GIT Architecture

Git Architecture: The Big Picture

Git's job is to track everything in your project—files, folders, changes, history, you name it. Imagine you've got a file called file.txt. At one point, it's version V1.0. You tweak it—add a line, delete something, whatever—and now it's V1.1. Git's role?

To remember that each space a time, there was a directory with file.txt at V1.0, there some dude (you?) messed with it and turned it into V1.1. It makes **who did what, when, and how**. That's the magic of version control.

Key Git Requirements

Git's gotta nail these things:

- **Track Everything:** Files, folders, name changes, and even permissions (like if someone flips a file's mode).
- **DIS Independent:** Works the same on everybody's Linux, macOS, or Windows.
- **Unique IDs:** Every tracked thing gets its own ID, so it can't get lost in the drama.
- **Track History:** Keeps every change—who did it, when it happened, and why (if someone writes a decent commit message).
- **Be Decent Storage:** Don't waste space storing stuff it doesn't need to.

Working Trees vs. Git Repository

Working Tree

This is your actual project folder where you see the files. It's just one copy at a time—like V1.1 after you've edited it.

Git Repository (Repo)

Lives inside your project as a hidden folder called .git. It's got all versions (V1.0, V1.1, etc.) stored away. Think of it as a **supplied pocket** Z: handing and recording everything in your directory.

Example: You've got file.txt V1.0, you change it to V1.1. Git notice working tree, you see V1.1. Cool. Fix that repo, you've got both V1.0 and V1.1 tracked. And all the history of previous versions.

Why File Names Can't Be IDs

You wanna track everything about file.txt, including its name. Cool. File name itself will be the identifier, you're minded.

Example: You see delete file.txt and create a new file.txt with the same name. Alice's Git supposed to tell them apart if the name's the ID? Cut back?

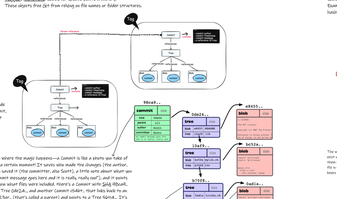
Solution: File names (and metadata) can't be identifiers. Gotta track stuff about we track.

That's why Git turns files and folders into objects with their own IDs.

Git Objects: What Are They?

Inside the repo, files and folders aren't what you know—they're objects.

- **Blob:** A file's content + metadata (type, permissions, size, name). File.txt becomes a "blob" for a file anyone.
- **Tree:** A folder's content + metadata. Your directory becomes a "tree" in Git's world.
- **Commit:** A snapshot of all your changes.
- **Tagged Annotations:** Labels for specific points in history. These objects flow (from inside) as file names or folder structures.


Commit
Tree
Blob
Tree
Blob
Blob
Tree
Blob
Blob
Tree
Blob