VISVESVARAYA TECHNOLOGICAL UNIVERSITY

JnanaSangama, Belagavi - 590 018, Karnataka

Acharya Institute of Technology

Acharya Dr Sarvepalli. Radhakrishnan Road
Acharya P.O Soladevanahalli, Bengaluru-560107

ACHARYA

LABORATORY MANUAL

Project Management GIT

BCS358C

III Semester

AY: 2025-26

Name :_____

USN :_____

Branch & Section :_____

Prepared By:

1.Mrs. Shrutika C Rampure
Assistant Professor, Dept. of CS&E

2.Mr. Akash Raj
Assistant Professor, Dept. of CS&E

3.Mr. Abhishek B M
Assistant Professor, Dept. of CS&E

Reviewed By:

Mr. Rajeev Bilagi
Assistant Professor, Dept. of CS&E

Approved By

Dr. Kala Venugopal
H.O.D, Dept. of CS&E

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

(Accredited by NBA & NAAC)

	TABLE OF CONTENTS	
Sl. No	**Content**	**Page Number**

1. Vision, Mission and Motto of the Institute:

Institute Vision:

Acharya Institute of Technology, committed to the cause of value-based education in all disciplines, envisions itself as a fountainhead of innovative human enterprise, with inspirational initiatives for Academic Excellence.

Institute Mission:

Acharya Institute of Technology strives to provide excellent academic ambiance to the students for achieving global standards of technical education, foster intellectual and personal development, meaningful research and ethical service to sustainable societal needs.

Motto of the Institute:

Nurturing Aspirations Supporting Growth

2. Vision and Mission of the Department:

Department Vision:

Envisions to be recognized for quality education and research in the field of Computing, leading to creation of competent engineers, who are innovative and adaptable to the changing demands of industry and society.

Department Mission:

- Act as a nurturing ground for young computing aspirants to attain the excellence by imparting quality education and professional ethics.
- Collaborate with industries and provide exposure to latest tools/ technologies.

Create an environment conducive for research and continuous learning.

3. Program Educational Objectives:

PEO-1: Students shall, have a successful career in academia, R&D organizations, IT industry or pursue higher studies in specialized field of Computer Science and Engineering and allied disciplines.

PEO-2: Students shall, be competent, creative and a valued professional in the chosen field

PEO-3: Students shall, engage in life-long learning, professional development and adapt to the working environment quickly

PEO-4: Students shall, become effective collaborators and exhibit high level of professionalism by leading or participating in addressing technical, business, environmental and societal challenges.

4. Program Outcomes:

ENGINEERING GRADUATES WILL BE ABLE TO:

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, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give

and receive clear instructions.

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

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

5. Program Specific Outcomes:

PSO-1: Students shall, apply the knowledge of hardware, system software, algorithms, networking and data bases.

PSO-2: Students shall, design, analyze and develop efficient, secure algorithms using appropriate data structures, databases for processing of data.

PSO-3: Students shall, be Capable of developing stand alone, embedded and web-based solutions having easy to operate interface using Software Engineering practices and contemporary computer programming language

6. Details of Course and Course administrator:

6.1. Course Details:

Course Title	Course Code	Core/Elective	Semester	Academic Year
Project Management with GIT	BCS358C	Ability Enhancement Course	III	2025-26

Contact Hours/week	Lecture	Tutorials	Practical
03	00	-	02

6.2. Course Administrator/Coordinator Details:

DETAILS OF THE FACULTY CO-ORDINATORS/COURSE ADMINISTRATORS				
Sl. No	**Name of the Faculty**	**Designation**	**Department**	**E-Mail-Id & Contact Number**
1.	**Mrs. Shrutika C Rampure**	**Assistant Professor**	Computer Science and Engineering	EMAIL-ID: shrutikarampure@acharya.ac.in CONTACT NUMBER: 8904603211
2.	**Mr. Akash Raj**	**Assistant Professor**	Computer Science and Engineering	EMAIL-ID: akash@acharya.ac.in CONTACT NUMBER: 7019318040
3.	**Mr. Abhishek B M**	**Assistant Professor**	Computer Science and Engineering	EMAIL-ID: abhishek3088@acharya.ac.in CONTACT NUMBER: 9741536310

6.3. Course Related Specific details:

List Of Prerequisites:	
1.	Basic Linux Commands

Recommended Text Books:	
1.	Version Control with Git, 3rd Edition, by Prem Kumar Ponuthorai, Jon Loeliger Released October 2022, Publisher(s): O'Reilly Media, Inc.

Recommended Reference Books:	
1.	Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, https://gitscm.com/book/en/v2

Web References:		
Sl. No	Web Reference URL	Topic Referred to
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_share	-

6.4. Course Objectives:

Course Objectives:	
The course will enable the students to:	
1	To be familiar with basic command of Git
2	To create and manage branches
3	To understand how to collaborate and work with Remote Repositories
4	To be familiar with version controlling commands

6.5. Course Outcomes:

Course Outcomes:		
After the completion of the course, Students will be able to:		
CO#	COURSE OUTCOME STATEMENT	RBT
1.	Demonstrate the commands related to GIT repository, managing branches, collaboration and remote repository, GIT tags, release, history, and advanced GIT operations to manage a project	3

6.6. CO-PO-PSO Mapping:

COs	Program Outcomes												Program Specific Outcomes		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO-1	3	1			3				2	2	2		2		2

6.7. List of the Programs/Experiments:

Table for listing programs/Experiments for NON-IPCC subject labs

SL	Name of Program
1	**Setting Up and Basic Commands** Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.
2	**Creating and Managing Branches** Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."
3	**Creating and Managing Branches** Write the commands to stash your changes, switch branches, and then apply the stashed changes.
4	**Collaboration and Remote Repositories** Clone a remote Git repository to your local machine.
5	**Collaboration and Remote Repositories** Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.
6	**Collaboration and Remote Repositories** Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge.
7	**Git Tags and Releases** Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.
8	**Advanced Git Operations** Write the command to cherry-pick a range of commits from "source-branch" to the current branch.
9	**Analysing and Changing Git History** Given a commit ID, how would you use Git to view the details of that specific

	commit, including the author, date, and commit message?
10	**Analysing and Changing Git History** Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31."
11	**Analysing and Changing Git History** Write the command to display the last five commits in the repository's history.
12	**Analysing and Changing Git History** Write the command to undo the changes introduced by the commit with the ID "abc123".

6.8. Assessment and Rubrics:

Evaluation Rubrics for lab Programs (Max marks 30)

a. Lab write-up and execution rubrics (Max marks 20)

Continuous Internal Evaluation Rubrics Evaluation out of 30 marks		
Component	**Max Marks**	**Evaluation Criteria**
Lab Record Write-Up	10	Based on the quality, clarity, and completeness of the written procedure and code in the record. **Low(0 to 3) , medium(4 to 6), high (7 to 10)**
Execution of Program	10	Based on whether the student can execute the program independently and successfully.
Viva-Voce	10	Based on understanding of logic, output, and related concepts.
Total for Individual Experiment	**30**	Absent students will be marked zero

Rubric for Program Execution (10 Marks)		
Execution Type	**Marks**	**Remarks**
Full and Independent Execution	10	Student executes the code independently with correct output
Partial Execution (Guided)	6 to 9	Student needs some help to complete the execution
Dependent or Unable to Execute	0 to 5	Student relies heavily on peers/instructor or fails to execute

Marks for Viva (10 Marks)		
Knowledge Level	**Marks**	**Criteria**
High	8 – 10	Answers all questions correctly with strong understanding
Medium	4 – 7	Answers basic to moderate questions with minor gaps in understanding
Low / Poor	0 – 3	Fails to answer or shows poor understanding

<table>
<tr><th colspan="3">CIE : 50 Marks</th><td rowspan="4">IA for 100 Marks Scaled down to 20 Marks

+

30 Marks for CIE</td><td rowspan="4">SEE : 50 Marks</td></tr>
<tr><td colspan="3">Practical Component of the IPCC: 50 Marks
Continuous Internal Evaluation: 30 Marks
Lab Internal Assessment : 20 Marks</td></tr>
<tr><td>Internal Assessments - 1</td><td>20</td><td rowspan="2">Lab Internal Assessment</td></tr>
<tr><td>Continuous Internal Evaluation</td><td>30</td></tr>
</table>

IA for 100 Marks Scaled down to 20 Marks		Conduction	Scale Down
Component	**Marks**		
Procedure	20	**100**	**20**
Execution	60		
Viva-Voce	20		
Total	**100**		

7. Do's and Don'ts in the laboratory:

Do's:

Sl. No	Do's statement
1	Maintain silence in the laboratory.
2	Bring observation book and lab record book and get them signed every time.
3	Fill the login details in the register provided.
4	Leave the bags and other belongings outside the lab, in designated places.
5	Know the location of the fire extinguisher and the first aid box and how to use them in case of an emergency.
6	Report any broken plugs or exposed electrical wires to your faculty/instructor immediately.
7	Shutdown the PC and arrange the chairs properly while leaving the laboratory.

Don'ts:

Sl. No	Do's statement
1	Do not eat or drink in the laboratory.
2	Avoid stepping on electrical wires or any other computer cables.
3	Do not connect pen drive or any other external storage devices to the computer.
4	Do not open the system unit casing or monitor casing, particularly when the power is turned on. Some internal components hold electric voltages of up to 30000 volts, which can be fatal.
5	Do not install any software without your faculty/instructor's permission.
6	Do not touch, connect or disconnect any plug or cable without your faculty/instructor's permission.
7	Usage of mobile phone inside the laboratory is prohibited and if found, phone shall be confiscated.

Git Basic

What is GIT?

Git is a distributed version control system (VCS) that is widely used for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control in the software development industry.

Git allows multiple developers to collaborate on a project by providing a history of changes, facilitating the tracking of who made what changes and when. Here are some key concepts and features of Git:

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

7. Cloning: Cloning is the process of creating a copy of a remote repository on your local machine. This allows you to work on the project and make changes locally.
8. Forking: Forking is a way to create your copy of a repository, typically on a hosting platform like GitHub. You can make changes to your fork without affecting the original project and later create pull requests to contribute your changes back to the original repository.

Git is known for its efficiency, flexibility, and ability to handle both small and large-scale software projects. It is used not only for software development but also for managing and tracking changes in various types of text-based files, including documentation and

configuration files. Learning Git is essential for modern software development and collaboration.

Why we need GIT?

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

1. **Version Control:** Git allows you to track changes in your project's files over time. It provides a complete history of all changes, making it easy to understand what was done, when it was done, and who made the changes. This is invaluable for debugging, auditing, and collaboration.
2. **Collaboration:** Git enables multiple developers to work on the same project simultaneously without interfering with each other's work. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.
3. **Branching:** Git supports branching, which allows developers to create isolated environments for developing new features or fixing bugs. This is essential for managing complex software projects and experimenting with new ideas without affecting the main codebase.
4. **Distributed Development:** Git is a distributed version control system, meaning that every developer has a complete copy of the project's history on their local machine. This provides redundancy, facilitates offline work, and reduces the reliance on a central server.
5. **Backup and Recovery:** With Git, your project's history is distributed across multiple locations, including local and remote repositories. This provides redundancy and makes it easy to recover from accidental data loss or system failures.
6. **Code Review:** Git-based platforms like GitHub, GitLab, and Bitbucket provide tools for code review and collaboration. Developers can propose changes, comment on code, and discuss improvements, making it easier to maintain code quality.
7. **Open Source and Community Development:** Git has become the standard for open-source software development. It allows anyone to fork a project, make contributions,

and create pull requests, which makes it easy for communities of developers to collaborate on a single codebase.

8. **Efficiency:** Git is designed to be fast and efficient. It only stores the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.
9. **History and Documentation:** Git's commit history and commit messages serve as a form of documentation. It's easier to understand the context and reasoning behind a change by looking at the commit history and associated messages.

10. **Customizability:** Git is highly configurable and extensible. You can set up hooks and scripts to automate workflows, enforce coding standards, and integrate with various tools.

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

What is version control System?

A Version Control System (VCS), also commonly referred to as a Source Code Management (SCM) system, is a software tool or system that helps manage and track changes to files and directories over time. The primary purpose of a VCS is to keep a historical record of all changes made to a set of files, allowing multiple people to collaborate on a project while maintaining the integrity of the codebase. There are two main types of VCS: centralized and distributed.

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

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

Key features and benefits of Version Control Systems include:

1. **History Tracking:** VCS systems maintain a complete history of changes, including who made the change, what was changed, and when it was changed. This makes it easy to review and understand the evolution of a project.

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

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

2. GIT life cycle

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

1. **Initializing a Repository**:
   - To start using Git, you typically initialize a new repository (or repo) in your project directory. This is done with the command git init.
2. **Working Directory**:
   - Your project files exist in the working directory. These are the files you are actively working on.
3. **Staging**:
   - Before you commit changes, you need to stage them. Staging allows you to select which changes you want to include in the next commit. You use the git add command to stage changes selectively or all at once with git add ..
4. **Committing**:
   - After you've staged your changes, you commit them with a message explaining what you've done. Commits create snapshots of your project at that point in time. You use the git commit command to make commits, like git commit -m "Add new feature".
5. **Local Repository**:
   - Commits are stored in your local repository. Your project's version history is preserved there.
6. **Branching**:
   - Git encourages branching for development. You can create branches to work on new features, bug fixes, or experiments without affecting the main codebase. Use the git branch and git checkout commands for branching.
7. **Merging:**
   - After you've completed work in a branch and want to integrate it into the main codebase, you perform a merge. Merging combines the changes from one branch into another. Use the git merge command.
8. **Remote Repository**:

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

9. **Pushing**:
   - To share your local commits with a remote repository, you push them using the git push command. This updates the remote repository with your changes.
10. **Pulling**:
    - To get changes made by others in the remote repository, you pull them to your local repository with the git pull command. This ensures that your local copy is up to date.
11. **Conflict Resolution**:
    - Conflicts can occur when multiple people make changes to the same part of a file. Git will inform you of conflicts, and you must resolve them by editing the affected files manually.
12. **Collaboration**:
    - Developers can collaborate by pushing, pulling, and making pull requests in a shared remote repository. Collaboration tools like pull requests are commonly used on platforms like GitHub and GitLab.
13. **Tagging and Releases**:
    - You can create tags to mark specific points in the project's history, such as version releases. Tags are useful for identifying significant milestones.
14. **Continuous Cycle**:
    - The Git lifecycle continues as you repeat these steps over time to manage the ongoing development and evolution of your project. This cycle supports collaborative and agile software development.

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

Git Installation

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

1. Installing Git on Windows:

a. Using Git for Windows (Git Bash):

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

b. Using GitHub Desktop (Optional):

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

2. Installing Git from Source (Advanced):

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

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

```
$ git --version
```

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

How to Configure the Git?

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

System Configuration:

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

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

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

Global Configuration:

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

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

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

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

You can also view your global Git configuration by using:

$ git config --global – list

4. Git command 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) **git blame <file>**: Shows who made each change to a file and when.
20. **git rm <file>**: Removes a file from both your working directory and the Git repository.
21. **git mv <oldfile> <newfile>**: Renames a file and stages the change

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

Experiment 1:

Setting Up and Basic Commands: Initialize a new Git repository in a directory. Create a new file and add it to the staging areaand 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
```

3. 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.
4. 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.

5. 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.

Additional commands:

1. Modify a File and Re-commit

Edit the file and commit again.

Use git diff to see what changed before committing.

2. Stage and Unstage Files

Add a file, then unstage it using git reset <file>.

Notice the difference between *working directory* and *staging area*.

3. Add Multiple Files

Create two or more files, stage them together (git add .), and commit.

Compare this with staging one file at a time.

4. Remove or Rename Files

Use git rm to remove a tracked file and commit.

Use git mv to rename a file and commit.

Experiment 2.

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

Solution:

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

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

$ git checkout master

2. Create a new branch named "feature-branch" and switch to it:

$ git checkout -b feature-branch

This command will create a new branch called "feature-branch" and switch to it.

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

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

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

5. Switch back to the "master" branch:

```
$ git checkout master
```

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

```
$ git merge feature-branch
```

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

Additional commands

1. **List and Verify Branches**

Use git branch to list all branches.
Highlight the active branch with *.

2. **Switch Branches with Checkout/Switch**
3. Use both:
   git checkout feature-branch
   git switch feature-branch

Compare the two commands.

4. **Make Commits in Branch**

Add/edit a file in feature-branch and commit.
Switch back to master and notice the file difference.

Experiment 3.

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

Solution:

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

1. Stash your changes:

$ git stash save "Your stash message"

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

2. Switch to the desired branch:

$ git checkout target-branch

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

3. Apply the stashed changes:

$ git stash apply

This command will apply the most recent stash to your current working branch. If you have multiple stashes, you can specify a stash by name or reference (e.g., git stash apply stash@{2}) if needed.

If you want to remove the stash after applying it, you can use git stash pop instead of git stash apply.

Remember to replace "Your stash message" and "target-branch" with the actual message you want for your stash and the name of the branch you want to switch to.

Additional commands

1. View the Stash List

git stash list

2. Save Stash with a Message

git stash save "Work in progress on feature-branch"

Experiment 4.

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

Solution:

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

1. Open your terminal or command prompt.
2. Navigate to the directory where you want to clone the remote Git repository. You can use the cd command to change your working directory.
3. Use the git clone command to clone the remote repository. Replace <repository_url> with the URL of the remote Git repository you want to clone. For example, if you were cloning a repository from GitHub, the URL might look like this:

$ git clone <repository_url>

Here's a full example:

$ git clone https://github.com/username/repo-name.git

Replace https://github.com/username/repo-name.git with the actual URL of the repository you want to clone.

4. Git will clone the repository to your local machine. Once the process is complete, you will have a local copy of the remote repository in your chosen directory.

You can now work with the cloned repository on your local machine, make changes, and push those changes back to the remote repository as needed

Additional commands

1. **Check Remote Information**

git remote –v

See the remote repositories linked (e.g., origin).

2. **Fetch Latest Changes**

git fetch origin- Downloads updates from the remote without merging them.

3. **Pull Updates (Fetch + Merge)**

git pull origin main

Updates your local branch with remote changes

Experiment 5.

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

Solution:

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

1. Open your terminal or command prompt.
2. Make sure you are in the local branch that you want to rebase. You can switch to the branch using the following command, replacing <branch-name> with your actual branch name:

$ git checkout <branch-name>

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

$ git fetch origin

Here, origin is the default name for the remote repository. If you have multiple remotes, replace origin with the name of the specific remote you want to fetch from.

4. Once you have fetched the latest changes, rebase your local branch onto the updated remote branch:

$ git rebase origin/<branch-name>

Replace <branch-name> with the name of the remote branch you want to rebase onto. This command will reapply your local commits on top of the latest changes from the remote branch, effectively incorporating the remote changes into your branch history.

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

$ git rebase –continue

6. After resolving any conflicts and completing the rebase, you have successfully updated your local branch with the latest changes from the remote branch.
7. If you want to push your rebased changes to the remote repository, use the git push command. However, be cautious when pushing to a shared remote branch, as it can potentially overwrite other developers' changes:

$ git push origin <branch-name>

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

Additional commands

1. **Check Remote Tracking Before Rebase**

git remote –v
git status
git branch –vv
See which remote branch the local branch is tracking.

2. **View Fetched Commits Without Merging**

git log HEAD..origin/main –oneline
Shows new commits available remotely.

3. **Do an Interactive Rebase**

git rebase -i origin/main

4. **Handle Rebase Conflicts**

Intentionally edit the same line locally and remotely.
During rebase:
git status
git add <file>
git rebase –continue

5. **Abort a Rebase**

git rebase –abort
Returns to the pre-rebase state.

6. **Skip a Commit During Rebase**

git rebase –skip
Useful when one conflicting commit is unnecessary.

Experiment 6.

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

Solution:

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

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

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

Additional commands

1. **Fast-Forward vs Non-Fast-Forward Merge**

Create a merge where Git just fast-forwards (no merge commit).
Then, force a merge commit even if fast-forward is possible:

```
git merge --no-ff feature-branch -m "Merged feature with --no-ff"
```

2. **Abort a Merge**

```
git merge feature-branch
# if conflict occurs
git merge –abort
```

Teaches students how to recover.

3. **Resolve Merge Conflicts**

Edit the same line in both branches, then merge.
Fix conflicts manually:

```
git status
git add <file>
git commit
```

Experiment 7.

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

Solution:

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

```
$ git tag v1.0
```

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

```
$ git tag v1.0 <commit-SHA>
```

Replace <commit-SHA> with the actual SHA-1 hash of the commit you want to tag.

Additional commands

1. **List All Tags**

```
git tag
```

Verify existing tags.

2. **Tag a Specific Commit (Not HEAD)**

```
git tag v1.0 <commit-hash>
```

Useful for tagging older commits.

3. **Create an Annotated Tag (with message & metadata)**

```
git tag -a v1.1 -m "Release version 1.1 with bug fixes"
```

Stores author info, date, and message.

4. **View Tag Details**

```
git show v1.1
```

Displays commit and tag info.

5. **Push Tags to Remote**

```
git push origin v1.0
git push origin --tags    # push all tags
```

Ensures teammates get the same tags.

6. **Delete a Tag**

```
git tag -d v1.0          # delete locally
git push origin :refs/tags/v1.0   # delete on remote
```

7. **To** list all tags on the remote repository **(origin) without fetching them locally.**

```
git ls-remote --tags origin
```

Experiment 8.

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

Solution:

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

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

Replace <start-commit> with the commit at the beginning of the range, and <end-commit> with the commit at the end of the range. The ^ symbol is used to exclude the <start-commit> itself and include all commits after it up to and including <end-commit>. This will apply the changes from the specified range of commits to your current branch.

For example, if you want to cherry-pick a range of commits from "source-branch" starting from commit ABC123 and ending at commit DEF456, you would use:

$ git cherry-pick ABC123^..DEF456

Make sure you are on the branch where you want to apply these changes before running the cherry-pick command.

Additional commands

1. **Cherry-Pick a Single Commit**

git cherry-pick <commit-hash>

Good to understand the difference between single vs range cherry-pick.

2. **Cherry-Pick Multiple Non-Sequential Commits**

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

Useful when only specific commits are needed.

3. **Handle Conflicts During Cherry-Pick**

If conflicts occur:

git status

resolve conflicts manually

git add <file>

git cherry-pick –continue

Teaches conflict resolution during cherry-pick.

4. **Abort a Cherry-Pick**

git cherry-pick –abort

Reverts the branch to the state before cherry-pick started.

Experiment 9:

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

Solution:

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

1. Using git show: bash

git show <commit-ID>

Replace <commit-ID> with the actual commit ID you want to view. This command will display detailed information about the specified commit, including the commit message, author, date, and the changes introduced by that commit.

For example:

$ git show abc123

2. Using git log:

$ git log -n 1 <commit-ID>

The -n 1 option tells Git to show only one commit. Replace <commit-ID> with the actual commit ID. This command will display a condensed view of the specified commit, including its commit message, author, date, and commit ID.

For example:

$ git log -n 1 abc123

Both of these commands will provide you with the necessary information about the specific commit you're interested in.

Additional commands

1. View Commit Logs in Different Formats

git log –oneline
git log –stat
git log --pretty=fuller
Shows summaries, stats, or detailed author/committer info

2. **View Commit Changes Only for a Specific File**

git show <commit-id> -- <filename>

Helps focus on a file's history.

3. **Compare Commits**

git diff <commit-id1> <commit-id2>

Shows differences between two commits.

4. **Browse Commit History Graphically**

git log --oneline --graph –all

Visualizes branches and merges.

Experiment 10.

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

Solution:

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

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

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

Additional commands

1. **Format the Log Output**

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

git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31" --pretty=format:"%h - %an, %ar : %s"

Shows commit in concise or custom format (%h=hash, %an=author, %ar=relative date, %s=message).

2. **View Commit Statistics**

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

Displays files changed, insertions, and deletions for each commit.

3. **Count Commits by Author**

git shortlog -s -n --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"

Shows total number of commits by the author.

4. **Search Commits by Author + Keyword**

git log --author="JohnDoe" --grep="bug fix" --since="2023-01-01" --until="2023-12-31"

Filters commits containing a specific keyword in the message.

Experiment 11.

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

Solution:

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

```
$ git log -n 5
```

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

Additional commands

1. **Show Commit Summary Only**

```
git log -5 --pretty=oneline
```

Shows each commit in one line for a quick overview.

2. **Display Commits with Stats**

```
git log -5 –stat
```

Shows files changed, insertions, and deletions in each commit.

3. **Custom Format Output**

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

Displays short hash, author, relative date, and commit message.

4. **View Changes in Each Commit**

```
git log -5 –p
```

Shows the diff for each of the last five commits.

5. **Filter Last Five Commits by Author**

```
git log -5 --author="JohnDoe"
```

Shows the most recent five commits by a specific author

Experiment 12.

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

Solution:

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

```
$ git revert abc123
```

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

Additional commands

1. **evert Multiple Commits**

git revert abc123 def456
Undo multiple specific commits by creating new commits.

2. **Revert a Range of Commits**

git revert abc123^..abc789
Undo all commits in the specified range sequentially.

3. **Undo Last Commit (Keep Changes in Working Directory)**

git reset --soft HEAD~1
Moves HEAD back one commit but keeps changes staged.

4. **Undo Last Commit (Discard Changes)**

git reset --hard HEAD~1
Deletes the last commit and discards changes permanently.

5. **Undo Changes in a Specific File**

git checkout HEAD~1 -- <filename>
Reverts a file to the state in a previous commit without affecting others.