GIT & GITHUB

1)Introduction to GitHub

GitHub is a web-based platform built on top of Git, used for hosting code repositories, collaborating with developers, and managing version control. It provides a graphical interface, automation tools, CI/CD pipelines, project management features, and social coding capabilities — making software development faster and more organized.

2)Why GitHub Is Used

GitHub is widely used because it:

1. Enables Collaboration

Multiple developers can work on the same project through:

- branches
- pull requests
- code reviews

2. Provides Version Control

All past versions of code are stored, allowing:

- rollbacks
- comparing changes
- tracking authors

3. Offers Cloud Hosting for Git Repos

No need to maintain your own Git server.

4. Integrates with DevOps

- GitHub Actions
- CI/CD
- Automated testing

5. Community & Open Source

Millions of open-source projects are hosted here.

3)GIT VS GITHUB

	
1. It is a software	1. It is a service
2. It is installed locally on the system	2. It is hosted on Web
3. It is a command line tool	3. It provides a graphical interface
4. It is a tool to manage different versions of edits, made to files in a git repository	4. It is a space to upload a copy of the **Git** repository
5. It provides functionalities like Version Control System Source Code Management	5. It provides functionalities of Git like VCS, Source Code Management as well as adding few of its own features

4)GIT ARCHITECTURE & WORKING

Git Thinks in Snapshots, Not Deltas

Traditional VCS tools view data as files plus the changes made to those files (deltas). Git fundamentally rejects this worldview.

1.1 Snapshot Model

Every time you commit, Git creates a snapshot of your entire project — what every file looks like at that moment.If a file hasn’t changed, Git doesn’t store it again.

Instead, it stores a reference (a pointer) to the previously stored identical file.

So a commit in Git is basically:

A tree object → representing your project directory

↳ contains references to blob objects → your files

And a commit object → pointing to this tree + metadata + parent commits

Why Snapshot Architecture Matters

Merging becomes a matter of comparing entire states, not line deltas.

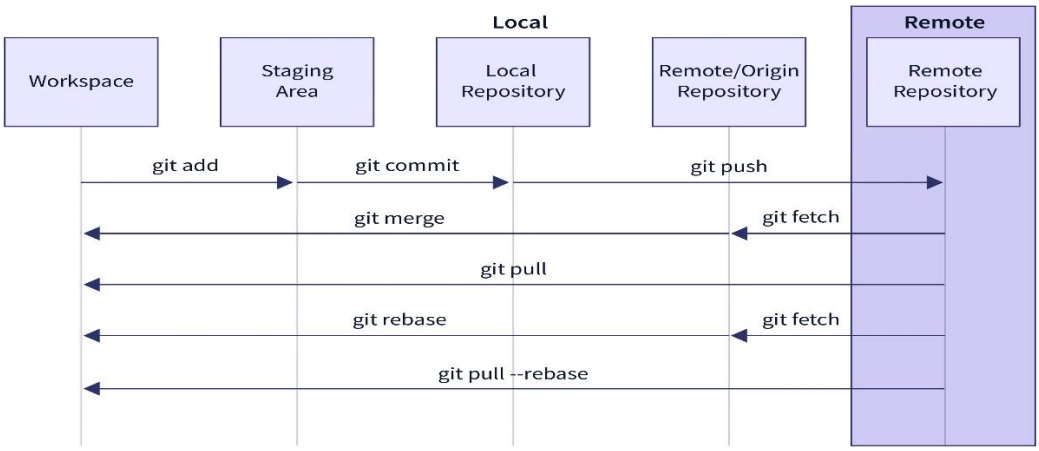
Branching becomes trivial because branches just point to commits (snapshots), not duplicated files.

The system becomes shockingly fast because structurally identical snapshots share objects.

Git's snapshot model lays the foundation for its two core internal data structures:

CAS (Content Addressable Storage)

DAG (Directed Acyclic Graph)



Git's Content Addressable Storage (CAS)

At its lowest level, Git is a persistent key–value store, located inside the .git/objects/ directory.

Mechanism

Anything you store in Git (file contents, directory structure, commit metadata) becomes an object:

Git computes a SHA-1 hash of the content.

That hash becomes the name of the object on disk.

Nuance: Deduplication

Because an object's identity is 100% determined by its content:

- Two identical files → same hash → stored only once.
- Ten thousand copies of a PNG image → stored once.

Git achieves implicit compression simply through hashing.

Nuance: Immutability

Objects in Git **can never be changed** because:

- Any change to content → new SHA-1 → new object.

Immutability ensures:

- History cannot be corrupted accidentally.
- Commits are stable and reproducible.
- Git can optimize heavily because it never updates old objects.

CAS is the storage engine.
The **DAG** is the structure built on top of this storage.

The Directed Acyclic Graph (DAG)

Git stores commits as **nodes** in a graph where each node:

- Contains a hash of its *parent commit*.
- Points to a tree (snapshot of the project).

Directed

Each commit points **backwards** to its parent(s).
Parents do NOT point forward.

C → B → A

This means the direction of history flows backward from child to parent.

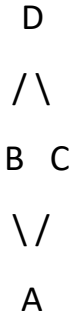
Acyclic

A commit can never be its own ancestor.
Cycles are impossible because hashes chain backwards in time.

Topology — the Real Power

The DAG structure is what makes Git's merging so intelligent.

When merging two branches:



Git finds the **Lowest Common Ancestor (LCA)** to understand where the histories diverged.

This means Git usually produces flawless auto-merges because the graph structure encodes intent, not just text differences.

Cryptographic Chain of History

Each commit includes its parent's hash inside its own header.

This creates a **Merkle Tree**, where:

- If anything in history changes,

- every commit hash after that point changes too.

This is the same concept used in blockchains.

This guarantees **integrity of the entire history**.

Complete Decentralization — A Core Architectural Philosophy

Git does not recognize any repository as "the server" or "the master copy."

Every clone of a repository is:

- A complete backup of the entire history
- A full database with all commits, branches, and tags
- A fully functional system capable of offline work

5) GIT COMMANDS AND THEIR OUTCOME

Git Configuration Commands

git config --global user.name "RakeshS03" →Set global username.

git config --global user.email "rakeshsankar2004@gmail.com" →Set global email.

git config --global color.ui auto → Enable colored output.

git config --list → Show all Git configurations.

git config --global core.editor "editor" → Set default editor.

Creating & Initializing a Repository

git init →Create a new local repository.

git clone <url> →Clone a remote repository.

git clone <url> <folder> →Clone repo into a specific folder.

Basic Snapshot Commands (Add → Commit → Push)

git status →View tracked/untracked file changes.

git add <file> →Add a file to staging.

git add . →Add all files to staging.

git commit -m "message"→Commit staged changes.

git commit -am "message"→Add & commit tracked files in one step.

git push →Push commits to remote.

git push origin <branch> →Push a specific branch.

Branching Commands

git branch →List local branches.

git branch -r →List remote branches.

git branch <name> →Create a new branch.

git checkout <branch> →Switch to a branch.

git checkout -b <branch> →Create + switch to branch.

git switch <branch> →Switch branches (newer).

git switch -c <branch> →Create + switch (newer).

git branch -d <branch> →Delete branch (safe).

git branch -D <branch> →Force delete branch.

Merging & Rebasing

git merge <branch> →Merge a branch into current branch.

git merge --abort →Stop merge & revert.

git rebase <branch> →Reapply commits on top of another branch.

git rebase --continue →Continue after conflict fixed.

git rebase --abort →Abort rebase.

Pull & Fetch

git fetch →Download remote changes (without merge).

git fetch --all →Fetch all remotes.

git pull →Fetch + merge remote branch.

git pull --rebase →Fetch + rebase with remote.

Log & History

git log →Show commit history.

git log --oneline →One-line history.

git log --graph →Visual commit graph.

git log -p →Show changes in each commit.

git blame <file> →Show who last modified each line.

git show <commit> →Show specific commit details.

Comparing Changes

git diff →Compare working directory to staging.

git diff --staged →Compare staging to last commit.

git diff <branch1> <branch2> →Compare two branches.

Undoing Changes

git restore <file> →Undo local file changes.

git restore --staged <file> →Unstage a file.

git reset <file> →Unstage file (older method).

git reset --soft HEAD~1 →Undo commit but keep staged changes.

git reset --mixed HEAD~1 →Undo commit & unstage changes.

git reset --hard HEAD~1 →Remove commit + all changes.

git revert <commit> →Create a new commit undoing 1 commit.

Remote Repository Commands

git remote -v →Show remotes.

git remote add origin <url> →Add remote repository.

git remote remove <name> →Delete a remote.

git remote rename <old> <new> Rename a remote.

Stashing (Temporary Save)

git stash →Save uncommitted changes.

git stash -u →Stash including untracked files.

git stash list →List all stashes.

git stash apply →Reapply last stash.

git stash pop →Reapply + delete stash.

git stash drop →Delete a stash.

git stash clear →Delete all stashes.

Tagging Versions

git tag →List all tags.

git tag <name> →Create a lightweight tag.

git tag -a <name> -m "msg" →Create annotated tag.

git push origin --tags →Push all tags.

Git Ignore

Create .gitignore →Specify files to ignore.

git rm -r --cached . →Clear cached files after updating .gitignore.

Cleaning Project Folder

git clean -n →Show files to be removed.

git clean -f →Delete untracked files.

git clean -fd →Delete untracked files + folders.

Archive & Submodules

git archive →Create a zip/tar of repo.

git submodule add <url> →Add submodule.

git submodule init →Initialize submodules.

git submodule update →Update submodules.

Advanced Plumbing Commands (Not used everyday)

git commit --amend →Modify last commit.

git cherry-pick <commit> →Apply specific commit from another branch.

git bisect →Find buggy commit via binary search.

git reflog →Show all HEAD changes, even deleted ones.

git gc →Cleanup unnecessary files.