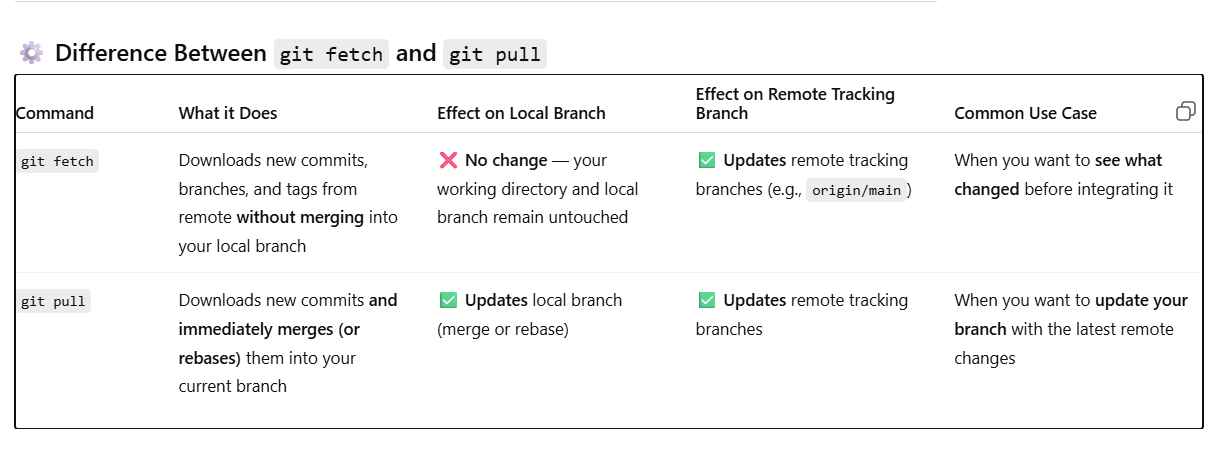
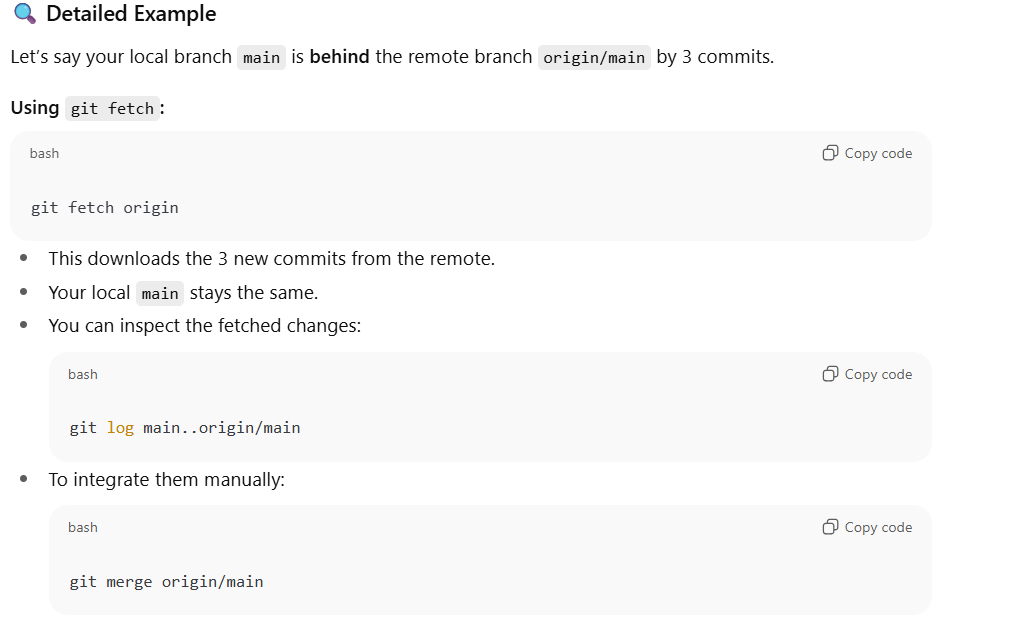
**GIT**

1. What is the difference between git fetch and git pull?

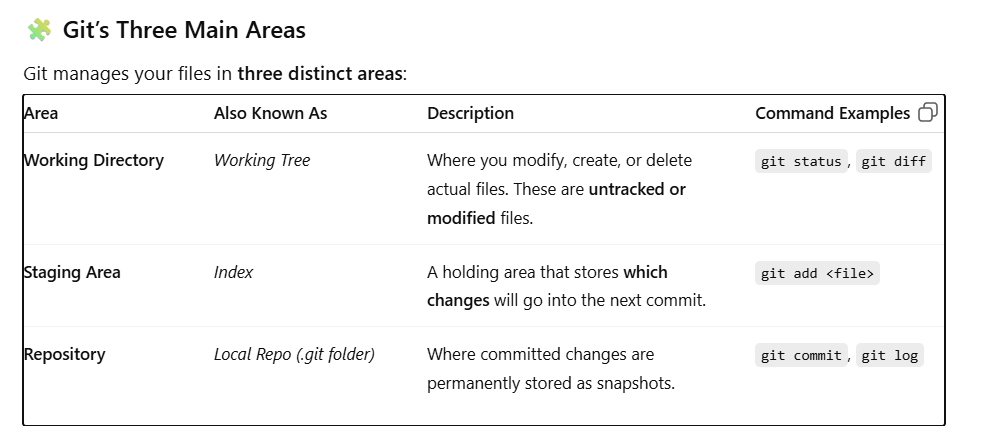




A screenshot of a computer

AI-generated content may be incorrect.

1. How does Git’s three main areas work — Working Directory, Staging Area, Repository?

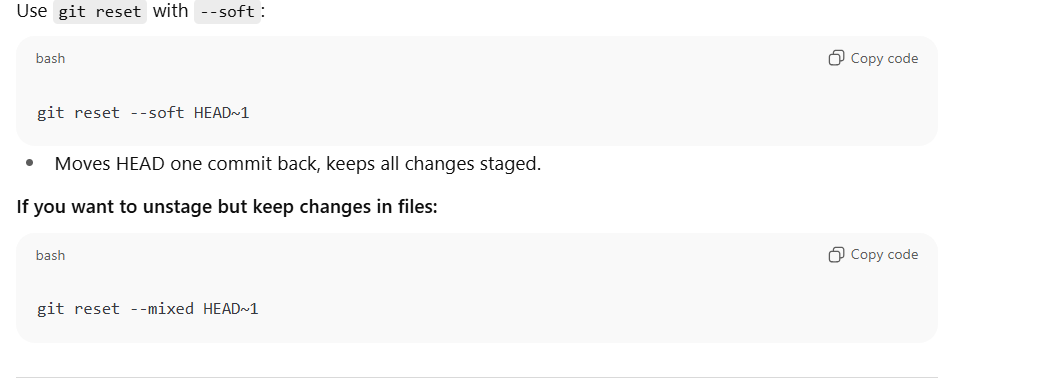


A screenshot of a computer

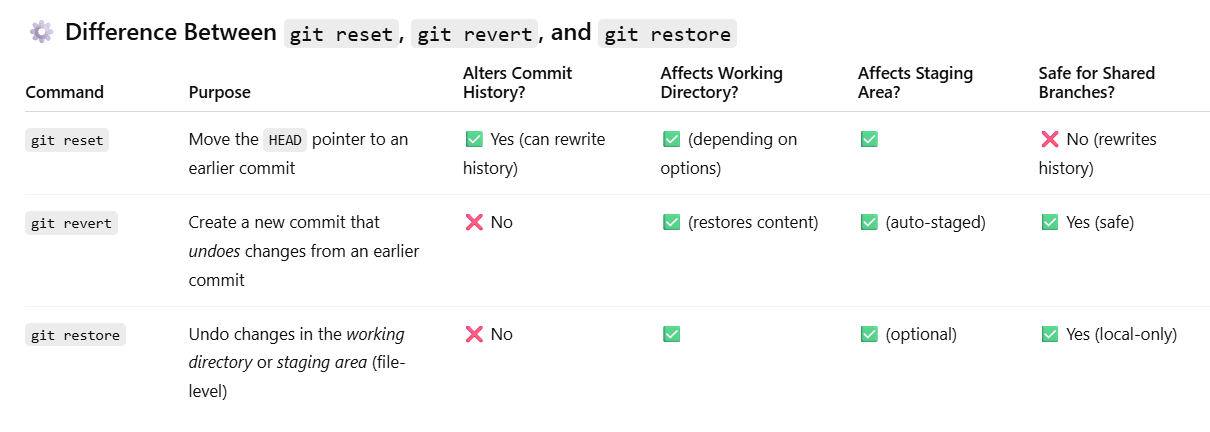
AI-generated content may be incorrect.



1. How do you undo a commit but keep changes locally?



1. What is the difference between git reset, git revert, and git restore?



**🧩 1️⃣ git reset — *Moves HEAD (rewrites history)***

**Purpose:**  
Undo commits or unstage/unstage changes by moving the **HEAD pointer** to another commit.

**Variants:**

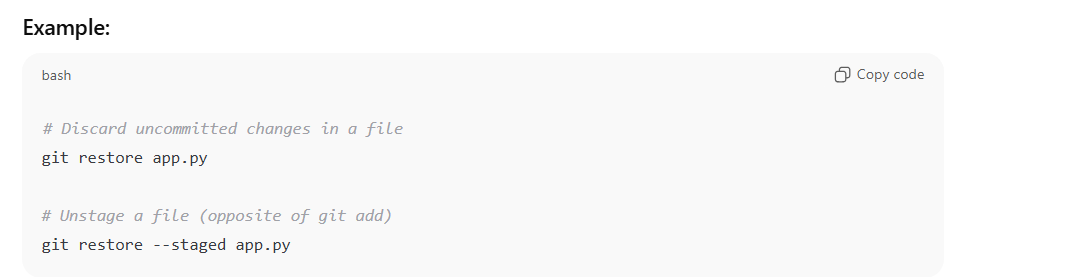
| **Command** | **Effect** |
| --- | --- |
| git reset --soft <commit> | Moves HEAD to <commit>, keeps changes staged |
| git reset --mixed <commit> (default) | Moves HEAD, unstages files, keeps changes in working dir |
| git reset --hard <commit> | Moves HEAD and **discards** all changes (staged + working dir) |

**🧩 2️⃣ git revert — *Creates a new “undo” commit (safe way)***

**Purpose:**  
Undo a commit **by adding a new commit** that reverses the previous one’s changes.

**🧩 3️⃣ git restore — *Fix working directory or staging area***

Introduced in newer Git versions (2.23+), restore provides a **clearer, safer** alternative to parts of git reset.

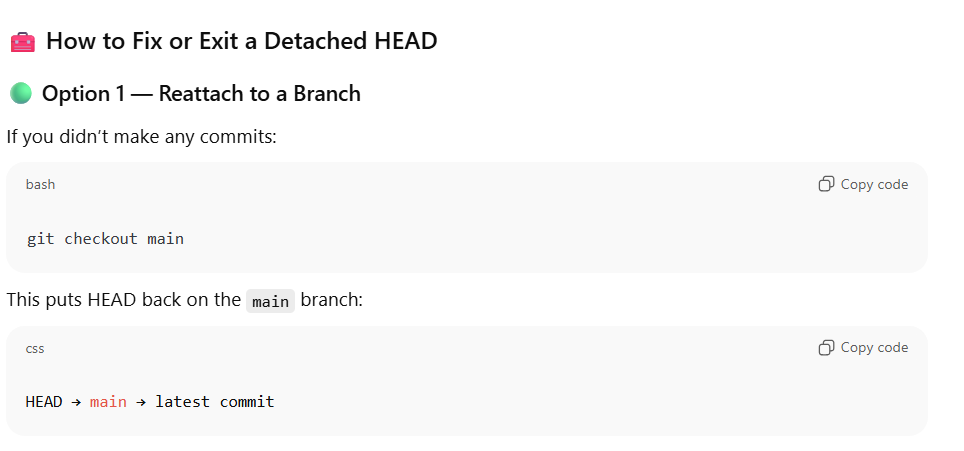


1. What is a detached HEAD state and how do you fix it?

In Git, **HEAD** is a special pointer that normally points to the **latest commit on the current branch**.

* HEAD → main → commit c3

But sometimes, you directly check out a **specific commit**, **tag**, or **remote branch** — not a branch name.  
When that happens, **HEAD becomes detached** — it points directly to a commit instead of a branch.

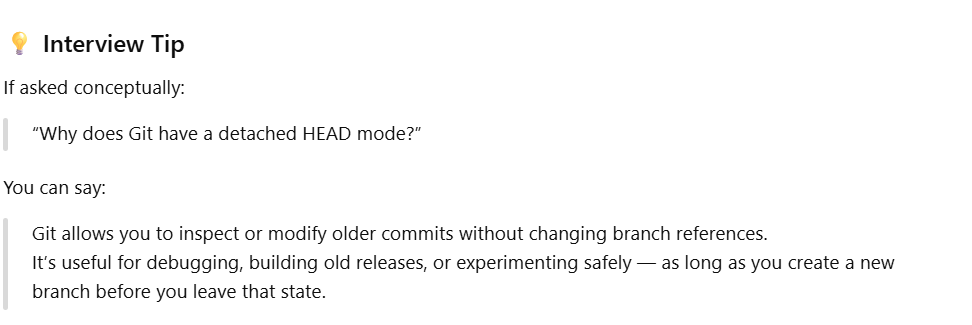


A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



1. How does Git handle merge conflicts, and how can you resolve them manually?

**⚙️ How Git Handles Merge Conflicts**

**🔹 What happens during a merge:**

When you run:

git merge <branch-name>

Git tries to automatically combine the changes from the target branch into your current branch.

Git can merge files **automatically** if:

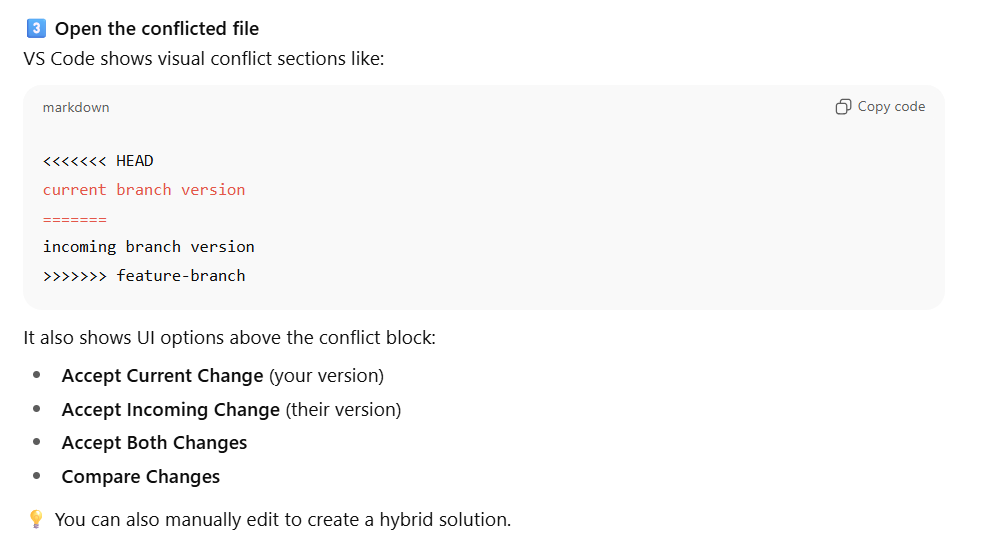
* The same lines haven’t been modified in both branches.
* Or, one branch only adds new files or lines.

**⚠️ When a Merge Conflict Occurs**

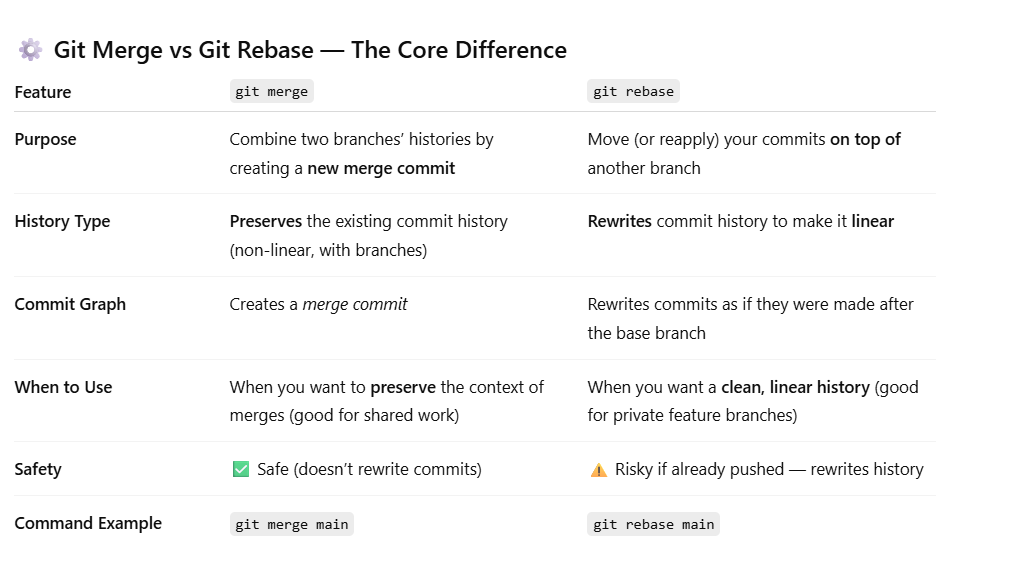
A **merge conflict** happens when Git cannot determine which version of a file to keep because:

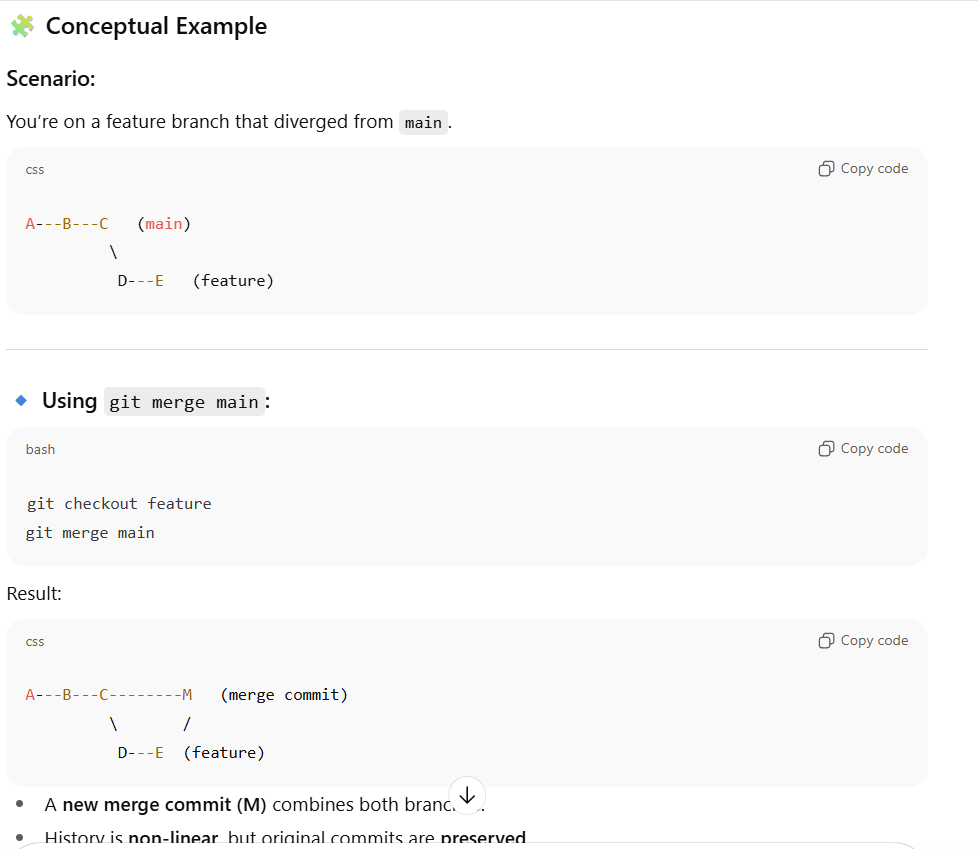
* The **same lines** were changed in both branches, or
* One branch **deleted** a file that the other **modified**.

🛠️ **How to Resolve Merge Conflicts Manually (in VS Code)**



1. What is the difference between git merge and git rebase?



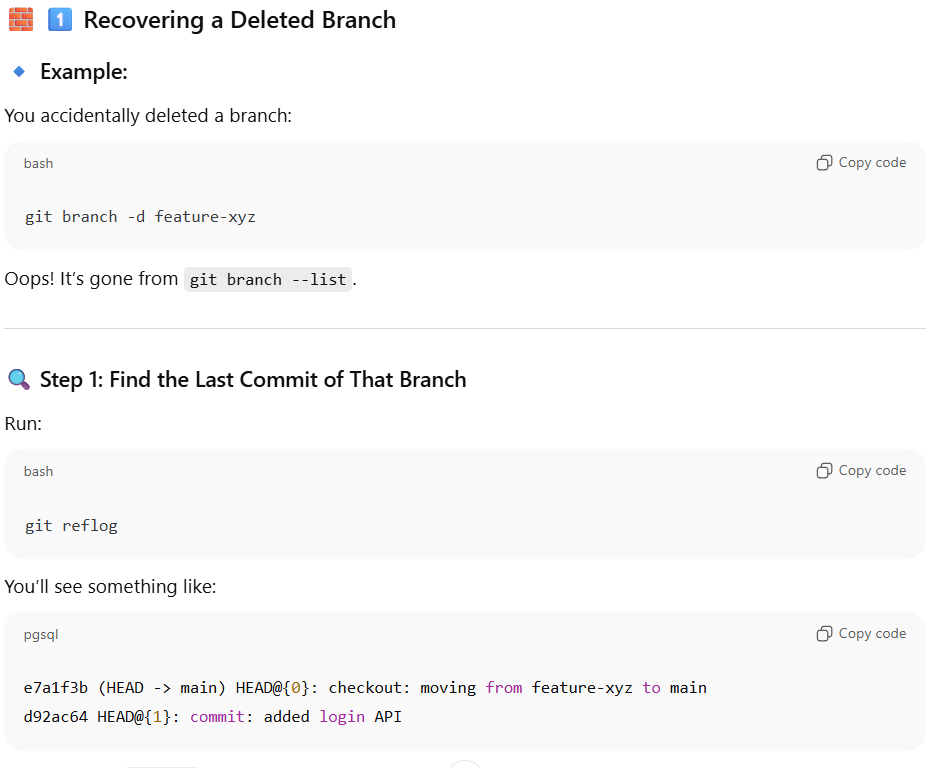


A screenshot of a computer

AI-generated content may be incorrect.

1. How do you recover a deleted branch or commit?

Even if you delete a branch or lose a commit, Git usually hasn’t erased it permanently.  
Git maintains a **reference log (reflog)** that tracks where your HEAD and branch pointers have been.



A screenshot of a computer

AI-generated content may be incorrect.

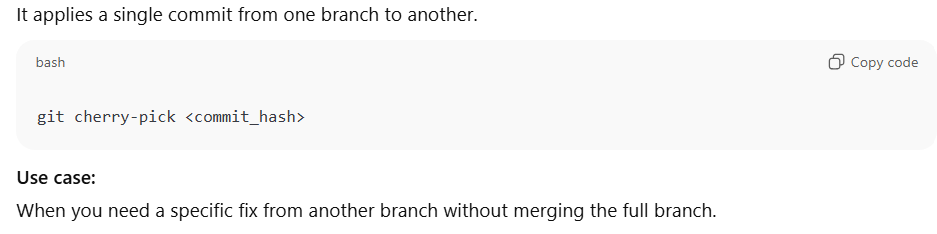
A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

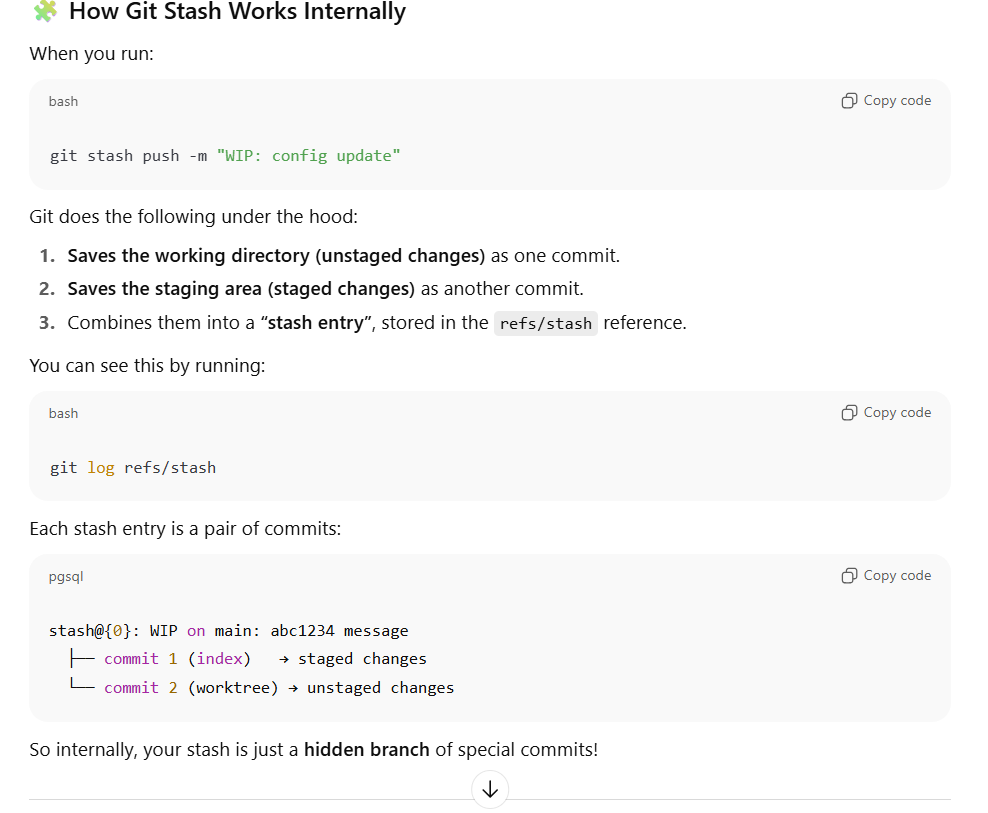
AI-generated content may be incorrect.

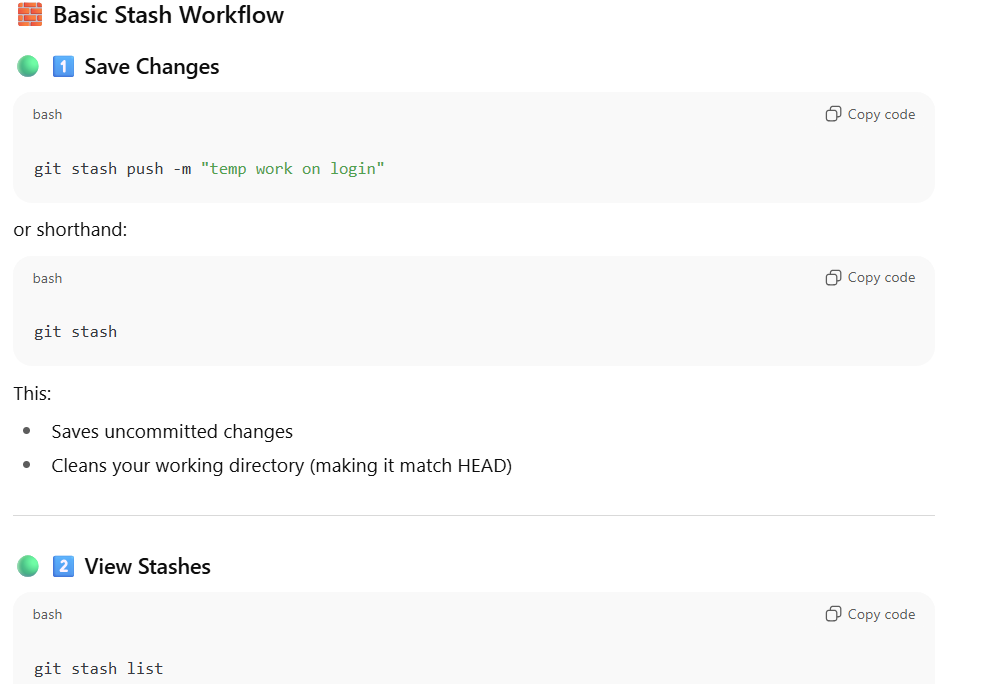
1. What is git cherry-pick and when would you use it?



1. How does git stash work internally and how to manage stashes?

git stash is a way to **temporarily save uncommitted changes** (both staged and unstaged) so you can switch branches or pull updates **without committing your work**.





A screenshot of a computer

AI-generated content may be incorrect.

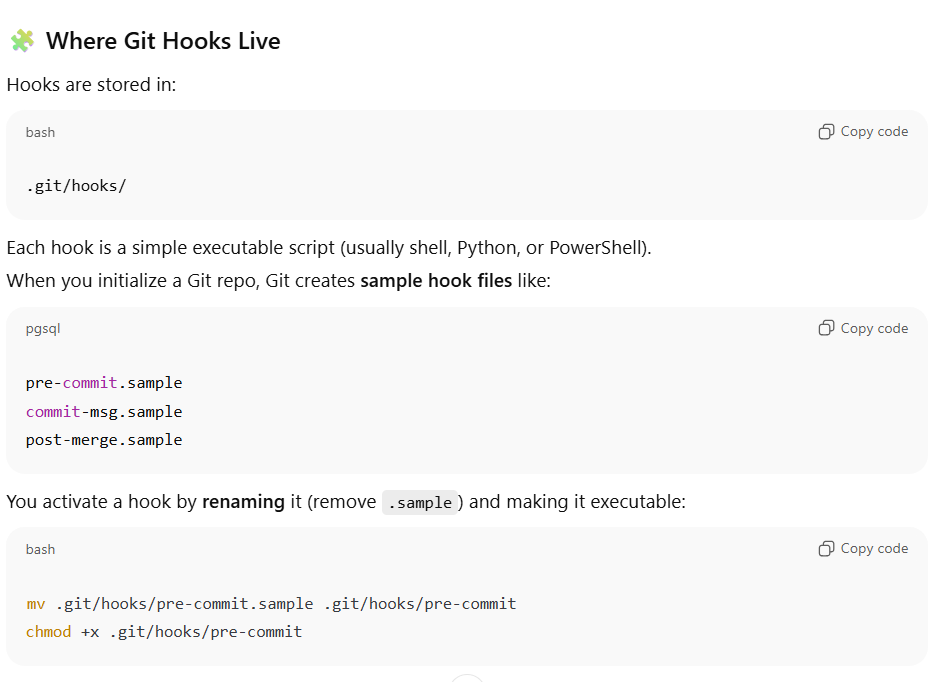
1. Explain Git hooks and their usage.

**⚙️ What Are Git Hooks?**

**Git hooks** are **scripts that Git automatically executes** in response to specific **repository events** such as committing, merging, pushing, or receiving changes.

They allow you to **customize and automate Git’s behavior** — for example:

* Enforcing commit message standards
* Running tests before pushing code
* Preventing accidental pushes to main
* Automatically updating version numbers



A screenshot of a web page

AI-generated content may be incorrect.

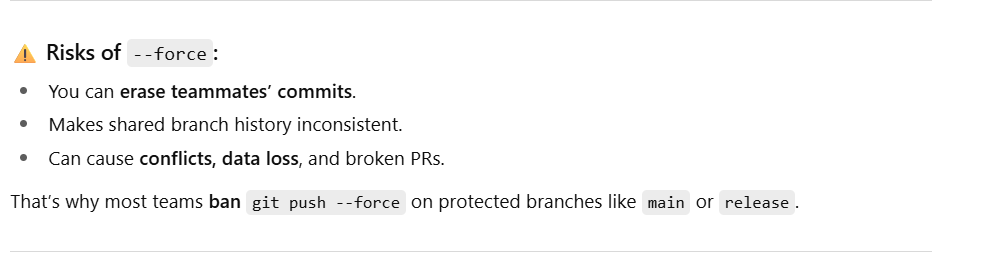
A screenshot of a computer

AI-generated content may be incorrect.

1. What happens when you run git push --force vs --force-with-lease?

A screenshot of a computer

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

1. How does Git handle large files (LFS)?

**🧩 1️⃣ The Problem — Why Git Struggles with Large Files**

Git was originally designed for **source code**, not **large binary files** (e.g., images, videos, datasets, build artifacts).

When you commit large files directly into Git:

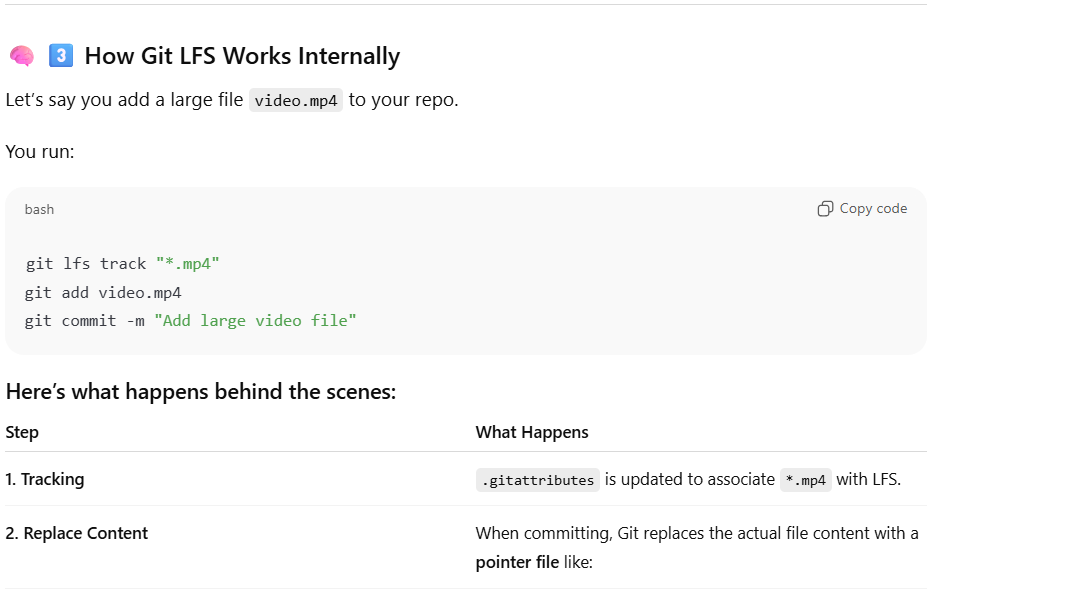
* The full file is **stored in every commit snapshot** — even small changes produce **full copies**.
* Repository size grows rapidly.
* Cloning and fetching become **extremely slow**.
* CI/CD pipelines using Git become **resource-heavy**.

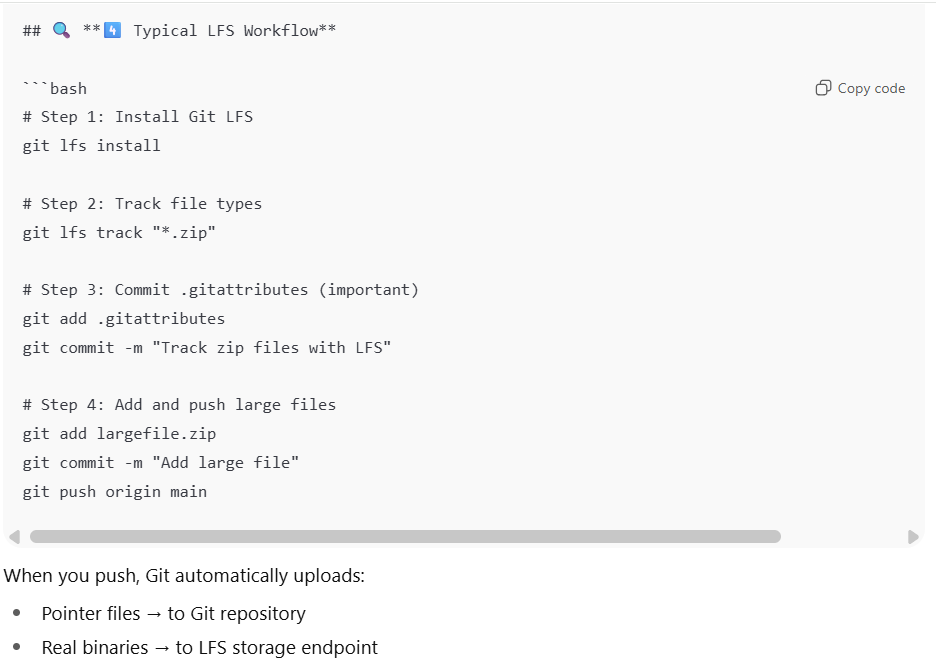
**🚀 2️⃣ The Solution — Git LFS (Large File Storage)**

**Git LFS** is a Git extension that manages large files **outside the main Git repository**, while keeping lightweight pointers inside Git history.

So instead of storing the big file content in .git/objects, Git stores:

* A **tiny text pointer file**, and
* The actual binary content in a **separate LFS storage** (like GitHub, GitLab, or your custom LFS server).







1. What is git worktree and how is it useful?

**🧩 1️⃣ The Problem — Why git worktree Exists**

Normally, a Git repository has **one working directory** checked out to **one branch at a time**.

So if you want to:

* Work on **multiple branches simultaneously**, or
* Test an older release without losing your current work

…you’d traditionally have to:

* Repeatedly git checkout different branches (slow and risky), **or**
* Clone the same repository multiple times (wastes disk space and time).

**🚀 2️⃣ The Solution — git worktree**

git worktree lets you **create multiple working directories** (each with its own checked-out branch) **that all share the same Git repository data** (.git).

So you can work on multiple branches **in parallel**, **without cloning** the repo again.

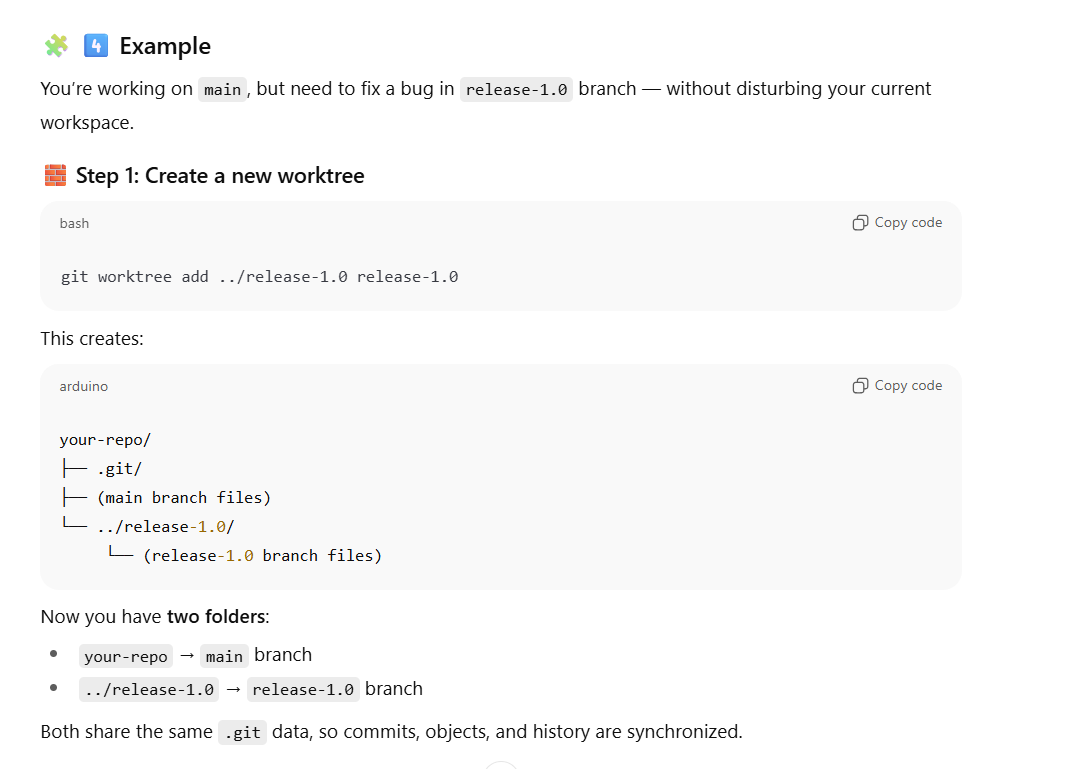
**🧠 3️⃣ How It Works**

Each *worktree*:

* Has its own **working directory** (its own files),
* Can checkout **a different branch or commit**,
* But **shares the same .git object database** (the internal history).

This means:

* Fast setup (no full clone)
* Minimal extra disk space
* Full Git functionality in each worktree



A screenshot of a computer

AI-generated content may be incorrect.

1. What’s the difference between origin/main and main?

* main: your **local branch**.
* origin/main: remote-tracking branch representing state of main on remote.

Run git fetch to update remote tracking branches.

1. How do you clean up your local repository safely?

**🧩 1️⃣ Why Cleanup Matters**

Over time, your local repo accumulates:

* Stale branches (merged or deleted from remote)
* Untracked files or build artifacts
* Old stashes and reflog entries
* Large packfiles and unreachable objects

These slow down Git commands and consume disk space.  
So a **safe cleanup routine** is about removing junk **without losing important history**.

**🧹 2️⃣ Step-by-Step: Safe Local Cleanup**

**🧱 Step 1: Fetch latest remote state**

Always start by syncing your local repo with remote before cleaning:

git fetch --prune

🧠 --prune deletes references to remote branches that no longer exist.

**🧱 Step 2: List local branches not on remote**

git branch -vv

Output example:

\* main abc123 [origin/main] latest commit

feature/old-branch def456 gone] work done long ago

If you see [gone], that branch no longer exists on remote.

🧹 To delete those stale branches safely:

git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -d

💡 Use -d (safe delete) — Git will refuse if branch isn’t merged.

If you’re sure and want to force:

git branch -vv | awk '/: gone]/{print $1}' | xargs git branch -D

**🧱 Step 3: Remove untracked files and directories**

Dry run first:

git clean -nd

Then actually clean (if safe):

git clean -fd

Options:

* -n → dry run (show what would be deleted)
* -f → force (required)
* -d → include directories
* -x → also remove ignored files (use with caution)

✅ Use -x **only** if you’re sure (e.g., to clean build artifacts).

**🧱 Step 4: Clean up old stashes**

List stashes:

git stash list

If they’re old and unused:

git stash clear

🧠 Or delete specific ones:

git stash drop stash@{2}

**🧱 Step 5: Clean up reflog (old commit references)**

Git keeps a **reflog** (reference log) of your branch changes — this can grow huge over time.

To see how big your repo is:

du -sh .git

Then prune unreachable commits:

git reflog expire --expire=30.days --all

git gc --prune=now --aggressive

✅ This reclaims space by compressing and deleting unreachable objects.

**🧱 Step 6: Remove unused worktrees (if any)**

If you’ve used git worktree:

git worktree prune

This removes references to deleted worktree folders.

**⚙️ 3️⃣ Optional: Clean up large objects (advanced)**

If your repo is huge because of accidentally committed large files, you can:

git verify-pack -v .git/objects/pack/\*.idx | sort -k3 -n | tail -10

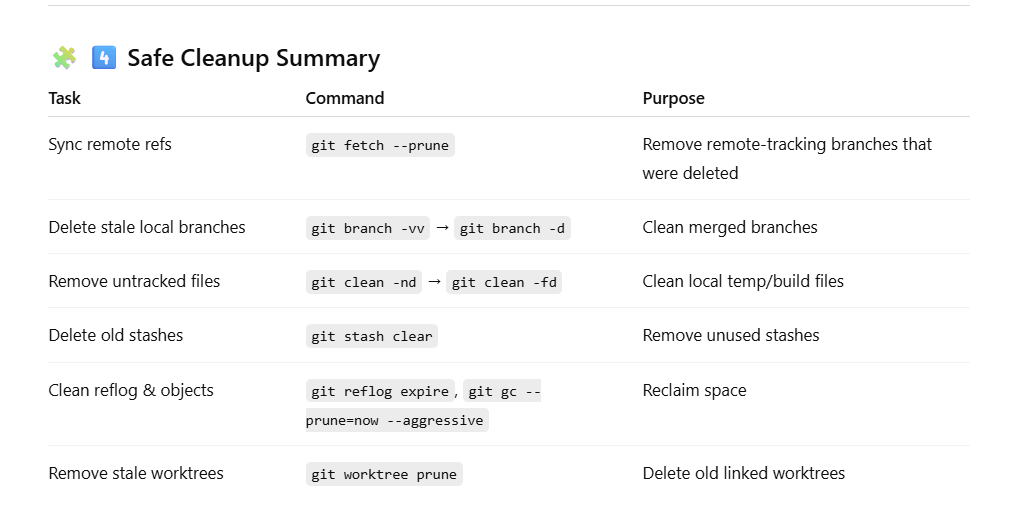
Shows largest objects.

Then use tools like:

* [git filter-repo](https://github.com/newren/git-filter-repo) (modern replacement for filter-branch)
* BFG Repo-Cleaner (simple for large file cleanup)

Example (remove all .zip files):

bfg --delete-files "\*.zip"



1. What is a squash commit in Git?

**🧩 1️⃣ The Concept — What “Squashing” Means**

**Squashing** means **combining multiple commits into a single commit**.

You typically do this to:

* Simplify your commit history,
* Merge a feature branch cleanly into main,
* Or remove noisy commits like “fix typo”, “small update”, “format code”.

