What is Git?

**Git** is a **version control system** that helps developers **track changes in code** and **work together** on projects.

Version Control system?

A version control system is a tool that helps track, manage, and restore changes to files over time, especially in collaborative software development.

History and why?

Git was created by **Linus Torvalds** in **2005** to manage the development of the Linux kernel after issues with a previous tool called BitKeeper.

----------Problem of traditional file storage system?

Traditional file storage lacks version tracking, making it hard to manage changes, collaborate, and recover previous versions of files.

**Benefits of Git Over Other Systems**

* **Distributed system** – Every user has a full copy of the project and history.
* **Fast and efficient** – Operations like commit, merge, and branch are very quick.
* **Strong branching and merging** – Easy to create and manage branches.
* **Offline access** – Work without internet; push later when connected.
* **Better collaboration** – Multiple people can work without overwriting each other’s changes.
* **Track history** – Easily view and revert to earlier versions.
* **Secure** – Uses SHA-1 to protect code integrity and track changes.

### Software Available Like Git

* **Apache Subversion (SVN)** – Centralized version control system.
* **Mercurial** – Distributed like Git, simple and fast.
* **Perforce (Helix Core)** – Used in large enterprises and game development.
* **Bazaar (bzr)** – User-friendly distributed VCS.
* **CVS (Concurrent Versions System)** – One of the oldest version control systems.
* **BitKeeper** – Distributed VCS that inspired Git.

### 📊 Git vs Other Version Control Systems

| **Feature** | **Git** | **SVN (Subversion)** | **Mercurial** | **CVS** |
| --- | --- | --- | --- | --- |
| **Type** | Distributed | Centralized | Distributed | Centralized |
| **Speed** | Very Fast | Slower than Git | Fast | Slower |
| **Branching** | Easy & Lightweight | Difficult | Easy | Limited |
| **Offline Work** | Fully Supported | Limited | Fully Supported | Limited |
| **Popularity** | Very High | Medium | Low | Very Low |
| **Security** | High (SHA-1) | Medium | High | Low |
| **Learning Curve** | Moderate | Easy | Easy | Easy |

### What is a Repository?------

* A **repository (repo)** is a folder that Git tracks, including your code and its entire change history.
* It can be **local** (on your computer) or **remote** (like on GitHub or GitLab).
* A repo contains everything: source code, commits, branches, tags, and more.
* Git uses the repo to **track changes**, manage **versions**, and enable **collaboration**.
* You can **clone**, **pull**, **push**, and **commit** to a repository.

### 🖥️ Local vs 🌐 Remote Repository

| **Feature** | **Local Repository** | **Remote Repository** |
| --- | --- | --- |
| Location | On your own computer | On a server (e.g., GitHub, GitLab) |
| Internet Required | ❌ Not required | ✅ Required to push/pull changes |
| Access | Only accessible by you | Shared with team members |
| Speed | Very fast (works offline) | Slower (depends on internet) |
| Use Case | Make changes, test, commit locally | Collaborate, share code, backup |
| Example Command | git commit | git push, git pull, git clone |

### 🧾 Snapshots vs Differences ---- learn more

In Git, every time you make a commit, it **saves a full snapshot** of your files — like taking a photo of your entire project at that moment.

Older version control systems (like SVN or CVS) saved only the **differences (also called diffs)** — meaning just the changes made to files since the last version.

Using snapshots makes Git **faster, more reliable**, and **easier to recover** or switch to any past version.

Git also uses smart compression to avoid wasting space, even though it's storing full snapshots.

### SHA-1 in Git

* Git uses **SHA-1 (Secure Hash Algorithm 1)** to generate a unique 40-character hash for every commit, file, and object.
* It acts like a **digital fingerprint**, uniquely identifying each version.
* Even a **tiny change** in the file creates a completely **new hash**.
* It helps Git **track changes**, **verify data integrity**, and **prevent tampering**.
* Git stores data based on these hashes, ensuring **reliable and secure versioning**.
* SHA-1 makes Git commits **immutable** — you can’t change a commit without changing its ID.
* Like this f3c2a0e8d674b2e3b4cd1059d7b9f76a68b2c9a0

Git Architecture flow

Working Directory → Staging Area → Local Repository → Remote Repository

Short key words like --- m,u,a etc

### Git Architecture

Git uses a **distributed architecture**, meaning every developer has a full copy of the project’s code and its entire version history, enabling offline work and independent development.

#### 🔹 1. Working Directory

This is the folder on your computer where you view, edit, and work with project files directly.

#### 🔹 2. Staging Area (Index)

A hidden space where Git stores the changes you’ve marked (git add) to be included in the next commit.

#### 🔹 3. Local Repository

Contains the full history of your commits and branches, stored locally and updated using git commit.

#### 🔹 4. Remote Repository

A centralized shared location (like GitHub or GitLab) that stores the project online for team collaboration and backup.

### Common Git Workflows

#### 1. **Centralized Workflow**

* Everyone works on a **single main branch (usually main or master)**.
* All changes are pushed to one shared remote repository.
* Simple but **not ideal for large teams or feature development**.

#### 2. **Feature Branch Workflow**

* Every new feature or bug fix is done in a **separate branch**.
* Once done, the branch is **merged into the main branch**.
* Helps keep the main branch clean and stable.

#### 3. **Git Flow**

* A well-defined branching model with multiple branches like:
  + main (production),
  + develop (integration),
  + feature/\*, release/\*, and hotfix/\*
* Good for **structured release cycles** but can be complex for beginners.

#### 4. **Forking Workflow**

* Common in open-source projects.
* Each contributor **forks** (copies) the main repository and works independently.
* Changes are submitted via **pull requests** to the original repo.

### Branching in Git – Key Ideas Explained Clearly

**What are branches?**

* A branch is an independent line of development that starts from a specific commit, letting you work on new features or fixes **without touching the main code**.  
     
  **Why do we need branches?**
* They **isolate work**, so experiments don’t break the stable codebase.
* Teams can **develop multiple features in parallel** and merge only when each is ready.
* They provide a safe space to **review, test, or discard** changes before they reach production.  
     
  **Why does Git make branching so easy?**
* Branches are just lightweight pointers to commits, so creating, switching, and deleting them is **instant and low‑cost**.
* Because every developer has a full local repository, branching and merging happen **offline and fast**, without waiting for a central server.  
     
  **Merging vs. Rebasing (high‑level view)**
* **Merging** brings two branches together by creating a new “merge commit” that combines their histories, preserving every branch’s diverging timeline.
* **Rebasing** moves (replays) your branch’s commits on top of another branch, creating a **linear, cleaner history** but rewriting commit IDs.  
     
  **When to choose which?**
* Use **merge** when you want a clear record of where branches diverged and re‑joined (e.g., finishing a feature branch).
* Use **rebase** when you prefer a tidy, straight line of commits (e.g., updating a feature branch with the latest main changes before opening a pull request).

### Distributed Version Control System (DVCS)

A **Distributed Version Control System** is a system where **every developer has a full copy** of the entire project repository, including all files and the complete history.

### 🔧 How it works:

* In DVCS, there is **no single central repository** that everyone depends on.
* Every user has their own **local repository** with full project history.
* You can **commit, branch, and view history locally** — without internet.
* To collaborate, users **push and pull** changes to/from a **remote repository** (like GitHub).

### ✅ Why it’s useful:

* **Work offline** anytime — you don’t need a server connection to make commits or view logs.
* **Faster operations** like commits, diffs, logs, and switching branches, since everything is local.
* **Safe from data loss** — every developer has a backup.
* **Better collaboration** — changes are shared when ready, avoiding conflicts during development.

### 🆚 Compared to Centralized VCS:

| **Centralized (e.g., SVN)** | **Distributed (e.g., Git)** |
| --- | --- |
| Requires constant internet | Can work fully offline |
| Single point of failure | Multiple backups (local copies) |
| Slower operations | Faster operations (local) |
| Less flexible for branching | Easy, cheap branching |

**1. Centralized Workflow**

* Everyone commits to a **single shared main branch**.
* Simple setup but **not ideal for teams or parallel features**.

**2. Feature Branch Workflow**

* Each task is done in a **separate branch**.
* Main branch stays **clean and stable**.

**3. Git Flow**

* Uses structured branches: **main, develop, feature, release, hotfix**.
* **Great for release management**, but complex for beginners.

**4. Forking Workflow**

* Developers **fork the repo** and work independently.
* Contributions via **pull requests** to the original project.

git log – View full commit history with detailed info.

git log --oneline – View commits in a brief, one-line format.