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

MEGHARANI RAJKUMAR GANI

4GW23CI026

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

Program outcomes

1 Engineering Knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2 Problem Analysis: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3 Design/Development of Solutions: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4 Conduct Investigations of Complex Problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

5 Modern Tool Usage: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

6 The Engineer and Society: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal, and cultural issues and the consequent responsibilities relevant to the professional engineering practice

7 Environment and Sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8 Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9 Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10 Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11 Project Management and Finance: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

12 Life-Long Learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change. Program Specific Outcomes

PSO 1: Apply the skills of core computer science engineering, artificial intelligence, machine learning, deep learning to solve futuristic problems.

PSO 2: Demonstrate computer knowledge, practical competency and innovative ideas in computer science engineering, artificial intelligence and machine learning using machine tools and techniques.

PROJECT MANAGEMENT WITH GIT
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
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 merg
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".

pogram 1

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 appropriate commit message.

Program:

- git config --global user.name "gitmanual" – Sets the global Git username.
- mkdir project – Creates a new directory named project.
- cd project – Moves into the project directory.
- touch git.txt – Creates a new file named git.txt
- git add . – Adds all files to the staging area.
- git commit -m "Initial commit - adding a new file" – Saves the staged file to Git with a commit message.

```
GSSS@DESKTOP-G36BAK2 MINGW64 ~
$ git config --global user.name "gitmanual"

GSSS@DESKTOP-G36BAK2 MINGW64 ~
$ mkdir project

GSSS@DESKTOP-G36BAK2 MINGW64 ~
$ cd project

GSSS@DESKTOP-G36BAK2 MINGW64 ~/project
$ git init
Initialized empty Git repository in C:/Users/GSSS/project/.git/

GSSS@DESKTOP-G36BAK2 MINGW64 ~/project (main)
$ touch git.txt

GSSS@DESKTOP-G36BAK2 MINGW64 ~/project (main)
$ git add .

GSSS@DESKTOP-G36BAK2 MINGW64 ~/project (main)
$ git commit -m "Initial commit - adding a new file"
[main (root-commit) 7d01e21] Initial commit - adding a new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 git.txt
```

Program 2:

Creating and Managing Branches Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

In Git, a **branch** is like a separate workspace. It lets you work on new ideas or features without changing the main project.

First, a new branch called **`feature-branch`** is created. This branch is a copy of the current state of the project. Any changes made here are kept separate from the main branch.

Next, we switch back to the **`master`** branch. The master branch represents the main, stable version of the project. Switching branches means Git updates the files so they match the version stored in that branch.

Finally, the **`feature-branch`** is merged into **`master`**. Merging means combining the changes from the feature branch into the main branch. After merging, the master branch now includes the new feature or updates made in the feature branch

Step 1: Create a New Branch (`feature-branch`)

When developing a new feature, it is considered best practice **not to work directly on the main branch** (usually called `master` or `main`).

- Creating a branch called **`feature-branch`** provides a **safe workspace** where all changes for this feature are isolated.
- Initially, the new branch points to the same commit as `master`, but any new commits will exist only on the `feature-branch` until you merge it.

Purpose:

- Encourages organized development
- Protects the stability of the main branch

- Makes collaboration easier since multiple features can be developed in parallel

Step 2: Switch Back to the Main Branch (`master`)

After working on a feature branch, you often return to the **master branch** to:

- Integrate completed work
- Continue stable development
- Merge other branches or fixes

Switching branches updates your working directory to reflect the state of the branch you switch to. Any changes that were committed on `feature-branch` remain intact and isolated until merged.

Purpose:

- Maintains a stable version of the project
- Prepares the branch for integrating new features

Step 3: Merge the Feature Branch into Master

Merging combines the changes from one branch into another.

- When you merge **`feature-branch`** into `master`, Git takes all commits that exist on the feature branch and integrates them into the main branch.
- Git performs one of two types of merges:
    - **Fast-forward merge:** If the master hasn't moved ahead, Git simply moves the pointer to include all commits from the feature branch.
    - **Three-way merge:** If both branches have diverged, Git combines the histories and creates a new merge commit to reconcile the changes.

Purpose:

- Brings the new feature into the main codebase
- Preserves project history
- Ensures that completed work is safely integrated

Step 4: Resolve Conflicts (if any)

Sometimes, the same lines of code may have been modified on both branches.

- Git will flag these conflicts during the merge.
- Conflicts must be manually resolved, after which the merge is completed.

Purpose:

- Maintains code integrity
- Prevents overwriting someone else's work

Step 5: Verify the Merge

After merging:

- The master branch should now contain all the new commits from the feature branch.
- You can inspect the commit history to confirm that the merge was successful and that all intended changes are included.

Purpose:

- Confirms that the feature has been fully integrated
- Helps maintain a clean, organized project history

.

```
GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ touch README.md

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ git add .

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ gt commit -m "Readme file"
bash: gt: command not found

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ git commit -m "Readme file"
On branch main
nothing to commit, working tree clean

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ git status
On branch main
nothing to commit, working tree clean

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ git add README.md

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ ls
README.md
```

```
GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (feature-branch)
$ git branch feature-branch
```

```
GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (master)
$ git checkout main
Switched to branch 'main'
```

```
GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$ git merge feature-branch
Updating c964361..a5786f8
Fast-forward
 README.md | 1 -
 file1.txt | 0
 2 files changed, 1 deletion(-)
 create mode 100644 file1.txt

GSSS@DESKTOP-G36BAK2 MINGW64 /d/git_lab (main)
$
```

Program 3: Creating and Managing Branches Write the commands to stash your changes, switch branches, and then apply the stashed changes.

Step 1: Stashing Your Changes (Saving Work Temporarily)

Stashing means **temporarily saving your current work** without committing it.
When you stash changes:

- Git stores your modified files in a safe place
- Your working directory becomes clean
- The project returns to the last committed state

This allows you to stop your current work without losing progress.

Purpose of this step:

- Protect unfinished work
- Keep the working directory clean
- Prepare for branch switching

Step 2: Switching to Another Branch

After stashing your changes, you can safely switch to another branch.
When switching branches:

- Git changes the project files to match the selected branch
- The stashed changes remain stored and are not affected
- You can work freely on the new branch

Purpose of this step:

- Move between different tasks
- Fix bugs or work on urgent features
- Avoid conflicts with unfinished work

Step 3: Applying the Stashed Changes (Restoring Work)

Once you return to the original branch or are ready to continue your work, the stashed changes can be applied.
Applying a stash:

- Restores the saved changes to the working directory
- Brings back the files exactly as they were before stashing
- Allows you to continue work from where you left off

The stash can either be kept for later use or removed after applying.

Purpose of this step:

- Resume unfinished work
- Maintain workflow continuity
- Avoid rewriting code

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 2
commits.
  (use "git push" to publish your local com
mits)

Untracked files:
  (use "git add <file>..." to include in wh
at will be committed)
        file1.txt

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

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (main)
$ git add .

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 2
commits.
  (use "git push" to publish your local com
mits)

Changes to be committed:
  (use "git restore --staged <file>..." to
unstage)
        new file:   file1.txt


admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (main)
$ git stash push -m"WIP: my cahnges"
Saved working directory and index state On
main: WIP: my cahnges

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (main)
$ git checkout -b target-branch
Switched to a new branch 'target-branch'

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (target-branch)
$ git stash pop
On branch target-branch
Changes to be committed:
```

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (target-branch)
$ git stash pop
On branch target-branch
Changes to be committed:
  (use "git restore --staged <file>..." to
unstage)
        new file:   file1.txt

Dropped refs/stash@{0} (0100c5e91ca18679496
c71a9f1e6bc4a83755fe3)

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New fo
lder (target-branch)
$ git status
On branch target-branch
Changes to be committed:
  (use "git restore --staged <file>..." to
unstage)
        new file:   file1.txt
```

Program 4: Collaboration and Remote Repositories Clone a remote Git repository to your local machine.

Step 1: Get the Repository URL

Before cloning, you need the **URL** of the remote repository.
This URL tells Git where the project is located online.

The URL can be:

- HTTPS based
- SSH based

Purpose of this step:

- Identify the remote project you want to copy
- Establish a link between local and remote repositories

Step 2: Run the Clone Command

When you clone a repository, Git:

- Downloads all files
- Copies the entire history
- Automatically sets the remote name as `origin`

Step 3: Access the Local Repository

After cloning:

- You can open the project folder
- Start editing files
- Create new branches
- Commit changes locally

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (master)
$ git clone https://github.com/meghagani21-png/megha1.git
Cloning into 'megha1'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
```

Program 5: Collaboration and Remote Repositories Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

1. **Check for updates from the remote repository**
   - Get the latest changes made by other team members from the remote repository.
   - This ensures your local repository is aware of the most recent commits.
2. **Prepare your local branch**
   - Make sure you are on the branch that you want to update with the remote changes.
3. **Rebase your local branch**
   - Move your local commits on top of the latest commits from the remote branch.
   - This applies your changes as if they were made after the newest updates, keeping the project history linear and clean.
4. **Resolve conflicts if any**
   - If your local changes and the remote changes affect the same lines of code, fix the conflicts.
   - After resolving, continue the rebase process.
5. **Verify your branch**
   - Check that your local branch now includes the latest remote changes and your commits are correctly applied on top.

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git add demo.txt
warning: in the working copy of 'demo.txt', LF will be replaced
e Git touches it

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git continue -m "added demo.txt"
git: 'continue' is not a git command. See 'git --help'.

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git commit -m "added demo.txt"
[main 27c64cf] added demo.txt
 1 file changed, 1 insertion(+)
 create mode 100644 demo.txt

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git config --global core.autocrlf true

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git rebase main
Current branch main is up to date.

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git fetch origin

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (main)
$ git rebase origin
Current branch main is up to date.
```

Program 6: Collaboration and Remote Repositories Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.

1. **Switch to the main branch** – Make sure you are on the branch where you want the changes to go (for example, `master`).
2. **Start the merge** – Combine the changes from the feature branch into the main branch.
3. **Provide a custom message** – Write a clear description explaining the purpose of the merge.
4. **Resolve any conflicts** – If the two branches have conflicting changes, fix them before completing the merge.
5. **Confirm the merge** – Check that the main branch now includes the changes from the feature branch with your message attached.

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (master)
$ git checkout 'feature-branch'
A       file1.txt
Switched to branch 'feature-branch'

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git merge -m "Merge feature-branch: add login login feature" feature-branch
Already up to date.
```

Program 7: Git Tags and Releases Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

Step 1: Identify the Commit

- Decide which commit you want to tag.
- If you don't specify a commit, Git will tag the **latest commit** by default.

Step 2: Create the Lightweight Tag

- Give your tag a name, for example: `v1.0`
- This creates a simple reference to the chosen commit.

Step 3: Verify the Tag

- Check your list of tags to make sure it was created successfully.
- The tag now points to the specific commit in your local repository.

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git tag v1.0

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ it log --oneline
bash: it: command not found

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git log --oneline
63b9f28 (HEAD -> feature-branch, tag: v1.0) initial commit
```

Program 8: Advanced Git Operations: Write the command to cherry-pick a range of commits from "source-branch" to the current branch

Step 1: Identify the Range of Commits

- Determine the **starting commit** and the **ending commit** you want to copy from `source-branch`.
- Example: `commitA` (first commit) to `commitD` (last commit)

Step 2: Switch to the Target Branch

- Make sure you are on the branch where you want to apply the commits.
- This is the branch that will receive the selected commits.

Step 3: Apply the Commits Using Cherry-Pick

- Use Git's cherry-pick feature to apply the range of commits.
- This creates new commits in the current branch with the same changes as in the source branch.

Command Example:

```
git cherry-pick commitA^..commitD
```

Explanation:

- `commitA^..commitD` means "from the commit **before commitA** up to commitD."
- Git will replay all these commits on the current branch.

Step 4: Resolve Conflicts (if any)

- If the changes conflict with the current branch, Git will pause the cherry-pick and ask you to resolve them.
- After resolving conflicts, continue with:

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git log --oneline
63b9f28 (HEAD -> feature-branch, tag: v1.0, source-branch) initial commit

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git cherry-pick 63b9f28
error: your local changes would be overwritten by cherry-pick.
hint: commit your changes or stash them to proceed.
fatal: cherry-pick failed

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git status
On branch feature-branch
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   file1.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        megha1/
```

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git add "megha1/"

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git commit -m"my local changes"
[feature-branch a40cb55] my local changes
 2 files changed, 1 insertion(+)
 create mode 100644 file1.txt
 create mode 160000 megha1

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git cherry-pick 63b9f28
On branch feature-branch
You are currently cherry-picking commit 63b9f28.
  (all conflicts fixed: run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

nothing to commit, working tree clean
The previous cherry-pick is now empty, possibly due to conflict resolution.
If you wish to commit it anyway, use:

    git commit --allow-empty

Otherwise, please use 'git cherry-pick --skip'

admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch|CHERRY-PICKING)
$ git cherry-pick --abort
```

Program 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?

Steps

1. **Identify the Commit ID**
   - Find the unique commit ID (hash) of the commit you want to view.
   - Example: `abc123`
2. **Request the Commit Details**
   - Ask Git to show all the information for that commit.
3. **Check the Output**
   - Git will display:
     - Author of the commit
     - Date and time of the commit
     - Commit message explaining the changes
4. **Use the Information**
   - You can understand what changes were made, who made them, and when.

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git show 63b9f28
commit 63b9f28e8cabf411d3db708c46bd1dbe14c3bd36 (tag: v1.0, source-branch)
Author: megha1 <meghagani21@gmail.com>
Date:   Mon Jan 5 21:22:18 2026 +0530

    initial commit

diff --git a/file.txt b/file.txt
new file mode 100644
index 0000000..e69de29
```

Program 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."

Step 1: Identify the Author

Decide which developer's work you want to track.
, the author name is **Megha**.

Purpose: Ensures that only Megha's commits are considered, not anyone else's work.

Step 2: Decide the Date Range

Choose the period you want to examine. For example:

- Start: **January 1, 2023**
- End: **December 31, 2023**

Purpose: Limits the results to the relevant timeframe and avoids clutter from older or future commits.

Step 3: Filter the Commits

Review the commit history using the filters for **author** and **date**.
This gives a list of all commits Megha made during the specified period.

Step 4: Review the Commit Details

For each commit in the list, check:

- Who made the change (should be Megha)
- When the change was made
- What the commit message says

Step 5: Use the Information

Once you have the filtered commits, you can:

- Track Megha’s contributions
- Identify the features or fixes she worked on
- Summarize her work for project review or reports

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git log --author="megha1" --since="2024-09-25" --until="2026-01-21" --oneline
921334b (HEAD -> feature-branch) fourth commit
2887105 commit2
b92fcce commit 1
a40cb55 my local changes
63b9f28 (tag: v1.0, source-branch) initial commit
```

Program 11: Analysing and Changing Git History Write the command to display the last five commits in the repository's history.

Step 1: Open the Repository

First, open the terminal or command prompt and navigate to the folder that contains your Git repository.
This ensures that Git knows which project's history you want to view.

Step 2: Request the Commit History

Ask Git to show the project's commit history.
Git stores every change as a commit, and this history allows you to review past work.

Step 3: Limit the Number of Commits Displayed

Instead of viewing the entire history, specify that only the **most recent five commits** should be displayed.
This helps focus on recent changes and avoids unnecessary information.

Step 4: Review the Output

Git displays:

- The commit IDs
- Author information
- Date of each commit
- Commit messages

These details help you understand what changes were made recently.

```
admin@DESKTOP-ENRDLON MINGW64 /d/usn/New folder (feature-branch)
$ git log -n 5
commit 921334b7fe05c40ed2efa2ac9d069f3b7256dd42 (HEAD -> feature-branch)
Author: megha1 <meghagani21@gmail.com>
Date:   Tue Jan 6 06:04:41 2026 +0530

    fourth commit

commit 28871054e23a6729dccc96d8eeb3257339266c03
Author: megha1 <meghagani21@gmail.com>
Date:   Tue Jan 6 06:03:29 2026 +0530

    commit2

commit b92fcce851278d3bc18f7245a83bf6f52e42b36e
Author: megha1 <meghagani21@gmail.com>
Date:   Tue Jan 6 06:02:08 2026 +0530

    commit 1

commit a40cb559a0b00ea1199d0ed66bec12843035c9f4
Author: megha1 <meghagani21@gmail.com>
Date:   Tue Jan 6 05:38:39 2026 +0530

    my local changes

commit 63b9f28e8cabf411d3db708c46bd1dbe14c3bd36 (tag: v1.0, source-branch)
Author: megha1 <meghagani21@gmail.com>
Date:   Mon Jan 5 21:22:18 2026 +0530

    initial commit
```

Program 12: Analyzing and Changing Git History Write the command to undo the changes introduced by the commit with the ID "abc123".

Step 1: Identify the Commit

First, identify the commit that introduced unwanted changes.
Every change in Git is saved as a commit with a unique ID. In this case, the commit with the ID **abc123** is the one that needs to be undone.

Why this matters:
Knowing the exact commit ensures that only the incorrect changes are targeted and nothing else is affected.

Step 2: Reverse the Changes from That Commit

Instead of deleting or editing the project's history, Git creates a new action that **reverses the effects** of the selected commit.
This means:

- The original commit stays in the history
- A new change is added that cancels out what was done earlier

```
[feature-branch f2ea364] Revert "fourth commit"
 1 file changed, 1 insertion(+), 1 deletion(-)
```