DAYANANDA SAGAR ACADEMY OF TECHNOLOGY AND MANAGEMENT

Udayapura, Kanakapura Road, Opp. Art of Living, Bangalore – 560082

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

(Affliated to VTU, Belagavi, Approved by AICTE, New Delhi, Accredited by NBA for 3 years, New Delhi)

2024-2025

Programming with GIT

LABORATORY MANUAL

(23CSE47)

Dr. C Nandini
HOD,
CSE, DSATM

Dr. M Ravishankar
Principal, DSATM

VISION AND MISSION OF THE INSTITUTE

Vision of the Institution:

To strive at creating the institution a center of highest caliber of learning, so as to create an overall intellectual atmosphere with each deriving strength from the other to be the best of engineers, scientists with management & design skills.

Mission of the Institution:

- **To serve its region, state, the nation and globally by preparing students to make meaningful contributions in an increasing complex global society challenges.**
- **To encourage, reflection on and evaluation of emerging needs and priorities with state of art infrastructure at institution.**
- **To support research and services establishing enhancements in technical, economic, human and cultural development.**
- **To establish inter disciplinary center of excellence, supporting/ promoting student's implementation.**
- **To increase the number of Doctorate holders to promote research culture on campus.**
- **To establish IIPC, IPR, EDC, innovation cells with functional MOU's supporting student's quality growth.**

QUALITY POLICY

Dayananda Sagar Academy of Technology and Management aims at achieving academic excellence through continuous improvement in all spheres of Technical and Management education. In pursuit of excellence cutting-edge and contemporary skills are imparted to the utmost satisfaction of the students and the concerned stakeholders.

1. Vision and Mission of the Department

Department Vision

Epitomize CSE graduate to carve a niche globally in the field of computer science to excel in the world of information technology and automation by imparting knowledge to sustain skills for the changing trends in the society and industry.

Department Mission

M1: To educate students to become excellent engineers in a confident and creative environment through world-class pedagogy.

M2: Enhancing the knowledge in the changing technology trends by giving hands-on experience through continuous education and by making them to organize & participate in various events.

M3: Impart skills in the field of IT and its related areas with a focus on developing the required competencies and virtues to meet the industry expectations.

M4: Ensure quality research and innovations to fulfill industry, government & social needs.

M5: Impart entrepreneurship and consultancy skills to students to develop self-sustaining life skills in multi-disciplinary areas.

2. Programme Educational Objectives

PEO1: Engage in professional practice to promote the development of innovative systems and optimized solutions for Computer Science and Engineering.

PEO2: Adapt to different roles and responsibilities in interdisciplinary working environment by respecting professionalism and ethical practices within organization and society at national and international level.

PEO3: Graduates will engage in life-long learning and professional development to acclimate the rapidly changing work environment and develop entrepreneurship skills

3. Program Outcomes (POs)

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 modelling 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.

4. Programme Specific Outcomes

PSO1: **Foundation of Mathematical Concepts:** Ability to use mathematical methodologies to solve the problem using suitable mathematical analysis, data structure and suitable algorithm.

PSO2: **Foundation of Computer System:** Ability to interpret the fundamental concepts and methodology of computer systems. Students can understand the functionality of hardware and software aspects of computer systems.

PSO3: **Foundations of Software Development:** Ability to grasp the software development lifecycle and methodologies of software systems. Possess competent skills and knowledge of software design process. Familiarity and practical proficiency with a broad area of programming concepts and provide new ideas and innovations towards research.

PSO4: **Foundations of Multi-Disciplinary Work:** Ability to acquire leadership skills to perform professional activities with social responsibilities, through excellent flexibility to function in multi-disciplinary work environment with self-learning skills.

Course Details

Course Name **:** Programming with GIT

Course Code **: 23CSE47**

Course Objectives

CO 1	Apply fundamental Git commands to manage and navigate a Git repository.
CO 2	Design and organize branches within a Git repository to facilitate parallel development and version control
CO 3	Implement Git commands to collaborate with others and manage remote repositories effectively.
CO 4	Utilize Git commands to manage tags, create releases, and perform advanced Git operations
CO 5	Examine and modify the Git history to maintain a clean and accurate project history.

Dayananda Sagar Academy of Technology & Management

(Autonomous Institute under VTU)

Semester	**:**	**4th**				
Course Title	**:**	**Programming with Git**				
Course Code	**:**	**23CSE47**				
Course Type (Theory/ Practical/ ntegrated)	**:**	**Practical – Experiential**				
Category	**:**	**AEC**				
Stream	**:**	**CSE**		**CIE**	**:**	**50 Marks**
Teaching hours/ week (L:T:P:S)	**:**	**0:0:2:0**		**SEE**	**:**	**50 marks**
Total Hours	**:**	**24**		**SEE Duration**	**:**	**2 hrs**
Credits	**:**	**1**				

Course Learning Objectives: Students will be able to:

Sl. No	Course Objectives
1	Understand the core concepts of Git and version control.
2	Learn to manage repositories, branches, and merges.
3	Master collaboration workflows and resolve conflicts.
4	Apply best practices in version control.

Teaching-Learning Process

Pedagogical Initiatives:

Some sample strategies to accelerate the attainment of various course outcomes are listed below:

- Adopt different teaching methods to attain the course outcomes.
- Include videos to demonstrate various concepts in C.
- Encourage collaborative (Group) Learning to encourage team building.
- Ask at least three **HOTS (Higher-order Thinking Skills)** module-wise questions to promote critical thinking.
- Adopt **Problem-Based Learning (PBL),** which fosters students' analytical skills, and develops thinking skills such as evaluating, generalizing, and analyzing information rather than simply recalling it.
- Show different ways to solve a problem and encourage the students to come up with creative and optimal solutions.
- Discuss various case studies to map with real-world scenarios and improve the understanding.
- Devise innovative pedagogy to improve **Teaching-Learning Process (TLP)**.

DSATM

Scheme of Teaching and Examinations for BE Programme -2024-25
Outcome Based Education and Choice Based Credit System (CBCS)
(Effective from the Academic Year 2024-25)

COURSE CURRICULUM

Module No.	Topics	Cos
1	**Module 1: Introduction to Git** Version Control Systems (VCS) Overview, Introduction to Git, Installing Git and Setting Up, Basic Git Commands: git init, git clone, git add, git commit, git status **Experiment 1:** • Setting up Git on local machines • Creating a new repository • Making and tracking changes • Viewing the commit history	**CO1**
Pedagogy		
2	**Module 2: Working with Repositories** Remote Repositories: git remote, git push, git pull, git fetch, Cloning repositories and working with remotes **Experiment 2:** • Cloning an existing repository • Pushing changes to a remote repository **Experiment 3:** • Pulling updates from a remote repository • Exploring git fetch and git merge	**CO3**
Pedagogy		
3	**Module 3: Branching and Merging** Branching in Git: git branch, git checkout, git switch**,** Merging branches: git merge, git rebase **Experiment 4:** • Creating and switching branches • Merging branches and resolving conflicts • Rebasing branches and understanding rebase vs. merge **Experiment 5:** Write the commands to stash your changes, switch branches, and then apply the stashed changes.	**CO2**
Pedagogy		
4	**Module 4: Collaboration Workflows** Collaboration Workflows: Centralized, Feature Branch, Forking**,** Pull Requests and Code Reviews**,** Managing Conflicts and Best Practices. **Experiment 6:** • Implementing a feature branch workflow	**CO3**

	• Creating pull requests • Conducting code reviews and resolving conflicts	
Pedagogy		
5	**Module 5: Advanced Git Concepts** Stashing Changes: git stash**,** Interactive Rebase and Amend: git rebase -i, git commit –amend**,** Git Tags and Releases: git tag**,** Undoing Changes: git reset, git revert **Experiment 7:** • Using git stash to manage work in progress • Performing an interactive rebase • Tagging commits and creating releases • Resetting and reverting changes **Experiment 8:** Write the command to display the last five commits in the repository's history. **Experiment 9:** Write the command to cherry-pick a range of commits from "source-branch" to the current branch. **Experiment 10:** a. Write the command to display the last five commits in the repository's history. b. Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31"	**CO4, CO1, CO5**
	Pedagogical Initiatives (Not limited to): • **Think Pair and Share (Blended Learning):** provides an opportunity for students to learn from one another • **Problem Solving**: encourages cognitive thinking and enables creative problem solving • **Poster Presentation:** allows students to represent the concepts visually in order to understand the topics easily. • **Case studies:** maps different domains in real time applications • **Demonstration:** exhibits the implementation process	

Text Books	
1	"Pro Git" by Scott Chacon and Ben Straub
2	Version Control with Git, 3rd Edition, by Prem Kumar Ponuthorai, Jon Loeliger Released October 2022, Publisher(s): O'Reilly Media, Inc.
Reference Books	
1	https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013094443347369984278 2_shared/overview
2	https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_013301347121774592119 26_shared/overview

Course Outcomes: At the end of the course, the student will be able to:

CO	Course Outcomes	RBT Level	RBT Level Indicator
CO1	Apply fundamental Git commands to manage and navigate a Git repository.	**Applying**	**L3**
CO2	Design and organize branches within a Git repository to facilitate parallel development and version control	**Creating**	**L6**
CO3	Implement Git commands to collaborate with others and manage remote repositories effectively.	**Applying**	**L3**
CO4	Utilize Git commands to manage tags, create releases, and perform advanced Git operations	**Applying**	**L3**
CO5	Examine and modify the Git history to maintain a clean and accurate project history.	**Analyzing**	**L4**

Mapping of Course Outcomes to Program Outcomes:

CO/PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1															
CO2	**3**														
CO3			**3**												
CO4			**3**												
CO5				**3**											

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.

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

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.

To download

https://git-scm.com/download/win

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.

Git Commands List

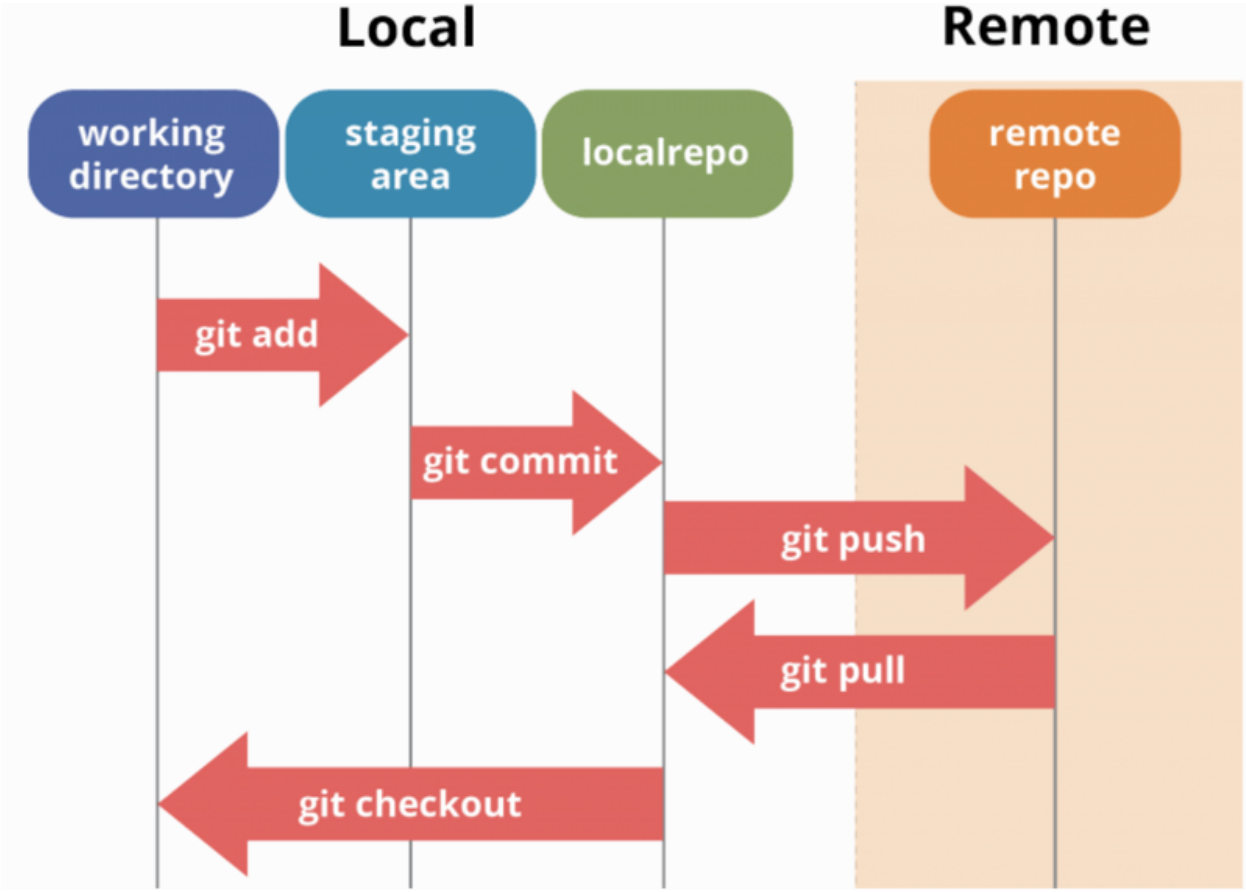
Git is a popular version control system used for tracking changes in software development projects. Here's a list of common Git commands along with brief explanations:

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 checked-out 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 mm <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 Git on local machines
- Creating a new repository
- Making and tracking changes
  - Viewing the commit history

Setting Up and Basic Commands:

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

Solution:

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

1. Open your terminal and navigate to the directory where you want to create the Git repository.
2. Initialize a new Git repository in that directory:

$ git init

1. Create a new file in the directory. For example, let's create a file named "my_file.txt."

You can use any text editor or command-line tools to create the file.

2. Add the newly created file to the staging area. Replace "my_file.txt" with the actual name of your file:

$ git add my_file.txt

This command stages the file for the upcoming commit.

1. Commit the changes with an appropriate commit message. Replace "Your commit message here" with a meaningful description of your changes:

$ git commit -m "Your commit message here"

Your commit message should briefly describe the purpose or nature of the changes you made.

For example:

$ git commit -m "Add a new file called my_file.txt"

After these steps, your changes will be committed to the Git repository with the provided commit message. You now have a version of the repository with the new file and its history stored in Git.

To make and track changes in Git, follow these steps:

1. Set Up Git (if not done already)

First, make sure you have Git installed. If you haven't done so, you can install Git from here.

Then, configure Git with your name and email:

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

2. Initialize a Git Repository

In your project folder, initialize a Git repository by running:

```
git init
```

This creates a .git directory that tracks changes in the repository.

3. Add Files to Track Changes

Once you've made some changes in your project, you need to add them to the staging area:

```
git add <file_name>
```

Or to add all files:

```
git add .
```

4. Commit Changes

After adding files to the staging area, commit them:

```
git commit -m "A message describing the changes"
```

The commit message should explain what changes have been made (e.g., "Fix typo in README").

5. Check Status

To check the status of your files (whether they are staged, modified, etc.), run:

git status

6. View Commit History

To see the history of commits, run:

git log

This shows a list of commits, with the commit hash, author, date, and commit message.

You can also use git log --oneline for a condensed view.

7. Track Changes with Diff

To see what has changed in the working directory since the last commit, use:

git diff

This will show you the exact differences (line-by-line) between the current state and the last commit.

8. Undoing Changes

- **Unstaging a file**: If you accidentally added a file to the staging area but haven't committed it:
- git reset <file_name>
- **Undoing local changes**: To discard changes you made to a file (before committing):
- git checkout -- <file_name>
- **Resetting to a previous commit**: If you want to go back to a previous commit:
- git reset --hard <commit_hash>

This removes all the changes made after that commit.

Useful Git Commands to Track Changes

- **git log**: View commit history.
- **git status**: Check the state of your working directory.
- **git diff**: See what's changed in your files.

- **git show <commit_hash>**: View details of a specific commit.

With these commands, you'll be able to make and track changes in your Git repository effectively.

Using Git to Track Changes in a Text Document

Git excels in version controlling **text documents**. This means that you can:

- **Track changes** made to a text document (e.g., a .txt, .md, .html file).
- **Commit changes** to these documents and push them to a remote repository.
- **View diffs** and **rollback changes** in the text documents.

Example:

1. Create a text file (e.g., notes.txt) and initialize a Git repository in that folder:

   bash
   Copy
   git init

2. Add your text document:

   bash
   Copy
   git add notes.txt

3. Commit the document:

   bash
   Copy
   git commit -m "Initial notes"

4. Make edits to the text document (notes.txt).
5. Stage and commit the changes:

   bash
   Copy
   git add notes.txt
   git commit -m "Updated notes"

6. Push the changes to a remote repository (e.g., GitHub):

   bash
   Copy
   git push origin main

Experiment 2:

- Cloning an existing repository
- Pushing changes to a remote repository

To create a remote repository and push changes from your local Git repository, follow these steps. I'll show you how to do it using GitHub, but the process is similar for other services like GitLab or Bitbucket.

1. Create a Remote Repository

On GitHub:

1. Go to GitHub and log in to your account.
2. In the top-right corner, click the **"+"** icon and select **New repository**.
3. Choose a name for your repository, and optionally, add a description.
4. Select the visibility (public or private).
5. **Do NOT initialize the repository with a README, .gitignore, or license**, as you'll be linking your local repository to this remote.
6. Click **Create repository**.

You'll be taken to a page with instructions to push your local repository to GitHub.

2. Link Your Local Repository to the Remote

In your local terminal, navigate to your project folder and initialize Git (if you haven't already):

git init

Now, link your local repository to the remote repository you created on GitHub.

1. Copy the remote URL from GitHub. It will look like one of these:
    - HTTPS: https://github.com/username/repository-name.git
    - SSH: git@github.com:username/repository-name.git
2. Add the remote repository using the following command:

3. git remote add origin <repository_url>

   For example:

   git remote add origin https://github.com/username/my-repository.git

3. Push Your Changes to the Remote Repository

If You Have Already Made Commits:

If you've made changes and committed them locally, you can push them to GitHub using:

git push -u origin master

Explanation:

- origin refers to the remote repository you added earlier.
- master (or main, depending on your repository) is the branch you want to push your changes to. Note that many repositories are now using main as the default branch instead of master, so if your remote repository uses main, you should push to that instead.

If you're pushing for the first time and the default branch is main, use:

git push -u origin main

If You Haven't Made Any Commits Yet:

If you haven't made any commits yet, you can still push the initial commit to the remote repository:

1. First, add your files to the staging area:
2. git add .
3. Commit your changes:
4. git commit -m "Initial commit"
5. Then, push your changes to the remote repository:
6. git push -u origin master

(or main if you're using that branch name).

4. Verify the Push

After pushing, you can verify that your changes were successfully uploaded by visiting the GitHub repository page. You should see your files and commit history there.

5. Pull Changes from Remote Repository

If other people are working on the project or if you want to sync with the remote repository, you can pull the latest changes from the remote:

git pull origin master

or

git pull origin main

Summary of Git Commands to Push Changes:

1. **Initialize Git (if needed)**:
2. git init
3. **Add remote URL**:
4. git remote add origin <repository_url>
5. **Stage and commit your changes**:
6. git add .
7. git commit -m "Your commit message"
8. **Push changes to the remote repository**:
9. git push -u origin master # or 'main' for newer repositories

After you've set this up, pushing changes to the remote repository is as simple as running git push whenever you make new commits.

To clone an existing repository (i.e., download a copy of a remote Git repository to your local machine), follow these steps:

1. Find the Repository URL

You need the URL of the repository you want to clone. This can be found on the repository page on GitHub, GitLab, Bitbucket, or other Git hosting services.

- For **GitHub**, navigate to the repository you want to clone.
- Click the **green "Code" button** on the repository page.
- You'll see options for **HTTPS** and **SSH**. Choose the one that works for you:
    - **HTTPS** (if you're unsure about SSH keys): https://github.com/username/repository-name.git
    - **SSH** (if you've set up SSH keys with GitHub): git@github.com:username/repository-name.git

2. Clone the Repository

Once you have the repository URL, follow these steps:

Open your terminal (Command Prompt, Git Bash, or terminal in your IDE):

1. **Navigate to the directory** where you want to clone the repository:
2. cd path/to/your/directory
3. **Run the clone command** with the URL of the repository:
4. git clone <repository_url>

   Example for HTTPS:

   git clone https://github.com/username/repository-name.git

   Example for SSH:

   git clone git@github.com:username/repository-name.git

3. Navigate into the Cloned Repository

After cloning, Git creates a new folder with the same name as the repository (unless you specify a different name).

Change to that directory:

cd repository-name

4. Verify the Clone

To check if the repository has been successfully cloned, you can list the files:

ls

You should see all the files and directories from the remote repository.

Also, check the remote URL using:

git remote -v

This should show the URL of the remote repository (e.g., origin https://github.com/username/repository-name.git).

5. Pull Latest Changes (if needed)

If you've already cloned the repository earlier, but want to get the latest changes, you can always pull the latest updates from the remote:

git pull origin main

or if it's using master as the default branch:

git pull origin master

Summary of Commands:

1. **Clone a repository:**
2. git clone <repository_url>
3. **Navigate into the cloned repository folder:**
4. cd repository-name
5. **Check remote repository URL:**
6. git remote -v

Cloning a repository is a simple way to get started with an existing project, and you'll be able to track changes, create branches, and push updates just like any other Git repository.

Experiment 3:

- Pulling updates from a remote repository
  - Exploring git fetch and git merge

Pulling updates from a remote repository allows you to fetch and merge changes that others have made (or that you've made on another machine) into your local repository. Here's how to do it:

1. Navigate to Your Local Repository

First, make sure you're in the local repository that you want to update. Open your terminal and navigate to the project folder:

cd path/to/your/repository

2. Pull Latest Changes from the Remote Repository

To pull the latest changes from the remote repository and merge them into your current branch, use the git pull command:

git pull origin <branch_name>

Explanation:

- origin: This is the default name for your remote repository. It points to the URL of the remote repository you cloned or linked.
- <branch_name>: The name of the branch you want to pull changes from. Typically, this will be main or master, but it could be any branch you're working on.

For example:

- If you're working on the main branch:
- git pull origin main
- Or if your default branch is master:
- git pull origin master

3. What Happens When You Pull?

When you pull changes, Git performs two steps:

1. **Fetch**: It retrieves the latest changes from the remote repository.
2. **Merge**: It merges the fetched changes into your local branch. If you're working on the same branch as the one you're pulling from (e.g., main), these changes will automatically be merged.

4. Handling Merge Conflicts

If there are conflicting changes between your local branch and the remote branch, Git will notify you of a **merge conflict**. You'll need to resolve the conflict manually before proceeding.

- Git will mark the conflicting sections of files with <<<<<<<, =======, and >>>>>>> so you can edit the file to keep the changes you want.
- After resolving the conflict, you'll need to add the resolved files:
- git add <file_name>
- Finally, commit the resolved changes:
- git commit -m "Resolved merge conflict"

5. Check the Status After Pulling

You can always check the status of your repository after pulling:

git status

This will show you any new changes, uncommitted files, or merge conflicts.

6. Verify the Update

After pulling, you can check the commit history to verify the update:

git log --oneline

This shows the commit history, and you should see the recent changes from the remote repository.

7. Optional: Pulling All Branches

If you want to update all of your branches (not just the current branch), you can use:

git fetch --all

This command fetches updates from all remotes without merging them into your local branches. You'd still need to manually merge each branch if you want to sync them.

Summary of Commands for Pulling Updates:

1. **Pull the latest changes from the remote repository**:
2. git pull origin <branch_name>

   Example:

   git pull origin main

3. **If there are merge conflicts**, resolve them manually, add the changes, and commit:
4. git add <file_name>
5. git commit -m "Resolved merge conflict"
6. **Check the status** of your repository:
7. git status
8. **View commit history** to verify changes:
9. git log --oneline

With this, you'll be able to keep your local repository up to date with the remote version and smoothly integrate any changes others have made.

Experiment 4:

- Creating and switching branches
- Merging branches and resolving conflicts
- Rebasing branches and understanding rebase vs. merge

Creating and Switching Branches

In **Git**, branches allow you to work on different versions of a project without affecting the main codebase. Here's how you can create and switch branches in Git:

---

1. Check Existing Branches

git branch

This command lists all local branches in your repository. The currently active branch will have an asterisk (*) next to it.

---

2. Create a New Branch

git branch <branch-name>

For example:

git branch feature-login

This creates a new branch called feature-login but does **not** switch to it.

---

3. Switch to a Branch

git checkout <branch-name>

OR (Recommended for newer Git versions)

git switch <branch-name>

For example:

```
git checkout feature-login
# OR
git switch feature-login
```

This switches to the feature-login branch.

---

4. Create and Switch to a New Branch (Shortcut)

git checkout -b <branch-name>

OR (For newer Git versions)

git switch -c <branch-name>

For example:

```
git checkout -b feature-logout
# OR
git switch -c feature-logout
```

This creates **and** switches to the feature-logout branch in one command.

---

5. Delete a Branch

git branch -d <branch-name>

If the branch has **unmerged changes**, use:

git branch -D <branch-name>

---

6. Switch to the Main Branch

```
git checkout main
# OR
git switch main
```

---

7. Merge a Branch into Main

Once you've finished working on a branch, merge it into the main branch:

```
git checkout main  # Switch to main
git merge <branch-name>  # Merge branch
```

For example:

```
git checkout main
git merge feature-login
```

---

8. Push a New Branch to Remote Repository

If you want to share your branch with others, push it to the remote repository:

```
git push -u origin <branch-name>
```

For example:

```
git push -u origin feature-login
```

Summary of Commands

Command	Description
git branch	List branches
git branch <name>	Create a branch
git checkout <name>	Switch to a branch
git checkout -b <name>	Create and switch to a branch
git switch <name>	Switch (alternative to checkout)
git switch -c <name>	Create and switch (alternative to checkout -b)
git branch -d <name>	Delete a branch
git merge <name>	Merge a branch into the current branch
git push -u origin <name>	Push a branch to remote

Merging Branches and Resolving Conflicts

Merging allows you to combine changes from one branch into another. However, sometimes conflicts arise if the same part of a file is modified in multiple branches. Here's how to **merge branches** and **resolve conflicts** in Git.

---

1. Merge a Branch into Another

Step 1: Switch to the Target Branch

Before merging, switch to the branch you want to merge changes into (usually main or develop):

```
git checkout main
# OR
git switch main
```

Step 2: Merge the Feature Branch

```
git merge <branch-name>
```

For example, if merging feature-login into main:

```
git merge feature-login
```

- If there are no conflicts, Git will automatically merge the branches.
- If conflicts exist, Git will notify you about **merge conflicts**.

---

2. Resolving Merge Conflicts

If there are conflicts, Git will display a message like:

```
CONFLICT (content): Merge conflict in filename.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Step 1: Identify Conflicted Files

Run:

```
git status
```

This will show which files have conflicts.

Step 2: Open the Conflicted Files

Open the conflicted files in a text editor. You will see sections like this:

<<<<<<< HEAD
This is the content from the main branch.
=======
This is the content from the feature branch.
>>>>>>> feature-login

- The section between <<<<<<< HEAD and ======= is from your **current branch**.
- The section between ======= and >>>>>>> feature-login is from the **feature branch** being merged.

Step 3: Resolve the Conflict

Edit the file to **keep the desired changes** and remove conflict markers (<<<<<<<, =======, >>>>>>>). For example, if you decide to keep the content from the feature branch, modify the file like this:

This is the content from the feature branch.

Step 4: Mark Conflict as Resolved

After resolving conflicts, **stage the changes**:

git add <filename>

For example:

git add filename.txt

Step 5: Complete the Merge

git commit -m "Resolved merge conflict between main and feature-login"

---

3. Abort a Merge (If Needed)

If you want to cancel the merge and reset everything:

git merge --abort

---

4. Merge Strategies

Git supports different merge strategies:

- **Fast-forward merge (default for linear history)**
  If no new commits were added to the main branch, Git will simply move the branch pointer forward.
- git merge --ff-only feature-login
- **Recursive merge (default when branches diverge)**
  If both branches have new commits, Git will perform a three-way merge.

---

5. Pushing Merged Changes to Remote

Once the merge is successful, push the updated branch:

git push origin main

Summary of Commands

Command	Description
git checkout main	Switch to the target branch
git merge <branch-name>	Merge the specified branch into the current branch
git status	Check for merge conflicts
git add <filename>	Mark conflicts as resolved
git commit -m "Resolved conflicts"	Complete the merge
git merge --abort	Cancel the merge
git push origin main	Push merged changes to the remote repository

---

Rebasing Branches and Understanding Rebase vs. Merge in Git

Git provides two primary ways to integrate changes from one branch into another:
- **Merging**
- **Rebasing**

Both serve the same purpose but work differently. Let's break it down.

---

1. What is Rebasing in Git?

Rebasing moves your feature branch's commits on top of the latest commits from another branch (usually main).
It **rewrites the commit history** to maintain a cleaner, linear sequence of commits.

How Rebasing Works

Suppose you have this commit history:

```
      A---B---C (feature-branch)
     /
D---E---F---G (main)
```

When you **merge**, Git creates a new **merge commit** (H):

```
      A---B---C
     /         \
D---E---F---G---H (main)
```

When you **rebase**, Git moves A-B-C on top of G:

```
D---E---F---G---A'---B'---C' (feature-branch)
```

Each commit (A, B, C) is **rebased** (rewritten) as a new commit (A', B', C').

---

2. How to Rebase a Branch

Step 1: Switch to Your Feature Branch

```
git checkout feature-branch
# OR
git switch feature-branch
```

Step 2: Rebase onto the Main Branch

git rebase main

- This applies all commits in feature-branch on top of the latest main commits.
- If there are no conflicts, rebase completes automatically.
- If conflicts occur, Git will stop and ask you to resolve them.

---

3. Resolving Conflicts During Rebase

If a conflict occurs, Git will show a message like:

CONFLICT (content): Merge conflict in file.txt

Step 1: Check Conflicted Files

git status

Step 2: Open and Fix Conflicts

Manually edit files to resolve conflicts and **remove conflict markers (**<<<<<<<, =======, >>>>>>>**)**.

Step 3: Continue the Rebase

git add <conflicted-file>
git rebase --continue

Step 4: Abort Rebase (If Needed)

If you want to cancel the rebase:

git rebase --abort

---

4. Pushing a Rebased Branch

Since rebasing **rewrites commit history**, you may need to force push:

git push --force-with-lease

□□ **Warning:** Force-pushing overwrites history and may cause issues if others are working on the same branch.

---

5. Rebase vs. Merge

Feature	Merge	Rebase
Commit History	Creates a new merge commit	Keeps a linear history
Preserves Context	Keeps all branches' history	Rewrites commit history
Best For	Preserving history in large teams	Keeping history clean for solo work
Command	git merge <branch>	git rebase <branch>

 Use Merge when working in a **team** to preserve commit history.
 Use Rebase when you want a **cleaner, linear history** (e.g., before merging a feature branch).

---

6. Interactive Rebase (Rewriting History)

You can **edit, reorder, or squash commits** interactively:

git rebase -i HEAD~3

This opens an editor where you can:

- pick → Keep commit as is
- squash → Merge commits
- edit → Modify commit message

Summary of Git Rebase Commands

Command	Description
git checkout feature-branch	Switch to the feature branch
git rebase main	Rebase feature branch onto the latest main
git status	Check for conflicts during rebase
git add <file>	Mark conflicts as resolved

Command	Description
git rebase --continue	Continue rebase after resolving conflicts
git rebase --abort	Cancel rebase and return to the previous state
git push --force-with-lease	Push rebased changes (use with caution)

- **Use merge** if you want to **preserve history** and avoid rewriting commits.
- **Use rebase** if you prefer a **clean and linear commit history**.
- **Avoid rebasing shared branches**, as it rewrites history and can cause conflicts.

Experiment 5:

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

Here are the Git commands to **stash your changes, switch branches, and then apply the stashed changes**:

1. Stash Your Changes

git stash

This temporarily saves your **uncommitted changes** so you can switch branches without committing them.

To stash changes with a custom message:

git stash push -m "Work in progress on feature-x"

---

2. Switch to Another Branch

git checkout main
OR
git switch main

Now you are on the main branch, and your changes are safely stashed.

---

3. Switch Back to Your Original Branch

git checkout feature-branch
OR
git switch feature-branch

---

4. Apply the Stashed Changes

git stash pop

This **applies the last stashed changes and removes them** from the stash list.

If you want to apply the changes **without removing them from the stash**, use:

git stash apply

---

5. View and Manage Stashed Changes

- List all stashed changes:
- git stash list
- Apply a specific stash:
- git stash apply stash@{1}
- Drop (delete) a specific stash:
- git stash drop stash@{0}
- Clear all stashed changes:
- git stash clear

Summary

Command	Description
git stash	Stash current changes
git stash push -m "message"	Stash with a custom message
git checkout <branch> / git switch <branch>	Switch to another branch
git stash pop	Apply and remove the latest stash
git stash apply	Apply the latest stash without removing it
git stash list	View all stashed changes
git stash drop stash@{n}	Delete a specific stash
git stash clear	Remove all stashes

Experiment 6:

- Implementing a feature branch workflow
- Creating pull requests
- Conducting code reviews and resolving conflicts

Implementing a **Feature Branch Workflow** in Git is a great way to collaborate effectively while keeping the main branch stable. Here's a step-by-step guide:

---

1. Clone the Repository

If you haven't already, clone the repository to your local machine:

```
git clone <repository-url>
cd <repository-name>
```

---

2. Create a New Branch for Your Feature

Always create a new branch for each feature you work on. Use a descriptive name that indicates the purpose of the branch.

```
git checkout -b feature-branch-name
```

Example:

```
git checkout -b feature-add-user-authentication
```

---

3. Work on Your Feature

Make the necessary changes, add new files, and modify existing ones.

---

4. Commit Your Changes

Stage and commit your changes with meaningful commit messages.

```
git add .
git commit -m "Added authentication functionality"
```

---

5. Push Your Feature Branch to Remote

Upload your branch to the remote repository to share it with others.

```
git push origin feature-branch-name
```

---

6. Create a Pull Request (PR)

Go to the repository on GitHub/GitLab/Bitbucket and create a Pull Request (PR) from your feature branch to the main branch (e.g., `main` or `develop`).

- Add a title and description explaining your changes.
- Request reviews from your teammates.

---

7. Review and Address Feedback

Your teammates will review your PR and suggest improvements. Make the necessary changes and commit them:

```
git add .
git commit -m "Refactored authentication logic based on feedback"
git push origin feature-branch-name
```

---

8. Merge the Feature Branch

Once approved, merge the feature branch into `main` (or `develop`). You can either:

- **Merge via the GitHub UI** (Recommended)
- **Merge locally**:
- `git checkout main`
- `git pull origin main`
- `git merge feature-branch-name`
- `git push origin main`

---

9. Delete the Feature Branch

After merging, delete the branch to keep the repository clean:

```
git branch -d feature-branch-name
git push origin --delete feature-branch-name
```

---

10. Pull the Latest Changes

Ensure your local repository is up-to-date before starting new work:

```
git checkout main
git pull origin main
```

6b. Creating a Pull Request (PR) in GitHub/GitLab/Bitbucket

A **Pull Request (PR)** is how you propose changes from a feature branch to be merged into the main branch (`main` or `develop`). Here's how you can do it:

---

1. Push Your Feature Branch to Remote

After making changes and committing them locally, push your branch to the remote repository:

```
git push origin feature-branch-name
```

Example:

```
git push origin feature-login-auth
```

---

2. Open a Pull Request (PR)

GitHub

1. Go to your repository on GitHub.
2. Click on the **"Pull Requests"** tab.
3. Click **"New pull request"**.
4. Select your **feature branch** as the source and **main** (or `develop`) as the target.
5. Add a **title** and **description** explaining what the PR does.
6. Assign **reviewers** (team members).
7. Add **labels** (e.g., `feature`, `bugfix`).
8. Click **"Create Pull Request"**.

GitLab

1. Navigate to **Merge Requests** in your GitLab repository.
2. Click **"New Merge Request"**.
3. Select the **source branch** (your feature branch) and **target branch** (main/develop).

4. Fill in the details (title, description, reviewers).
5. Click **"Submit Merge Request"**.

Bitbucket

1. Go to **Pull Requests** in your Bitbucket repository.
2. Click **"Create pull request"**.
3. Select the **source branch** (your feature branch) and **destination branch** (main/develop).
4. Add a **title**, **description**, and assign **reviewers**.
5. Click **"Create pull request"**.

---

3. Review and Address Feedback

- Your teammates will **review the PR** and provide feedback.
- Make necessary changes and commit them:
- `git add .`
- `git commit -m "Refactored authentication logic based on feedback"`
- `git push origin feature-branch-name`

---

4. Merge the Pull Request

Once approved, **merge the PR**:

- **GitHub**: Click **"Merge pull request"** and confirm.
- **GitLab**: Click **"Merge"**.
- **Bitbucket**: Click **"Merge"**.

Alternatively, you can merge locally:

```
git checkout main
git pull origin main
git merge feature-branch-name
git push origin main
```

---

5. Delete the Feature Branch

After merging, delete the branch to keep the repo clean:

```
git branch -d feature-branch-name
git push origin --delete feature-branch-name
```

6c. Conducting Code Reviews and Resolving Conflicts in Git

A **code review** ensures that code is high quality, follows best practices, and is free of errors before merging into the main branch. Resolving **merge conflicts** is a crucial step in this process.

---

1. Conducting a Code Review

Once a **Pull Request (PR)** is created, team members should review the code and suggest improvements.

How to Review a PR in GitHub, GitLab, or Bitbucket

1. Navigate to the **Pull Request / Merge Request** tab in your repository.
2. Open the PR and review the **title, description, and changes**.
3. Leave comments on specific lines of code if needed.
4. Approve or request changes:
    - □ Approve the PR if everything looks good.
    - □ Request changes if improvements are needed.
5. Assign the PR back to the developer.

---

2. Addressing Code Review Feedback

If changes are requested, the developer should:

1. Make the necessary updates in their feature branch.
2. Commit the changes:

```
git add .
git commit -m "Refactored authentication logic based on feedback"
```

5. Push the updated branch:

```
git push origin feature-branch-name
```

7. Notify the reviewer that changes have been made.

---

3. Resolving Merge Conflicts

A **merge conflict** happens when Git cannot automatically merge two branches due to overlapping changes.

Steps to Resolve Merge Conflicts Locally

1. **Pull the latest changes from `main`**

2. `git checkout feature-branch-name`
3. `git pull origin main`
4. If there are conflicts, Git will highlight them in the affected files:
5. `<<<<<<< HEAD`
6. `Code from main branch`
7. `=======`
8. `Code from feature branch`
9. `>>>>>>> feature-branch-name`
10. **Manually edit the file**, keeping the correct version.
11. **Stage and commit the resolved files**:
12. `git add resolved-file.js`
13. `git commit -m "Resolved merge conflict in resolved-file.js"`
14. **Push the updated branch**:
15. `git push origin feature-branch-name`

---

4. Merging the Pull Request

Once the PR is approved and conflicts are resolved:

- Click "**Merge Pull Request**" in GitHub/GitLab/Bitbucket.
- Or merge locally:
- `git checkout main`
- `git pull origin main`
- `git merge feature-branch-name`
- `git push origin main`

---

5. Delete the Feature Branch

Once merged, delete the branch:

```
git branch -d feature-branch-name
git push origin --delete feature-branch-name
```

Experiment 7:

- Using git stash to manage work in progress
- Performing an interactive rebase
- Tagging commits and creating releases
- Resetting and reverting changes

7a. Using `git stash` to Manage Work in Progress

Sometimes, you need to **temporarily set aside** your uncommitted changes (e.g., switching branches, fixing an urgent bug). Instead of committing half-done work, you can use `git stash`.

---

1. Stashing Your Changes

To save your **current uncommitted changes** without committing:

```
git stash
```

This moves your changes into the stash and restores your working directory to a clean state.

 Stashing with a Message (recommended for clarity):

```
git stash push -m "WIP: Refactoring login feature"
```

---

2. Viewing Stashed Changes

To see a list of your stashes:

```
git stash list
```

Example output:

```
stash@{0}: WIP: Refactoring login feature
stash@{1}: WIP: Fixing dashboard layout
```

---

3. Applying Stashed Changes

To bring back the most recent stash:

```
git stash apply
```

Or apply a specific stash:

```
git stash apply stash@{1}
```

Note: This keeps the stash in the list (it doesn't delete it).

---

4. Removing a Stash After Applying

If you no longer need a stash after applying it, use:

```
git stash drop stash@{0}
```

---

5. Popping a Stash (Apply & Remove in One Step)

To **apply and delete** the latest stash in one command:

```
git stash pop
```

---

6. Stashing Only Specific Files

If you want to stash only some files, use:

```
git stash push -m "WIP: Styling updates" -- path/to/file.js
```

---

7. Creating a Branch from a Stash

If you realize your stashed work should be in a new branch:

```
git stash branch new-feature-branch stash@{0}
```

This creates a new branch and applies the stash to it.

---

8. Clearing All Stashes

To delete **all** stashed changes:

```
git stash clear
```

7b. Performing an Interactive Rebase in Git □□

Interactive rebase (`git rebase -i`) allows you to **edit, combine, reorder, or remove commits** before merging them into a branch. It's useful for keeping a clean and readable commit history.

---

1. When to Use Interactive Rebase

□ **Clean up commit history** (squash multiple commits into one)
□ **Reorder commits** to make history more logical
□ **Edit commit messages** to improve clarity
□ **Remove unnecessary commits** (e.g., debugging logs)
□ **Split a commit** into smaller commits

---

2. Starting an Interactive Rebase

First, check your commit history:

```
git log --oneline
```

Example output:

```
a1b2c3d Fix typo in login page
e4f5g6h Add authentication feature
i7j8k9l Update README
m0n1o2p Initial commit
```

To **rebase the last N commits**, run:

```
git rebase -i HEAD~N
```

For example, to rebase the last **3 commits**:

```
git rebase -i HEAD~3
```

---

3. Understanding the Interactive Rebase Menu

After running `git rebase -i`, Git opens an editor (e.g., Vim) with a list of commits:

```
pick e4f5g6h Add authentication feature
pick a1b2c3d Fix typo in login page
pick i7j8k9l Update README
```

Each commit starts with `pick`. You can change this to modify commits.

Available Commands:

Command	Action
`pick`	Keep the commit as is
`reword`	Edit the commit message
`edit`	Modify the commit content
`squash`	Combine with the previous commit (keeps message)
`fixup`	Combine with the previous commit (drops message)
`drop`	Remove the commit

---

4. Common Use Cases

A. Squashing Multiple Commits Into One

If you have multiple commits that should be combined:

```
pick e4f5g6h Add authentication feature
squash a1b2c3d Fix typo in login page
squash i7j8k9l Update README
```

- Git will let you **edit the commit message** for the combined commit.
- Save and exit the editor.

B. Editing a Commit Message

Change `pick` to `reword`:

```
reword e4f5g6h Add authentication feature
pick a1b2c3d Fix typo in login page
pick i7j8k9l Update README
```

After saving, Git will prompt you to edit the commit message.

C. Reordering Commits

Simply **rearrange the order** of the commits in the editor:

```
pick i7j8k9l Update README
pick e4f5g6h Add authentication feature
pick a1b2c3d Fix typo in login page
```

Save and exit, and Git will replay the commits in the new order.

D. Editing a Previous Commit

If you need to modify a commit's content:

```
pick e4f5g6h Add authentication feature
edit a1b2c3d Fix typo in login page
pick i7j8k9l Update README
```

- Git will pause the rebase at that commit.
- Modify the files as needed.
- Stage and commit the changes:
- `git add .`
- `git commit --amend -m "Updated login page"`
- Continue the rebase:
- `git rebase --continue`

---

5. Dealing With Rebase Conflicts

If a conflict occurs during rebasing:

1. Git will stop and show a conflict message.
2. Open the conflicting files and **resolve the conflicts** manually.
3. Stage the resolved files:
4. `git add resolved-file.js`
5. Continue the rebase:
6. `git rebase --continue`
7. If needed, **abort the rebase** and go back to the original state:
8. `git rebase --abort`

---

6. Pushing Changes After Rebase

Since rebase rewrites history, you must **force push** the updated branch:

```
git push origin feature-branch –force
```

7c. Tagging Commits and Creating Releases in Git

Tagging in Git helps **mark important points** in history, such as version releases (e.g., `v1.0.0`). This is useful for versioning and release management.

---

1. Creating a Tag

Tags are like bookmarks for specific commits.

Lightweight Tag (Simple Reference)

```
git tag v1.0.0
```

- Creates a simple tag pointing to the latest commit.

Annotated Tag (Recommended for Releases)

```
git tag -a v1.0.0 -m "First stable release"
```

- Includes metadata (author, date, message) for better tracking.

---

2. Listing and Showing Tags

To **list all tags** in the repository:

```
git tag
```

To **view details of a tag**:

```
git show v1.0.0
```

---

3. Tagging an Older Commit

If you need to tag a past commit:

```
git tag -a v0.9.0 <commit-hash> -m "Pre-release version"
```

Find the commit hash using:

```
git log --oneline
```

---

4. Pushing Tags to Remote Repository

By default, tags **aren't pushed** to remote when using `git push`. You must explicitly push them:

Push a Specific Tag

```
git push origin v1.0.0
```

Push All Tags

```
git push origin --tags
```

---

5. Deleting Tags

If you need to remove a tag:

Delete a Local Tag

```
git tag -d v1.0.0
```

Delete a Remote Tag

```
git push --delete origin v1.0.0
```

---

6. Creating a GitHub/GitLab Release

GitHub Releases

1. Go to your repository on GitHub.
2. Navigate to the **"Releases"** tab.
3. Click **"Draft a new release"**.
4. Choose an existing tag or create a new one.
5. Add a **title, description**, and attach any necessary files.
6. Click **"Publish release"**.

GitLab Releases

1. Go to **"Deployments" → "Releases"**.
2. Click **"New release"**.
3. Select a tag and add a description.
4. Optionally attach release artifacts (binaries, docs).
5. Click **"Create release"**.

Experiment 8:

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

To display the last five commits in the repository's history, use:

```
git log --oneline -n 5
```

Explanation:

- `git log` → Displays commit history
- `--oneline` → Shows each commit in a single line (shorter output)
- `-n 5` → Limits output to the last **5 commits**

More detailed view, including commit messages and authors:

To display the last five commits with **detailed information** (including commit messages, authors, and timestamps), use:

```
git log -n 5 --pretty=format:"%h - %an, %ar : %s"
```

Explanation:

- `git log -n 5` → Shows the last **5 commits**
- `--pretty=format:"..."` → Customizes the log output:
  - `%h` → Short commit hash
  - `%an` → Author name
  - `%ar` → Relative commit time (e.g., *2 days ago*)
  - `%s` → Commit message

Example Output:

```
a1b2c3d - Alice, 2 hours ago : Fixed login bug
e4f5g6h - Bob, 1 day ago : Improved UI for dashboard
i7j8k9l - Charlie, 3 days ago : Added authentication middleware
m0n1o2p - Alice, 5 days ago : Updated README file
q3r4s5t - Bob, 1 week ago : Refactored user profile logic
```

Experiment 9:

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

To **cherry-pick a range of commits** from `"source-branch"` to the current branch, use:

```
git cherry-pick <start-commit-hash>^..<end-commit-hash>
```

Steps:

1. Switch to the branch where you want to apply the commits:
2. `git checkout target-branch`
3. Identify the commit hashes using:
4. `git log --oneline source-branch`
5. Cherry-pick the range of commits:
6. `git cherry-pick <start-commit-hash>^..<end-commit-hash>`

Example:

If you want to cherry-pick commits from `e4f5g6h` to `i7j8k9l`:

```
git cherry-pick e4f5g6h^..i7j8k9l
```

Note: The `^` before the first commit ensures that it includes `e4f5g6h` in the range.

Cherry-Pick Multiple Non-Consecutive Commits

If you need to cherry-pick specific commits (not a range), list them individually:

```
git cherry-pick e4f5g6h i7j8k9l m0n1o2p
```

Experiment 10:

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

To list all commits made by the author **"JohnDoe"** between **January 1, 2023, and December 31, 2023**, use:

```
git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31" --
pretty=format:"%h - %ad - %s" --date=short
```

Explanation:

- `--author="JohnDoe"` → Filters commits by the author name
- `--since="2023-01-01"` → Includes commits from January 1, 2023
- `--until="2023-12-31"` → Includes commits up to December 31, 2023
- `--pretty=format:"%h - %ad - %s"` → Custom output format:
  - `%h` → Short commit hash
  - `%ad` → Commit date
  - `%s` → Commit message
- `--date=short` → Formats the date as `YYYY-MM-DD`

Example Output:

```
a1b2c3d - 2023-03-15 - Fixed login bug
e4f5g6h - 2023-07-21 - Updated dashboard UI
i7j8k9l - 2023-11-05 - Improved API security
```

GIT Viva Questions

1. Git Basics

1. What is Git, and why is it used?
2. What is the difference between Git and GitHub/GitLab/Bitbucket?
3. How do you check the current status of a Git repository?
4. What is the command to initialize a Git repository?
5. How do you configure your username and email in Git?

---

2. Git Branching & Merging

6. What is a Git branch, and why is it used?
7. How do you create a new branch in Git?
8. How do you switch between branches?
9. What is the difference between **git merge** and **git rebase**?
10. What is the purpose of the `git checkout` and `git switch` commands?
11. How do you delete a branch locally and remotely?

---

3. Git Commit & History

12. What is the difference between **git commit** and **git push**?
13. How do you amend the last commit?
14. What is the command to see the commit history?
15. How do you check the details of a specific commit?
16. How can you revert a commit?

---

4. Git Stashing

17. What is `git stash`, and when should you use it?
18. How do you apply stashed changes?
19. How do you stash only specific files?
20. How do you delete all stashes?

---

5. Git Reset, Revert, and Clean

21. What is the difference between `git reset` and `git revert`?

22. What are the differences between `--soft`, `--mixed`, and `--hard` options in `git reset`?
23. How do you remove untracked files from your working directory?

---

6. Git Cherry-Pick & Interactive Rebase

24. What is `git cherry-pick`, and when should you use it?
25. How do you cherry-pick a specific commit from another branch?
26. What is an interactive rebase, and how is it different from a regular rebase?
27. How do you squash multiple commits into one using rebase?

---

7. Git Conflict Resolution

28. What causes a merge conflict in Git?
29. How do you resolve a merge conflict manually?
30. How do you abort a merge?

---

8. Git Tags & Releases

31. What is a Git tag, and how is it different from a branch?
32. How do you create an annotated tag?
33. How do you push tags to a remote repository?
34. How do you delete a tag locally and remotely?

---

9. Git Remote & Collaboration

35. What is the difference between `git pull` and `git fetch`?
36. How do you clone a remote repository?
37. What is a pull request, and how does it work?
38. How do you fork a repository and contribute to an open-source project?
39. What are Git hooks?

---

10. Miscellaneous

40. How do you undo a commit that has already been pushed?
41. What is the purpose of the `.gitignore` file?

42. How do you check which files are being ignored by Git?
43. How do you track a file that was previously ignored?
44. What is the difference between `origin` and `upstream` in Git?