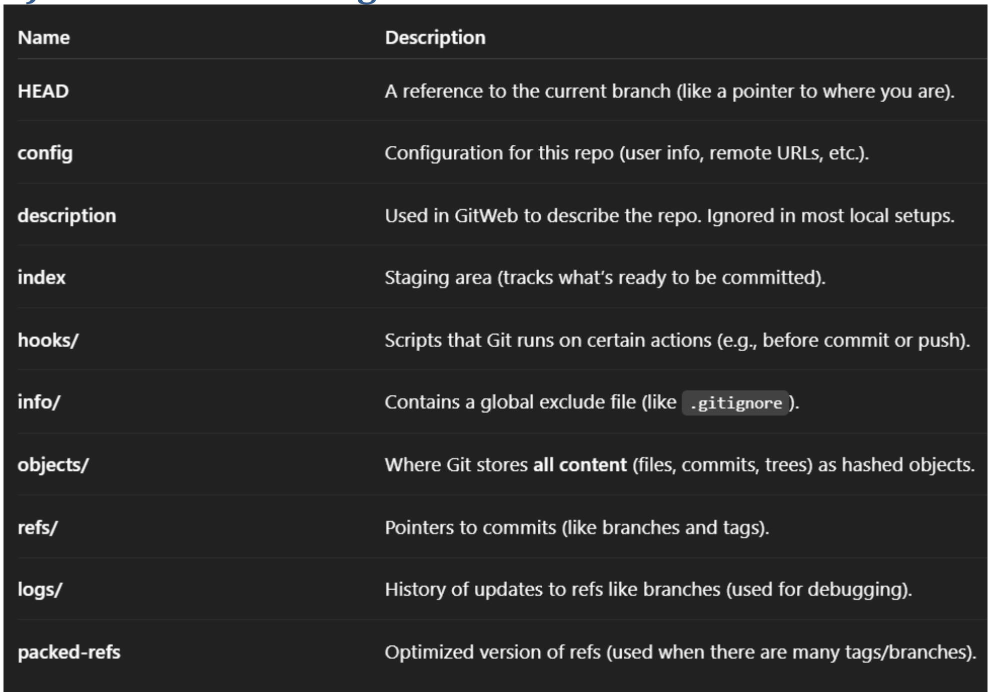
What is git?
Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer.

1)what are the other platform same as a github to version and maintain your code?
platforms like GitHub that are used for **version control, code hosting, and collaboration**
GitLab: **Best for:** All-in-one DevOps lifecycle.
Bitbucket:
SourceForge
AWS CodeCommit
Azure Repos :

2)git init

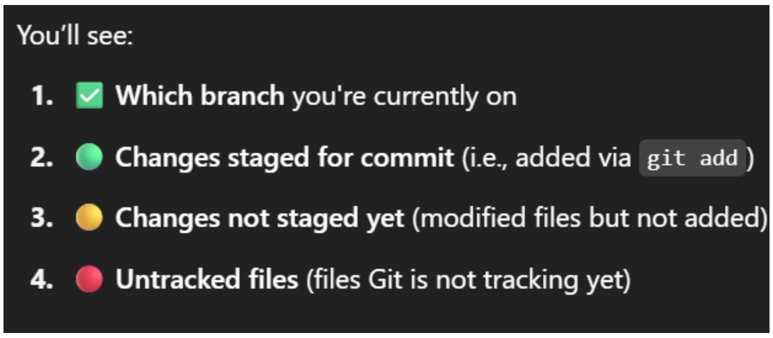
- It **creates a hidden folder** named `.git` in your directory.

- This `.git` folder contains all the metadata (version history, configs, logs, etc.) required to track changes in your code.

3)what is inside the .git folder?


Name	Description
HEAD	A reference to the current branch (like a pointer to where you are).
config	Configuration for this repo (user info, remote URLs, etc.).
description	Used in GitWeb to describe the repo. Ignored in most local setups.
index	Staging area (tracks what's ready to be committed).
hooks/	Scripts that Git runs on certain actions (e.g., before commit or push).
info/	Contains a global exclude file (like `.gitignore`).
objects/	Where Git stores **all content** (files, commits, trees) as hashed objects.
refs/	Pointers to commits (like branches and tags).
logs/	History of updates to refs like branches (used for debugging).
packed-refs	Optimized version of refs (used when there are many tags/branches).

4) What is the use of the git status command
The `git status` command is used to **check the current state of your working directory and staging area**.

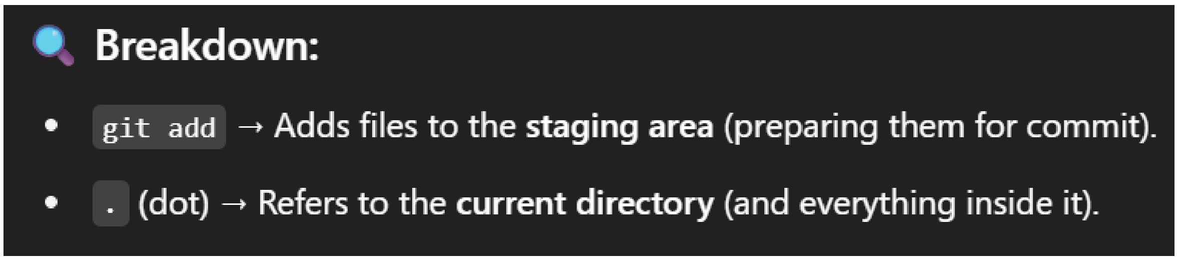

You'll see:

1. ✅ **Which branch** you're currently on
2. 🟢 **Changes staged for commit** (i.e., added via `git add`)
3. 🟡 **Changes not staged yet** (modified files but not added)
4. 🔴 **Untracked files** (files Git is not tracking yet)

Useful for if any changes are made after the git commit(modified file tracks)

5) use of git add .

"Stage **all changes** (new files, modified files, and deletions) in the **current directory and subdirectories** for the next commit."


🔍 **Breakdown:**

- `git add` → Adds files to the **staging area** (preparing them for commit).
- `.` (dot) → Refers to the **current directory** (and everything inside it).

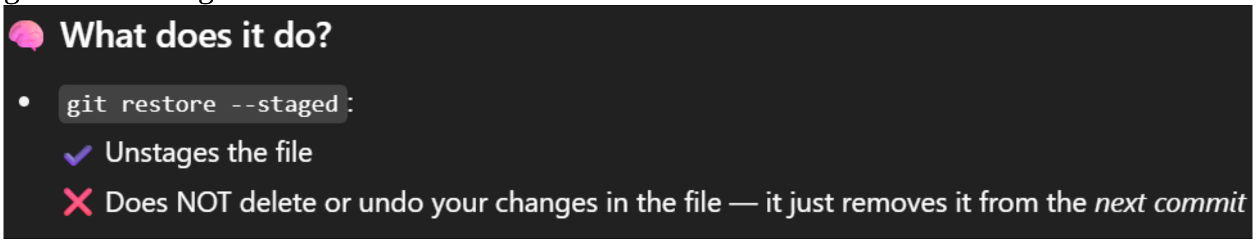
For a particular file if you want to add then command is git add <filename>

6) git commit –m "new commit"

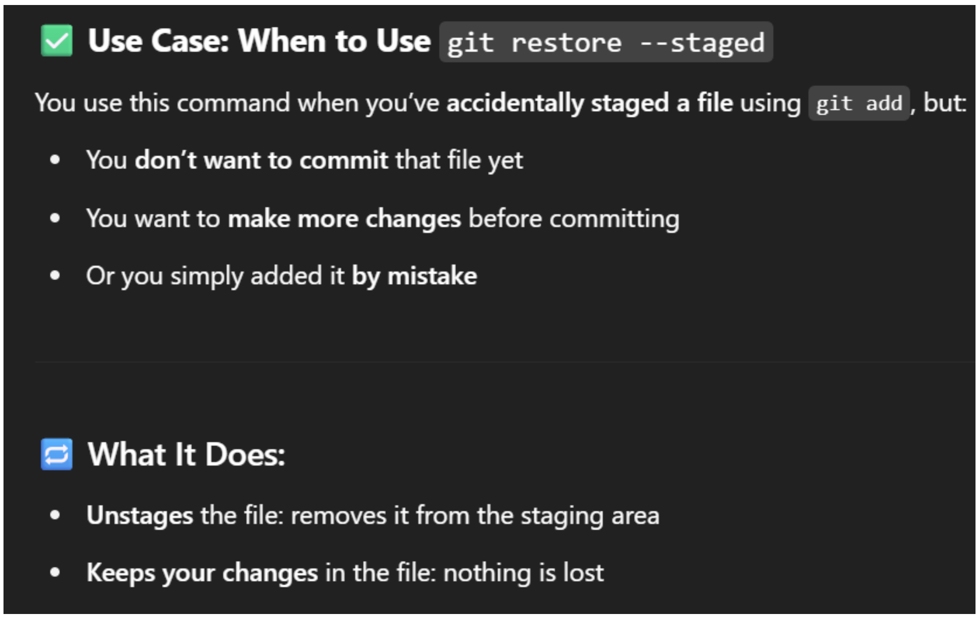
The `git commit -m` command is used to **save (commit) your staged changes** in Git **with a message**.

7) If you want to remove the stage of current file then you write git restore

git restore --staged <filename>


🧠 **What does it do?**

- `git restore --staged`:
  ✔ Unstages the file
  ❌ Does NOT delete or undo your changes in the file — it just removes it from the *next commit*


✅ Use Case: When to Use `git restore --staged`

You use this command when you've **accidentally staged a file** using `git add`, but:

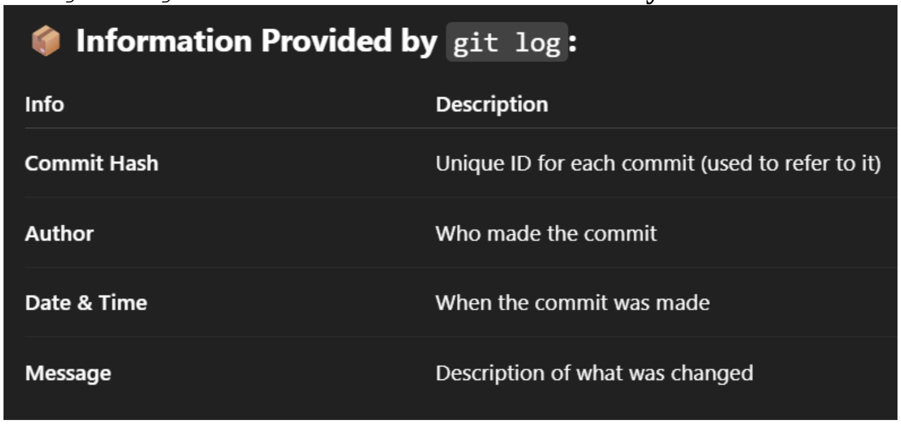
- You **don't want to commit** that file yet
- You want to **make more changes** before committing
- Or you simply added it **by mistake**

🔁 What It Does:

- **Unstages** the file: removes it from the staging area
- **Keeps your changes** in the file: nothing is lost

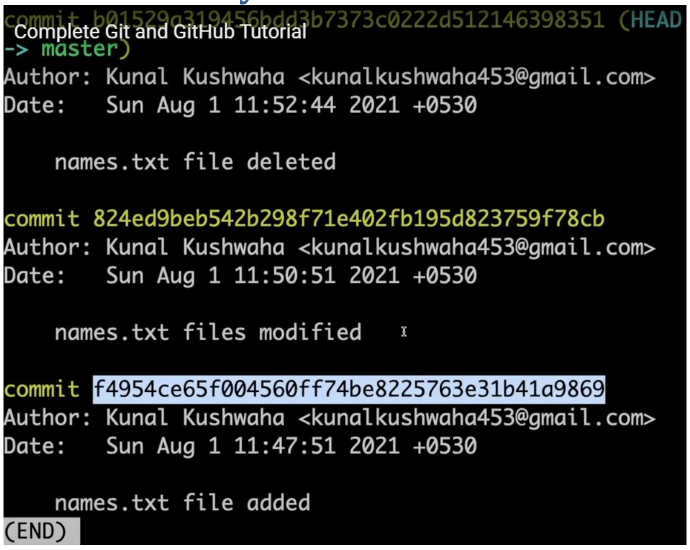
8) git log

The `git log` command is used to **view the history of commits** in a Git repository.


📦 Information Provided by `git log`:

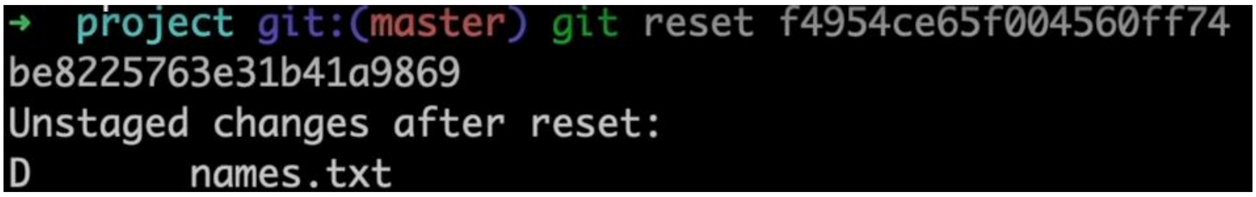
Info	Description
Commit Hash	Unique ID for each commit (used to refer to it)
Author	Who made the commit
Date & Time	When the commit was made
Message	Description of what was changed

9) if you want to remove the last commit because the file has been deleted by mistake then how can you do it?



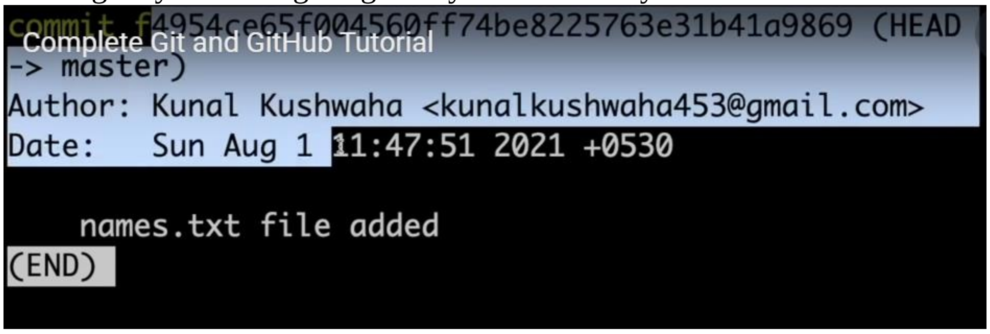
```
commit b01529a319456bdd3b7373c0222d512146398351 (HEAD
-> master)
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 11:52:44 2021 +0530

    names.txt file deleted

commit 824ed9beb542b298f71e402fb195d823759f78cb
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 11:50:51 2021 +0530

    names.txt files modified

commit f4954ce65f004560ff74be8225763e31b41a9869
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 11:47:51 2021 +0530

    names.txt file added
(END)
```

Suppose here you want to delete last two commits then
Then copy the hash before that last two commits



```
→  project git:(master) git reset f4954ce65f004560ff74
be8225763e31b41a9869
Unstaged changes after reset:
D       names.txt
```

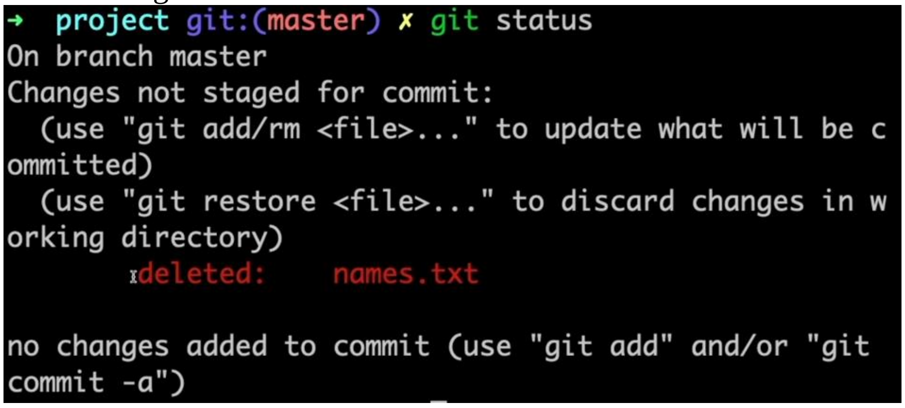
Then again you check git log now you will be only see the one commit



```
commit f4954ce65f004560ff74be8225763e31b41a9869 (HEAD
-> master)
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 11:47:51 2021 +0530

    names.txt file added
(END)
```

10) where these all changes has gone?

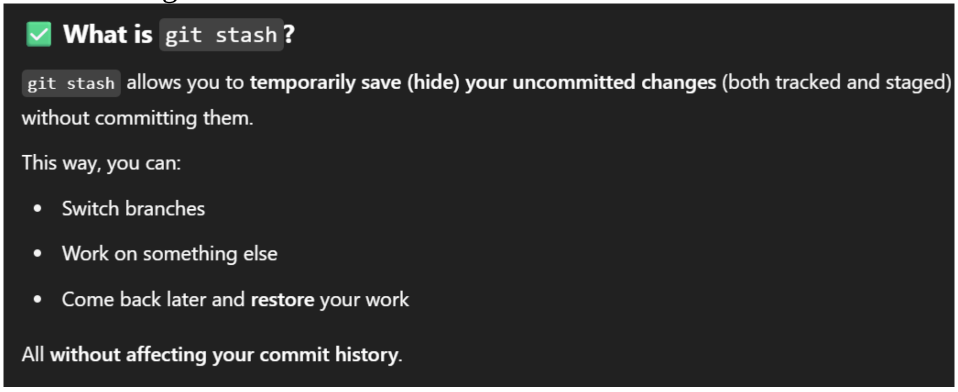
In the unstaged area



```
→  project git:(master) ✗ git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be c
ommitted)
  (use "git restore <file>..." to discard changes in w
orking directory)
        deleted:    names.txt

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

11) If you want that your progress or a feature of the project that code you have written , is there is anyway so that put your work somewhere else without making a commit and history and whenever you want that things back you get it

Command : git stash



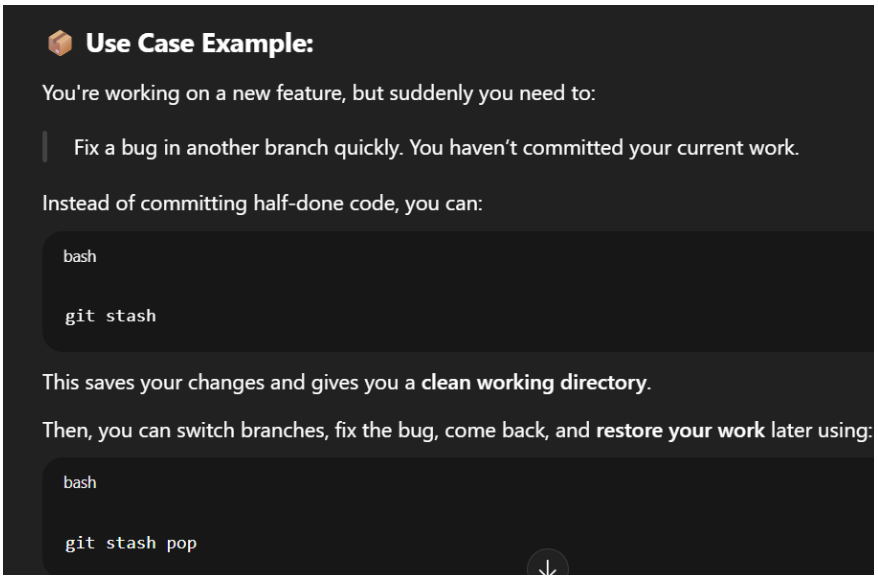
✅ **What is `git stash`?**

`git stash` allows you to **temporarily save (hide) your uncommitted changes** (both tracked and staged) without committing them.

This way, you can:

- Switch branches
- Work on something else
- Come back later and **restore** your work

All **without affecting your commit history**.



📦 **Use Case Example:**

You're working on a new feature, but suddenly you need to:

> Fix a bug in another branch quickly. You haven't committed your current work.

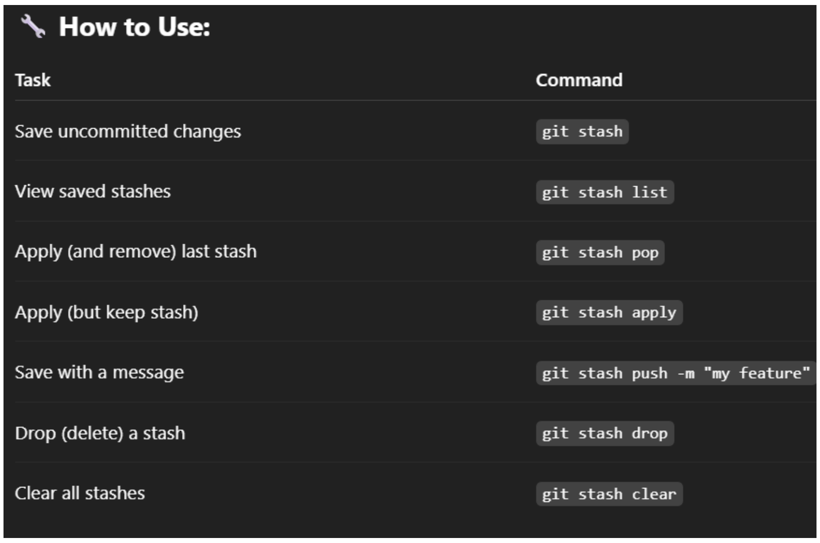
Instead of committing half-done code, you can:

```bash
git stash
```

This saves your changes and gives you a **clean working directory**.

Then, you can switch branches, fix the bug, come back, and **restore your work** later using:

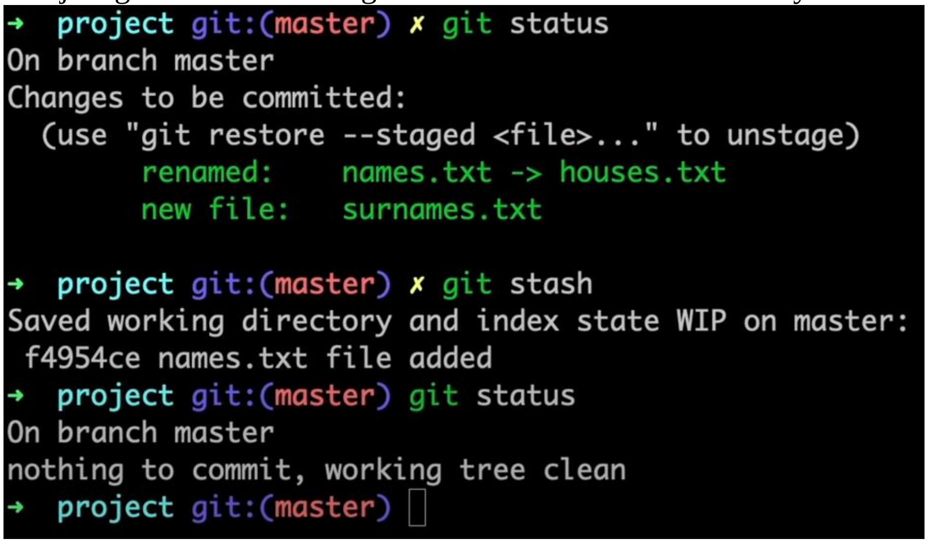
```bash
git stash pop
```



🔧 **How to Use:**

Task	Command
Save uncommitted changes	`git stash`
View saved stashes	`git stash list`
Apply (and remove) last stash	`git stash pop`
Apply (but keep stash)	`git stash apply`
Save with a message	`git stash push -m "my feature"`
Drop (delete) a stash	`git stash drop`
Clear all stashes	`git stash clear`

If you do not want to lose your current status of the git and also do not want that changes in your repo You just saying like just go a back to the stage and when I need I will call for you



```
→  project git:(master) ✗ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    names.txt -> houses.txt
        new file:   surnames.txt

→  project git:(master) ✗ git stash
Saved working directory and index state WIP on master:
 f4954ce names.txt file added
→  project git:(master) git status
On branch master
nothing to commit, working tree clean
→  project git:(master) 
```

12) for calling stash in project folder

Command: git stash pop

13)if you want to clear or remove the stash which you have stored then

Command: git stash clear

14) to add project on github repo
Command: git remote add origin <link of repo>.git

15)If you want to see the link which is associated with this folder then command is
Command: git remote –v

16)for push code in the repository

Command: git push origin <branch name_master>

17)how to create a branch
Command: git branch <branch name>

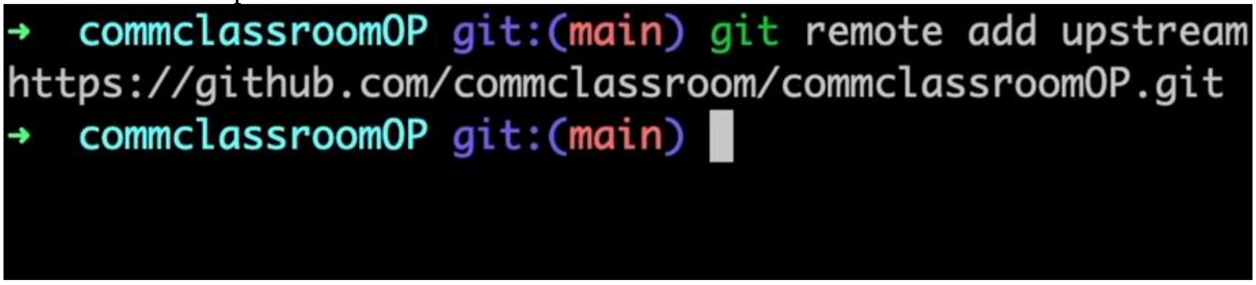
18)how to change the branch
Command: git checkout <branchname>
Means head is changing towards the branch name that you have provided
Note: Whenever you are creating a new branch at that time the new branch is created from your head is currently pointing

Create a new branch **with the given name** and switch to it immediately**.**
git checkout -b feature1

19) How to clone the repository which is already exist
Command: git clone <link>

20)Why we required to clone the repository
Because whenever you are trying to add some code in some organization you did not able to push the code into the directly main repository It is so risky for that organization , so for that first of all you have to fork that repository into your github account after that you can change the code by your own on your repo

21) From where you have forked the repository that is known as upstream url
And how to add upstream url

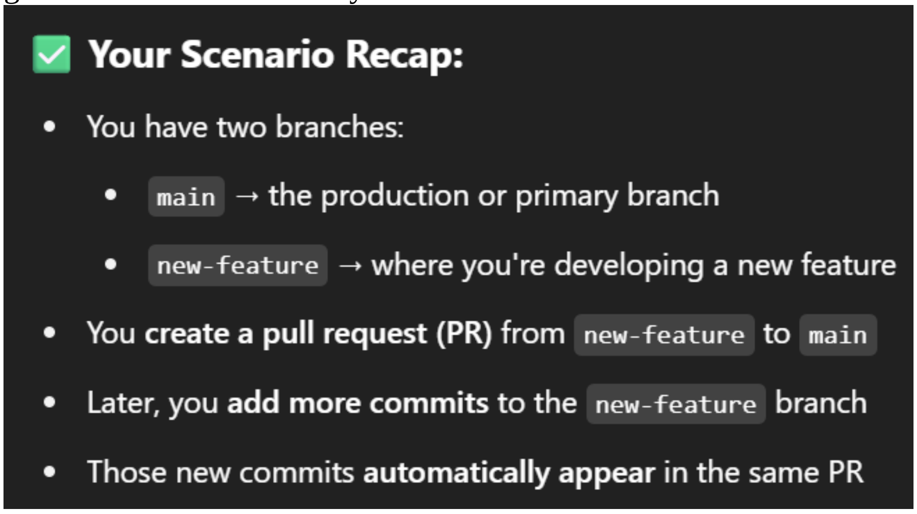


```
→  commclassroomOP git:(main) git remote add upstream
https://github.com/commclassroom/commclassroomOP.git
→  commclassroomOP git:(main)
```

22) If you have created a type of branch separately and add one feature on that and you want to merge that feature on the main branch then how can you do it

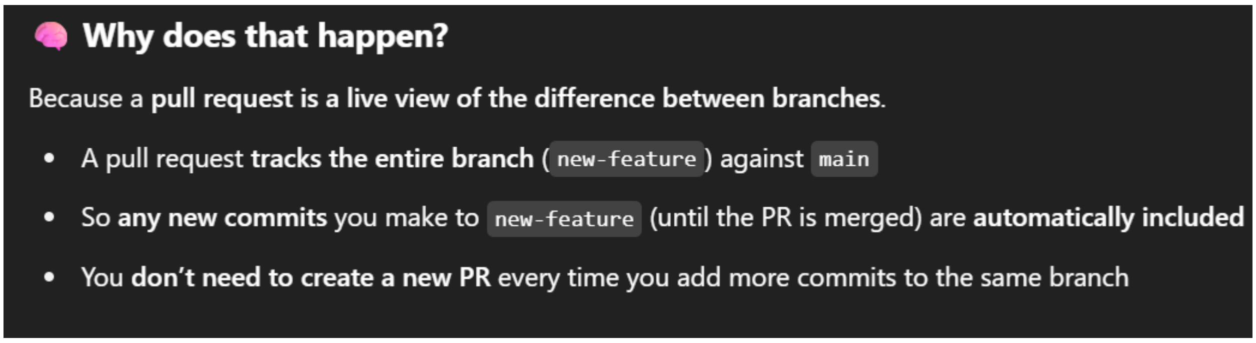
For that you need to create pr request

Note: If one branch is if one branch is <main> and another branch is <new feature> then if <new feature> branch has made a request to push the code onto the main branch only one time pull request has been created And after that again if you commit some other things onto the <new feature> branch at that time new pull request is not being generated it is automatically added into the main branch



✅ Your Scenario Recap:

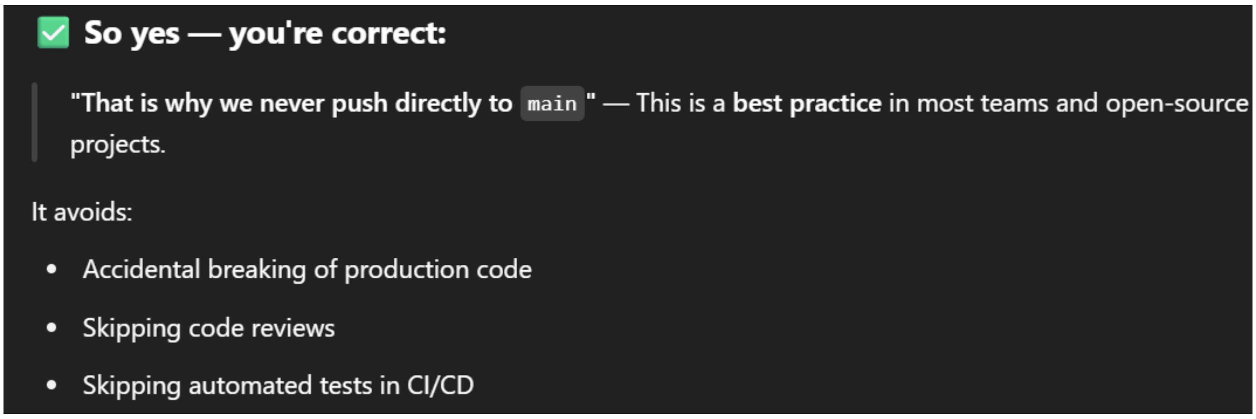
- You have two branches:
  - `main` → the production or primary branch
  - `new-feature` → where you're developing a new feature
- **You create a pull request (PR)** from `new-feature` to `main`
- Later, you **add more commits** to the `new-feature` branch
- Those new commits **automatically appear** in the same PR



🧠 Why does that happen?

Because a **pull request is a live view of the difference between branches.**

- A pull request **tracks the entire branch** (`new-feature`) against `main`
- So **any new commits** you make to `new-feature` (until the PR is merged) are **automatically included**
- You **don't need to create a new PR** every time you add more commits to the same branch



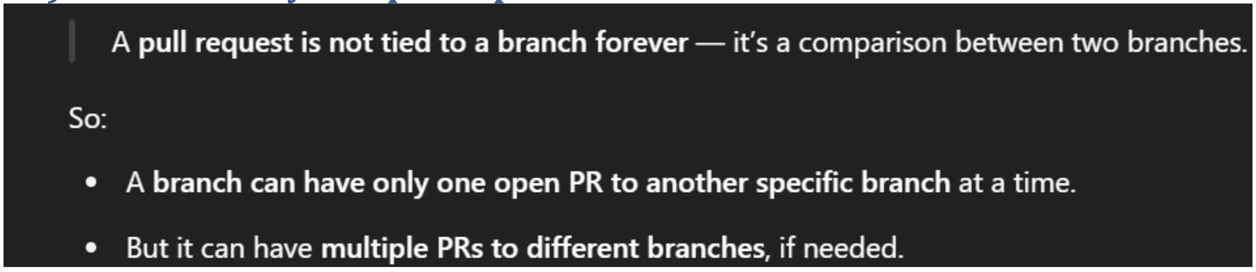
✅ So yes — you're correct:

> **"That is why we never push directly to `main`"** — This is a **best practice** in most teams and open-source projects.

It avoids:

- Accidental breaking of production code
- Skipping code reviews
- Skipping automated tests in CI/CD

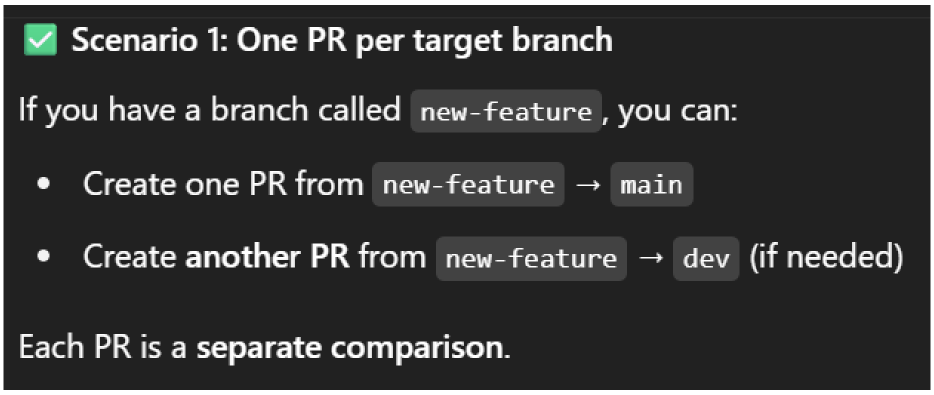
23) At a time only one pr is open



> **A pull request is not tied to a branch forever** — it's a comparison between two branches.

So:

- **A branch can have only one open PR to another specific branch** at a time.
- But it can have **multiple PRs to different branches**, if needed.

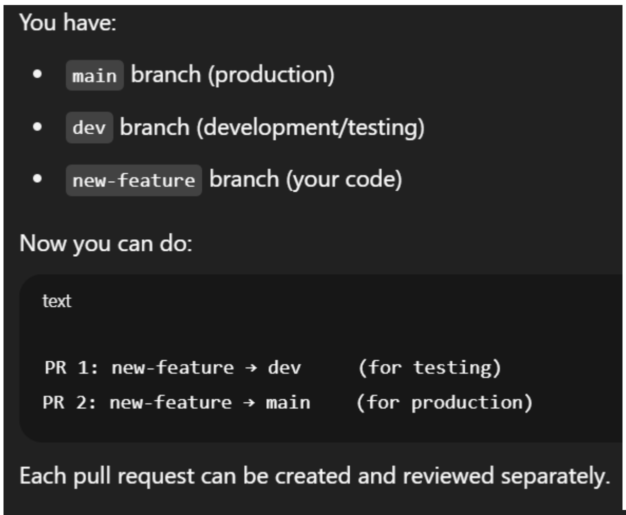


✅ **Scenario 1: One PR per target branch**

If you have a branch called `new-feature`, you can:

- Create one PR from `new-feature` → `main`
- Create **another** PR from `new-feature` → `dev` (if needed)

Each PR is a **separate comparison**.



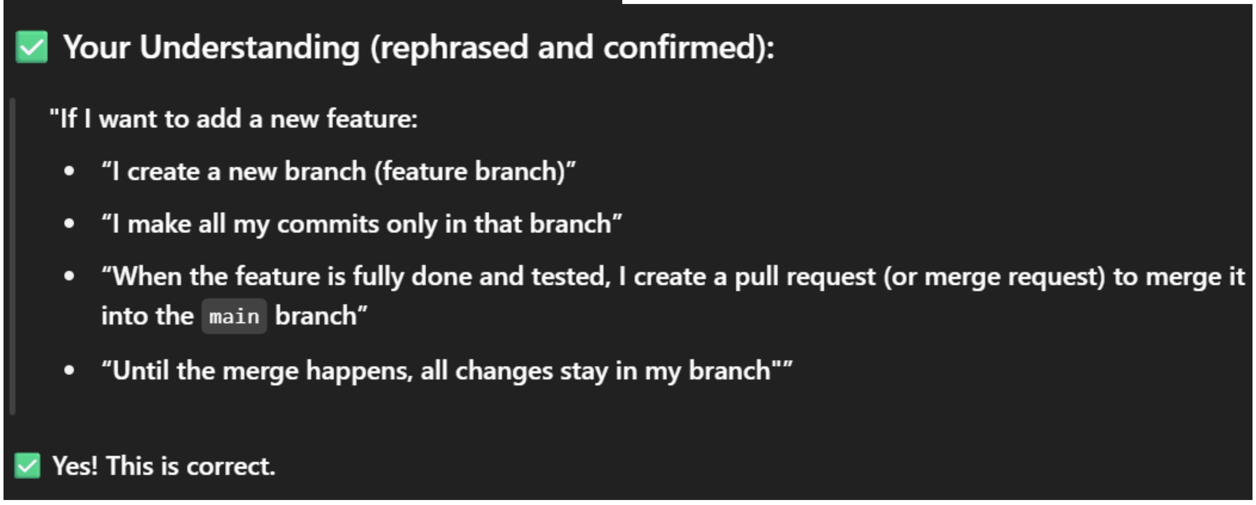
You have:

- `main` branch (production)
- `dev` branch (development/testing)
- `new-feature` branch (your code)

Now you can do:

```text
PR 1: new-feature → dev      (for testing)
PR 2: new-feature → main     (for production)
```

Each pull request can be created and reviewed separately.

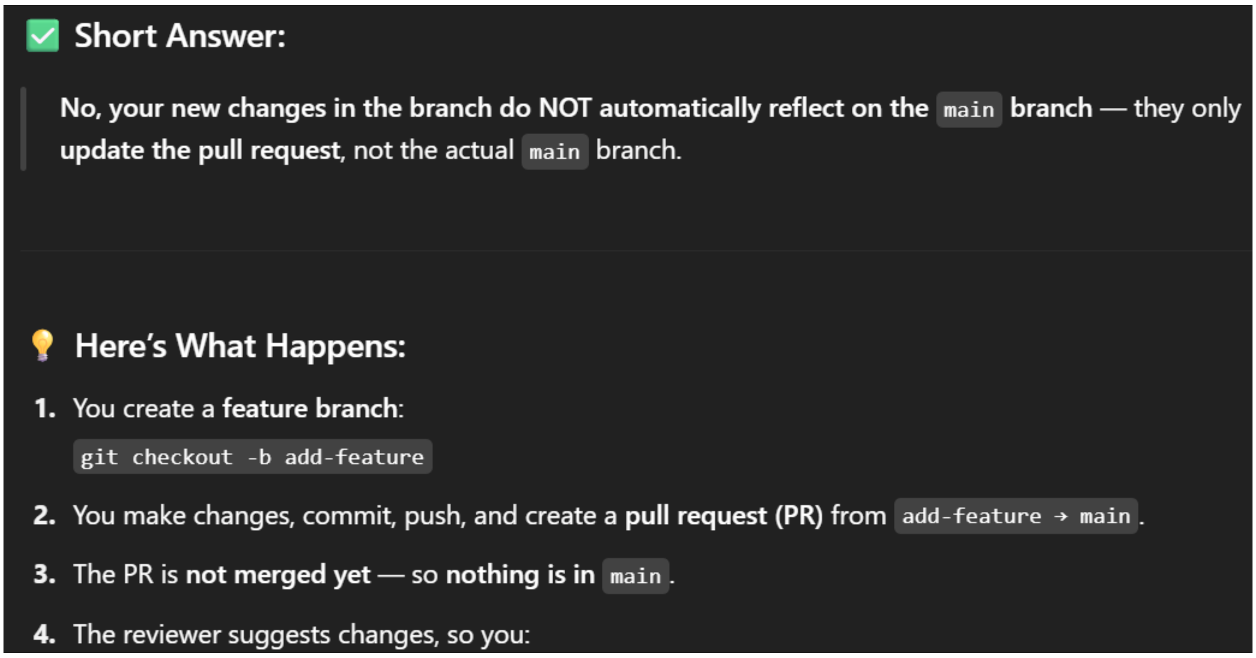


✅ **Your Understanding (rephrased and confirmed):**

> **"If I want to add a new feature:**
>
> - **"I create a new branch (feature branch)"**
> - **"I make all my commits only in that branch"**
> - **"When the feature is fully done and tested, I create a pull request (or merge request) to merge it into the `main` branch"**
> - **"Until the merge happens, all changes stay in my branch""**

✅ **Yes! This is correct.**

24) But I have confused in that if I feel like I have completed my feature and after the code reviewer says that you have to add these things and I have to add those things into the my branch but I have already created one pr for pushing my code already with the main branch so if I commit the new changes in my branch that is also reflected on the main branch?

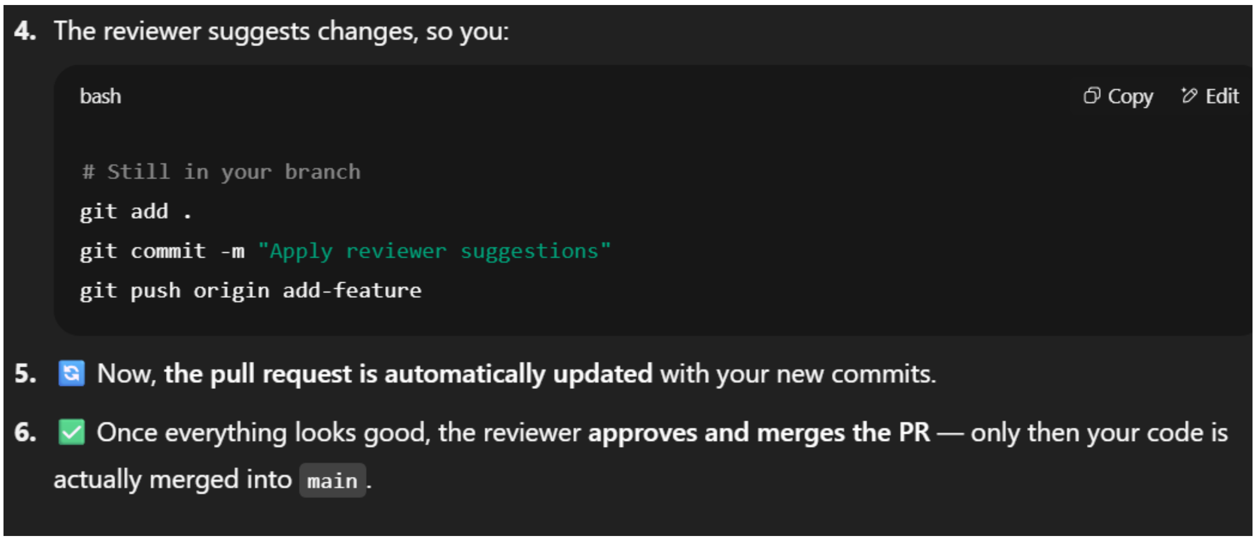


✅ Short Answer:

> **No, your new changes in the branch do NOT automatically reflect on the `main` branch** — they only **update the pull request**, not the actual `main` branch.

💡 Here's What Happens:

1. You create a **feature branch**:
   `git checkout -b add-feature`
2. You make changes, commit, push, and create a **pull request (PR)** from `add-feature → main`.
3. The PR is **not merged yet** — so **nothing is in** `main`.
4. The reviewer suggests changes, so you:

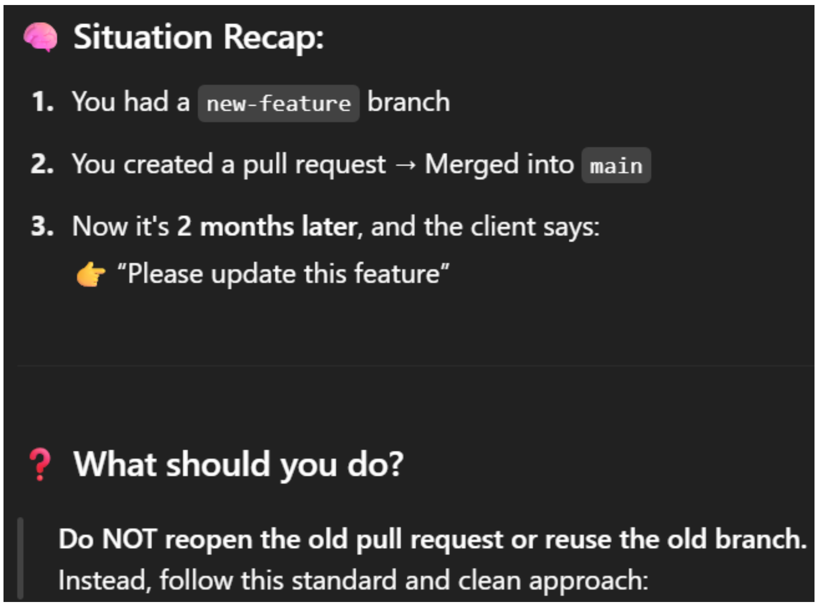


4. The reviewer suggests changes, so you:

```bash
# Still in your branch
git add .
git commit -m "Apply reviewer suggestions"
git push origin add-feature
```

5. 🔄 Now, **the pull request is automatically updated** with your new commits.
6. ✅ Once everything looks good, the reviewer **approves and merges the PR** — only then your code is actually merged into `main`.

25) merging main branch with my feature branch and after two months the client says that I want these changes in this feature so I have already created pr previously I have just explained then I need to change the code or change the new feature then in that case what should be done

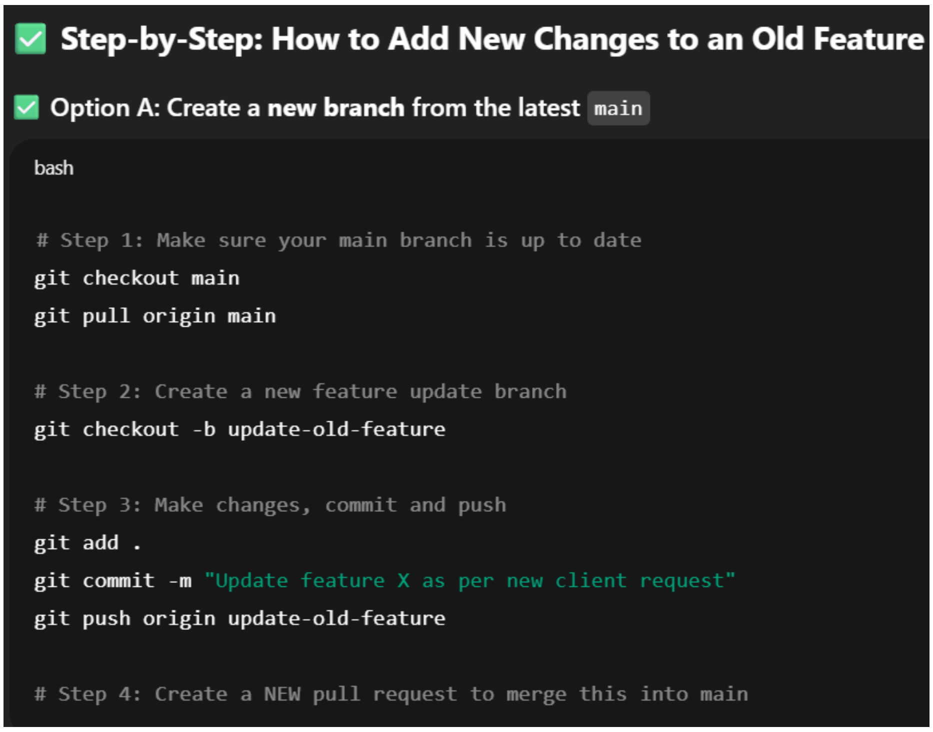


🧠 Situation Recap:

1. You had a `new-feature` branch
2. You created a pull request → Merged into `main`
3. Now it's **2 months later**, and the client says:
   👉 "Please update this feature"

❓ What should you do?

> **Do NOT reopen the old pull request or reuse the old branch.**
> Instead, follow this standard and clean approach:

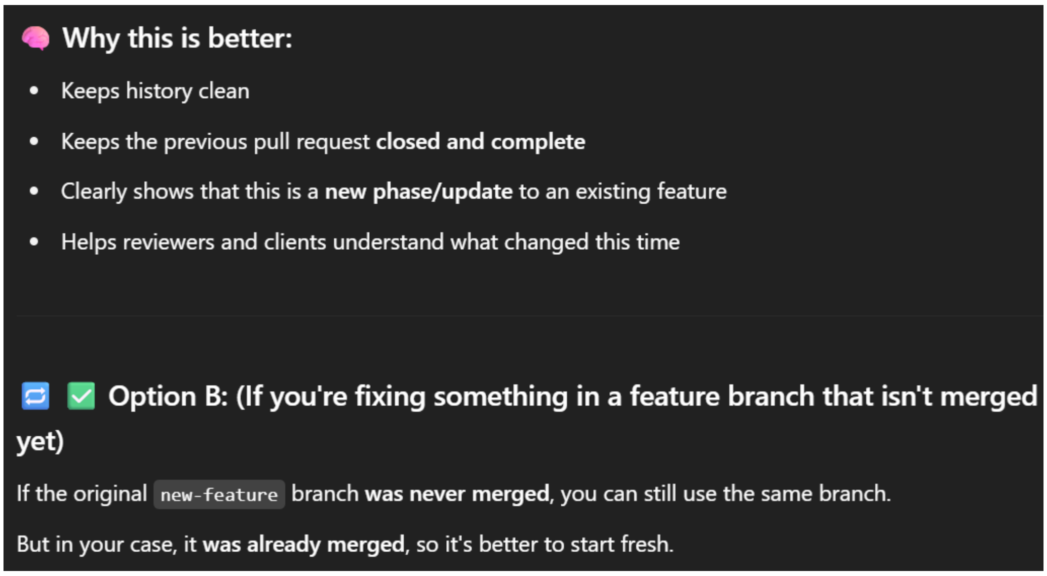


✅ Step-by-Step: How to Add New Changes to an Old Feature

✅ Option A: Create a **new branch** from the latest `main`

bash

```bash
# Step 1: Make sure your main branch is up to date
git checkout main
git pull origin main

# Step 2: Create a new feature update branch
git checkout -b update-old-feature

# Step 3: Make changes, commit and push
git add .
git commit -m "Update feature X as per new client request"
git push origin update-old-feature

# Step 4: Create a NEW pull request to merge this into main
```



🧠 Why this is better:

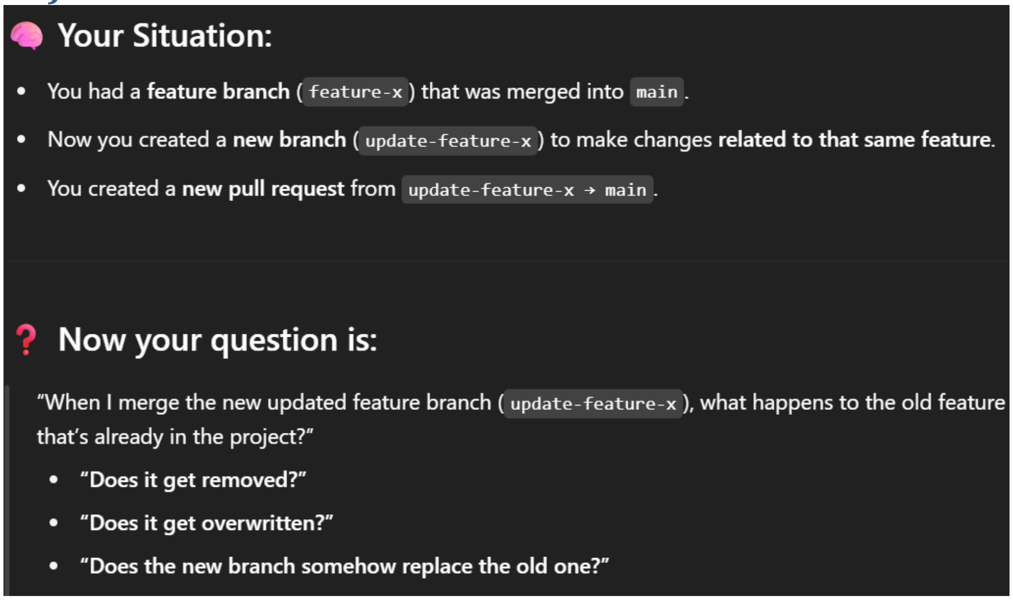
- Keeps history clean
- Keeps the previous pull request **closed and complete**
- Clearly shows that this is a **new phase/update** to an existing feature
- Helps reviewers and clients understand what changed this time

🔁 ✅ Option B: (If you're fixing something in a feature branch that isn't merged yet)

If the original `new-feature` branch **was never merged**, you can still use the same branch.

But in your case, it **was already merged**, so it's better to start fresh.

26)

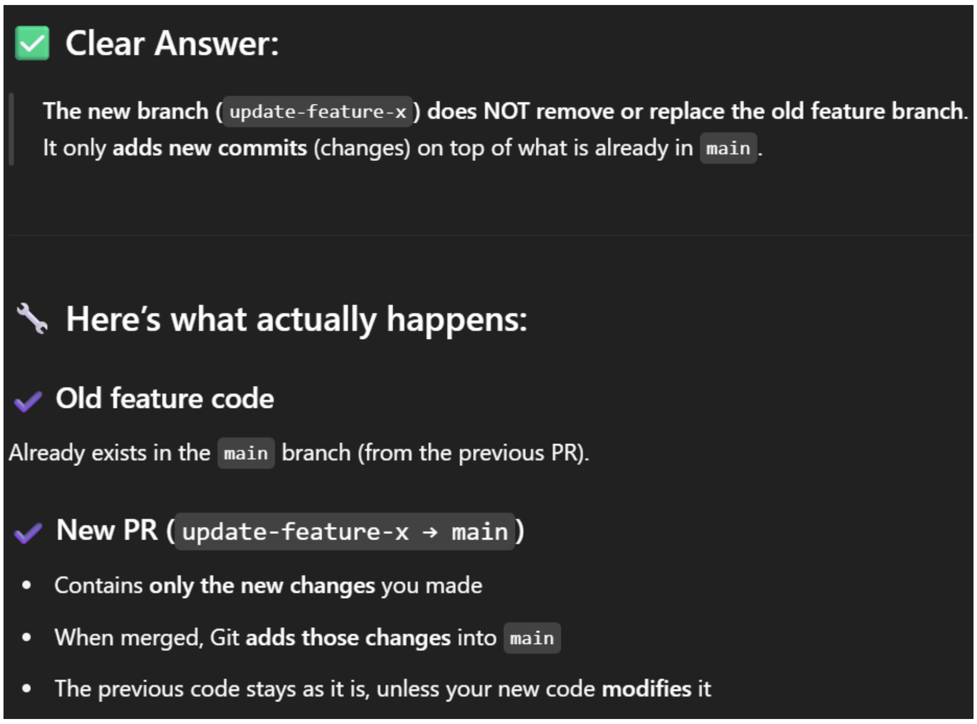


🧠 Your Situation:

- You had a **feature branch** (`feature-x`) that was merged into `main`.
- Now you created a **new branch** (`update-feature-x`) to make changes **related to that same feature**.
- You created a **new pull request** from `update-feature-x → main`.

❓ Now your question is:

> "When I merge the new updated feature branch (`update-feature-x`), what happens to the old feature that's already in the project?"
>
> - **"Does it get removed?"**
> - **"Does it get overwritten?"**
> - **"Does the new branch somehow replace the old one?"**



✅ Clear Answer:

> The new branch (`update-feature-x`) **does NOT remove or replace the old feature branch.**
> It only **adds new commits** (changes) on top of what is already in `main`.

🔧 Here's what actually happens:

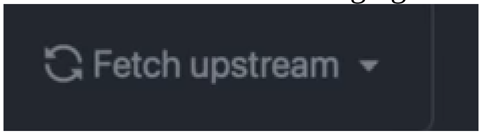
✔ **Old feature code**

Already exists in the `main` branch (from the previous PR).

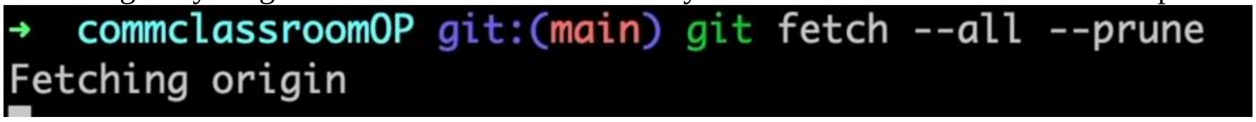
✔ **New PR (`update-feature-x → main`)**

- Contains **only the new changes** you made
- When merged, Git **adds those changes** into `main`
- The previous code stays as it is, unless your new code **modifies** it

27)if update like open source project of kubernetes has accept some request of one of the user then definitely it will not reflect onto your local branch or the repo that you have forked in your machine or github account because you have forked before the merging that code so for that what do you use ?

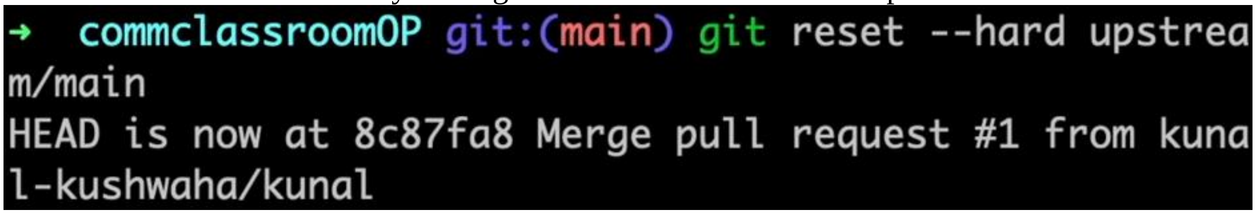


You can go to your github account and do with ui or you can also with do with manual step



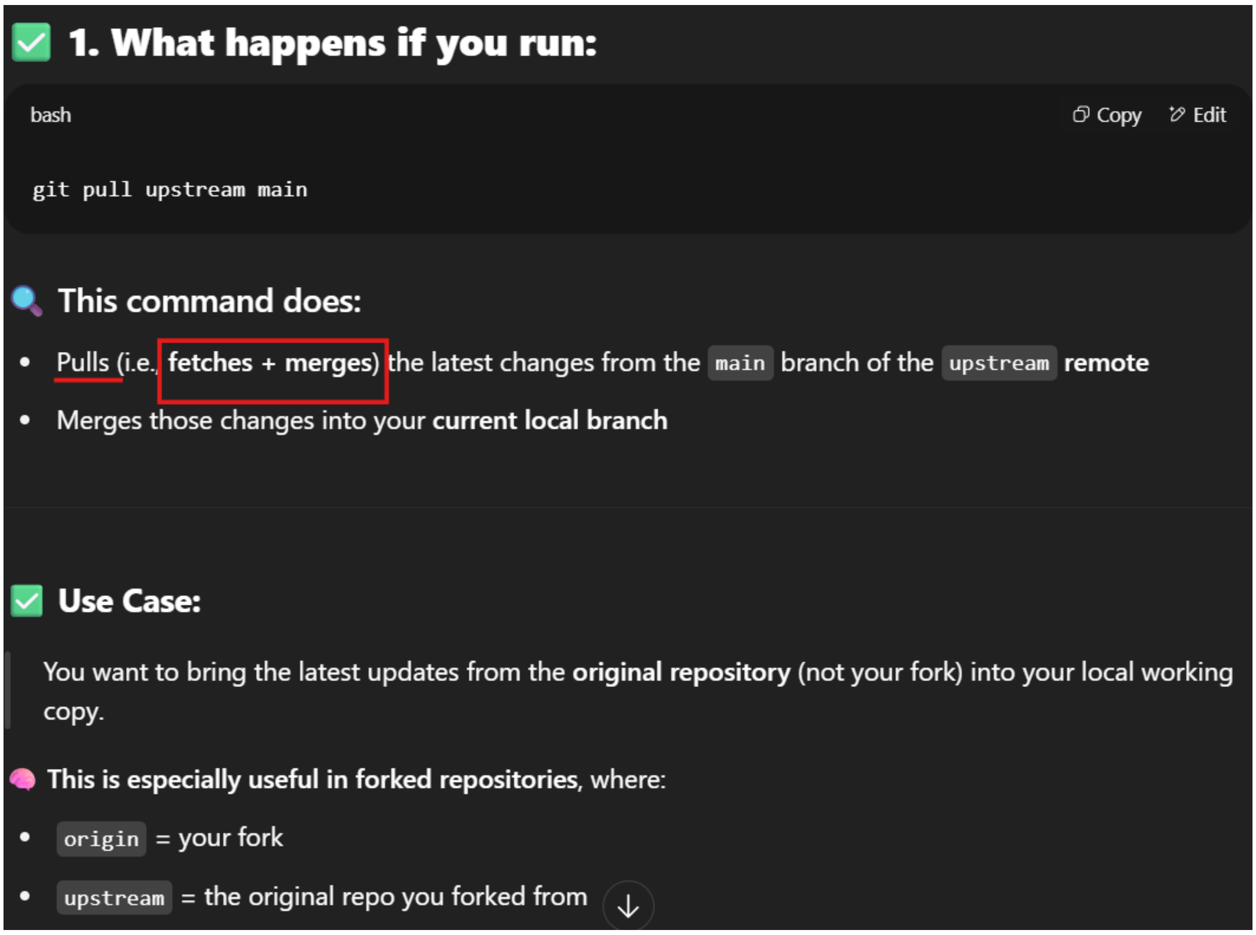
```
➜  commclassroomOP git:(main) git fetch --all --prune
Fetching origin
```

Now reset the main branch of your origin to the main branch of the upstream



```
➜  commclassroomOP git:(main) git reset --hard upstrea
m/main
HEAD is now at 8c87fa8 Merge pull request #1 from kuna
l-kushwaha/kunal
```

You can also runs git pull upstream main but there is a difference between these two commands



✅ **1. What happens if you run:**

```bash
git pull upstream main
```

🔍 **This command does:**

- Pulls (i.e. **fetches + merges**) the latest changes from the `main` branch of the `upstream` **remote**
- Merges those changes into your **current local branch**

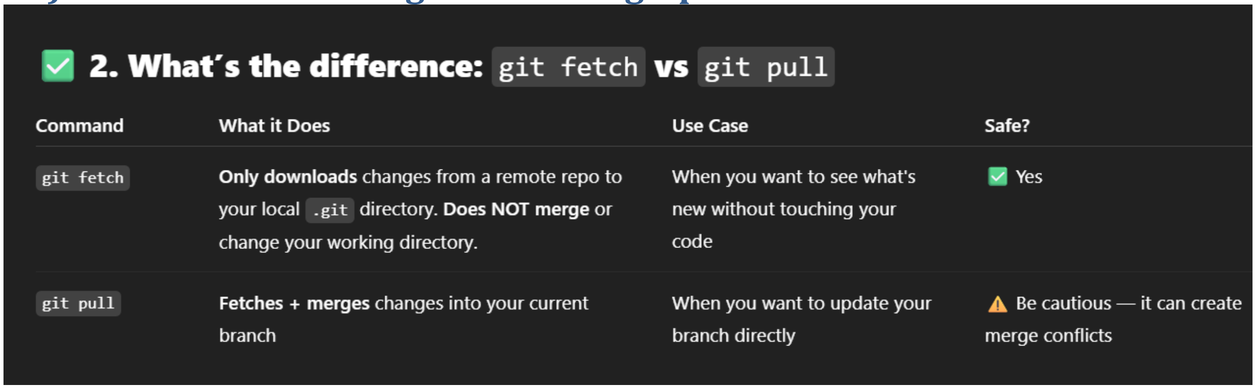
✅ **Use Case:**

> You want to bring the latest updates from the **original repository** (not your fork) into your local working copy.

🧠 **This is especially useful in forked repositories**, where:

- `origin` = your fork
- `upstream` = the original repo you forked from

28)Difference between git fetch and git pull

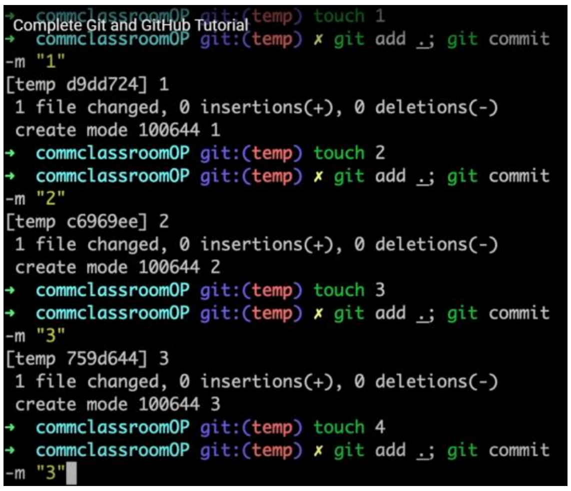


✅ **2. What's the difference: `git fetch` vs `git pull`**

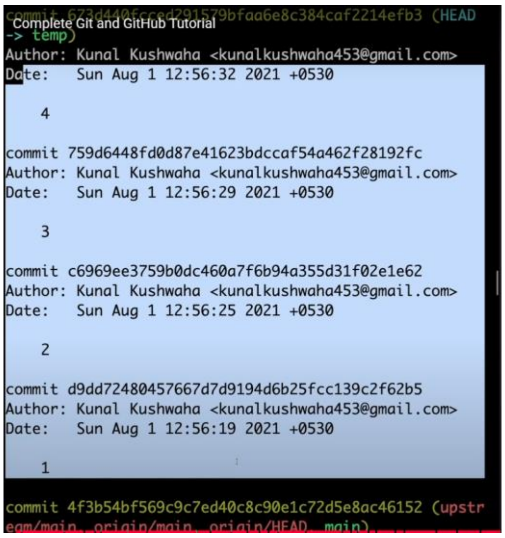
Command	What it Does	Use Case	Safe?
`git fetch`	**Only downloads** changes from a remote repo to your local `.git` directory. **Does NOT merge** or change your working directory.	When you want to see what's new without touching your code	✅ Yes
`git pull`	**Fetches + merges** changes into your current branch	When you want to update your branch directly	⚠️ Be cautious — it can create merge conflicts

29)rebase command

If I have created one file "1" and then commit ,again created file "2" and commit , again created file "3" and commit, again created file "4" and commit
But now I want to merge that all four commits into the one commit then using rebase –i you can use it

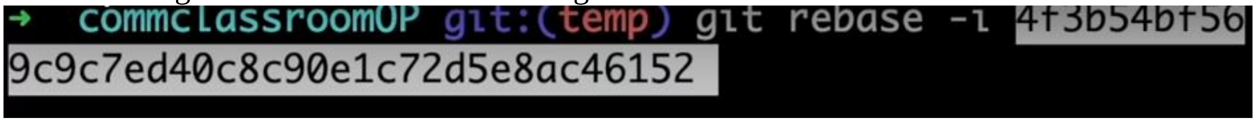


```
→ commclassroomOP git:(temp) touch 1
→ commclassroomOP git:(temp) ✗ git add .; git commit
-m "1"
[temp d9dd724] 1
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 1
→ commclassroomOP git:(temp) touch 2
→ commclassroomOP git:(temp) ✗ git add .; git commit
-m "2"
[temp c6969ee] 2
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 2
→ commclassroomOP git:(temp) touch 3
→ commclassroomOP git:(temp) ✗ git add .; git commit
-m "3"
[temp 759d644] 3
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 3
→ commclassroomOP git:(temp) touch 4
→ commclassroomOP git:(temp) ✗ git add .; git commit
-m "3"
```

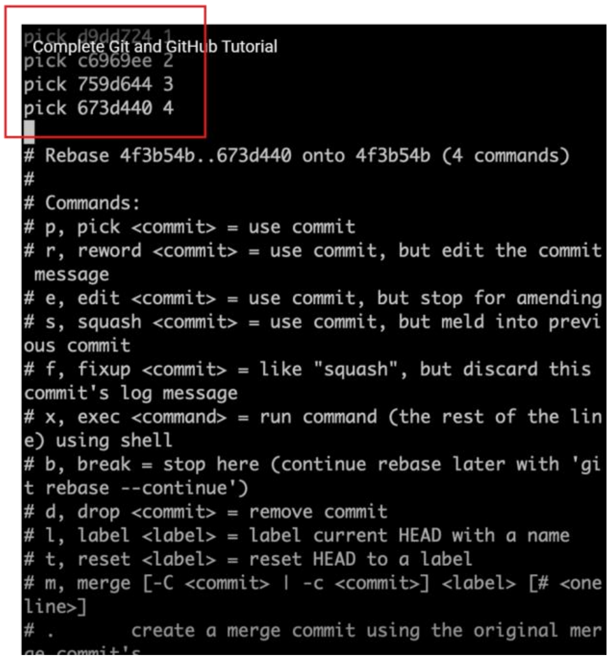


```
commit 673d440fcced291579bfaa6e8c384caf2214efb3 (HEAD
-> temp)
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 12:56:32 2021 +0530

    4

commit 759d6448fd0d87e41623bdccaf54a462f28192fc
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 12:56:29 2021 +0530

    3

commit c6969ee3759b0dc460a7f6b94a355d31f02e1e62
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 12:56:25 2021 +0530

    2

commit d9dd72480457667d7d9194d6b25fcc139c2f62b5
Author: Kunal Kushwaha <kunalkushwaha453@gmail.com>
Date:   Sun Aug 1 12:56:19 2021 +0530

    1

commit 4f3b54bf569c9c7ed40c8c90e1c72d5e8ac46152 (upstr
eam/main, origin/main, origin/HEAD, main)
```

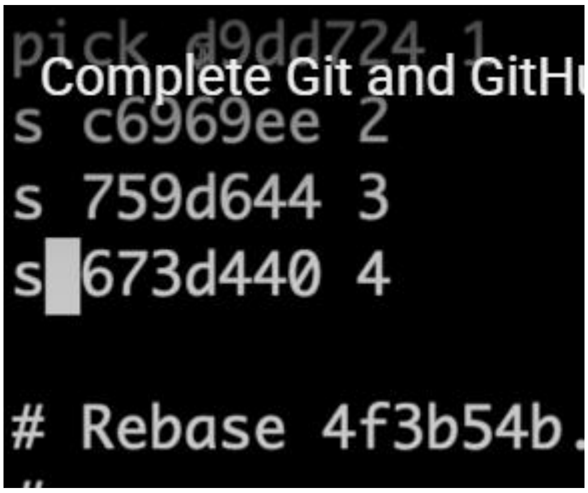
I want to merge all these 4 commit into a single commit



```
→ commclassroomOP git:(temp) git rebase -i 4f3b54bf56
9c9c7ed40c8c90e1c72d5e8ac46152
```



```
pick d9dd724 1
pick c6969ee 2
pick 759d644 3
pick 673d440 4

# Rebase 4f3b54b..673d440 onto 4f3b54b (4 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit
 message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previ
ous commit
# f, fixup <commit> = like "squash", but discard this
commit's log message
# x, exec <command> = run command (the rest of the lin
e) using shell
# b, break = stop here (continue rebase later with 'gi
t rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <one
line>]
# .       create a merge commit using the original mer
ge commit's
```



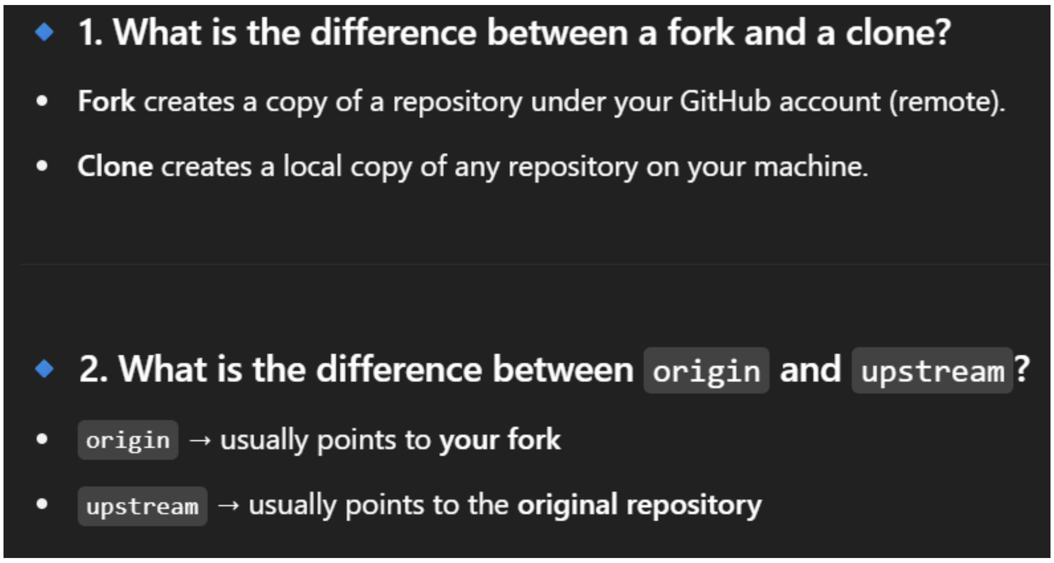
```
pick d9dd724 1
s c6969ee 2
s 759d644 3
s 673d440 4

# Rebase 4f3b54b.
```

Then change the pick here to squash and update the code like in the ss
Squash means about a squash commit merge into that pick commit
Means here it is three squash has been merged in first pick

30)merging conflicts

These conflict often happen when if one user has changed the code in line number 3 and another people have also changed the code in line number three at that time git will ask you to help which changes should be done

Another screenshots of the interview questions



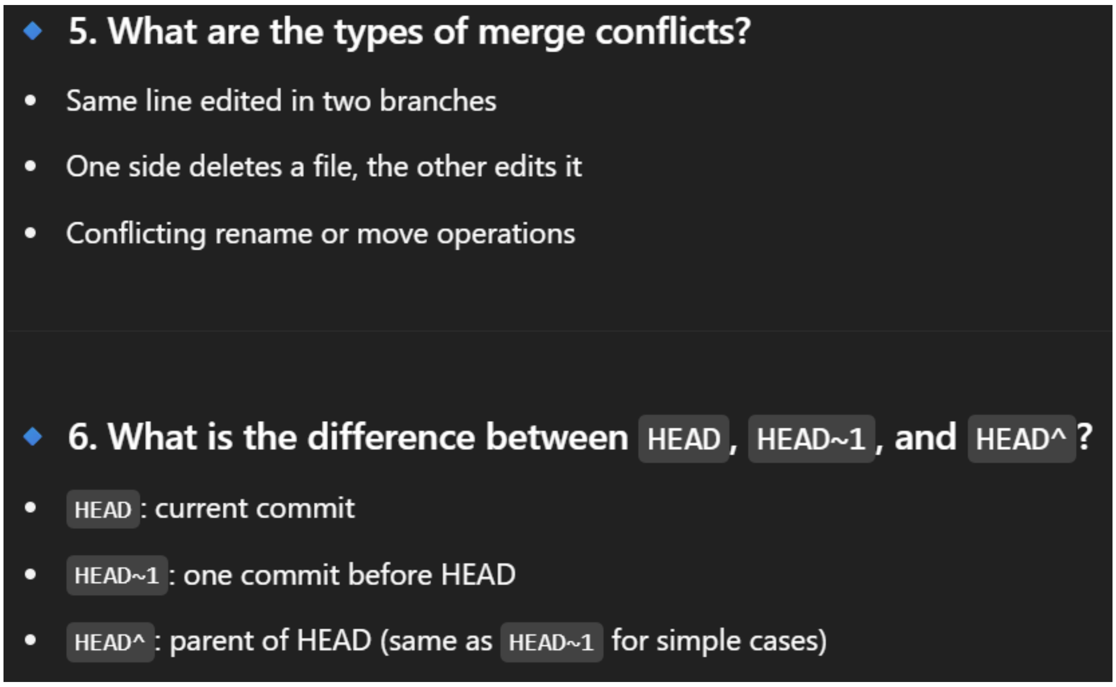
1. What is the difference between a fork and a clone?

- **Fork** creates a copy of a repository under your GitHub account (remote).
- **Clone** creates a local copy of any repository on your machine.

---

2. What is the difference between `origin` and `upstream`?

- `origin` → usually points to **your fork**
- `upstream` → usually points to the **original repository**



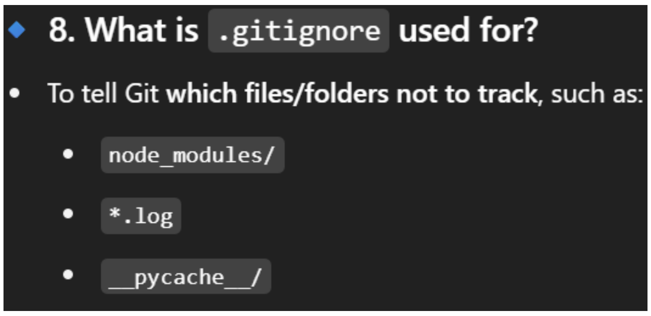
5. What are the types of merge conflicts?

- Same line edited in two branches
- One side deletes a file, the other edits it
- Conflicting rename or move operations

---

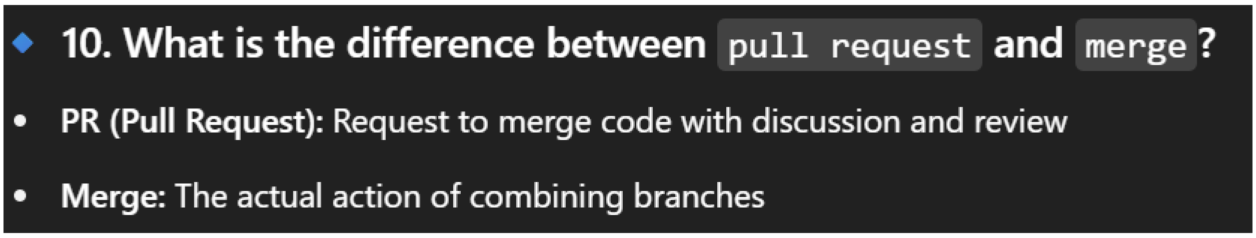
6. What is the difference between `HEAD`, `HEAD~1`, and `HEAD^`?

- `HEAD`: current commit
- `HEAD~1`: one commit before HEAD
- `HEAD^`: parent of HEAD (same as `HEAD~1` for simple cases)



8. What is `.gitignore` used for?

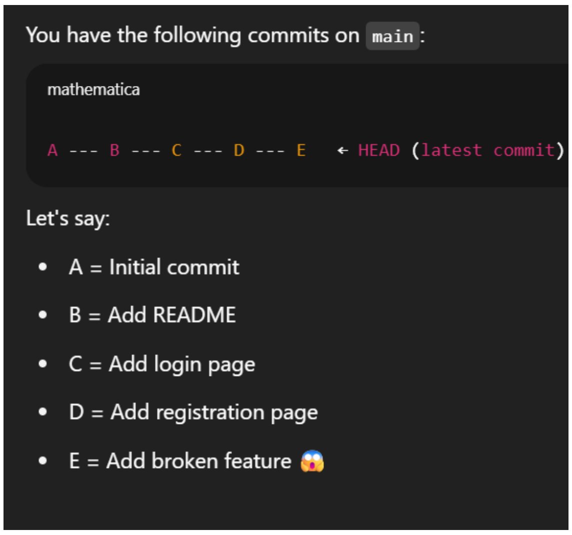
- To tell Git **which files/folders not to track**, such as:
  - `node_modules/`
  - `*.log`
  - `__pycache__/`



10. What is the difference between `pull request` and `merge`?

- **PR (Pull Request):** Request to merge code with discussion and review
- **Merge:** The actual action of combining branches

Example of git reset vs git revert

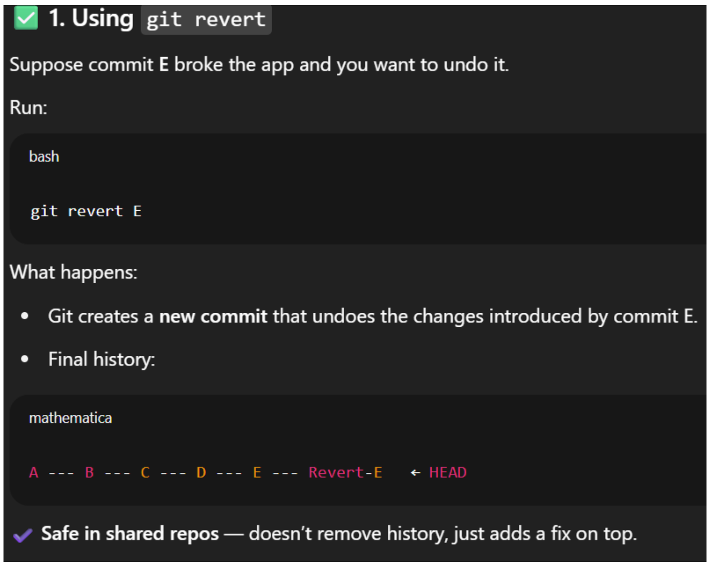


You have the following commits on `main`:

```mathematica
A --- B --- C --- D --- E   ← HEAD (latest commit)
```

Let's say:

- A = Initial commit
- B = Add README
- C = Add login page
- D = Add registration page
- E = Add broken feature 😱



✅ **1. Using `git revert`**

Suppose commit E broke the app and you want to undo it.

Run:

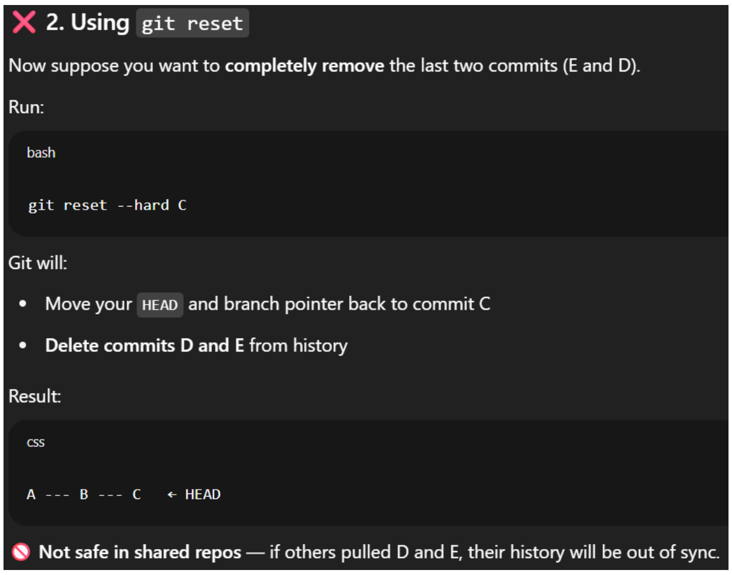
```bash
git revert E
```

What happens:

- Git creates a **new commit** that undoes the changes introduced by commit E.
- Final history:

```mathematica
A --- B --- C --- D --- E --- Revert-E   ← HEAD
```

✔ **Safe in shared repos** — doesn't remove history, just adds a fix on top.



❌ **2. Using `git reset`**

Now suppose you want to **completely remove** the last two commits (E and D).

Run:

```bash
git reset --hard C
```

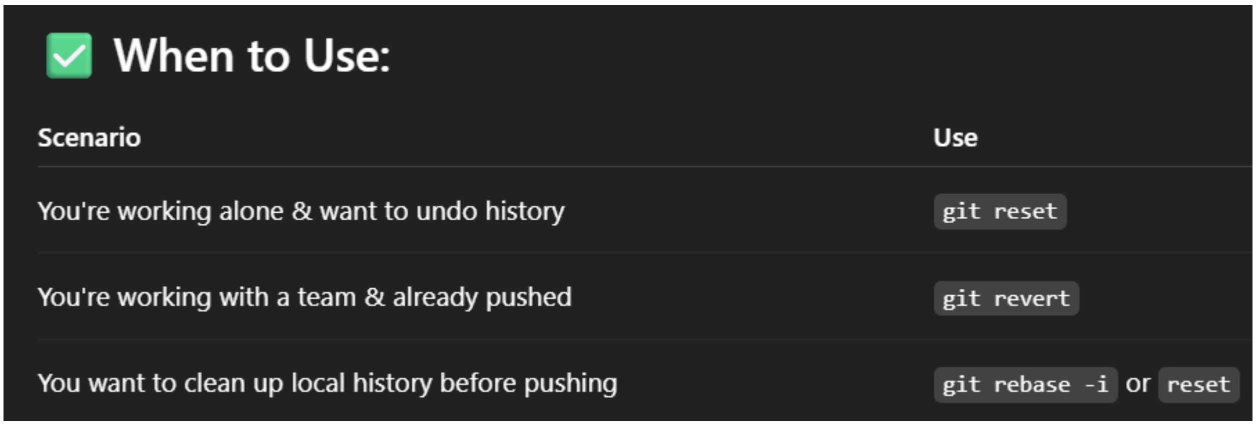
Git will:

- Move your `HEAD` and branch pointer back to commit C
- **Delete commits D and E** from history

Result:

```css
A --- B --- C   ← HEAD
```

🚫 **Not safe in shared repos** — if others pulled D and E, their history will be out of sync.



✅ **When to Use:**

Scenario	Use
You're working alone & want to undo history	`git reset`
You're working with a team & already pushed	`git revert`
You want to clean up local history before pushing	`git rebase -i` or `reset`

Chat link of chatgpt of git and github manually:
https://chatgpt.com/share/6854f029-3ebc-8002-bbe6-2e0292d127ba

git cheatsheet ss

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`	set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`	set an email address that will be associated with each history marker
`git config --global color.ui auto`	set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`	initialize an existing directory as a Git repository
`git clone [url]`	retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`	show modified files in working directory, staged for your next commit
`git add [file]`	add a file as it looks now to your next commit (stage)
`git reset [file]`	unstage a file while retaining the changes in working directory
`git diff`	diff of what is changed but not staged
`git diff --staged`	diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`	commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`	list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`	create a new branch at the current commit
`git checkout`	switch to another branch and check it out into your working directory
`git merge [branch]`	merge the specified branch's history into the current one
`git log`	show all commits in the current branch's history

INSPECT & COMPARE

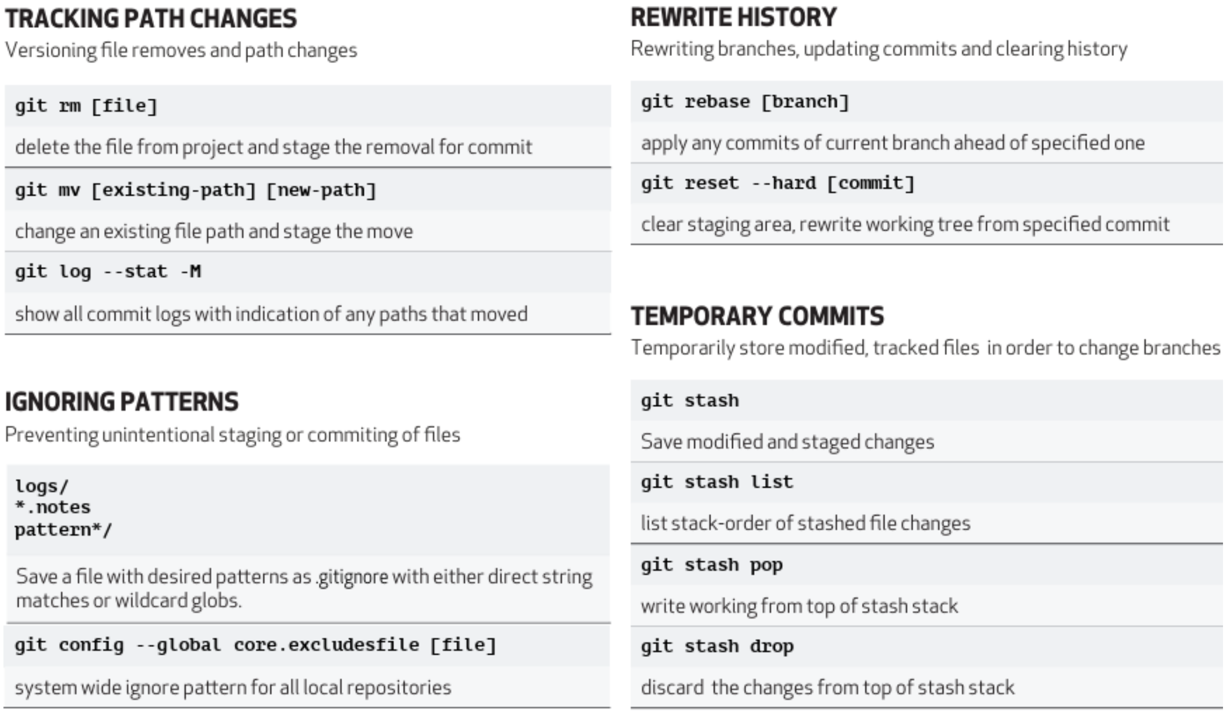
Examining logs, diffs and object information

`git log`	show the commit history for the currently active branch
`git log branchB..branchA`	show the commits on branchA that are not on branchB
`git log --follow [file]`	show the commits that changed file, even across renames
`git diff branchB...branchA`	show the diff of what is in branchA that is not in branchB
`git show [SHA]`	show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`	add a git URL as an alias
`git fetch [alias]`	fetch down all the branches from that Git remote
`git merge [alias]/[branch]`	merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`	Transmit local branch commits to the remote repository branch
`git pull`	fetch and merge any commits from the tracking remote branch



TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`	delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`	change an existing file path and stage the move
`git log --stat -M`	show all commit logs with indication of any paths that moved

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`	system wide ignore pattern for all local repositories

REWRITE HISTORY

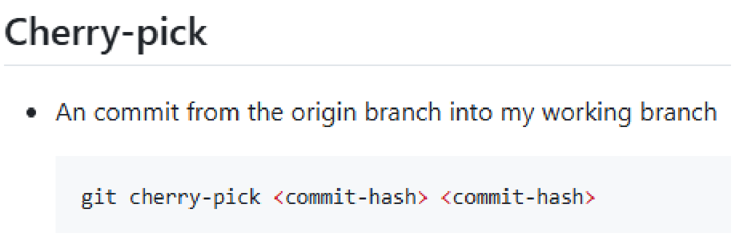
Rewriting branches, updating commits and clearing history

`git rebase [branch]`	apply any commits of current branch ahead of specified one
`git reset --hard [commit]`	clear staging area, rewrite working tree from specified commit

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`	Save modified and staged changes
`git stash list`	list stack-order of stashed file changes
`git stash pop`	write working from top of stash stack
`git stash drop`	discard the changes from top of stash stack



Cherry-pick

- An commit from the origin branch into my working branch

```
git cherry-pick <commit-hash> <commit-hash>
```

Git commands link:

https://github.com/Bhavya2520/DevOps-Learning/blob/main/docs/git/commands.md