**: Shows the differences between the working directory and the last committed version.
8. **git branch**: Lists all branches in the repository and highlights the currently checkedout branch.
9. **git branch<branchname>** : Creates a new branch with the specified name.
10. **git checkout<branchname>** : Switches to a different branch.
11. **git merge<branchname>** : Merges changes from the specified branch into the currently checked-out branch.
12. **git pull**: Fetches changes from a remote repository and merges them into the current branch.
13. **git push**: Pushes your local commits to a remote repository.
14. **git remote**: Lists the remote repositories that your local repository is connected to.
15. **git fetch**: Retrieves changes from a remote repository without merging them.
16. **git reset<file>** : Unstages a file that was previously staged for commit.
17. **git reset --hard<commit>** : Resets the branch to a specific commit, discarding all changes after that commit.
18. **git stash**: Temporarily saves your changes to a "stash" so you can switch branches without committing or losing your work.
19. **git tag**: Lists and manages tags (usually used for marking specific points in history, like releases).
20. **git blame<file>** : Shows who made each change to a file and when.
21. **git rm<file>** : Removes a file from both your working directory and the Git repository.
22. **git mv <oldfile><newfile>**: Renames a file and stages the change.

These are some of the most common Git commands, but Git offers a wide range of features and options for more advanced usage. You can use git --help followed by the command name to get more information about any specific command, e.g., git help commit.

EXPERIMENT 1: SETTING UP AND BASIC COMMANDS

AIM: Initialize a new local Git repository, create a physical file, and perform the first commit to establish a project history.

This experiment introduces the core Git lifecycle. You will move a project from being a standard folder to a version-controlled repository, then move data through the "Working Directory," "Staging Area," and finally the "Repository."

STEP-BY-STEP EXECUTION:

[Step 1] Create Workspace

- Every Git project requires a dedicated physical directory on your operating system to hold the project files.
- Create a new folder named `Lab_Exp1`.
- Right-click inside the folder and select **Git Bash Here**.

[Step 2] Initialize Git

- This command turns a normal folder into a Git repository by creating a hidden `.git` directory that tracks all future changes.
- **Command:** `git init`

[Step 3] Create File

- Before Git can track anything, a file must exist in the working directory.
- **Command:** `touch sample.txt`

[Step 4] Stage the File

- Staging tells Git which files you want to include in the next "snapshot" (commit). It allows you to select specific changes to save.
- **Command:** `git add sample.txt`

[Step 5] Commit Changes

- Committing takes the staged files and saves them permanently in the project's history with a descriptive message.
- **Command:** `git commit -m "Initial commit"`

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/Lab_Exp1
$ git init
Initialized empty Git repository in E:/Lab_Exp1/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/Lab_Exp1 (master)
$ touch sample.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/Lab_Exp1 (master)
$ git add sample.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/Lab_Exp1 (master)
$ git commit -m "Initial commit"
[master (root-commit) 1841332] Initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 sample.txt
```

EXPERIMENT 2: CREATING AND MANAGING BRANCHES

AIM: Create a "feature-branch" for isolated development, switch focus to it, and merge it back into the "master" branch.

Branching allows multiple versions of a project to exist simultaneously. This is essential for working on new features without breaking the stable "master" version of the code.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Initialize a fresh project and create a starting point on the master branch so we have a base to branch from.
- **Commands:**
  - `git init`
  - `touch main.txt`
  - `git add .`
  - `git commit -m "Main commit"`

[Step 2] Create a New Branch

- This creates a new timeline called `feature-branch` that is an exact copy of `master` at this specific moment.
- **Command:** `git branch feature-branch`

[Step 3] Switch to Branch

- By default, you stay on `master`. You must move your "HEAD" pointer to the new branch to begin working there.
- **Command:** `git checkout feature-branch`

[Step 4] Add Work to Branch

- Any changes made here are isolated to the `feature-branch` and do not affect the `master` branch.
- **Commands:**
  - `touch feature.txt`
  - `git add .`
  - `git commit -m "Feature work complete"`

[Step 5] Return to Master

- Before you can merge changes into `master`, you must navigate back to the `master` branch.

- **Command:** `git checkout master`

[Step 6] Merge the Changes

- This command pulls the unique history from the `feature-branch` and integrates it into the `master` branch.
- **Command:** `git merge feature-branch`

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder
$ git init
Initialized empty Git repository in E:/New folder/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$ touch main.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$ git add .

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$ git commit -m "Main commit"
[master (root-commit) c841b1d] Main commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 main.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$ git branch feature-branch

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$  git checkout feature-branch
Switched to branch 'feature-branch'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (feature-branch)
$ touch feature.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (feature-branch)
$ git add .

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (feature-branch)
$ git commit -m "Feature work complete"
[feature-branch 9bfb4ed] Feature work complete
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 feature.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (feature-branch)
$ git checkout master
Switched to branch 'master'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (master)
$ git merge feature-branch

Updating c841b1d..9bfb4ed
Fast-forward
 feature.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 feature.txt
```

EXPERIMENT 3: STASHING CHANGES

AIM: Use Git Stash to temporarily store uncommitted work-in-progress to clean the working directory.

Stashing is useful when you are in the middle of a task and need to switch branches immediately without committing half-finished work. It "parks" your changes safely.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Initialize the project and create a stable commit.
- **Commands:**
  - git init
  - touch file.txt
  - git add .
  - git commit -m "Base commit"

[Step 2] Create Unfinished Work

- Modify the file to simulate a task that is only partially complete.
- **Command:** echo "Drafting updates..." >> file.txt

[Step 3] Stash the Changes

- This command takes all uncommitted changes and puts them in a temporary storage area (the stash stack), returning your files to their last committed state.
- **Command:** git stash

[Step 4] Switch Branches

- Move to a different branch to perform an urgent task while your work is "parked."
- **Commands:**
  - git checkout -b fix-branch
  - git checkout master

[Step 5] Re-apply the Stash

- Once you have finished your urgent task, you can bring your "parked" changes back into your working directory.
- **Command:** git stash apply

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2)
$ git init
Initialized empty Git repository in E:/New folder (2)/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ echo "Stable version 1.0" > file.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git add .
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next
 time Git touches it

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git commit -m "Base commit"
[master (root-commit) cb2a499] Base commit
 1 file changed, 1 insertion(+)
 create mode 100644 file.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ echo "Drafting updates..." >> file.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git status
On branch master
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:   file.txt

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

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git stash
warning: in the working copy of 'file.txt', LF will be replaced by CRLF the next
 time Git touches it
Saved working directory and index state WIP on master: cb2a499 Base commit

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git checkout -b fix-branch
Switched to a new branch 'fix-branch'
```



```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (fix-branch)
$ git checkout master
Switched to branch 'master'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (2) (master)
$ git stash apply
On branch master
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:   file.txt
```

EXPERIMENT 4: COLLABORATION (CLONING)

AIM: Download a complete copy of a remote Git repository to your local machine.

Cloning is the primary way to start working on an existing project hosted on platforms like GitHub or GitLab. It downloads the entire history, not just the current files.

STEP-BY-STEP EXECUTION:

[Step 1] Prepare Parent Folder

- Create a location to hold the external project.
- Create a folder named Remote_Projects, open Git Bash.

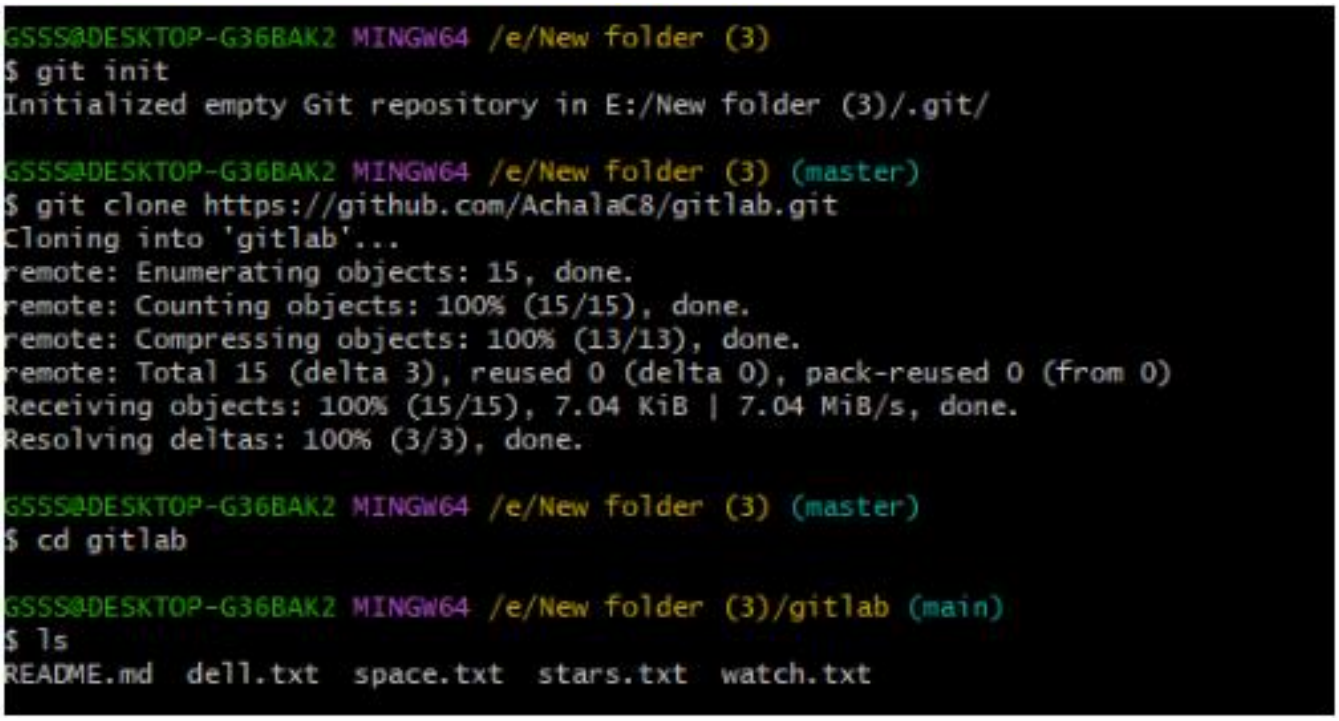
[Step 2] Clone Repository

- This command connects to the remote server and downloads all data associated with the repository.
- **Command:** git clone https://github.com/AchalaC8/gitlab.git

[Step 3] Verify Download

- Navigate into the newly created project folder and list the files to ensure everything was transferred.
- **Commands:**
  - cd gitlab
  - ls

Expected output:



EXPERIMENT 5: FETCHING AND REBASING

AIM: Download the latest changes from a remote repository and re-align your local work on top of those changes.

Fetching allows you to see what others have done without overwriting your files. Rebasing is a way to maintain a clean, linear project history by moving your local commits to the "end" of the updated remote history.

STEP-BY-STEP EXECUTION:

[Step 1] Setup Local Copy

- Begin with a project that is linked to a remote source.
- **Command:** git clone https://github.com/AchalaC8/gitlab.git

[Step 2] Fetch Remote Updates

- Git checks the remote "origin" server for any new commits made by team members and downloads that metadata.
- **Command:** git fetch origin

[Step 3] Rebase Local Commits

- This integrates the remote changes by placing your local work after the latest remote commit, avoiding unnecessary merge commits.
- **Command:** git rebase origin

Example output:

```
DELL@SNEHA MINGW64 /e/New folder (7)
$ git clone https://github.com/AchalaC8/gitlab.git
Cloning into 'gitlab'...
remote: Enumerating objects: 12, done.
remote: Counting objects: 100% (12/12), done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 12 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (12/12), 5.40 KiB | 614.00 KiB/s, done.
Resolving deltas: 100% (2/2), done.

DELL@SNEHA MINGW64 /e/New folder (7)
$ cd gitlab

DELL@SNEHA MINGW64 /e/New folder (7)/gitlab (main)
$ git fetch origin
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 1.67 KiB | 342.00 KiB/s, done.
From https://github.com/AchalaC8/gitlab
   aaa055e..1d6483b  main       -> origin/main

DELL@SNEHA MINGW64 /e/New folder (7)/gitlab (main)
$ git rebase origin
Successfully rebased and updated refs/heads/main.
```

EXPERIMENT 6: MERGING WITH CUSTOM MESSAGES

AIM: Merge a branch into master while providing a specific, descriptive commit message for the merge event.

In complex projects, standard auto-generated merge messages aren't descriptive enough. Providing a custom message helps team members understand the "why" behind the integration.

STEP-BY-STEP EXECUTION:

[Step 1] Create Scenario

- Setup a master branch and a development branch with different histories to prepare for a merge.
- **Commands:**
  - git init
  - touch file1.txt && git add . && git commit -m "First"
  - git checkout -b feature-branch
  - touch file2.txt && git add . && git commit -m "Second"
  - git checkout master

[Step 2] Merge with Message

- Use the -m flag to skip the default editor and provide a manual explanation of what this merge contains.
- **Command:** git merge feature-branch -m "Merging development work into stable master branch"

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4)
$ git init
Initialized empty Git repository in E:/New folder (4)/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4) (master)
$ touch file1.txt && git add . && git commit -m "First"
[master (root-commit) 3d9daef] First
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4) (master)
$ git checkout -b feature-branch
Switched to a new branch 'feature-branch'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4) (feature-branch)
$ touch file2.txt && git add . && git commit -m "Second"
[feature-branch e5d747d] Second
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4) (feature-branch)
$ git checkout master
Switched to branch 'master'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (4) (master)
$ git merge feature-branch -m "Merging development work into stable master branch"
Updating 3d9daef..e5d747d
Fast-forward (no commit created; -m option ignored)
 file2.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2.txt
```

EXPERIMENT 7: GIT TAGS AND RELEASES

AIM: Create a lightweight tag to mark a specific point in history as a version release.

Tags are pointers to specific commits that don't move (unlike branches). They are typically used to mark release points like v1.0, v2.0, etc.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Create a commit that represents a finished version of your software.
- **Commands:**
  - git init
  - touch app.js
  - git add .
  - git commit -m "Final production build"

[Step 2] Create Tag

- This command assigns the label v1.0 to the current commit for easy reference in the future.
- **Command:** git tag v1.0

[Step 3] View Tags

- List all tags in the repository to verify that the version mark exists.
- **Command:** git tag

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8)
$ git init
Initialized empty Git repository in E:/New folder (8)/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch app.js

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ git add .

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ git commit -m "Final production build"
[master (root-commit) 3460683] Final production build
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 app.js

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ git tag v1.0

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ git tag
v1.0
```

EXPERIMENT 8: ADVANCED OPERATIONS (CHERRY-PICK)

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

Cherry-picking is useful when a branch contains several changes, but you only want to move one specific fix or feature into another branch.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Create two branches. Add an important update to the feature branch that we will eventually want to "pick."
- **Commands:**
  - git init
  - touch main.txt&& git add . && git commit -m "main commit"
  - git checkout -b source-branch
  - touch fileA.txt && git add . && git commit -m "Commit A"
  - touch fileB.txt && git add . && git commit -m "Commit B"
  - touch fileC.txt && git add . && git commit -m "Commit C"
  - touch fileD.txt && git add . && git commit -m "Commit D"
  - git checkout master

[Step 2] Identify Commit Hash

- View the history of all branches to find the unique 7-character ID of **B, C, and D** into your main branch.
- **Command:** git log --oneline --all

[Step 3] Cherry-Pick the Commit

- While on the master branch, apply only the changes from that specific commit ID.
- **Command:** git cherry-pick [hash_B]^..[hash_D]

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6)
$ git init
Initialized empty Git repository in E:/New folder (6)/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (master)
$ touch main.txt&& git add . && git commit -m "main commit"
[master (root-commit) 874e474] main commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 main.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (master)
$ git checkout -b source-branch
Switched to a new branch 'source-branch'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (source-branch)
$ touch fileA.txt && git add . && git commit -m "Commit A"
[source-branch 9ce5158] Commit A
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileA.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (source-branch)
$ touch fileB.txt && git add . && git commit -m "Commit B"
[source-branch 94a2e2e] Commit B
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileB.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (source-branch)
$ touch fileC.txt && git add . && git commit -m "Commit C"
[source-branch 76e168c] Commit C
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (source-branch)
$ touch fileD.txt && git add . && git commit -m "Commit D"
[source-branch 5cfec13] Commit D
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileD.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (source-branch)
$ git checkout master
Switched to branch 'master'

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (master)
$ git log --oneline --all
5cfec13 (source-branch) Commit D
76e168c Commit C
94a2e2e Commit B
9ce5158 Commit A
874e474 (HEAD -> master) main commit

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (6) (master)
$ git cherry-pick 94a2e2e^..5cfec13
[master 3e56005] Commit B
 Date: Tue Jan 6 12:34:25 2026 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileB.txt
[master 9f4647d] Commit C
 Date: Tue Jan 6 12:34:30 2026 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileC.txt
[master 305998b] Commit D
 Date: Tue Jan 6 12:34:36 2026 +0530
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fileD.txt
```

EXPERIMENT 9: ANALYZING SPECIFIC COMMITS

AIM: View the detailed metadata and content changes of a single past commit.

The show command allows you to look inside a commit to see who made the change, when it was made, and exactly which lines of code were modified.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Create a repository and add multiple commits to build a history to analyze.
- **Commands:**
  - git init
  - echo "Start" > log.txt && git add . && git commit -m "Step 1"
  - echo "Update" >> log.txt && git add . && git commit -m "Step 2"

[Step 2] Find Target ID

- Review the log to find the ID of the commit you wish to investigate.
- **Command:** git log --oneline

[Step 3] Show Details

- Display the full patch (diff) and information for that specific commit.
- **Command:** git show <commit-id>

Expected output:

```
DELL@SNEHA MINGW64 /e/New folder (8)
$ git init
Initialized empty Git repository in E:/New folder (8)/.git/

DELL@SNEHA MINGW64 /e/New folder (8) (master)
$ echo "Start" > log.txt && git add . && git commit -m "Step 1"
warning: in the working copy of 'log.txt', LF will be replaced by CRLF the next time Git touches it
[master (root-commit) a00d093] Step 1
 1 file changed, 1 insertion(+)
 create mode 100644 log.txt

DELL@SNEHA MINGW64 /e/New folder (8) (master)
$ echo "Update" >> log.txt && git add . && git commit -m "Step 2"
warning: in the working copy of 'log.txt', LF will be replaced by CRLF the next time Git touches it
[master 79baf0c] Step 2
 1 file changed, 1 insertion(+)

DELL@SNEHA MINGW64 /e/New folder (8) (master)
$ git log --oneline
79baf0c (HEAD -> master) Step 2
a00d093 Step 1

DELL@SNEHA MINGW64 /e/New folder (8) (master)
$ git show 79baf0c
commit 79baf0c9be6e6d5716e99cd216107bf86f373a9b (HEAD -> master)
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Mon Jan 5 22:16:18 2026 +0530

    Step 2

diff --git a/log.txt b/log.txt
index 3b62dc1..cc68aee 100644
--- a/log.txt
+++ b/log.txt
@@ -1 +1,2 @@
 Start
+Update
```

EXPERIMENT 10: FILTERING HISTORY

AIM: Use advanced log filtering to find commits by specific authors or within specific time frames.

In large projects with thousands of commits, you often need to find specific work. Git allows you to filter the log output to find exactly what you are looking for.

STEP-BY-STEP EXECUTION:

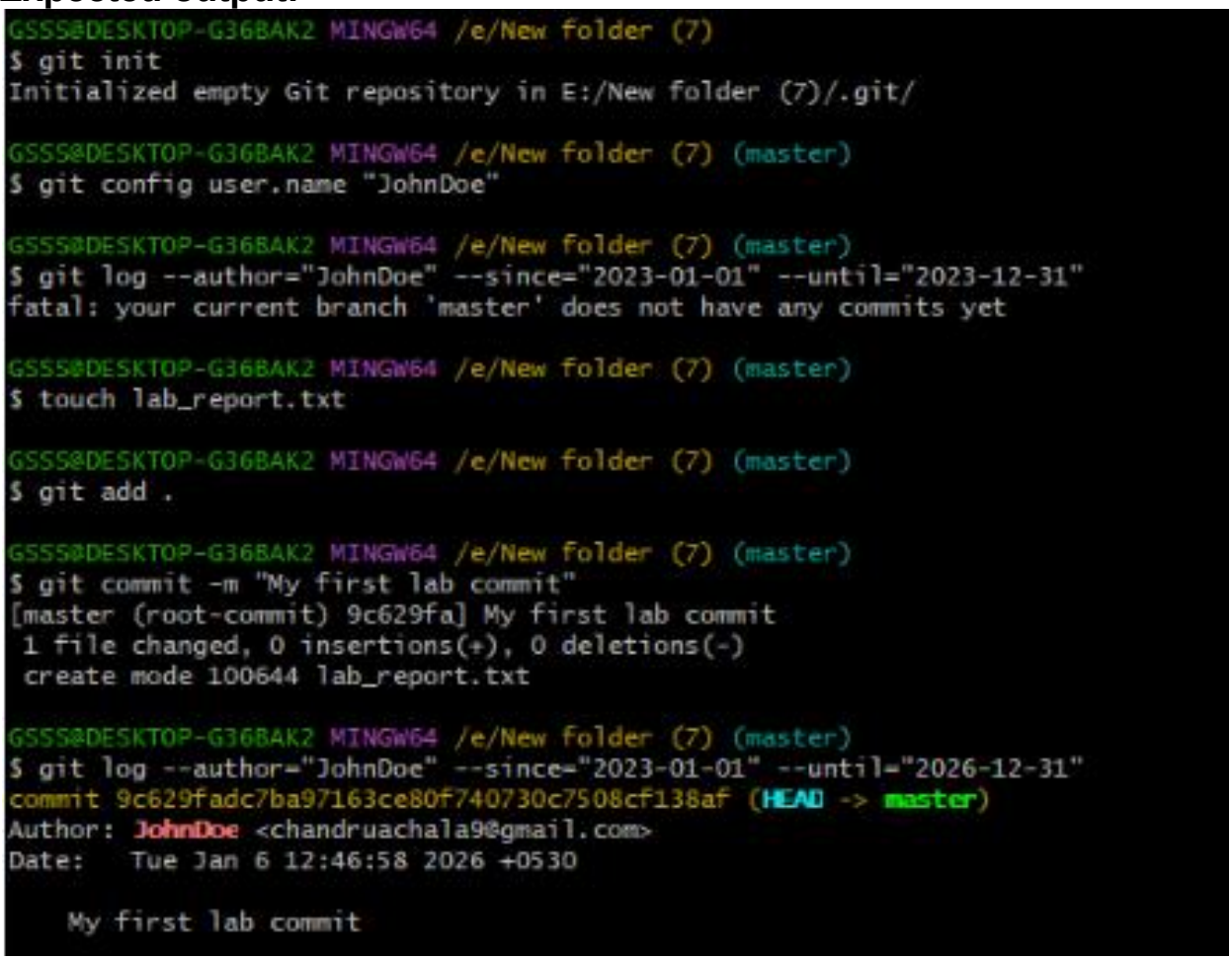
[Step 1] Setup from Scratch

- Initialize a project and configure the author identity to "JohnDoe" so the log has matching data to find
- **Commands:**
  - git init
  - git config user.name "JohnDoe"
  - touch lab_report.txt
  - git add .
  - git commit -m "My first lab commit"

[Step 2] Running the Search

- Execute a search that limits the results to JohnDoe and commits made recently.
- **Command:** git log --author="JohnDoe" --since="2023-01-01" --until="2026-12-31"

Expected output:



EXPERIMENT 11: DISPLAYING RECENT HISTORY

AIM: Limit the number of commits displayed in the terminal to avoid information overload.

When you only care about the most recent work, you can limit the log output to a specific number of entries.

STEP-BY-STEP EXECUTION:

[Step 1] Setup from Scratch

- Create a repository with several rapid-fire commits.
- **Commands:**
  - git init
  - touch file1.txt && git add . && git commit -m "C1"
  - touch file2.txt && git add . && git commit -m "C2"
  - touch file3.txt && git add . && git commit -m "C3"
  - touch file4.txt && git add . && git commit -m "C4"
  - touch file5.txt && git add . && git commit -m "C5"
  - touch file6.txt && git add . && git commit -m "C6"
  - touch file7.txt && git add . && git commit -m "C7"

[Step 2] Limit Log View

- Instruct Git to only show the top N results from the history stack.

Command: git log -n 5

Expected output:

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8)
$ git init
Initialized empty Git repository in E:/New folder (8)/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file1.txt && git add . && git commit -m "C1"
[master (root-commit) 37a4ea2] C1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file1.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file2.txt && git add . && git commit -m "C2"
[master 80b903a] C2
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file2.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file3.txt && git add . && git commit -m "C3"
[master b61f55b] C3
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file3.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file4.txt && git add . && git commit -m "C4"
[master 92b5438] C4
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file4.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file5.txt && git add . && git commit -m "C5"
[master 1938b42] C5
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file5.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file6.txt && git add . && git commit -m "C6"
[master b626b74] C6
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file6.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ touch file7.txt && git add . && git commit -m "C7"
[master adda919] C7
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 file7.txt
```

```
GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (8) (master)
$ git log -n 5
commit adda919eaa92482a228eec085fb78bd864a9d0f6 (HEAD -> master)
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Tue Jan 6 13:02:45 2026 +0530

    C7

commit b626b744ac30f5d61193b3520d26b2cd7a22fd48
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Tue Jan 6 13:02:39 2026 +0530

    C6

commit 1938b4224c6f43cff2ceeb3ad0ac8304a888081c
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Tue Jan 6 13:02:33 2026 +0530

    C5

commit 92b5438aba2ea856ba86f9ba87856778c14589cf
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Tue Jan 6 13:02:27 2026 +0530

    C4

commit b61f55b8ca67c962bf698270107e3eae4f1a3f5b
Author: AchalaC8 <chandruachala9@gmail.com>
Date:   Tue Jan 6 13:02:21 2026 +0530

    C3
```

EXPERIMENT 12: UNDOING CHANGES (REVERT)

AIM: Undo the effects of a specific commit by creating a new "inverse" commit.

Reverting is the safest way to undo changes in a shared repository because it doesn't delete history; it simply adds a new commit that subtracts the previous changes.

STEP-BY-STEP EXECUTION:

[Step 1] Setup and Error

- Create a stable state followed by a commit that contains a "mistake."
- **Commands:**
  - git init
  - touch ok.txt && git add . && git commit -m "Stable"
  - touch error.txt && git add . && git commit -m "Mistaken Commit"

[Step 2] Perform Revert

- Tell Git to calculate the opposite of the "Mistaken Commit" and apply it.
- **Commands:**
  - git log –oneline
  - git revert <bad-commit-id>

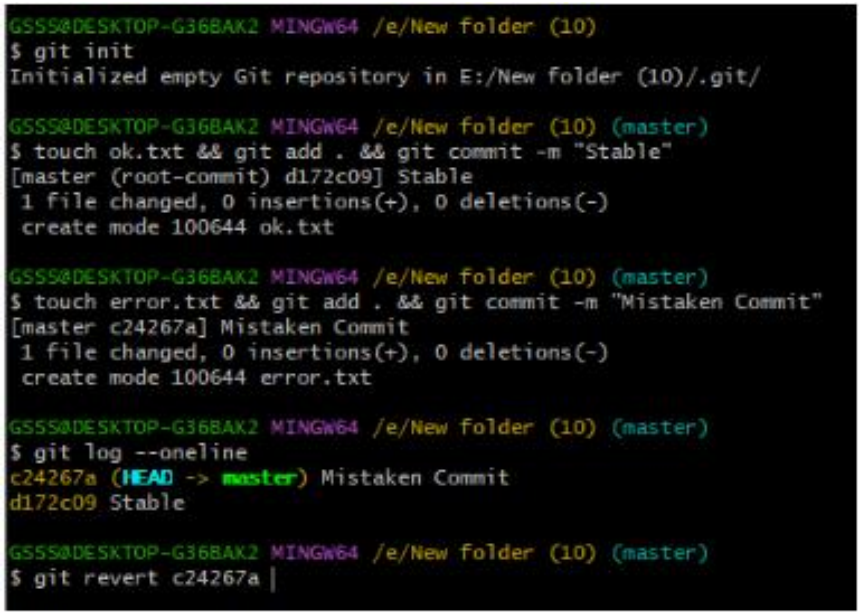
[Step 3] Finalize Revert Message

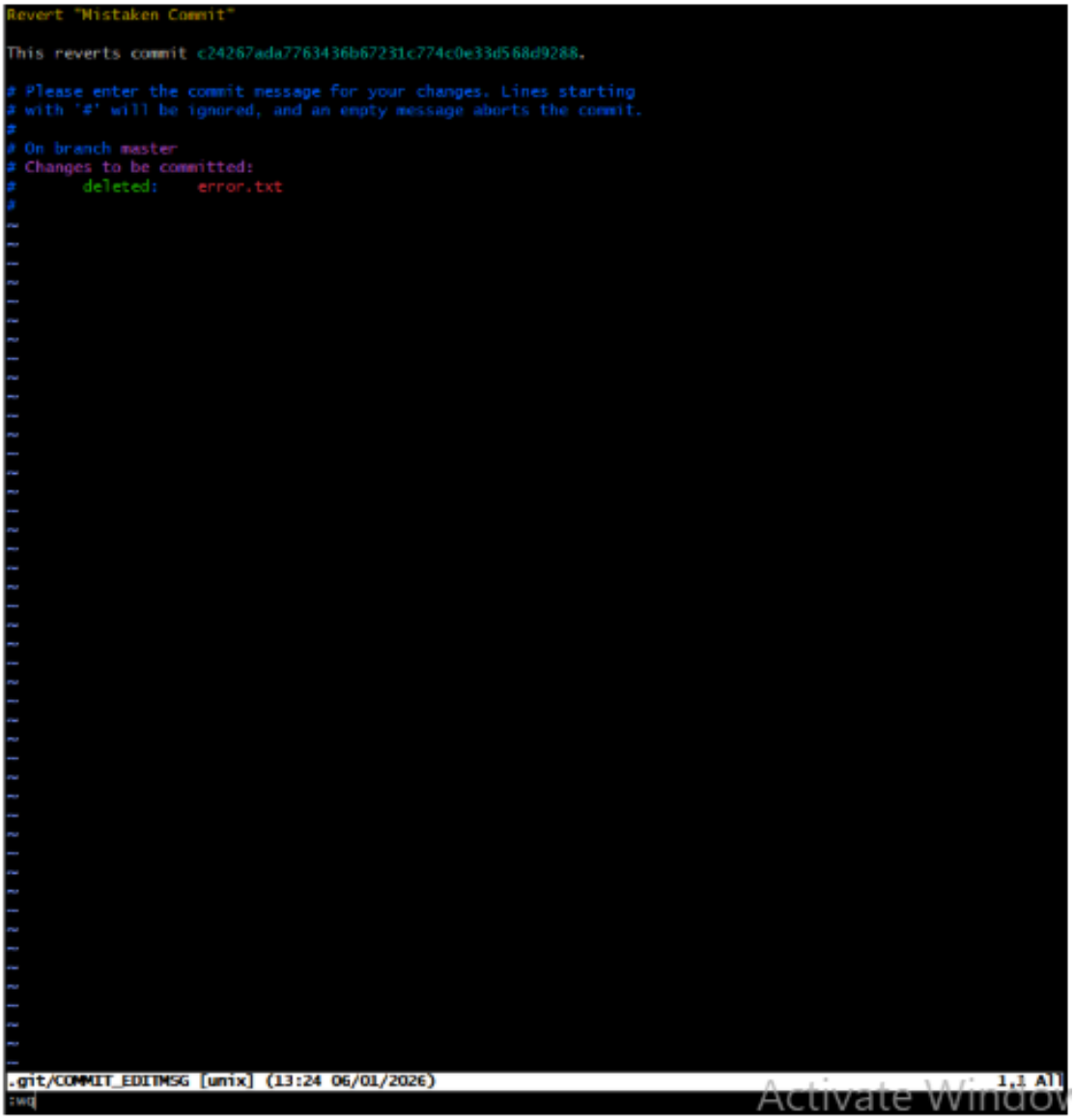
- Git opens an editor (Vim) to let you confirm the revert message.
- **Commands:**
  - Ctrl+c
  - Type :wq
  - press Enter to save and exit

[Step 4] Verify the changes

- Enter ls to see the files in repository
- **Commands:**
  - ls

Expected output:






```
[master 51ba1a8] Revert "Mistaken Commit"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 error.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /e/New folder (10) (master)
$ ls
ok.txt
```

Appendix

Repositoy Name: https://github.com/AchalaC8/USN.git