ACHARYA INSTITUTE OF TECHNOLOGY

Acharya Dr. Sarvepalli Radhakrishnan Road, Soladevanahalli,

Bengaluru, Karnataka 560107

(*Affiliated to VTU, Jnana Sangama, Belagavi***)**

Department of Computer Science and Engineering (Data Science)

PROJECT MANAGEMENT WITH GIT

B.E - III Semester, Computer Science and Engineering (Data Science)

[As per Choice Based Credit System (CBCS) scheme]

Subject Code - BCS358C

Lab Manual – 2024-25

Name : ____

USN:____

Branch:__________________________**Section:**________________

ACHARYA INSTITUTE OF TECHNOLOGY

Acharya Dr. Sarvepalli Radhakrishnan Road, Soladevanahalli,

Bengaluru, Karnataka 560107

(*Affiliated to VTU, Jnana Sangama, Belagavi***)**

Department of Computer Science and Engineering (Data Science)

PROJECT MANAGEMENT WITH GIT

B.E - III Semester, Computer Science and Engineering (Data Science)

[As per Choice Based Credit System (CBCS) scheme]

Subject Code - BCS358C

Prepared By:
Mohammad Tahir Mirji
Assistant Professor, Department of Computer Science and Engineering (Data Science),
AIT, Acharya, Bangalore -107

Reviewed By:
Dr. Nagendra J.
Associate Professor, Department of Computer Science and Engineering (Data Science),
AIT, Acharya, Bangalore -107

Approved By:
Dr. Vijayshekhar S. S.
Head of Department of AIML & Computer Science and Engineering (Data Science),
AIT, Acharya, Bangalore -107

ACHARYA INSTITUTE OF TECHNOLOGY MOTTO

"Nurturing Aspirations Supporting Growth"

VISION

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

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.”

Department of Computer Science and Engineering (Data Science)

VISION

To be recognized as the leader in the field of Artificial Intelligence and Machine Learning by nurturing and producing quality next-generation academicians and researchers with human values, who are creative, innovative, and versatile in this fast-growing field.

MISSION

The Department of Artificial Intelligence and Machine Learning (AI and ML) @ Acharya Institute of Technology’s mission is to produce quality students with a sound understanding of the fundamentals of the theory and practice of Artificial Intelligence and Machine Learning. The mission is also to enable students to be leaders in the industry and academia nationally and internationally. Finally, the mission is to meet the strong demands of the nation in the areas of Artificial Intelligence and Machine Learning.

PROGRAM EDUCATIONAL OBJECTIVES (PEOs)

PEO1: Students shall have a successful professional career in industry, academia, R & D organization or entrepreneur in specialized fields of Computer Science and Engineering (Data Science) and allied disciplines.

P**EO2:** Students shall be competent, creative and valued professionals in the chosen field.

PEO3: Engage in life-long learning and professional development.

PEO4: Become effective global collaborators, leading or participating to address technical, business, environmental and societal challenges.

PROGRAM OUTCOMES (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

PO2: 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.

PO3: 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.

PO4: 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.

PO5: 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.

PO6: 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.

PO7: 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.

PO8: Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

O9: Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

PO10: 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.

PO11: 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.

PO12: 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.

Table of Contents

A. COURSE DETAILS & COURSE COORDINATOR DETAILS:

i) Course Details:

Course Title	Course Code	Core/Elective	Semester	Academic Year
PROJECT MANAGEMENT WITH GIT	BCS358C	AEC	III A(AIML)	2024-25

Contact Hours/week	Lecture	Tutorials	Practical
2	0	-	2

ii) Course Administrator/Coordinator Details:

DETAILS OF THE FACULTY CO-ORDINATORS/COURSE ADMINISTRATORS				
S. N	**NAME OF THE FACULTY**	**DESIGNATION**	**DEPARTMENT**	**E-MAIL-ID & CONTACT NUMBER**
1.	Mr. Mohammad Tahir Mirji	Assistant Professor	**AI&ML**	EMAIL-ID: tahir2968@acharya.ac.in CONTACT NUMBER: 7795368246

iii) Course Related Specific details:

List Of Prerequisites:	
1.	Basics of computer operations, Files, Folders structure

viii) Do's and Don'ts in the lab

DO'S

1. Please leave footwear outside the laboratory at the designated place.
2. Please keep your belongings such as bags in the designated place.
3. Turn off the respective systems and arrange the chairs before you
4. leaving the laboratory.
5. Maintain Silence and Discipline inside the laboratory.
6. Wear your ID card & Carry observation and record while coming to
7. the laboratory.

DON'TS

1. Don't use mobile cell phones during lab hours.
2. Do not eat food, chew gum in the laboratory.
3. Do not install, uninstall or alter any software on the computer.

4. Students are not allowed to work in a laboratory alone or without the
5. presence of faculty or instructor.
6. Do not move any equipment from its original position.
7. We are not responsible for any belongings left behind.
8. Do not enter the laboratory without permission.

Course Outcomes:		**RBT Level**
After the completion of the course, Students will be able to:		
CO1	● Use the basics commands related to git repository	L1
CO2	● Create and manage the branches	L4
CO3	● Apply commands related to Collaboration and Remote Repositories	L3
CO4	● Use the commands related to Git Tags, Releases and advanced git operations	L1
CO5	● Analyze and change the git history	L4

v. CO-PO-PSO Mapping:
Course Outcomes - Program Outcomes – Program Specific Outcomes 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	-	2	-	3	-	-	-	-	-	-	-	-	-	3
CO-2	3	-	2	-	3	-	-	-	-	-	-	-	-	-	3
CO-3	3	-	2	-	3	-	-	-	2	-	-	-	-	-	3
CO-4	3	-	2	-	3	-	-	-	-	-	-	2	-	-	3
CO-5	3	-	2	-	3	-	-	-	-	-	-	-	-	-	3

vi. Course Outcomes/Program Outcomes assessment methods:
Course Assessment Procedure:

Procedure for Internal Assessment	:	2-IA-Tests + Records+Attendance

Maximum Marks for Internal Assessment : 50-Marks

Maximum Marks for Final Exam : 50-Marks

Assessment Tools			**Weightage**	**Frequency**	**Responsibility**
Direct Assessment	Continuous Internal Evaluation (CIE)	Lab Internal Assessment	50%	Twice in a semester	Department level
	Semester End Exam (SEE)		50%	Once in a semester	Department level

Course Outcome Attainment Computation

Marks scored by students	1% to 55%	56 % to 80 %	81% to 100%
Weightage	1	2	3

Target attainment = 75%

B. <u>COURSE PLAN</u>:

<u>Course plan/Lesson Plan</u>

Hr. No.	Experiment	Content
1	**1**	Zero Session (Context Setting)
2	2	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.
3	3	Creating and Managing Branches Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."
4	4	Creating and Managing Branches Write the commands to stash your changes, switch branches, and then apply the stashed changes.
5	5	Collaboration and Remote Repositories Clone a remote Git repository to your local machine.
6	6	Collaboration and Remote Repositories Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.
7	7	Collaboration and Remote Repositories Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge
8	8	Git Tags and Releases Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

9	9	Advanced Git Operations Write the command to cherry-pick a range of commits from "source-branch" to the current branch.
10	10	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?
9	9	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."
10	10	Analysing and Changing Git History Write the command to display the last five commits in the repository's history.
11	11	Analysing and Changing Git History Write the command to undo the changes introduced by the commit with the ID "abc123"
12	12	Manage 1 program with 4 different function written by your project mates and manage to collaborate amongst each other via GIT and GITHUB
13	13	Comprehensive Quiz with 1st, 2nd, 3rd Prices.

NOTES:

Introduction to Git & basic keywords

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.

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 rm <file>: Removes a file from both your working directory and the Git repository.

22. git mv <oldfile> <newfile>: Renames a file and stages the change. These are some of the most common Git commands, but Git offers a wide range of features and options for more advanced usage. You can use git --help followed by the command name to get more information about any specific command, e.g., git help commit.

Q1. 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.

1. Initialize a new Git repository:

```
mkdir your_project_directory

cd your_project_directory

git init
```

2. Create a new file:

```
touch your_new_file.txt
```

3. Add the file to the staging area:

```
git add your_new_file.txt
```

If you want to add all new and modified files to the staging area, you can use:

```
git add
```

4. Commit the changes with a message:

```
git commit -m "Initial commit - adding a new file"
```

Q2. Creating and Managing Branches
Create a new branch named "feature-branch." Switch to the "master" branch. Merge the "feature-branch" into "master."

1. Create a new branch named "feature-branch":

```
git branch feature-branch
```

Alternatively, you can create and switch to the new branch in one step:

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

2. Switch to the "master" branch:

```
git checkout master
```

3. Merge "feature-branch" into "master":

```
git merge feature-branch
```

If there are no conflicts, Git will automatically perform a fast-forward merge. If there are conflicts, Git will prompt you to resolve them before completing the merge. If you used `git checkout -b feature branch` to create and switch to the new branch, you can switch back to master and merge in a single command:

```
git checkout master
git merge feature-branch
```

4. **Resolve any conflicts (if needed) and commit the merge:** If there are conflicts, Git will mark the conflicted files. Open each conflicted file, resolve the conflicts, and then:

```
git add
git commit -m "Merge feature-branch into master"
```

Now, the changes from “feature-branch” are merged into the “master” branch. If you no longer need the “feature-branch,” you can delete it:

```
git branch -d feature-branch
```

This assumes that the changes in “feature-branch” do not conflict with changes in the “master” branch. If conflicts arise during the merge, you’ll need to resolve them manually before completing the merge.

Q3. Creating and Managing Branches
Write the commands to stash your changes, switch branches, and then apply the stashed changes.

1. Stash your changes:

```
git stash save "Your stash message"
```

This command will save your local changes in a temporary area, allowing you to switch branches without committing the changes.

2. Switch to another branch:

```
git checkout your-desired-branch
```

a

3. **Apply the stashed changes:**

```
git stash apply
```

If you have multiple stashes and want to apply a specific stash, you can use:

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

After applying the stash, your changes are reapplied to the working directory.

4. **Remove the applied stash (optional):**

If you no longer need the stash after applying it, you can remove it:

```
git stash drop
```

To remove a specific stash:

```
git stash drop stash@{1}
```

If you want to apply and drop in one step, you can use `git stash pop`:

```
git stash pop
```

Now, you've successfully stashed your changes, switched branches, and applied the stashed changes.

Q4. Collaboration and Remote Repositories

Clone a remote Git repository to your local machine.

To clone a remote Git repository to your local machine, you can use the `git clone` command. Here's the general syntax:

```
git clone <repository_url>
```

Replace `<repository_url>` with the actual URL of the Git repository you want to clone. For example:

```
git clone https://github.com/example/repo.git
```

This command will create a new directory with the name of the repository and download all the files from the remote repository into that directory. If the repository is private and requires authentication, you might need to use the SSH URL or provide your credentials during the cloning process.

For SSH:

```
git clone git@github.com:example/repo.git
```

After running the `git clone` command, you'll have a local copy of the remote repository on your machine, and you can start working with the code.

Q5. Collaboration and Remote Repositories
Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

1. **Fetch the latest changes from the remote repository:**

```
git fetch
```

This command fetches the latest changes from the remote repository without automatically merging them into your local branches.

2. Rebase your local branch onto the updated remote branch:

Assuming you are currently on the branch you want to update (replace `your-branch` with the actual name of your branch):

```
git rebase origin/your-branch
```

This command applies your local commits on top of the changes fetched from the remote branch. If conflicts arise, Git will pause the rebase process and ask you to resolve them. Alternatively, you can use the interactive rebase to review and modify commits during the rebase:

```
git rebase -i origin/your-branch
```

This opens an editor where you can pick, squash, or edit individual commits.

3. Continue the rebase or resolve conflicts:

If conflicts occurred during the rebase, Git will prompt you to resolve them. After resolving conflicts, you can continue the rebase with:

```
git rebase --continue
```

If you decide to abort the rebase at any point, you can use:

```
git rebase --abort
```

4. **Push the rebased branch to the remote repository:**

After successfully rebasing your local branch, you may need to force-push the changes to the remote repository:

```
git push origin your-branch --force
```

Be cautious with force-pushing, especially if others are working with the same branch, as it rewrites the commit history. Now, your local branch is rebased onto the updated remote branch. Keep in mind that force-pushing should be done with caution, especially on shared branches, to avoid disrupting collaborative work.

Q6. Collaboration and Remote Repositories
Write the command to merge “feature-branch” into “master” while providing a custom commit message for the merge.

To merge “feature-branch” into “master” and provide a custom commit message, you can use the following command:

```
git merge feature-branch -m "Your custom commit message"
```

Replace “Your custom commit message” with the actual message you want to use for the merge commit. This command performs the merge and creates a new commit on the “master” branch with the specified message. If there are no conflicts, Git will complete the merge automatically.

If conflicts occur during the merge, Git will pause and prompt you to resolve the conflicts manually. After resolving conflicts, you can continue the merge process with:

```
git merge --continue
```

Alternatively, you can use an interactive merge to modify the commit message before finalizing the merge:

```
git merge feature-branch --no-ff -e
```

This opens an editor where you can edit the commit message before completing the merge. Again, replace “feature-branch” with the name of your actual feature branch.

Q7. Git Tags and Releases

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

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

```
git tag v1.0 <commit_hash>
```

Replace **`<commit_hash>`** with the actual hash of the commit for which you want to create the tag.

For example, if you want to tag the latest commit, you can use the following:

```
git tag v1.0 HEAD
```

This creates a lightweight tag pointing to the specified commit. Lightweight tags are simply pointers to specific commits and contain only the commit checksum.

If you want to push the tag to a remote repository, you can use:

```
git push origin v1.0
```

This command pushes the tag named “v1.0” to the remote repository. Keep in mind that Git tags, by default, are not automatically pushed to remotes, so you need to explicitly push them if needed.

Q8. Advanced Git Operations

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, you can use the following command:

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

Replace `<start-commit>` and `<end-commit>` with the commit hashes or references that define the range of commits you want to cherry-pick. The ^ (caret) symbol is used to exclude the starting commit itself from the range.

For example, if you want to cherry-pick the commits from commit A to commit B (excluding A) from "source-branch" to the current branch, you would run:

```
git cherry-pick A^..B
```

After running this command, Git will apply the specified range of commits onto your current branch. If there are any conflicts, Git will pause the cherry-pick process and ask you to resolve them. After resolving conflicts, you can continue the cherry-pick with:

```
git cherry-pick --continue
```

If you encounter issues and need to abort the cherry-pick operation, you can use:

```
git cherry-pick --abort
```

Remember that cherry-picking introduces new commits based on the changes from the source branch, so conflicts may arise, and manual intervention might be required.

Q9. Analyzing 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?

To view the details of a specific commit, including the author, date, and commit message, you can use the following Git command:

```
git show <commit-id>
```

Replace `<commit-id>` with the actual commit hash or commit reference of the commit you want to inspect.

For example:

```
git show abc123
```

This command will display detailed information about the specified commit, including the author, date, commit message, and the changes introduced by that commit.

If you only want a more concise summary of the commit information (without the changes), you can use:

```
git log -n 1 --pretty=format:"%h - %an, %ar : %s" <commit-id>
```

This command displays a one-line summary of the commit, showing the abbreviated commit hash (`%h`), author name (`%an`), relative author date (`%ar`), and commit message (`%s`).

Remember to replace `<commit-id>` with the actual commit hash or reference you want to inspect.

Q10. Analyzing and Changing Git History

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 “2023-01-01” and “2023-12-31,” you can use the following `git log` command with the `--author` and `--since` / `--until` options:

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

This command will display the commit history that meets the specified criteria. Adjust the author name and date range according to your requirements. The `--since` and `--until` options accept a variety of date and time formats, providing flexibility in specifying the date range.

Q11. Analyzing and Changing Git History
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, you can use the following `git log` command with the `-n` option:

```
git log -n 5
```

This command shows the latest five commits in the repository, with the most recent commit displayed at the top. Adjust the number after the `-n` option if you want to see a different number of commits.

If you want a more concise output, you can use the `--oneline` option:

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

This provides a one-line summary for each commit, including the abbreviated commit hash and the commit message.

Q12. Analyzing and Changing Git History

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

To undo the changes introduced by a specific commit with the ID “abc123,” you can use the `git revert` command. The `git revert` command creates a new commit that undoes the changes made in a previous commit. Here’s the command:

```
git revert abc123
```

Replace “abc123” with the actual commit hash or commit reference of the commit you want to undo. After running this command, Git will open a text editor for you to provide a commit message for the new revert commit.

Alternatively, if you want to completely remove a commit and all of its changes from the commit history, you can use the `git reset` command. However, keep in mind that using `git reset` can rewrite history and should be used with caution, especially if the commit has been pushed to a remote repository.

```
git reset --hard abc123
```

Again, replace “abc123” with the actual commit hash or commit reference. After using `git reset --hard`, your working directory will be modified to match the specified commit, discarding all commits made after it. Be cautious when using `--hard` as it is a forceful operation and can lead to data loss.

- **POINTS TO BE REMEMBERED**
  - Repository: A storage location for your project files, including all revisions.
  - Local vs. Remote: Understand the difference between the local repository (on your machine) and the remote repository (on GitHub).
  - Working Directory: The current state of your project, which can have untracked, modified, or staged files.
  - Staging Area: The place to prepare changes before committing.

Git Workflow Best Practices

- Write Clear Commit Messages: Describe changes concisely and meaningfully.
- Frequent Commits: Commit frequently to save progress and facilitate tracking.
- Use Feature Branches: Separate different features into branches for better organization.

Security and Access Control

- SSH Keys: Secure access to GitHub with SSH authentication.
- Collaborator Access: Control who can view and modify your repositories on GitHub.

- **THINK BEYOND CONCEPTS**

1. Automating Workflows with GitHub Actions

Customizable Automation: Use GitHub Actions to automate CI/CD (Continuous Integration/Continuous Deployment) pipelines, testing, and code analysis.

Predefined Workflows: Set up workflows for code linting, deployment, or build checks. It ensures your codebase remains high quality and that tests are run on every push or pull request.

Collaborative Code Review Automation: Automate issue triaging, pull request labeling, or notifying reviewers for code review.

2. Leveraging GitHub for Documentation

GitHub Pages: Host project documentation directly from your repository, creating a static website for project details, API references, or user guides.

README and Wiki: The README file acts as a project's landing page, while GitHub Wiki can provide an organized space for detailed documentation, tutorials, or project FAQs.

3. Contributing to Open Source Projects

Learning from Community Code: By contributing to open-source projects, you gain exposure to best coding practices, varied codebases, and coding standards.

Social Coding: Learn how to handle real-world code reviews, raise issues, and participate in discussions to help solve problems collaboratively.

Network Expansion: Collaboration in open-source projects helps you connect with developers worldwide, fostering learning and professional growth.

4. Advanced Branching Strategies

Git Flow: Utilize Git Flow for structured branching, where main, develop, feature, release, and hotfix branches help manage large-scale projects.

Trunk-Based Development: For continuous deployment environments, trunk-based development (single mainline branch) speeds up release cycles.

Forking Workflow: Ideal for open-source or multi-contributor projects, where each developer works independently in their fork, submitting changes via pull requests.

5. Data-Driven Project Management

Metrics Tracking: Track repository metrics (contributions, pull request count, response times) using GitHub Insights to improve team performance and transparency.

Kanban with Projects: Use GitHub Projects to visualize task progress, setting up Kanban boards for tasks like feature development, issue resolution, or backlog grooming.

Milestones and Labels: Organize and prioritize work using milestones for project phases or key releases and labels to categorize issues or PRs (e.g., “bug,” “enhancement,” “urgent”).

6. Code Quality and Security

Code Scanning and Analysis: Use tools like CodeQL in GitHub to identify security vulnerabilities or code quality issues automatically.

Dependency Management: Utilize Dependabot to automatically update dependencies and manage potential security vulnerabilities.

Commit Message Standards: Adopt standardized formats for commit messages (e.g., Conventional Commits) to make version history cleaner and more informative.

7. Effective Collaboration through Communication

Issue Templates: Use custom issue templates to streamline bug reporting, feature requests, and documentation requests, making collaboration efficient.

Pull Request Templates: Outline the necessary information for every PR to ensure consistency in code reviews and discussions.

Draft Pull Requests: Share work-in-progress code with team members, allowing early feedback and avoiding last-minute fixes.

8. Git Aliases and Scripts for Efficiency

Git Aliases: Set up aliases to save time on frequently used commands (e.g., git co for git checkout, git lg for a simplified log view).

Custom Scripts: Create scripts for multi-step Git operations like branch cleanup, commit automation, or repository setup to streamline tasks and reduce manual errors.

9. History Manipulation for Clean Commit Logs

Interactive Rebase: Clean up commit history by reordering, editing, or squashing commits for a clearer history.

Cherry-Picking: Use cherry-pick to apply specific commits from one branch to another, which is useful for hotfixes or feature migrations.

Commit Amend and Reset: Amend previous commits for small changes, and use reset (carefully) for reverting local commits while maintaining a clean history.

10. Creating Custom GitHub Bots and Integrations

Custom Bots: Write bots to automate interactions, such as greeting first-time contributors, reminding reviewers, or updating statuses.

Third-Party Integrations: Integrate with tools like Slack, JIRA, or Trello to streamline communication, task management, and project tracking.

Webhooks: Use GitHub webhooks to trigger custom events in external applications, allowing for integration with other development or CI/CD systems.

11. Experimenting with Alternative Git-Based Platforms

GitLab/Gitea for CI/CD: Experiment with platforms like GitLab for more integrated CI/CD options, or Gitea for self-hosting options.

Collaboration Across Platforms: Connect repositories from different platforms if collaborating with diverse teams to manage dependencies effectively.

12. Version Control Best Practices for Machine Learning (ML) Models and Data

Data and Model Versioning: Use tools like DVC (Data Version Control) in conjunction with Git for handling large datasets and tracking ML models.

Experiment Tracking: Track experiment parameters, code changes, and dataset versions to ensure reproducibility in model training and evaluation.

● QUESTION BANK

1. GitHub Actions & Automation

- **Q1**: What is GitHub Actions, and how can it improve a project's workflow?
- **Q2**: Describe a use case where GitHub Actions would help automate testing in a collaborative project.
- **Q3**: How would you set up a GitHub Action to deploy a project to a production environment after merging to the main branch?

2. Documentation & GitHub Pages

- **Q4**: What is GitHub Pages, and how can it be used to enhance documentation for a project?
- **Q5**: How can the README file in a GitHub repository impact new contributors?

3. Branching Strategies

- **Q6**: Explain the concept of Git Flow. How does it structure the development workflow?
- **Q7**: What is trunk-based development, and why might it be preferred in a continuous deployment environment?
- **Q8**: Describe how you would handle hotfixes in a Git Flow branching strategy.

4. Project Management and Collaboration

- **Q9**: How can GitHub Projects be utilized to track tasks and organize development work?
- **Q10**: Describe the importance of labels and milestones in a large project with multiple contributors.
- **Q11**: What is the significance of using draft pull requests, and when would it be useful?

5. Security and Code Quality

- **Q12**: What role does CodeQL play in maintaining code quality on GitHub?
- **Q13**: How does Dependabot assist with dependency management, and why is it crucial for security?
- **Q14**: Why is it important to follow a commit message standard, such as Conventional Commits, in a collaborative project?

6. Efficiency Tools in Git

- **Q15**: What is the purpose of using Git aliases, and give an example of a useful alias?
- **Q16**: Explain how the git stash command can be beneficial when switching branches.

7. Managing Commit History

- **Q17**: Describe the purpose of interactive rebase and give an example of when it might be used.
- **Q18**: What is cherry-picking in Git, and how does it differ from a typical merge?

- **Q19**: Explain how you would use git reset to undo a commit locally without affecting the remote repository.

8. GitHub Bots and Integrations

- **Q20**: How can custom GitHub bots help streamline collaboration in large projects?
- **Q21**: Describe how webhooks can be used to trigger external applications from a GitHub repository.
- **Q22**: Name two benefits of integrating GitHub with tools like Slack or JIRA.

9. Data and Model Versioning for ML Projects

- **Q23**: Why is DVC (Data Version Control) important for managing data in Git-based machine learning projects?
- **Q24**: How does experiment tracking improve the reproducibility of machine learning models?
- **Q25**: Explain the challenges of versioning large datasets with Git and describe a solution.

10. Practical Scenarios

- **Q26**: Imagine you need to apply a bug fix in multiple branches without merging. How would you do it?
- **Q27**: Your team wants to track code reviews and contributions. Which GitHub features would you use, and why?
- **Q28**: If a merge conflict occurs, how would you go about resolving it effectively? Provide steps.

Comprehensive Quiz with 1st,2nd 3rd Prize distribution

Details:

Number of Questions: 50 Questions
Duration: 1 Hour Time
Venue: Class / Alive platform

Prizes: 🎉

1st - Student profile addition to department magazine with Certificates from the Department as Expert in Excel.
2nd - Certificates from the Department as Expert in Excel.
3rd - Certificates from the Department as Beginners in Excel.

Recommended Reference Books:	
1.	Version Control with Git, 3rd Edition, by Prem Kumar Ponuthorai, Jon Loeliger Released October 2022,Publisher(s): O'Reilly Media, Inc.
2.	Pro Git book, written by Scott Chacon and Ben Straub and published by Apress, https://git-scm.com/book/en/v2

Softwares to be used:
Git - https://git-scm.com/
Githiub - https://github.com/

WEB REFERENCES

S. N	**WEB URL**	**TOPIC REFERRED TO**
1.	https://youtu.be/8JJ101D3knE	Practice
2.	https://youtu.be/SWYqp7iY_Tc	Theory