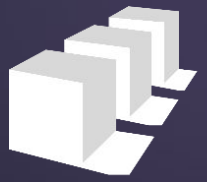
WORKSHOP 4 – RESCUER MSCA DOCTORAL NETWORK 2024-2028

Collaborating Effectively on Research Code

An Introduction to Git for Reproducible Science

Mario Morales Hernández (mmorales@unizar.es)

Universidad Zaragoza



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza

Outline

1. Why collaboration on research code fails
2. Version Control System (VCS): why Git wins for research
3. Git fundamentals (solo work)
4. Collaborative workflow on a real codebase
5. Code review, conflicts, and recovery
6. Making research code reproducible

Outline

1. **Why collaboration on research code fails**
2. Version Control System (VCS): why Git wins for research
3. Git fundamentals (solo work)
4. Collaborative workflow on a real codebase
5. Code review, conflicts, and recovery
6. Making research code reproducible

Why Are We Here?

code_fvm.py

code_fvm_updated.py

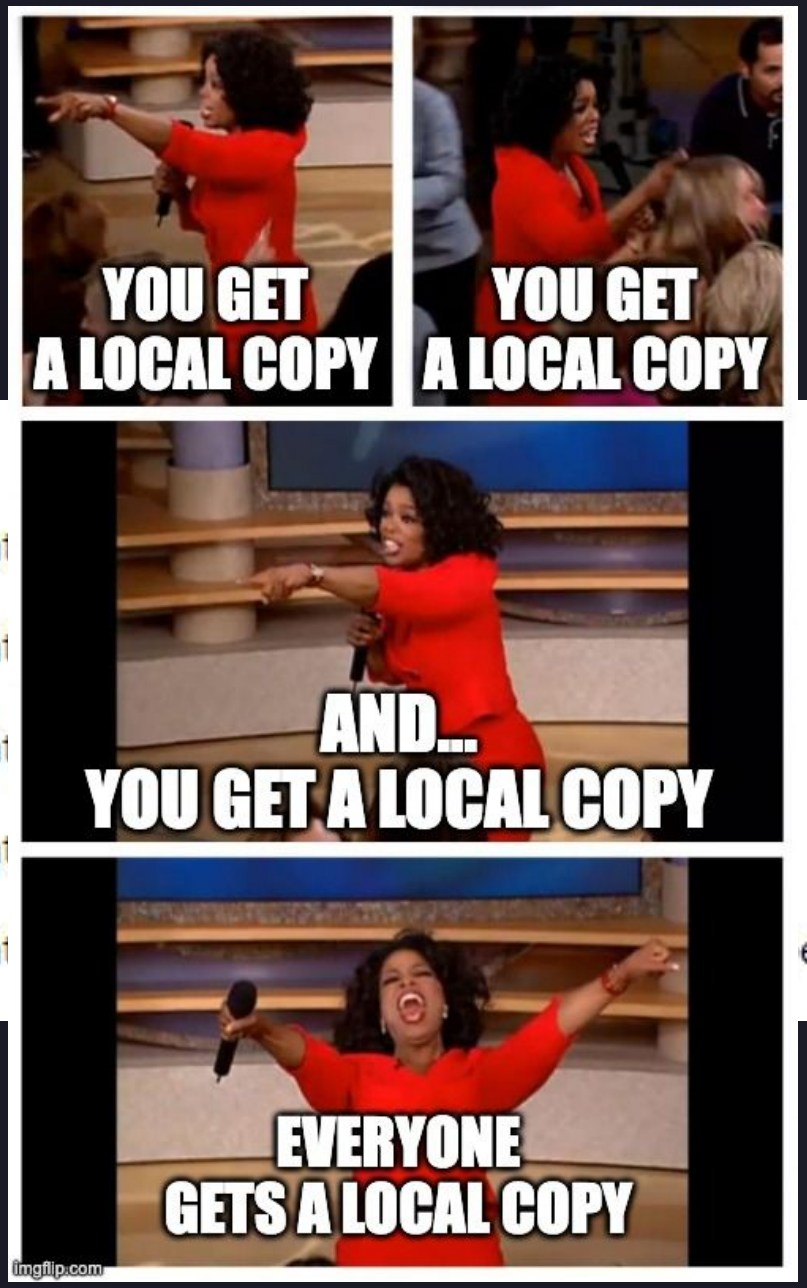
code_fvm_updated_after_summer_break.py

code_fvm_updated_after_summer_break_definitive.py

code_fvm_updated_after_summer_break_definitive_final.py

code_fvm_updated_after_summer_break_definitive_final_superfinal.py

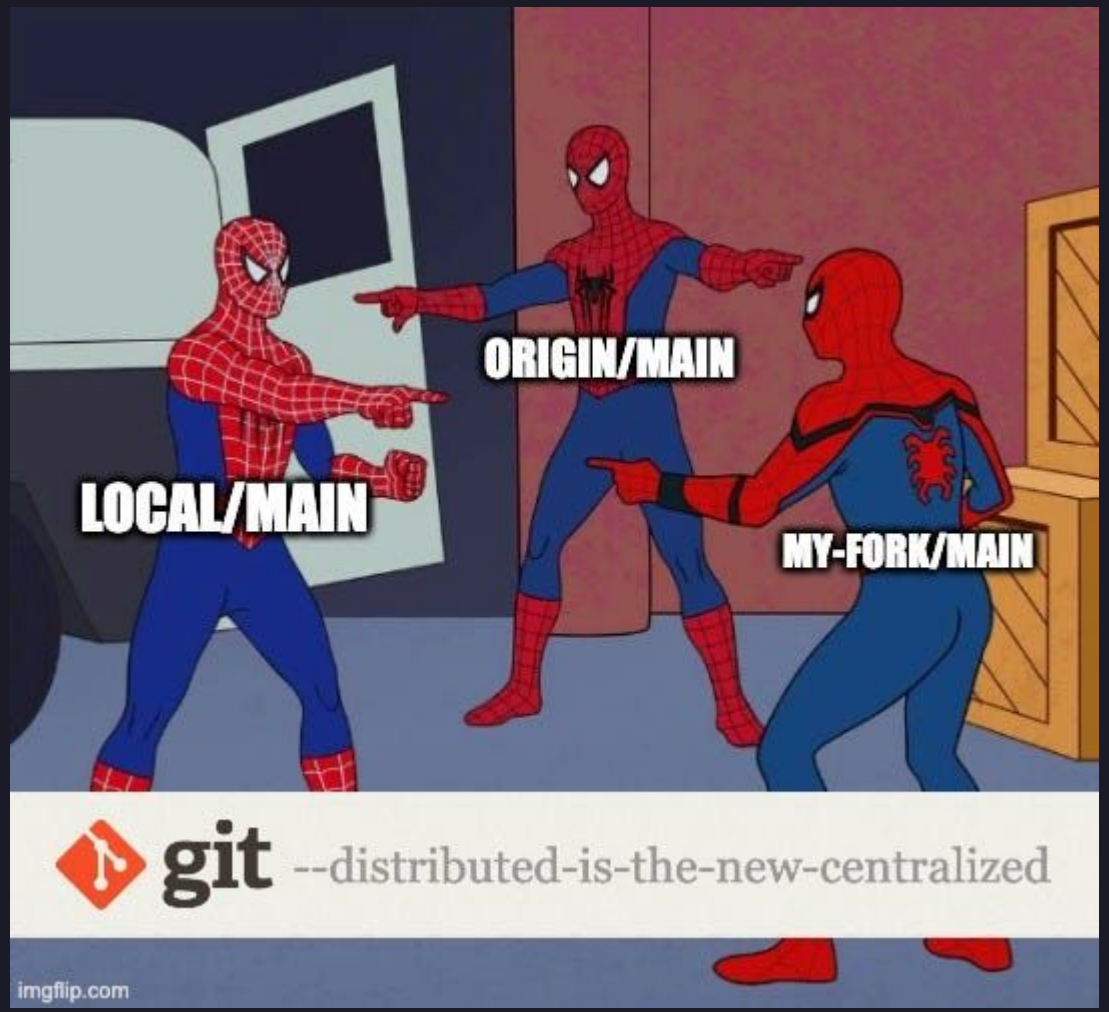
Why Are We Here?



code_fvm.py
code_fvm_updat
code_fvm_updat
code_fvm_updat
code_fvm_updat
code_fvm_updat

erfinal.py

Why Are We Here?



Why Are We Here?

- Research projects are long, messy, and collaborative
- Code underpins papers, figures, and theses
- Many of us learned coding “on the job”, not systematically
- Result: pain, confusion, and lost time

- Most of you can code – but not everyone uses version control yet
- Focus: collaboration, not just “learning Git commands”
- Goal: make your future-self and collaborators happier

What “Collaboration on Code” Really Means

- Multiple people changing the same project over months/years

- Mixing code, data, experiments, and manuscripts

- Keeping track of which version produced which

- Making it possible for others (and future-you) to reproduce work

The "Why?": From Chaos to Structure. Familiar Pain Points

The Old Way (Chaos)

Email attachments:

`model_v3_Juan.py,`

No chronological trace:

Which script produced Figure 4?

This version was before or after this change?

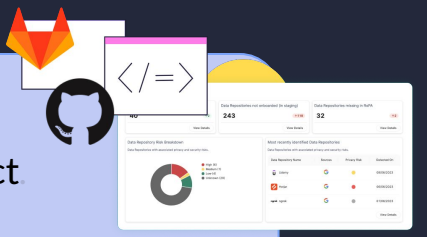
Multiple local versions:

`simulation_final_super_really_definitive.py,`

This ad-hoc system risks losing work, offers no clear history, and makes collaboration nearly impossible.

The New Way (Structure)

A single, versioned project.



Track every change, experiment safely with new ideas, and build a foundation for transparent, reproducible science.

A Quick Failure Story

Student A and B both modify the same script

No version control; code shared via email / USB / Teams

One overwrites the other’s changes “by accident”

After submission: can’t reconstruct the exact version for a reviewer

Outcome: panic, wasted days, weaker credibility

Without structure,
collaboration amplifies chaos.

A Quick Success Story

Shared repository for the project (code + minimal docs)

Each new feature on its own branch; changes reviewed before merging

Tagged release used for the paper and archived with a DOI

Months later: “re-run the experiments” → checkout tag + environment

With structure, collaboration
amplifies results.

What Usually Goes Wrong (Patterns)

- No single “source of truth” for the code
- Informal naming/versioning (v2, new_new, final_final)
- No record of why changes were made
- Everyone edits main/master (or the same file) directly
- No agreed rules: “What is okay to change?” “How do we review?”
- We rely on memory and goodwill instead of tools and workflows.

What we need

- A shared, reliable history of changes (version control)
- Separate spaces for experiments vs stable code (branches)
- Lightweight review steps to catch mistakes early
- Minimal documentation so others can actually run things
- Simple habits that scale as projects grow

Today’s focus

- Understand how Git supports collaborative research code
- Know how to work safely with others (branches, PRs, conflict resolution)
- Learn how to protect yourself from mistakes (“undo” without panic)
- See how docs, tests, and automation fit into the picture
- Apply it to one real project in your PhD, not just a toy example.

Outline

1. Why collaboration on research code fails
2. **Version Control System (VCS): why Git wins for research**
3. Git fundamentals (solo work)
4. Collaborative workflow on a real codebase
5. Code review, conflicts, and recovery
6. Making research code reproducible

From “Saving Files” to Version Control

Ad-hoc systems

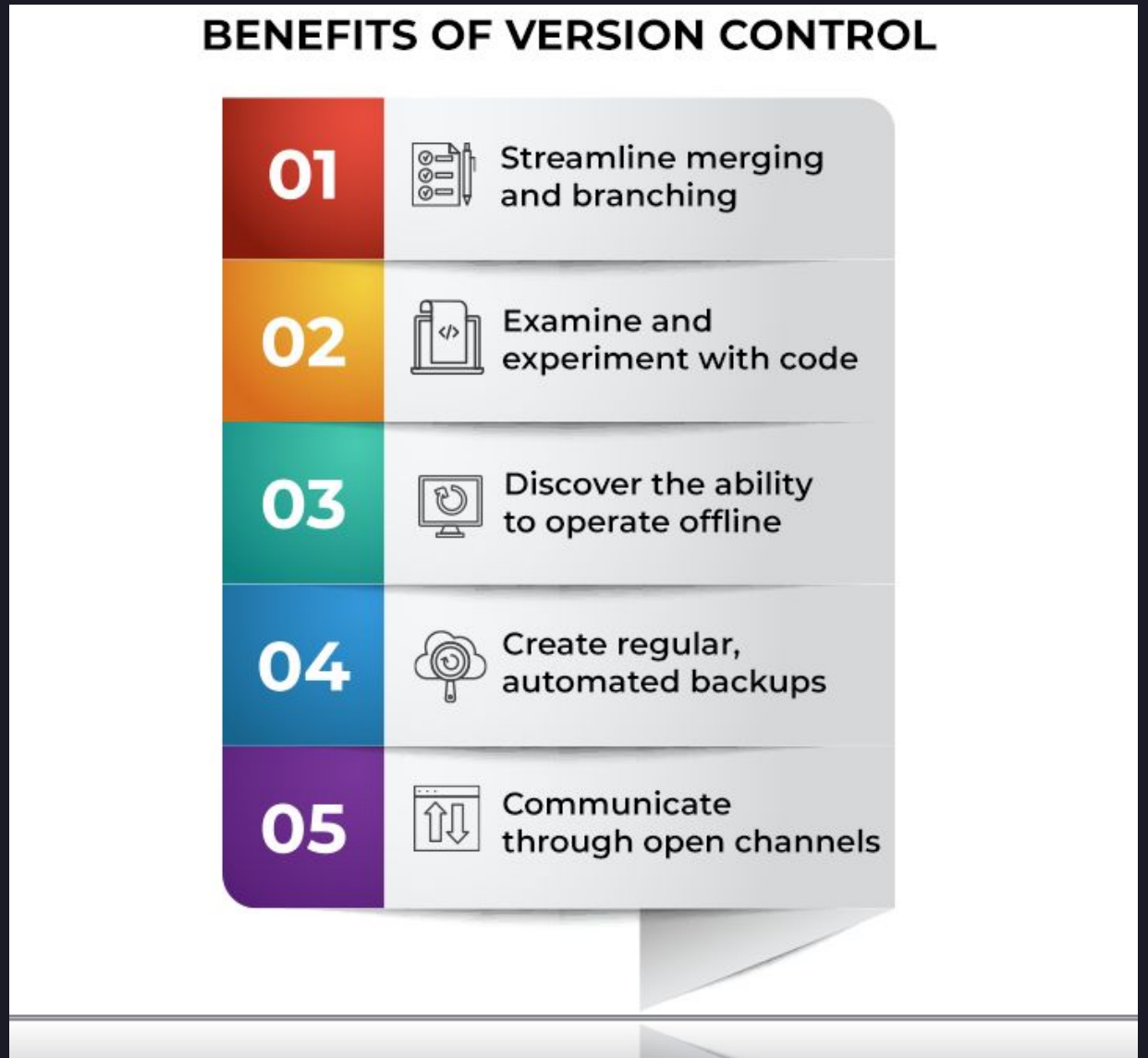
- Copying files, renaming with v2, final, definitive, dates…
- Works for 1 person, 1 week
- Collapses for teams & long projects

Version Control System (VCS)

- Structured history of changes
- Benefits:
  - Track who changed what, when, and why
  - Roll back or compare versions safely
  - Support multiple people modifying the same project

What Is a Version Control System?

- Records snapshots of your project over time
- Lets you move backwards and forwards in history
- Supports branching: parallel lines of development
- Helps merge divergent changes back together
- Provides a log: messages, authors, timestamps



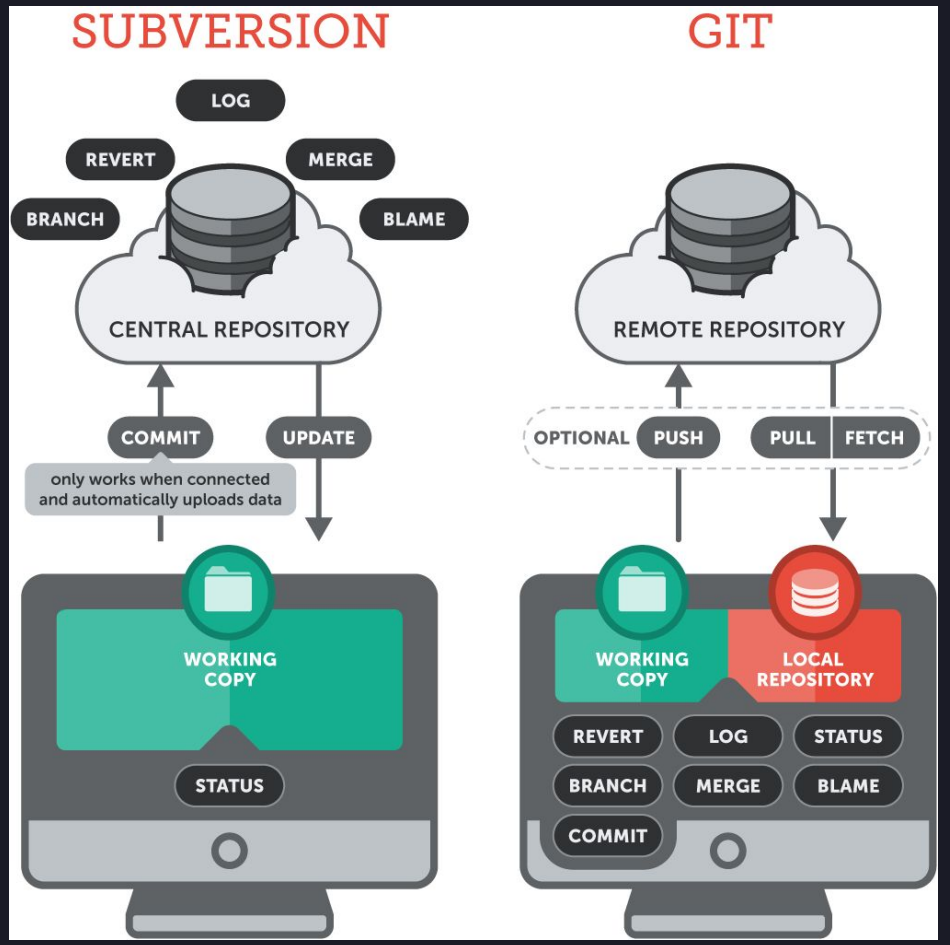
Centralized vs Distributed VCS

Centralized (SVN, Perforce):

- One central server holds the full history
- Clients checkout working copies; commits go to the server
- Offline: limited (can't commit without server)

Distributed (Git, Mercurial):

- Every clone has the full history
- Commits are local first; push to share
- Offline: you can commit, branch, inspect history



✓ **Distributed is particularly nice for research on laptops / clusters!**

Why Git Dominates Today. Git's the standard

- De facto standard for open-source and most research groups
- Powerful branching/merging model
- Integrates tightly with GitHub, GitLab, Bitbucket, etc.
- Large ecosystem: GUIs, CI, code review, integrations
- Still: concepts transfer to other systems if your lab uses SVN

Based on the 2022 Stack Overflow Survey, Git is used by over 96% of professional developers, making it the clear standard.

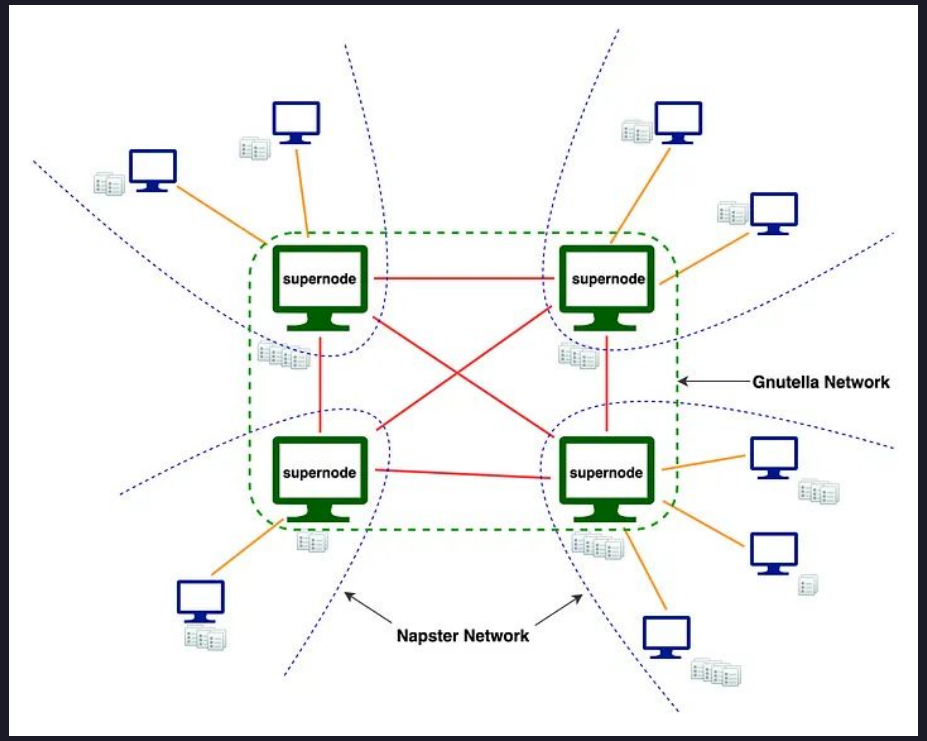
Outline

1. Why collaboration on research code fails
2. Version Control System (VCS): why Git wins for research
3. **Git fundamentals (solo work)**
4. Collaborative workflow on a real codebase
5. Code review, conflicts, and recovery
6. Making research code reproducible

What is Git? A Distributed System

- Most older systems (like SVN) are Centralized:

  A single server holds the project history.
- Git is Distributed (DVCS): Every developer's computer has a full copy of the entire project history.
- Key Benefits:
  - You can work completely offline.
  - Every clone is a full backup.
  - Branching is fast, local, and safe.



Source: migarapramodm.medium.com

The Core Workflow: A 3-Step Process Git's Mental Model in One Picture

1. Work (Working Directory)

You modify your files in your project folder, just as you normally would.

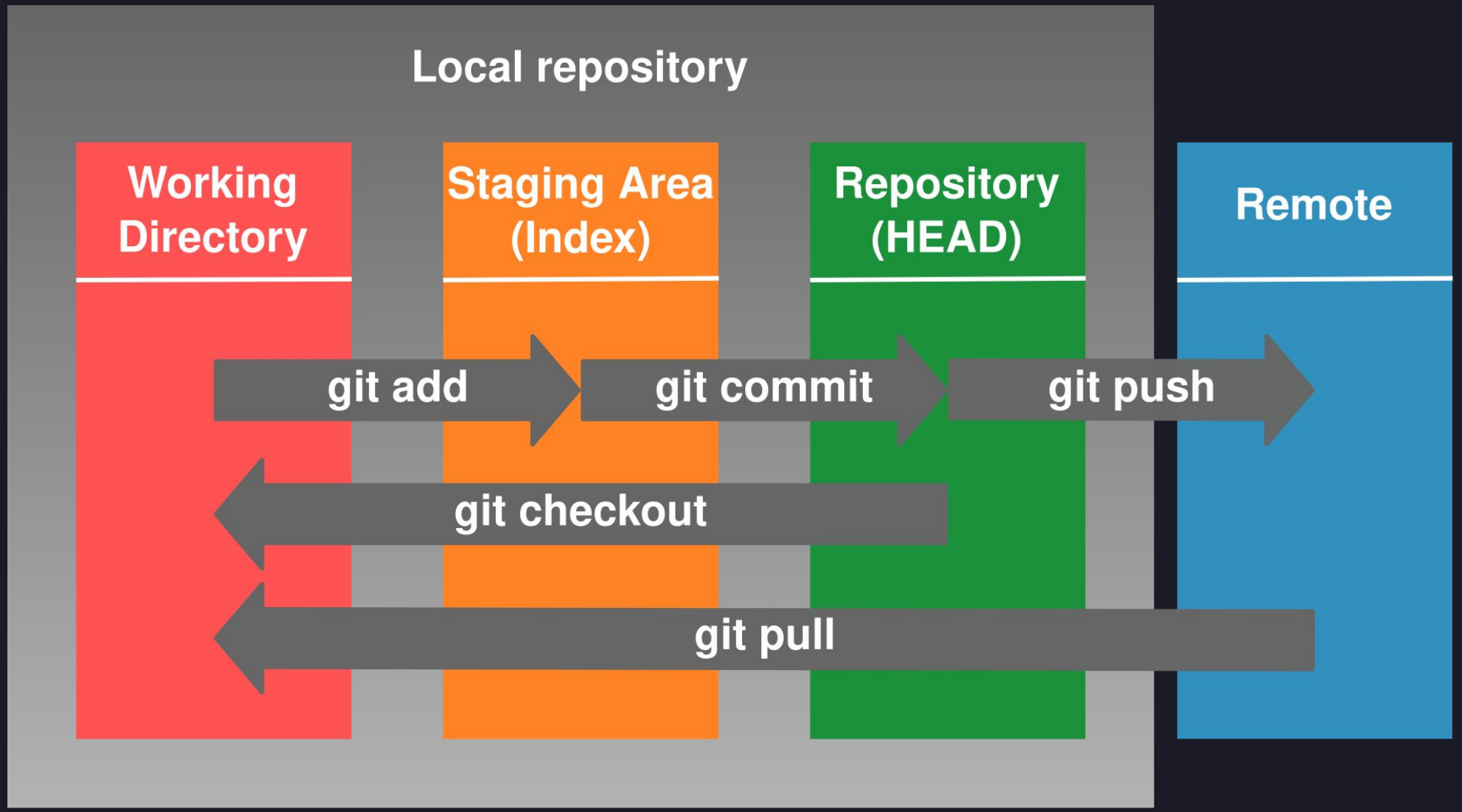
2. Stage (Staging Area)

Use git add to select *which* specific changes you want to save in your next snapshot.

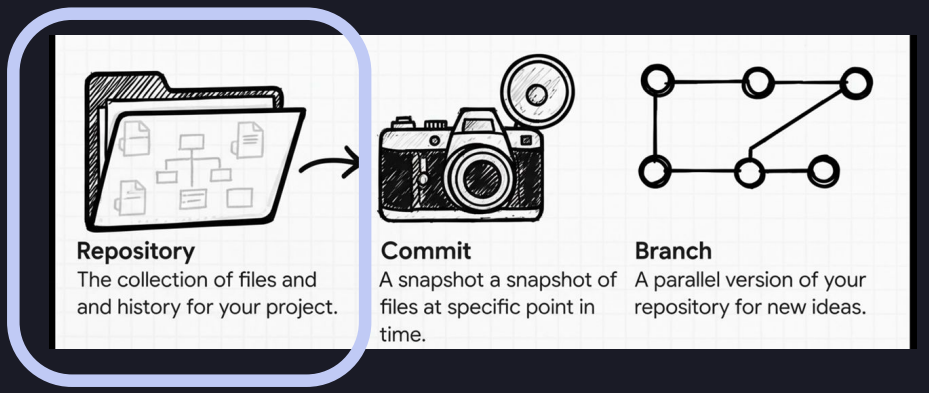
3. Commit (Repository)

Use git commit to take a permanent "snapshot" of the staged files and save it to your local history.

How Git works



Git basic concepts: repository



To work with Git, you need at least one repository

Where the entire history of changes to your project files is stored

It is a normal directory called .git by default

By default, it is invisible

In normal use, we do not manipulate this directory directly

We manipulate it indirectly using Git commands

To put a project under version control with Git, we will need to

- Create a repository. Create a directory (folder) for your project. This will be your working directory (working tree).
- Initialize an empty Git repository inside it. This will create the repository (.git directory) inside your working directory
- Work normally in your working directory, creating and modifying files and directories. And from time to time, save the status of your files and directories so that a history of changes is created

Git basic commands

`$ git init`

- Creates an empty repository in the directory you are in.
- If there are files in that directory, they will not yet be under version control.

`$ git status`

- Gives us information about the repository in the directory we are in.

`$ git add`

- Adds files to the stage.

`$ git commit`

- Creates a commit with the contents of the stage.

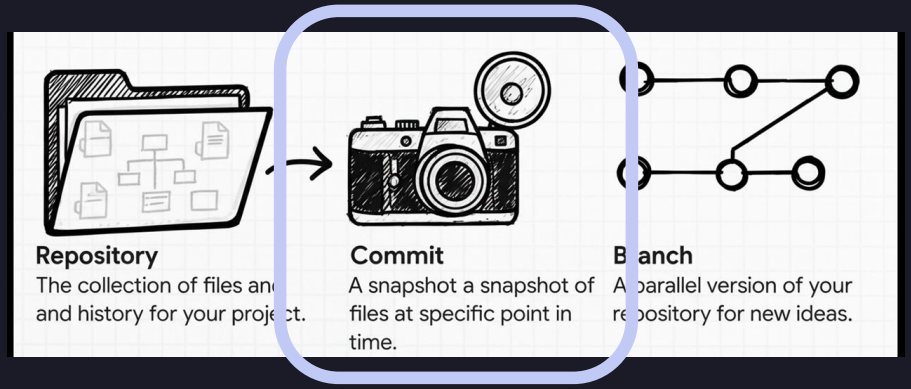
Git – Commits I

A commit is a “snapshot” of the current state of your files and directories, with some additional information (author, a commit message…)

We make a commit when we decide that we want to capture the state of our files and directories

We don't want to save every change in every file, we want to save sets of changes that have meaning

- For example, if we have modified two code files to correct an error, we may decide to make a commit
- In general, we want frequent commits: “commit early, commit often”



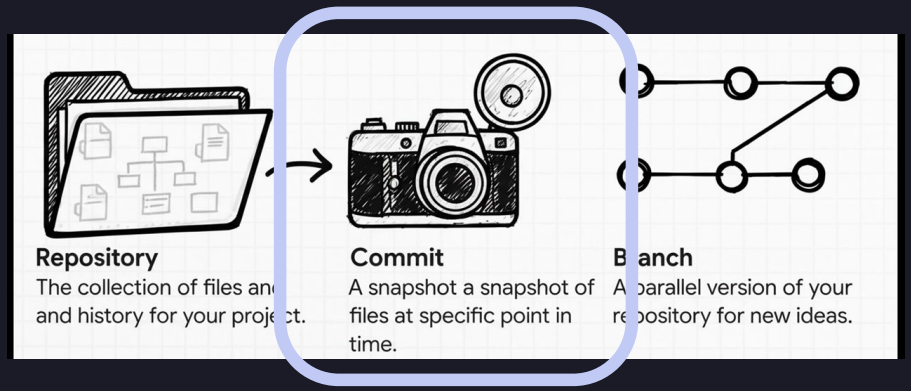
Git – Commits II

Commits are immutable

Each commit points to its predecessor (or predecessors when it is a merge-commit)

Each commit is assigned a unique identifier calculated with a hash function (SHA-1) based on its contents

- A SHA-1 hash is 20 bytes long and is usually represented as 40 hexadecimal digits
- In Git, we can generally use any prefix of 4 or more of those 40 digits as long as there is no ambiguity



Git – Stage

The commit process in Git has two phases

1. Decide which changes you want to put in the stage (preparation area)
2. Make the commit

Making a commit saves what is in the stage

- What is in the stage is a combination of what was in the previous commit and the changes we have added to that stage (`$git add`, `$git rm` ...)
- Staged, cached, and indexed are the same thing.
- But the three terms appear when using different Git commands.

The idea of the stage is to help you prepare more organized commits.

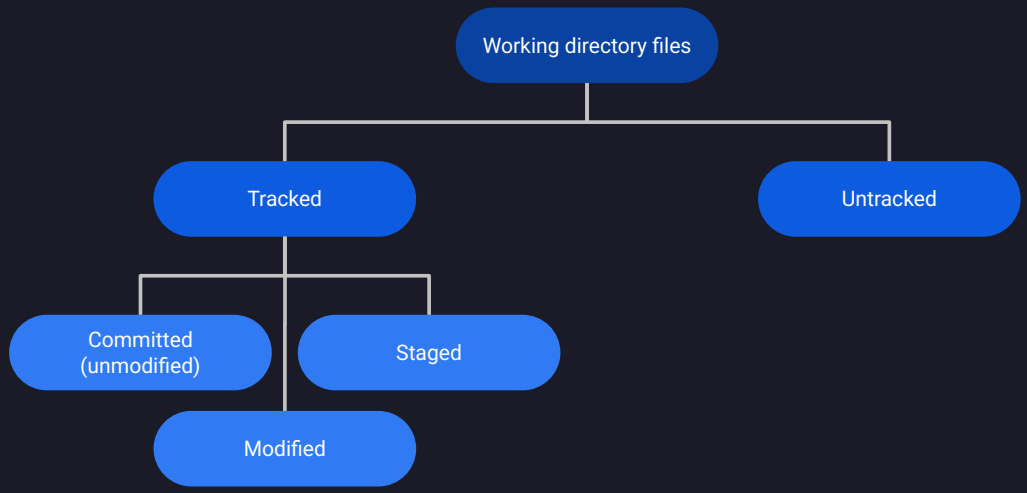
- But in practice, many people consider it a nuisance.

Git – File status

Files in the working directory can be in several states

- Tracked
  - Committed (unmodified). They were in the last commit and have not changed in the working directory
  - Modified. They were in the last commit and have changed in the working directory, but are not yet
  - Staged. Either they were in the last commit and have changed in the working directory, or they are new and they are already staged

- Untracked. They were not in the last commit, nor are they staged



Git – .gitignore

- There is another type of file, which are ignored files.
  - Git sees them, but acts as if they are not there.
- We indicate them in a file called .gitignore.
- This is necessary because there will often be files in the working directory that we do not want to share, such as local settings for our code editor, temporary files, files that contain secrets, etc

At https://github.com/github/gitignore there are templates for different types of projects.

Git – Viewing History

`$ git log`

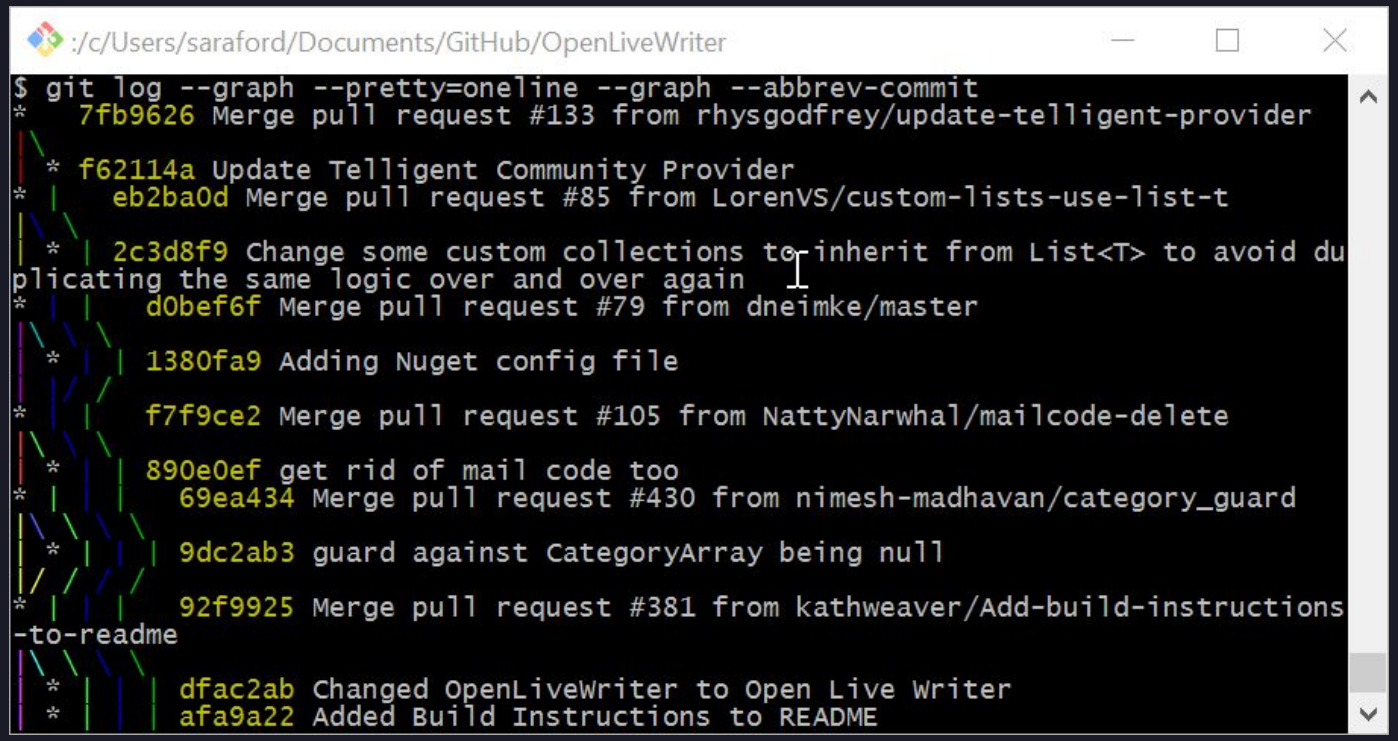
- Simple log

$ git log --oneline

- One-line condensed

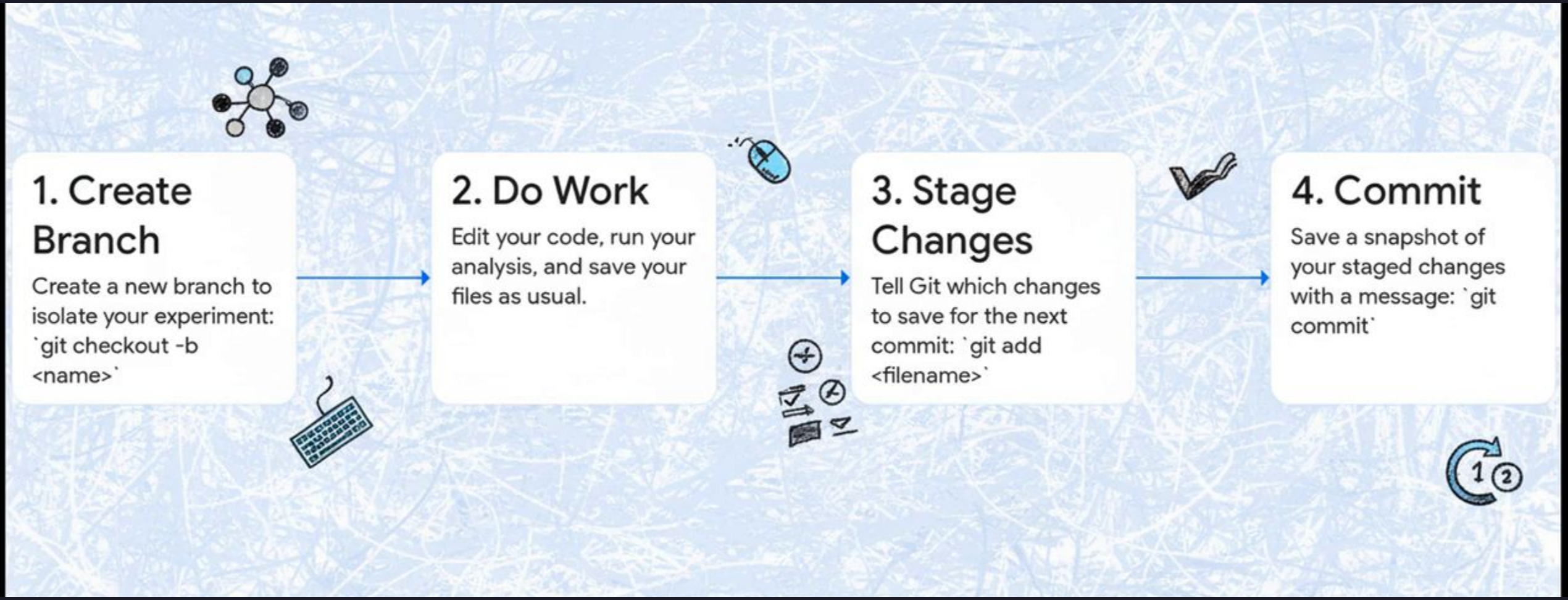
$ git log --oneline --graph --all

- Visual graph.



In VS Code / GUI: see commit history, diffs, blame

Solo researcher cycle



Outline

1. Why collaboration on research code fails
2. Version Control System (VCS): why Git wins for research
3. Git fundamentals (solo work)
4. **Collaborative workflow on a real codebase**
5. Code review, conflicts, and recovery
6. Making research code reproducible

Core Ideas of a Collaborative Workflow

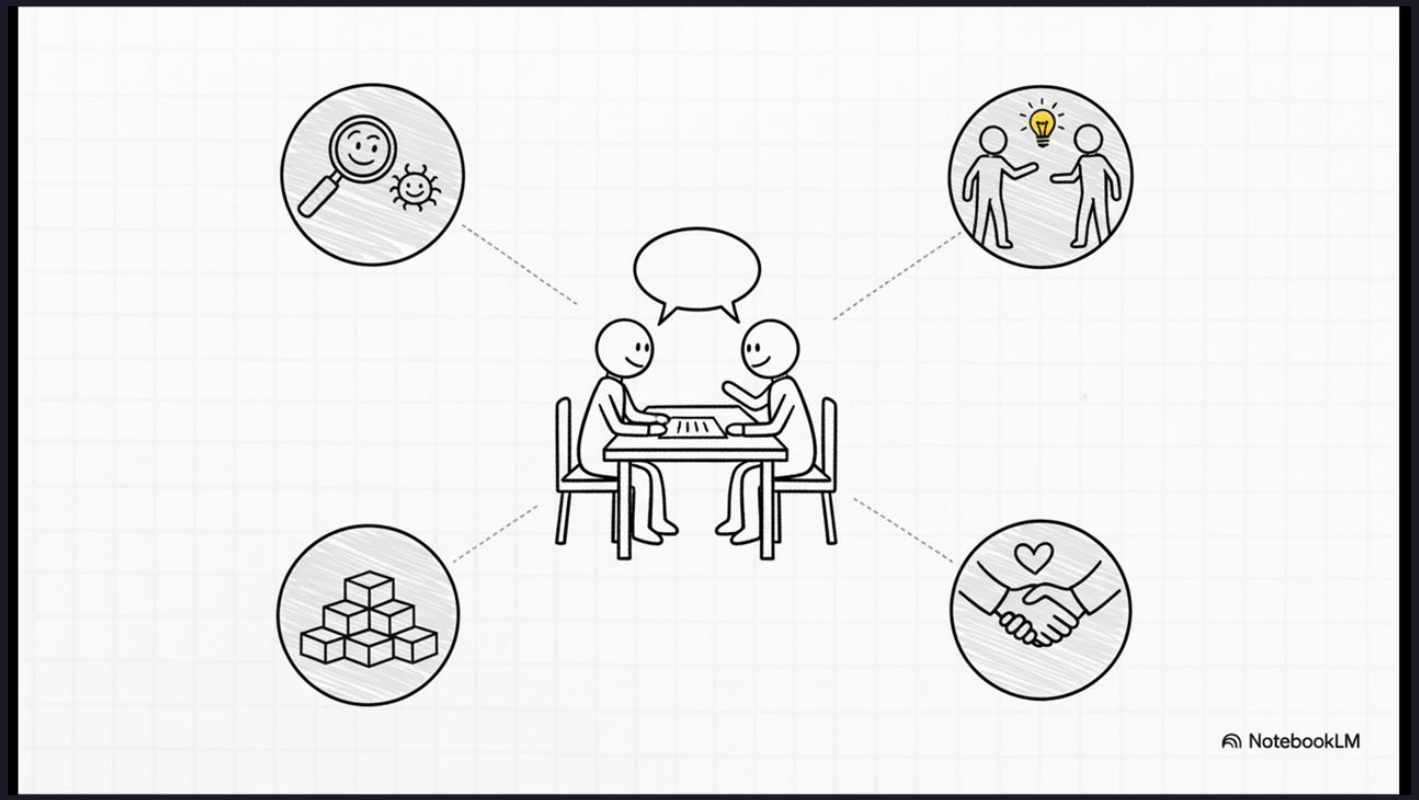
A shared remote repository
(GitHub, GitLab, Bitbucket, …)

A stable branch (main)
“Known-good” code

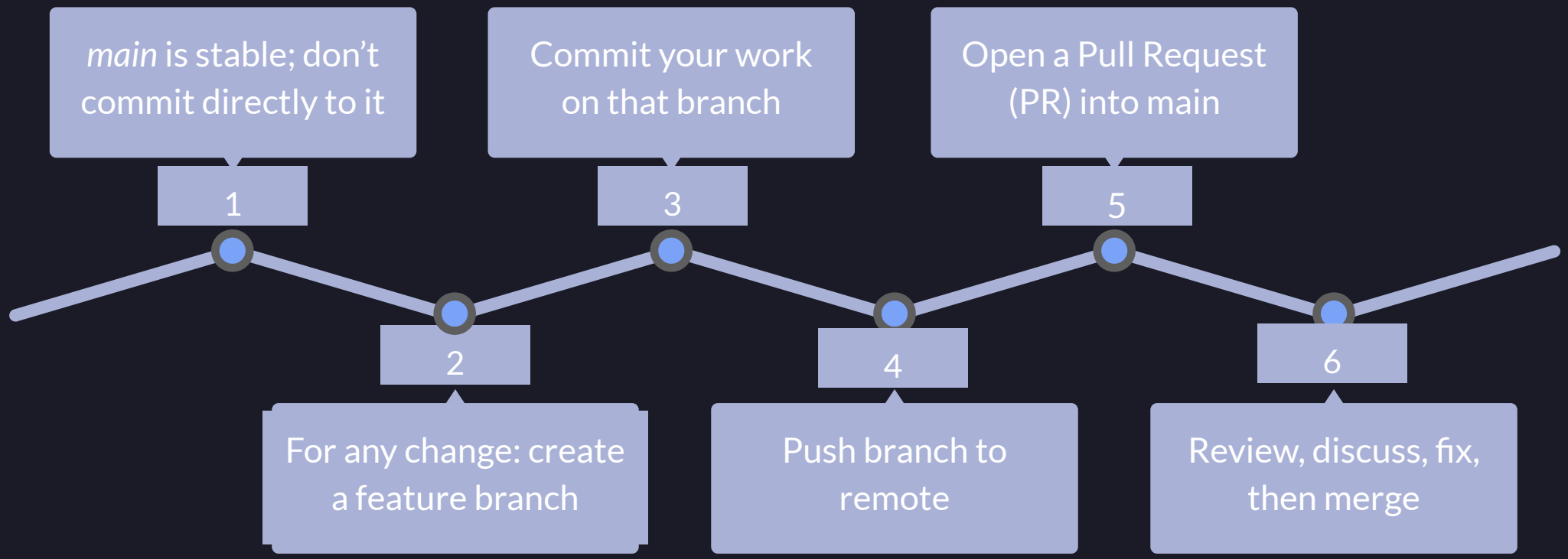
Feature branches
New work and experiments

Pull / Merge Requests
Discussion and review

Simple rules
Who can merge, how, and when



Recommended Simple Workflow



Same pattern works for 2 people or 10!

Branching: The "Killer Feature"

Source: jeffsearle.blogspot.com

Experiment Safely

A branch is a lightweight pointer to a commit. It's Git's "killer feature" for research.

- Want to try a new analysis? Create a branch: feature/new-analysis.
- If it works, merge it. If it fails, delete it.
- The main branch is always kept stable.

What is a branch?

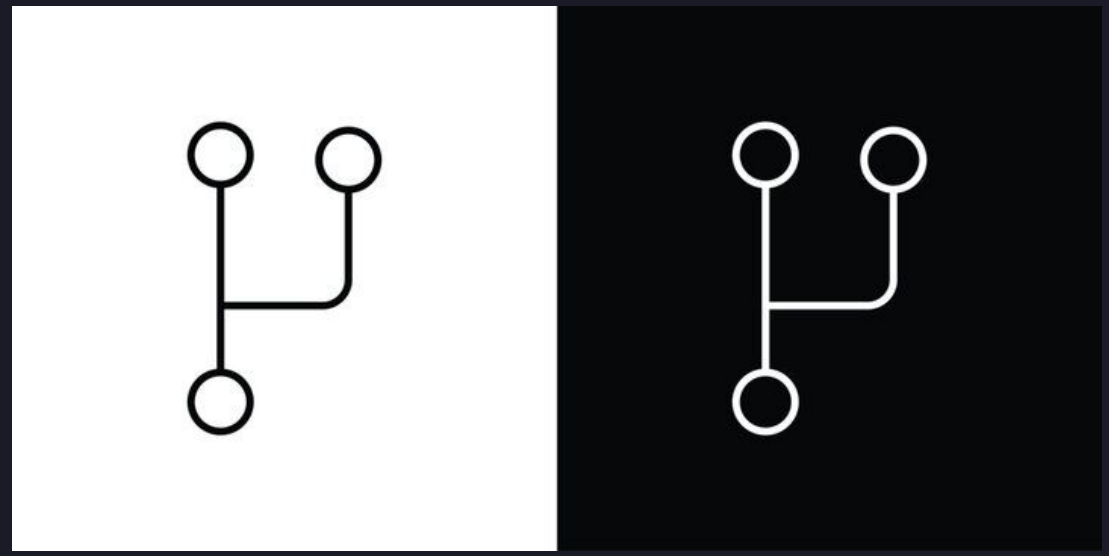
- A branch is a name that points to a commit
- When we create a repository (`$ git init`), a branch is created by default

  Traditionally called master, today the Git ecosystem is switching to other names

- The most common is main

  This branch has nothing special about it; it is simply the one that is created by default

- The name of the branch is not important; it is the name of the commit that is important



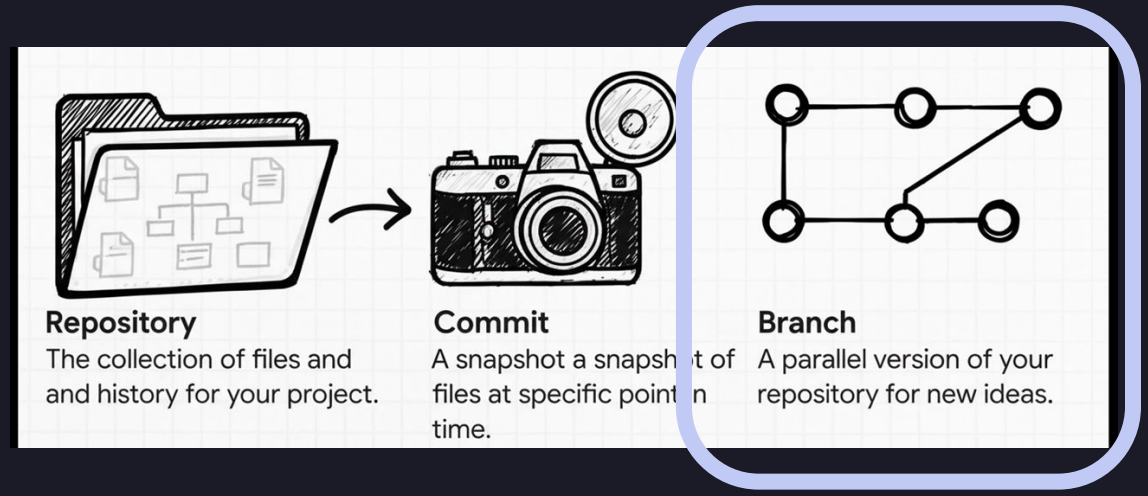
Creating a Branch and Pushing to Remote

Create and move to new branch:

```
$ git branch testing
$ git checkout testing
```

You are now on testing:

- Commits stay off main until merged
- Edit files as usual
- Stage and commit



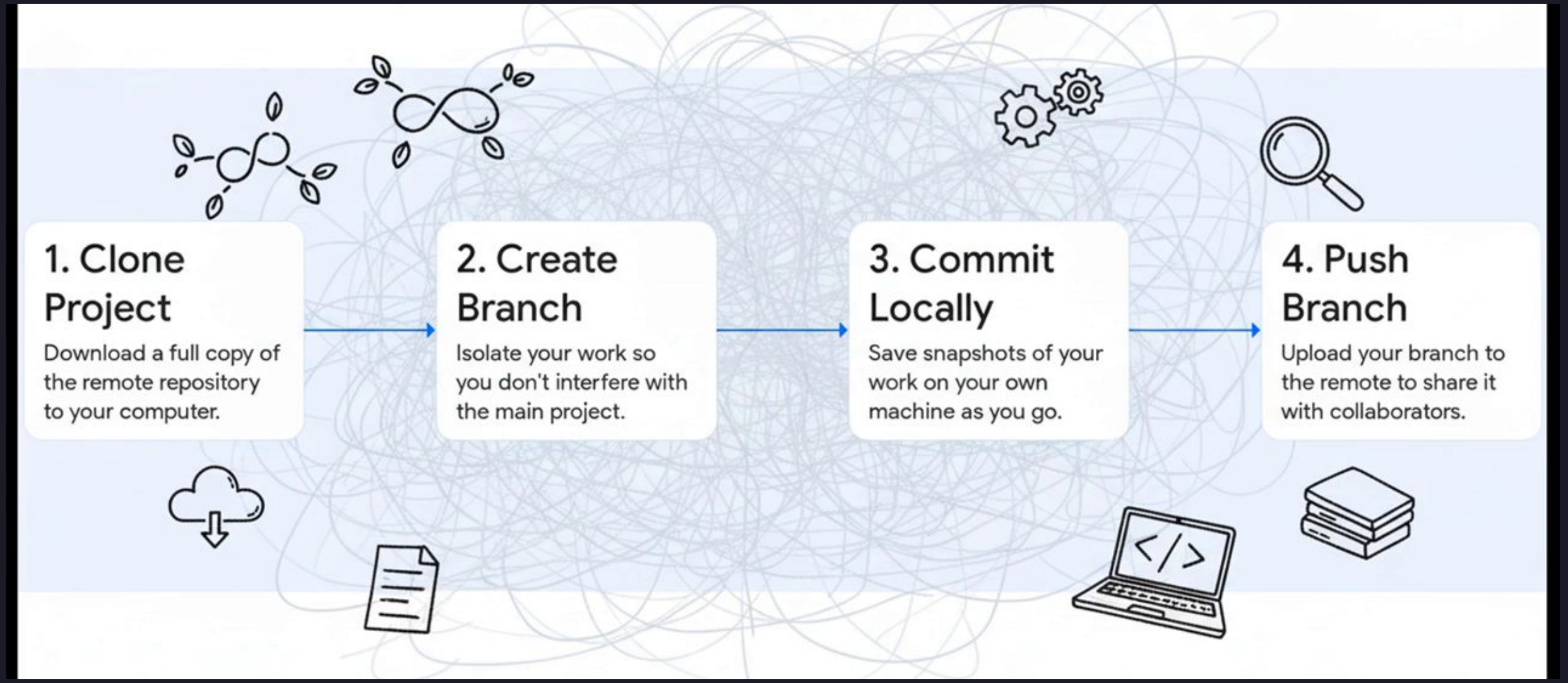
Push your branch to remote [-u (or --set-upstream) links local branch to remote branch]:

```
$ git push -u origin testing
```

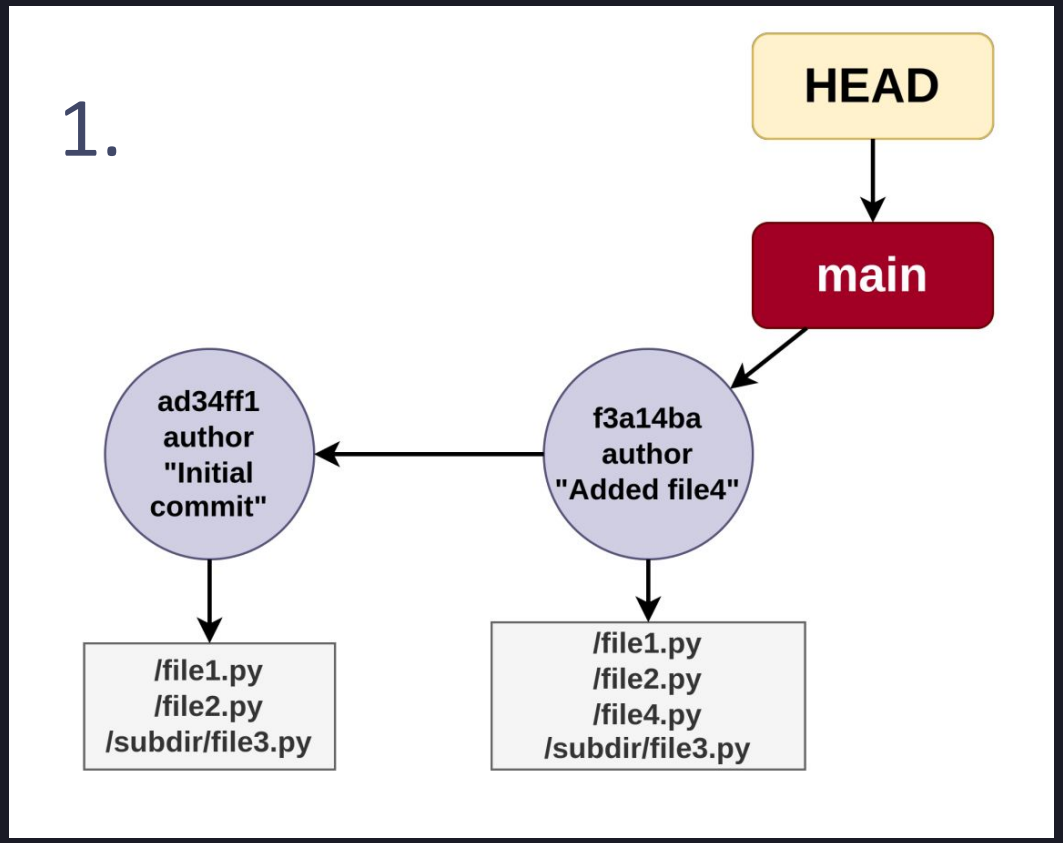
Collaborators can now:

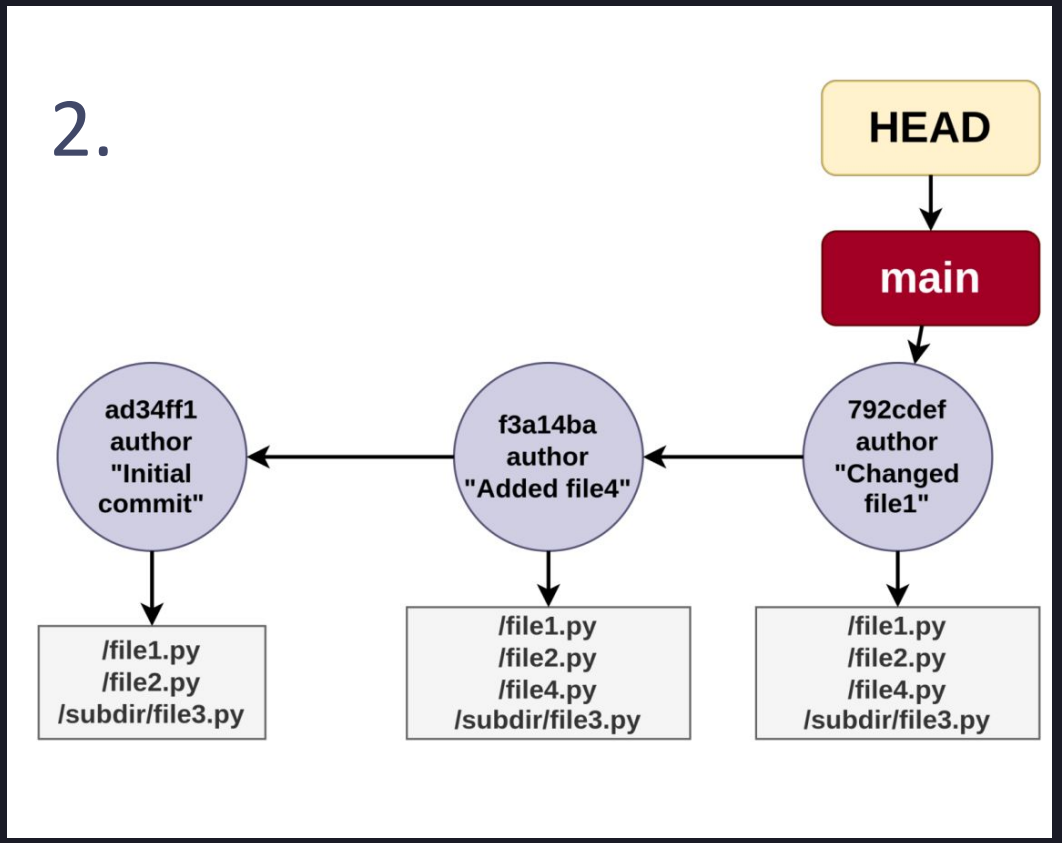
- Fetch your branch
- Review changes
- Run your code

Collaborative researcher cycle

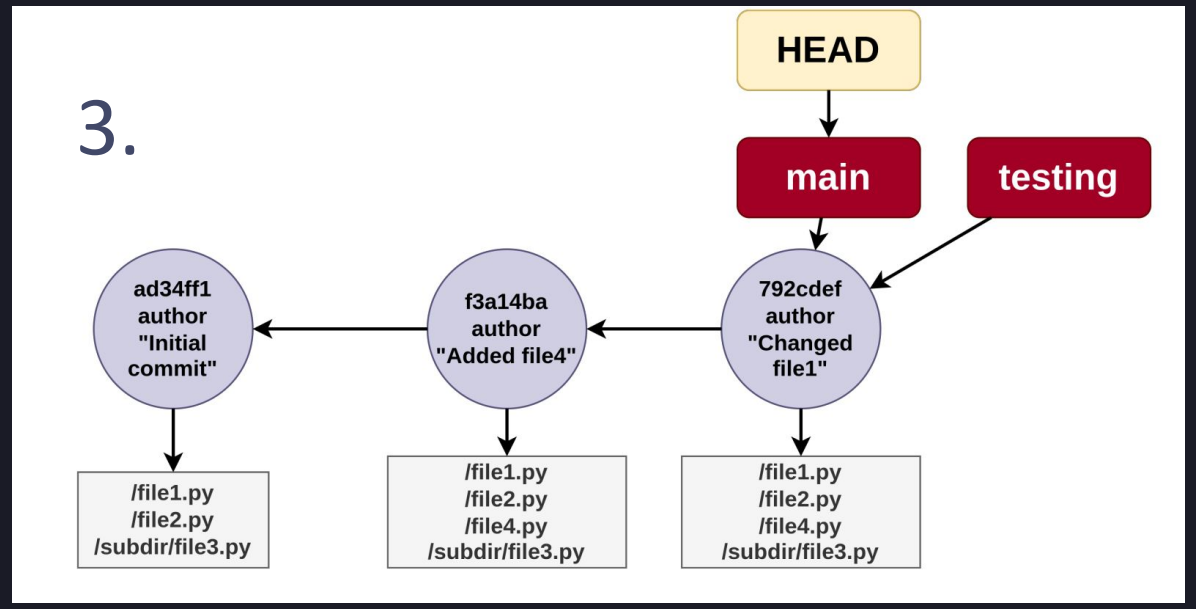


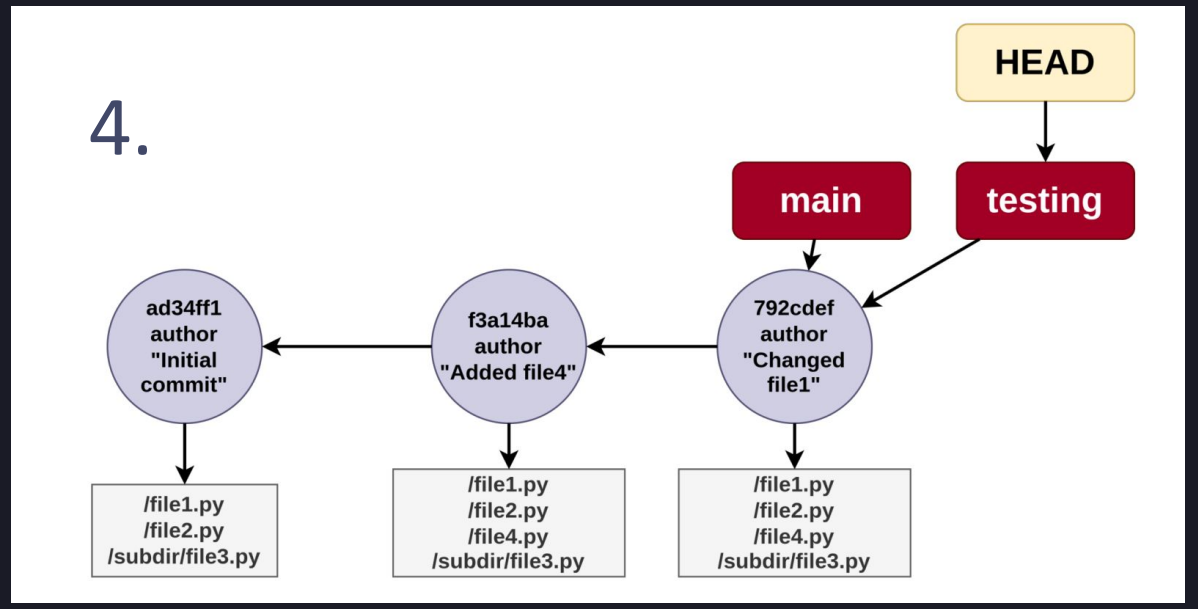
An example


1.
HEAD
main
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py


2.
HEAD
main
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
792cdef
author
"Changed
file1"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py

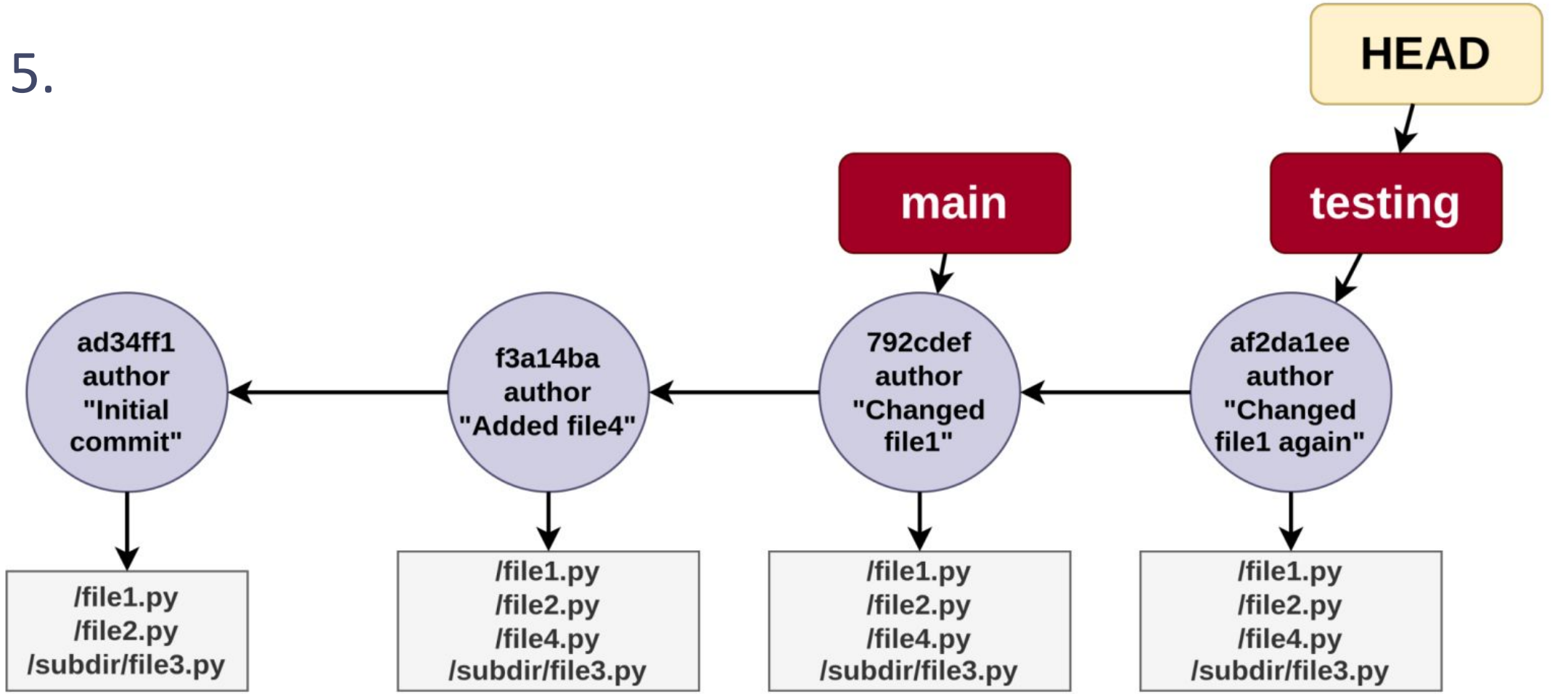
An example


3.
HEAD
main
testing
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
792cdef
author
"Changed
file1"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py


4.
HEAD
main
testing
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
792cdef
author
"Changed
file1"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py

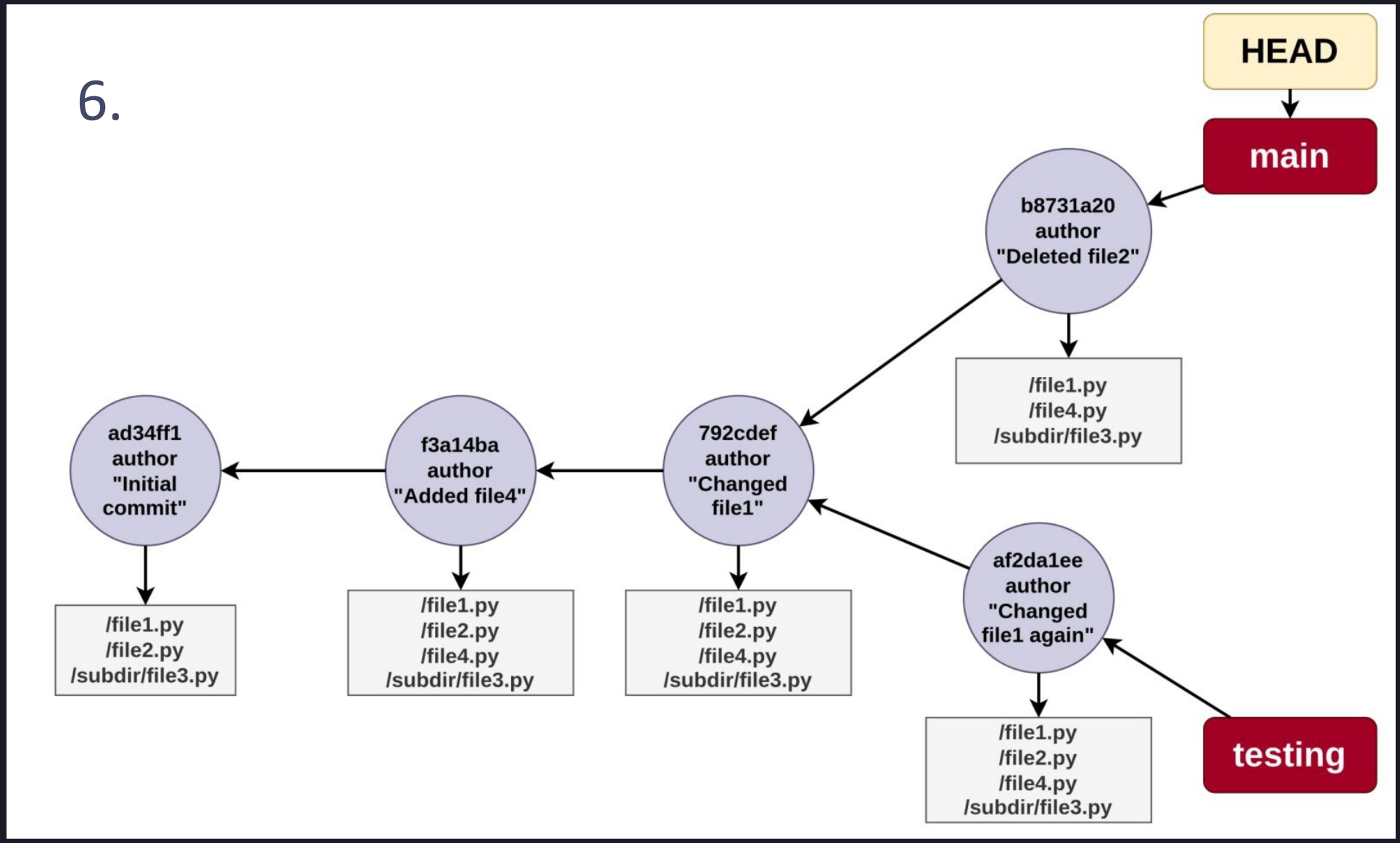
An example

5.


HEAD
testing
main
af2da1ee
author
"Changed
file1 again"
792cdef
author
"Changed
file1"
f3a14ba
author
"Added file4"
ad34ff1
author
"Initial
commit"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/subdir/file3.py

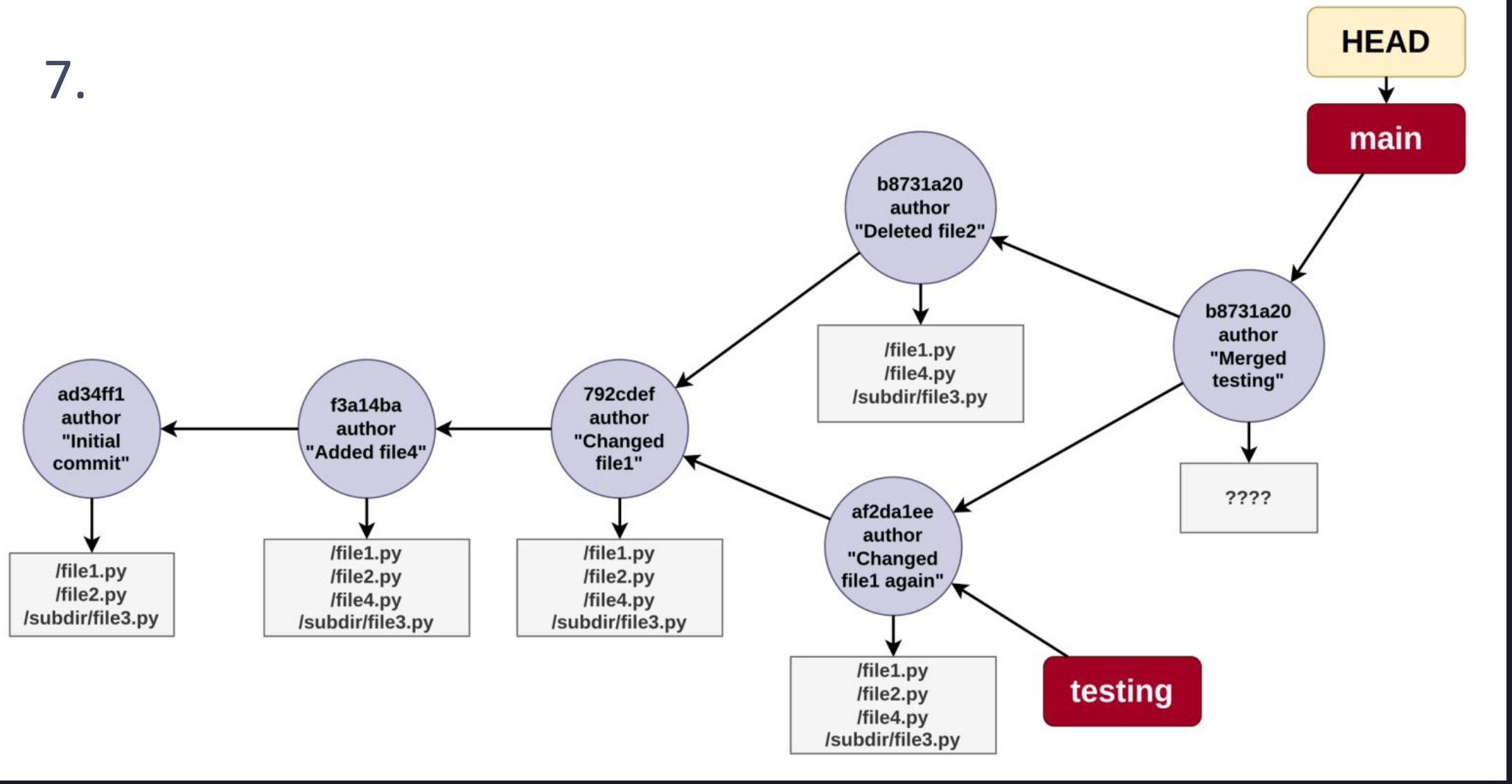
An example

6.


HEAD
main
b8731a20
author
"Deleted file2"
/file1.py
/file4.py
/subdir/file3.py
792cdef
author
"Changed
file1"
f3a14ba
author
"Added file4"
ad34ff1
author
"Initial
commit"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/subdir/file3.py
af2da1ee
author
"Changed
file1 again"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
testing

An example

7.


HEAD
main
b8731a20
author
"Deleted file2"
/file1.py
/file4.py
/subdir/file3.py
b8731a20
author
"Merged
testing"
????
ad34ff1
author
"Initial
commit"
/file1.py
/file2.py
/subdir/file3.py
f3a14ba
author
"Added file4"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
792cdef
author
"Changed
file1"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
af2da1ee
author
"Changed
file1 again"
/file1.py
/file2.py
/file4.py
/subdir/file3.py
testing

An example

1 → 2 : `$ git commit`

2 → 3: `$ git branch testing`

3 → 4: `$ git checkout testing`

4 → 5: `$ git commit`

5 → 6: `$ git checkout main + git add + git commit`

6 → 7: `$ git merge testing`

Pull Requests / Merge Requests

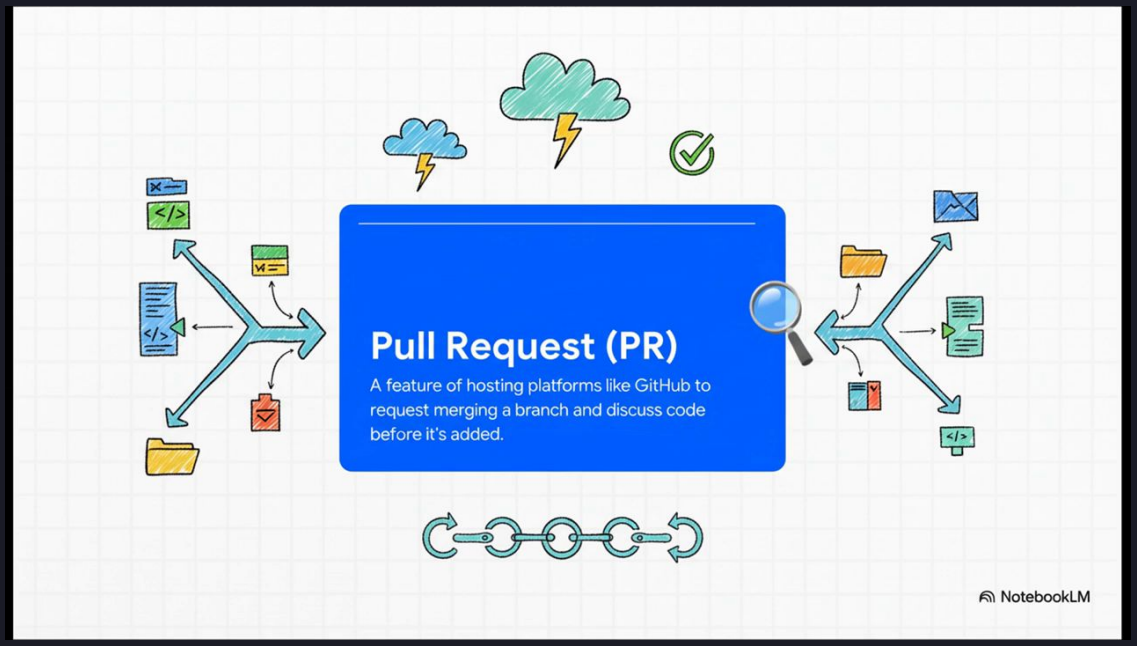
PR/MR = request to merge your branch into main

Includes:

1. List of commits and changed files
2. Description of the change
3. Discussion thread and review comments

Good practice:

1. One topic per PR (small, focused)
2. Clear title and summary



On GitHub: ‘Pull Request’. On GitLab: ‘Merge Request’.

Pull Request Workflow

1. Push branch to remote

2. Open PR: testing → main
   Fill in description (what / why / how to test)

3. Request reviewers

4. Address comments with follow-up commits

5. All checks green → merge into main

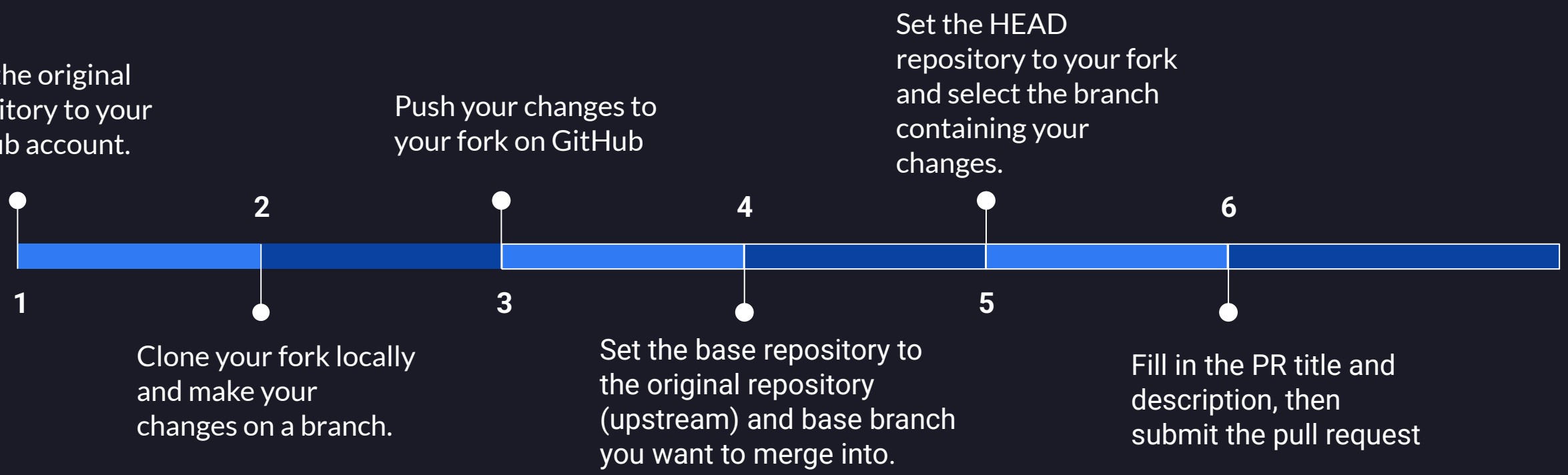
6. Delete branch (optional) after merge

Fork vs Branch

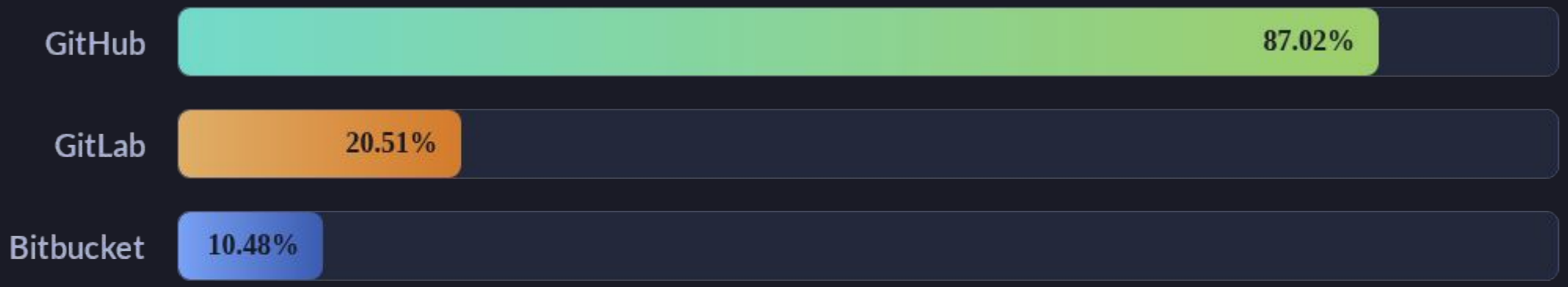
1	Branch	• Created inside the same repository • Same permissions / access as the repo • Typical for work within a team repo
2	Fork	• Your own copy of someone else's repository (on GitHub/GitLab) • You control it fully; original repo is separate • Typical for external contributions (open source, other groups)

👍 Inside a group project → branches. Contributing to others → fork.

Fork workflow



Collaboration: GitHub is The Hub



GitHub is the most popular platform for hosting Git repositories, enabling powerful team collaboration (2022 Stack Overflow Survey, Personal Use).

The GitHub Flow: Fork & Pull Request

1. Fork

Create a personal copy (fork) of the main project repository on GitHub.

2. Clone & Branch

Clone your fork to your computer and create a new branch for your changes.

3. Work & Push

Make your commits locally, then git push your new branch to *your fork*.

4. Pull Request (PR)

Open a "Pull Request" on GitHub, asking the main project to review and "pull" in your changes.

Outline

1. Why collaboration on research code fails
2. Version Control System (VCS): why Git wins for research
3. Git fundamentals (solo work)
4. Collaborative workflow on a real codebase
5. **Code review, conflicts, and recovery**
6. Making research code reproducible

Code Review is Peer Review

A Pull Request is a forum for discussion. A good review checks for:

- **Correctness:** Does the code actually do what it claims? Do the results make sense?
- **Clarity:** Is the code readable and understandable? Are variable and function names clear?
- **Reproducibility:** Does it include all dependencies and instructions needed to re-run the analysis?
- **Style:** Does it follow the project's agreed-upon coding standards for consistency?

Conflicts and “Undo”: Why Bother?

In collaborative work, branches will diverge. You will:

- Pull changes that conflict with yours
- Commit something you regret
- Fear of “breaking everything” makes people avoid Git

Goal:

- Resolve conflicts calmly
- Recover from mistakes without losing work

Merge vs Rebase: Big Picture

Merge

Creates a new “merge commit” that joins histories

Keeps full, true history (including branches)

Safer and simpler for beginners

Rebase

“Replays” your commits on top of another branch

Makes history look linear and cleaner

More powerful, but must be used carefully

Rule of thumb: merge for shared branches; rebase for your own feature branches.

Typical Scenarios for Merge vs Rebase

Merge

Feature branch behind main

Bring main into your branch:

```
$ git checkout feature/x
$ git fetch origin
$ git merge origin/main
```

Keeps a merge commit that records “feature/x was updated with main”.

Rebase

Cleaning up your own branch before PR (optional)

Replay your work on top of current main:

```
$ git checkout feature/x
$ git fetch origin
$ git rebase origin/main
```

Gives a cleaner, linear history for the PR (only safe on your own branch).

What Is a Merge Conflict?

Git auto-merges when changes don’t touch the same lines

Conflict = Git sees two incompatible edits to the same area

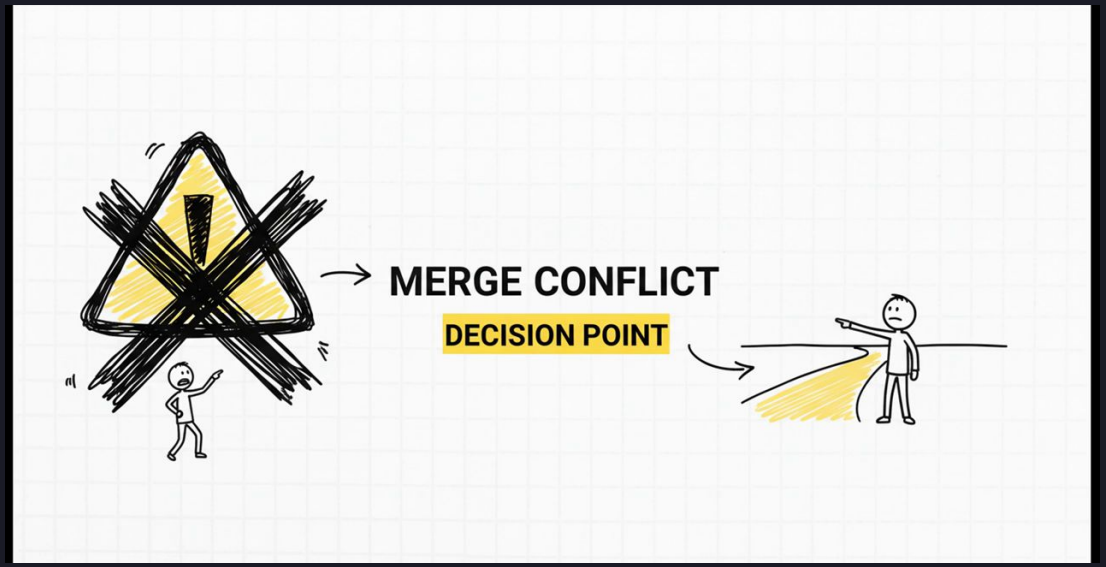
Typical message: “CONFLICT (content): Merge conflict in solver.py”

Files get conflict markers:

```
<<<<<<< HEAD

=======

>>>>>>> branch-name
```



This is normal; it just means you must decide which version (or combination) is correct!

Resolving a Conflict (Workflow)

1. Attempt merge/rebase → conflict message appears
2. Check status:

   `$ git status` (see “both modified” files)

3. Open conflicting file in editor / merge tool (e.g. VS Code)
4. Decide how to combine changes; edit to a valid final version
5. Mark as resolved:

   `$git add solver.py`

6. Finish merge/rebase:

   For merge: `$git commit` (if needed)

   For rebase: `$git rebase --continue`

If stuck, you can abort: `$git merge --abort` or `$git rebase --abort`.

History Safety: “Undo” Tools

- For local file changes:

  `$git restore file.py`

  (discard working changes)

- For staged changes:

  `$git restore --staged file.py`

- For “oops, wrong last commit message”:

  `$git commit --amend`

- For local branch resets:

  `$git reset --hard <commit>` (dangerous if shared)

- Safety net: `$git reflog` (records where your HEAD has been)

Other commands

Reflog = black box recorder. You can often get back to ‘lost’ commits.

- `$git revert`
- `$git cherry-pick`
- `$git reset`
- …

Resolving a Conflict

1. Git Pauses

Git stops the merge and adds conflict markers (`<<<`, `===`, `>>>`) to files.

2. Edit File

Open the conflicted file, choose the changes to keep, and delete the markers.

3. Stage Changes

Mark the conflict as resolved by staging the cleaned-up file: `git add`.

4. Commit

Finalize the merge by creating a new commit that resolves the conflict.

Outline

1. Why collaboration on research code fails
2. Version Control System (VCS): why Git wins for research
3. Git fundamentals (solo work)
4. Collaborative workflow on a real codebase
5. Code review, conflicts, and recovery
6. **Making research code reproducible**

Why Docs & Tests Matter in Research

Code underpins figures, results, and papers

Future-you and collaborators need to rerun and extend work

Reviewers and other groups may want to reproduce your results

Minimal docs + tests:

- Turn “works on my laptop” into reproducible research
- Increase trust and citability of your code

Good commits and PR messages

Commit messages:

- Imperative mood: “Add…”, “Fix…”, “Refactor…”
- Be specific: “Add unit test for inlet BC”, not “changes”

PR descriptions:

- What problem? What solution?
- Any limitations / side effects?
- How to run tests?

Example: “Add regression tests for shear layer instability case”

Minimal Documentation Stack

README.md

- What does this code do?
- How to install & run a basic example?

CONTRIBUTING.md

- How to propose changes (branches/PRs)?
- Coding style, tests to run before PR

CITATION.cff

- How should others cite this code?

Docstrings in key modules/functions

- Clearly state inputs, outputs, assumptions

Aim for clarity, not perfection!

Essential Practices for Reproducibility

Documentation (`README.md`)

The "front page" of your project. It is non-negotiable and **must** include:

- A clear project description.
- Installation instructions for all dependencies.
- A step-by-step guide to run the analysis and reproduce the final results.

Automated Testing (CI)

Tests are a safety net. Continuous Integration (CI) services (like GitHub Actions) automatically run your tests every time you push code.

This ensures that a new change doesn't accidentally break a part of the analysis that used to work.

Minimal Testing Strategy

Unit tests

Small functions (e.g. flux computation, CFL condition)

Regression tests

Check that known cases still give expected results
Golden files: reference outputs for a standard case

Reproducibility

Fix random seeds where relevant
Pin key dependencies (requirements.txt, environment.yml)

Use pytest (or similar) to run tests

pytest tests/

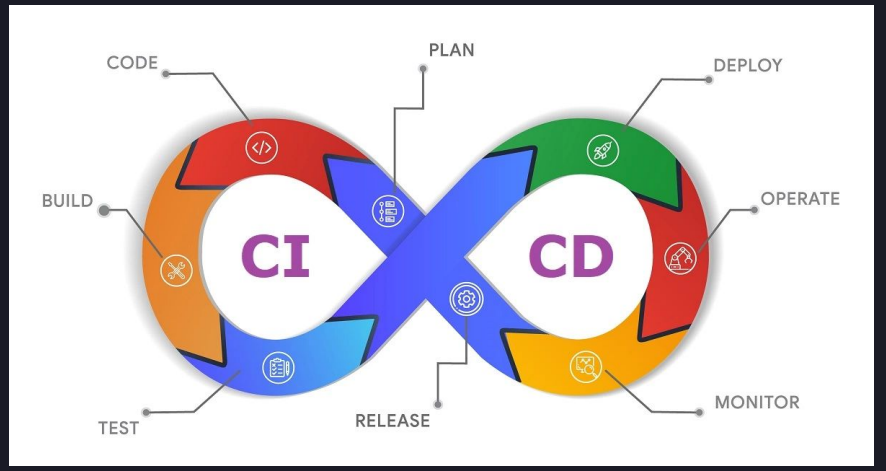
Automation: Let the Robot Help You

Pre-commit hooks

Run quick checks before each commit
Examples: formatting (black), linting (ruff/flake8), basic tests

Continuous Integration (CI) (e.g. GitHub Actions)

Run tests automatically on every PR / push
Prevent merging changes that break the code



Benefits

Catch errors early
Shared confidence that main is always “runnable”

Additional resources

Git tutorials and lessons

- Software Carpentry – Version Control with Git
  Step-by-step intro, great for self-study and refreshing basics.
- Pro Git (official book)
  Free online book; excellent reference once you know the basics.

Interactive sandboxes / visualizers

- Learn Git Branching
  Interactive puzzles and visual branch diagrams.
- NDP Software Git Cheatsheet
  Clickable map of common Git commands by “area” (workspace, index, history).

Branching models and workflows

- A successful Git branching model (nvie)
  Classic article on branch structure and release flow.

Atlassian Git guides

- Atlassian Git Tutorials
  Short, practical guides to Git basics and everyday workflows.
- Atlassian – Comparing Workflows
  Centralized vs feature-branch vs Gitflow vs forking workflows.
- Atlassian – Git Cheatsheet
  Printable cheat sheet with the most common commands.

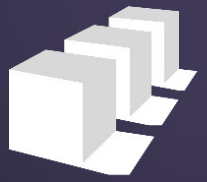
WORKSHOP 4 – RESCUER MSCA DOCTORAL NETWORK 2024-2028

Collaborating Effectively on Research Code

An Introduction to Git for Reproducible Science

Mario Morales Hernández (mmorales@unizar.es)

Universidad Zaragoza



Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Instituto Universitario de Investigación en Ingeniería de Aragón Universidad Zaragoza