**Git Version Control Lab Manual**

**Table of Contents**

1. **Introduction to Version Control Systems**
2. **Features and Architecture of Git**
3. **Life Cycle of a File in Git**
4. **Git Installation on Windows**
5. **Working Directory, Staging Area, and Local Repository**
6. **Six Essential Git Commands**
7. **Git Log Command**
8. **Git Diff Command**
9. **Helix Visual Merge Tool (P4Merge)**
10. **Removing Files with Git Rm**
11. **Undoing Changes with Git Checkout**
12. **Git References (Master and HEAD)**
13. **Git Reset Command**
14. **Git Aliases**
15. **Ignoring Files with .gitignore**
16. **Special Treatment for Directories**
17. **Branching and Merging**
18. **Merging with Rebase**
19. **Stashing in Git**
20. **Working with Remote Repositories**
21. **Git Tagging**
22. **Git Revert Command**
23. **Cherry-Picking**
24. **Git Reflog Command**
25. **Advanced Git Tagging**
26. **Advanced Git Revert**
27. **Advanced Cherry-Picking**
28. **Advanced Git Reflog**

**1. Introduction to Version Control Systems**

A Version Control System (VCS) is a tool that tracks and manages changes to files, such as code or documents, by storing a history of modifications, enabling collaboration, and allowing you to revert to previous versions if needed. It helps organize projects, prevents data loss, and supports teamwork by letting multiple people work on the same files without conflicts. Popular tools like Git, used with platforms like GitHub or GitLab, allow you to create repositories, commit changes, branch for new features, and merge updates efficiently. In a lab, a VCS keeps experiments, scripts, and data organized, ensuring changes are tracked, recoverable, and shareable, making your workflow smoother and more reliable.

**2. Features and Architecture of Git**

Git is a distributed version control system (VCS) designed for speed, flexibility, and reliability in managing project files. Its robust features and unique architecture make it a popular choice for developers and researchers.

**Key Features of Git**

1. **Distributed System**: Every user has a complete local copy of the repository, including its full history, enabling offline work and faster operations.
2. **Branching and Merging**: Create lightweight branches for experimenting or adding features, and merge them seamlessly with conflict resolution.
3. **Speed**: Git performs operations like committing, branching, and merging quickly, even in large projects, due to its efficient design.
4. **Data Integrity**: Uses cryptographic hashing (SHA-1) to ensure the integrity of files and history, preventing data corruption.
5. **Staging Area**: Allows selective staging of changes (via git add) before committing, giving precise control over what changes are saved.
6. **Collaboration**: Supports team collaboration through remote repositories on platforms like GitHub, GitLab, or Bitbucket, with push/pull workflows.
7. **Flexibility**: Handles both small and large projects, supports non-linear development, and integrates with various tools and workflows.
8. **Open Source**: Free to use, with a large community contributing to its development and support.

**Git Architecture**

Git’s architecture is built around a few core components that enable its functionality:

1. **Repository**: The storage unit containing all project files and their version history, stored in a .git directory. A repository can be local or remote.
2. **Working Directory**: The local folder where you edit files. It reflects the current state of your project.
3. **Staging Area (Index)**: A temporary area where changes are prepared before committing. You stage specific changes using git add.
4. **Commit History**: A series of snapshots (commits) stored as objects, each with a unique SHA-1 hash, tracking changes to files over time.
5. **Objects**: Git stores data as four types of objects:
   * **Blob**: Stores file content.
   * **Tree**: Represents directories, linking blobs and other trees.
   * **Commit**: Captures the state of the project at a point in time, including a tree, parent commit(s), and metadata (e.g., author, message).
   * **Tag**: Marks specific commits, often for releases.
6. **Branches**: Pointers to specific commits, allowing parallel development. The default branch is often called main or master.
7. **HEAD**: A pointer to the current branch or commit you’re working on.
8. **Remote Repositories**: Hosted on servers (e.g., GitHub), allowing collaboration by syncing local repositories via push/pull.

**How Git Works**

* You work in the **working directory**, editing files.
* Changes are **staged** in the index using git add.
* Staged changes are saved as a **commit** with git commit, creating a snapshot in the repository.
* **Branches** allow you to work on different features simultaneously, and **merging** combines them.
* **Push** sends local commits to a remote repository, and **pull** retrieves updates from it.

**Benefits for Lab Use**

* Track changes to scripts, data, or experiment notes with precision.
* Experiment on branches without affecting the main project.
* Collaborate with lab members by sharing repositories.
* Ensure data integrity and recover from errors easily.

Git’s distributed architecture and powerful features make it an essential tool for managing complex projects efficiently and reliably.

**3. Life Cycle of a File in Git**

In Git, every file in your project goes through a simple life cycle as you work on it. Understanding this cycle helps you manage and track changes easily.

1. **Untracked**: When you create a new file, Git doesn't know about it yet. It's called "untracked" because Git isn't tracking its changes.
2. **Staged**: You tell Git to track the file by adding it to the staging area with **git add <filename>**. This prepares the file (or specific changes) to be saved.
3. **Committed**: When you run **git commit**, Git saves a snapshot of the staged files. The file is now "committed" and its changes are stored in the repository's history.
4. **Modified**: If you edit a committed file, it becomes "modified." Git notices the changes, but they aren't saved yet until you stage and commit again.
5. **Cycle Repeats**: After modifying a file, you stage the changes (git add) and commit them (git commit) again, repeating the cycle as needed.

**Simple Workflow Example**

* Create a file (untracked).
* Use git add myfile.txt to stage it.
* Use git commit -m "Added myfile.txt" to commit it.
* Edit the file (modified), then stage (git add) and commit again.

This cycle helps you keep track of all changes, work with others, and revert to earlier versions if needed, making your lab work organized and safe!

**4. Git Installation on Windows**

Installing Git on Windows is simple and allows you to use version control for your projects. Follow these easy steps to set it up.

**Steps to Install Git**

1. **Download Git**:
   * Visit the official Git website: <https://git-scm.com>.
   * Click the "Download for Windows" button to get the latest installer.
2. **Run the Installer**:
   * Open the downloaded .exe file (e.g., Git-2.x.x-64-bit.exe).
   * Click "Yes" if prompted by Windows User Account Control.
3. **Follow the Setup Wizard**:
   * **License Agreement**: Read and click "Next."
   * **Installation Location**: Keep the default folder (e.g., C:\Program Files\Git) and click "Next."
   * **Select Components**: Choose the default options (e.g., Git Bash, Git GUI) and click "Next."
   * **Default Editor**: Select an editor like Notepad++ or VS Code (or keep the default, Vim) and click "Next."
   * **Branch Name**: Choose "Let Git decide" for the default branch name (usually main) and click "Next."
   * **Adjust PATH**: Select "Git from the command line and third-party software" (recommended) and click "Next."
   * **Line Endings**: Choose "Checkout Windows-style, commit Unix-style line endings" (default) and click "Next."
   * **Terminal Emulator**: Select "Use MinTTY" (default) for Git Bash and click "Next."
   * **Extra Options**: Keep defaults (e.g., enable symbolic links) and click "Next."
   * **Install**: Click "Install" and wait for the process to complete.
4. **Finish Installation**:
   * Click "Finish" to close the wizard.
   * Optionally, check "Launch Git Bash" to open the Git Bash terminal.

**Verify Installation**

* Open **Git Bash** (search for it in the Start menu).
* Type git --version and press Enter. You should see the Git version (e.g., git version 2.x.x).

**Basic Setup**

* Set your name: git config --global user.name "Your Name"
* Set your email: git config --global user.email "your.email@example.com"
* Check settings: git config --list

**Why Use Git on Windows?**

* Manage lab scripts, data, or notes with version control.
* Collaborate with others using platforms like GitHub.
* Track changes and revert mistakes easily.

Now Git is ready to use on your Windows computer for all your lab projects!

**5. Working Directory, Staging Area, and Local Repository**

In Git, your project files move through three key areas: the **Working Directory**, **Staging Area**, and **Local Repository**. These areas help you manage and track changes easily.

* **Working Directory**: This is your project folder where you edit files (e.g., code, notes). It shows the current state of your files. Files are either **tracked** (Git monitors them) or **untracked** (new files Git doesn’t know about yet).
  + *Command*: git status – Shows which files are modified, staged, or untracked in the working directory.
  + *Example*: You edit labnotes.txt in your project folder; it’s now modified in the working directory.
* **Staging Area**: A temporary area where you select changes to save. You choose which files or changes to include for the next save using git add.
  + *Command*: git add <filename> – Stages a specific file (e.g., git add labnotes.txt).
  + *Command*: git add . – Stages all modified files.
  + *Example*: After editing labnotes.txt, run git add labnotes.txt to stage those changes.
* **Local Repository**: A hidden .git folder in your project that stores the full history of your changes as commits (snapshots). You save staged changes here with git commit.
  + *Command*: git commit -m "message" – Saves staged changes with a message (e.g., git commit -m "Updated labnotes").
  + *Command*: git log – Shows the commit history in the local repository.
  + *Example*: After staging, run git commit -m "Added notes" to save changes to the repository.

**How They Work Together**:

1. Edit files in the **working directory** (e.g., change labnotes.txt).
2. Stage changes with git add labnotes.txt to move them to the **staging area**.
3. Commit staged changes with git commit -m "message" to save them in the **local repository**.
4. Repeat this cycle for every change.

**Why It’s Useful for Labs**:

* Keep your code, data, or notes organized.
* Stage only the changes you want to save.
* Store a safe history of all changes to revisit or undo if needed.

This simple process helps you manage lab projects efficiently with Git!

**6. Six Essential Git Commands**

Git is a powerful tool for managing project files, and these six essential commands will help you get started with version control in your lab work. Each command is simple and commonly used to track and manage changes.

1. git init
   * Purpose: Creates a new Git repository in your project folder.
   * Use: Starts version control for your project, setting up the .git folder to store history.
   * Example: git init
     + Run in your project folder to make it a Git repository.
2. git add
   * Purpose: Stages changes (new, modified, or deleted files) for the next commit.
   * Use: Selects which changes to save from the working directory to the staging area.
   * Example: git add labnotes.txt
     + Stages changes in labnotes.txt.
   * Example: git add .
     + Stages all modified files in the working directory.
3. git commit
   * Purpose: Saves staged changes to the local repository with a descriptive message.
   * Use: Permanently stores changes as a snapshot in the project’s history.
   * Example: git commit -m "Added experiment notes"
     + Commits staged changes with a message describing the update.
4. git status
   * Purpose: Shows the current state of your working directory and staging area.
   * Use: Checks which files are modified, staged, or untracked.
   * Example: git status
     + Displays files that are changed, staged, or ready to commit.
5. git log
   * Purpose: Displays the history of commits in the repository.
   * Use: Helps you review past changes, including commit messages, authors, and dates.
   * Example: git log
     + Shows a list of all commits in the current branch.
   * Example: git log --oneline
     + Shows a compact version of the commit history.
6. git push
   * Purpose: Sends your committed changes from the local repository to a remote repository (e.g., GitHub).
   * Use: Shares your work with others or backs it up online.
   * Example: git push origin main
     + Pushes commits from your local main branch to the remote repository named origin.

Why These Commands Matter for Lab Work:

* Initialize a repository (git init) to start tracking your lab scripts or data.
* Stage (git add) and commit (git commit) changes to save experiment updates.
* Check progress (git status) to stay organized.
* Review history (git log) to track your work.
* Share or back up projects (git push) for collaboration.

These commands form the foundation of using Git to manage your lab projects efficiently!

**7. Git Log Command**

The git log command is a powerful tool in Git that displays the commit history of a repository. It helps you see what changes were made, who made them, and when, making it essential for tracking progress in your lab projects.

**Purpose**

* Shows a list of commits in the current branch, including details like commit ID, author, date, and commit message.
* Helps you review the history of your project to understand changes or find specific commits.

**Basic Usage**

* **Command**: git log
  + Displays the full commit history in the current branch, from newest to oldest.
  + Each commit shows:
    - Commit ID (a unique SHA-1 hash).
    - Author’s name and email.
    - Date and time of the commit.
    - Commit message.

**Useful Variations**

1. **Compact View**:
   * **Command**: git log --oneline
   * Shows each commit in a single line with the commit ID (shortened) and message.
   * *Example*: git log --oneline
     + Output: a1b2c3d Updated labnotes.txt
2. **Limit Commits**:
   * **Command**: git log -n <number>
   * Shows only the specified number of recent commits.
   * *Example*: git log -n 3
     + Displays the last 3 commits.
3. **Show File Changes**:
   * **Command**: git log --stat
   * Includes a summary of files changed and lines added/removed in each commit.
   * *Example*: git log --stat
     + Shows commit details plus file changes (e.g., labnotes.txt | 5 +--).
4. **Filter by Author**:
   * **Command**: git log --author="Name"
   * Shows commits by a specific author.
   * *Example*: git log --author="John"
     + Displays only John’s commits.
5. **Show Specific File History**:
   * **Command**: git log -- <filename>
   * Shows commits that affected a specific file.
   * *Example*: git log -- labnotes.txt
     + Lists commits that changed labnotes.txt.
6. **Graphical View**:
   * **Command**: git log --graph --oneline
   * Displays a visual representation of branches and merges.
   * *Example*: git log --graph --oneline
     + Shows a branch diagram with commit messages.

**Why It’s Useful for Lab Work**

* Track changes to scripts, data, or experiment notes over time.
* Identify who made specific updates and when.
* Find a commit to revert or review changes (e.g., when an experiment was modified).
* Understand project progress or debug issues by examining commit history.

**Tips**

* Use q to exit the log view in the terminal.
* Combine options (e.g., git log --oneline --author="John") for specific needs.
* Run git log regularly to stay updated on your project’s history.

The git log command is your go-to tool for reviewing and understanding the evolution of your lab projects in Git!

**8. Git Diff Command**

The git diff command in Git shows the differences between file changes in your project. It’s a key tool for reviewing modifications before staging or committing, helping you understand what’s changed in your lab work.

**Purpose**

* Displays changes in your files, such as added, removed, or modified lines.
* Helps you compare files in the **working directory**, **staging area**, or **local repository** to track progress or spot errors.

**Basic Usage**

* **Command**: git diff
  + Shows changes in the **working directory** (modified but unstaged files) compared to the last commit.
  + Output includes:
    - File names.
    - Lines added (marked with +) or removed (marked with -).
    - Context lines for reference.
  + *Example*: git diff
    - Shows changes in labnotes.txt since the last commit (e.g., +Added new experiment data).

**Useful Variations**

1. **Compare Staged Changes**:
   * **Command**: git diff --staged (or git diff --cached)
   * Shows differences between the **staging area** and the last commit.
   * *Example*: git diff --staged
     + Displays changes in staged files (e.g., after git add labnotes.txt).
2. **Compare Specific Files**:
   * **Command**: git diff <filename>
   * Shows changes for a specific file in the working directory.
   * *Example*: git diff labnotes.txt
     + Shows only changes in labnotes.txt.
3. **Compare Commits**:
   * **Command**: git diff <commit1> <commit2>
   * Shows differences between two commits (use commit IDs from git log).
   * *Example*: git diff a1b2c3d e4f5g6h
     + Compares changes between two specific commits.
4. **Compare Branches**:
   * **Command**: git diff <branch1> <branch2>
   * Shows differences between two branches.
   * *Example*: git diff main experiment
     + Displays differences between the main and experiment branches.
5. **Compact Summary**:
   * **Command**: git diff --summary
   * Shows a summary of changed files without detailed line-by-line differences.
   * *Example*: git diff --summary
     + Lists files that were modified, added, or deleted.

**Why It’s Useful for Lab Work**

* Review changes to scripts, data, or notes before saving them.
* Check what’s staged to ensure only intended changes are committed.
* Compare versions to debug or verify experiment updates.
* Understand differences when collaborating or merging branches.

**Tips**

* Use q to exit the diff view in the terminal.
* Combine with git status to see which files are modified before running git diff.
* Redirect output to a file for review: git diff > changes.txt.
* Use a GUI tool (e.g., GitKraken) if the terminal output is hard to read.

The git diff command is essential for checking and understanding changes in your lab projects, ensuring you stay in control of your work!

**9. Helix Visual Merge Tool (P4Merge)**

The Helix Visual Merge Tool (P4Merge) is a free, user-friendly tool from Perforce for comparing and merging files, ideal for developers and researchers managing version control. It visually displays differences between text or image files (e.g., JPEG, PNG) with color-coded highlights, supporting side-by-side or three-way comparisons. P4Merge allows you to resolve merge conflicts, edit text differences, and ignore line endings or whitespace for flexibility. It integrates with Git and other version control systems, making it valuable for lab work to track changes in scripts or data. You can download it from the Perforce website and configure it easily for use with Git.[](<https://www.perforce.com/products/helix-core-apps/merge-diff-tool> p4merge)[](<https://www.perforce.com/downloads/visual-merge-tool> )

**10. Removing Files with Git Rm**

 **Purpose**: Removes files from Git tracking and (optionally) working directory.

 **Basic Command**: git rm <filename>

* Deletes file and stages removal.
* *Example*: git rm olddata.txt

 **Stop Tracking Only**: git rm --cached <filename>

* Keeps file in working directory, stops Git tracking.
* *Example*: git rm --cached temp.txt

 **Remove Directory**: git rm -r <directory>

* Deletes folder and its files.
* *Example*: git rm -r old\_experiment/

 **Next Step**: Commit with git commit -m "Removed file".

 **Use for Lab**: Clean up outdated scripts or data, organize projects.

 **Tips**:

* Check with git status after git rm.
* Undo with git restore <filename> before committing.
* Be cautious with git rm -r.

**11. Undoing Changes with Git Checkout**

The git checkout command in Git helps you undo changes in your working directory or switch to a specific state of your project. It’s useful for reverting unwanted changes to files in your lab work before they are staged or committed.

* Purpose: Restores files in the working directory to their last committed state or switches to a specific commit or branch.
* Basic Command: git checkout -- <filename>
  + Discards changes to a file in the working directory, reverting it to the last commit.
  + *Example*: git checkout -- labnotes.txt
    - Reverts changes in labnotes.txt to the last committed version.
* Undo Multiple Files: git checkout -- .
  + Discards all uncommitted changes in the working directory.
  + *Example*: git checkout -- .
    - Reverts all modified files to their last committed state.
* Switch to a Commit: git checkout <commit-id>
  + Moves to a specific commit (use commit ID from git log).
  + *Example*: git checkout a1b2c3d
    - Views the project state at that commit (detached HEAD mode).
* Use for Lab: Revert accidental changes to scripts, data, or notes before staging/committing.
* Tips:
  + Use git status to check modified files before undoing.
  + Changes are lost permanently unless staged or committed.
  + Use git checkout <branch> (e.g., git checkout main) to return to a branch from a commit.
  + For Git versions 2.23+, git restore <filename> is an alternative to git checkout -- <filename>.

**12. Git References (Master and HEAD)**

Git uses references like master (or main) and HEAD to point to specific commits or branches in your repository, helping you navigate and manage your lab project’s history.

* Master (or Main):
  + The default branch in a Git repository, often named master or main.
  + Represents the primary line of development for your project.
  + *Example*: git checkout main
    - Switches to the main branch to work on the latest project state.
  + *Use in Lab*: Stores the stable version of your scripts, data, or experiment notes.
* HEAD:
  + A pointer to the current commit or branch you’re working on.
  + Indicates where Git is currently “looking” in the repository.
  + *Example*: git log HEAD
    - Shows the commit history up to the current HEAD position.
  + In a “detached HEAD” state, HEAD points to a specific commit instead of a branch (e.g., after git checkout <commit-id>).
  + *Use in Lab*: Helps you track your current position in the project’s history.
* How They Work Together:
  + HEAD usually points to the latest commit in the main (or master) branch.
  + When you make a commit, HEAD and the branch (e.g., main) move to the new commit.
  + Switching branches (e.g., git checkout feature) updates HEAD to point to that branch.
* Tips:
  + Check HEAD position with cat .git/HEAD (shows the current branch or commit).
  + Use git branch to see all branches and confirm the active one (marked with \*).
  + Avoid working in detached HEAD unless reviewing a specific commit; create a branch with git branch new-branch to save changes.
  + Many repositories now use main instead of master as the default branch name.
* Use for Lab Work:
  + Keep stable project versions in main.
  + Use HEAD to track your current work or revert to specific commits for experiments.

**13. Git Reset Command**

Used to undo commits or changes in your working directory or staging area.

Syntax: git reset [mode] [commit]

Modes:

* --soft → Move HEAD to commit, keep index & working directory.
* --mixed (default) → Move HEAD & reset index, keep working directory.
* --hard → Move HEAD, reset index & working directory (DANGER: discards changes).

Examples:

git reset --soft HEAD~1 # Undo last commit, keep changes staged

git reset --mixed HEAD~1 # Undo last commit, keep changes unstaged

git reset --hard HEAD~1 # Undo last commit, discard all changes

**14. Git Aliases**

Used to create shortcuts for Git commands to save time.

**Syntax: git config --global alias.<shortcut> "<command>"**

**Examples:**

**git config --global alias.st "status" # git st → git status**

**git config --global alias.co "checkout" # git co → git checkout**

**git config --global alias.ci "commit" # git ci → git commit**

**git config --global alias.br "branch" # git br → git branch**

**To view aliases: git config --get-regexp alias**

**15. Ignoring Files with .gitignore**

.gitignore specifies files/folders Git should ignore (not track).

**Steps:**

* 1. Create a .gitignore file in your repo.
  2. Add patterns to ignore.

**Example contents:**

\*.log # Ignore all .log files

node\_modules/ # Ignore node\_modules folder

secret.txt # Ignore specific file

Command:

git status # See ignored files won't show up

**16. Special Treatment for Directories**

Git **tracks files**, not empty directories.  
To include an empty directory, add a placeholder file (e.g., .gitkeep).

Example:

mkdir data

touch data/.gitkeep

git add data/.gitkeep

In .gitignore, to ignore a directory: logs/ # Ignores entire logs directory

**17. Branching and Merging**

**Branching:** Create independent lines of development.  
**Merging:** Combine changes from one branch into another.

Commands:

git branch new-feature # Create branch

git checkout new-feature # Switch to branch

git merge new-feature # Merge into current branch

git branch -d new-feature # Delete branch after merge

Example:

git checkout -b feature1 # Create + switch to feature1

git checkout main

git merge feature1 # Merge feature1 into main

**18. Merging with Rebase**

**Rebase:** Moves your branch commits on top of another branch for a cleaner history.

Command: git rebase main # Reapply commits from current branch onto main

Example flow:

git checkout feature

git rebase main # Update feature branch with main's latest commits

After rebase:

git checkout main

git merge feature # Fast-forward merge (clean history)

Use carefully if sharing branches (can rewrite history).

**19. Stashing in Git**

**Stash:** Temporarily save uncommitted changes without committing.

Commands:

git stash # Save changes

git stash list # Show stashes

git stash apply # Reapply latest stash

git stash pop # Apply + remove from stash list

git stash drop # Remove specific stash

Example:

git stash

git pull

git stash pop

**20. Working with Remote Repositories**

Remote repositories let you collaborate and sync code.

**Common commands:**

git remote -v # List remotes

git remote add origin <url> # Add remote

git push -u origin main # Push to remote main branch

git pull origin main # Fetch + merge from remote

git clone <url> # Copy remote repo locally

git fetch origin # Download updates (no merge)

**21. Git Tagging**

**Tags** mark specific points in history (e.g., releases).

**Commands:**

git tag v1.0 # Create lightweight tag

git tag -a v1.0 -m "v1.0" # Create annotated tag

git tag # List tags

git show v1.0 # Show tag details

git push origin v1.0 # Push tag to remote

git push origin --tags # Push all tags

**22. Git Revert Command**

Revert creates a new commit that undoes changes from a specific commit (safe for shared history).

Command: gi t revert <commit>

Example: git revert HEAD~1 # Revert the previous commit

Useful for undoing commits without rewriting history (unlike reset).

**23. Cherry-Picking**

Apply a specific commit from another branch onto your current branch.

Command: git cherry-pick <commit>

Example: git cherry-pick a1b2c3d # Apply commit with hash a1b2c3d

May cause conflicts if changes overlap.

**24. Git Reflog Command**

Records all changes to the HEAD (including resets, checkouts).

Command: git reflog

Usage:

* Helps recover lost commits or branches.
* Shows history of HEAD movements with commit hashes.

Example:

git reflog

git checkout <commit\_hash> # Restore to a previous state

**25. Advanced Git Tagging**

Tags with signing, verification, and deletion.

Create a signed tag: git tag -s v1.1 -m "Signed release v1.1"

Verify a tag: git tag -v v1.1

Delete a local tag: git tag -d v1.1

Delete a remote tag: git push origin --delete tag v1.1

**26. Advanced Git Revert**

Revert multiple commits or ranges safely.

Revert multiple commits (one by one): git revert <commit1> <commit2> <commit3>

Revert a range of commits: git revert <oldest\_commit>^..<newest\_commit>

Options:

* --no-commit to stage revert changes without committing:

git revert --no-commit <commit>

**27. Advanced Cherry-Picking**

Apply specific commits with options for conflict resolution and multiple commits.

Cherry-pick multiple commits:

git cherry-pick <commit1> <commit2> <commit3>

Cherry-pick a range of commits:

git cherry-pick <oldest\_commit>^..<newest\_commit>

Options:

* -n or --no-commit: Apply changes but don’t commit automatically.

git cherry-pick -n <commit>

--edit: Edit commit message during cherry-pick.

git cherry-pick --edit <commit>