Git Commands Explained

This presentation provides a **visual guide** to essential Git commands, complete with flowcharts and illustrations, making it easy to understand and reference for learners and developers alike.

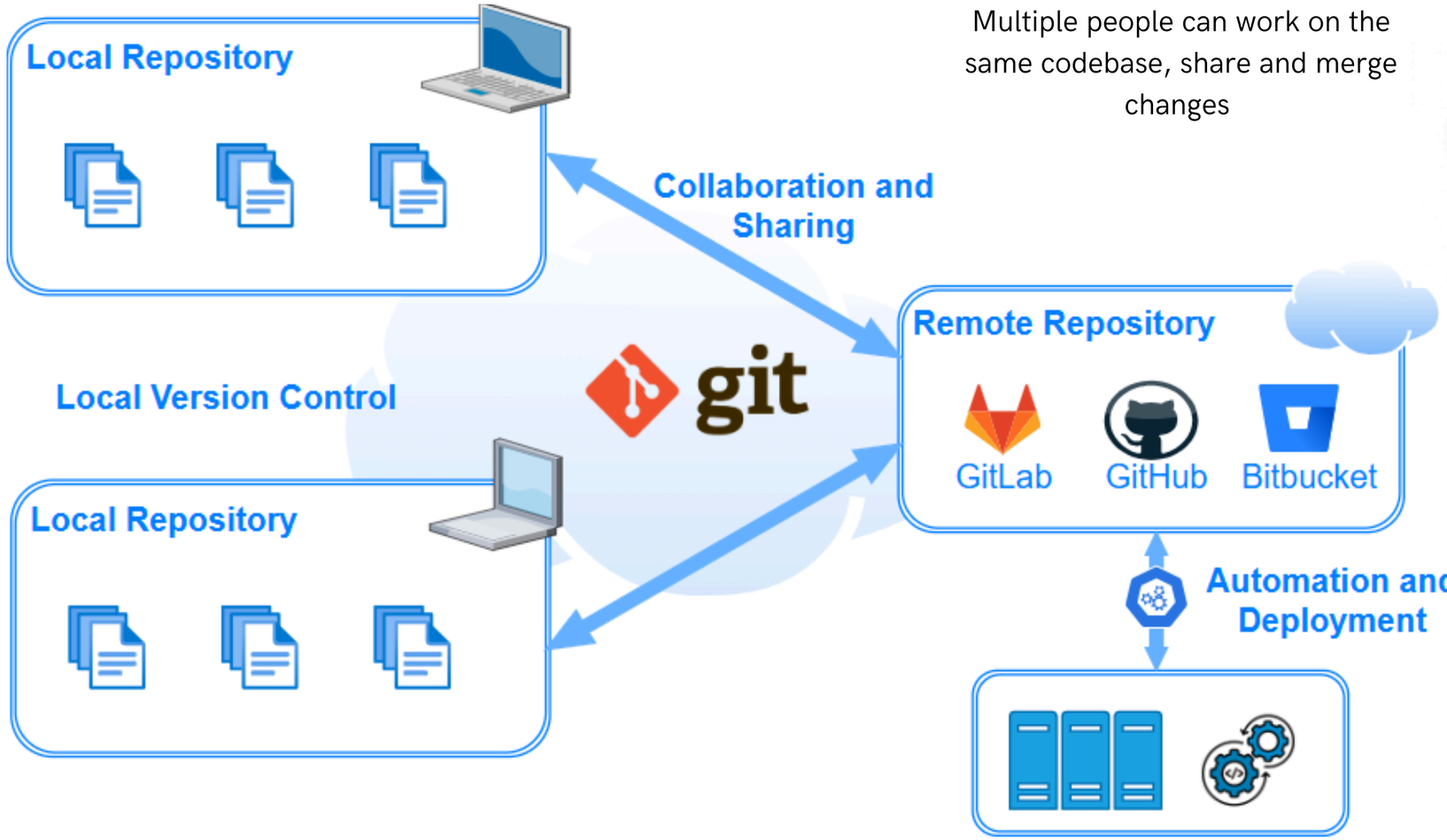
NOVEMBER 2025

BY WALID MALLAT

What is git?

Git is a free and open-source distributed version control system

Track changes, commit history, revert, etc.



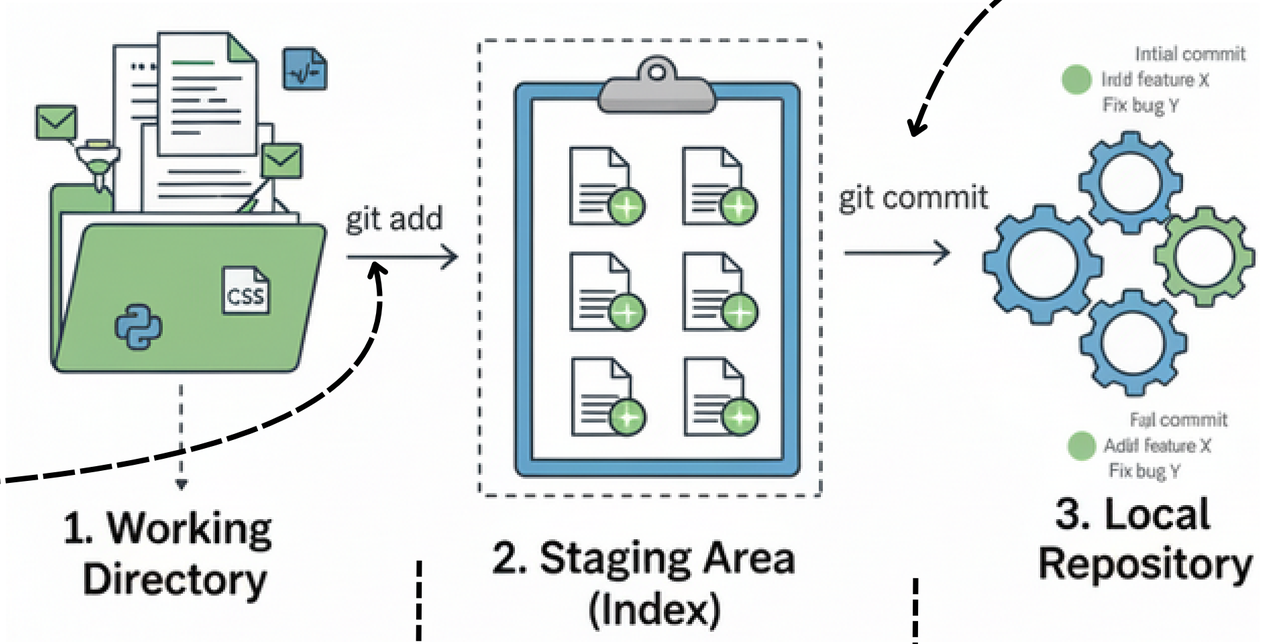
Multiple people can work on the same codebase, share and merge changes

Remote platforms often include or integrate tools for issue tracking, CI/CD, workflow automation

How to manage versions in git?

Git tracks versions only for committed files, not every single change.

Use **git add** to stage changes you want to include in the next commit.



A commit in Git is like a snapshot of a set of files at a given point in time.

This is where you edit files.

It corresponds to the project directory structure you see.

Acts as a buffer to prepare files for commit.

When you run **git commit**, staged changes are stored here.

This is your local version history (on your machine).

How does collaboration work in git?

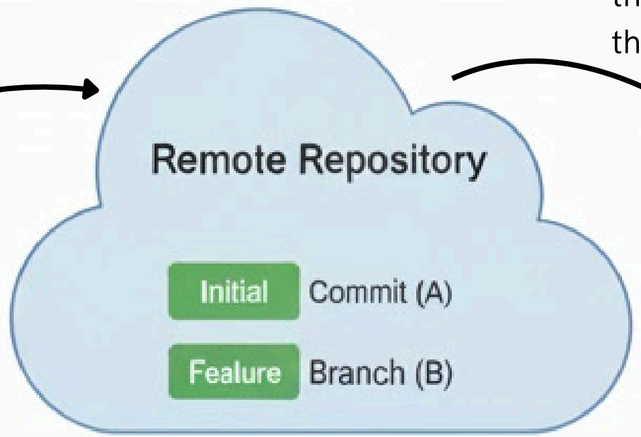
1 Team Member 1 writes code locally and commits to their repository.

2 Team Member 1 pushes the commit to the remote repository.

3 Team Member 1 shares access to the remote repository with Team Member 2.

4 Team Member 2 clones the remote repository to their local machine.

5 Team Member 2 makes changes, stages them, and commits locally.



Team Member 1's Local Repository

Initial	Commit (A)
Feature	Branch (A)

7 Team Member 1 pulls the new updates to stay synchronized.

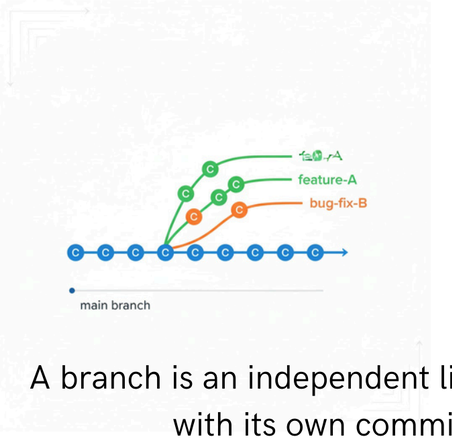
6 Team Member 2 pushes their changes to the remote repository.

Team Member 2's Local Repository

Initial	Commit (A)
Feature	Branch (B)

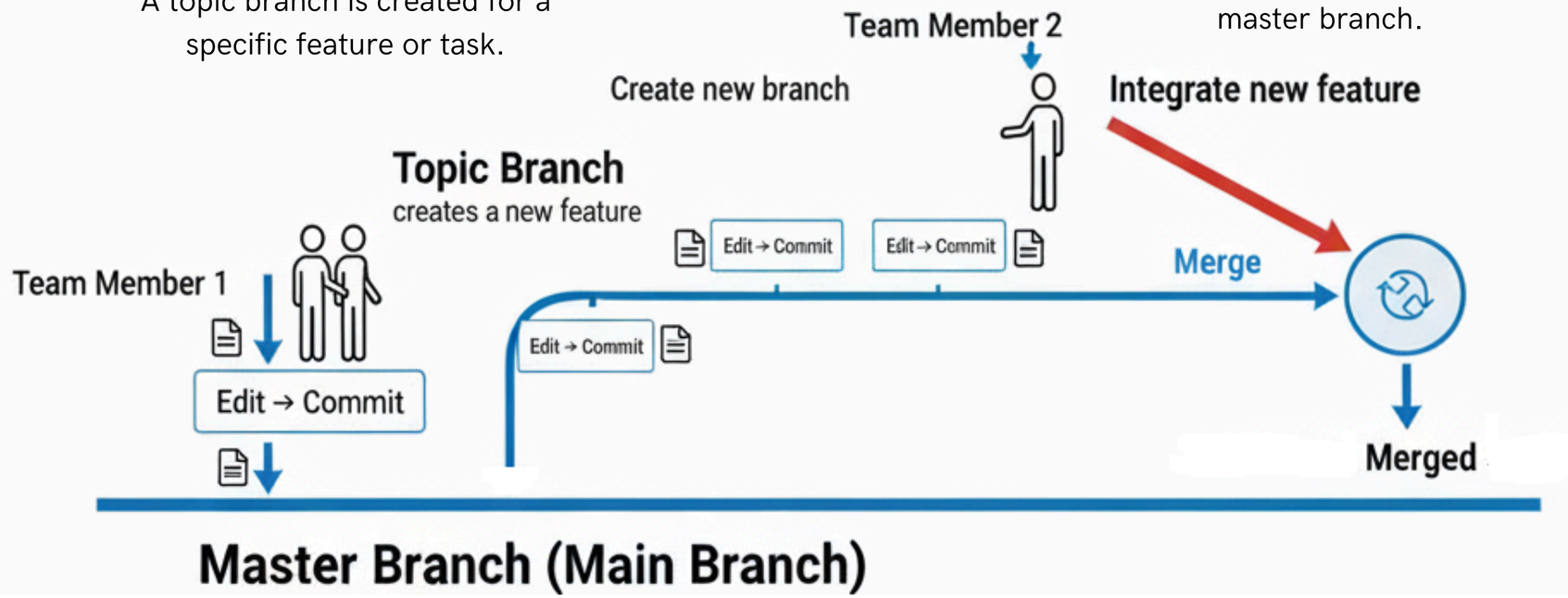
Why use branches in git?

Git starts with a default branch usually called master.



A branch is an independent line of development with its own commit history.

A topic branch is created for a specific feature or task.



Once work is complete, the topic branch is merged back into the master branch.

Branches allow you to work on multiple features or fixes in parallel.

Branching supports efficient collaboration, avoiding interference between developments.

Which tools are needed for git?

1 Git Software

Download the installer from the official Git website.

2 Text Editor

Use VS Code as your main editor.

Run Git Bash inside VS Code for terminal commands.

3 Web Browser

Any browser can be used to access any remote repositories.

Project Setup

How does git know who you are?

🧠 Git needs your identity for every commit:

Every time you make a commit, Git saves who made the change using:

- 👤 Your name
- 📧 Your email address

➡️ This information appears in the commit history and helps teams track contributions.

Git settings are stored at different levels, depending on where you want the configuration to apply.

GIT CONFIG LEVELS

SYSTEM LEVEL --system

- Stored in /etc/gitconfig
- System configs are available for all the users/projects

GLOBAL LEVEL --global

- Stored in ~/.gitconfig
- Global configs are available for all projects for the current user

PROJECT LEVEL default --local

- Stored in .git/config
- Project configs are only available for the current project

What is git config?

🧠 Git needs your identity for every commit

- **git config** is a command used to configure Git settings.
- It allows you to personalize Git with your name, email, and preferred editor.

⚙️ Configure Identity

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
```

The **--global** flag means it applies to all repos.

📝 Set Default Editor

Configure Git to use your preferred text editor for commit messages.

➡️ For Visual Studio Code:

```
$ git config --global core.editor "code --wait"
```

➡️ For Vim:

```
$ git config --global core.editor "vim"
```

➡️ For Nano:

```
$ git config --global core.editor "nano"
```

🧹 Remove or Edit Settings

Check Current Configuration

```
$ git config --global --list
    user.name=Your Name
    user.email=you@example.com
    core.editor=code --wait
```

Unset or Change Settings

```
$ git config --global --unset user.name
$ git config --global --unset user.email
$ git config --global --unset core.editor
```

How do you authenticate in git?

🧩Problem

- When you collaborate on a remote project, Git needs to confirm your identity before allowing you to push, pull, or clone code.
- Without authentication, anyone could modify your repository, which is risky!

Git uses authentication methods to verify your identity safely

🔑 Personal Access Token (PAT)

- A special "password" generated by your Git hosting platform (like GitHub or GitLab).
- It replaces your normal password for secure access via HTTPS.

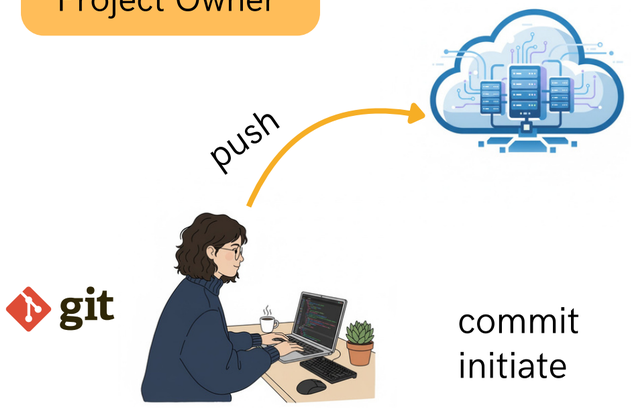
🧭 SSH Key (Secure Shell Key)

- A pair of cryptographic keys (public + private) that lets you connect securely without typing a password.
- Once configured, you can authenticate automatically.

How can you begin working on a project?

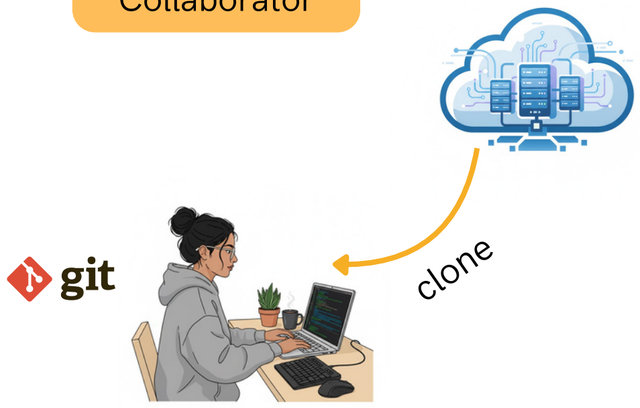
Projects can be created or joined in different ways depending on your role.
There are three main cases: Project Owner, Collaborator, and External Contributor.

Project Owner



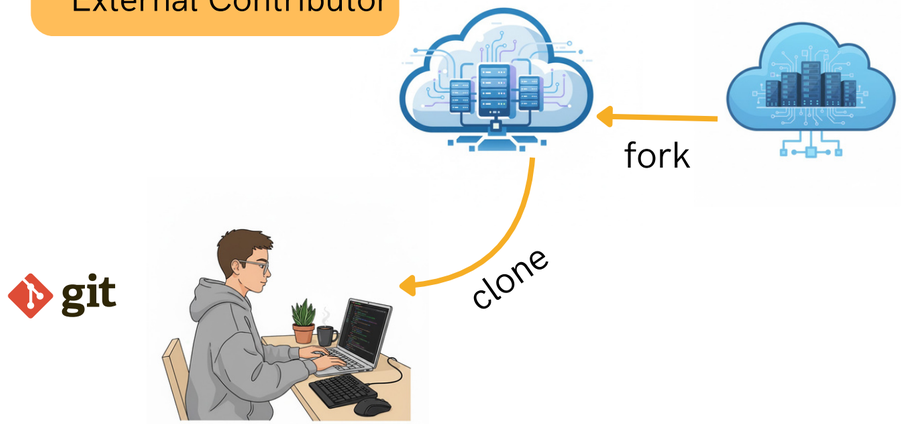
You start the project yourself. You initialize git locally, do work, then create a remote repository, add remote link, and push.

Collaborator



You're invited to contribute to an existing project.
You clone the repository, make changes locally, and push updates with the owner's permission.

External Contributor



You're not a direct member, but you want to contribute to a public repository.
You fork it (create your own copy), make changes, and propose them through a Pull Request/Merge Request.

How do you kickstart a git project?

Before Git can track your work, your project needs to become a repository.

- **git init** is the first step in turning any folder into a Git repository
- It gives Git the ability to manage your project history.

💡 Purpose of git init

- To start version control in a new or existing project folder.
- To let Git track file changes over time.
- To prepare the project for commits, branches, and collaboration.
- To create the hidden **.git** folder that stores all repository metadata.

🕓 Before **git init**

- Folder is a normal directory not tracked by Git.
- No **.git** folder exists.
- You can edit files, but there's no version control.
- Git commands (like git status) won't work.

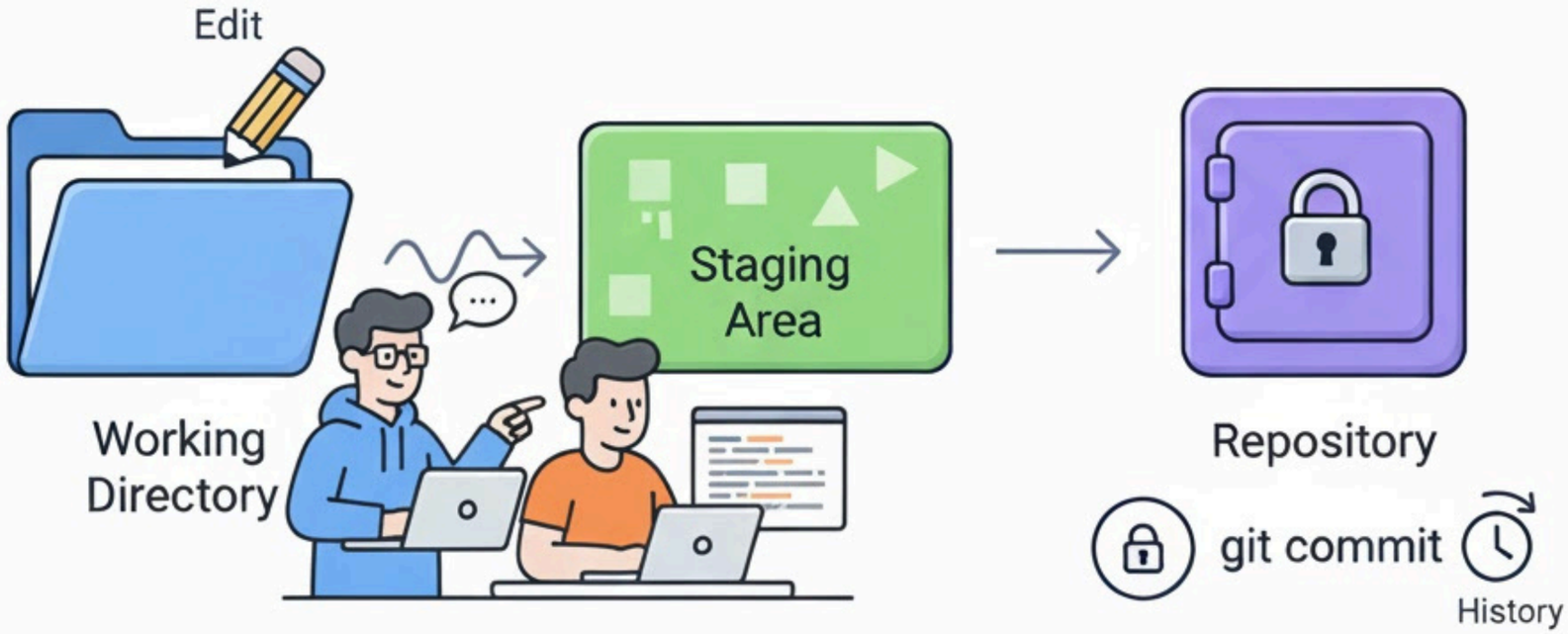
```
my_project/
├── index.html
└── style.css
```

⚡ After **git init**

- Git creates a hidden **.git** directory → initializes repository.
- The folder is now tracked by Git.
- You can add, commit, and track changes.
- The project is ready for collaboration and remote linking.

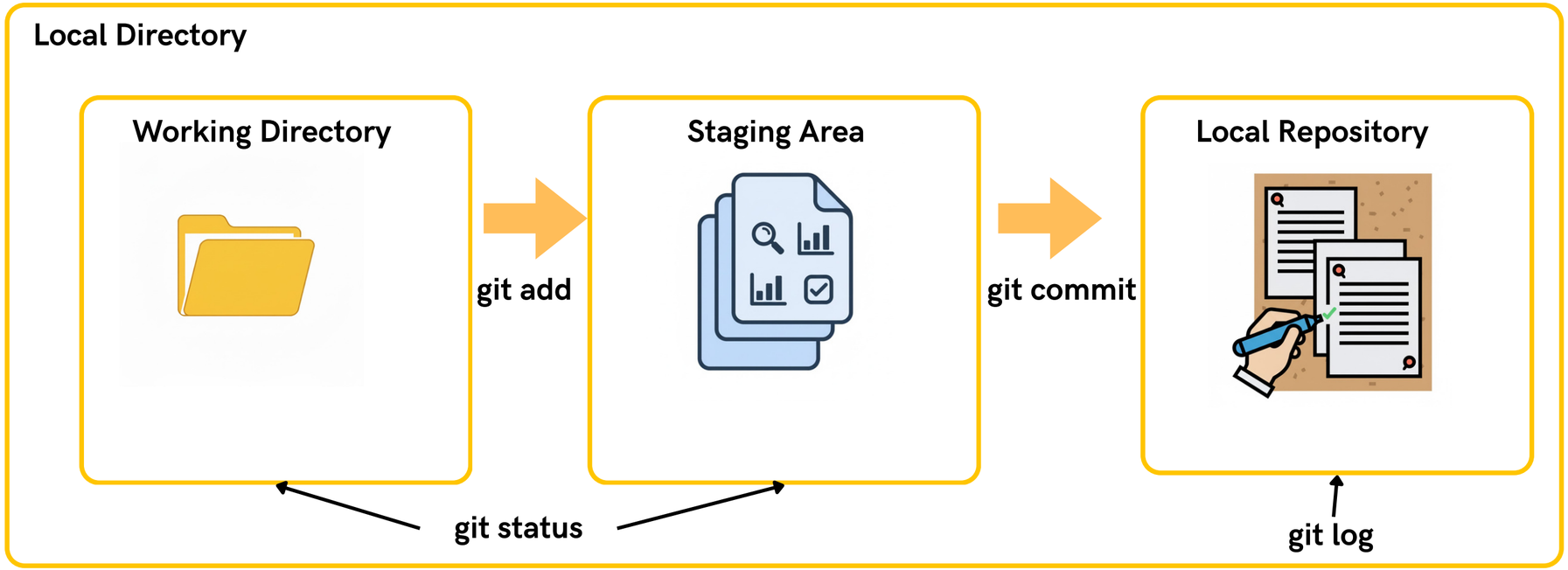
```
my_project/
├── .git/
├── index.html
└── style.css
```

Edit & Commit


Edit
Staging
Area
Working
Directory
Repository
git commit
History

How do you make the first commit in git (1/6)?

Before Git can record any history, your repository must have at least one commit.
This first commit becomes the foundation of the project's version history, everything after it builds upon that initial snapshot.



How do you make the first commit in git (2/6)?

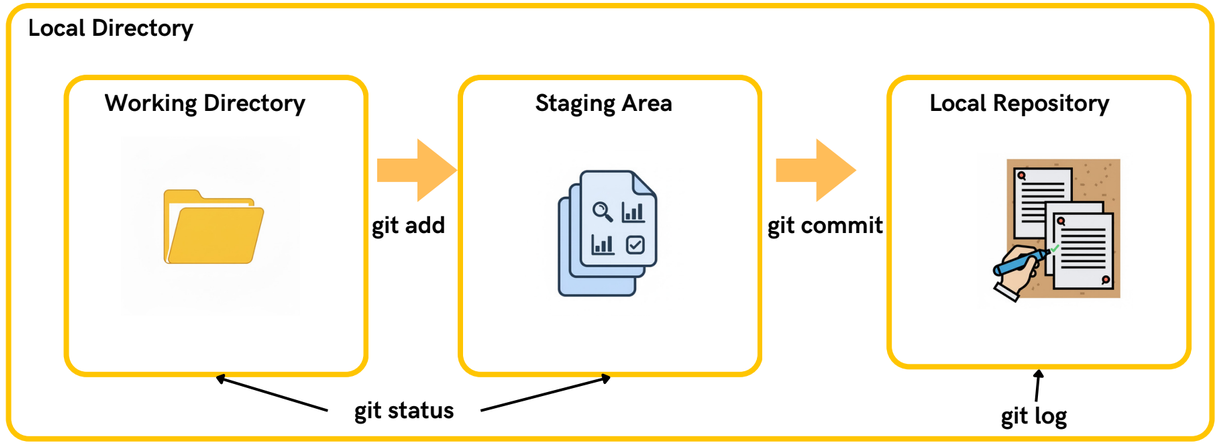
1

🔍 git status: Before Staging

💡 Purpose

- **git status** shows the current state of your working directory and staging area.
- Before staging, it helps you see which files are new, modified, or deleted.

```
$ git status
On branch main
No commits yet

Untracked files:
   (use "git add <file>..." to include in what will be committed)
        index.html
        style.css

nothing added to commit but untracked files present (use "git add" to track)
```



🟦 Explanation

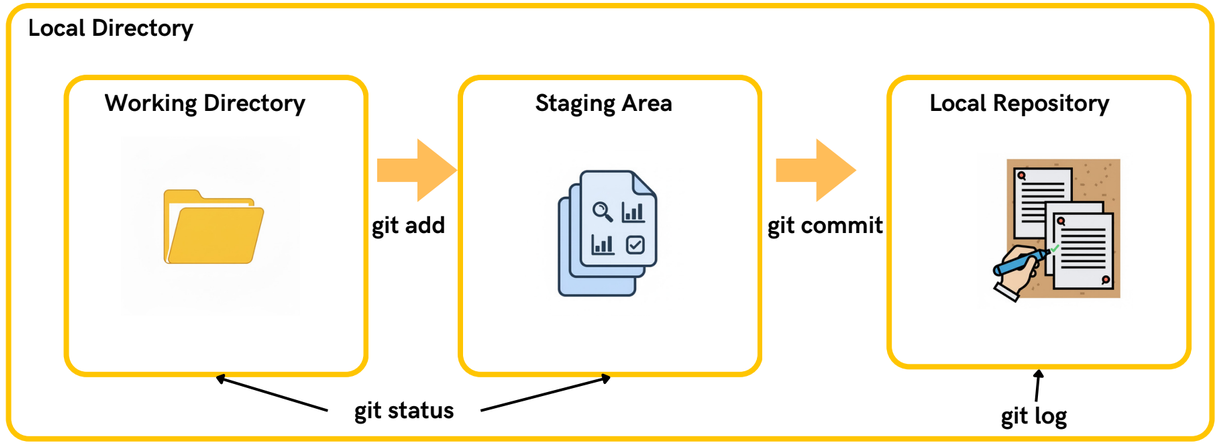
- **Untracked files:**
  → Files that exist in your directory but are not yet tracked by Git.
  Example: newly created files like index.html or style.css (in red).
- **Modified files:**
  → Files that were tracked before but have unsaved changes since the last commit.
- **Nothing added to commit:**
  → Means nothing is staged yet.
  → You must use git add to move changes to the staging area.

How do you make the first commit in git (3/6)?

2 Add Files to the Staging Area (git add)

💡 Purpose

- git add tells Git which files you want to include in your next commit.
- It moves files from the working directory → to the staging area (also called the index).



🧠 What git add does

- Prepares files for the next commit.
- Allows you to commit specific files instead of everything.
- You can use it repeatedly as you edit files each time you stage updated versions.

💻 Common Usage

```
# Add a specific file
git add index.html
# Add multiple files
git add index.html style.css script.js
# Add all changes (tracked + new files)
git add .
# Add all modified and deleted files, but not new ones
git add -u
```

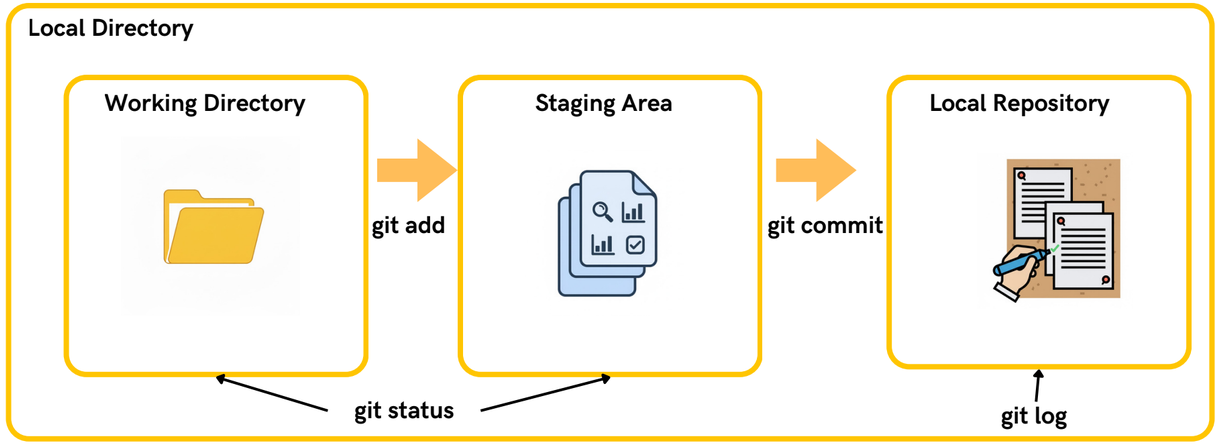
How do you make the first commit in git (4/6)?

3 git status: After Staging

Purpose

- After running git add, git status shows which files are ready to be committed.
- This lets you confirm that your changes are properly staged before you commit.

```
$ git add index.html style.css
$ git status
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   index.html
        new file:   style.css
```



Explanation

- **Changes to be committed:**
  → Files listed here are in the staging area, ready to be included in the next commit.
  → Git highlights them in green to show they're tracked.
- **Unstaged/Untracked changes:**
  → If there are other files not added yet, they'll still appear in red under Untracked files or Changes not staged for commit.

How do you make the first commit in git (5/6)?

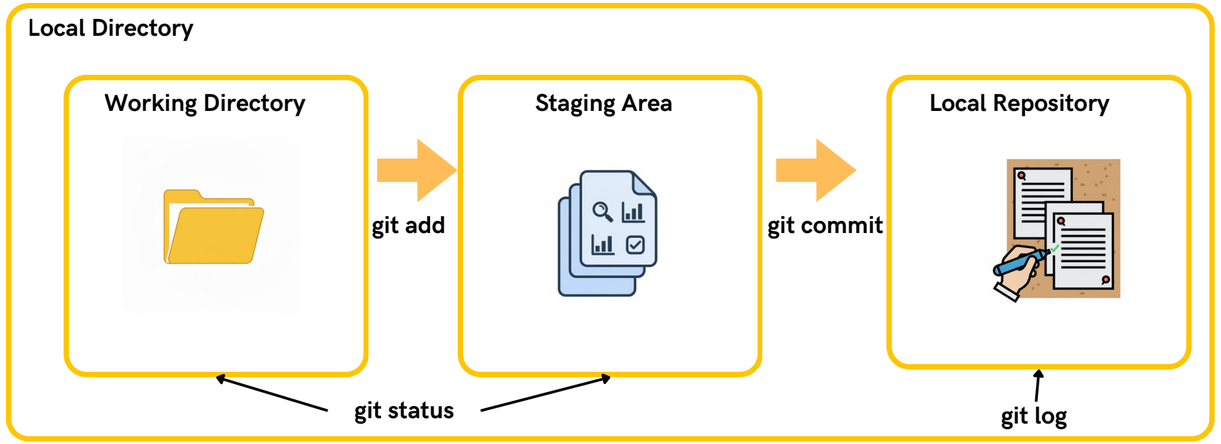
4 📝 git commit: Saving Your Changes

💡 Purpose

- git commit records the current snapshot of your project.
- It's the moment your changes become part of the repository history.

Each commit includes:

- The files you staged.
- The author information (from git config).
- A commit message describing the change.
- A unique commit ID (hash).




```
$ git commit -m "Add homepage and stylesheet"
[main (root-commit) a3b9d42] Add homepage and stylesheet
 2 files changed, 15 insertions(+)
 create mode 100644 index.html
 create mode 100644 style.css
```

🟦 Explanation

- a3b9d42 → Unique commit ID (SHA-1 hash)
- root-commit → It's the very first commit
- Shows number of files and changes made

How do you make the first commit in git (6/6)?

5 📜 git log: Viewing the Commit History

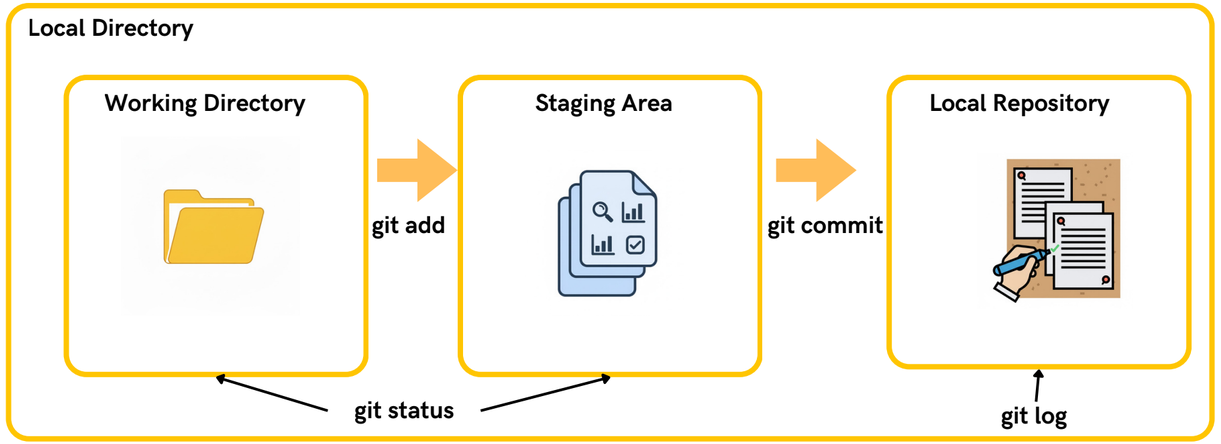
💡 Purpose

- **git log** displays the history of commits in your repository.
- It helps you see who made changes, when, and why => making it essential for tracking project evolution.

```
$ git log
commit a3b9d42c3f13e7c0a58e7cc8d8e7f9d6b73ad5c4 (HEAD -> master)
Author: Walid Mallat <walid.mallat@example.com>
Date: Sat Oct 18 10:32:41 2025 +0100
Add homepage and stylesheet
```

🟦 Explanation

- commit a3b9d42… → Unique commit ID (hash) identifying the snapshot
- (HEAD → main) → Shows that your current branch (master) points to this commit
- Author → Comes from your git config user.name and user.email
- Date → Timestamp of the commit
- Message → Describes what the commit did



💻 Common Usage

```
$ git log # Show full commit history
$ git log --oneline # Shows each commit in one compact line
$ git log --name-only # Show only the names of changed files per commit
$ git log --graph # Show commits as a tree/graph
$ git log --oneline --graph --all --decorate # Full visual summary
with branches and tags
$ git log branch_name # Show commits of a specific branch
```

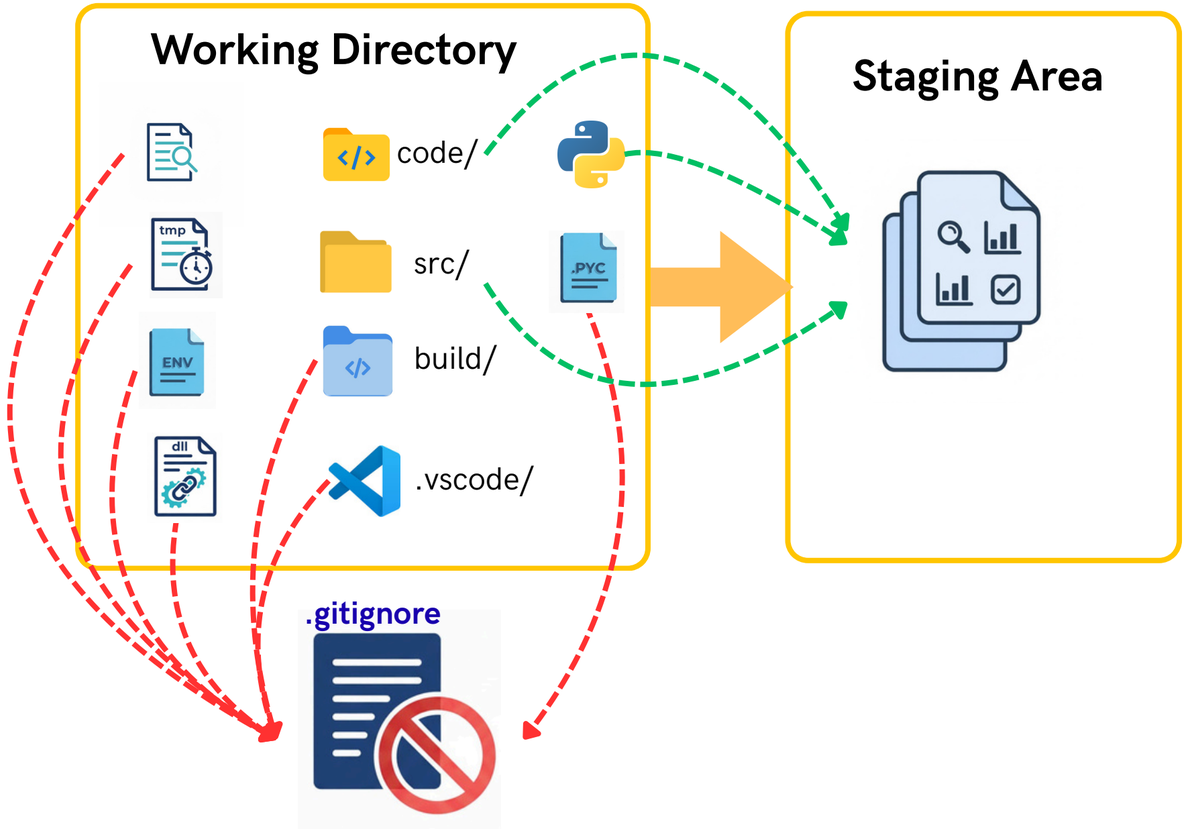
How do you exclude files from git tracking?

.gitignore

- Git tracks all files in your project by default.
- Some files are unnecessary, temporary, or sensitive (e.g., .env, logs, system files).
- **.gitignore** tells Git which files to skip, keeping the repository clean and secure.

🎯 Purpose of .gitignore

- Prevents Git from tracking unnecessary or sensitive files (e.g., .env, log files, system files like .DS_Store).
- Keeps your repository clean and secure



How to use .gitignore in git?

1 **Create the .gitignore file:**

Create it in the root of your project:

```
$ touch .gitignore
```

2 📝 **Add patterns of files to ignore:**

Edit .gitignore and add filenames, extensions, or folders you don’t want Git to track.

```
# Ignore environment files
.env

# Ignore system and IDE files
.DS_Store
.vscode/

# Ignore logs
logs/
*.log
```

3 🔍 **Check with git status:**

Before adding .gitignore, git status might show:

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .env
        .DS_Store
        logs/error.log
```

After creating and saving .gitignore, run git status::

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
```

Notice that .env, .DS_Store, and logs/ are now excluded => Git no longer shows them.

4 🧱 **Stage and commit .gitignore**

```
$ git add .gitignore
$ git commit -m "Add .gitignore to exclude unnecessary files"
```

How do you create a remote repository?

1 🧭 **Choose a platform**

Pick your hosting service:

GitHub GitLab Bitbucket

2 ⚙️ **Create the repository**

- Click “New Repository”.
- Enter a repository name.
- Add a description (optional but helpful).
- Choose Public 🔓 or Private 🔒.
- ⚠️ Do not initialize with a README if you already have one locally.

3 📋 **Copy the repository URL**

After creation, Git gives you two options:

https://github.com/username/myproject.git

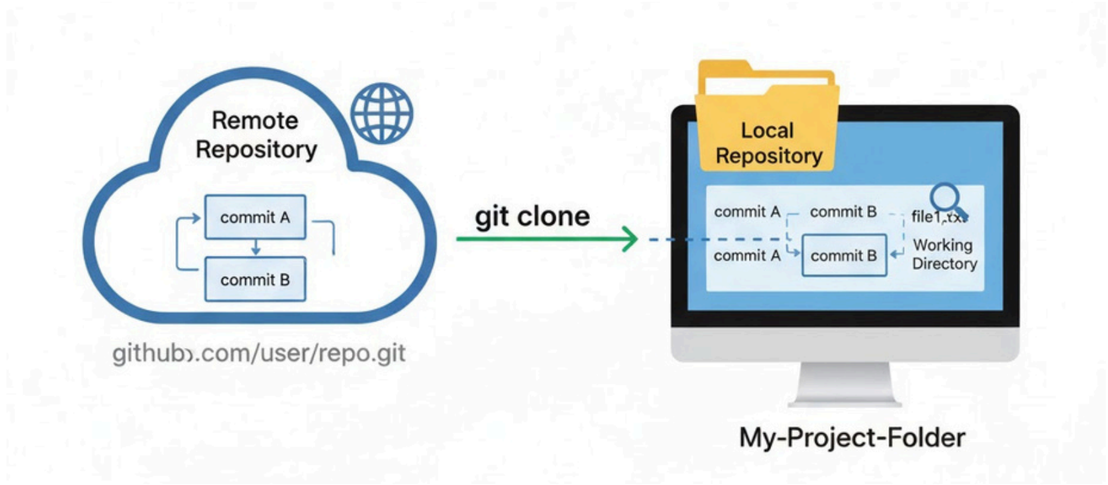
git@github.com:username/myproject.git

Use SSH for secure access (no need to type password each time).

How to download a project locally?

💡 Why Clone?

- Before contributing to a project, you first need your own local copy of the remote repository.
- That's where **git clone** comes in. It copies the entire project and history so you can work freely on your computer. 💻



1 **Find the repository URL:**

- Go to your Git hosting platform (GitHub, GitLab, etc.)
- Copy the HTTPS or SSH URL

(SSH is more secure and avoids password prompts)

2 **Run the clone command:**

git clone <repository-URL>

3 **Move into the project folder:**

cd <repository-name>

4 **Check your remote connection:**

```
$ git remote -v
```

Confirms that your local repo is linked to the remote (origin).

How to contribute using a fork?

Sometimes you want to improve or experiment with a project that isn't yours, but you can't directly push changes to it.

👉 That's where forking comes in!

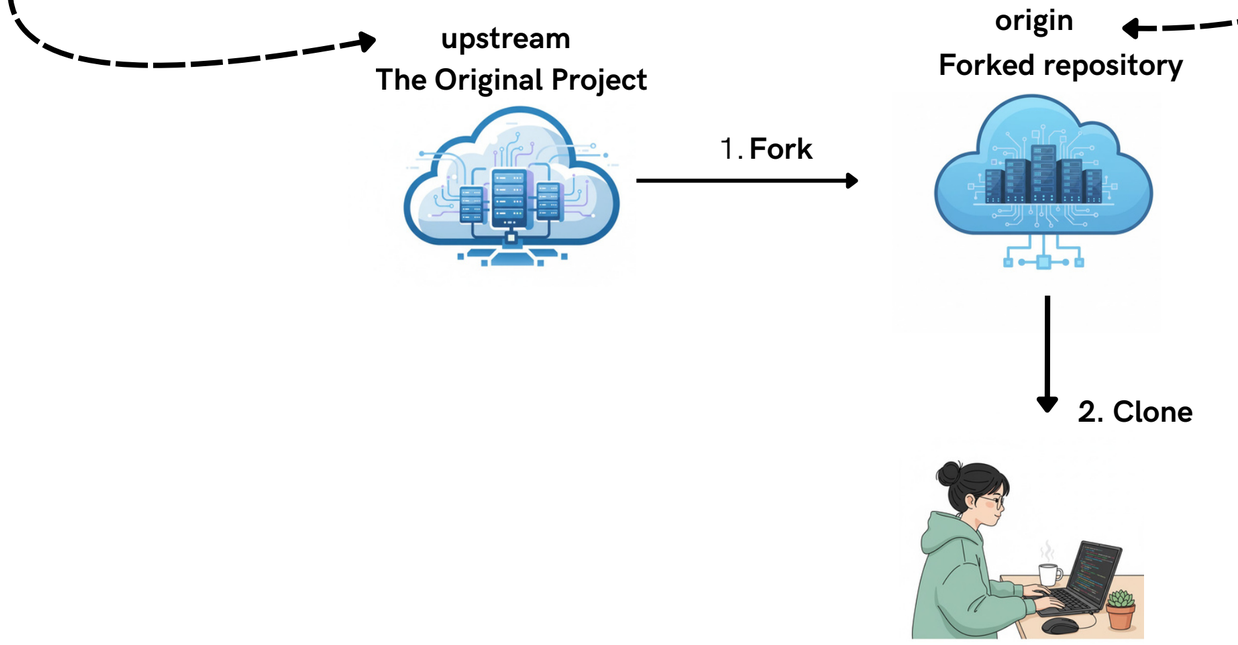
- Forking creates your own personal copy of someone else's repository.
- You can freely edit, test, and even propose changes without touching the original project.

🔗 **What is "Upstream"?**

- Upstream refers to the original repository you forked from.
- It's useful for keeping your fork updated with the latest changes.

🌐 **What is "Origin"?**

- Origin is the default name for your remote copy (your fork).
- It points to your own repository.



How to contribute to another project?

⚙️ Steps to Fork and Work on It

1. **Fork the repository:**
   - Go to the project page (e.g., on GitHub or GitLab).
   - Click "Fork" 🔁 → this creates a copy in your account.

2. **Clone your fork locally:**

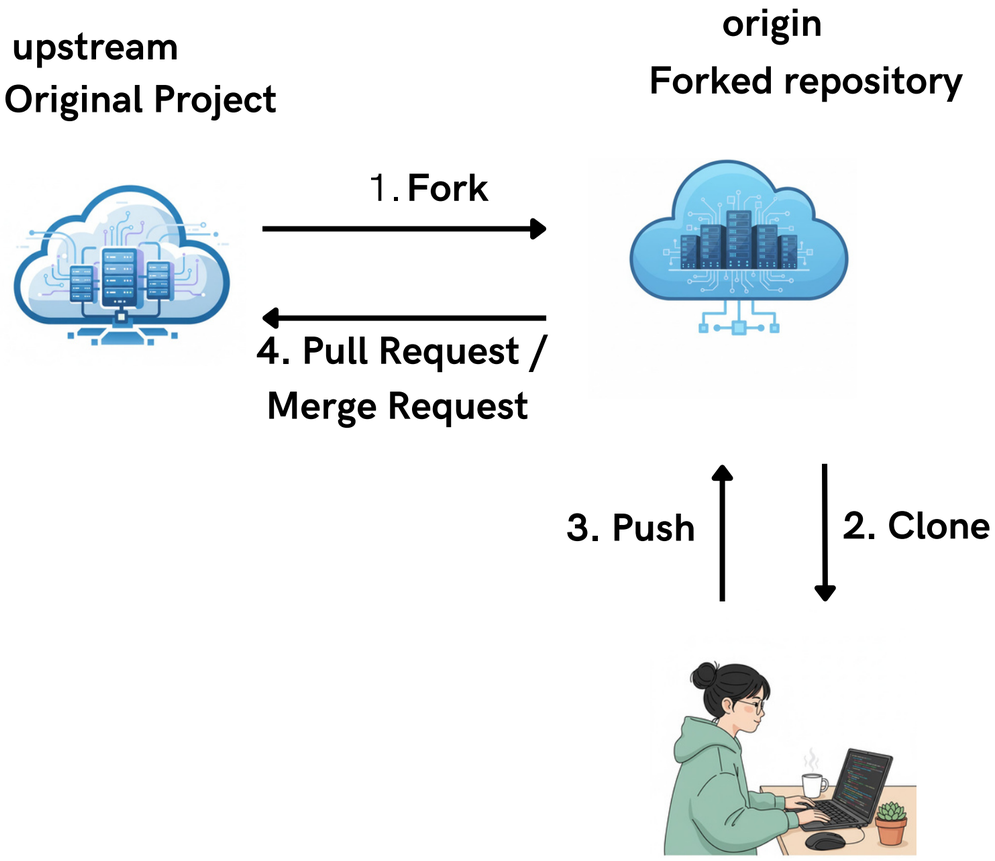
```
$ git clone https://github.com/yourname/project.git
```

3. **Make your changes:**

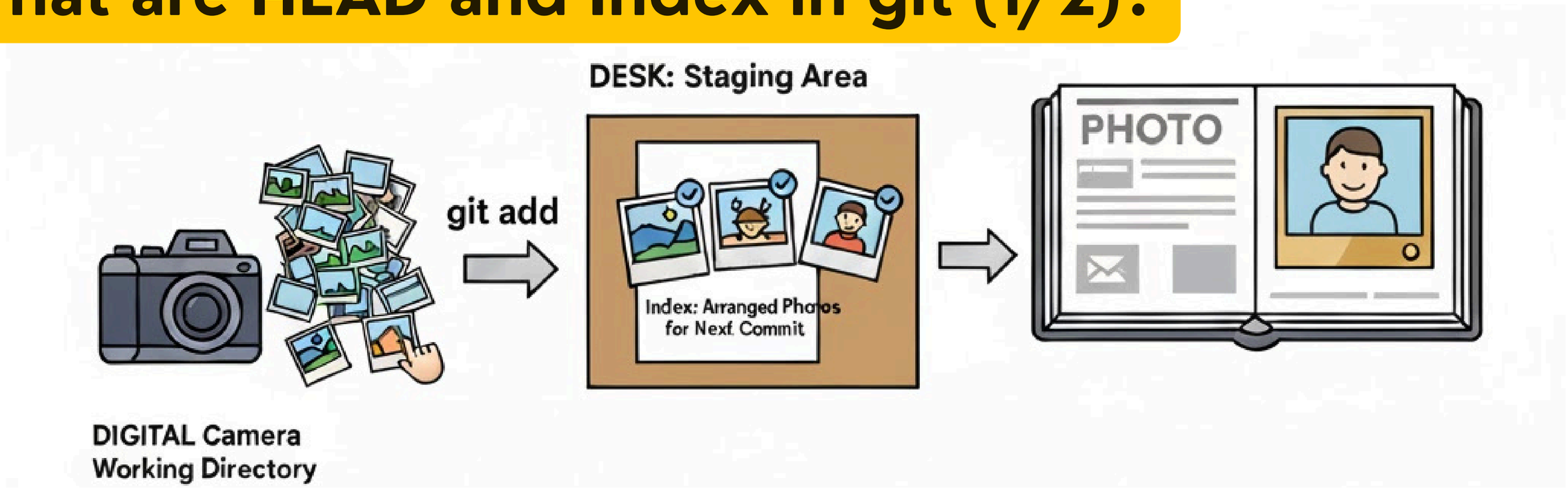
```
$ git push origin main
```

4. **Propose your improvements:**

   Open a Pull Request (PR)/ Merge Request (MR) to suggest merging your changes into the original repo.



What are HEAD and Index in git (1/2)?

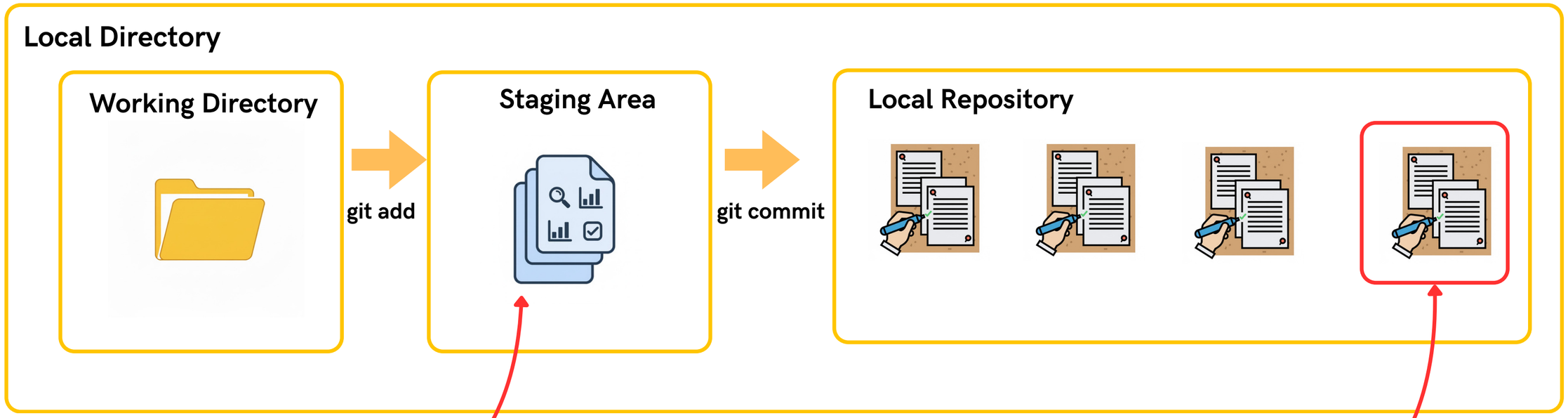


Working Directory:
This is like your camera 📷, where you're constantly taking new photos (making changes to your files). These photos are still "loose" and not organized.

Index (Staging Area):
You look through your camera, pick the best photos you want to keep, and lay them out on your desk 📝. These are your staged changes (git add).

HEAD:
When you're happy with the arrangement on your desk, you take a "snapshot" of those chosen photos and put it at the very end of your album. That new snapshot becomes your HEAD (git commit).

What are HEAD and Index in git (2/2)?



Index

A temporary area where you put changes before committing.

HEAD (latest commit)

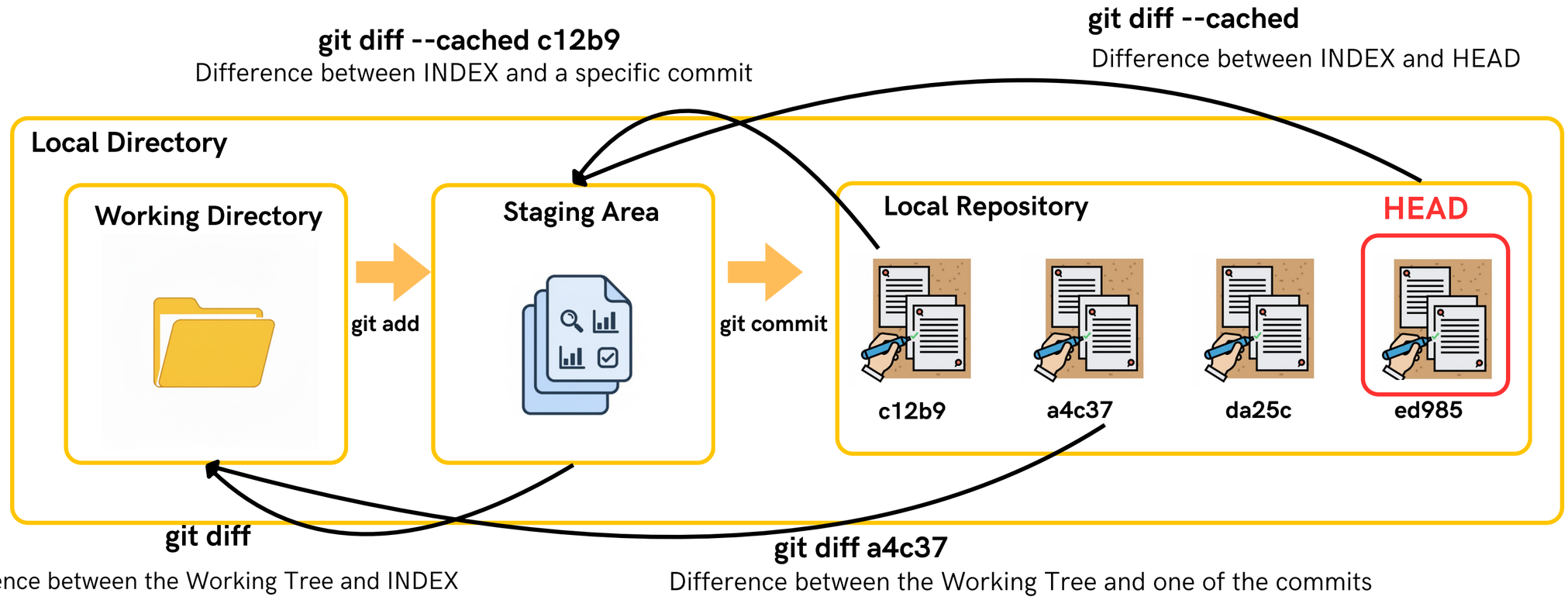
Points to your latest commit (the current snapshot of your project).
Moves forward when you make a new commit.

How can you see what's changed in git?

Before committing your changes, you might wonder:

"What exactly did I modify in my code?"

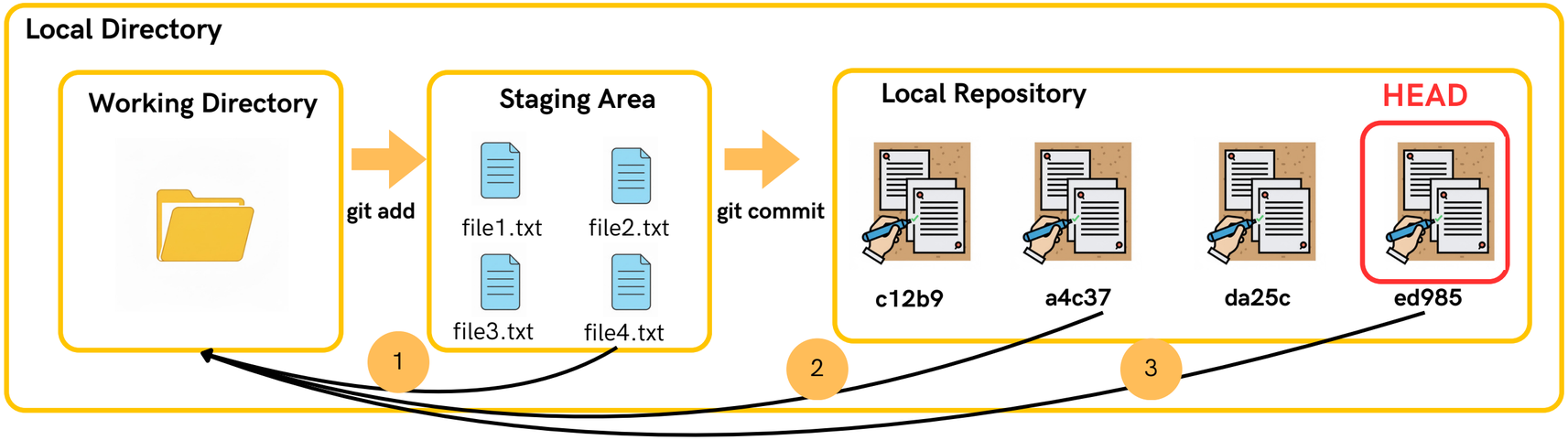
That's where **git diff** helps.it shows the exact differences between your files and Git's history.



How can you undo changes safely (1/6)?

Sometimes you modify files but later realize
"Oops! I want to go back to the previous version."

That's when **git restore** comes to the rescue. It helps you restore files in your working directory or staging area without affecting your commit history.

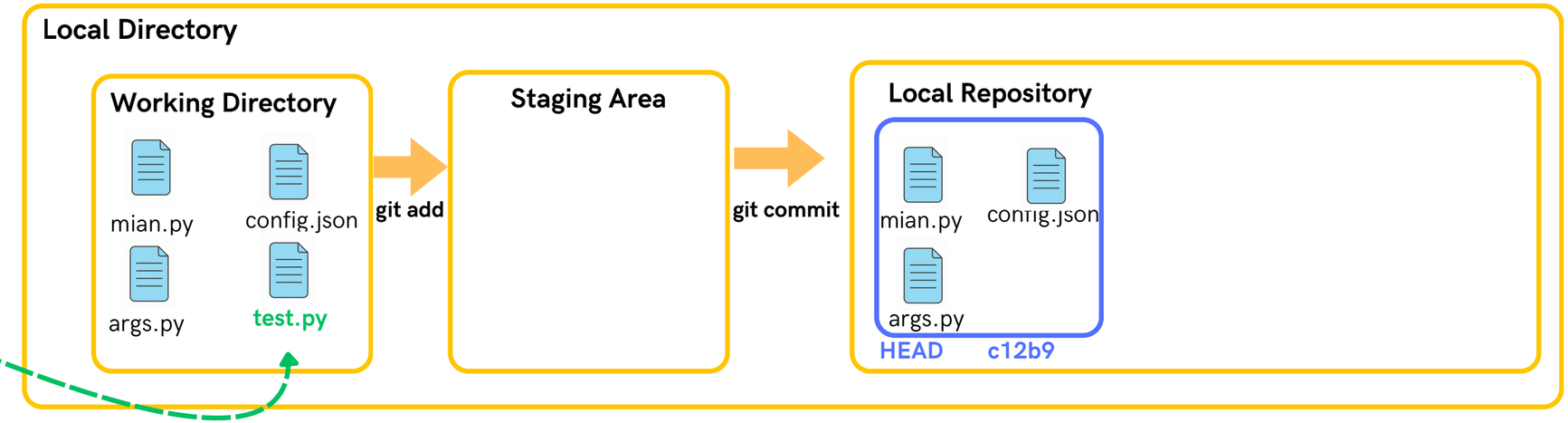


How can you undo changes safely (2/6)?

Untracked file (new, never added)

🧩 **Context**:

You created **test.py** and never did **git add** or **git commit** it.




```
$ git status
Untracked files:
  test.py
```

You run

```
$ git restore test.py
error: path 'test.py' is untracked
```

What Git does:

- Git cannot restore test.py because there is no entry for it in HEAD or Index (no saved version).

How to remove or clean untracked files (safe steps):

Preview what would be removed:

```
$ git clean -n
```

Remove untracked files:

```
git clean -f
```

How can you undo changes safely (3/6)?

Tracked file (was committed before)

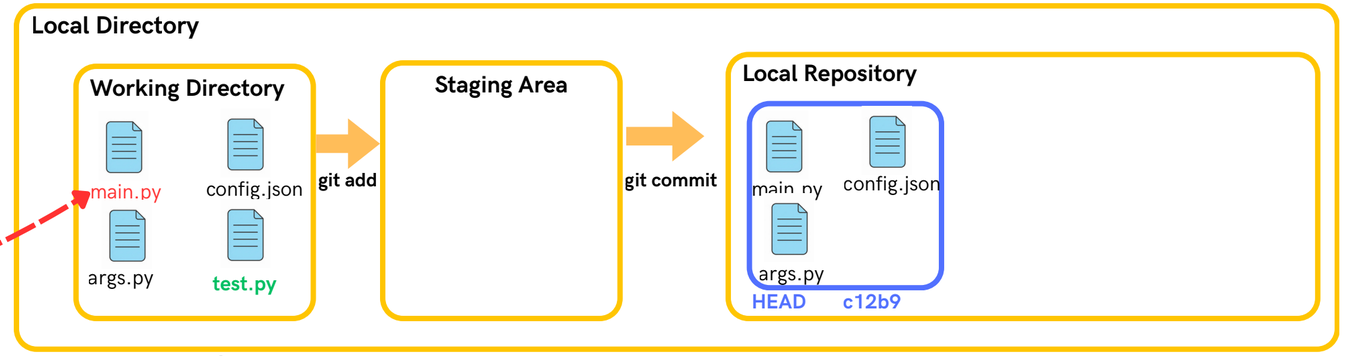
Context:

main.py → ✅ tracked file, previously committed
You open **main.py** and modify some code, but you haven't staged or committed it yet.

main.py → ✅ modified
config.json → ✅ clean (unchanged)
args.py → ✅ clean (unchanged)
test.py → 🆕 untracked

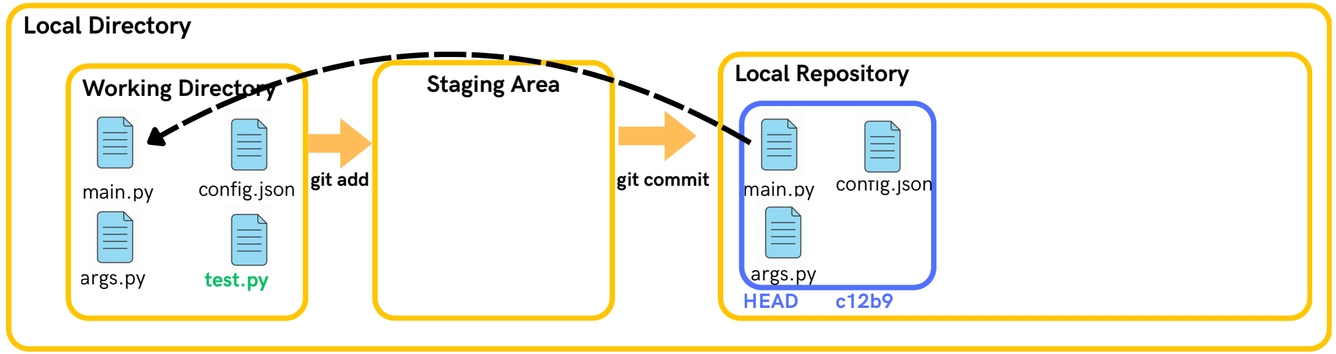
If I want to discard my local modifications in **main.py** and restore it to the last committed version from HEAD, I use:

```
$ git restore main.py
```



What happens:

- Git replaces the working directory version of main.py
- It takes the version from the staging area (index) or HEAD if index is clean
- Your modifications are lost (unless saved elsewhere)



main.py is reset to last commit (HEAD: c12b9), looks like you never edited it.

How can you undo changes safely (4/6)?

Tracked file (was committed before)

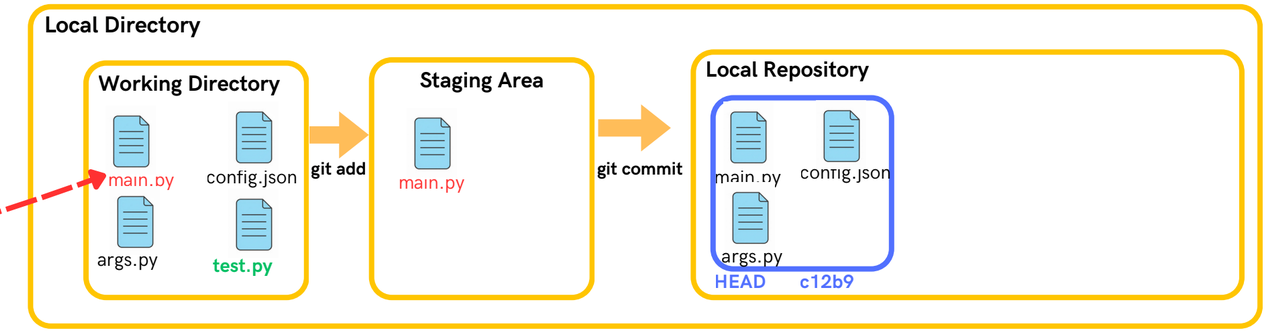
🧩 **Context**:

main.py → ✅ tracked file, previously committed
You open **main.py** and modify some code, but you stage it.

main.py → ✅ modified
config.json → ✅ clean (unchanged)
args.py → ✅ clean (unchanged)
test.py → 🆕 untracked

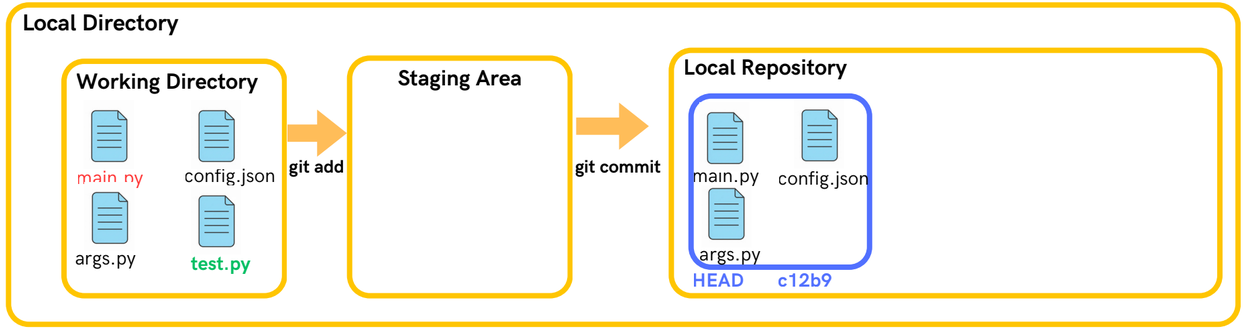
If I want to unstage a file that I previously added to the staging area (without losing my changes in the working directory), I use:

```
$ git restore --staged main.py
```



⚙️ **What happens:**

- Used after you've run **git add main.py**
- Git removes the file from the index (staging)
- Keeps your working directory changes intact



➡️ The file is unstaged, but your changes are still visible in the editor.

How can you undo changes safely (5/6)?

Tracked file (was committed before)

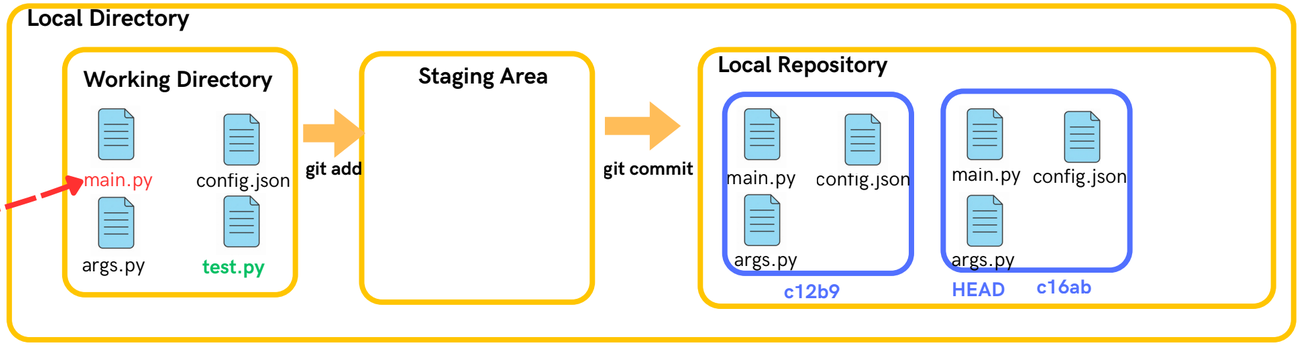
🧩 **Context**:

main.py → ✅ tracked file, previously committed
You open **main.py** and modify some code.

main.py → ✅ modified
config.json → ✅ clean (unchanged)
args.py → ✅ clean (unchanged)
test.py → 🆕 untracked

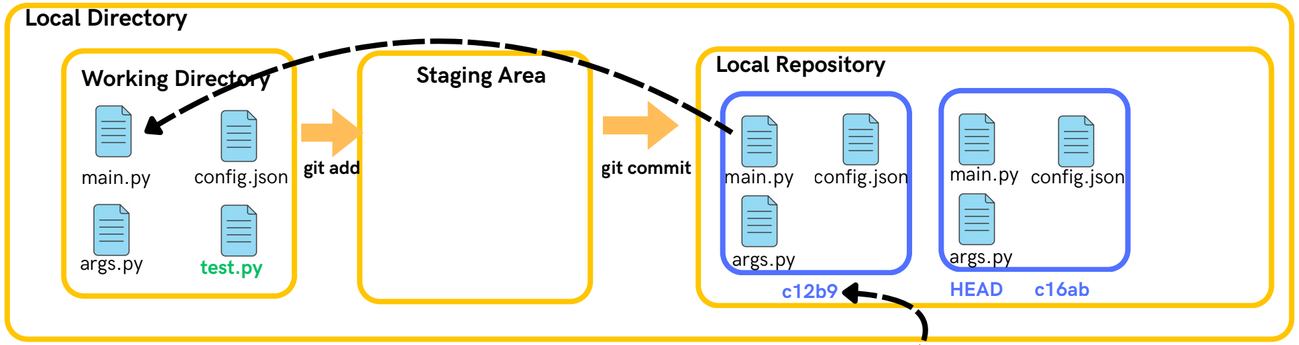
If I want to restore main.py from a specific commit instead of the last one (HEAD), I use:

```
$ git restore --source=c12b9 main.py
```



⚙️ **What happens:**

- Git looks at the commit hash you specify.
- It pulls the file version from that commit into your working directory (by default).



➡️ Restores main.py as it was in commit **c12b9**.

How can you undo changes safely (6/6)?

Tracked file (was committed before)

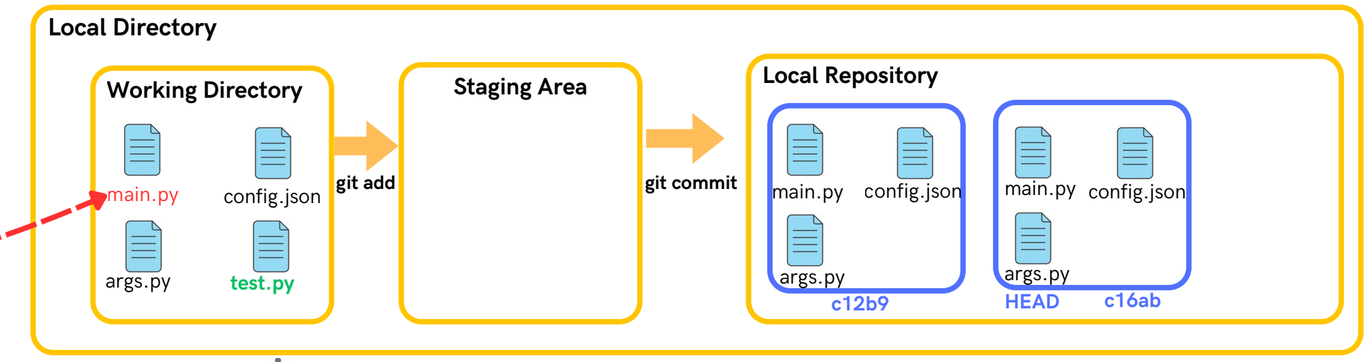
🧩 **Context**:

main.py → ✅ tracked file, previously committed
You open **main.py** and modify some code, but you haven't staged or committed it yet.

main.py → ✅ modified
config.json → ✅ clean (unchanged)
args.py → ✅ clean (unchanged)
test.py → 🆕 untracked

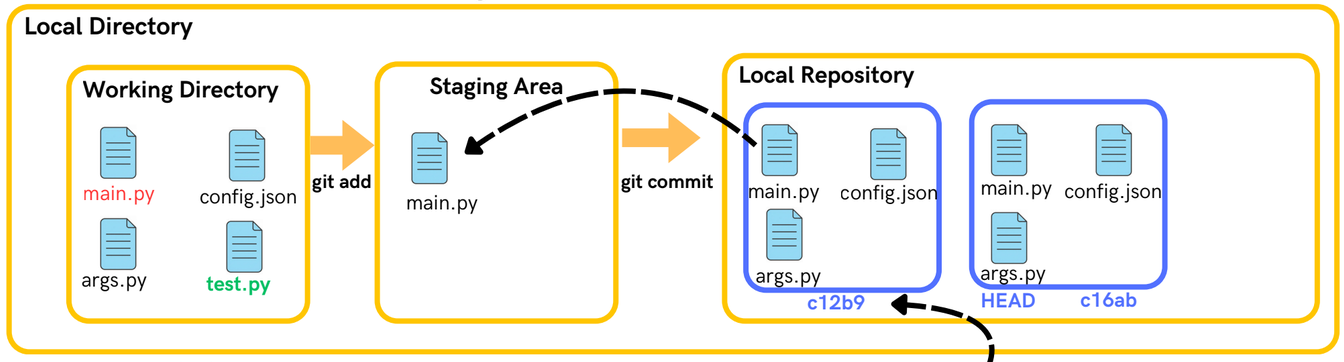
If I want to replace the staged version of main.py with the version from a specific commit (without changing the working directory), I use:

```
$ git restore --source=c12b9 --staged main.py
```



⚙️ **What happens:**

- Git looks up the version of main.py in the specified commit.
- Git replaces the staged version (index) with that commit's version.
- Your working directory file remains untouched => your edits are still there.



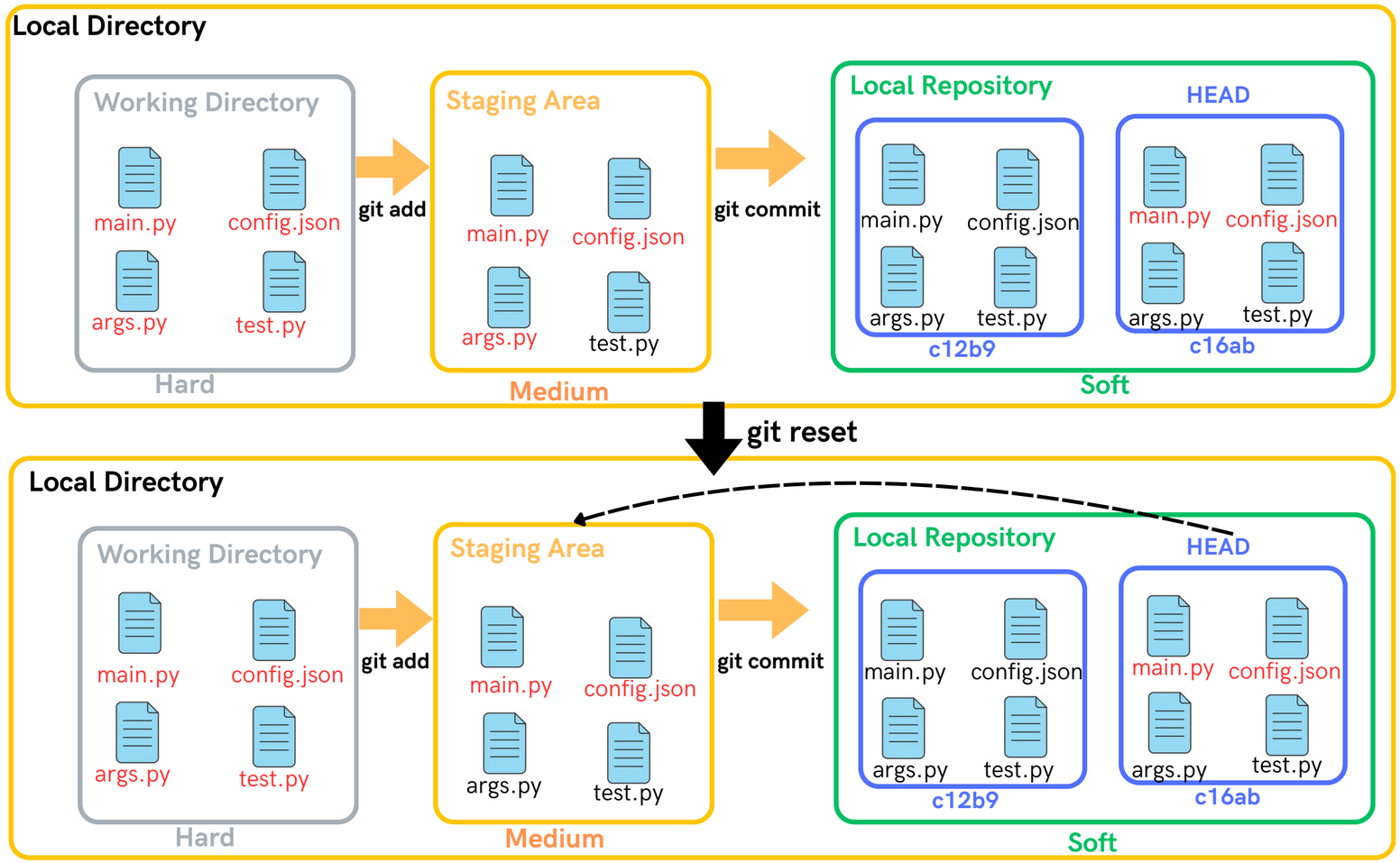
➡️ The content staged is now the version from **c12b9**

How can you undo changes with git reset (1/4)?

Clear the Staging Area

Remove files from staging without touching working directory:

```
$ git reset
```



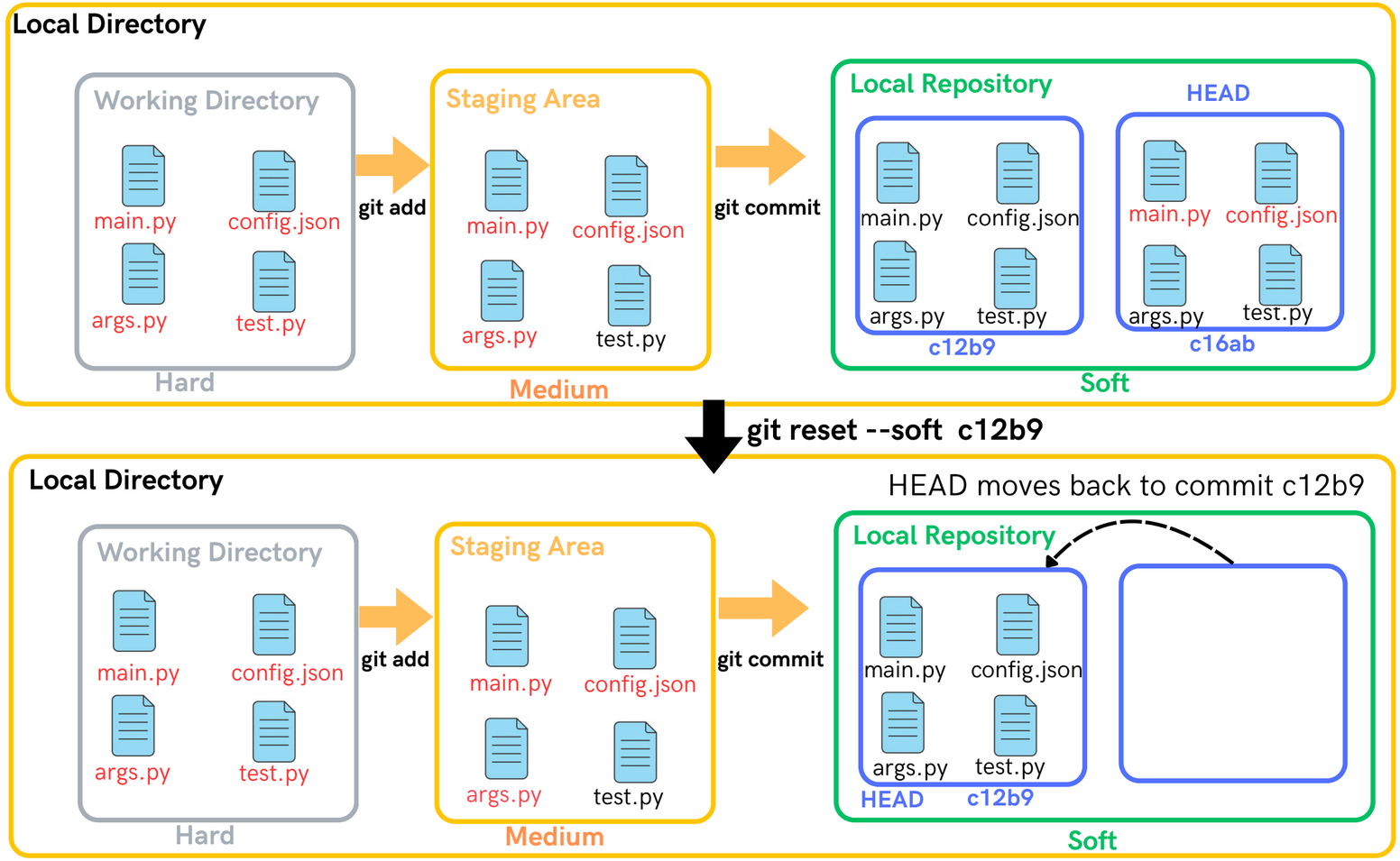
Modified code Original code

How can you undo changes with git reset (2/4)?

```
$ git reset --soft  <commit hash>
```

--soft

- Moves HEAD to the specified commit.
- Leaves both the staging area and working directory unchanged.



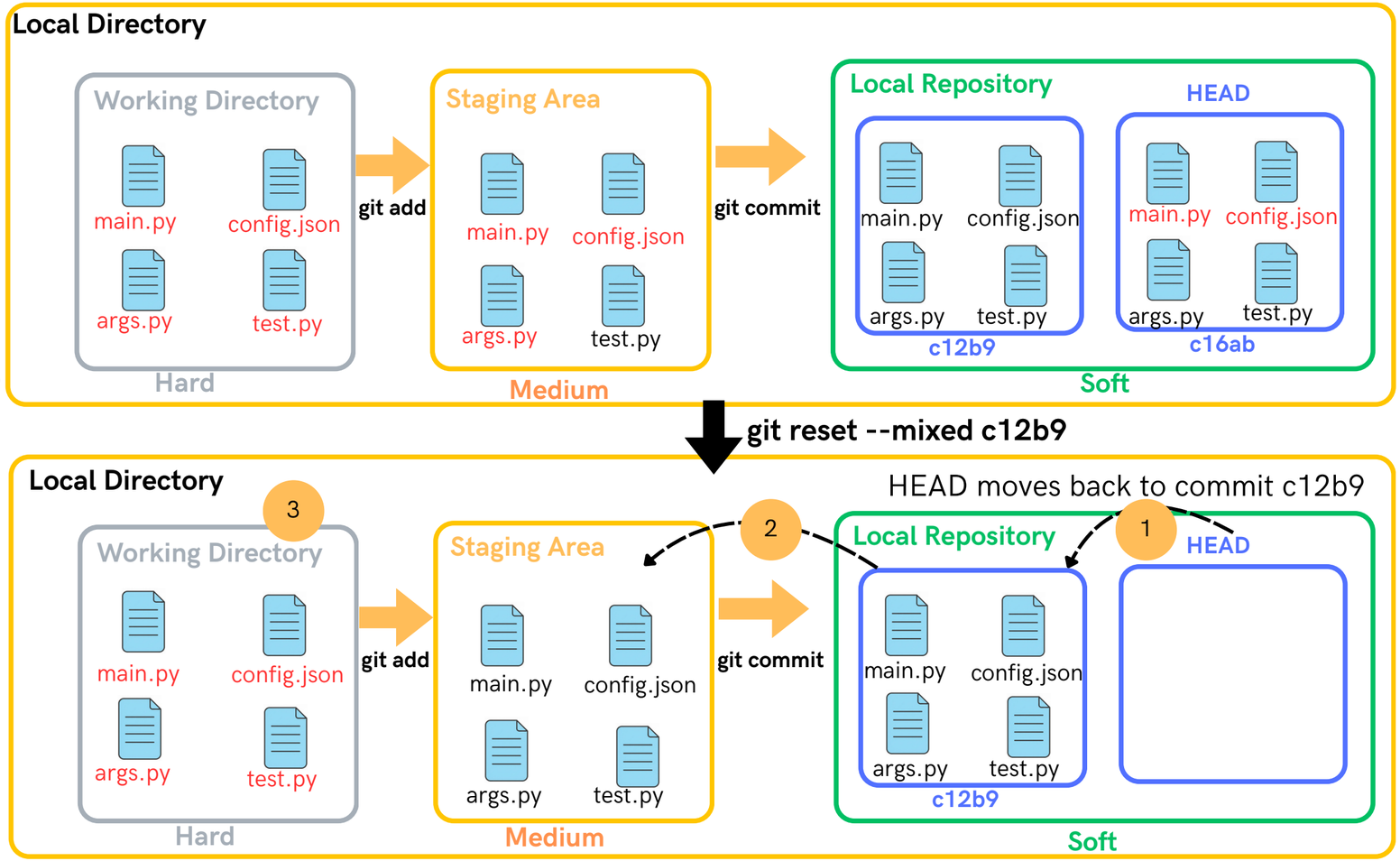
Modified code Original code

How can you undo changes with git reset (3/4)?

```
$ git reset --mixed <commit hash>
```

--mixed (default).

1. Moves HEAD to the specified commit.
2. Resets the staging area to match the specified commit.
3. Leaves the working directory unchanged.



Modified code Original code

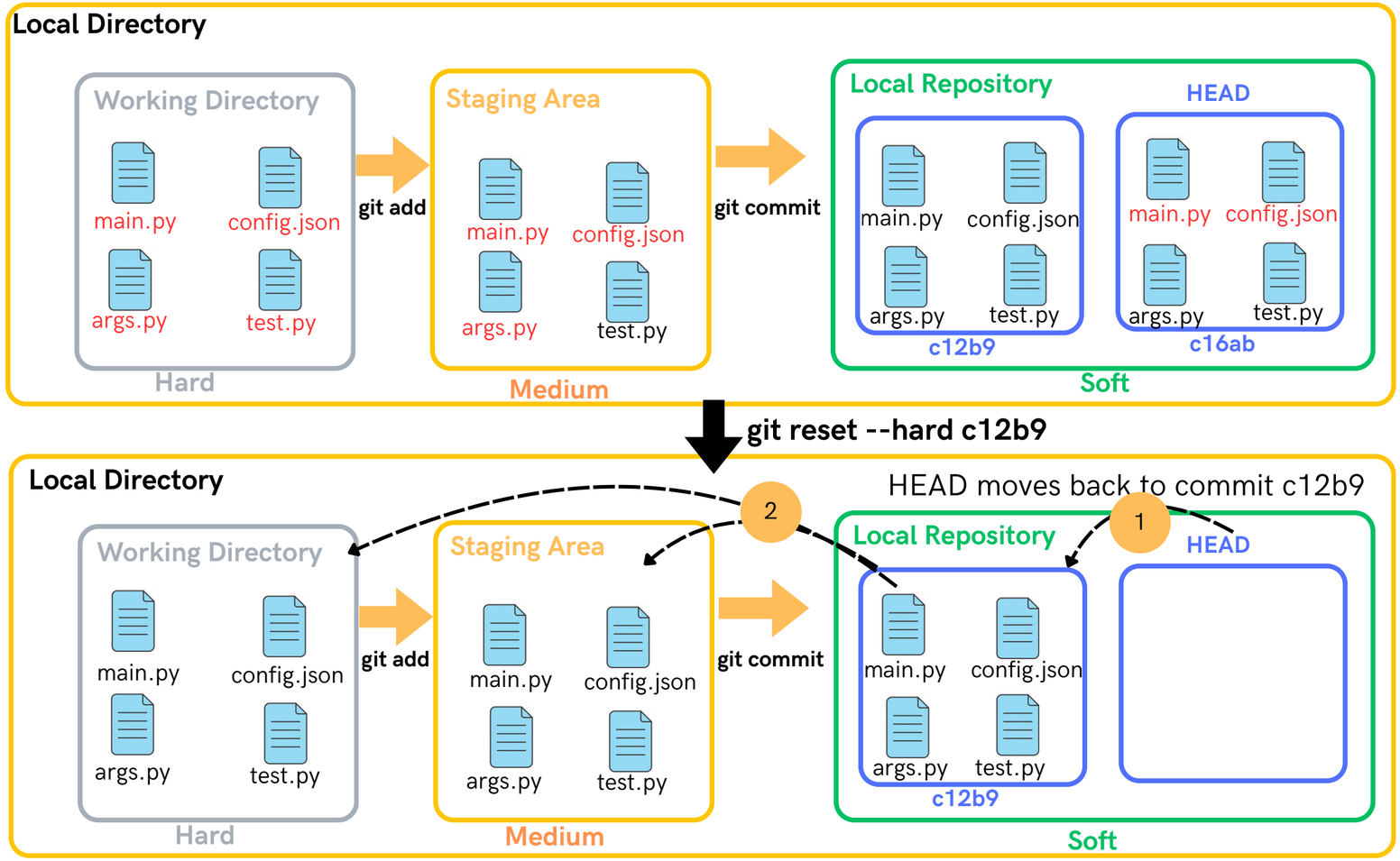
How can you undo changes with git reset (4/4)?

```
$ git reset --hard <commit hash>
```

--hard

1. Moves HEAD to the specified commit.
2. Resets both the staging area and working directory to match the specified commit.

⚠ This option discards all changes in the staging area and working directory.



Modified code Original code

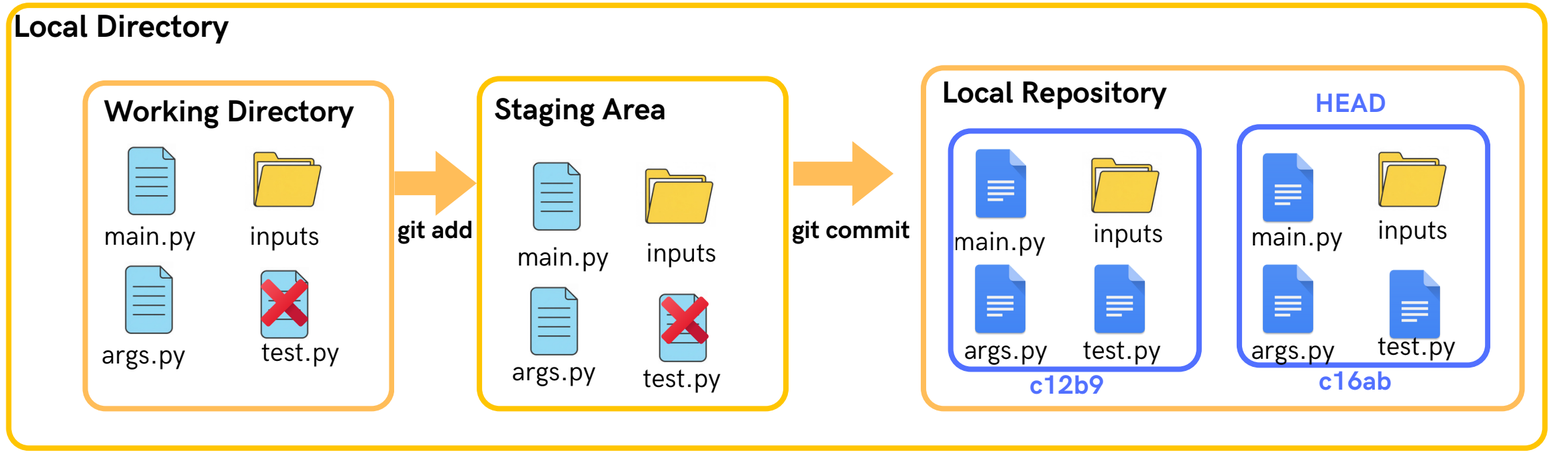
How to delete files in git (1/2)?

```
$ git rm test.py
```

🟢 The file is removed from:

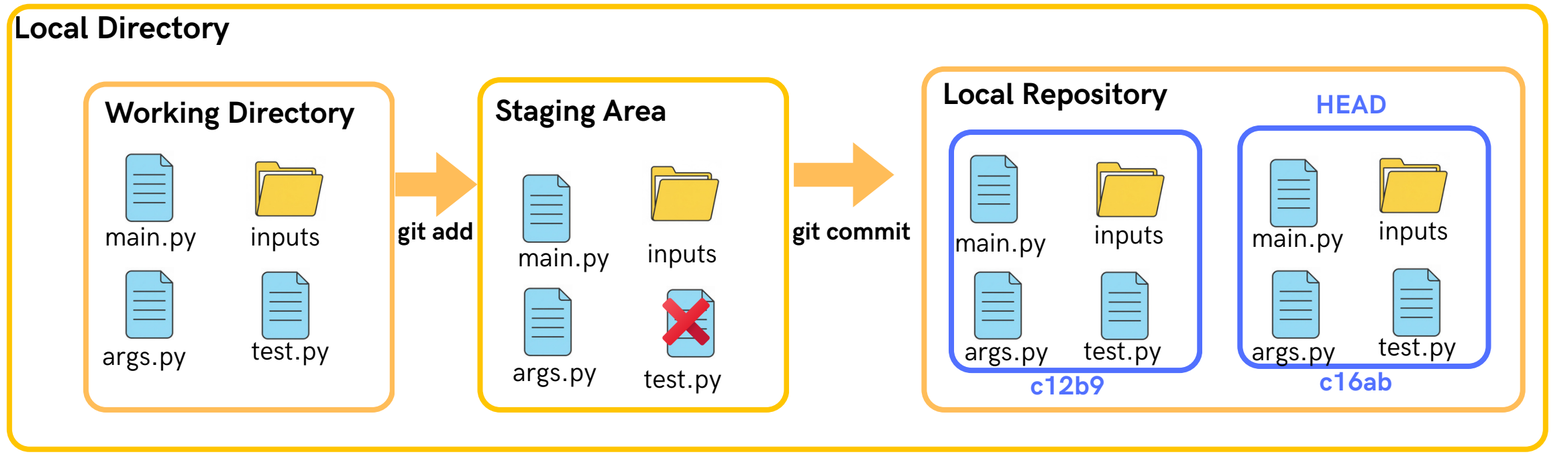
- 🗂️ Working directory → test.py is deleted from your local folder.
- 🧾 Staging area → File is marked as "deleted" and ready to be committed.

➡️ Next step: **git commit -m "Remove test.py"**




```
$ git rm --cached test.py
```

1. Git removes the file from the staging area → it's no longer tracked.
2. Git keeps the file in the working directory → it's still on your disk.
3. The file still exists in the last commit (HEAD) until you commit the removal.

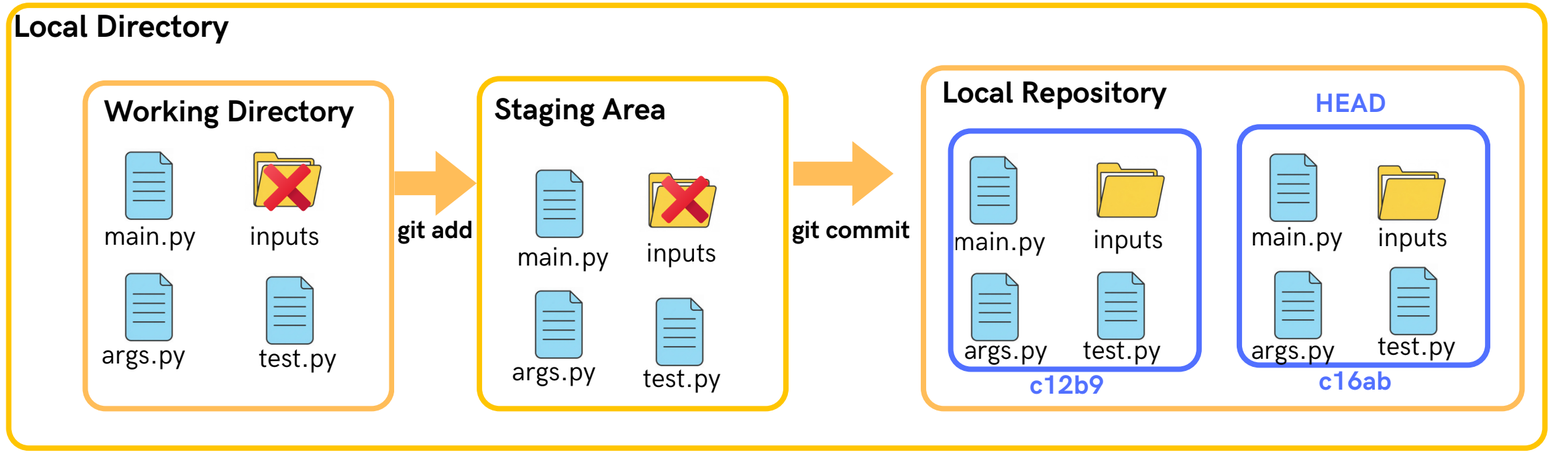


How to delete files in git (2/2)?

```
$ git rm -r inputs
```

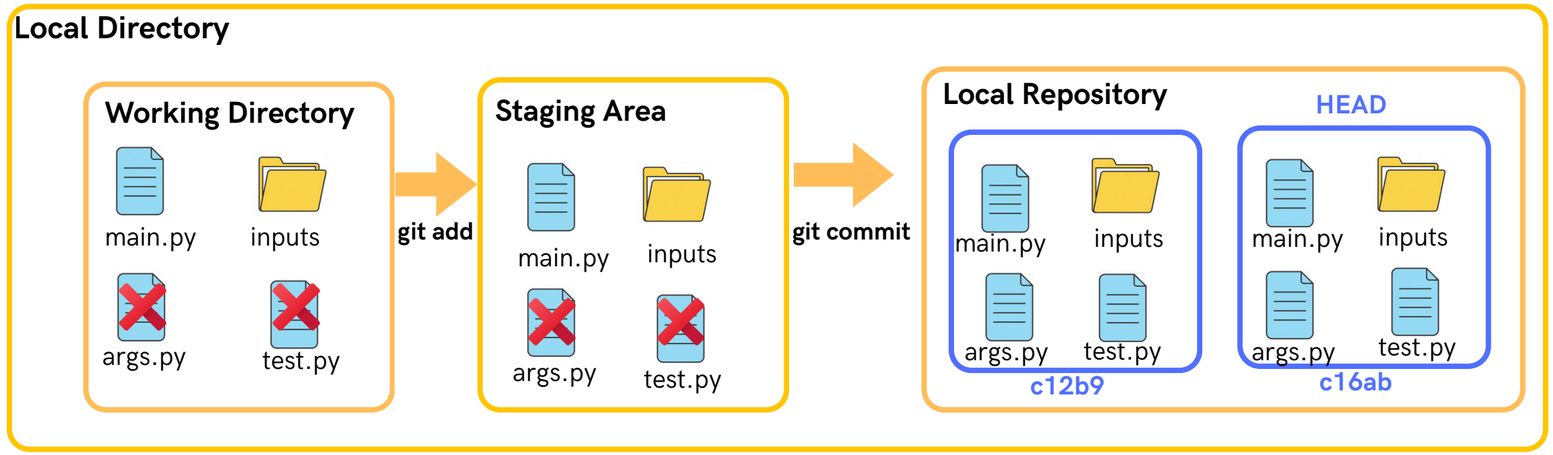
The command removes a directory and all its tracked files from both:

- your working directory (files are deleted from disk),
- and the staging area (they'll be removed from the next commit).

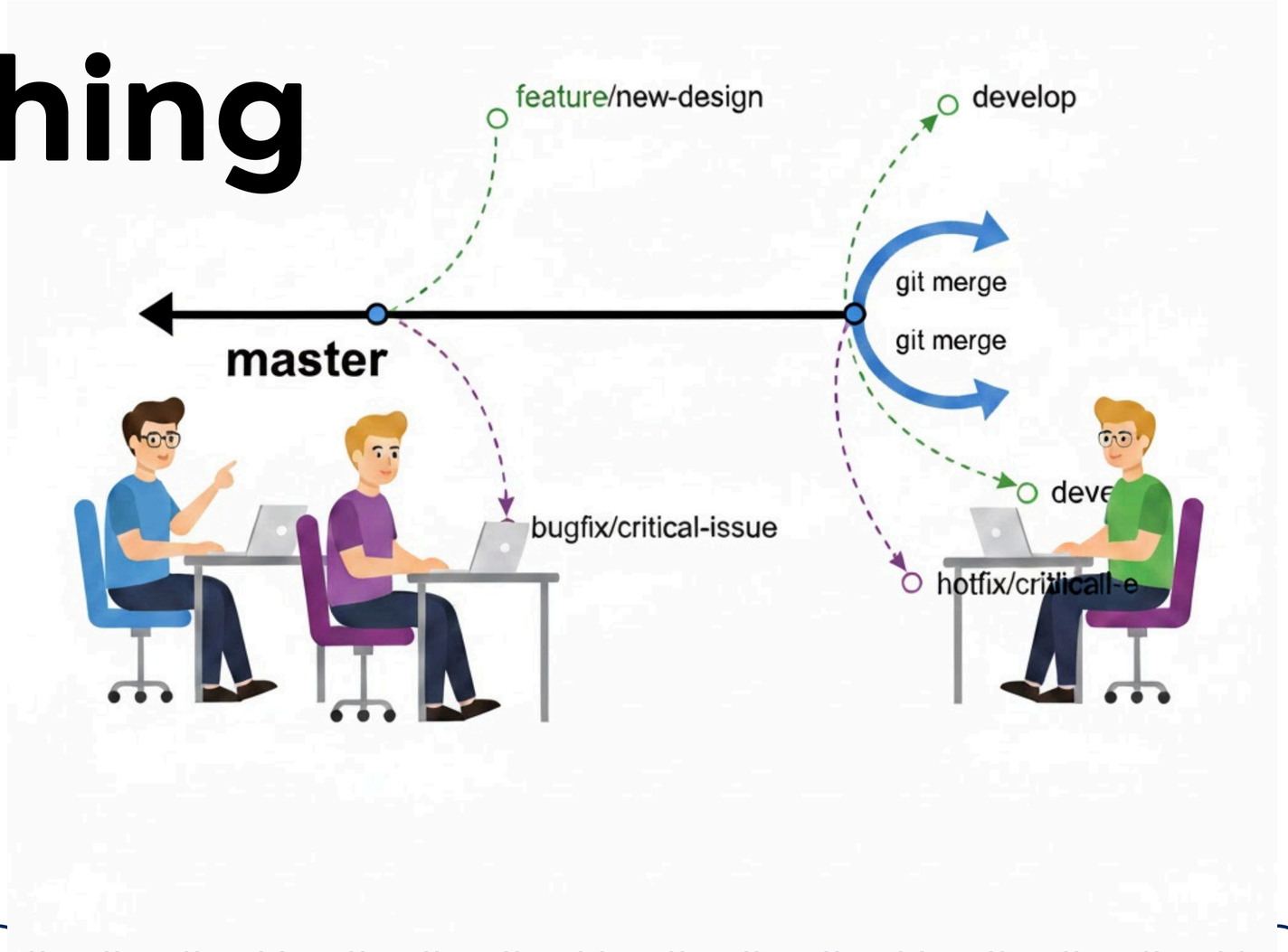



```
$ git rm args.py test.py
```

You can remove several files or folders at once using git rm, just by listing them together.



Branching

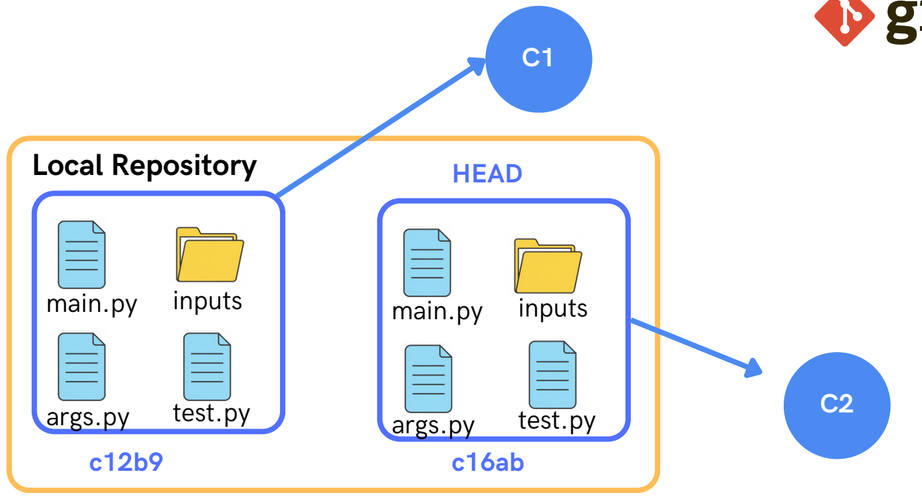

feature/new-design
develop
git merge
git merge
master
bugfix/critical-issue
deve
hotfix/critical-e

What is a branch in git?

A branch is like a copy of your project where you can make changes safely without affecting the master branch.

🧠 **Why Use Branches?**

- 🧩 Work on new features or bug fixes separately
- 👥 Collaborate: everyone can have their own branch
- 🔄 Test ideas without breaking the main project



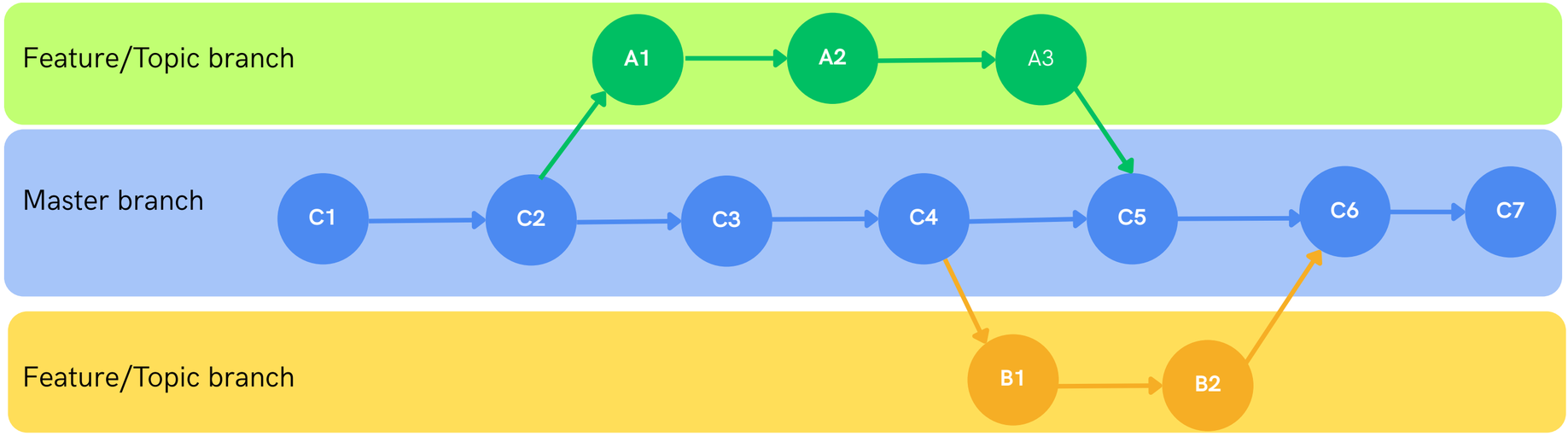
For simplicity, each Git commit will be represented by a small circle

Master branch:

- Represents the official, stable version of the project.
- Always kept clean and functional.
- All completed features or fixes from other branches are merged back here after review and testing.

Feature/Topic branch:

- Created for a specific task: new feature, bug fix, or experiment.
- Keeps work separate from the master branch.

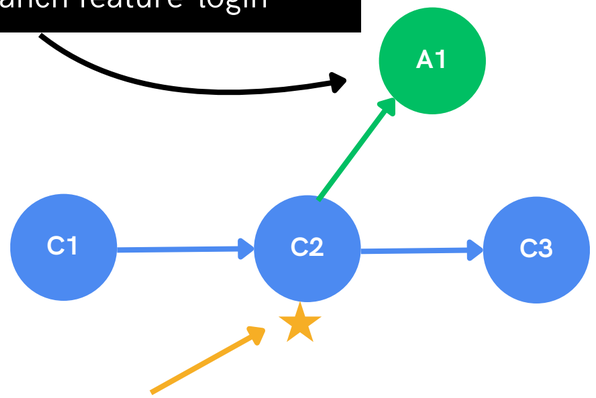


What can you do with the git branch command?

1 Create a New Branch

Create a branch named feature-login for a new feature.

```
$ git branch feature-login
```



The command you used successfully created the new branch, but it did not include the instruction to switch to it.
Stays on your current branch.

2 Check Branch Status

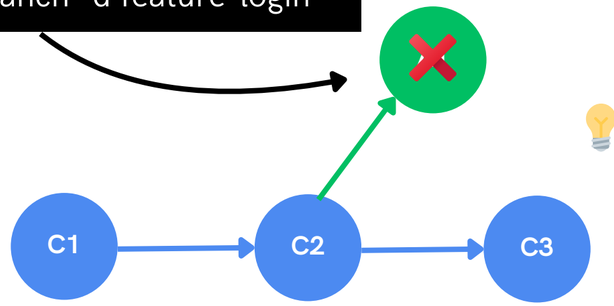
```
$ git branch
  master
* feature-login
```

- See all branches in your repository.
- The current branch is marked with an asterisk *.

3 Delete an Unused Branch

Once your branch is merged and no longer needed, clean it up.

```
$ git branch -d feature-login
```



💡 Use **-D** instead of **-d** to force deletion if it's not merged yet.

★ Current branch

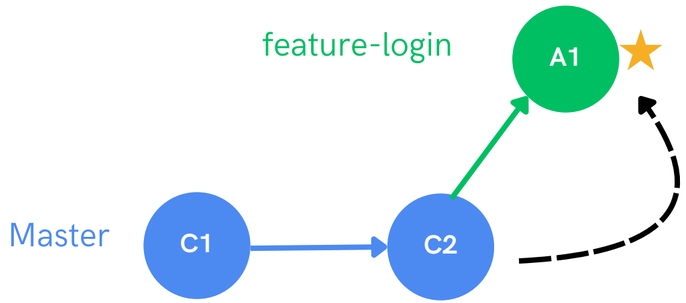
How do you switch between branches?

💡 Why This Matters

Switching branches lets you move between different versions of your project. For example, working on a new feature without touching the main code.

```
$ git checkout feature-login
```

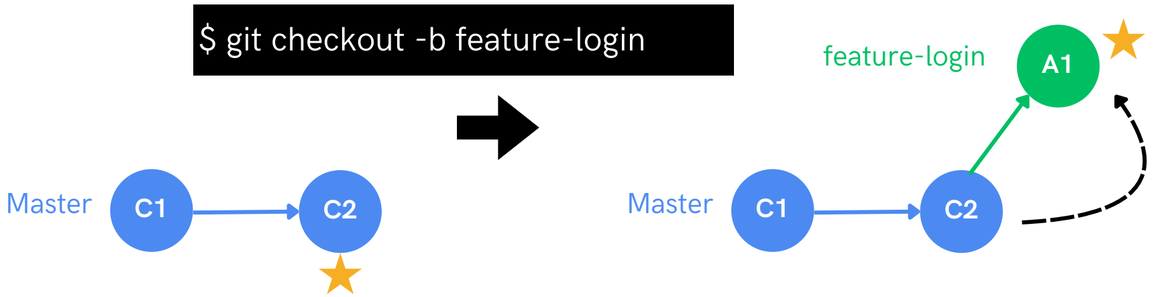
Moves you from your current branch (e.g., master) to the branch named feature-login.



👉 The branch **feature-login** must already exist, when you run the command: $ git checkout feature-login.

If you want to create and switch at the same time:

```
$ git checkout -b feature-login
```



✅ This creates a new branch named feature-login and switches to it immediately.

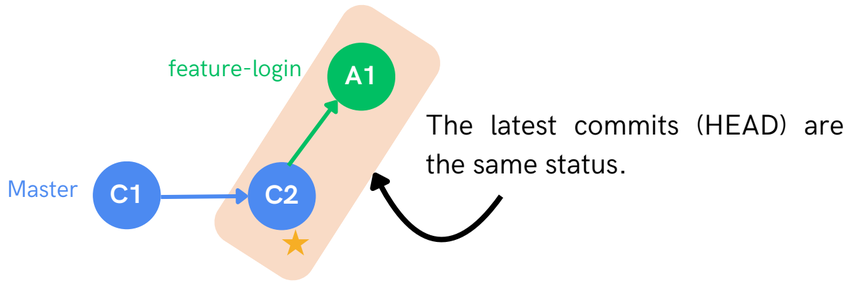
- There's a newer Git command that simplifies branch switching: **git switch**.
- It focuses only on switching branches, unlike git checkout which also handles files.

★ Current branch

When can I switch branches without any problem?

You can switch branches smoothly when both branches share the **same latest commit,** meaning no conflicting changes between them.

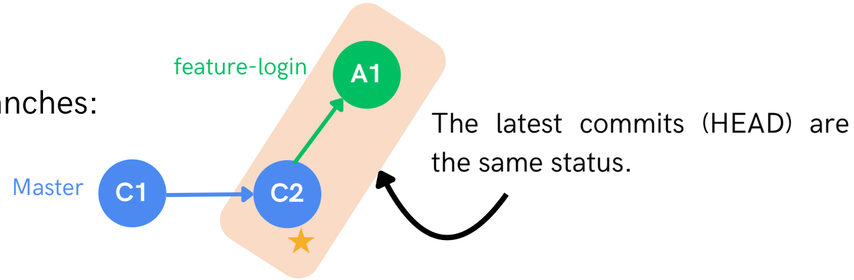
- ✅ Git can safely carry over your current uncommitted changes.
- ✅ The working directory and staging area (index) are updated to match the target branch.
- ✅ Your edits remain, nothing is lost!



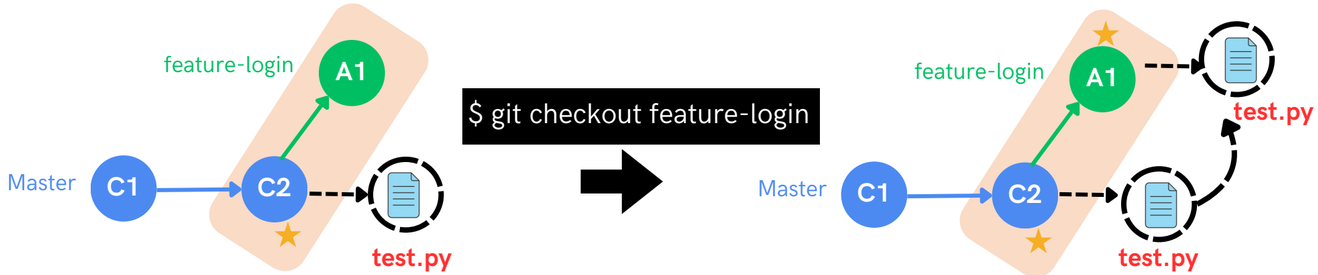
🧩 Example

Imagine you have two branches:

- master
- feature-login



Both branches point to the same latest commit, they're identical for now.



You edit **test.py** (but haven't committed yet)

✅ What Happens:

Git notices that feature-login and master have the same base commit, so it can carry your uncommitted change (test.py) safely to the new branch.
Your working directory still contains your edit, and now you're on the feature-login branch.

Changes in the Working Tree and/or INDEX (Not committed yet)

★ Current branch

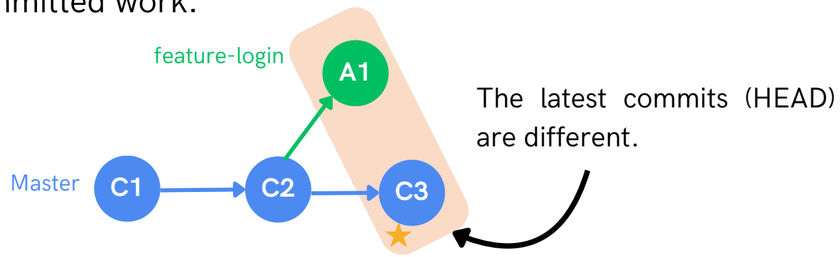
Why does git stop me from switching branches?

You cannot switch when your current branch and the destination branch have different latest commits. Git can't safely merge your working changes.

- ⚠️ Git detects conflicting file versions between branches.
- 🚫 It blocks the switch to prevent data loss or overwrite.
- 💬 You'll see an error like:

```
error: Your local changes to the following files would be overwritten by checkout:
  main.py
Please commit your changes or stash them before you switch branches.
```

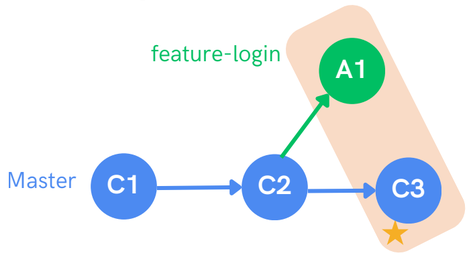
- That's because files (e.g., main.py) may have diverged, meaning the same files were changed in both branches.
- Git blocks the switch to prevent overwriting or losing your uncommitted work.



The latest commits (HEAD) are different.

★ Current branch

Example

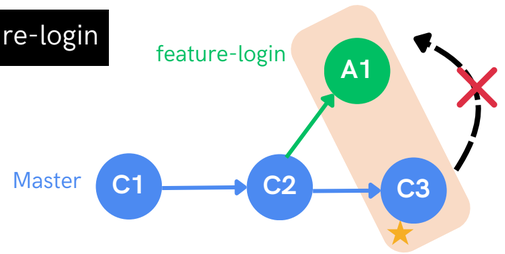


You are on the master branch and you edit **test.py**

```
$ git checkout feature-login
```

Now you try to switch to another branch

Git refuses the switch



But in feature-login, that same file (**test.py**) was already modified and committed differently

How to Fix It

You have three options:

1

Commit your work on main first:

```
$ git add test.py
$ git commit -m "Update print message"
$ git checkout feature-login
```

2

Temporarily stash changes:

```
$ git stash
$ git checkout feature-login
$ git stash pop
```

We'll cover git stash in the next slide.

3

Use force checkout

```
git checkout -f feature-login
```

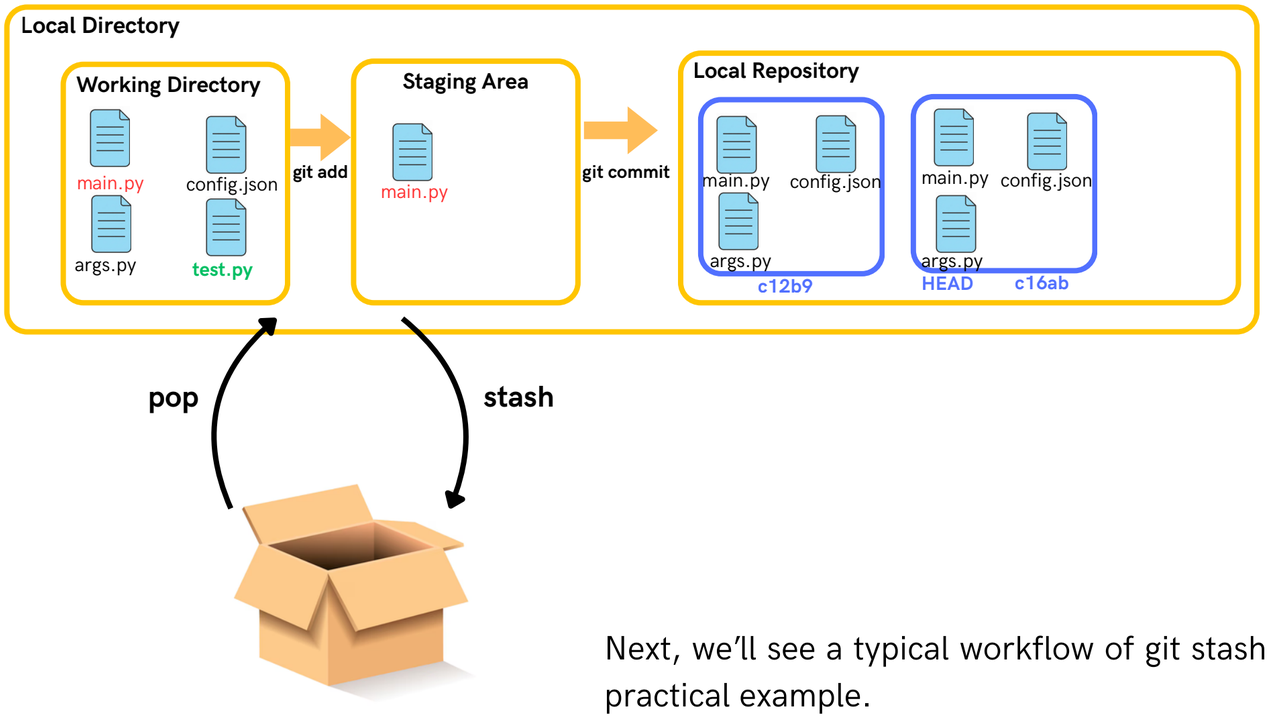
This means you lose any uncommitted edits permanently. ❌

What is a stash in git, and why would you use it?

A stash is a temporary storage area in Git where you can save uncommitted changes in your working directory and staging area.

🎯 Purpose

- Temporarily save your work-in-progress.
- Switch branches or perform other Git operations without committing unfinished changes.
- Keep your master branch clean while working on experimental or incomplete features.



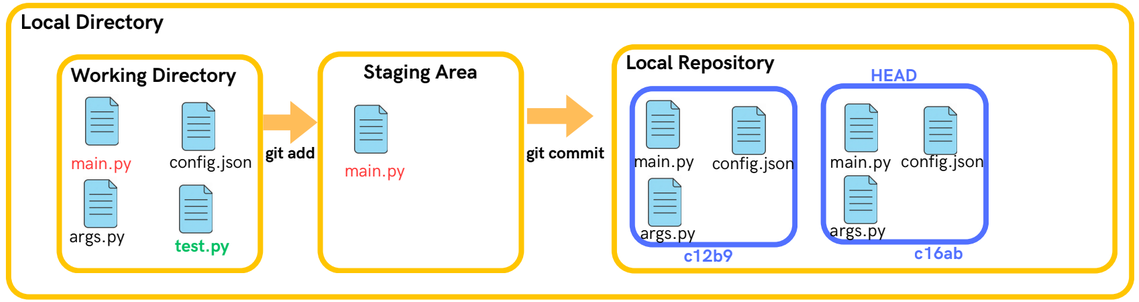
Next, we'll see a typical workflow of git stash through a practical example.

How can I temporarily save my work?

💡 **Scenario:**

You're working on the master branch, fixing a bug in main.py, when your teammate asks you to quickly switch to another branch (feature-login) to check something.

But… you haven't finished your bug fix yet, so you don't want to commit these unfinished changes.



1. Check current branch and files

```
$ git status
On branch master
Changes not staged for commit:
  modified: main.py
```

You have changes in your working directory (not committed yet).

2. Stash your current work

```
$ git stash save "WIP: fixing bug in main.py"
Saved working directory and index state WIP on master: fixing bug in main.py
```

3. Check current branch

```
$ git status
On branch master
nothing to commit, working tree clean
```

4. Switch to another branch safely

```
$ git checkout feature-login
Switched to branch 'feature-login'
```

5. When done, go back to your original branch

```
$ git checkout master
Switched to branch 'master'
```

6. Reapply your stashed work and remove it

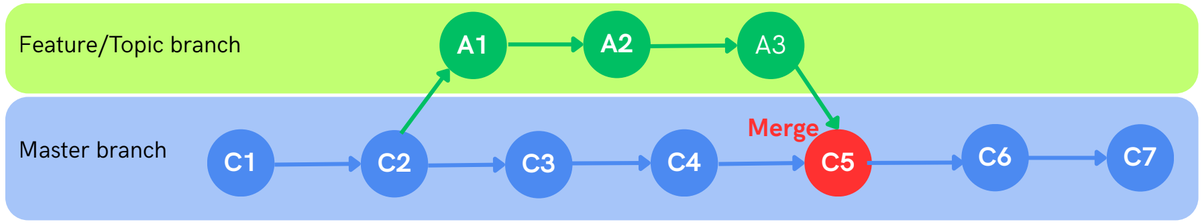
```
$ git stash pop
```

- Restores your changes in main.py to the working directory.
- Automatically deletes this stash from the stash list.

```
$ git stash list
# (empty)
```

How does git merge branches together?

When you finish working on a feature or fix in a separate branch, you'll want to combine your changes into the main project. That's where git merge comes in. It joins different lines of development into one.



When two branches come together, Git can merge them in different ways, depending on their history and your goal. Each method tells Git how to combine the changes.

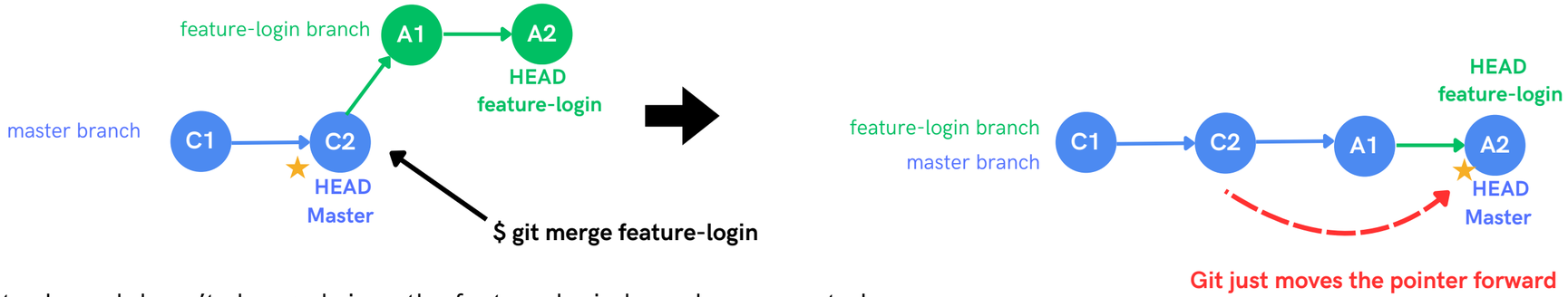
Four Merge Types

- Fast-Forward Merge → Moves the branch pointer forward.
- Non-Fast-Forward Merge → Creates a merge commit when histories differ.
- --no-ff Merge → Forces a merge commit even if a fast-forward is possible.
- Squash Merge → Combines all commits into one clean commit.

In the next slides, we'll explore each merge type with simple diagrams

What is a Fast-Forward merge?

- A fast-forward merge happens when the target branch (like master) hasn't changed since the feature branch was created.
- Git simply moves the branch pointer forward to the latest commit => no new merge commit is created.
- The result is a clean and linear history.
- It works only if there are no new commits on the target branch.

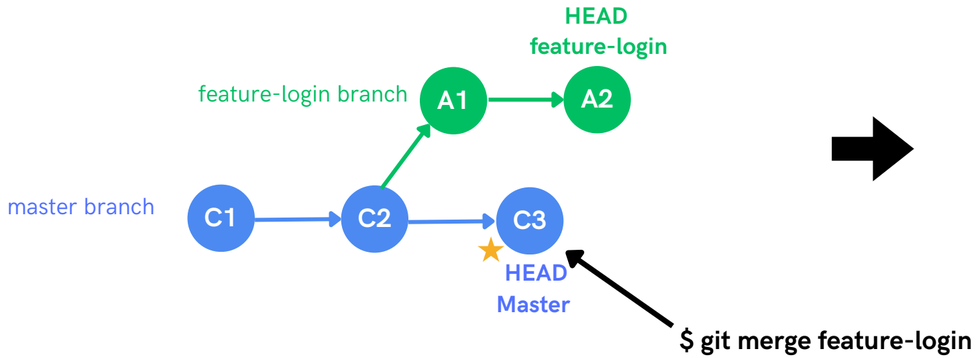


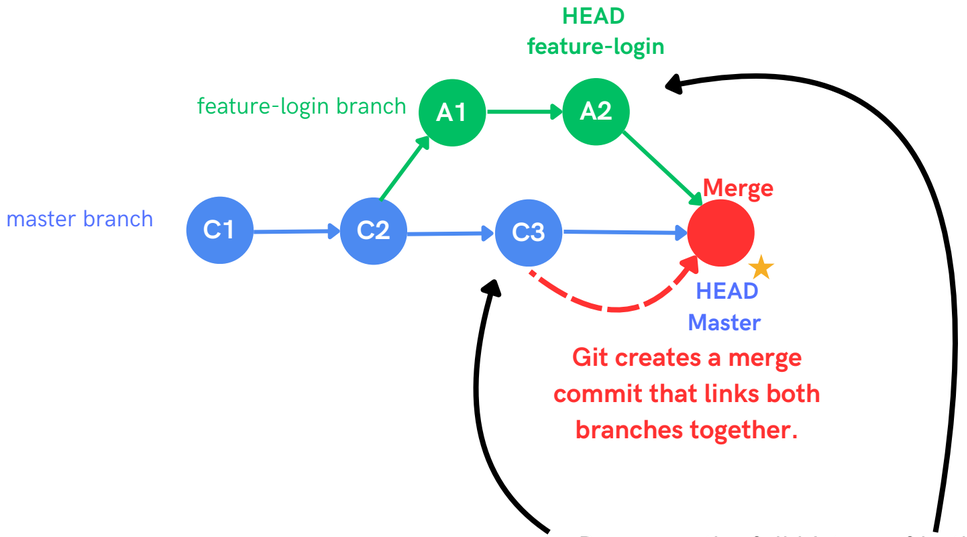
the master branch hasn't changed since the feature-login branch was created.
only C1 and C2

★ Current branch

What happens in a Non-Fast-Forward merge?

When the target branch (e.g., master) has new commits that the feature branch does not, Git cannot just move the branch pointer forward.
It needs to create a merge commit to combine the histories.



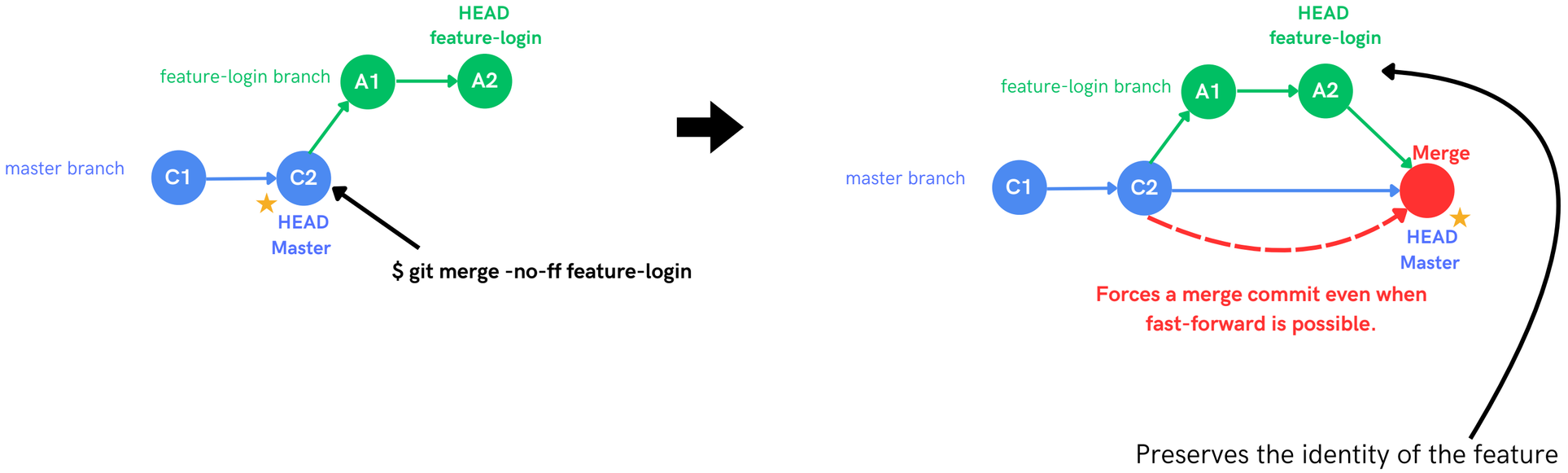


Preserves the full history of both branches.
Every commit from both branches (C1, C2, C3, A1 and A2) remains visible in Git history.

★ Current branch

What is a git merge with --no-ff?

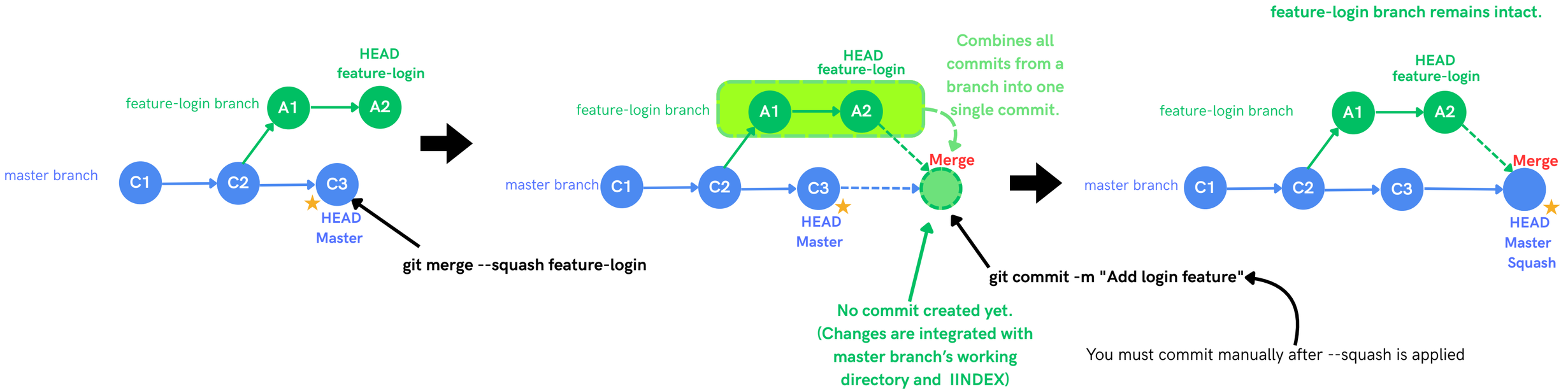
Even if Git could do a fast-forward merge, using --no-ff forces a merge commit.
This keeps the feature branch history clear and shows exactly when it was integrated.



★ Current branch

What is a squash merge in git?

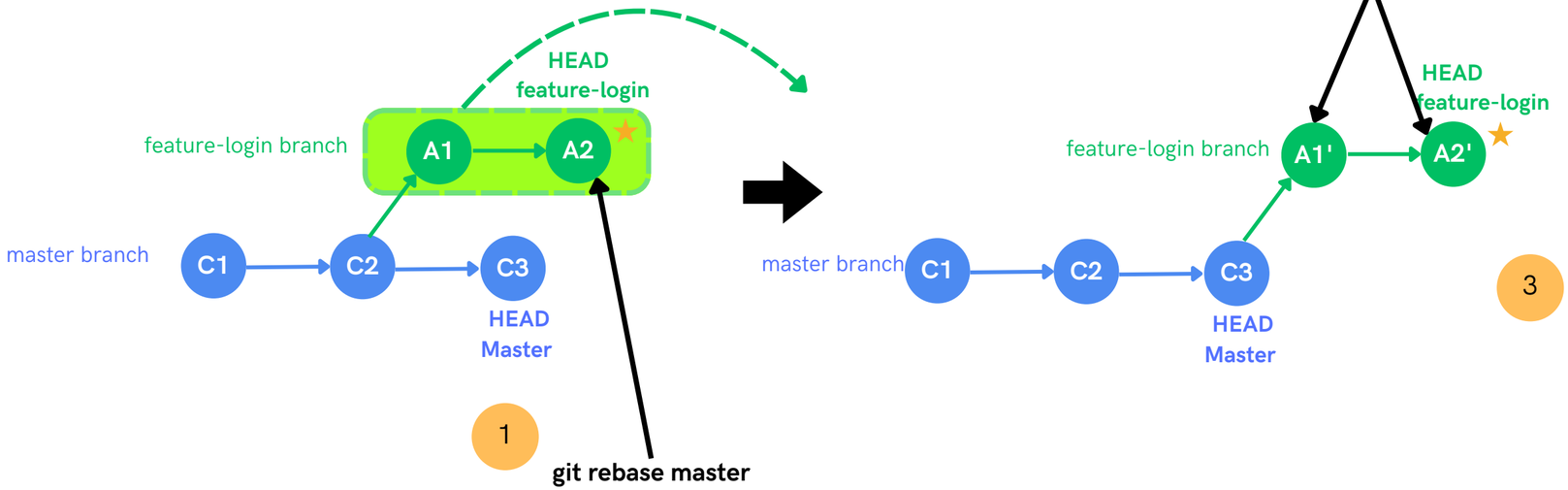
A squash merge lets you combine all changes from a feature branch into a single commit when merging, keeping the main branch history clean and concise.
Useful for small features or bug fixes with multiple small commits.



★ Current branch

What does git rebase really do?

git rebase lets you move your branch's commits so they appear on top of another branch's latest commit.
It helps keep your project history linear and clean, as if your work started from the most recent version of the base branch.



Rebase it onto the base branch (usually master):

2 Git doesn't just move your commits:

- It actually recreates them on top of the latest commits from master.
- Each of your old commits is copied and given a new commit ID (SHA).
- That's why we say "rebase rewrites history."
- A1' and A2' are new commits, even though their content is the same as A1 and A2.

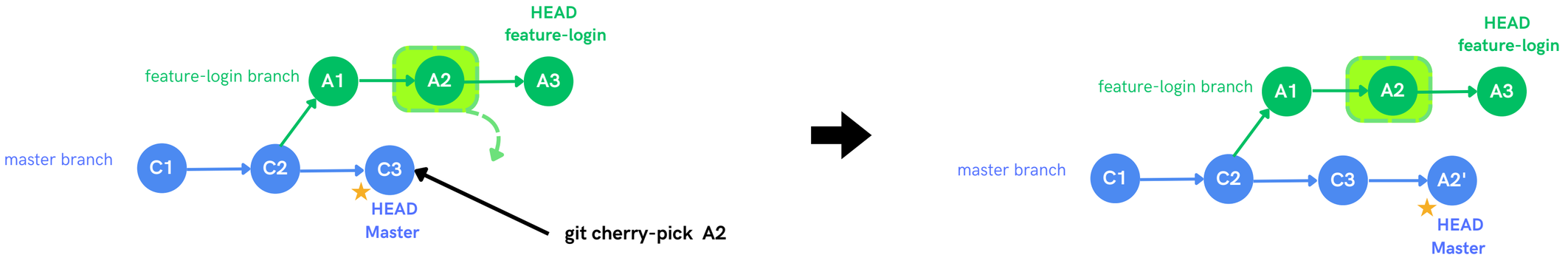
3 After rebasing, when you merge the topic branch into the base branch (master), a **fast-forward merge** is typically possible because the topic branch now sits on top of the base branch.

★ Current branch

What is git cherry-pick?

💡 **Problem**:

Sometimes, you want to apply a **specific commit** from one branch to another without merging the entire branch.
For example, maybe you fixed a bug on **feature-login** and want the same fix on **master**.



You cherry-pick a commit to master branch

🗝️ **Key Points:**

- 🎯 Picks specific commits instead of merging a whole branch.
- 🧱 Useful for hotfixes or isolated bug fixes.

★Current branch

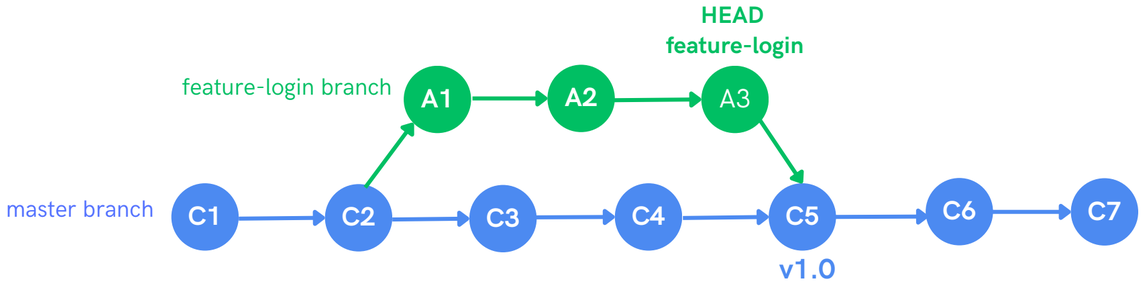
What is git tag and why do we use it?

💡 **Problem**:

When a project reaches an important point (a release, a milestone, a stable version), you need a way to mark that specific commit so you can always return to it easily.

🧩 **Explanation**

- A Git tag is a label attached to a commit, usually used to mark releases such as v1.0, v2.5, etc.
- Tags do not move like branches; they stay fixed on one commit forever.



There are two types of tags:

1 **Lightweight tag**

A simple name pointing to a commit.

$ git tag v1.0

2 **Annotated tag**

Includes message, author, date (recommended for releases).

$ git tag -a v1.0 -m "Release version 1.0"

🗝️ **Key Points:**

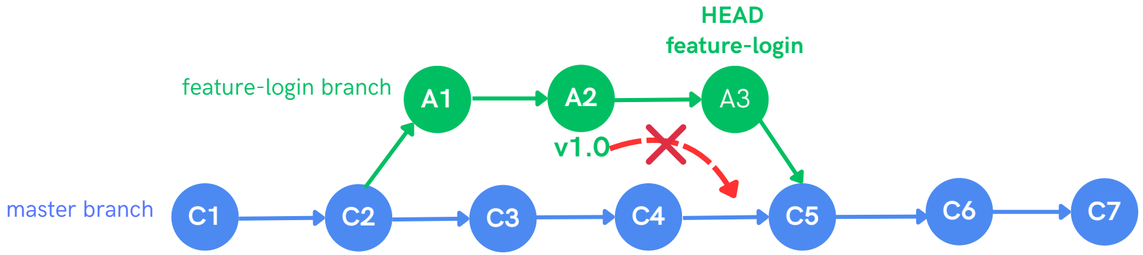
- 🏷️ Tags identify important commits (release versions).
- 🔒 Tags are immutable markers, they don't change automatically.

Do tags follow when merging a branch?

❌ **No: Git tags do NOT follow during a merge**

If you create a tag on a commit (for example A2 in feature-login) and then you merge the branch into master:

- ➡️ The tag stays attached to commit A2.
- ➡️ It does NOT move to the merge commit.
- ➡️ It does NOT automatically appear on the master branch.



✔️ **Why?**

- Because tags point to commits, not branches.
- Merging brings commits, but does not copy the tag reference.

🗝️ **Key Points:**

- 👉 Tag v1.0 is pointing to commit A2 on the feature branch only.
- 👉 If you merge this branch, the tag does not follow.

How do you manage a merge conflict in git?

A merge conflict happens when Git can't decide which version of a file to keep because two people made changes to the same part of the code.
You must manually choose or combine the correct version before continuing.

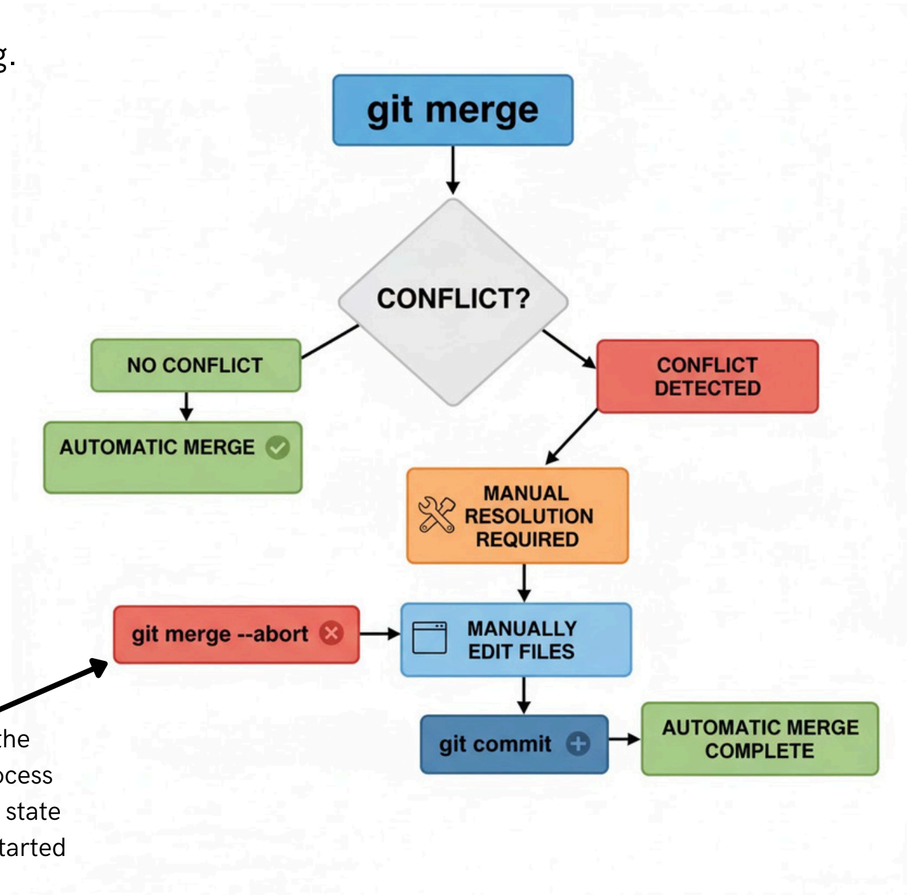
⚙️ **Steps to Manage a Conflict:**

1. Git detects the conflict and stops the merge.
2. Open the conflicted file, you'll see conflict markers:

```
<<<<<<< HEAD
your version
=======
incoming version
>>>>>>> branch-name
```

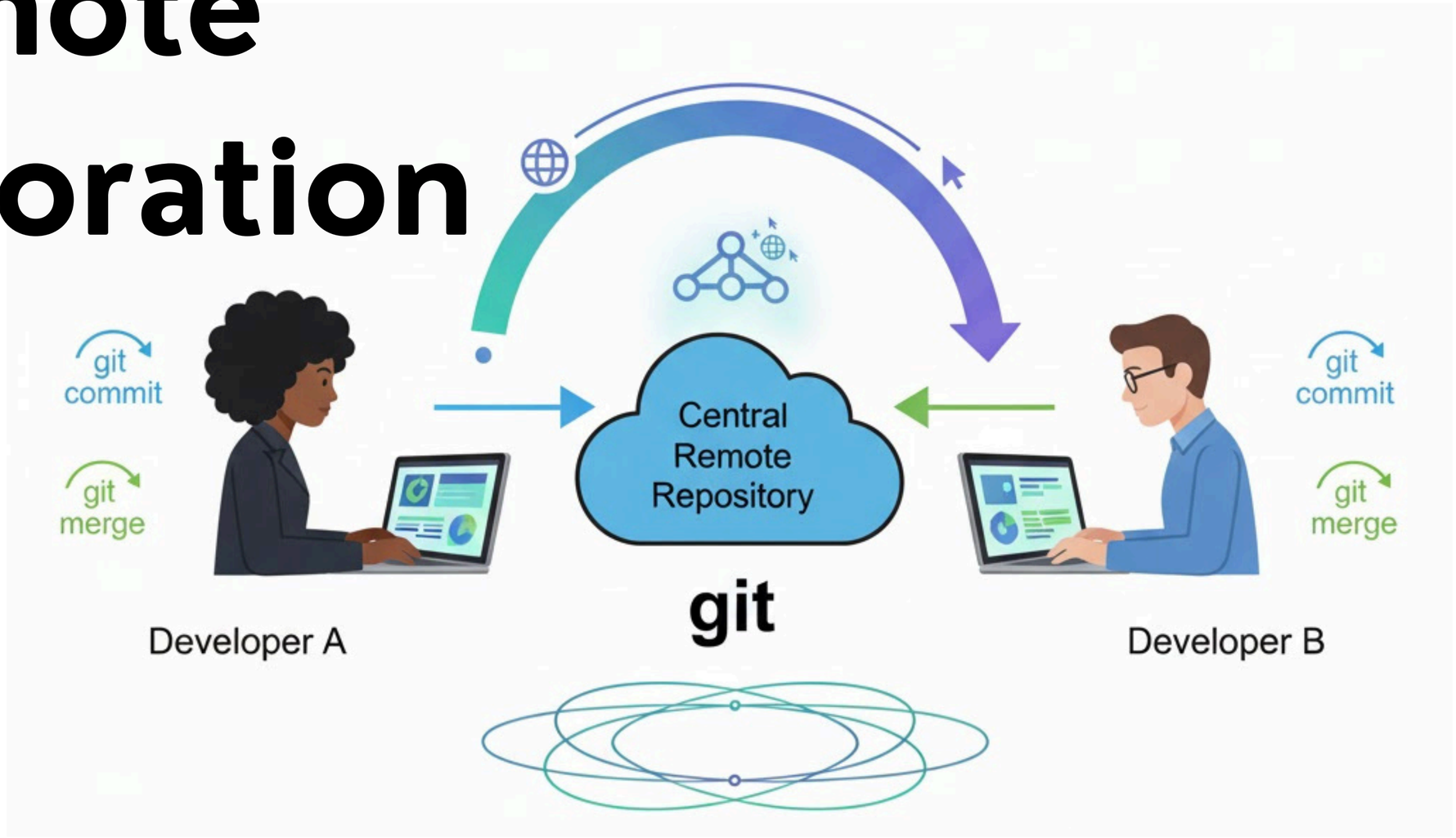
3. Choose one of the following:
   - ✅ Accept current change
   - 📥 Accept incoming change
   - 🔀 Keep both changes
   - ✏️ Edit manually
4. After resolving:

```
$ git add <file>
$ git commit
```



you can cancel the current merge process and go back to the state before the merge started

Remote Collaboration



How does remote collaboration work in git?

Developer A

Developer B

Remote Repository

1. A links local and remote repositories
2. A pushes the project online
3. A gives access to B
4. B clones the project for the first time
5. B updates local repository
6. B creates a topic branch to work on a new feature
7. B edits, commits, and tests locally
8. B pushes the topic branch into the remote repository
9. B creates a Merge Request/ Pull Request (MR/PR).
10. A reviews the changes and gives feedback
11. After approval, A merges the topic branch into master

How to link a local repository to a remote one?

1. What does linking mean?

- Linking means connecting your local Git project (on your computer) with a remote repository (on GitHub, GitLab, etc.)
- This allows you to push, pull, and collaborate with others.

Developer A

1 A links local and remote repositories

Remote Repository

2. What's the basic syntax?

```
git remote add origin <remote_repository_URL>
```

- `<remote_repository_URL>` is the HTTPS or SSH URL of your remote repo
- `origin` is a default name (you can choose another name if needed)
- `git remote add` adds a remote connection

Use this command to list linked remotes:

```
$ git remote -v
origin  https://github.com/username/my-project.git (fetch)
origin  https://github.com/username/my-project.git (push)
```

You can rename or delete a remote later:

```
$ git remote rename origin mainrepo
$ git remote rm origin
```

Updates the existing remote's URL

git remote set-url <name> <newURL>

```
git remote set-url origin https://github.com/user/new-repo.git
```

How to push changes to remote?

🧭 What is git push?

- git push is used to upload your local commits to the remote repository (like GitHub or GitLab).
- It keeps your remote project up to date with your latest changes.
- You can push new commits, or even create new branches on the remote.

Developer A

2 A pushes the project online

Remote Repository

🪜 2. What's the basic syntax?

```
git push <remote-name> <branch-name>
```

branch to push

default remote name

How do you push a branch for the first time?

? How do you push a new branch?

- You created a new branch locally. How do you make it appear on the remote for the first time?

You can use of -u (or --set-upstream) option

When pushing a branch for the first time, using -u sets the remote branch as upstream, so future pushes can be simpler:

```
$ git push -u <remote-name> <branch-name>
```

After that, you can just use **$ git push**

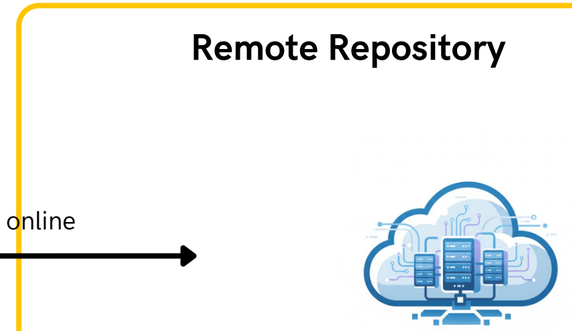
★Current branch

What happens if the push fails?

? What if someone else pushed first?

- Sometimes, Git rejects your push because the remote branch has new commits that your local branch doesn't have.
- This usually happens when someone else has pushed changes to the same branch before you.

Developer A



Git rejects push if your branch has diverged:

```
To https://github.com/user/repo.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/user/repo.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

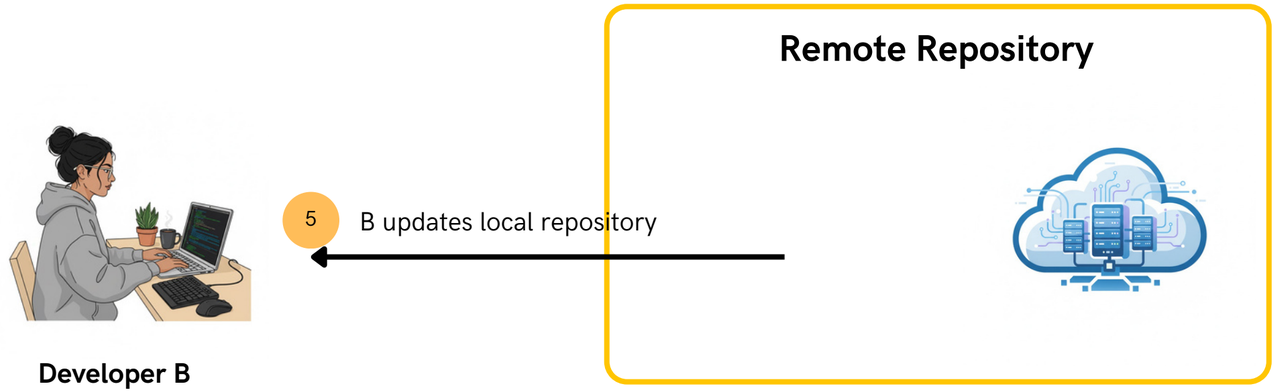
Your local branch is behind the remote branch, someone else has pushed new commits.
You need to update your branch before pushing again: **git pull**

We'll cover git pull in the next slide.

What does git pull do?

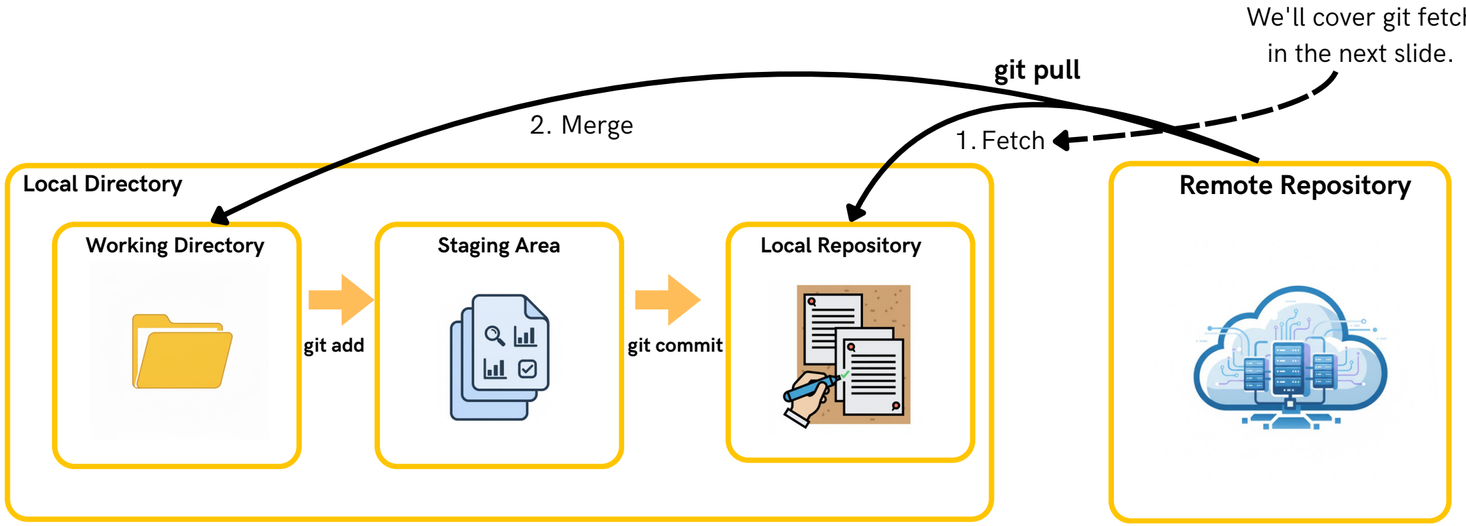
Problematic:

- When working with a team, other developers might update the remote repository while you're still working locally.
- You need a way to bring those new updates into your local branch => that's what git pull does!



What is git pull?

git pull is used to download (fetch) changes from the remote repository and merge them into your local branch automatically.



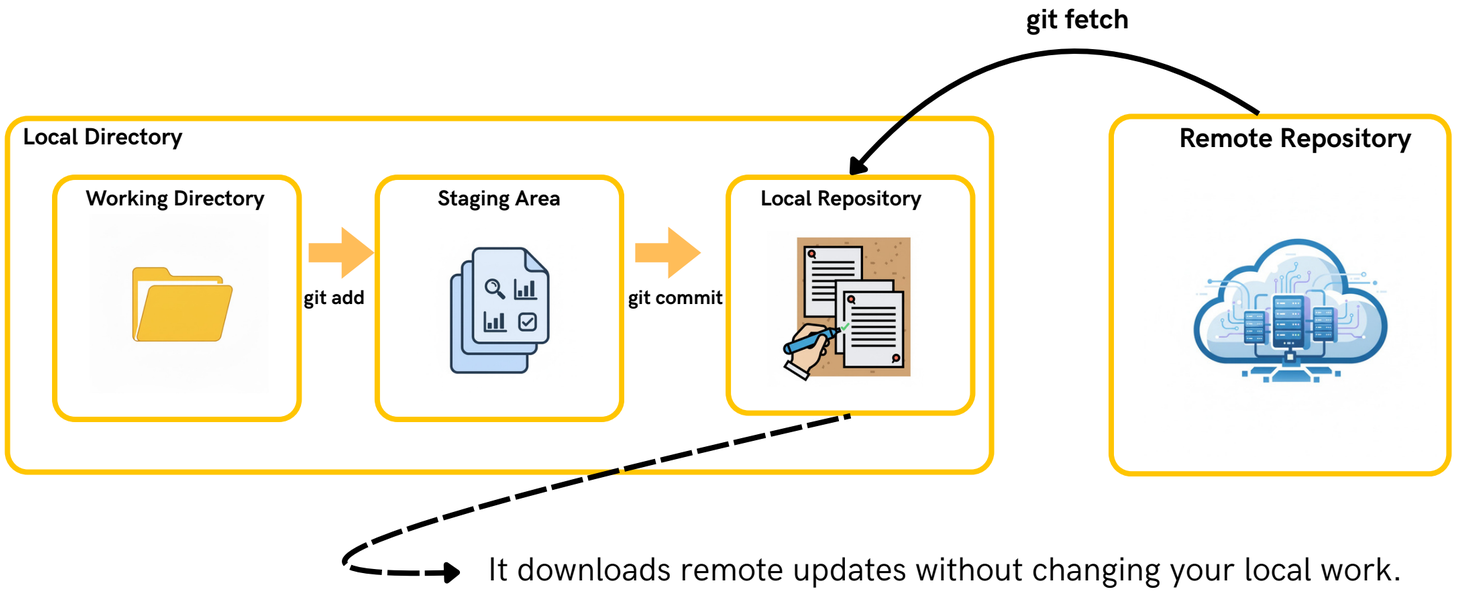
How It Works

1. Fetch - Git checks the remote repository for any new commits.
2. Merge - Git integrates those commits into your current local branch.

What does git fetch do?

Problematic:

- When working with others, changes might be made on the remote repository that you don't have locally.
- You need a safe way to check for updates without affecting your local work.



💡 What is git fetch?

- **git fetch** downloads all new commits, branches, and tags from the remote repository.
- It updates your remote-tracking branches (like origin/main) but does not merge them automatically.
- Your working directory stays unchanged, so it's safe to use anytime.

🗝️ Key Points:

✅ Safe: doesn't modify local branches.
🔍 Lets you review updates before merging.
🤝 Keeps your local repository aware of remote changes.

These updates are stored in special branches called **remote-tracking branches**.

let's see what they are in the next slide.

What happens behind the scenes after git fetch?

Problematic:

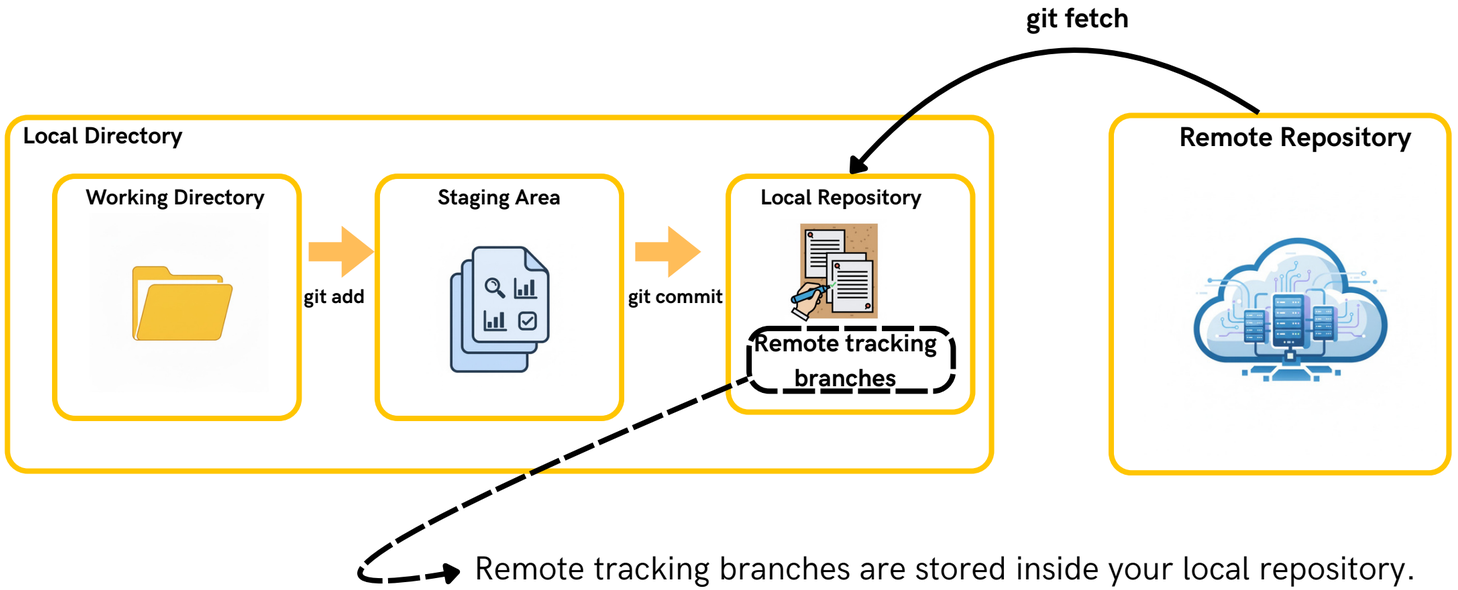
- After running git fetch, your local files don’t change…
- So where do the fetched updates go?

🧩 Explanation:

- The fetched updates are stored in remote tracking branches, such as:

```
origin/main
origin/feature-login
```

- These are read-only snapshots of branches from the remote repository.
- They allow you to compare or merge updates later without modifying your working directory.



🧠 Example:

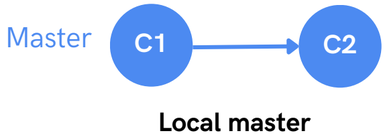
```
$ git branch -a
* master
  feature-login
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/feature-login
```

Here, remotes/origin/… shows your remote tracking branches.

Let’s explore a simple example to better understand how git fetch works in action.

What happens after git fetch?

What's my repository's state before fetching?



Local master

🧠 Your remote-tracking branch origin/master in your local repo doesn't yet know about the new commit C3 that exist on the remote server. **(outdated)**

Remote-tracking branch (origin/master)

Your teammate pushed new commit C3 to the remote.

Remote (on server)

git fetch

What updates after I run git fetch?

your working directory still reflects C2 => no files have changed yet.

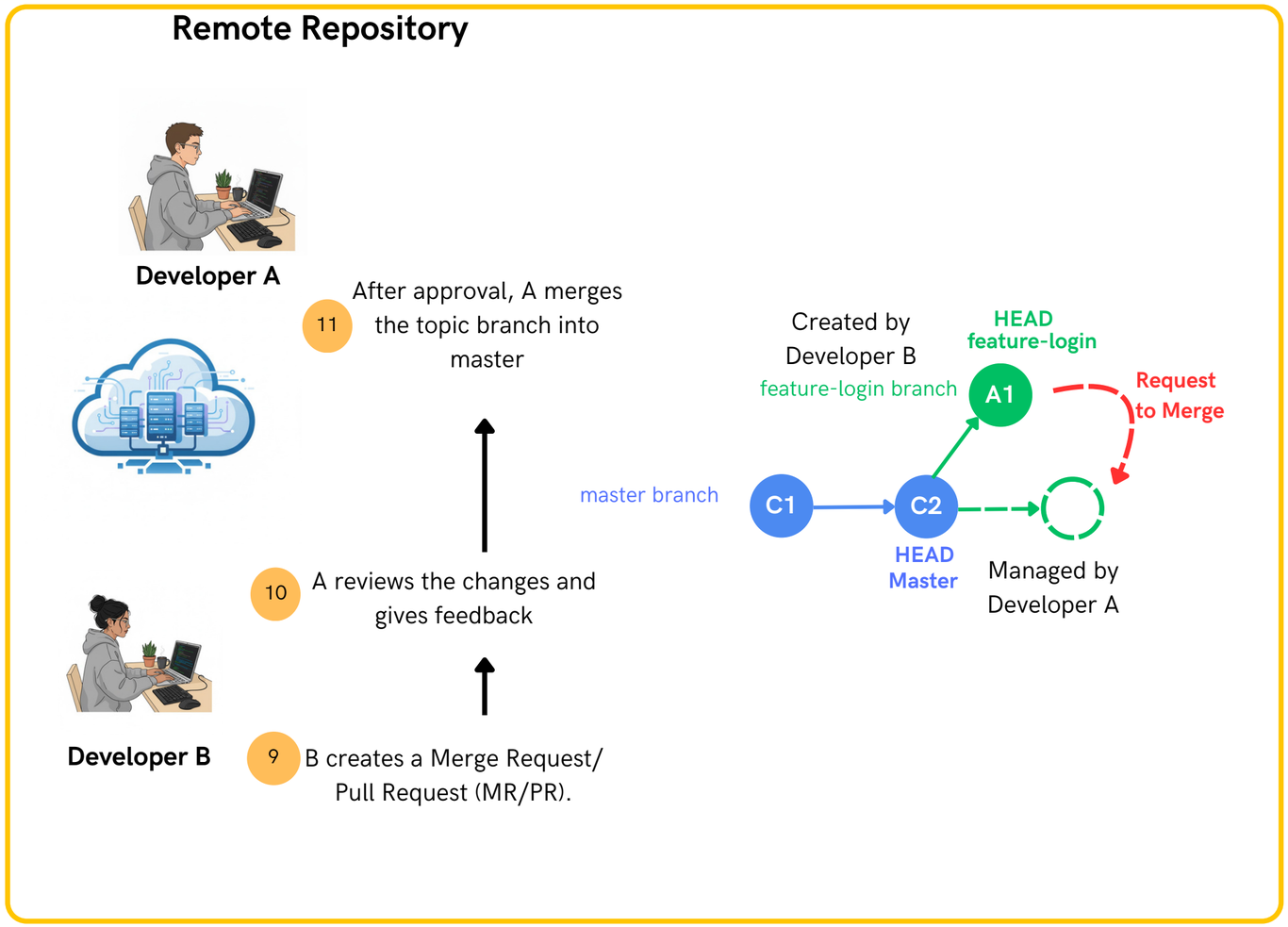
Local master

✨ After fetching, your local repo now knows about the latest commits on the remote. **(updated)**

Remote-tracking branch (origin/master)

Remote (on server)

What is a PR/MR and how does it work (1/2)?



💡 Context:

You've finished your feature or bug fix on a topic branch
=> now it's time to share it!

🚀 Step-by-Step

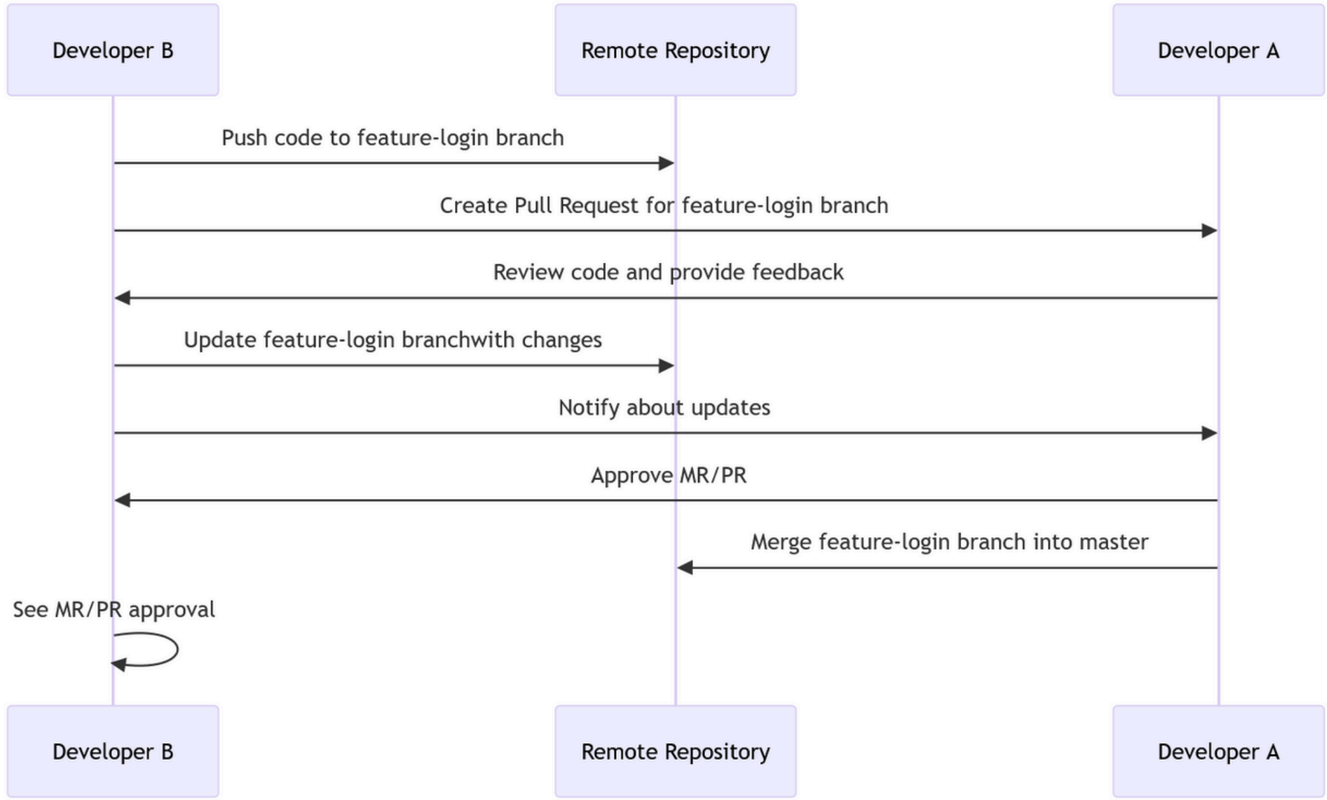
1. Push your branch to the remote repository.
2. Open a Pull/Merge Request (PR/MR), a formal request to merge your work.
3. Select branches:
   - Base branch: usually master (where you want your code merged).
   - Compare branch: your topic branch with the new commits.
4. Describe your changes (title + explanation).
5. Request reviews from teammates.

👀 Review & Merge Flow

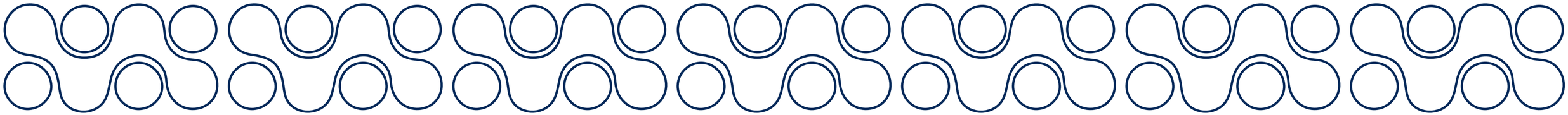
- Reviewers inspect commits and file differences.
- They can:
  ✅ Approve 💬 Comment 🛠️ Request changes
- You can update your branch, the PR/MR refreshes automatically.
- Once approved → Merge into master → 🎉 Done!

What is a PR/MR and how does it work (2/2)?

1. **Push Code:** Developer B creates a new feature on feature-login branch and pushes it to the remote repository.
2. **Create Pull/Merge Request:** Developer B requests Developer A to review and merge the branch.
3. **Code Review:** Developer A reviews feature-login branch and provides feedback to Developer B.
4. **Update Branch:** Developer B updates feature-login branch based on feedback and pushes the changes.
5. **Approval & Merge:** Developer A approves the PR/MR and merges feature-login branch into the master branch.
6. **Confirmation:** Developer B sees that the PR/MR has been approved and merged successfully.



Git Key Commands



Git Key Commands

1 Initial Setup

```
# Set your name
git config --global user.name "Your Name"

# Set your email
git config --global user.email "your_email@example.com"

# Set your default editor
git config --global core.editor "code --wait"   # VS Code
# git config --global core.editor "vim"
# git config --global core.editor "nano"

# Enable colored output
git config --global color.ui true

# Set default branch name
git config --global init.defaultBranch master

# Set merge tool
git config --global merge.tool vscode

# Set diff tool
git config --global diff.tool vscode

# View all config
git config --list

# Edit config file directly
git config --global --edit

# Remove a config entry
git config --global --unset user.email
```

2 Project Setup

```
# Initialize a new Git repository in the current directory
git init

# Clone an existing remote repository
git clone <repository_url>

# Clone into a custom folder name
git clone <repository_url> <folder_name>

# Clone only a specific branch
git clone -b <branch_name> <repository_url>
```

3 Edit & Commit

```
# Stage a single file
git add <file>

# Stage multiple files
git add <file1> <file2>

# Stage all changes in the repository
git add .

# Check the status of files
git status

# Commit staged changes with a message
git commit -m "Your commit message"

# Commit all changes (including unstaged files)
git commit -a -m "Your commit message"

# Amend the last commit (edit message or add changes)
git commit --amend

# Show commit history
git log

# Show a compact, one-line log with graph
git log --oneline --graph --decorate

# Show changes between working directory and staging area
git diff

# Show staged changes ready to commit
git diff --staged

# Show changes for a specific file
git diff <file>

# Discard changes in working directory for a file
git restore <file>

# Unstage a file (keep changes)
git restore --staged <file>

# Remove a file and stage deletion
git rm <file>
```

Git Key Commands

3 Edit & Commit

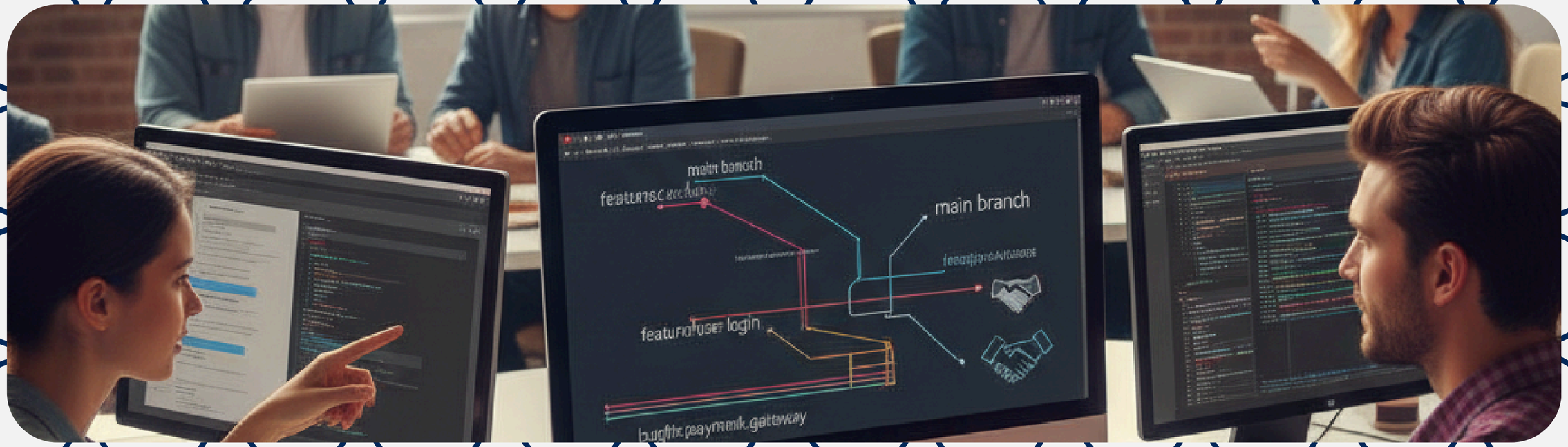
```
# Undo last commit but keep changes staged
git reset --soft HEAD~1

# Undo last commit and unstage changes (keep edits in working directory)
git reset --mixed HEAD~1
# or simply
git reset HEAD~1

# Undo last commit and discard all changes
git reset --hard HEAD~1

# Reset to a specific commit (replace <commit_hash>)
git reset --soft <commit_hash>    # keep changes staged
git reset --mixed <commit_hash>   # unstage changes
git reset --hard <commit_hash>    # discard all changes

#  Unstage a file without discarding changes
git reset <file>
```

4 Branch Management

```
# List branches
git branch # Local branches
git branch -r # Remote branches
git branch -a # All branches

# Create a new branch
git branch <branch_name>

# 🔹 Switch to a branch
git checkout <branch_name>
# or (modern alternative)
git switch <branch_name>

# Create and switch in one step
git checkout -b <branch_name>
# or (modern alternative)
git switch -c <branch_name>

# Merge another branch into current branch
git merge <branch_name>

# Merge and prevent fast-forward (always create a merge commit)
git merge --no-ff <branch_name>

#  Rebase current branch onto another branch
git rebase <branch_name>

# Delete a branch
git branch -d <branch_name>      # safe delete (checks merged)
git branch -D <branch_name>      # force delete

# Stash changes (save work in progress)
git stash
git stash list
git stash pop                # apply last stash and remove it
git stash apply <stash_id>      # apply a specific stash
git stash drop <stash_id>      # delete a specific stash
```

5 Remote Collaboration

```
# View configured remotes
git remote
git remote -v              # show remote URLs

# Add a remote repository
git remote add origin <repository_url>

# Remove a remote repository
git remote remove origin

# Rename a remote
git remote rename origin upstream

# Fetch changes from remote without merging
git fetch
git fetch --all            # fetch all remotes

# Pull changes from remote and merge
git pull

# Push local commits to remote
git push origin <branch_name>

# Force push (overwrite remote branch, use carefully!)
git push --force origin <branch_name>

# Push and set upstream tracking (first push)
git push -u origin <branch_name>
```



Get in Touch

LINKEDIN

www.linkedin.com/in/walidmallat