🛠 GIT NOTES 📌

Concept-1

#History of Git--1

--Git was created by Linus Torvalds in 2005 to manage the development of the Linux kernel.

#Before Git:

--The Linux kernel community originally used BitKeeper, a proprietary version control system.

--However, when BitKeeper's free access was revoked, the Linux community needed a new version control system.

#Why Git Was Created:

Linus Torvalds decided to develop a new open-source, distributed version control system with the following key goals:

✅ Fast – Efficiently handling large-scale projects like the Linux kernel.

✅ Distributed – Every developer has a full copy of the repository, enabling offline work.

✅ Secure – Ensuring data integrity and preventing corruption.

#After Git's Release:

Since its release, Git has become the most widely used version control system in the world. It powers platforms like:

 # remote hosting platforms for Git repositories.

GitHub

GitLab

Bitbucket

Tuleap

#What is Git?

Git is a free, open-source version control system (VCS) that enables developers to store, track, and manage code changes in software development projects. Git allows multiple developers to work on the same project simultaneously, without interfering with each other's work.

Or

Git is a DevOps tool used for source code management. It is a free and open-source version control system used to handle small to very large projects efficiently. Git is used to tracking changes in the source code, enabling multiple developers to work together on non-linear development.

#Main Git definition

#Git - Distributed Version Control System (DVCS):

Git is a Distributed Version Control System (DVCS) that allows each developer to have a local copy of the entire project, including the full project history and all versions of files. Developers can work on their own local repositories offline and commit changes locally. Later, they can push or pull updates to/from a central repository (e.g., GitHub, GitLab, Tuleap) to synchronize their work with others.

#Key Points:

--Local Repositories: Every developer has a full copy of the project and its history.

--Work Offline: Developers can make changes, commit, and track history without needing an internet connection.

--Synchronization: Changes can be synchronized with a central repository when needed (via commands like git push and git pull).

--Branching and Merging: Git allows for easy creation of branches to work on different features independently and later merge them back into the main project.

#What is the Meaning of "Version" in Git?

#It seems like you are asking about Version Control Systems (VCS) and Distributed Version Control Systems (DCVS).

A Version Control System (VCS) is a software tool used by developers to keep track of changes made to a project over time. It allows them to navigate through different versions of their code, view changes, and revert back to previous versions when necessary.

#The two main types of version control systems are:

1. #What is a Centralized Version Control System (CVCS)?

--Centralized Version Control System (CVCS): In a CVCS, there is a central server that stores all versions of a project's files and the complete history. Developers get their own working copies, but the historical data is only stored on the central server.

--Developers must connect to the central server to commit changes, fetch updates, or view the project history.

--Single point of failure: If the central server goes down, developers cannot commit changes or access the project history until the server is restored.

#remote machine or a cloud server

Example: SVN (Subversion)

2. #What is a Distributed Version Control System (DVCS)?

--Distributed Version Control System (DVCS): In a DVCS, every contributor has a full local repository that contains the entire project history. Changes are shared between repositories as a set of changes, or "changesets."

--This allows for offline work and greater flexibility, as developers do not need to rely on a central server for most operations.

--There is no single point of failure because each developer has a complete copy of the repository.

#Examples: Git, Mercurial

#Key Concepts of "Version" in Git

- ✅ Version = A snapshot of your project

Every commit represents a unique version of your code.

You can go back to any previous version anytime.

- ✅ Git assigns a unique ID (SHA) to each version

Example: a1b2c3d4 (commit hash)

You can use this ID to checkout or revert changes.

- ✅ Git tracks changes over time

It does not store full copies of files.

Instead, it saves only the differences (deltas) between versions to save space.

- ✅ Version control = Managing different versions of files

Developers can collaborate without overwriting each other's work.

Multiple versions (branches) can exist at the same time.

#Example of Versions in Git

1. Create a project and initialize Git

mkdir my_project && cd my_project

- ➢ git init

2. Create a file and add the first version
- ➢ echo "Hello Git!" > file1.txt
- ➢ git add file1.txt
- ➢ git commit -m "First version of the project"

3. Modify the file and create a new version
   - ➢ echo "Git is powerful!" >> file1.txt
   - ➢ git add file1.txt
   - ➢ git commit -m "Updated file1.txt with more content"
4. View all versions (commit history)
   - ➢ git log --oneline

#Note

✅ Version in Git means a saved state (commit) of your project.

✅ Git allows you to track, switch, and manage different versions efficiently.

✅ It ensures collaboration without losing past work.

#Key Differences Between DVCS and CVCS:

#Comparison:	Centralized VCS	Distributed VCS (DVCS)
Feature	Centralized VCS	Distributed VCS (DVCS)
Repository	Central server only	Local copy for each developer
Offline Work	No (requires a server)	Yes (can work offline)
Data Redundancy	Only on central server	Each developer has a full copy
Speed	Slower (requires server access)	Faster (local operations)
Collaboration	Depends on central server	More flexible collaboration (works locally, then sync)
Example Tools	SVN	Git (used with GitHub, GitLab, Bitbucket,Tuleap)

#Key Features of Git- (What Git Can Do)

Git Features

- Open Source – Anyone can contribute to GitHub projects. Example: The Linux Kernel is managed using Git, with thousands of developers contributing worldwide.
- Security – Git ensures commit integrity. Example: When you run git log --show-signature, you can verify if a commit is signed and trusted.
- Scalability – Large companies use Git for massive projects. Example: Google's Android OS repository is managed with Git, handling millions of lines of code.

- Speed – Local operations make Git fast. Example: Running git checkout -b new-feature creates a new branch instantly without network dependency.

✅ Track Changes Over Time: Git remembers every change you make to your project, so you can always go back and see who changed what and when.

✅ Branching & Merging: Imagine you're working on a new feature but don't want to mess up the main project. You can create a separate "branch" to work on it, and once it's ready, you merge it back into the main project.

✅ Go Back to Previous Versions: Made a mistake? No problem! Git lets you revert to an older version of your project, just like an "undo" button.

✅ Team Collaboration with Pull Requests: If you're working in a team, Git helps you propose changes, review them with your teammates, and only merge them into the main project when everyone is happy. This is done using "Pull Requests" (PRs) on platforms like GitHub and GitLab.

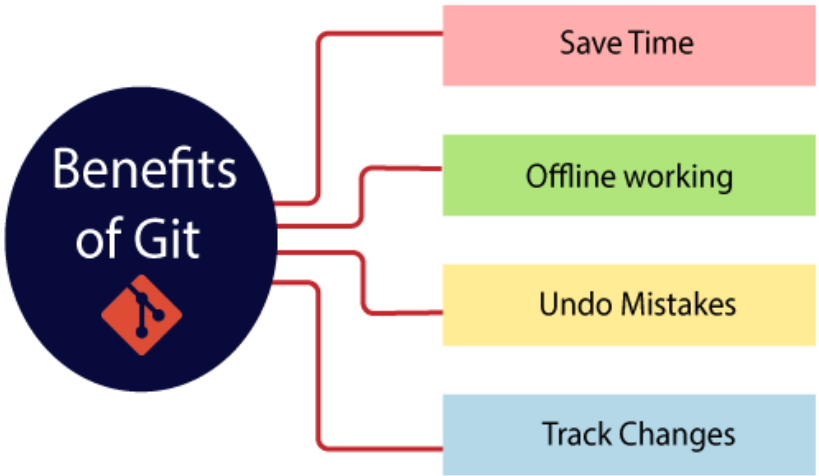
✅ Work Offline, Sync Later: Unlike older version control systems that require a constant internet connection, Git allows you to work on your project offline. When you're ready, you can sync your changes with the team.

✅ Fast & Efficient: Git is designed to handle large projects quickly, so saving changes, switching branches, and merging updates happen almost instantly.

✅ Secure & Reliable: Git ensures your project history is safe and cannot be changed accidentally. It uses strong security methods to track every change made.

✅ Offline Work

#Benefits of Using Git- (Why Git is Useful)



✅ Collaboration – Multiple developers can work on the same project without conflicts.

✅ Faster Development – Branching allows independent feature development and bug fixes.

✅ Offline Work – Developers can work without an internet connection.

✅ Backup & Safety – Every developer has a full project copy, preventing data loss.

✅ Code Reviews – Pull Requests ensure high-quality code before merging.

✅ Version Control – Easy to track, revert, and manage code changes.

✅ Speed & Performance – Fast local operations like committing, branching, and merging.

✅ Free & Open-Source – No cost, widely supported, and highly flexible.

The two main types of version control systems are:

#Distributed Version Control System (DVCS)

1 Every developer has a full local copy of the repository, including project history.

2. Developers can work offline, committing changes locally without needing a connection to a central server.

3. Developers sync their local repositories with a remote repository (or other developers' repositories) when needed (e.g., git push, git pull, or git fetch).

4. Even if the central server fails, developers can continue working locally and share changes directly with each other if necessary.

5. Examples: Git, Mercurial, Bazaar.

#Centralized Version Control System (CVCS)

1. A single central repository stores all project files and history. Developers do not have a full copy of the repository locally.

2. Developers must be online to commit changes or fetch updates from the central server.

3. Developers rely heavily on the central server for most operations, such as committing changes, viewing history, or updating their working copies.

4. If the central server goes down, developers cannot commit changes or fetch updates until the server is restored.

5. Examples: SVN (Subversion), CVS.

#Concept-2

#Installing Git on Ubuntu 🐧

- ✓ Step 1: Update the System
  - sudo apt update
  - sudo apt upgrade -y
- ✓ Step 2: Install Git
  - sudo apt install git -y
- ✓ Step 3: Verify Installation
  - git --version
- ✓ ✅ Example Output:
  - git version 2.34.1
- ✓ Step 4: Configure Git (First-Time Setup)
  - git config --global user.name "Your Name"
  - git config --global user.email "your.email@example.com"
  - git config --global -e
- ✓ Step 6: View Git Configuration
  - git config --list
- ✓ Step 7: Generate SSH Key (Optional)
  - #Add your SSH key to your GitHub or GitLab, Tuleap account.
  - ssh-keygen -t rsa -b 4096 -C "your.email@example.com"

#Installing Git on Red Hat (RHEL/CentOS) 🧢

- ✓ Step 1: Update the System
  - sudo yum update -y
- ✓ Step 2: Install Git
  - sudo yum install git -y
- ✓ Step 3: Verify Installation
  - git --version

#SSH-CONCEPT-2

#How to Add Your SSH Key to GitHub, GitLab, or Tuleap 🚀

Using SSH keys allows secure authentication to Git services like GitHub, GitLab, or Tuleap without using passwords. Follow these steps to set up SSH authentication:

1 Generate an SSH Key

```
ssh-keygen -t rsa -b 4096 -C "your.email@example.com"
-t rsa → Specifies the RSA algorithm
-b 4096 → Generates a 4096-bit key for strong security
-C "your.email@example.com" → Adds an identifier (your email)
```

2. Start the SSH Agent & Add Your Key

```
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_rsa  #Now add your SSH key
```

3. Copy Your Public Key

```
cat ~/.ssh/id_rsa.pub  #Run this command to display the public key
```

4. Add SSH Key to GitHub, GitLab, or Tuleap

Tuleap:

Go to Tuleap → Account Settings → SSH Keys

Paste and save

5. Test the Connection

```
ssh -T git@github.com  # For GitHub
ssh -T git@gitlab.com  # For GitLab
```

6. Configure Git to Use SSH

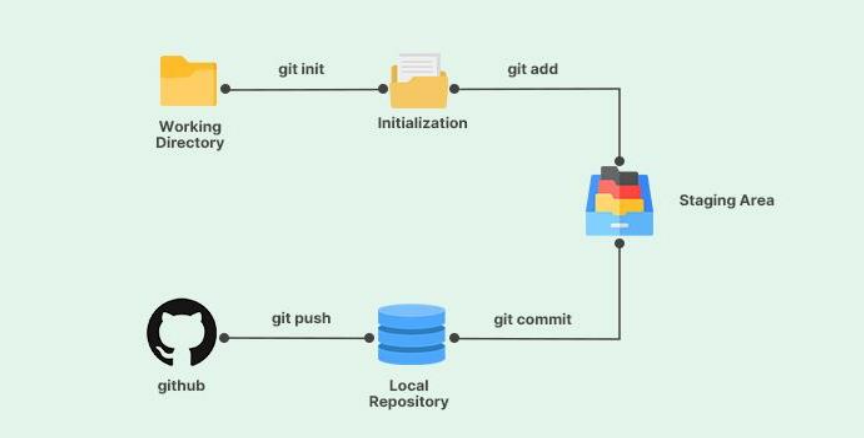
```
git remote set-url origin git@github.com:username/repository.git   #Set Git to use SSH instead of HTTPS
git remote -v
```

#Note

Now, you can push, pull, and clone repositories using SSH without entering a password every time! 🚀

#Concept-3

Basic Git Workflow (With Example) 🚀



The Git workflow follows these main steps:

👉 Working Directory → 👉 Staging Area → 👉 Local Repository → 👉 Remote Repository

Step 1: Initialize a Git Repository

Create a new Git repository inside a folder.

- mkdir my_project # Create a new project folder
- cd my_project # Move into the project folder
- git init # Initialize Git (creates a. git folder)

✅ Example Output:

Initialized empty Git repository in /my_project/.git/

Step 2: Create & Modify Files

- Create a file and add some content.
- echo "Hello, Git!" > file.txt # Create a file with content
- cat file.txt # View file content

Step 3: Add Changes to Staging Area

- Before committing, files need to be added to the staging area.
- git add file.txt # Add a specific file to staging
- git status # Check the current status

✅ Example Output:

Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
new file: file.txt

Step 4: Commit the Changes

Save the staged changes to the local repository.

- git commit -m "Initial commit - added file.txt"

✅ Example Output:

```
[main (root-commit) e4f3a4d] Initial commit - added file.txt
 1 file changed, 1 insertion (+)
 create mode 100644 file.txt
```

Step 5: Connect to a Remote Repository

To collaborate with others, push the project to GitHub/GitLab/Tuleap.

- git remote add origin <repo_url> # Connect local repo to remote
- git branch -M main # Rename branch to 'main' if needed
- git push -u origin main # Push the code to remote repository

Step 6: Update Local Repo with Remote Changes

If teammates make changes, fetch and merge them.

- git pull origin main # Get the latest changes from the remote repo

Step 7: Clone an Existing Repository

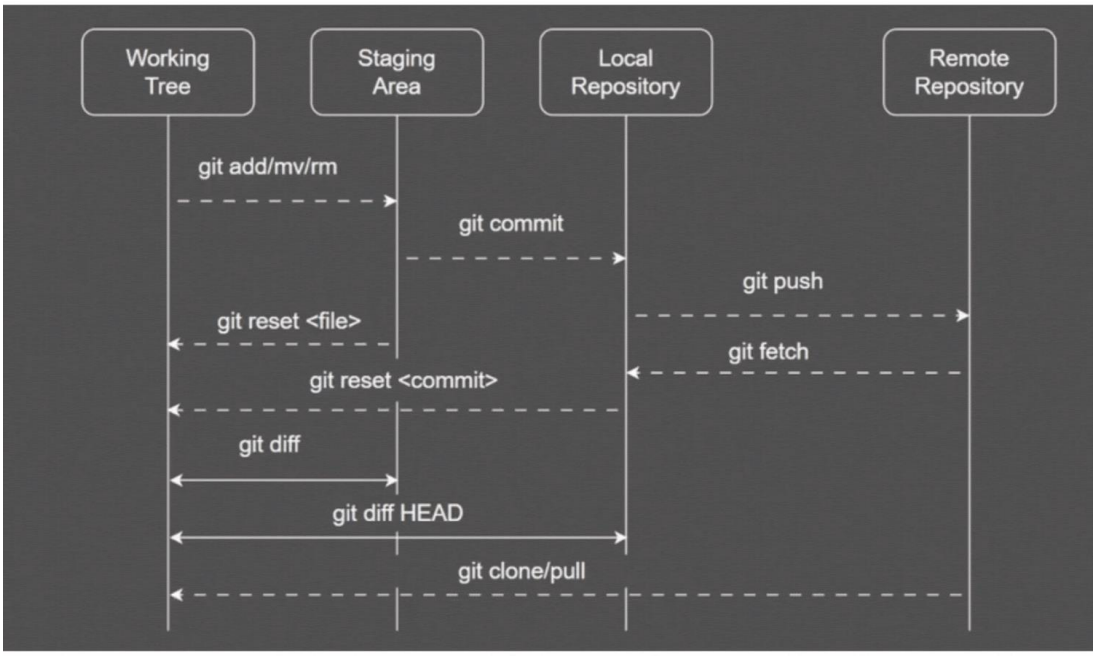
To work on an existing project from GitHub/GitLab/Tuleap.

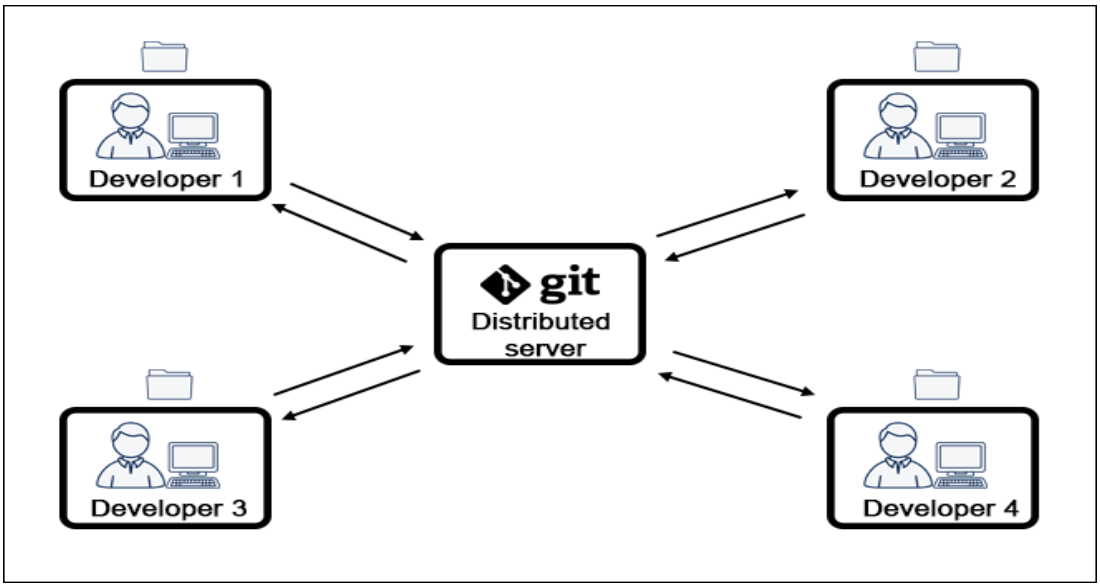
- git clone <repo_url> # Copy the remote repository to your local system
- cd cloned_repo # Move into the cloned repo

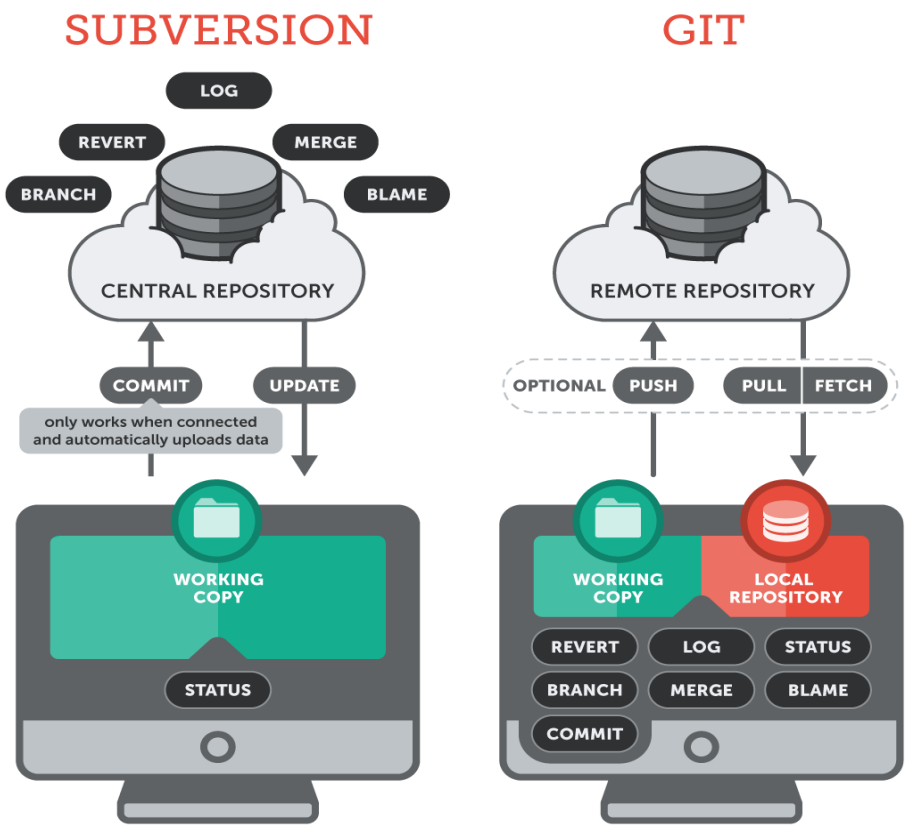
```
MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git status
On branch master

No commits yet

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

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

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git add .

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git commit -m "file-added"
[master (root-commit) 12c2f03] file-added
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 testfile

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git status
On branch master
nothing to commit, working tree clean

MAHESH@DELL MINGW64 ~/Documents/demo (master)
```

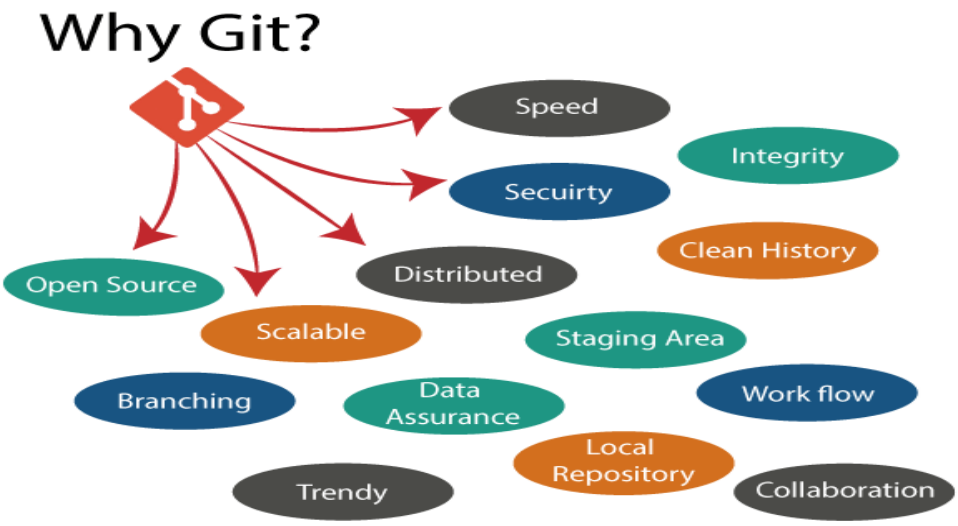
#Concept-4

#Git Architecture






SUBVERSION
GIT
LOG
REVERT
MERGE
BRANCH
BLAME
CENTRAL REPOSITORY
COMMIT
UPDATE
only works when connected
and automatically uploads data
WORKING
COPY
STATUS
REMOTE REPOSITORY
OPTIONAL
PUSH
PULL
FETCH
WORKING
COPY
LOCAL
REPOSITORY
REVERT
LOG
STATUS
BRANCH
MERGE
BLAME
COMMIT


Why Git?
Speed
Integrity
Secuirty
Clean History
Open Source
Distributed
Scalable
Staging Area
Branching
Data
Assurance
Work flow
Local
Repository
Trendy
Collaboration

Working Directory ---> Staging Area ---> Local Repository ---> Remote Repository

(git add) (git commit) (git push)

🛠 Key Components of Git Architecture

Git is built around four main areas where files move during the development process:

I. Working Directory (Workspace)

The working directory is where you create, modify, and delete files. It represents the current state of your project on your local machine.

✅ Example:

1. When you create or edit a file in your project folder, it's in the working directory.

#Commands

- git init # Initialize a new Git repository (creates .git folder)
- git clone <repo_url> # Clone an existing repository from a remote server
- touch file.txt # Create a new empty file
- echo "Hello Git" > file.txt # Create/Modify a file by adding content
- nano file.txt # Open and edit a file using nano editor (Linux/macOS)
- rm file.txt # Delete a specific file
- rm -rf foldername # Forcefully delete a folder and its contents
- mkdir new_folder # Create a new folder/directory
- rmdir new_folder # Remove an empty directory
- mv oldname.txt newname.txt # Rename a file
- mv file.txt new_folder/ # Move a file to another directory
- cp file.txt copy_file.txt # Copy a file
- cp -r folder1/ folder2/ # Copy a folder and its contents
- ls # List files in the directory (Linux/macOS)
- git status # Show the status of modified, staged, and untracked files
- git diff # Show changes in the working directory before staging
- git clean -f # Remove untracked files permanently
- git clean -n # Show untracked files that will be removed

```
MAHESH@DELL MINGW64 ~/Documents/demo (main)
$ echo "hello" >> f1.txt

MAHESH@DELL MINGW64 ~/Documents/demo (main)
$ ls
f1.txt

MAHESH@DELL MINGW64 ~/Documents/demo (main)
$ git init
Initialized empty Git repository in C:/Users/MAHESH/Documents/demo/.git/

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        f1.txt

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

MAHESH@DELL MINGW64 ~/Documents/demo (master)
```

II. Staging Area (Index or Cache)

The staging area acts as a buffer zone between the working directory and the local repository. Files in the staging area are prepared for the next commit, meaning only staged files will be included in the next version of the project.

or

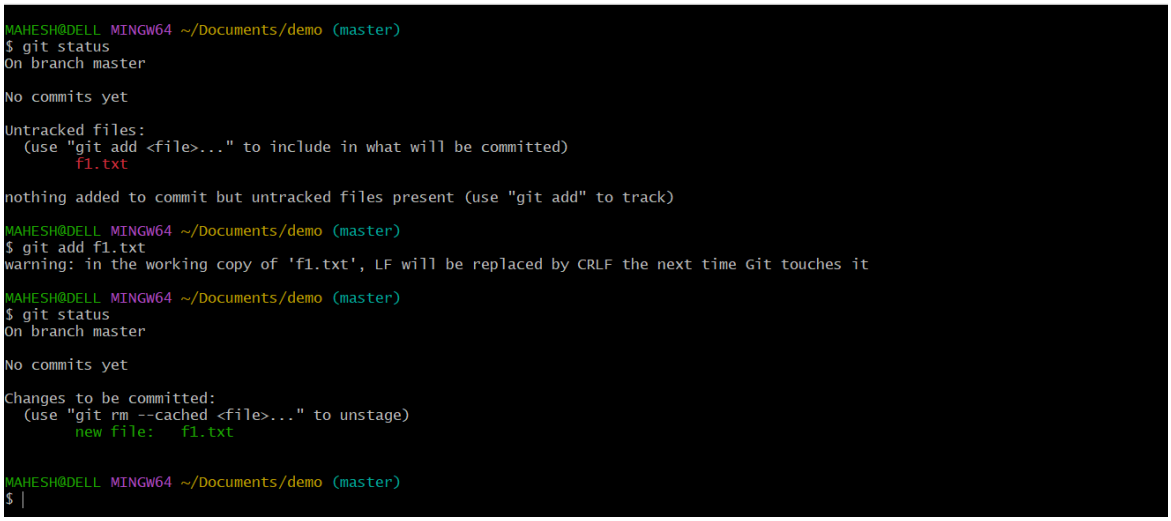
The staging area holds changes that are ready to be committed. Only staged files will be saved in the next commit.

✅ Example:

1. When you run git add file.txt, the file moves from the working directory to the staging area.

#Commands

- git add file.txt # Stage a specific file
- git add. # Stage all modified and new files
- git status # Check which files are staged
- git reset file.txt # Unstage a file (move it back to working directory)
- git diff --staged # Show differences in staged files
- git ls-files --stage # List staged files with details
- git reset file.txt # Unstage a specific file (move it back to the working directory)
- git reset. # Unstage all files
- git reset --hard # Reset both the staging area and working directory
- git restore --staged file.txt # Alternative way to unstage a file
- git clean -n # Show untracked files that will be deleted
- git clean -f # Force deletes untracked files
- git clean -fd # Delete untracked files and directories



III. Local Repository (. git folder)

Your local repository stores all committed changes along with the project's history. The data is stored inside the. git folder.

or

The local repository is where Git permanently stores committed changes. It maintains the full history of the project, allowing developers to track changes, revert, and collaborate effectively. The repository is stored in a hidden. git directory inside the project folder.

✅ Example:

1. After staging a file, you run git commit -m "Added new feature", which saves the changes in your local repository.
2. When you run git commit -m "Added new feature", the changes are permanently saved in your local repository.

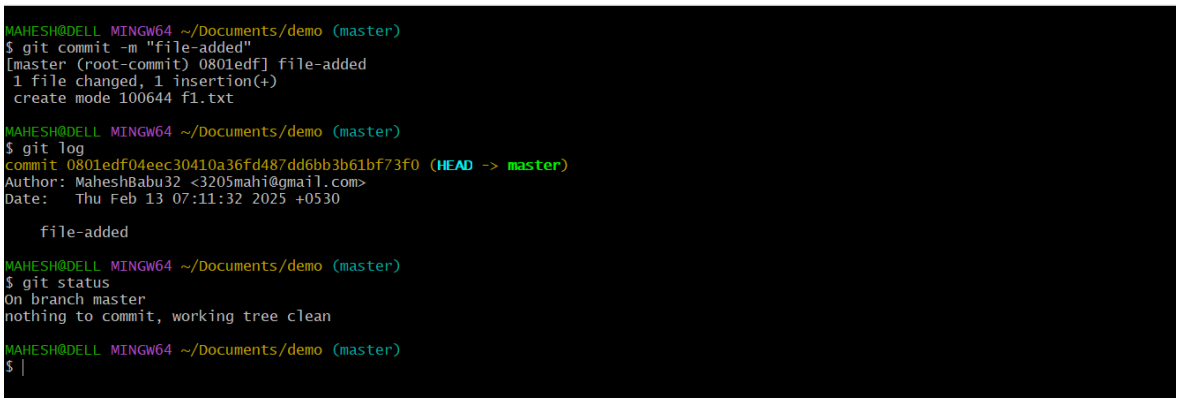
#Commands

- git commit -m "Your commit message" # Save changes to the local repository
- git log # View commit history
- git log --oneline #Show compact commit history
- git log --graph --decorate #Show commit graph with branch names
- git log --all # Show commits from all branches
- git log --graph # Display commits in a graph-like structure
- git log --decorate # Show branch names and tags in the log
- git log --stat # Show commit history with file changes
- git log --patch # Show detailed changes in each commit
- git status # Check the current status of the working directory
- git diff # View upstaged changes
- git diff --staged # View changes staged for commit
- git diff HEAD~1 # Compare current version with the previous commit
- git diff <commit_id> <commit_id> # Compare two specific commits
- git reset --soft HEAD~1 # Undo last commit but keep changes staged
- git reset --mixed HEAD~1 # Undo last commit and unstage changes
- git reset --hard HEAD~1 # Undo last commit and delete changes

- git revert <commit_id> # Create a new commit that undoes a specific commit
- git show <commit_id> # Show details of a specific commit
- git reflog # Show a history of all HEAD movements
- git shortlog #Summarize commit history by author
- git blame <file_name> # Show who modified each line of a file

#Note

💡 Important: The local repository is separate from the remote repository. Until you push your changes, they exist only on your local machine.



IV. Remote Repository (GitHub, GitLab, Bitbucket, Tuleap, etc.)

--A remote repository is a shared version of a Git project stored on an online platform (like GitHub, GitLab, Bitbucket, or Tuleap). Developers use remote repositories to collaborate by pushing and pulling changes between local and remote copies.

✅ Example:
You run git push origin main/master to upload your local commits to GitHub, Tuleap making them accessible to other developers.

✅ Key Concepts of a Remote Repository

1. A remote repository enables multiple developers to work on the same project from different locations.
2. Developers push their local changes to the remote repository to share them with others.
3. They pull updates from the remote repository to keep their local copy up to date.
4. Teams use remote repositories for collaboration, backups, and CI/CD automation.

✅ Common Remote Repository Platforms

Platform	Description
GitHub	The most popular Git hosting service, widely used for open-source and private projects.

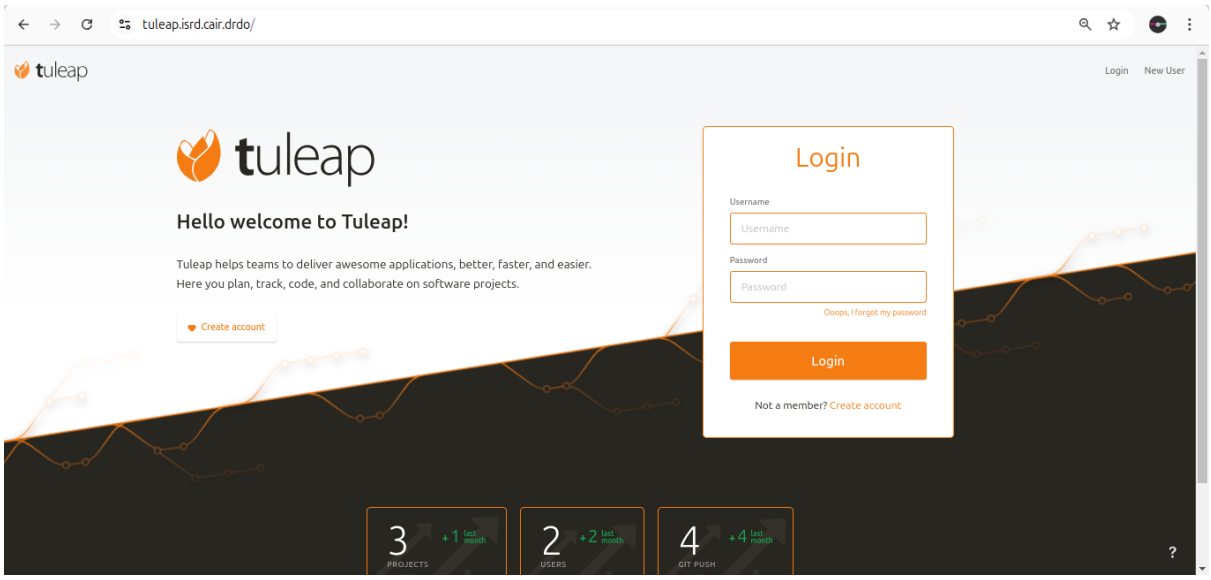
GitLab	Provides Git hosting with built-in DevOps and CI/CD features.
Bitbucket	Offers Git repository management, commonly used in enterprise environments.
Tuleap	An open-source project management and Git repository hosting tool.

#Commands

- git remote add origin <repo_url> # Connect local repo to a remote repository
- git push origin main/master # Pushes local commits to the remote repository. (main or master depending on the default branch)
- git pull origin main/master # Fetch and merge latest changes from remote repository
- git fetch origin # Fetch changes from remote without merging
- git merge origin/main/master # Merge fetched changes into local branch
- git clone <repo_url> #To copy an existing remote repository to your local machine
- git remote -v #To check which remote repositories are linked to your project
- git remote remove origin #Removes a remote repository from your project.

#Note:

By default, Git used to name the default branch "master", but now "main" is the new standard default branch name. In some repositories, you might need to use



#Concept-5

#Git Status Life Cycle--5

The git status command shows the state of your working directory and staging area, helping you track changes, file statuses, and what needs to be committed. The life cycle of the status is tied to how files move between the Working Directory, Staging Area (Index), and Local Repository.

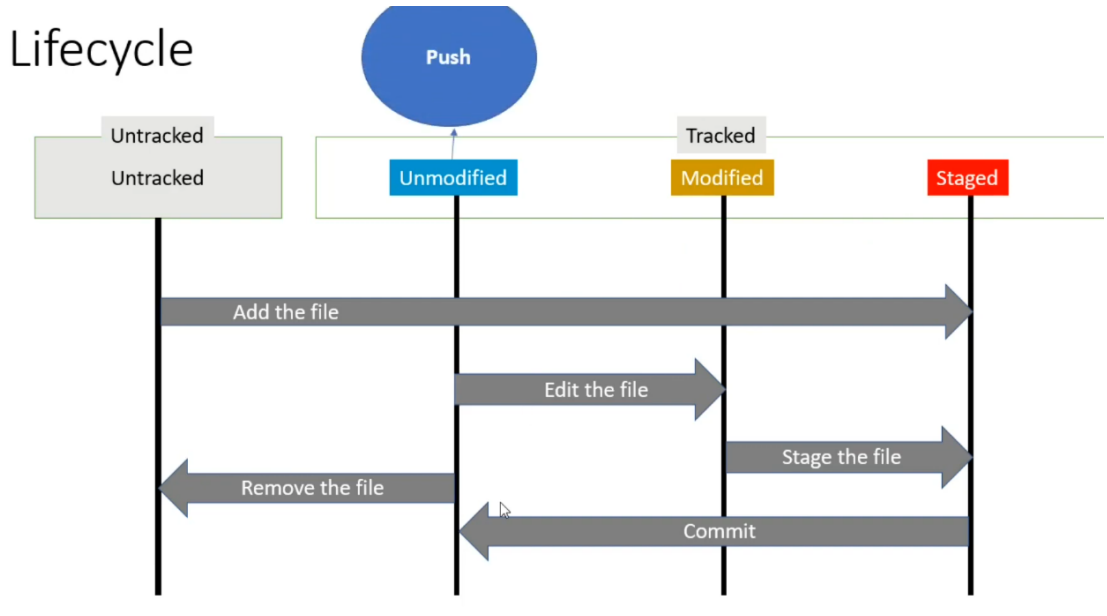
#Architecture

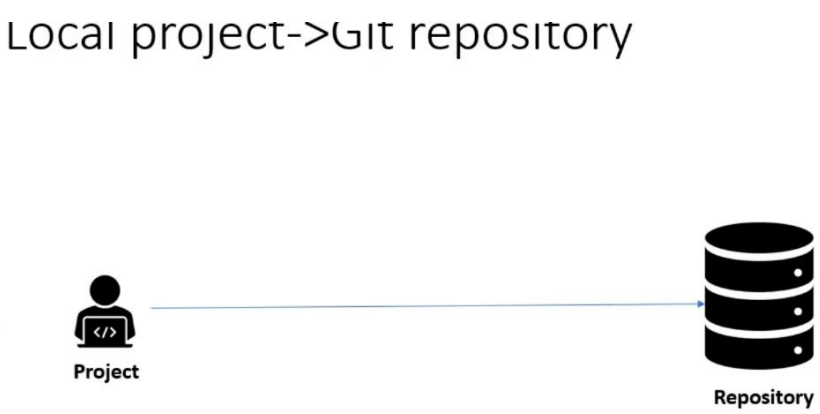
Working Directory ——> Staging Area ——> Local Repository ——> Remote Repository

(Untracked)	(Staged)	(Committed)	(Pushed)
↓	↓	↓	↓
git add file.txt	git commit -m "Message"	git push origin main	→
|	|	|	
git status	git status	git status	
|	|	|	
git rm	git pull / git push	git fetch	

#Git Status Life Cycle: Overview

1. Untracked: Files that have been created but not yet added to Git.
2. Tracked: Files that are already part of the Git repository.
3. Staged: Files that are added to the staging area, ready to be committed.
4. Committed: Files that are safely stored in the local repository.





#Example: Complete Workflow with git status

- mkdir my_project
- cd my_project
- git init
- touch file1.txt file2.txt file3.txt

#Step Check the Status of Untracked Files

- git status

#Add Files to the Staging Area

- git add .
- git status #Check the Status of Staged Files

#Commit the Changes

- git commit -m "Initial commit with three files"
- git status #Check the Status After Committing

1. Untracked Files 🚫

What it is:

When you first create a new file in your project directory, Git doesn't track it. The file is in the untracked state. It status

#git status

#git add new file.txt #To start tracking the file

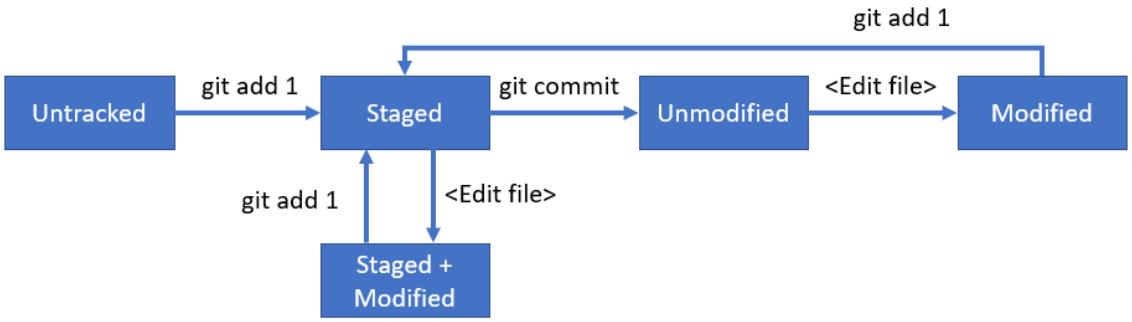
2. Modified Files 📝

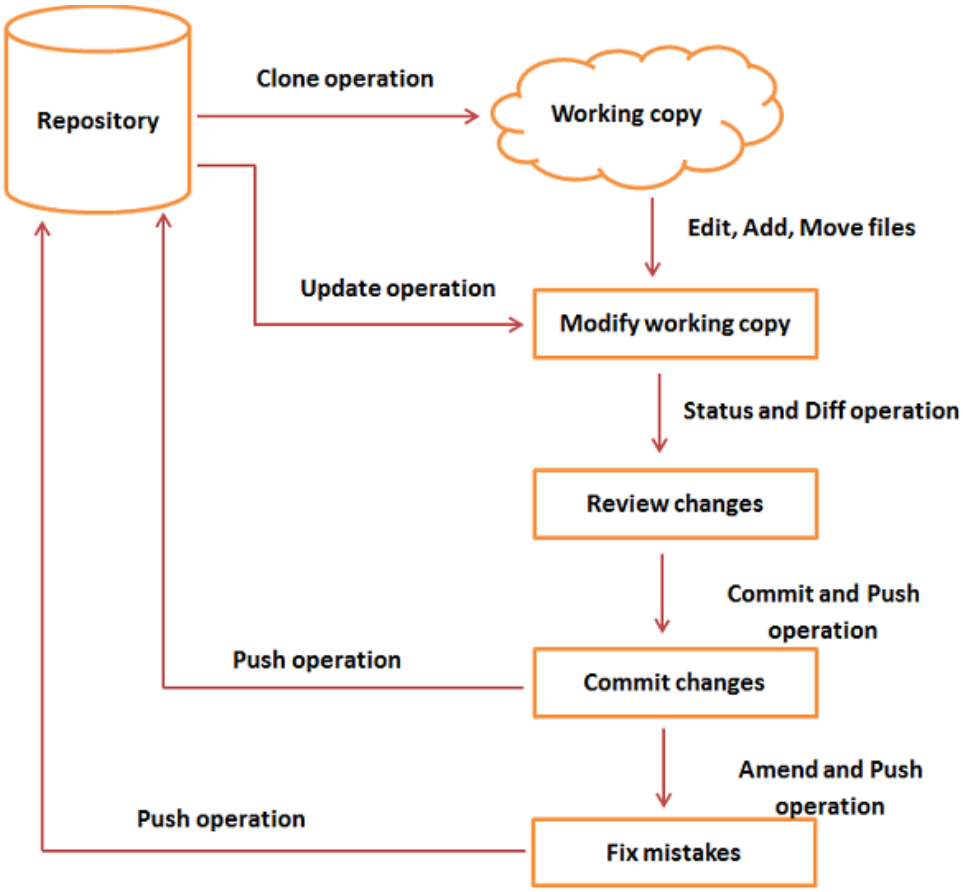
What it is:

A modified file is one that has already been added to Git's tracking system, and you've made changes to it. However, the changes haven't yet been staged for commit.

#Git Status Lifecycle Summary

File State	Command/Action	Example Command
Untracked	Add file to Git tracking system	git add file.txt
Modified	Stage modified file for commit	git add file.txt
Staged	Commit staged files to local repository	git commit -m "Message"
Committed	Push changes to remote repository	git push origin main
Unstaged (after commit)	Re-stage modified file	git add file.txt





#Concept-6

#. Ignoring files in Git

#The main role of. gitignore is to tell Git which files or directories to ignore and not track in the version control system.

#What is. gitignore?

The. gitignore file is a plain text file where each line contains a pattern for files or directories that Git should ignore. These files/directories won't be staged, committed, or tracked by Git.

#Why Use .gitignore?

--Avoid tracking unnecessary files (e.g., build artifacts, logs, temporary files).

--Prevent sensitive data (e.g., API keys, passwords) from being committed.

--Keep your repository clean and focused on source code.

#Note

1. Git Tracks Files, Not Empty Directories:

#Creating a .gitignore File

--Location: Place the. gitignore file in the root directory of your Git repository.

--Naming: The file must be named. gitignore (with a leading dot).

--Format: Each line in the file represents a pattern to ignore.

#How to Create a .gitignore File:

1. Create a. gitignore File: You can create a. gitignore file in the root directory of your project:

process:

#Example-1

- mkdir my_project
- cd my_project
- git init

2. Create the .txt files:

- touch f1.txt f2.txt f3.txt f4.txt f5.txt f6.txt

3. Create the directories (m1, m2, etc.):

- mkdir m1 m2 m3 m4 m5
- ls
- touch. gitignore
- nano. gitignore

Ignore all .txt files

*.txt

Ignore all directories

- */git status
- git rm -r --cached.

4. add and commit

- git add.
- git commit -m "Add initial files and directories"
- git status –ignored

#Example2

Step 1: Initialize a Git Repository

- mkdir my-project
- cd my-project
- git init

Step 2: Create Some Files and Directories

- touch index.html style.css script.js
- mkdir logs
- touch logs/error.log logs/access.log
- touch .env
- touch temp.txt

#structure

my-project/

├── index.html

├── style.css

├── script.js

├── logs/

│ ├── error.log

│ └── access.log

├── .env

└── temp.txt

Step 3: Create a .gitignore File

- touch .gitignore
- nano .gitignore

Ignore all .log files

*.log

Ignore the .env file (sensitive data)

.env

Ignore the temp.txt file

temp.txt

Ignore everything in the logs directory

logs/*

- git status

Step 4: Add and Commit the Files

- git add.
- git commit -m "Initial commit with. gitignore"

Step 5: Verify Ignored Files

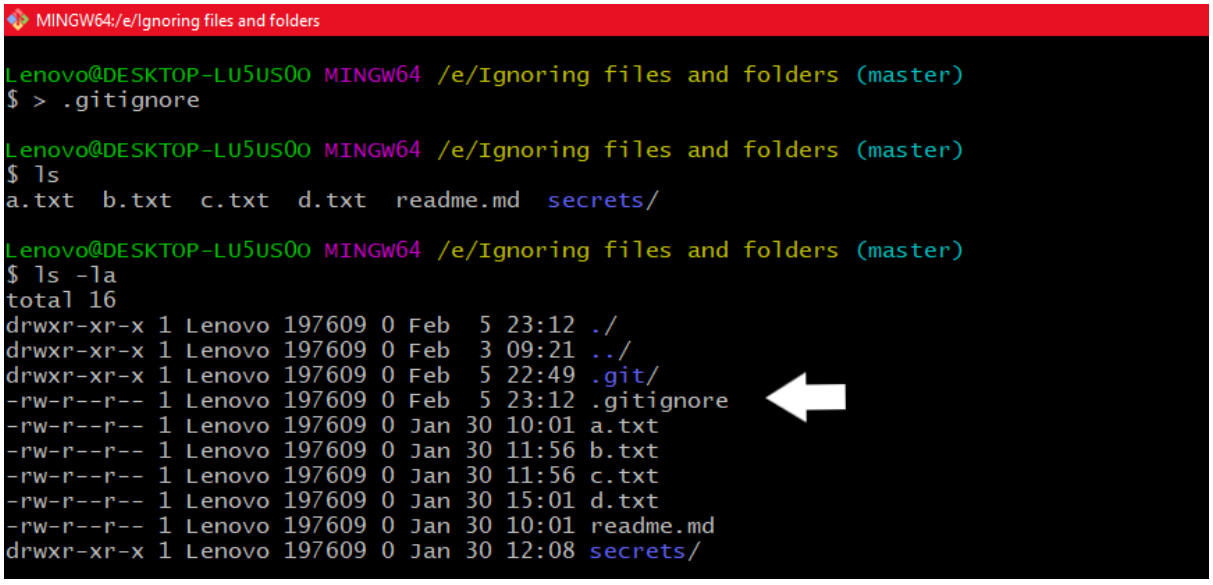
- git status –ignored

Rules

- #Comments or blank lines
- Specific file: **intro.html**
- File pattern: *.txt

Example

```
[user@localhost] $ touch .gitignore
```

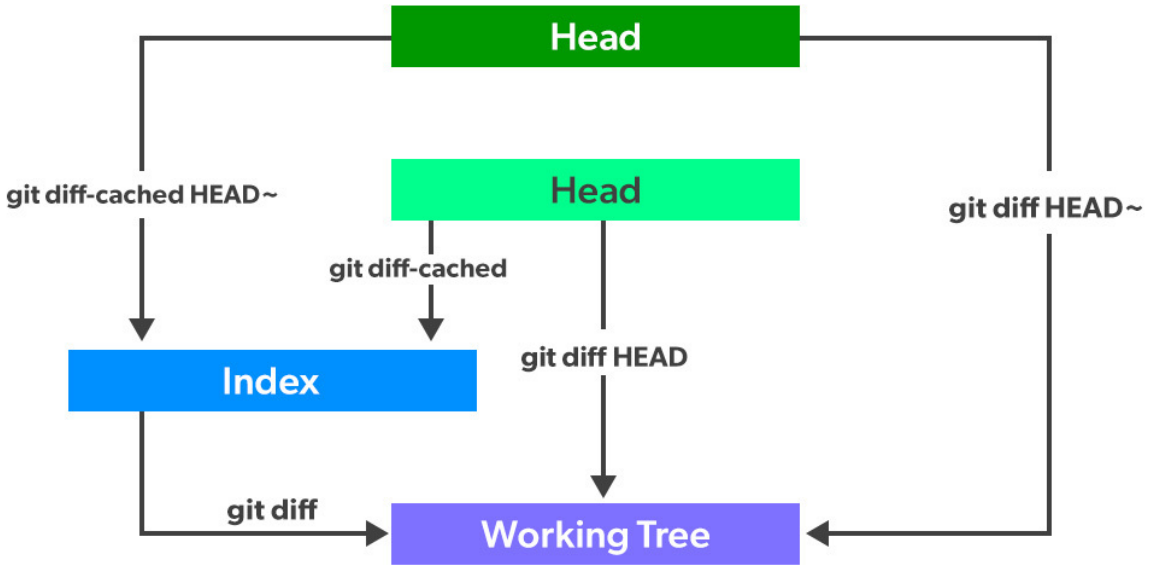


#Concept-7

#Git Diff

git diff is a Git command used to display the differences between various states of a repository. It shows you the changes made to files in the working directory, staging area, or commits. These differences are presented at the line level, helping you see exactly what has been added, removed, or modified in a file.

#Git diff Architecture



#What git diff Shows

1. Unstaged Changes: The differences between the working directory and the staging area.
2. Staged Changes: The differences between the staging area and the last commit.
3. File Changes: The actual changes made to the contents of the files.

Output:

+: Line added.

-: Line removed.

#Create a New Directory for Your Project

- mkdir my_project
- cd my_project

#Step 2: Initialize a Git Repository

- git init

#Step 3: Create Files in the Directory

- touch file1.txt file2.txt file3.txt

#Step 4: Check the Status of the Repository

- git status

#Step 5: Add Files to the Staging Area

- git add .

#Step 6: Commit the Files

- git commit -m "Initial commit with files"

#Step 7: Modify One of the Files

- echo "Hello, Git!" >> file1.txt

#Step 8: Check Git Status Again

- git status

#Step 9: Use Git Diff (Before Staging)

- git diff

#Step 10: Stage the Changes

- git add file1.txt

#Step 11: Check Git Diff Again (After Staging)

- git diff
- git commit -m "Updated file1.txt with greeting"
- git status

#Key Differences: git status vs git diff

git status:

-Tells you the state of your working directory and staging area.

-Shows untracked files, modified files, and files staged for commit.

-Gives overall information on what needs to be added or committed.

git diff:

-Compares changes between files in the working directory and the staging area or between commits.

-Shows line-by-line differences in the content of files.

```
MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ nano f1.txt

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   f1.txt

no changes added to commit (use "git add" and/or "git commit -a")

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git diff
warning: in the working copy of 'f1.txt', LF will be replaced by CRLF the next time Git touches it
diff --git a/f1.txt b/f1.txt
index 4bff921..8381df5 100644
--- a/f1.txt
+++ b/f1.txt
@@ -1 +1 @@
-hello welcome
+hello welcome to git

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$
```

```
MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git commit -m "modified"
[master c689005] modified
 1 file changed, 1 insertion(+), 1 deletion(-)

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$ git diff

MAHESH@DELL MINGW64 ~/Documents/demo (master)
$
```

#Concept-8

#Git Commit

A commit in Git represents a snapshot of your changes. It is essentially a record of changes that you've made in your project at a specific point in time. When you commit, you're saving the state of your files and the changes you made, with a commit message explaining what the changes are about.

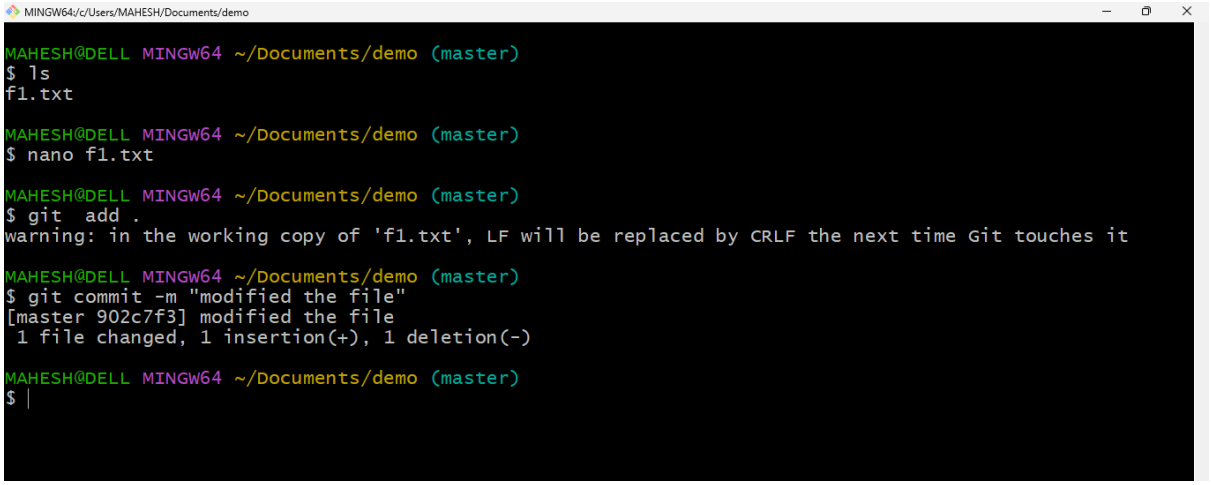
#What Happens When You Commit in Git?

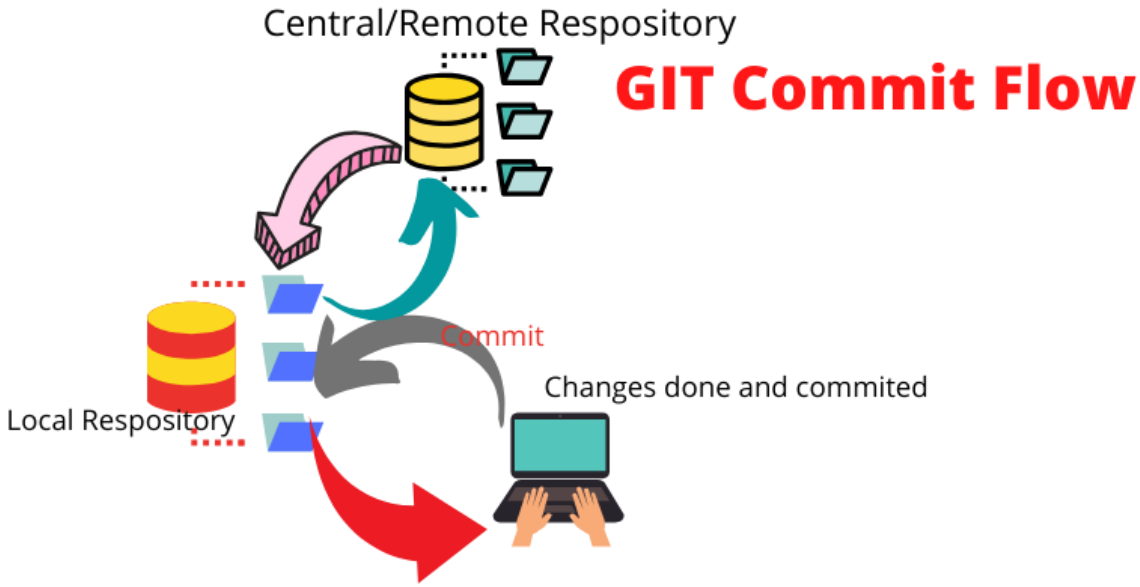
--It takes a snapshot of the changes in the staging area (the files that you've added with git add).

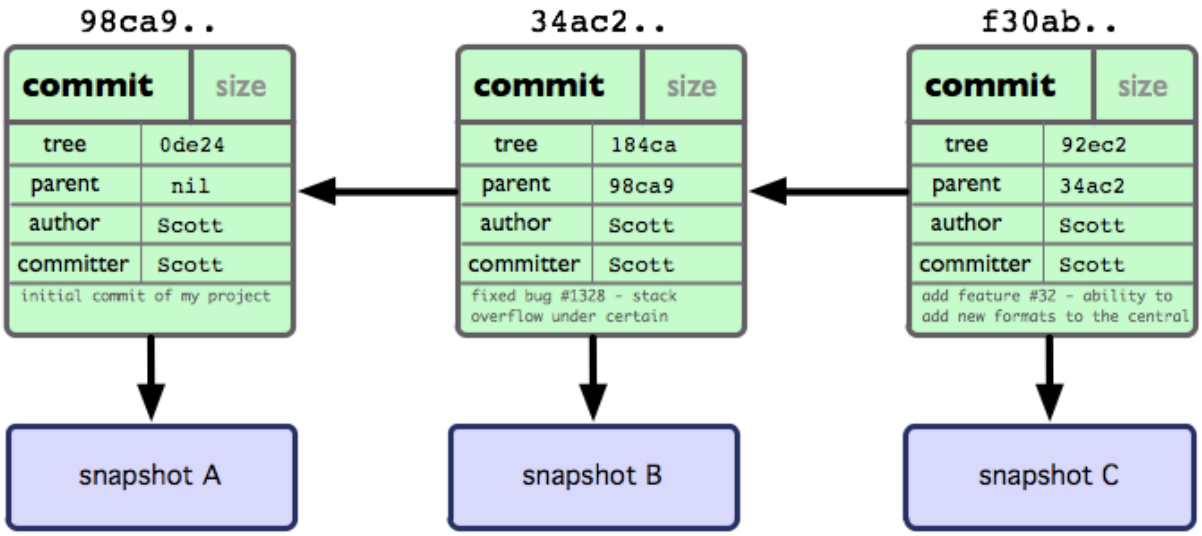
--It records this snapshot in the repository's history along with a unique ID (called a commit hash).

#Git Commit Workflow Example (From Start to Finish)

- mkdir my_project
- cd my_project
- git init
- touch index.html
- echo "Hello World!" > index.html
- git status
- git add index.html
- git commit -m "Initial commit with index.html"
- echo "<h1>Welcome</h1>" >> index.html
- git status
- git add index.html
- git commit -m "Added header to index.html"
- git push origin master
- git show(commit)




Central/Remote Respository
GIT Commit Flow
Commit
Changes done and commited
Local Respository


98ca9..
commit
size
tree
0de24
parent
nil
author
Scott
committer
Scott
initial commit of my project
34ac2..
commit
size
tree
184ca
parent
98ca9
author
Scott
committer
Scott
fixed bug #1328 - stack overflow under certain
f30ab..
commit
size
tree
92ec2
parent
34ac2
author
Scott
committer
Scott
add feature #32 - ability to add new formats to the central
snapshot A
snapshot B
snapshot C

#Concept-9

#Git log

The git log command in Git is used to display the commit history of a repository. It shows a list of commits, including their unique commit IDs (hashes), author information, dates, and commit messages. This allows you to view the history of your project, track changes over time, and navigate the commit history.

#What Does git log Show?

1. Commit Hash: A long, unique identifier for the commit (a SHA-1 hash).

2. Author: The name and email of the person who made the commit.

3. Date: The date and time when the commit was made.

4. Commit Message: A brief description of what changes were made in the commit.

#Basic Git Log Command:

- git log

#Example: Basic Git Log Usage

- mkdir my_project
- cd my_project
- git init
- touch file1.txt
- echo "Hello Git!" > file1.txt
- git add file1.txt
- git commit -m "Initial commit with file1.txt"
- touch file2.txt
- echo "This is another file." > file2.txt
- git add file2.txt
- git commit -m "Added file2.txt"

#View Commit History:

- git log

#Show Commit History in a Simple One-Line Format:

- git log --oneline

#View Commit History with Graph:

- git log --oneline --graph

#View Specific Author's Commits:

- git log --author="Your Name"

View Commits Within a Specific Date Range

- git log --since="1 week ago"

#Show Changes with Each Commit (Patch Format):

- git log -p

#Git Log and Branching

- git log --oneline --graph --all

#Note

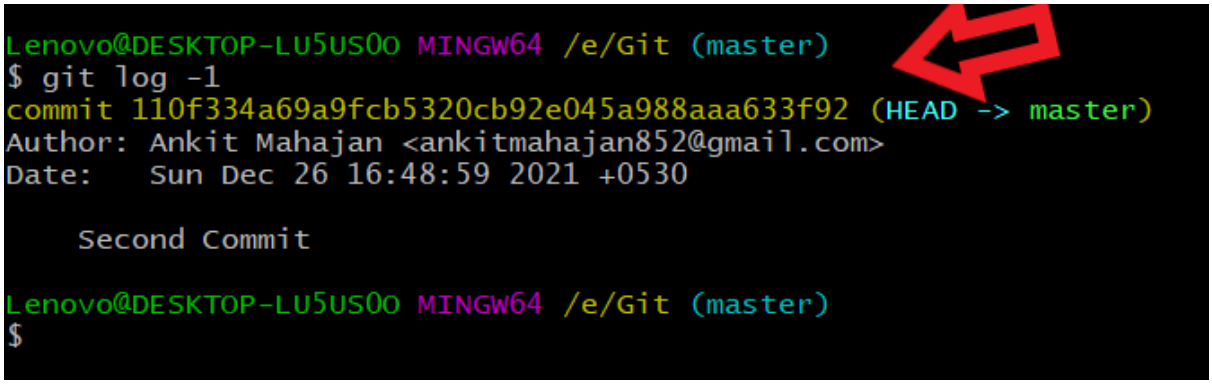
In case your project has multiple branches, git log can also be used to visualize commits across different branches:

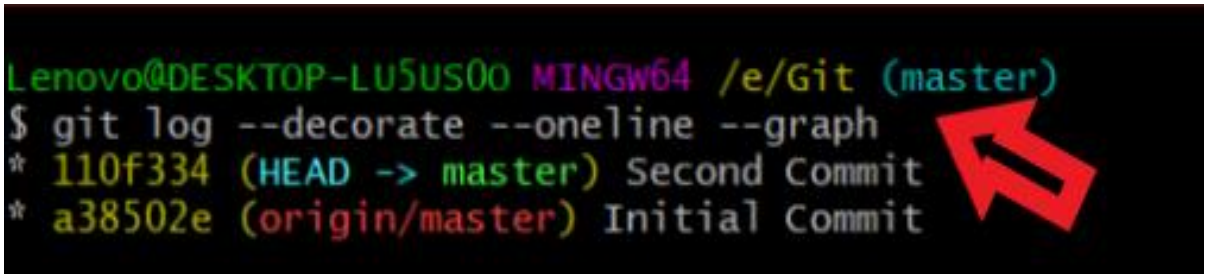
#Git Log with Specific File History

- git log file1.txt

#Summary:

1. git log shows the history of commits in a Git repository, including commit hashes, authors, dates, and messages.
2. You can customize git log with options like --oneline, --graph, and --since to view commits in different formats.
3. git log can be used to review changes, filter commits, and track project history.
4. It's a powerful tool for inspecting the history of a repository and navigating between different commits.


Lenovo@DESKTOP-LU5US0O MINGW64 /e/Git (master)
$ git log -1
commit 110f334a69a9fcb5320cb92e045a988aaa633f92 (HEAD -> master)
Author: Ankit Mahajan <ankitmahajan852@gmail.com>
Date: Sun Dec 26 16:48:59 2021 +0530
Second Commit
Lenovo@DESKTOP-LU5US0O MINGW64 /e/Git (master)
$


Lenovo@DESKTOP-LU5US0O MINGW64 /e/Git (master)
$ git log --decorate --oneline --graph
* 110f334 (HEAD -> master) Second Commit
* a38502e (origin/master) Initial Commit

#Concept-10

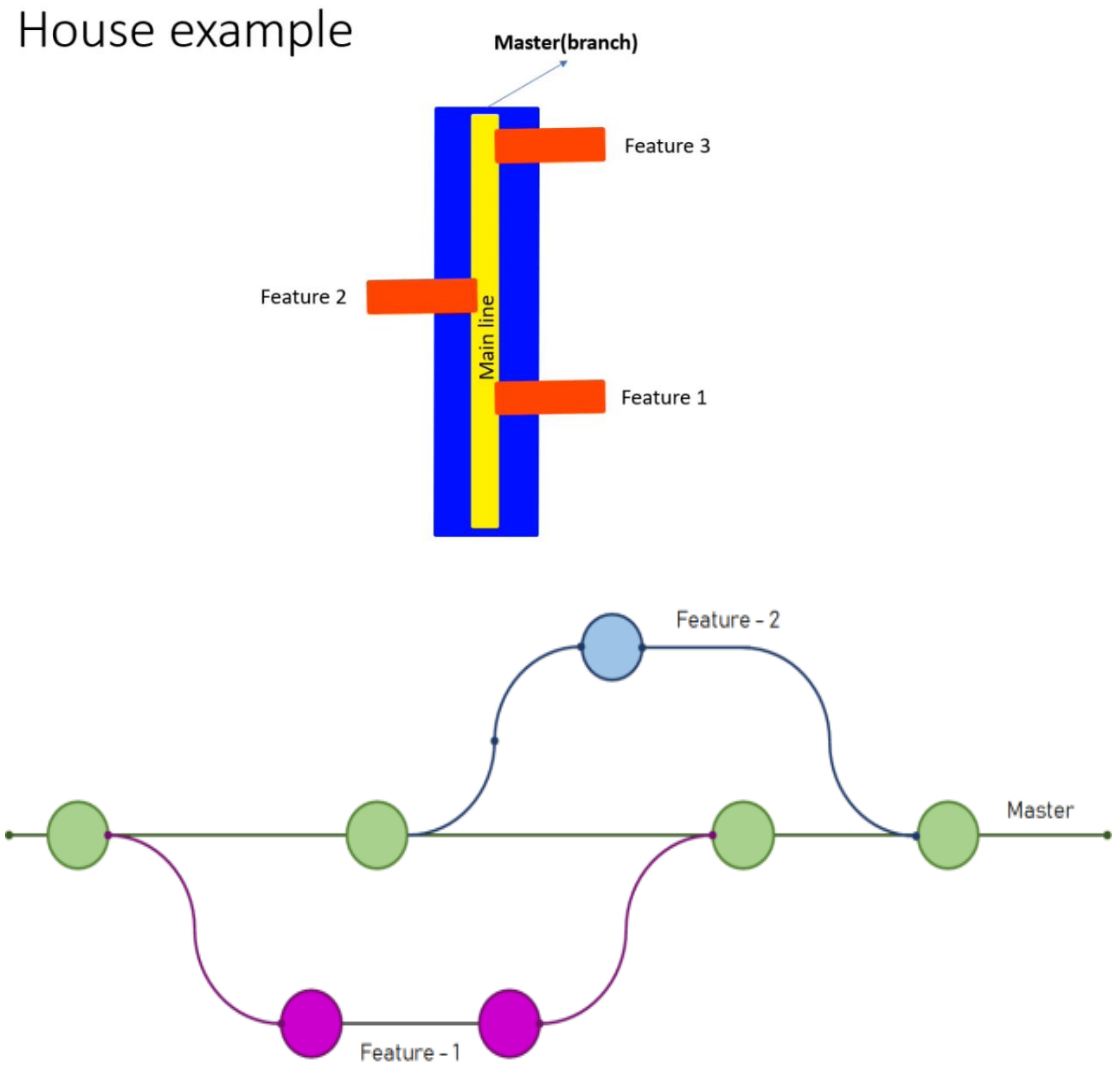
#Git Branching & Merging (With Examples)

#What is Git Branching?

Branching allows developers to work on different features or fixes independently without affecting the main project.

- Main branch (Default): main (or master)
- Feature branches: Used for development, bug fixes, or testing
- Merging: Combines branch changes back into main/master

#Architecture of git branch



◈ Step 1: Check Existing Branches

Before creating a new branch, check the current branches in the repository.

- git branch # List all branches
- git branch -a # List all local & remote branches

◈ Step 2: Create a New Branch

- git branch feature-1 # Create a new branch called "feature-1"

◈ Step 3: Switch to the New Branch

- git checkout feature-1 # Switch to "feature-1" branch
  # OR (newer command)
- git switch feature-1

◈ Step 4: Make Changes and Commit

- echo "New Feature" > feature.txt # Create a new file
- git add feature.txt # Stage the file
- git commit -m "Added new feature file" # Commit the changes

◈ Step 5: Merge the Branch into main

- git checkout main # Switch back to main branch
  # OR
- git switch main/master
- git merge feature-1 # Merge feature-1 branch into main

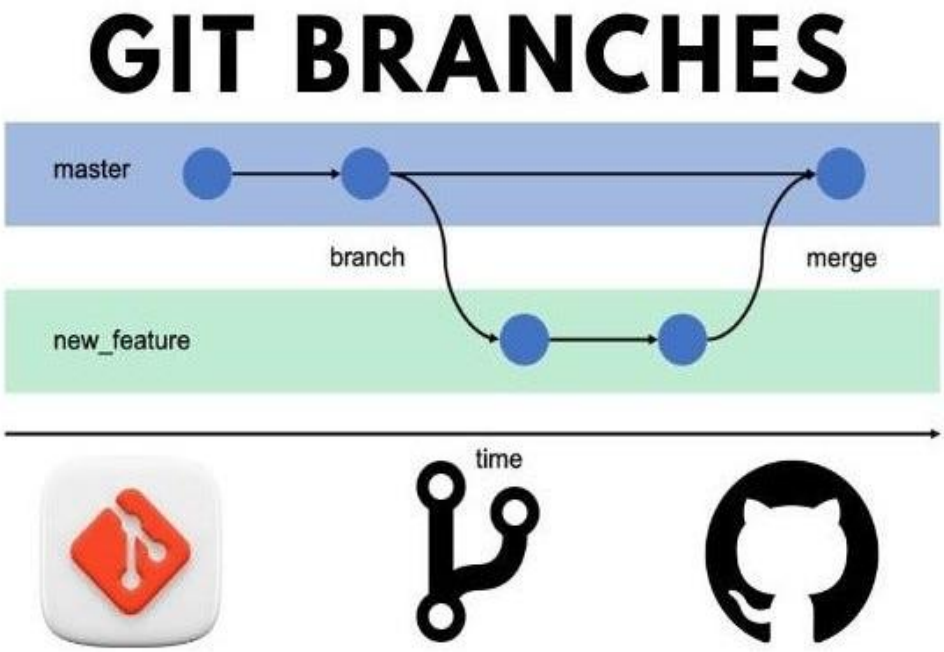
◈ Step 6: Delete the Merged Branch

After merging, delete the feature branch to keep things clean.

- git branch -d feature-1 # Delete the local branch
- git push origin --delete feature-1 # Delete the remote branch

◈ Step 7: Handling Merge Conflicts (If Any)

- git add file.txt
- git commit -m "Resolved merge conflict"



Branch - Pointer

- Pointer to one of the commit's

#What is a Git Merge?

Git Merge is a command used to integrate changes from one branch into another. It combines the history of two branches, bringing changes from the source branch into the target branch, without losing any commit history.

In other words, merging is the process of joining different lines of development into a single branch.

#Types of Merges:

#Fast-Forward Merge

- git checkout main
- git merge feature-1 # No conflicts, fast-forward merge

#Three-Way Merge

- git checkout main
- git merge feature-1

#Example:

- git checkout -b feature-1 # Create and switch to a new branch

##git branch:(concept) in-depth

#Note

1. Why? Because both branches (master and mahesh) point to the same commit, so they share the same content.

2. Now, mahesh, and master all point to the same commit.

3. git rm any branch without commits

#Real Example (Branches are just pointers to commits)

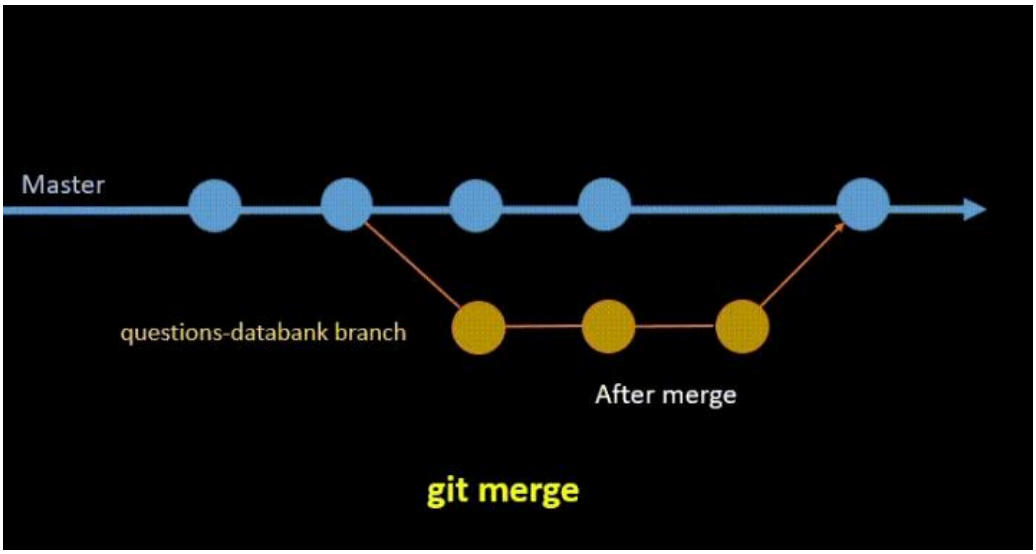
◈ Think of Git as a Library 🕮

--Imagine a library where different peope (branches) are reading the same book (commit).

--When a new person (branch) starts reading, they begin from the same page (commit).

--The book (files) hasn't been copied, just referenced from the same place.

--If one person (branch) starts writing notes (committing changes), only their book changes.



##Best Example to Prove That Git Branches Are Independent 🚀

1. Step 1: Create a New Git Repository

- mkdir test-git # Create a new folder
- cd test-git # Move into the folder
- git init # Initialize a Git repository

2. Step 2: Create and Commit a File in master

- echo "Hello, this is the master branch" > file.txt
- git add file.txt
- git commit -m "Initial commit in master"

#Note ✓ This creates file.txt and commits it in the master branch.

3. Step 3: Create a New Branch (feature)

- git checkout -b feature # Create and switch to the 'feature' branch

#Note ✓ At this point, both master and feature have the same content.

4. Step 4: Modify file.txt in the feature Branch

- echo "This is a change in the feature branch" >> file.txt
- git commit -am "Updated file.txt in feature branch"

#Note ✓ Now, feature has a new commit with changes.

5. Step 5: Switch Back to master and Check the File

- git checkout master # Switch back to master
- cat file.txt # Check the file content

#Note

✓ You will NOT see the changes from the feature branch!

✓ This proves that branches are independent after changes.

6. Step 6: Merge feature into master

- git merge feature

#Note ✓ Now master will have the changes from feature.

##Final Understanding

1. Before making changes, all branches share the same content.
2. After modifying a branch, changes stay in that branch unless merged.
3. Switching between branches restores the files to that branch's latest commit.

Real-Time Git Workflow in Software Development

Setting Up the Project (Initial Stage)

#stages

1. Step 1: Create a Remote Repository

--A project repository is created on Tuleap, GitHub, GitLab, Bitbucket, or any Git hosting service.

--This acts as the centralized storage for the code.

2. Step 2: Clone the Repository (Developers Get the Code)

#Each developer clones the project repository onto their local system.

- --git clone https://github.com/company/project.git
- cd project

#Note: This creates a local copy of the remote project.

3. Working on a New Feature or Bug Fix

#Note: A developer never works directly on the main (or master) branch. Instead, they create a new feature branch.

- git checkout -b feature

4. Make Changes and Commit

#Note: Developers write code, add files, and commit their work frequently.

- echo "New login feature" > login.js
- git add login.js
- git commit -m "Added login feature"

5. Push the Branch to the Remote Repository

#Note: After making changes, the branch is pushed to the remote repository.

- git push origin feature

6. Create a Pull Request (PR)

#Note

On Tuleap/GitHub/GitLab/Bitbucket, the developer creates a Pull Request (PR).

Other developers review the code, suggest improvements, and approve changes.

Automated CI/CD pipelines may run tests to ensure the new code doesn't break existing functionality.

7. Merge the Feature Branch Into main

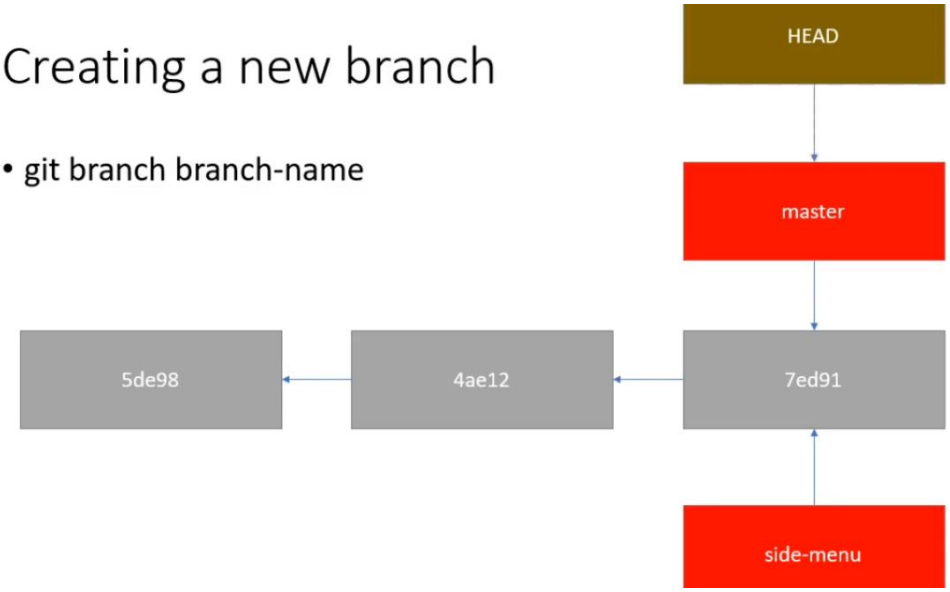
#Note: Once approved, the feature branch is merged into the main branch.

- git checkout main
- git pull origin main # Ensure its updated
- git merge feature
- git push origin main

#Note ✓ This ensures the latest code is now part of the main project.

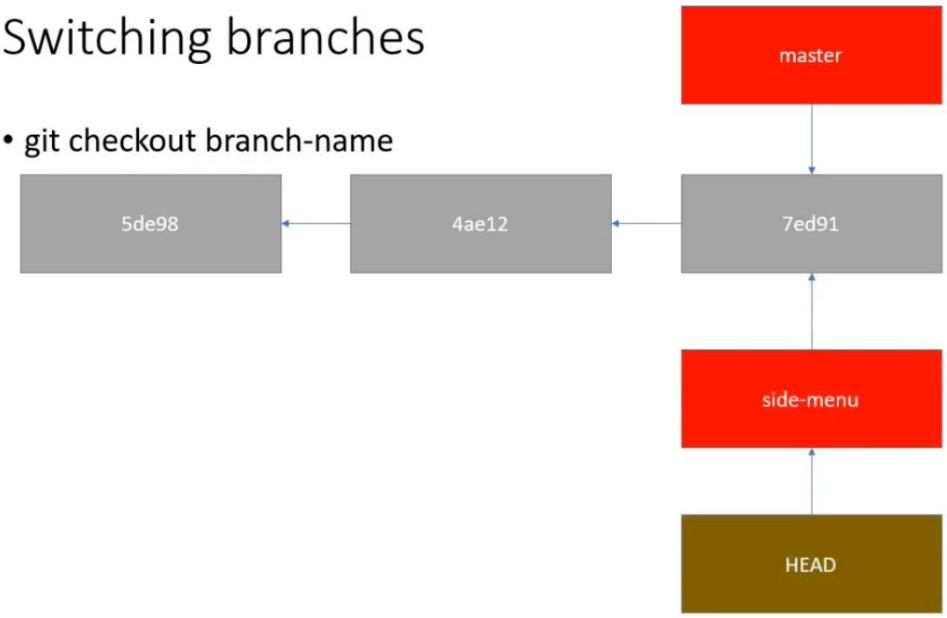
Creating a new branch

- git branch branch-name



Switching branches

- git checkout branch-name




```
newuser@localhost:~/my_project$ git log --oneline --all
d85467e (HEAD -> mahi) k
d5dc776 (master) k
ccaa89b (mahesh) new
54a0b0a new
b0679fc new
83f5c27 new
newuser@localhost:~/my_project$ git log --oneline --graph --all
* d85467e (HEAD -> mahi) k
* d5dc776 (master) k
| * ccaa89b (mahesh) new
|/
* 54a0b0a new
* b0679fc new
* 83f5c27 new
newuser@localhost:~/my_project$
```

#Difference Between a Pull Request (PR) and Merge in Git

#What is a Pull Request (PR)?

A Pull Request (PR) is a request to merge code from one branch into another (e.g., from feature-login → main).

Purpose of a PR:

✅ Allows code review before merging.

✅ Enables team collaboration (comments, suggestions).

✅ Triggers CI/CD pipelines to check code quality.

✅ Helps keep main branch stable by preventing direct merges.

#🔷 When a PR is Created:

--After a developer completes their work on a feature branch.

--The developer pushes their changes to the remote repository.

--They open a PR in GitHub, GitLab, Bitbucket, or Tuleap.

#Eg: - git push origin feature-login

Go to GitHub/GitLab.

Click on "New Pull Request".

Select feature-login → main.

Add a description & request a review.

##Real time example

👉 PR is like asking for permission before merging code.

Example in Real Life:

#Imagine you are working on a school project with a team.

--You finish writing your part of the report.

--Before adding it to the final document, you show it to your teacher for review.

--Your teacher checks it, gives feedback, and approves it.

Once approved, you add it to the final report.

🔷 In Git, this process is called a Pull Request (PR).

🔷 You ask your team (or manager) to review your code before merging it.

#What is a Merge in Git (10)

👉 Merge is the actual process of combining your code into the main branch.

#Example in Real Life:

After your teacher approves your work, you take your content and add it to the final report.

No more reviews needed; it's just adding your part to the main document.

🔷 In Git, merge is the process of combining the feature branch into the main branch.

🔷 Happens after PR approval.

💻 Git Example of Merging

- git checkout main
- git pull origin main # Get the latest changes
- git merge feature-login
- git push origin main

Where PR (Pull Request) and Merge Happen

🔷 PR Happens Only on Remote (GitHub, Tuleap, etc.)

✓ You cannot create a PR in local Git.

✓ PR is a feature provided by platforms like GitHub, GitLab, Tuleap, and Bitbucket.

✓ PR is used for code review, discussions, and approval before merging.

#git push origin feature-branch

#Note

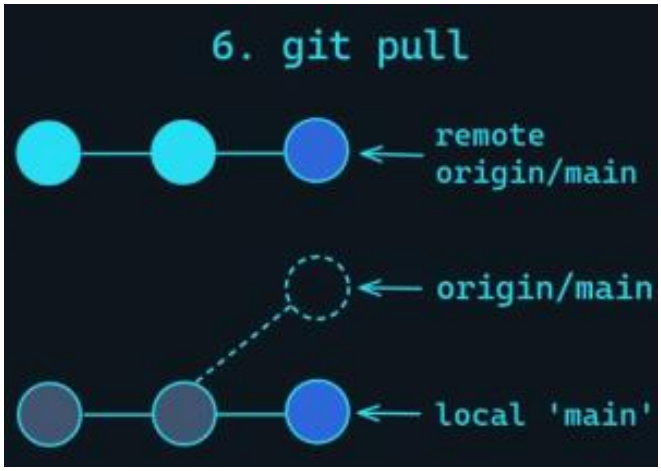
Then, go to GitHub/Tuleap and create a Pull Request (PR).

Other developers review it before merging.

#🔷 Merge Can Be Done Locally OR on Remote

✓ If you merge locally, it happens on your computer before pushing.

✓ If you merge on remote (GitHub/Tuleap), the platform does the merging for you.



#💻 Example of Merging Locally (Git Commands)

- git checkout main
- git pull origin main # Get the latest changes
- git merge feature-branch # Merge locally
- git push origin main # Push the merged code to remote

#final

🚀 Best Practice in Real Projects:

1. Create a PR first (on remote) → Get approvals → Merge after approval.

2. Merging directly in local Git is fine for small projects, but PRs are better for team projects.

Deploy to Production

Once all features are merged into main, the latest code is deployed to production using CI/CD tools like Jenkins, GitHub Actions, GitLab CI, or AWS Code Deploy.

#Commands of Git Branch & Merge Commands Cheat Sheet:

#🔷 Branch Commands

- git branch # List all local branches
- git branch -r # List remote branches
- git branch -a # List all branches (local + remote)
- git branch my-feature # Create a new branch (without switching)
- git checkout my-feature # Switch to an existing branch
- git switch my-feature # Alternative to checkout (newer command)
- git checkout -b my-feature # Create and switch to a new branch
- git switch -c my-feature # Alternative to checkout -b
- git branch -m new-branch-name # Rename the current branch
- git branch -d my-feature # Delete a local branch (if merged)
- git branch -D my-feature # Force deletes a local branch (even if unmerged)
- git push origin --delete my-feature # Delete a remote branch
- git branch -vv # Show tracking branches with upstream info
- git branch --merged # Show merged branches
- git branch --no-merged # Show branches that have not been merged

#🔷 Merge Commands

- git checkout main # Switch to main branch
- git pull origin main # Fetch and merge latest changes from remote main
- git merge my-feature # Merge my-feature into the current branch
- git checkout main && git merge my-feature # Merge and switch in one command
- git merge --no-ff my-feature # Merge with a new commit, even if fast-forward
- git merge --squash my-feature # Squash all commits before merging
- git merge --abort # Abort a merge if conflicts occur
- git merge --continue # Continue a merge after resolving conflicts
- git reset --merge # Reset merge changes (similar to --abort)
- git log --merge # Show commits related to a merge conflict

#🔷 Remote Branch Handling

- git push origin my-feature # Push local branch to remote
- git push --set-upstream origin my-feature # Set up remote tracking for a branch
- git push origin --delete my-feature # Delete a remote branch
- git fetch origin # Fetch updates from remote without merging
- git pull origin my-feature # Pull changes from a specific remote branch
- git rebase origin/main # Reapply commits from the current branch onto main
- git checkout -b my-feature origin/my-feature # Create a local branch from a remote branch

#🔷 Handling Merge Conflicts

- git status # Check which files have conflicts
- git diff # See differences before resolving
- git merge tool # Open merge tool for conflict resolution
- git add. # Add resolved files to staging
- git commit -m "Resolved merge conflicts" # Commit after resolving conflicts
- git merge --abort # Abort merge and go back to the previous state

#Git Concepts Overview

#Note (Git)

Basic Concepts:

- Repository: A database storing project files and their revision history.
- Commit: A snapshot of the project at a specific point in time.
- Branch: A parallel version of the repository that allows for separate development.
- Merge: Combining changes from different branches.
- Clone: Creating a copy of a repository.
- Remote: A version of the repository hosted on the internet or network.

#Key Commands:

- git init: Initializes a new Git repository.
- git clone: Creates a copy of a remote repository.
- git add: Adds changes to the staging area.
- git commit: Records changes to the repository.
- git status: Shows the status of the working directory and staging area.
- git log: Displays the commit history.
- git branch: Lists, creates, or deletes branches.
- git checkout: Switches between branches or restores files.
- git merge: Combines changes from different branches.
- git pull: Fetches and merges changes from a remote repository.
- git push: Uploads local changes to a remote repository.
- git fetch: Downloads changes from a remote repository.
- git rebase: Reapplies commits on top of another base tip.
- git tag: Marks specific points in history as important.
- git stash: Temporarily saves changes.
- git diff: Shows differences between commits, branches, etc.
- git reset: Moves the current branch to a specified commit.
- git revert: Creates a new commit that undoes changes from a previous commit.
- git cherry-pick: Applies changes from an existing commit.
- git bisect: Finds the commit that introduced a bug using binary search.
- git remote: Manages remote repositories.
- git config: Configures Git settings.

#Advanced Concepts

- Staging Area: A place to prepare changes before committing.
- HEAD: A reference to the current branch or commit.
- Detached HEAD: A state where HEAD points directly to a commit, not a branch.
- Rebase vs. Merge: Rebase rewrites commit history, while merge creates a new commit.
- Conflict Resolution: Manually resolving differences between branches.
- Hooks: Scripts that run automatically during Git events.
- Submodules: Nested Git repositories within a main repository.
- Bare Repositories: Repositories without a working directory, used for sharing.

#Workflows

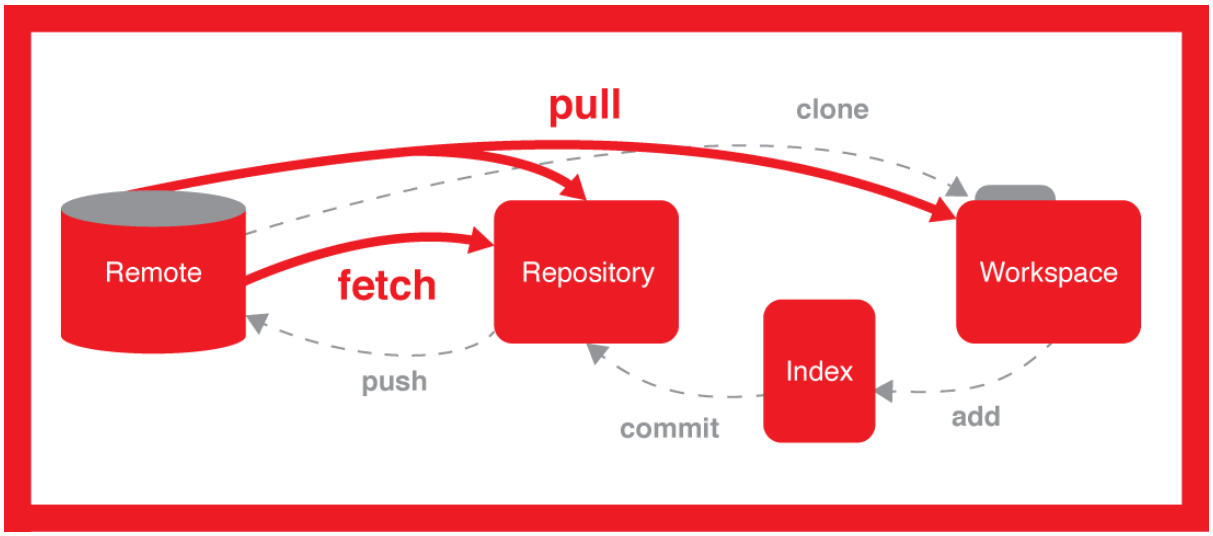
- Feature Branch Workflow: Creating branches for new features.
- Git Flow: A branching model with specific branches for development, release, and hotfixes.
- Forking Workflow: Creating a personal copy of a repository for contributions.

#Collaboration

- Pull Requests: Proposing changes to a repository.
- Code Reviews: Reviewing changes before merging.
- Continuous Integration: Automating the testing and deployment of code.
- Issue Tracking: Managing bugs and feature requests.

#Additional Notes

- Default Branch: Older versions used master; newer versions use main.
- Branch Visibility: The branch becomes visible only after the first commit.
- Distributed Version Control: Allows multiple developers to work on a project simultaneously, track modifications, and revert changes if needed.



#Concept-11

Remote Branching in Git (Using Tuleap)

The fundamental concepts of Git remote branches are the same across all platforms (GitHub, GitLab, Bitbucket, Tuleap), but the interaction may vary slightly based on the platform's specifics, like how repositories are configured or accessed. In this case, I'll explain how you would interact with remote branches using Tuleap.

#Key Concepts

1. Remote Repository: A version-controlled repository hosted on a remote server (in your case, Tuleap).

2. Remote Branch: A reference to a branch in the remote repository.

3. Local Repository: Your local version of the repository on your machine.

4. Git Fetch: Retrieves new changes from the remote repository without merging them into your local branch.

5. Git Pull: Fetches changes from the remote repository and automatically merges them into your local branch.

#Step-by-Step Workflow for Remote Branching in Git with Tuleap

1. Clone a Repository from Tuleap

- git clone https://tuleap.example.com/your_project.git
- git remote
- origin

2. Create and Checkout a Local Branch

- git checkout -b feature-branch

3. Make Changes to the Files

- echo "New changes in the feature branch" > feature_file.txt

4. Add and Commit Changes Locally

- git add feature_file.txt
- git commit -m "Add new feature changes"
- git status
- git log --oneline --graph –all

5. Push Changes to Tuleap Remote Repository

- git push origin feature-branch

6. Fetch Remote Changes

While working locally, someone else might have pushed changes to the remote repository. To fetch those changes without merging them into your local branch, use the git fetch command.

- git fetch origin
- git branch -a

7. Merge Changes from Remote to Local

- git merge origin/master

8. Pull Changes Directly from Remote (Alternative to Fetch + Merge)

- git pull origin master

#Note

Instead of running git fetch and then manually merging the changes, you can use git pull to directly fetch and merge changes from the remote branch to your local branch.

9. Merging Remote Changes

- git checkout master
- git merge feature-branch
- git push origin master
- git branch -d feature-branch
- git push origin --delete feature-branch #Delete the remote branch

#Diff git fetch and git pull

git fetch:

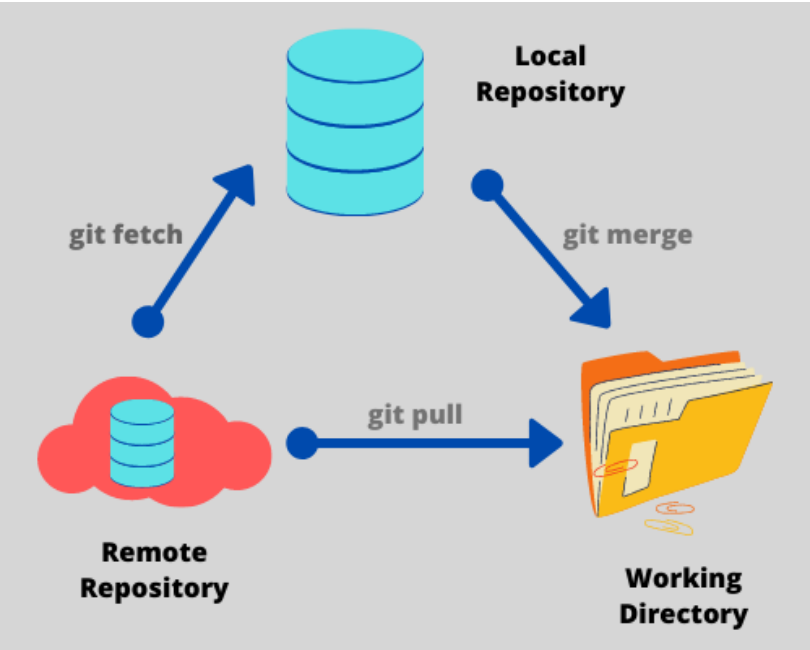
Downloads changes from remote but does not modify your working directory or merge them.

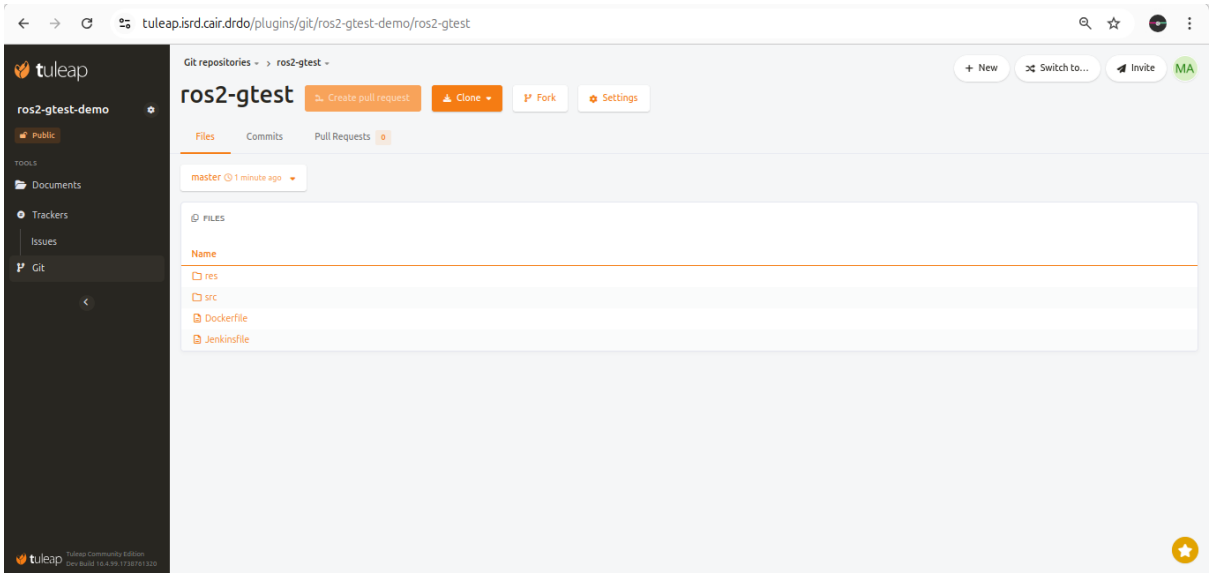
Gives you more control over how you handle changes.

git pull:

Downloads and automatically merges changes from the remote into your local branch.

A quick way to sync your local branch with the remote.




tuleap.isrd.cair.drdo/plugins/git/ros2-gtest-demo/ros2-gtest
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
Git repositories
ros2-gtest
ros2-gtest
Create pull request
Clone
Fork
Settings
New
Switch to...
Invite
MA
Files
Commits
Pull Requests
0
master
1 minute ago
FILES
Name
res
src
Dockerfile
Jenkinsfile

#Concept-12

Git Alias

What is a Git Alias?

A Git alias is a custom shortcut for a Git command. Instead of typing out long Git commands repeatedly, you can define shorter, easier-to-remember commands. This is especially useful for long commands that you often use in your day-to-day Git workflow.

#Why Create Git Aliases?

1. Save Time: Shorten frequently used Git commands.

2. Increase Productivity: Avoid repetitive typing.

3. Custom Workflow: Tailor commands to fit your needs.

#Workflow

- mkdir git-alias-example
- cd git-alias-example
- git init
- touch file1.txt file2.txt
- git add.
- git commit -m "Initial commit with file1.txt and file2.txt"

Create a Simple Alias

- git config --global alias.co checkout
- git co -b feature-branch

#Step 5: Viewing Your Existing Aliases

- git config --get-regexp alias

#Remove an Alias

- git config --global --unset alias.co

GIT alias shortcuts in terminal

```
alias gs='git status'
alias ga='git add'
alias gp='git push'
alias  c='git commit'
```

#Concept-13

#Git Rebasing

What is Git Rebase?

git rebase is a way to reapply commits on top of another base commit. It helps in maintaining a clean and linear history by moving or combining commits.

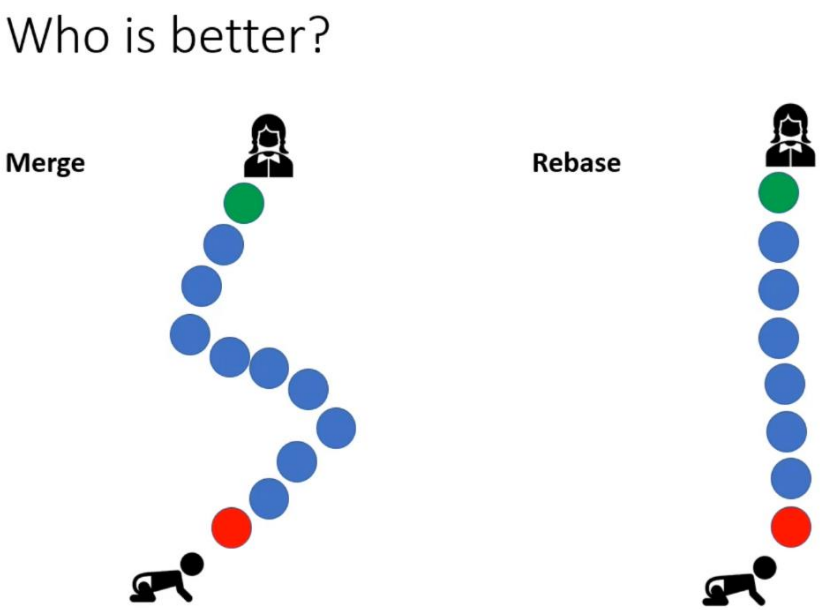
1 Why Use Git Rebase?

1. Keeps commit history clean and linear.

2. Avoids unnecessary merge commits.

3. Makes the history easier to read.

#Git rebase Architecture



2. How Git Rebase Works?

1. Takes your branch's commits and temporarily stores them.

2. Moves your branch's base to a new commit (e.g., latest master).

3. Applies your stored commits on top of the new base.

3. Git Rebase vs Git Merge

Feature	git rebase	git merge
History	Linear, clean	Includes merge commits
Commit Order	Rewrites History	Preserves all commits
Usage	Good for private branches	Preferred for shared branches

4. Git Rebase Example – Step by Step

You have a feature branch feature-branch, and master has new commits. You want to update your branch with the latest master changes without creating a merge commit.

Step 1: Create and switch to a new feature branch

- git checkout -b feature-branch
- echo "Feature Work" > feature.txt
- git add feature.txt
- git commit -m "Added feature work"

Step 2: Meanwhile, master gets updated

- git checkout master
- echo "Master Update" > master.txt
- git add master.txt
- git commit -m "Updated master branch"

Step 3: Rebase feature-branch onto master

- git checkout feature-branch
- git rebase master

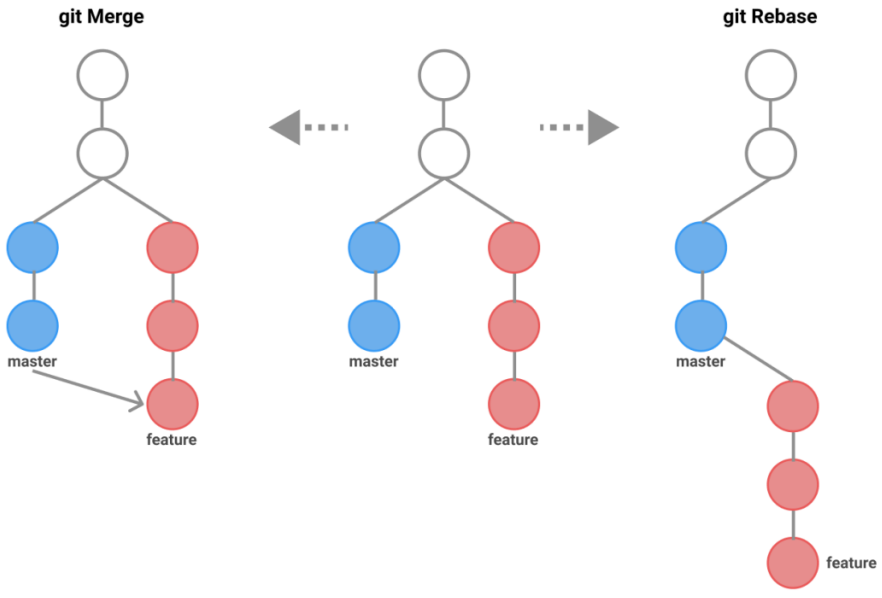
4.Handling Rebase Conflicts

- git add feature.txt
- git rebase --continue
- git rebase --abort

5. When to Use Git Rebase?

✅ When working on feature branches before merging into master.

✅ When keeping history clean (e.g., before pushing to remote).



#Git Rebase (online)—process

✅ Git rebase creates a linear commit history.

✅ Rebase rewrites commit history by applying commits from one branch onto another as if they were made in sequence.

✅ To see the difference after merging or rebasing, use:

git log --oneline --graph --all

✅ To inspect individual commit

git show <commit-id>

✅ Developers need to understand both merge and rebase to choose the best strategy.

#Task 1: Merge (Preserves History)

Scenario 1: Merging Same File (Conflict Occurs)

1. Initialize Git and create a project

- git init
- echo "hello" > f1.txt
- git add f1.txt
- git commit -m "Initial commit"
- git log --oneline --graph –all

2. Create a new branch (mahesh) and modify f1.txt

- git checkout -b mahesh
- echo "hello welcome" >> f1.txt
- git add f1.txt
- git commit -m "Added welcome message"
- git log --oneline --graph --all

3. Switch back to master and modify the same file (f1.txt)

- git checkout master
- echo "hi" >> f1.txt
- git add f1.txt
- git commit -m "Added hi message"
- git log --oneline --graph --all

4. Merge mahesh branch into master

- git merge mahesh

.....

🚨 Conflict occurs:

Auto-merging f1.txt

CONFLICT (content): Merge conflict in f1.txt

Automatic merge failed; fix conflicts and then commit the result.

.....

5. Resolve conflict in f1.txt (edit manually)

<<<<<<< HEAD

hello

hi

=======

hello welcome

to

>>>>>>> mahesh

✅ Final version after resolving conflict:

hello

hi

hello welcome

to

6. Mark conflict as resolved and complete the merge

- git add f1.txt
- git commit -m "Resolved merge conflict in f1.txt"
- git log --oneline --graph –all

#Scenario 2: Merging Different Files (No Conflict, Safe Merge)

1. Create a new file in mahesh branch

- git checkout -b mahesh
- echo "new file content" > f2.txt
- git add f2.txt
- git commit -m "Added f2.txt"

2. Switch to master and create a different file

- git checkout master
- echo "another file" > f3.txt
- git add f3.txt
- git commit -m "Added f3.txt"

3. Merge mahesh branch into master

- git merge mahesh

✅ No conflicts occur since different files were modified.

✅ Check commit history:

- git log --oneline --graph –all

#Task 2: Rebase (Linear History)

Scenario 1: Rebasing Same File (Conflict Occurs)

1. Modify the same file in both master and mahesh branches

- git checkout -b mahesh
- echo "hello welcome" >> f1.txt
- git add f1.txt
- git commit -m "Added welcome message"

2. Switch to master, modify the same file, and commit

- git checkout master
- echo "hi" >> f1.txt
- git add f1.txt
- git commit -m "Added hi message"

3. Rebase mahesh onto master

- git rebase mahesh

.....

🔒 Conflict occurs:

CONFLICT (content): Merge conflict in f1.txt

.....

4. Resolve the conflict in f1.txt (edit manually)

hello

hi

hello welcome

to

5. Mark conflict as resolved and continue rebase

- git add f1.txt
- git rebase --continue
- git log --oneline --graph –all

#Scenario 2: Rebasing Different Files (No Conflict, Safe Rebase)

1. Create a new file in mahesh

- git checkout -b mahesh
- echo "new file content" > f2.txt
- git add f2.txt
- git commit -m "Added f2.txt"

2. Switch to master and create a different file

- git checkout master
- echo "another file" > f3.txt
- git add f3.txt
- git commit -m "Added f3.txt"

3. Rebase mahesh onto master

- git rebase mahesh

✅ No conflicts occur since different files were modified.

✅ Check commit history:

- git log --oneline --graph –all

#Final Notes

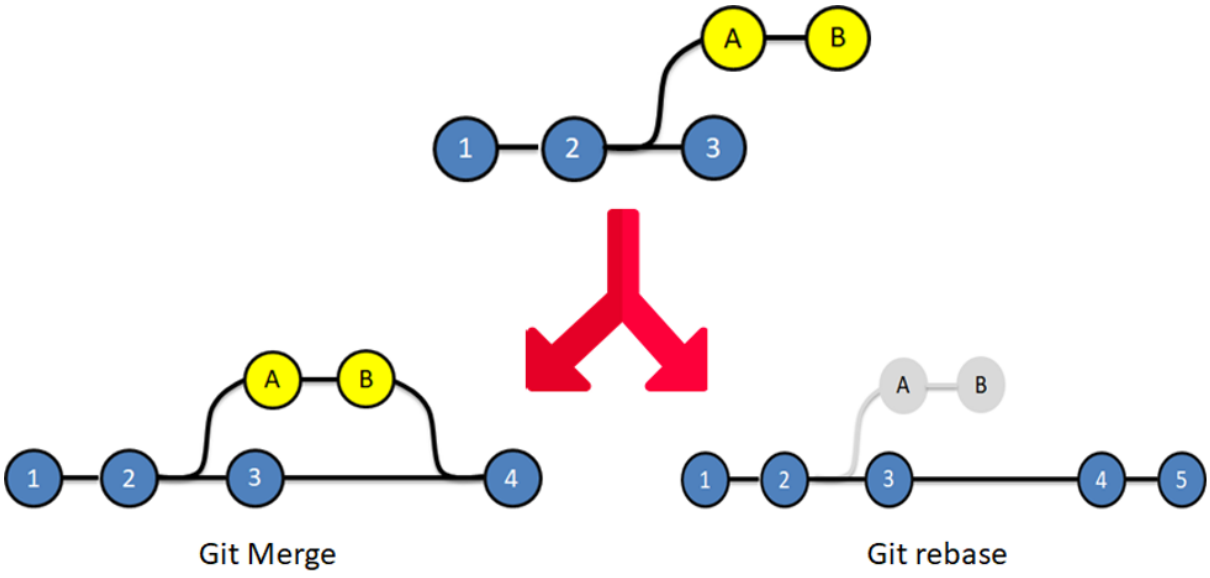
✅ Both merge and rebase combine changes, but they handle commit history differently.

✅ Merge creates an additional commit to combine histories.

✅ Rebase moves commits one by one to keep history linear.

✅ Use git log --oneline --graph --all to see the difference visually.

✅ Use merge in shared branches, and rebase in local feature branches.



#Concept-14

Git Stash--14

✅ Git Stash allows you to temporarily save uncommitted changes without committing them.

✅ It helps when you need to switch branches or pull updates but don't want to commit unfinished work.

✅ Stashed changes can be reapplied later using git stash pop or git stash apply.

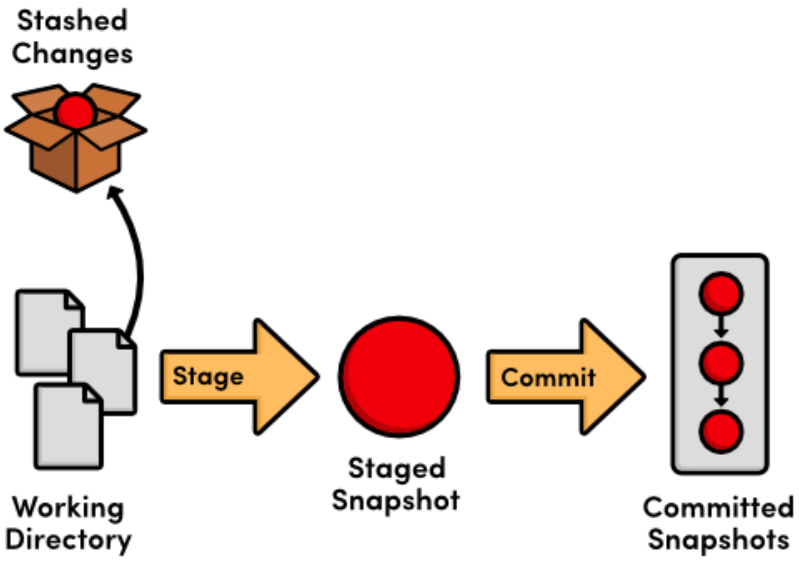
Why Use Git Stash?

🔷 Suppose you're working on a feature, but suddenly need to:

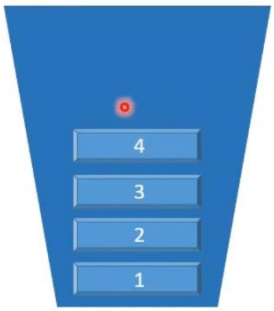
- Switch to another branch to fix a bug
- Pull the latest updates from the remote repository
- Keep your working directory clean without committing incomplete work
- Instead of committing unfinished changes, you can stash them and retrieve them later.

Recycle bin

- Delete(backup)
- Shift+delete(permanent)



Dustbin(stack)



#Basic Git Stash Commands

Command	Description
git stash	Stashes only tracked changes (ignores untracked files)
git stash -u or git stash --include-untracked	Stashes both tracked and untracked files
git stash list	Shows all stashed entries
git stash pop	Applies the most recent stash and removes it from the stash list
git stash apply	Applies the most recent stash but keeps it in the stash list
git stash drop	Deletes the most recent stash
git stash clear	Removes all stashed changes

#Example 1: Stashing and Retrieving Changes

1. Create a Git Repository

- git init my_project
- cd my_project
- echo "Initial version" > file.txt
- git add file.txt
- git commit -m "Initial commit"

2. Modify a File (But Don’t Commit)

- echo "Unfinished work" >> file.txt
- git status

◈ This will show modified changes in file.txt but not staged or committed.

3. Stash the Changes

- git stash

◈ Now, git status will show a clean working directory.

4. Switch to Another Branch and Do Some Work

- git checkout -b hotfix
- echo "Bug fix applied" > fix.txt
- git add fix.txt
- git commit -m "Hotfix applied"

5. Switch Back and Restore Stashed Changes

- git checkout master
- git stash pop

◈ This will restore file.txt changes and remove the stash from the list.

#Example 2: Stashing Untracked Files

If you have new files that are not yet added to Git, use:

- git stash -u

◈ This will stash both tracked and untracked files.

- git stash apply

#Example 3: Stashing Multiple Changes

1. Stash Multiple Changes

- git stash

Modify some files again

- echo "More work" >> file.txt
- git stash

2. View Stash List

- git stash list
- git stash apply stash@{1}
- git stash drop stash@{1}
- git stash clear #Clearing All Stashed Changes

#Git Stash vs. Commit

Feature	Git Stash	Git Commit
Saves changes temporarily?	✅ Yes	❌ No (Permanent)
Shows up in commit history?	❌ No	✅ Yes
Helps when switching branches?	✅ Yes	✅ Yes
Requires a commit message?	❌ No	✅ Yes

#When to Use Git Stash?

✅ When you want to temporarily save work without committing

✅ When switching branches but you don't want to commit unfinished work

✅ When pulling changes from remote without losing local modifications

#Git Stash (online-process)

✅ Git Stash acts like a temporary "recycle bin" 🗑️ for your uncommitted changes. It helps when you need to switch branches or pull updates without committing unfinished work.

🔷 Git Stashing Process - Step by Step

1. Initialize a Git repository

- git init

2. Create and commit a file

- echo "file" >> f1.txt
- git add f1.txt
- git commit -m "Initial commit"

3. Modify the file but don't commit

- echo "new changes" >> f1.txt

4. Stash the changes (remove them from the working directory)

- git stash

5. Check the status (no changes should be visible now)

- git status
- cat f1.txt # The file is now back to its last committed state

🔷 Important Git Stash Commands

- git stash list
- git stash apply stash@{1} #📌 Apply a specific stash (without deleting it from the stash list):
- git stash drop stash@{1} 📌 Drop a specific stash (delete one stash entry):
- git stash pop 📌 Pop stash (restore changes & remove from stash list):
- git stash -u 📌 Stash untracked files as well
- git stash branch new-feature 📌 Create a new branch from a stashed state:

1. Modify an existing file and create a new file (f2.txt)

- echo "modifying f1" >> f1.txt
- echo "new file" >> f2.txt

2. Stash only tracked changes (f1.txt will be stashed, but f2.txt remains untracked)

- git stash

3. Stash both tracked & untracked files

- git stash -u

4. Restore stashed changes and remove them from the stash list

- git stash pop

#Note

✅ Stashing only affects tracked files unless -u is used.

✅ Git stash is temporary storage and can be dropped/deleted anytime.

✅ git stash pop restores changes & deletes them from stash history, whereas git stash apply restores changes but keeps them in the stash list.

#Concept-15

Git Clean (□ Git Clean - Remove Untracked Files)

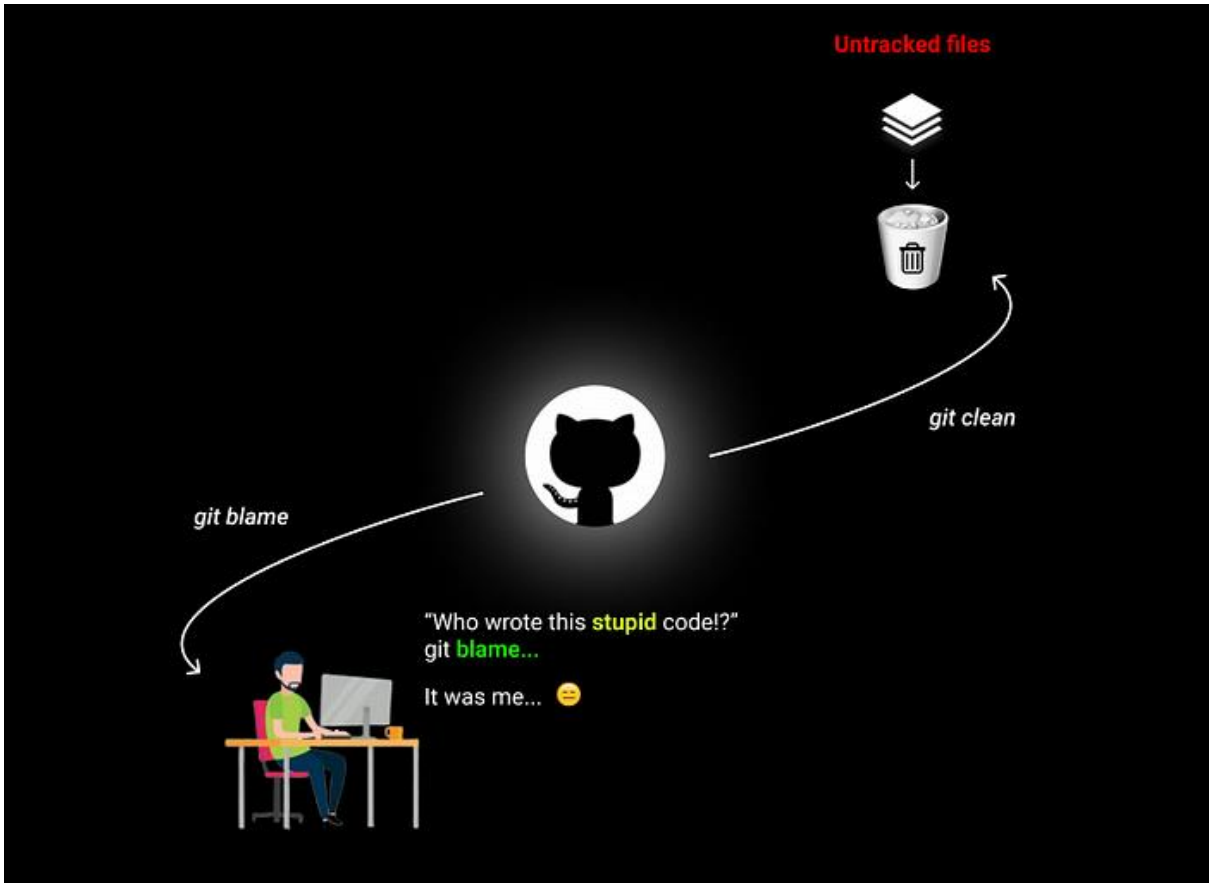
git clean is a convenience method for deleting untracked files in a repo's working directory. Untracked files are those that are in the repo's directory but have not yet been added to the repo's index with git add.

✅ git clean is used to remove untracked files and directories from your working directory.

✅ It helps clean up unnecessary files that are not tracked by Git, such as build artifacts, logs, or temporary files.

◈ When to Use git clean?

- When you want to delete untracked files (files not added to Git).
- To remove unnecessary build files before committing changes.
- After switching branches and wanting to reset your working directory.



#◈ Basic git clean Commands

1. Check what will be deleted (dry run):

   ➢ git clean -n

2. Delete untracked files:

- git clean -f

3. Delete untracked directories:

- git clean -d

4. Delete ignored files (like files in. gitignore):

- git clean -X

5. Delete both untracked & ignored files

- git clean -x

Main Differences Between `git blame` and `git clean`

Command	Purpose	What It Does	Common Use Case
`git blame`	Tracks changes to a file	Shows who last modified each line of a file and when	Checking who made changes to a specific line of code
`git clean`	Removes untracked files	Deletes untracked files and directories from your working directory	Cleaning up unnecessary or temporary files

◈ Git Clean - Step by Step Example

1. Initialize a Git repository

- git init

2. Create and commit a tracked file

- echo "Hello Git" >> tracked.txt
- git add tracked.txt
- git commit -m "Initial commit"

3. Create untracked files & folders

- echo "Untracked file" >> untracked.txt
- mkdir my_folder
- touch my_folder/temp.log

4. Check untracked files

- git status

5. See what git clean will remove (dry run)

- git clean -n

6. Delete untracked files & directories

- git clean -f -d

7. Verify the cleanup

- git status

🔷 git clean vs. git reset vs. git stash

Command	What It Does
git clean	Removes untracked files from working directory permanently.
git reset --hard	Resets tracked files to last commit and discards changes.
git stash	Saves uncommitted changes temporarily for later use.

#Note

✅ git clean permanently deletes files! Use -n first to check what will be removed.

✅ Use git clean -f cautiously, as deleted files cannot be recovered.

✅ If a file is tracked and you want to remove changes, use git reset instead.

#Git clean online process

✔️ git clean is used to remove untracked files from the working directory.

✔️ It does not affect tracked files or staged changes.

✔️ It is permanent—deleted files cannot be recovered unless backed up.

📌 Git Clean Commands & Explanation

- git init

2. Create a file and a directory (both untracked)

- nano f1.txt # Create a file
- mkdir my_dir && echo "Hello" > my_dir/file.txt # Create a directory with a file
- git status

#Forcefully clean untracked files only

- git clean -f # Removes only untracked files, not directories
- git clean -f -d # -d removes untracked directories as well
- git clean -f -d -x # -x removes ignored files too
- git clean -f -n # -n lists files that will be removed

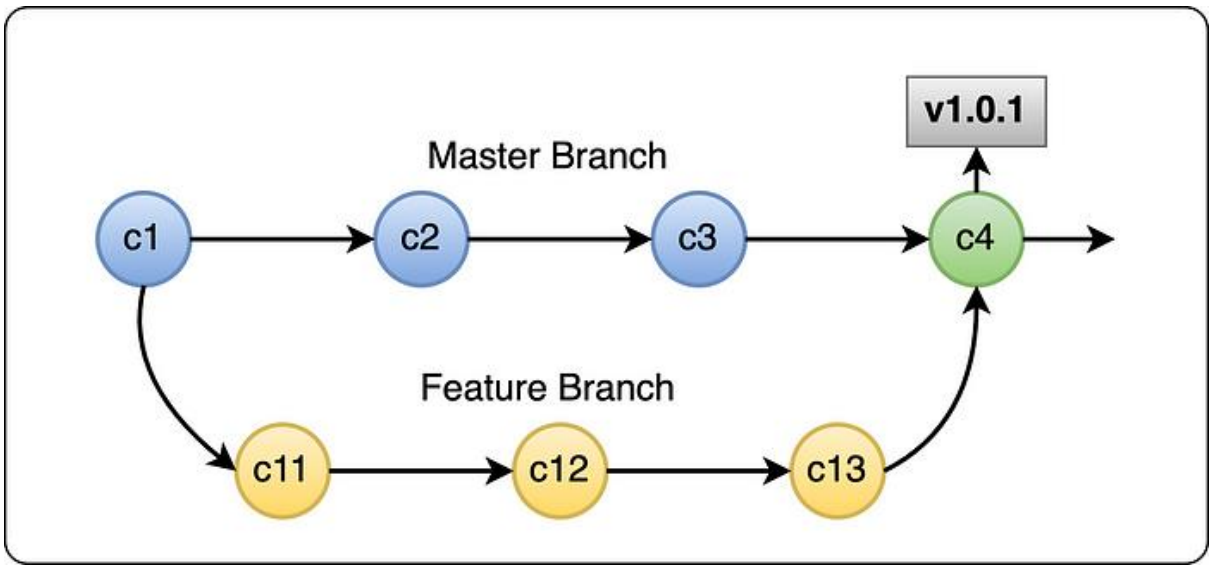
#Concept-16

📌 Git Tagging (git tag)

✅ What is Git Tagging?

Git tagging is used to mark specific points in a repository's history, such as releases (v1.0, v2.0) or important milestones.

Tags are like checkpoints that help developers track and reference specific commits.



✅ Types of Git Tags

Annotated Tags (Recommended) – Stores metadata like the author, date, and message.

Lightweight Tags – A simple reference to a commit (without metadata).

🚀 Git Tagging Commands & Examples

#1. Create a Git Repository

- git init

2. Add and Commit Files

- echo "Version 1" > file.txt
- git add file.txt
- git commit -m "Initial Commit"

🔷 Creating Tags

3. Create a Lightweight Tag (Simple label without metadata)

- git tag v1.0

4. Create an Annotated Tag (With details & message)

- git tag -a v1.0 -m "First stable release"

🔷 Viewing Tags

5. List all tags

- git tag

6. View details of an annotated tag

- git show v1.0

🔷 Working with Tags

7. Pushing Tags to Remote Repository

- git push origin v1.0 # Push a specific tag
- git push origin --tags # Push all tags

8. Checking Out a Tag (Read-only state)

- git checkout v1.0

9. Creating a Branch from a Tag

- git checkout -b new-feature v1.0

🔷 Deleting Tags

🔟 Delete a Local Tag

- git tag -d v1.0

11. Delete a Remote Tag

- git push origin --delete v1.0

📌 When to Use Git Tags?

✅ To mark releases (e.g., v1.0, v2.0)

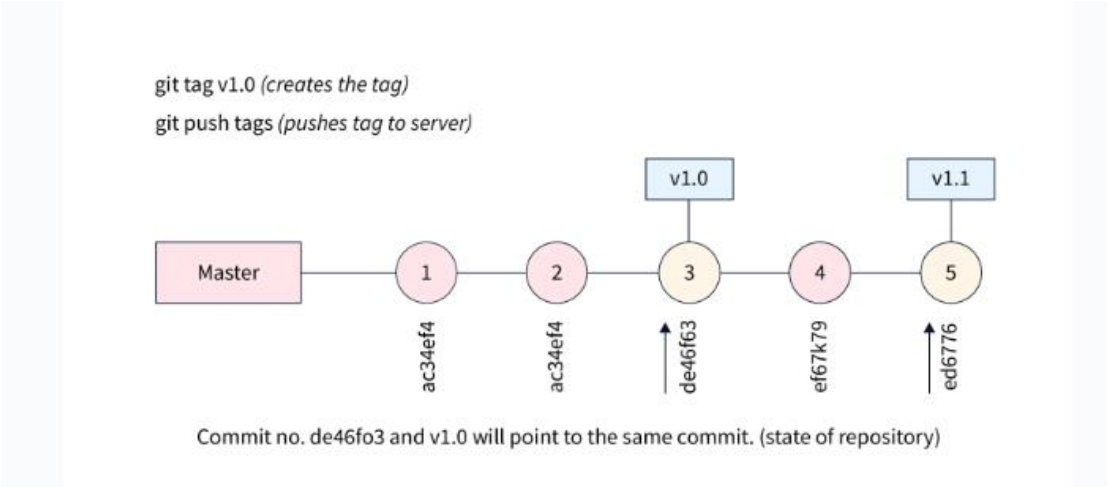
✅ To label important commits (e.g., "Bug Fix Release")

✅ To help track versions in production deployments

#GIT TAG(ONLINE)

- #git tagging
- git tag -l
- latest commit only tag will create
- git init

- nano f1
- git add and commit
- git log
- git tag v1.0
- git log (verify the tag)



#Concept-17

#Git Reset & Revert

🔸 What is git reset?

git reset moves the HEAD (branch pointer) backward to a previous commit, removing commits from the history.

- Soft Reset (--soft) → Keeps changes staged (in the index).
- Mixed Reset (--mixed) (Default) → Unstages changes but keeps them in the working directory.
- Hard Reset (--hard) → Deletes changes permanently.

Example for git reset

- git init
- echo "First commit" > file.txt
- git add file.txt
- git commit -m "First commit"

- echo "Second commit" >> file.txt
- git add file.txt
- git commit -m "Second commit"
- git log --oneline # Check commit history

👉 If you want to undo the last commit but keep the changes staged

- git reset --soft HEAD~1

👉 If you want to undo the last commit and unstage the changes

- git reset --mixed HEAD~1

👉 If you want to undo the last commit and delete the changes completely

- git reset --hard HEAD~1

What is git revert?

git revert creates a new commit that undoes the changes of a previous commit without removing history.

✅ Example for git revert

git revert HEAD # Reverts the last commit by creating a new commit

👉 If you want to revert a specific commit

- ➢ git revert <commit-id>

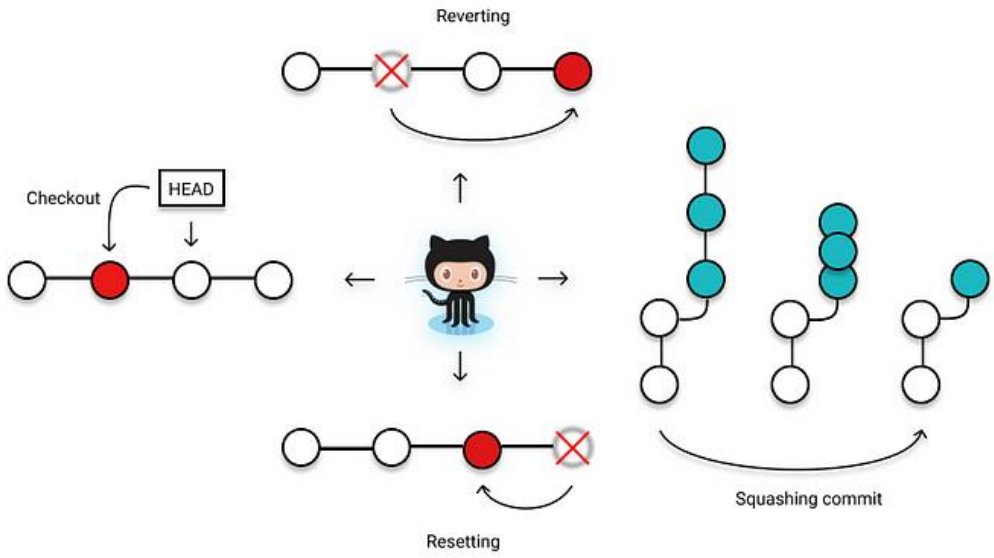
Difference Between git reset & git revert

Feature	git reset	git revert
Removes commits?	Yes	No
Keeps history?	No	Yes
Creates a new commit?	No	Yes
Can be used in shared repositories	❌ No (dangerous)	✅ Yes (safe)

When to Use?

✅ Use git reset if you're working locally and want to undo commits without keeping history.

✅ Use git revert if you're working with others and need to preserve commit history while undoing changes.



#Concept-18

#Git Pull

◈ What is git pull?

git pull fetches the latest changes from a remote repository and merges them into your local branch.

git fetch + git merge

This means:

git fetch → Downloads changes from the remote repository.

git merge → Merges those changes into your current local branch.

#✅ Example: Using git pull

Step 1: Clone a Repository

- git clone https://github.com/user/repo.git
- cd repo

#Step 2: Check for Remote Changes

- git status # Check current status
- git branch # See which branch you are on
- git remote -v # Verify the remote repository

#Step 3: Pull Changes from Remote

- git pull origin main # Pull latest changes from 'main' branch

#◈ Types of git pull

1. Fast-Forward Merge (git pull with no local changes)

If your local branch has no new commits, Git moves the branch pointer forward without creating a merge commit.

- git pull origin main

#output

Updating 123abc..456def

Fast-forward

2. Merge Commit (git pull when local and remote branches have diverged)

If you have local commits that conflict with remote changes, Git will create a merge commit.

- git pull origin main

#output

Auto-merging file.txt

CONFLICT (content): Merge conflict in file.txt

Automatic merge failed; fix conflicts and then commit the result.

#Fixing the Conflict

1. Open the conflicted file.

2. Resolve the conflicts manually.

3. Add the resolved file:

- git add file.txt
- git commit -m "Resolved merge conflict"
- git push origin main

Difference Between git pull & git fetch

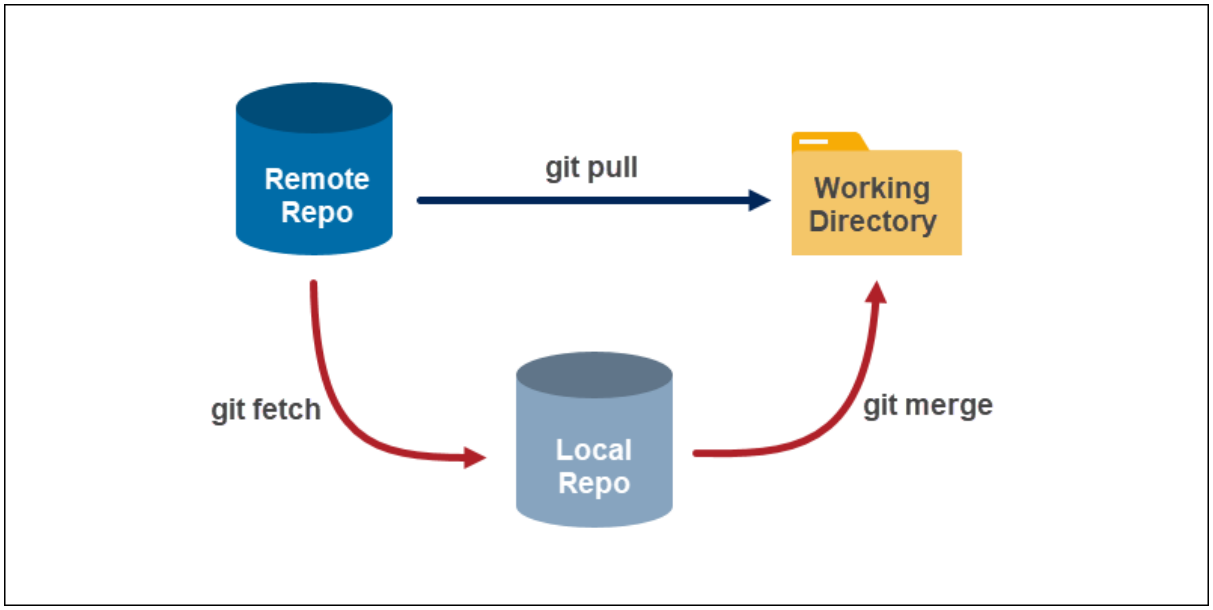
Command	Action
git fetch	Downloads changes from the remote repository but does not merge them.
git pull	Downloads changes and merges them into the current branch.

When to Use?

✅ Use git pull when you want to fetch and merge changes automatically.

✅ Use git pull --rebase if you want to avoid unnecessary merge commits.

✅ Use git fetch first if you want to check for changes before merging.



Detailed Explanation with Examples

1. git clone – Copy a Repository

Creates a full local copy of a remote repository.

- git clone https://github.com/user/repo.git

What happens?

- Downloads all commits, branches, and history from repo.git.
- Creates a new folder named repo in your current directory.

📌 **Use Case:** First time setting up a project on your machine.

2. git fetch – Get Remote Changes (But Don't Merge)

Checks for updates without modifying your working directory.

- git fetch origin

What happens?

- Downloads new commits and branches from origin (remote repository).
- **Does NOT update** your local branch or working files.

🔍 **Check fetched commits**

- git log origin/main –oneline

📌 **Use Case:** When you want to check for new changes **without modifying your work**.

3. git pull – Fetch + Merge Remote Changes

Brings the latest changes into your local branch.

- git pull origin main

What happens?

- First, git fetch gets the latest commits.
- Then, git merge integrates those commits into main.

📌 **Use Case:** When you want to update your branch with the latest remote changes.

- ❖ **git clone** when setting up a new repo.
- ❖ **git fetch** to check for updates without merging.
- ❖ **git pull** to update your local branch with remote changes.

Difference Between `git pull`, `git clone`, and `git fetch`

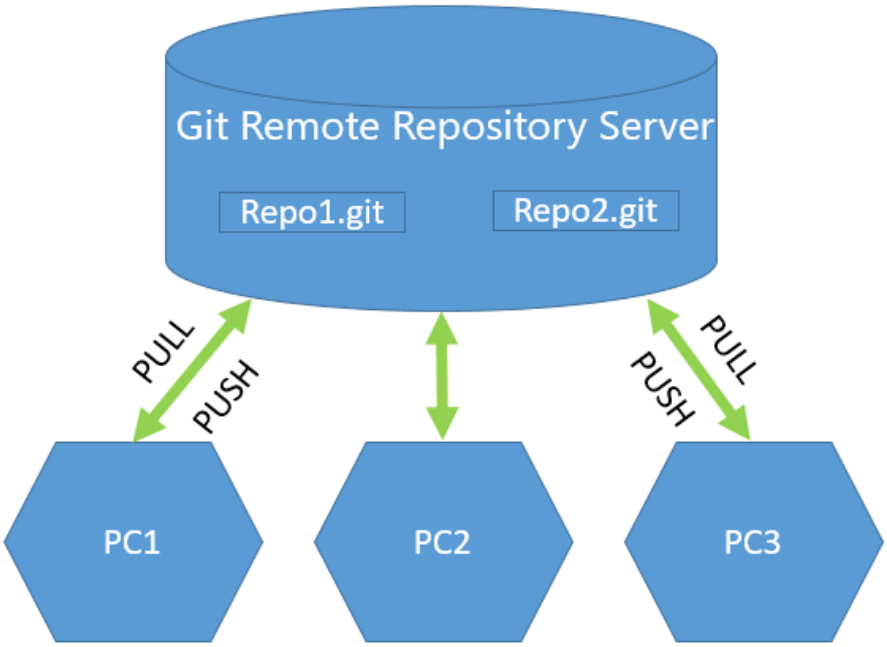
Command	Purpose	What It Does	When to Use
`git clone`	Copy a remote repository	Downloads the entire repository, including history, and sets up a local version	When you need to create a local copy of a repository for the first time
`git fetch`	Check for updates from the remote	Downloads new commits and branches but **does not merge** them into your working directory	When you want to see the latest changes before integrating them
`git pull`	Fetch and merge updates from the remote	Combines `git fetch` + `git merge`, bringing remote changes into your current branch	When you want to update your local branch with the latest remote changes

##Concept-20

#Remote Servers

Remote Servers & Tuleap in Detail

Now that Git concepts are complete, let's move to Remote Servers like Tuleap and how they integrate with Git for version control, collaboration, and project management.



#🔷 What is a Remote Server?

A remote server is a central system where Git repositories are hosted. It allows multiple developers to collaborate by pushing, pulling, and managing code.

Examples of remote servers include: ✅ GitHub – Popular for open-source projects

✅ GitLab – Provides built-in CI/CD

✅ Bitbucket – Focused on teams using Jira

✅ Tuleap – Open-source ALM (Application Lifecycle Management)

1. DevOps 🚀

DevOps is a software development practice that combines development (Dev) and operations (Ops) teams to automate and streamline the software delivery lifecycle. It focuses on continuous integration, continuous delivery (CI/CD), automation, and monitoring to improve efficiency and reduce deployment failures.

2. Agile 🏗️

Agile is a software development methodology that emphasizes iterative progress, collaboration, and adaptability. It breaks projects into small, manageable sprints with frequent feedback from stakeholders, ensuring continuous improvement. Agile includes Scrum, Kanban, SAFe, and XP as popular frameworks.

3. Tuleap 📌

Tuleap is an open-source Agile & DevOps tool for managing software development projects. It supports Scrum, Kanban, SAFe, and integrates with Git, Jenkins, and CI/CD pipelines for a complete Agile-DevOps workflow. It helps teams plan, track, and deliver software efficiently while maintaining quality and compliance.

#Note

- Agile is a methodology, while Tuleap is a tool that helps teams implement Agile efficiently.
- Tuleap combines Agile & DevOps, making it an all-in-one solution for project management, Git versioning, and CI/CD.
- If you're looking for a self-hosted, open-source Agile tool, Tuleap is a great alternative to Jira or Trello.

#🔷 What is Tuleap?

Tuleap is an open-source ALM (Application Lifecycle Management) platform that integrates Git with advanced project management features.

It provides issue tracking, document management, continuous integration, and agile tools in a single platform.

✅ Self-hosted or Cloud-based

✅ Supports Git & SVN

✅ Integrated CI/CD

✅ Agile & DevOps ready

✅ Fine-grained access control

#🚀	Tuleap	Other Remote Servers
Feature	Tuleap	GitHub/GitLab/Bitbucket
Open-Source	Yes	❌ GitHub, Bitbucket (Paid)
Git Support	✅ Yes	✅ Yes
CI/CD	✅ Integrated	✅ GitHub Actions, GitLab CI
Agile Tools	✅ Built-in	❌ Limited (Needs plugins)

Self-Hosting	✅ Yes	✅ GitLab (Paid for full features)

🔷 How Git Works with Tuleap

Step-by-Step Workflow

1. Create a Tuleap Project

- Login to Tuleap
- Create a new project
- Enable Git as a source control tool

2. Clone the Repository Locally

- git clone https://tuleap.example.com/your_project.git

3. Work on Your Code

- git checkout -b feature-branch
- echo "New feature" > feature.txt
- git add feature.txt
- git commit -m "Added new feature"

4. Push Changes to Tuleap

- git push origin feature-branch

5. Review & Merge in Tuleap

- Use Tuleap's code review tool
- Merge after approval

🔷 Tuleap Git Best Practices

✅ Use feature branches for development

✅ Enable code reviews before merging

✅ Integrate with CI/CD for automated builds

✅ Manage project tasks using Tuleap's agile tools

✅ Use access controls to define permissions

🔷 Tuleap Integrations

✅ Jenkins – Continuous Integration

✅ Mattermost/Slack – Team Collaboration

✅ Docker/Kubernetes – DevOps & Deployment

✅ SVN – Supports mixed Git + SVN environments

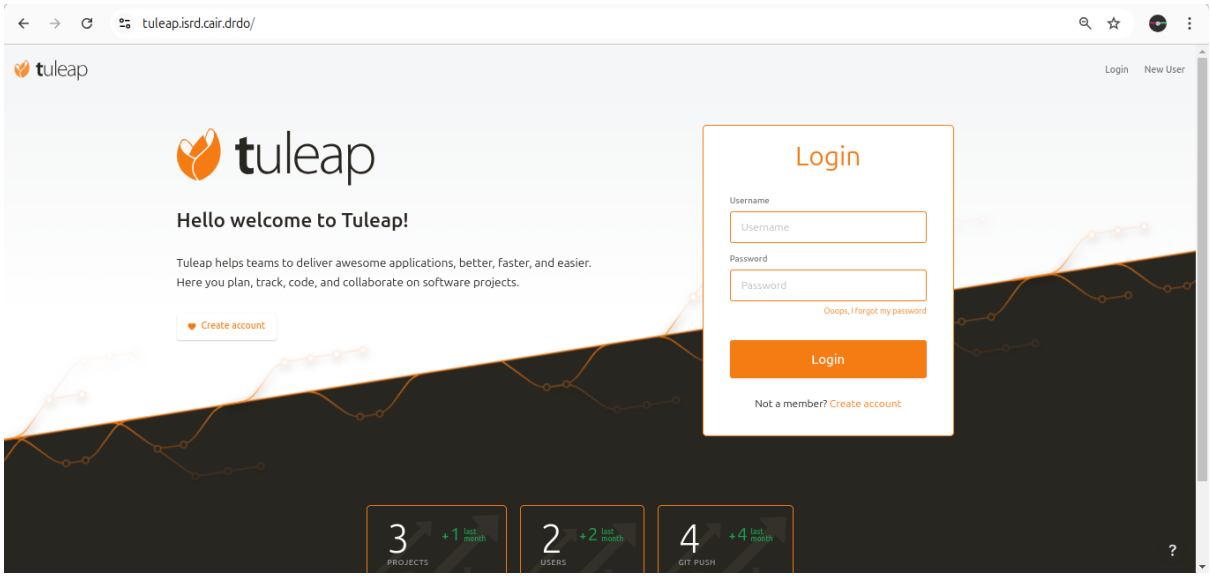
📌 Why Choose Tuleap?

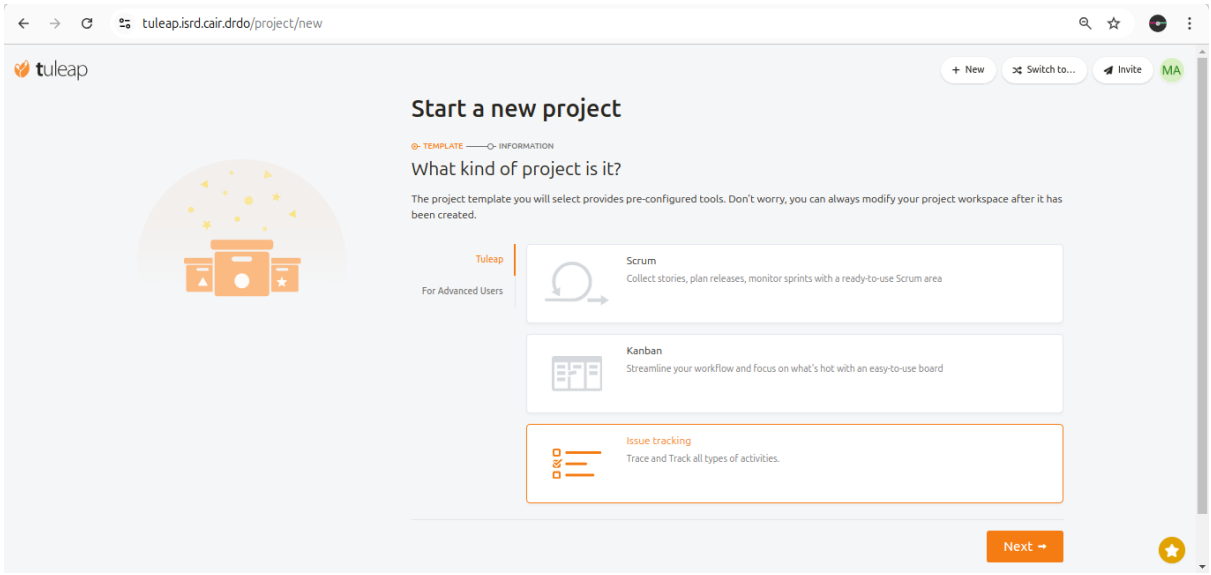
✅ Open-source alternative to GitHub/GitLab

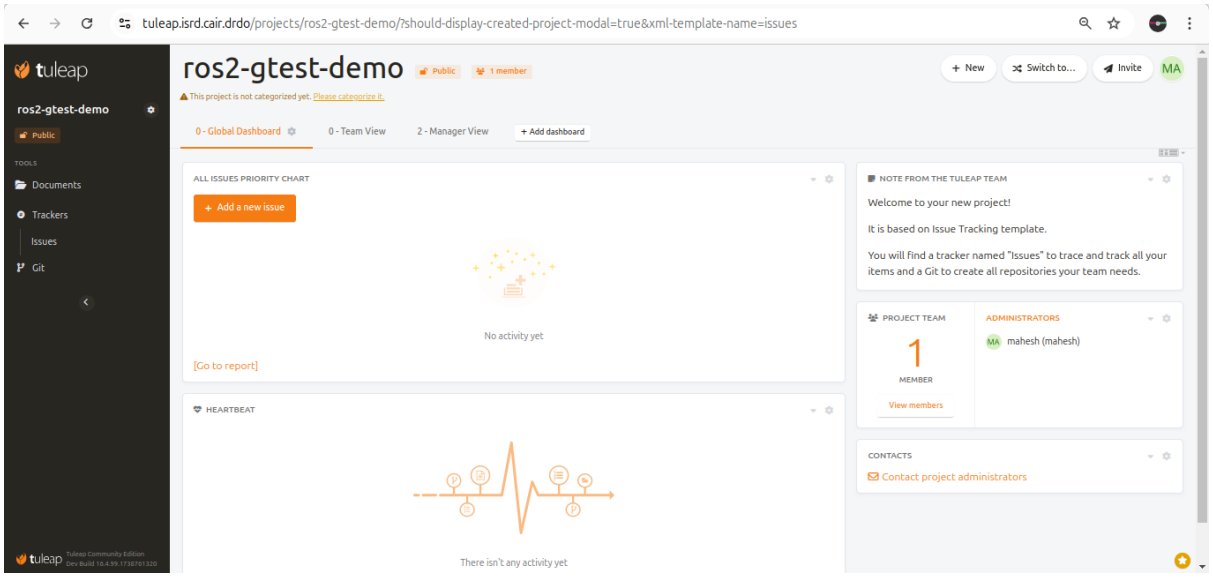
✅ Complete ALM solution (Agile, Git, CI/CD, Issue tracking)

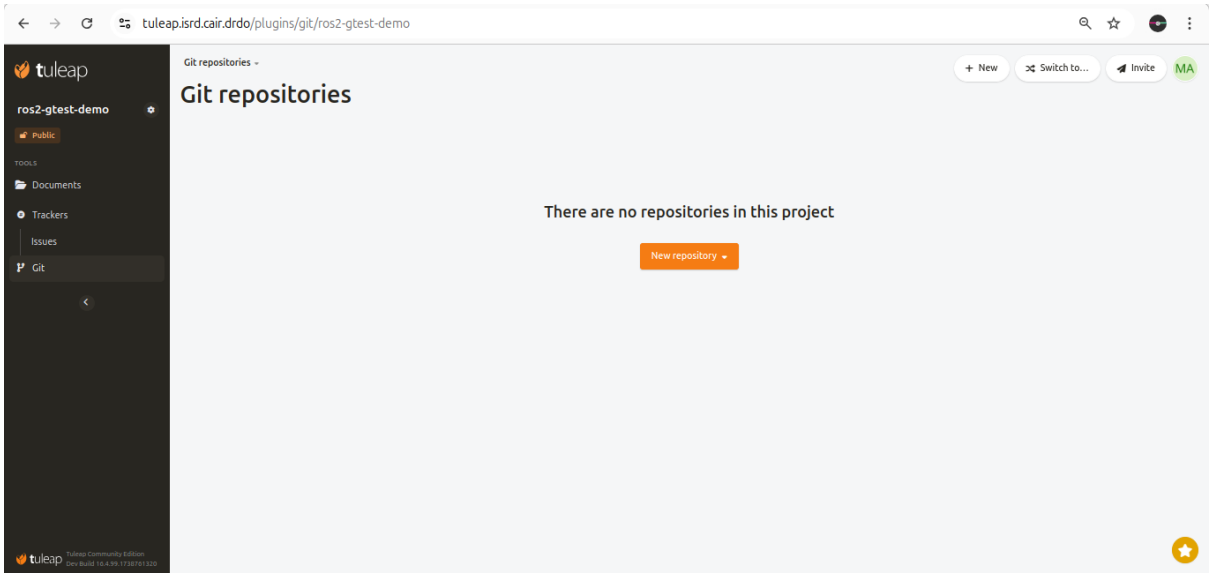
✅ Better control for enterprises & government projects

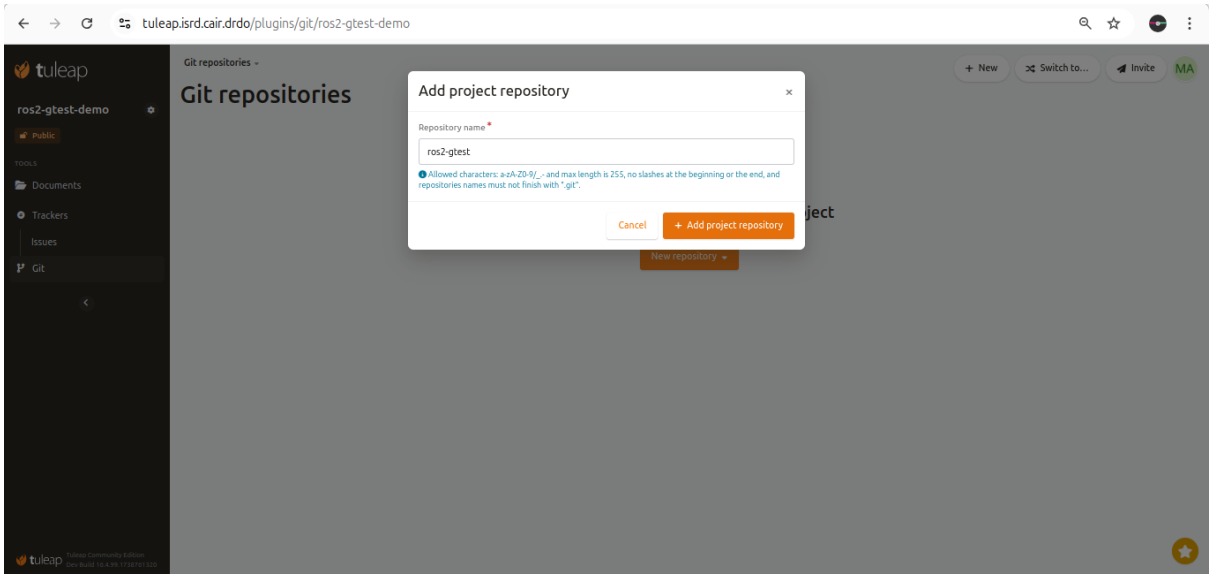
✅ Self-hosted option for full data privacy

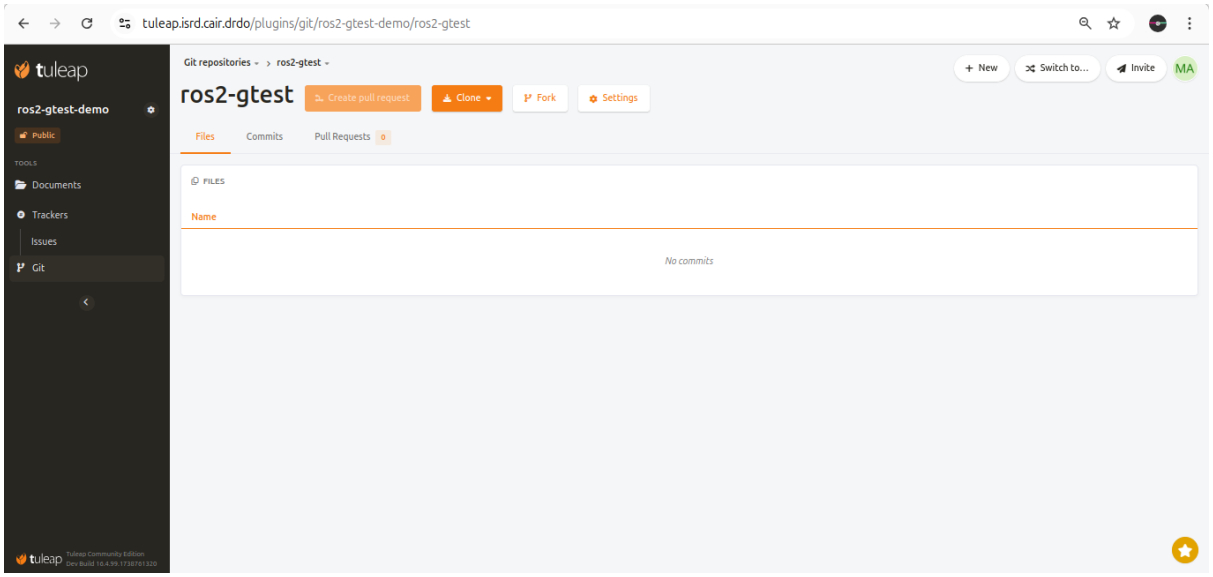


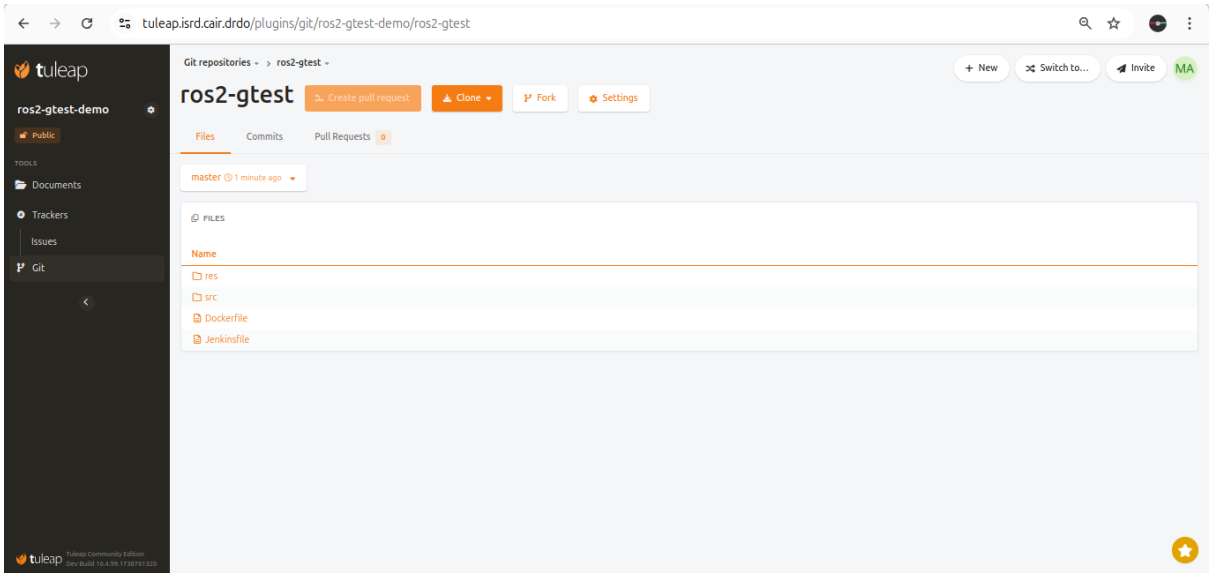



tuleap.isrd.cair.drdo/projects/ros2-gtest-demo/?should-display-created-project-modal=true&xml-template-name=issues
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
ros2-gtest-demo
Public
1 member
This project is not categorized yet. Please categorize it.
New
Switch to...
Invite
MA
0 - Global Dashboard
0 - Team View
2 - Manager View
+ Add dashboard
ALL ISSUES PRIORITY CHART
+ Add a new issue
No activity yet
[Go to report]
NOTE FROM THE TULEAP TEAM
Welcome to your new project!
It is based on Issue Tracking template.
You will find a tracker named "Issues" to trace and track all your items and a Git to create all repositories your team needs.
PROJECT TEAM
1
MEMBER
View members
ADMINISTRATORS
mahesh (mahesh)
HEARTBEAT
There isn't any activity yet
CONTACTS
Contact project administrators


tuleap.isrd.cair.drdo/plugins/git/ros2-gtest-demo
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
Git repositories
New
Switch to...
Invite
MA
Git repositories
There are no repositories in this project
New repository


tuleap.isrd.cair.drdo/plugins/git/ros2-gtest-demo
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
Git repositories
New
Switch to...
Invite
MA
Git repositories
Add project repository
Repository name *
ros2-gtest
Allowed characters: a-zA-Z0-9/_.- and max length is 255, no slashes at the beginning or the end, and repositories names must not finish with ".git".
Cancel
+ Add project repository
New repository


tuleap.isrd.cair.drdo/plugins/git/ros2-gtest-demo/ros2-gtest
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
Git repositories
ros2-gtest
New
Switch to...
Invite
MA
ros2-gtest
Create pull request
Clone
Fork
Settings
Files
Commits
Pull Requests 0
FILES
Name
No commits


tuleap.isrd.cair.drdo/plugins/git/ros2-gtest-demo/ros2-gtest
tuleap
ros2-gtest-demo
Public
TOOLS
Documents
Trackers
Issues
Git
Git repositories
ros2-gtest
New
Switch to...
Invite
MA
ros2-gtest
Create pull request
Clone
Fork
Settings
Files
Commits
Pull Requests 0
master 1 minute ago
FILES
Name
res
src
Dockerfile
Jenkinsfile

Git Main Differences

Here's a structured comparison for **#Concept-1** based on your request:

Feature	Centralized VCS (CVCS)	Distributed VCS (DVCS)
Repository	Central server only	Local copy for each developer
Offline Work	No (requires server connection)	Yes (work offline)
Data Redundancy	Only on central server	Each developer has a full copy
Speed	Slower (requires server access)	Faster (local operations)
Collaboration	Depends on central server	More flexible (works locally, then syncs)
Single Point of Failure	Yes (if the server fails, work stops)	No (each developer has a full copy)
Examples	SVN, Perforce	Git, Mercurial, Bazaar

##Concept-2

Differences in Installing Git on Ubuntu vs. RHEL/CentOS

Step	Ubuntu 🐧	RHEL/CentOS 🖥️
Update System	`sudo apt update && sudo apt upgrade -y`	`sudo yum update -y`
Install Git	`sudo apt install git -y`	`sudo yum install git -y`

#Concept-3

#Differences in Basic Git Workflow Commands

Step	Command
Initialize Repository	git init
Create & Modify Files	echo "Hello, Git!" > file.txt
Add to Staging	git add file.txt
Commit Changes	git commit -m "Initial commit"
Connect to Remote Repo	git remote add origin <repo_url>
Push Changes	git push -u origin main
Pull Remote Changes	git pull origin main
Clone a Repository	git clone <repo_url>

#Concept-4

Differences in Git Architecture Components & Commands

Component	Description	Key Commands
Working Directory	Where files are created, modified, and deleted	`git status`, `git diff`, `rm`, `mv`, `cp`
Staging Area	Holds changes before committing to local repo	`git add`, `git reset`, `git restore --staged`
Local Repository	Stores committed changes & full project history	`git commit`, `git log`, `git reset`, `git revert`
Remote Repository	Centralized repo for collaboration (GitHub, GitLab, etc.)	`git push`, `git pull`, `git fetch`, `git clone`

#Concept-5

Differences in Git Status Life Cycle Stages & Commands

File State	Description	Example Command
Untracked	New file not yet added to Git	`git add file.txt`
Modified	Tracked file with unsaved changes	`git status`, `git diff`
Staged	File added to staging, ready to commit	`git add file.txt`
Committed	File saved in local repository	`git commit -m "Message"`
Pushed	Committed changes sent to remote repo	`git push origin main`
Unstaged	Changes made after commit	`git add file.txt` (re-stage)

#Concept-6

Differences in Git Ignore File Usage

Feature	Example 1	Example 2
Files Ignored	`*.txt` (all `.txt` files)	`*.log`, `.env`, `temp.txt`
Directories Ignored	`*/` (all directories)	`logs/*` (specific directory)
Purpose	Ignores all `.txt` files and directories globally	Ignores specific file types and sensitive files
Git Command Used	`git rm -r --cached .`	`git status --ignored`

#Concept-7

Feature	git status	git diff
Purpose	Shows file status (untracked, modified, staged)	Shows exact content changes in files
Scope	File-level overview	Line-by-line differences
Before Staging	Lists modified files	Shows changes between working directory & staging area
After Staging	Lists staged files	`git diff --staged` shows staged changes
Use Case	Checking what needs to be committed	Reviewing exact modifications

#Concept-8

Comparison: `git add` vs. `git commit`

Feature	git add	git commit
Purpose	Moves changes to the staging area	Saves staged changes to the local repository
Scope	Prepares specific files for commit	Records all staged changes as a snapshot
Required Before?	Yes, before committing	No, but requires staged files
Effect on Repo	No changes to repo history	Creates a new commit in history
Example Command	`git add file.txt`	`git commit -m "Message"`

#Concept-9

Comparison: `git log` vs. `git show`

Feature	git log	git show
Purpose	Displays commit history	Shows details of a specific commit
Scope	Lists multiple commits	Focuses on one commit at a time
Output	Commit hashes, authors, messages, dates	Commit details + file changes (patch format)
Common Options	`--oneline`, `--graph`, `--author`, `--since`	`git show <commit-hash>`
Example Command	`git log --oneline --graph`	`git show abc1234`

#Concept-10

Comparison: `git merge` vs. `git rebase`

Feature	git merge	git rebase
Purpose	Combines changes from one branch into another	Reapplies commits on top of another branch
Commit History	Preserves all commits and history	Creates a linear history by modifying commit order
Use Case	When preserving branch history is important	When a clean, linear history is preferred
Creates a Merge Commit?	Yes (for non-fast-forward merges)	No (rewrites history instead)
Command Example	`git merge feature-branch`	`git rebase main`
Handling Conflicts	Merges conflicts normally	Resolves conflicts commit-by-commit
Best For	Team collaboration, maintaining feature branches	Keeping a clean commit history before merging

#Concept-11

- **Difference: `git fetch` vs. `git pull`**

Command	What It Does	When to Use	Key Behavior
`git fetch`	Downloads changes from the remote but does **not** merge them into your local branch.	When you want to check for updates **before merging**.	Safer, lets you inspect changes first.
`git pull`	Fetches **and automatically merges** changes from the remote branch into your local branch.	When you want to **quickly sync** your branch with the remote.	Directly updates your working directory.

#Concept-12

Difference: Git Alias vs. Regular Git Commands

Feature	Git Alias	Regular Git Command
Definition	A shortcut for a Git command	The full default Git command
Usage	`git co -b feature` (Alias for `checkout`)	`git checkout -b feature`
Customization	Fully customizable	Fixed, cannot be modified
Typing Effort	Less, saves time	More, requires full command
Configuration	Needs manual setup using `git config`	Available by default
Flexibility	Can be personalized for any workflow	Standard commands only

#Concept-13

Difference: Git Rebase vs. Git Merge

Feature	Git Rebase	Git Merge
History	Linear, clean	Includes merge commits
Commit Order	Rewrites history	Preserves all commits
Merge Commits	Eliminates unnecessary merge commits	Creates a merge commit
Usage	Good for private branches	Preferred for shared branches
Conflict Resolution	Needs to be resolved interactively	Merges all conflicts in one step
Best For	Keeping a clean, readable commit history	Preserving a full commit timeline

#Concept-14

Difference: Git Stash vs. Git Commit

Feature	Git Stash	Git Commit
Purpose	Temporarily saves uncommitted work	Saves changes permanently in history
Visibility in History	× Not visible in commit history	✅ Appears in commit history
Switching Branches	✅ Useful for switching branches	✅ Can be used, but affects history
Commit Message Required?	× No	✅ Yes
Temporary or Permanent?	Temporary	Permanent
Applied Automatically?	No, needs `git stash pop/apply`	Yes, changes are committed

Example Usage:

- **Git Stash:** When you have unfinished work but need to switch branches or pull updates.
- **Git Commit:** When your changes are complete and should be recorded permanently.

#Concept-15

Difference: Git Clean vs. Git Reset vs. Git Stash

Command	What It Does	Effect on Tracked Files?	Effect on Untracked Files?
`git clean`	Removes untracked files permanently	× No effect	✅ Deletes
`git reset --hard`	Resets tracked files to last commit, discards all changes	✅ Resets changes	× No effect
`git stash`	Temporarily saves uncommitted changes for later use	✅ Saves temporarily	× No effect

- Git Clean: When you need to remove untracked files (like logs, temporary files, or build artifacts).

- Git Reset: When you want to discard changes in tracked files and revert to the last commit.
- Git Stash: When you want to save your work temporarily without committing it.

#Concept-16

Difference: Annotated Tags vs. Lightweight Tags

Feature	Annotated Tag (`git tag -a`)	Lightweight Tag (`git tag`)
Stores Metadata?	✅ Yes (author, date, message)	✗ No
Includes Message?	✅ Yes (`-m "Message"`)	✗ No
More Permanent?	✅ Yes, recommended for releases	✗ No, just a simple reference
Usage	Best for official releases & milestones	Quick labeling without details

#Concept-17

- Use Annotated Tags (git tag -a v1.0 -m "Release v1.0") for versioning official releases.
- Use Lightweight Tags (git tag v1.0) for temporary or internal checkpoints.

⚠ Important Notes:

- Tags are not pushed to remote automatically; use git push origin --tags.
- Checking out a tag (git checkout v1.0) puts you in a detached HEAD state (read-only).

#Concept-18

Difference: `git reset` vs. `git revert`

Feature	git reset	git revert
Modifies Commit History?	✅ Yes (Deletes commits)	✗ No (Creates a new commit)
Removes Commits?	✅ Yes (Moves HEAD backward)	✗ No (Keeps commit history)
Preserves History?	✗ No (Alters commit timeline)	✅ Yes (Maintains integrity)
Safe for Shared Branches?	✗ No (Changes history)	✅ Yes (Used in collaborative work)
Use Case	Undo commits (private branches)	Undo changes while keeping history

- Use git reset for local, private changes when you want to completely erase commits.
- Use git revert in shared repositories to undo changes without losing commit history.

⚠ Important:

- git reset --hard permanently deletes changes! Use with caution.
- git revert is safer because it does not modify past commits but instead creates a new commit.

#Concept-19

Difference: `git pull` vs. `git fetch`

Feature	git pull (Fetch + Merge)	git fetch (Only Fetch)
Downloads Remote Changes?	✅ Yes	✅ Yes
Merges Changes Automatically?	✅ Yes	× No
Creates Merge Commits?	✅ Yes (If conflicts exist)	× No
Safer for Reviewing Changes First?	× No (Merges directly)	✅ Yes (Requires manual merge)
Use Case	Auto-update local branch	Check remote changes before merging

- Use git pull when you want to fetch and merge remote changes immediately.
- Use git fetch when you want to review changes before merging them manually.

⚠ Important:

- If you want a cleaner history, consider using git pull --rebase to avoid extra merge commits.
- Always resolve conflicts manually if Git encounters issues during merging.

#Concept-20

🚀 Tuleap vs. Other Remote Servers

Feature	Tuleap	GitHub/GitLab/Bitbucket
Open-Source	✅ Yes	× GitHub, Bitbucket (Paid)
Git Support	✅ Yes	✅ Yes
CI/CD	✅ Integrated	✅ GitHub Actions, GitLab CI
Agile Tools	✅ Built-in	× Limited (Needs plugins)
Self-Hosting	✅ Yes	✅ GitLab (Paid for full features)

🔹 Agile vs. Tuleap vs. DevOps

Feature	Agile 🏃	Tuleap 🔧	DevOps 🚀
Definition	Agile is a software development methodology focusing on iterative and incremental development.	Tuleap is an **ALM (Application Lifecycle Management) tool** that supports Agile, Git, and DevOps.	DevOps is a **culture & practice** that combines development and operations for faster, automated delivery.
Purpose	Improves **team collaboration**, flexibility, and continuous feedback in software development.	Provides **tools to manage Agile, Git repositories, issue tracking, CI/CD, and documentation** in one platform.	Automates **software development, testing, and deployment** for **continuous integration & delivery (CI/CD)**.
Key Focus	**Processes** (Scrum, Kanban, Iterations)	**Tooling & Management** (Git, Issue tracking, Agile boards)	**Automation & Operations** (CI/CD, Infrastructure as Code)
Who Uses It?	Developers, Project Managers, Scrum Masters	Teams needing **complete project tracking & source code management**	Developers, IT Ops, DevOps Engineers
Core Components	- Scrum/Kanban		

🔹 Tuleap vs. Jira vs. Trello

Feature	Tuleap	Jira	Trello
Agile Support	✅ Scrum, Kanban, SAFe	✅ Scrum, Kanban	⚠️ Kanban only
Git Integration	✅ Built-in	⚠️ Requires Bitbucket	× No
CI/CD Support	✅ Built-in	⚠️ Requires Jira DevOps	× No
Self-Hosted Option	✅ Yes (On-prem & Cloud)	⚠️ Limited	× No
Open Source	✅ Yes	× No	× No
Enterprise Ready	✅ Yes	✅ Yes	⚠️ Limited

🔹 Difference Between DevOps and Agile

Feature	Agile 🏗️	DevOps 🚀
Definition	Agile is a **software development methodology** focused on iterative development and continuous feedback.	DevOps is a **culture and practice** that unites development (Dev) and operations (Ops) to automate software delivery.
Goal	Deliver software **faster and efficiently** with changing requirements.	Automate and streamline the **entire software lifecycle** from development to deployment.
Focus	**Development process** (coding, testing, feedback).	**Software delivery & deployment** (CI/CD, automation, monitoring).
Teams	Devs, testers, and business analysts collaborate.	Devs, Ops, and security engineers work together.
Delivery	Small, incremental releases.	Continuous delivery with automated deployments.
Key Practices	Scrum, Kanban, daily standups, sprints.	CI/CD, Infrastructure as Code (IaC), automated testing, monitoring.
Tools	Jira, Trello, Tuleap (for Agile planning).	Jenkins, GitLab CI/CD, Kubernetes, Docker.
Best For	Managing changing requirements, user stories, and iterative development.	Automating deployments, scaling applications, and reducing release failures.

🔹 Difference Between Agile and Tuleap

Feature	Agile 🏗️	Tuleap (Agile Tool) 📌
Definition	Agile is a **methodology** for iterative software development.	Tuleap is an **Agile project management and DevOps tool** that helps implement Agile practices.
Purpose	Provides a **framework** for flexible, iterative development.	Provides **features** to manage Agile projects (Scrum, Kanban, SAFe).
Implementation	Agile can be applied manually or using tools like Jira, Trello, Tuleap.	Tuleap provides **built-in Agile features** like backlogs, Kanban boards, and burndown charts.
Methodologies	Supports **Scrum, Kanban, SAFe, XP, Lean, etc.**	**Supports Scrum, Kanban, and SAFe** with built-in tools.
Tracking	Agile requires **manual tracking** of sprints, tasks, and progress.	Tuleap offers **automated tracking**, dashboards, and real-time collaboration.
Integration	Agile itself does not integrate with DevOps tools.	Tuleap integrates with **Git, Jenkins, CI/CD, and code review systems**.
Best For	Teams that follow **Agile principles**, with or without a tool.	Teams that need **an Agile & DevOps platform** to manage projects efficiently.