Fundamentals of Git and GitHub

Rubén Béjar
rbejar@unizar.es

TWEED Project, SC1

Contents

Change Control

We often work with files that change and may have done some of these:

- Proposal_signed_signed_signed.pdf
- DeliverableD1_Final.pdf
- DeliverableD1_Final_Reviewed.pdf
- DeliverableD1_2024-03-01(*).pdf

(*) Use ISO 8601 for dates: https://www.w3.org/QA/Tips/iso-date.html

Change Control

Working alone, we would like:

- To recover previous versions of those files
- To explore different options to end up choosing one of them, or a mixture

And as a team:

- To know who made what changes, when and why
- To be able to work in parallel, distributing tasks and integrating the individual contributions easily

Real-time Collaborative Edition

Google Docs, Microsoft 365, Cryptpad, OnlyOffice and other online office suites offer real-time, collaborative edition of documents, slides and spreadsheets, and also:

- Change proposals
- Comments/Notes
- Recovering previous versions of the documents

Version Control Systems (VCS)

Software Engineering requires more control

Version Control Systems (VCS) are the tools that provide this

Git

The most popular VCS among software developers is Git

https://survey.stackoverflow.co/2022/#section-version-control-version-control-systems

Git

Git /gɪt/ is a free VCS initially created by Linus Torvalds for the development of the Linux core

Designed to control changes in files, especially in text files (such as the source code in any programming language)

Git

Powerful, flexible, widely used…

…but neither intuitive nor easy

Git Tools

Git is a CLI (Command Line Interface) Tool

There are applications with GUIs (Graphical User Interfaces) and most development environments and code editors offer some support

- But each of them hides/simplifies the Git commands in its own way

Git Tools

For an introduction to Git, it is worth to see how the command line is used

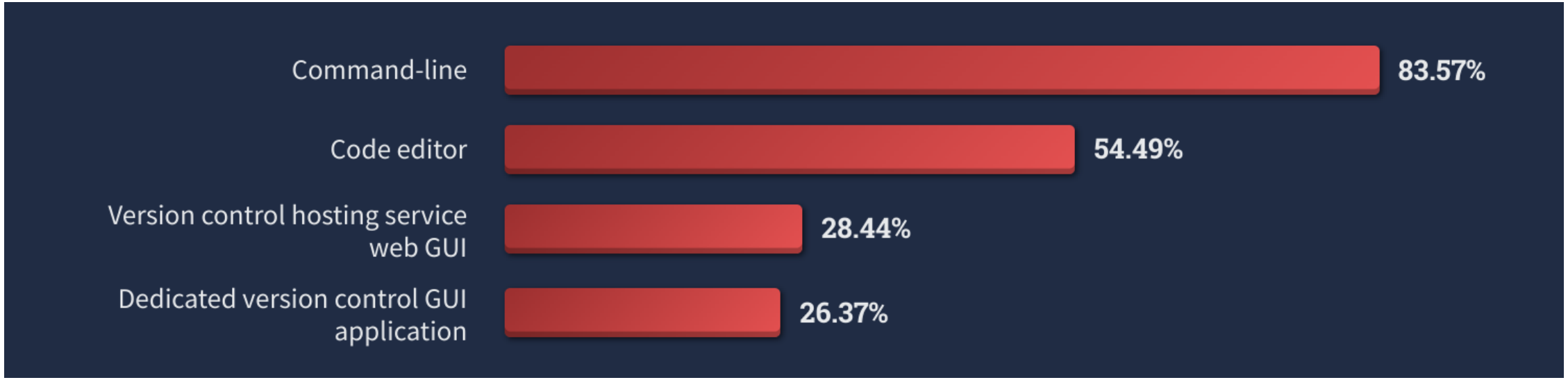
- It is the canonical version
- It has the most comprehensive documentation
- Understand that, and learning a GUI tool will be easy. The opposite is not true

A number of things are easier, some of them much easier, with a GUI

- Undoing changes, solving conflicts, working with repositories with long histories and many branches...

Git Tools

Among developers, the command line is the most used tool, followed by the ones provided by the code editor



https://survey.stackoverflow.co/2022/#section-version-control-interacting-with-version-control-systems

Git: Essential Concepts

Git Repositories

To work with Git you always need at least one *repository*

- The history of changes in your project will be stored there

It is a normal directory (folder)

- By default named `.git` and invisible

We do not manipulate the contents of that directory (only indirectly, using the Git commands)

- But we can backup, copy or move that directory as any other

Creating a repository

To start from zero, either you have a directory for your project or you create one

- That will be your *working directory* (or *working tree*)

Then you initialize an empty git repository there with `git init`

After that, you work normally with the files and folders in your working directory

- And from time to time you save the state of those files and folders, creating a history of changes

Basic Git commands

`git init`

- Creates an empty repository
- Any existing file or directory there will not yet be under control version

`git status`

- Get information about the current repository

`git add`

- It adds changes to files and folders to the *stage*

`git commit`

- It creates a *commit* with the contents of the stage

Commits

A commit is a snapshot of the state of your files and folders, with some additional information

- Author, commit message...

We make a commit when we decide to capture the state of our files and folders

- We don't want to commit every change, we want to create meaningful commits
  - For example: we have fixed a bug in our code and want to save that change
- We prefer small commits: "commit early, commit often"

Commits

Commits are immutable

- Once created you can't modify their contents

Commits point to their parent/s commit/s

- The initial commit does not have a parent
- *Merge commits* have more than one parent

Commits

Commits pointing to other commits form a graph structure (a directed acyclic graph, with a path from each commit to a unique root commit)

- We will see this later, when we see the concept of *branch*

Each commit has a unique identifier

- Calculated by applying a SHA-1 hash to its contents
  - A SHA-1 hash has 20 bytes, usually represented as 40 hexadecimal digits

The Stage

Commits in Git are done in two steps

1. Putting changes to files and folders in the *staging area* (also known as *stage*, *cache* and *index*) with `git add`
2. Committing those changes

A commit saves the changes in the stage

- The stage holds a combination of the previous commit and the changes we have added

The stage is there to help you to create more organized commits

- But many people consider it just a nuisance

State of files and folders in the working directory

Git tracks the state of files and directories in the working directory
We can change their state modifying them, and also with the `git add` and `git commit` commands

State	In the stage?	In the previous commit?	Changed in the working directory?
Tracked/Committed (Unmodified)	Yes	Yes	No
Tracked/Modified	No	Yes	Yes
Tracked/Staged	Yes	Yes	Yes (or it is new)
Untracked	No	No	-- (1)

(1) It does not make sense to ask if they have changed, because they did not exist in the previous commit. They can be new, or they may have been there for a while and we just didn't want to include them in our commits.

Ignored files

Git ignores the files we specify (using patterns) in the `.gitignore` file

This is important, because there are often files and directories which we do not want to share

- Local configuration, temporary files, files with secrets (e.g. passwords) etc.

There are templates for `.gitignore` files useful for different programming languages and projects in https://github.com/github/gitignore

History

After creating a number of commits (changes of files and folders) in a repository...

- Using `git add` + `git commit`

...we will have a history of changes

- `git log` allows us to examine that history
- `git checkout` allows us to go back to a previous commit in that history

This is useful on itself, but we will want to do more; for that, we need to understand what *branches* are

Git: Working with branches

Branches

Understanding how branches work is essential in Git

They are often useful when working alone, but a requirement to collaborate with others

Branches

A branch is a name that points to a commit

- Not always to the same commit, branches move
  - We can see them as "growing" instead of moving

When we create a new repository with `git init`, Git creates a default branch

- Originally named `master`, but these days the Git ecosystem is changing to `main`
- This branch is not special, it is just the one created by default

Branches

Commits are linked

- Each one points to its parents, forming a tree-like graph structure (a directed acyclic graph with a unique root and paths from each commit to that root)

As commits point to each other, and branches point to commits, branches give us access to, at least, a part of that tree-like structure

- That is why they are called branches

Branches

Branches might seem to be just a way to refer to commits which is easier for humans than their SHA-1 identifiers

But, as mentioned beofre, they do not always refer to the same commit(*): they allow us to name a sub-graph within the repository which will be growing when we commit new changes

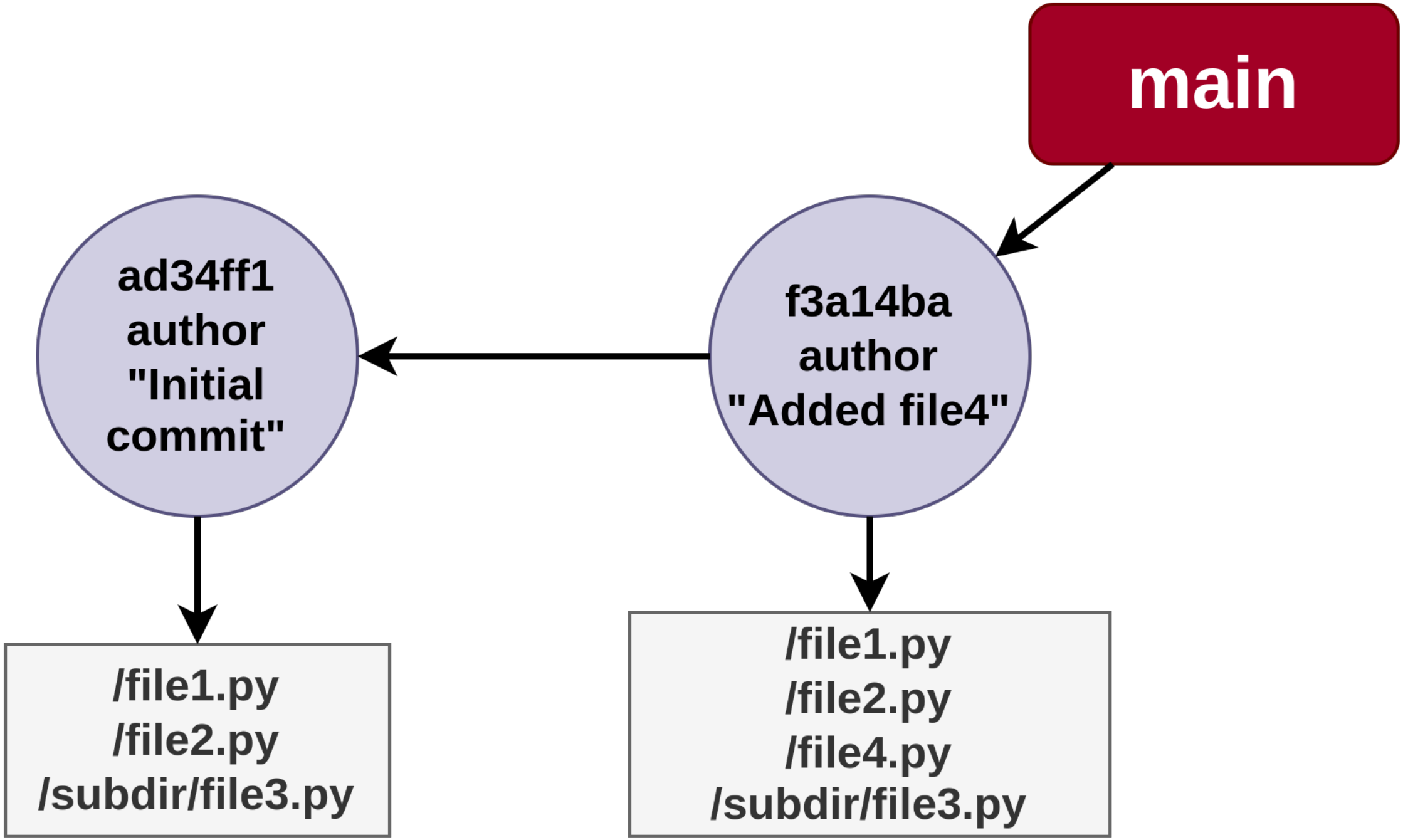
- When we make a commit, the current branch moves automatically to it
  - The branch "grows": now it has one more commit than before

Commits which are unreachable from at least one branch are eventually deleted by Git

(*) We will use *tags* if we just want to give a name to a certain commit

An example

- We create a directory and inside it we create `file1.py`, `file2.py` and `subdir/file3.py`
- `git init`: we create an empty git repository in our directory
- `git add .`: we add all the files to the stage
- `git commit -m "Initial commit"`: we create our first commit
- We create a `file4.py`, `git add ./file4.py` and `git commit -m "Added file4"` (our second commit)

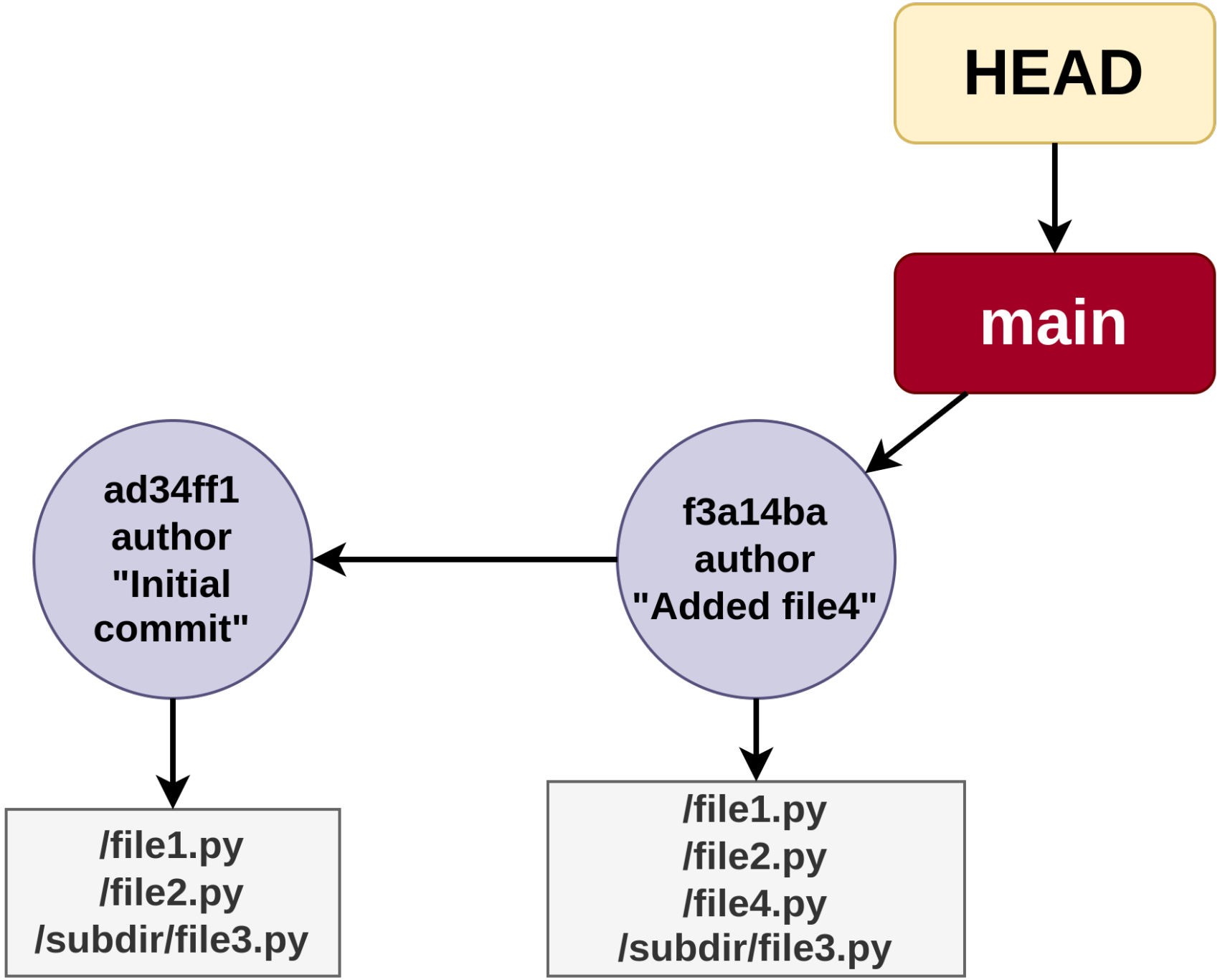

main
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py

Where are we within the graph of commits of a repository?

In Git we are, by default and most often, in a branch which points to a commit

- "Being in a branch" means that new commits will be added to that branch
- "Being in a commit" means that our working directory reflects the state of our files and directories when we did that commit
    - Plus the changes we may have made since then
- The "previous commit" is the one pointed out by our current branch

Git knows which is our current branch because it has a pointer named *HEAD* pointing to it

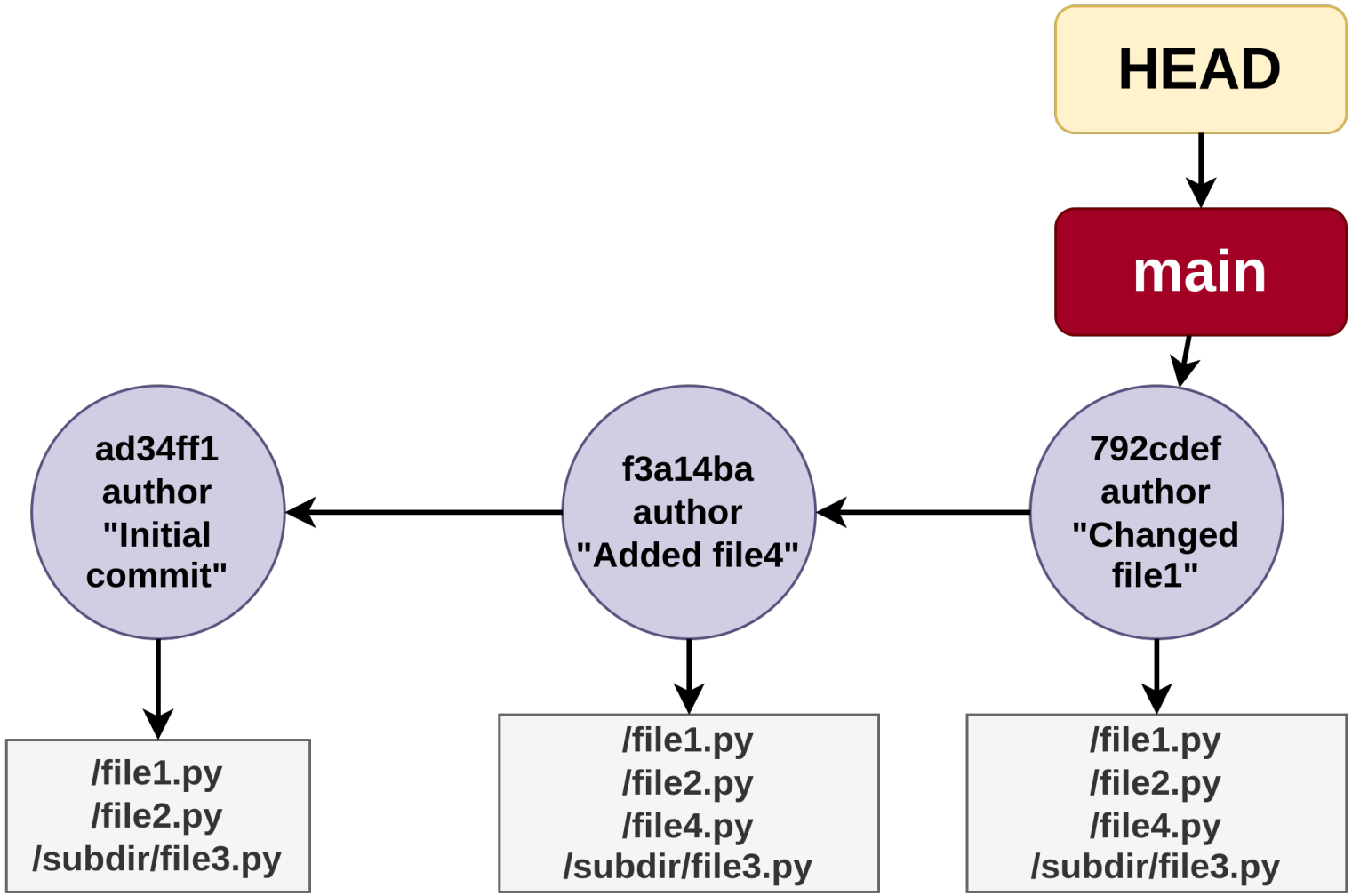

HEAD
main
ad34ff1
author
"Initial
commit"
f3a14ba
author
"Added file4"
/file1.py
/file2.py
/subdir/file3.py
/file1.py
/file2.py
/file4.py
/subdir/file3.py

Branches

When we make a new commit, the current branch (pointed by HEAD) moves to that new commit

- The branch has grown

```
$ git commit -a -m "Changed file1"
# The -a tells git to automatically add modified or deleted files (so we can skip the `git add`)
```

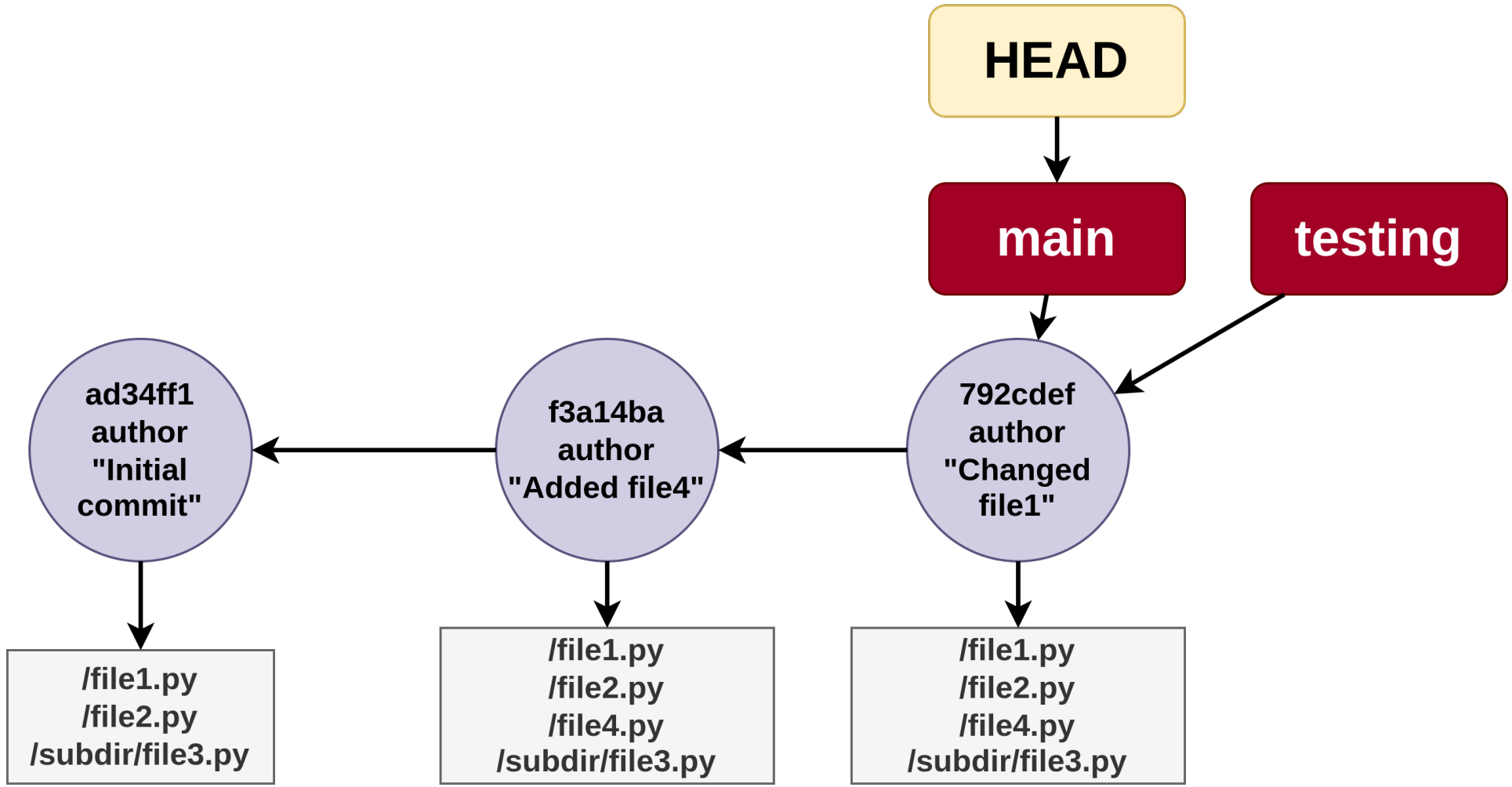


Branches

The `git branch` command creates a new branch pointing to the commit where you are

- By default does not move you to that new branch: the HEAD pointer does not change

```
$ git branch testing
```



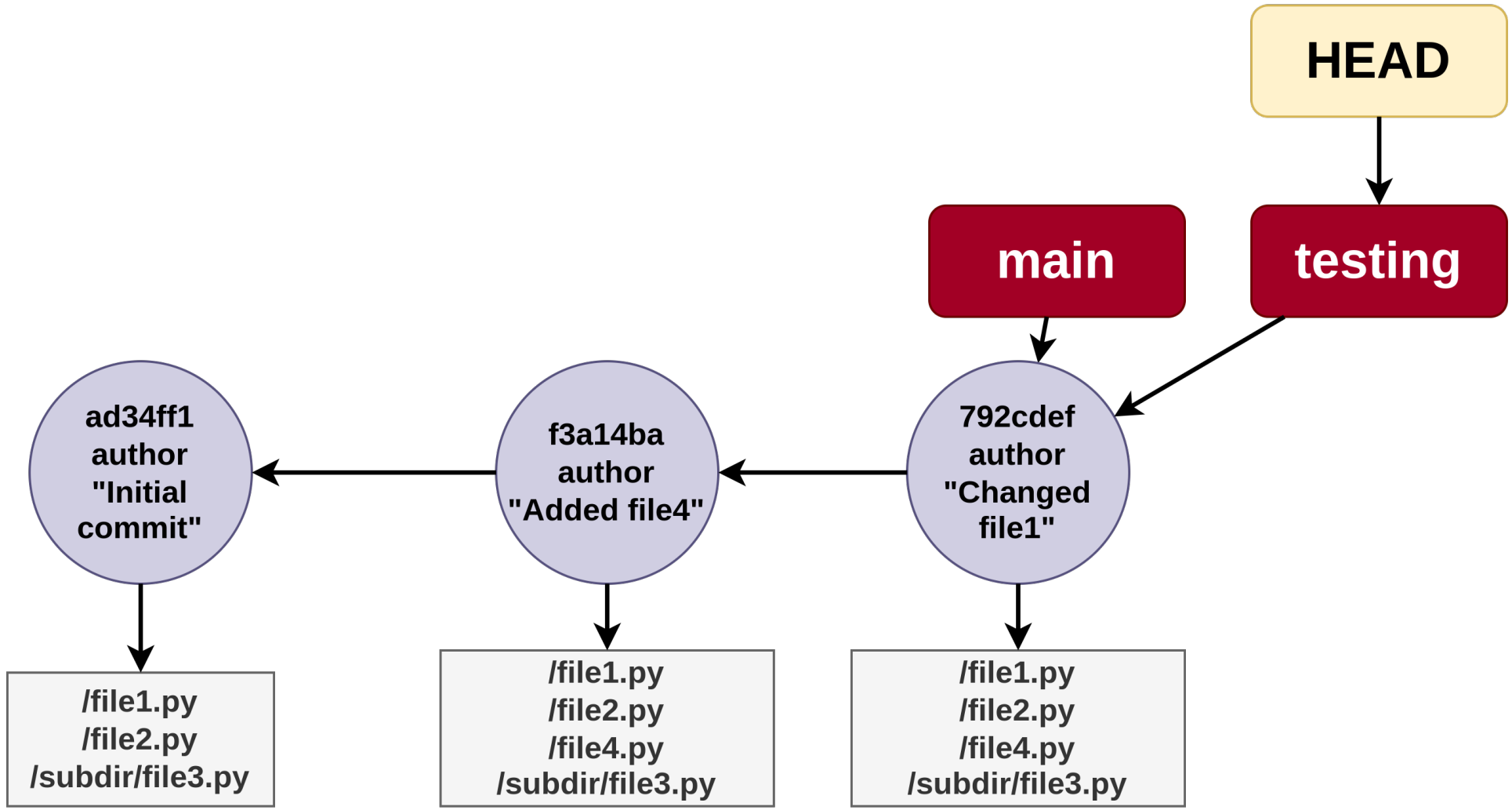
Branches

The `git checkout` command moves us to a different branch

- Moving the HEAD pointer
- And **modifying the working directory** so its files and folders are exactly how they were in the commit pointed by that different branch

Recent versions of Git have introduced the `git switch` command to change branches, because `git checkout` does several things. However, even in recent versions (checked this for 2.49.0) it is still experimental: its behavior might change.

```
$ git checkout testing
```

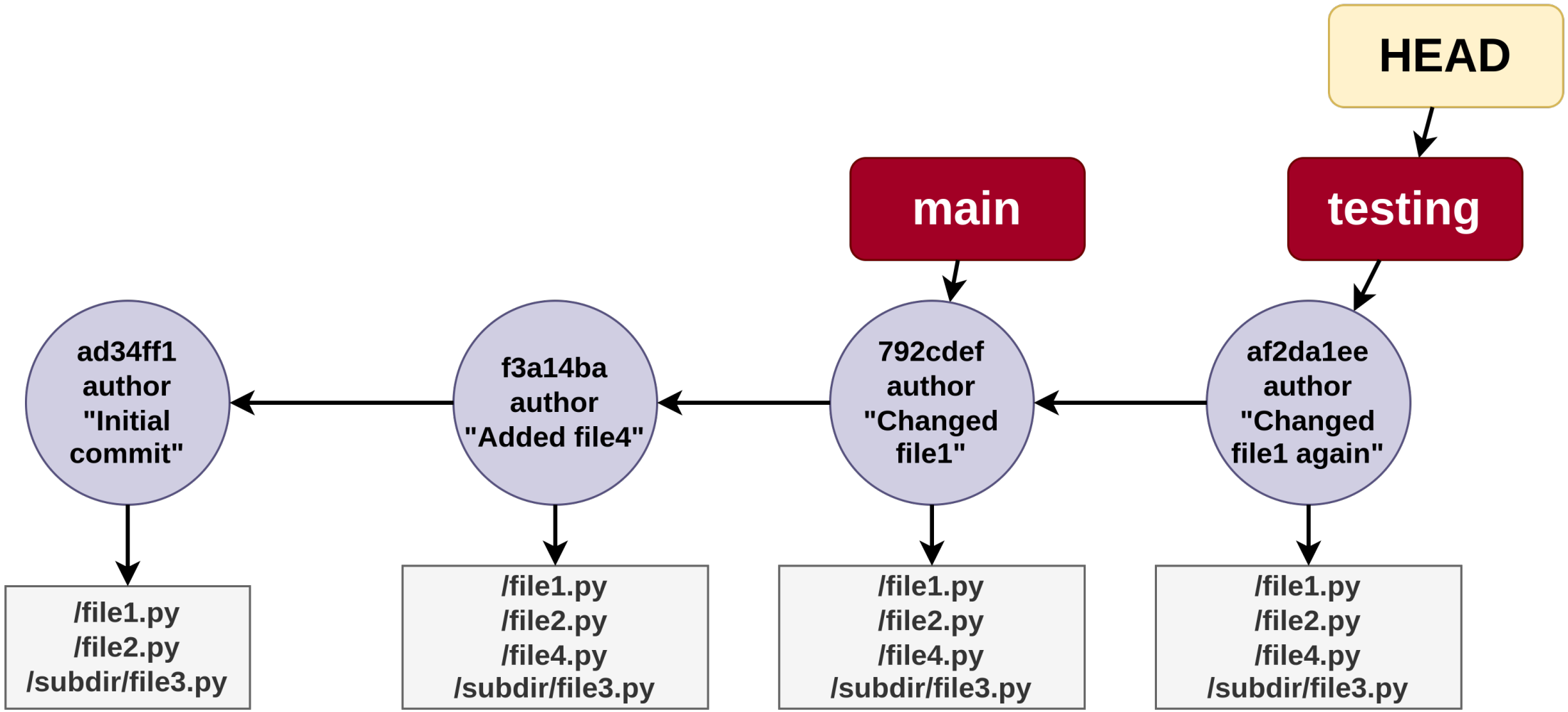


Branches

If we make a commit now, the branch that is going to move is the testing branch

- Because the branch pointed by HEAD is always the one that moves

```
$ git commit -a -m "Changed file1 again"
```

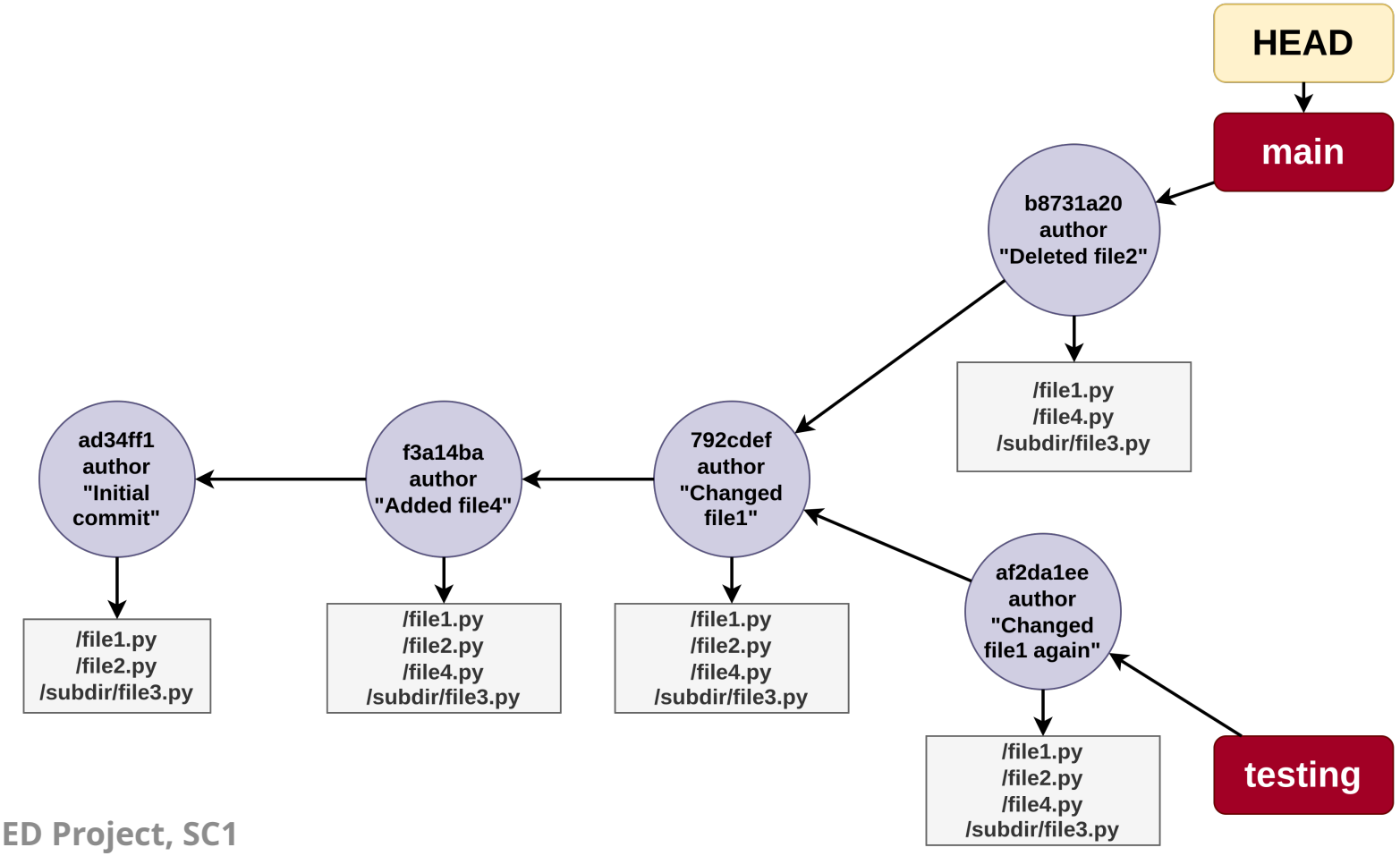


Branches

If we do a checkout of the main branch (moving thus to that branch) and a new commit:

- Some of the commits of the repository will only be reachable from main, others only from testing, and others from both
- We will thus have two branches which changes can evolve independently of each other

```
$ git checkout main
$ rm file2
$ git add .
$ git commit -m "Deleted file2"
```



Merging branches

The basic command to combine branches is `git merge`

This command takes two or more branches and:

- If possible, it moves one of them so both end up pointing to the same commit
  - This is a *fast-forward* merge, and it is only possible when all the commits reachable from one of the branches are also reachable from the other
- If a fast-forward is not possible, this is most often the case, it creates a new commit (a merge commit)

A merge commit has more than one parent

Merging branches

The `git merge` command may finish automatically

- Because Git has been able to combine the changes in the commits pointed by the branches being merged because those changes are compatible

But it may also stop in the middle pointing out some conflicts

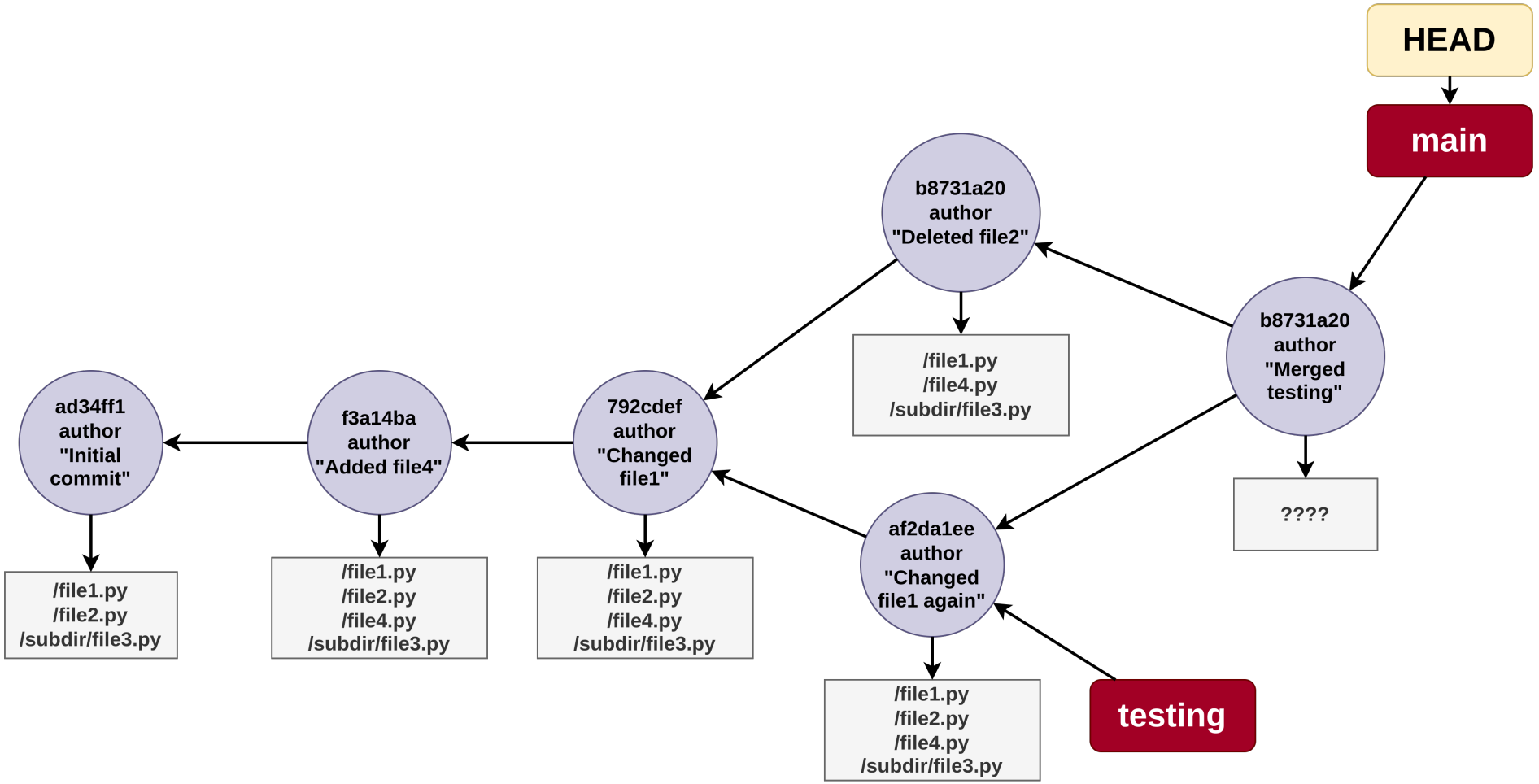
- Because some changes can't be automatically combined and a person has to decide how to integrate them
- This will happen from time to time, and we need to learn how to solve it
  - But if this is very frequent, we may have code which is badly modularized, or perhaps we are distributing the work in our team the wrong way

Merging branches

Once we have solved all the conflicts, we will be able to finish the merge by making a commit

- If we don't know how to solve them, we may interrupt the merge with `git merge --abort`, but this command will not always work

```
$ git merge testing
```



Collaborative work with Git and GitHub

Collaborative work

Everything so far has been focused on a single person working with a repository

The simplest way to work in a team with Git requires:

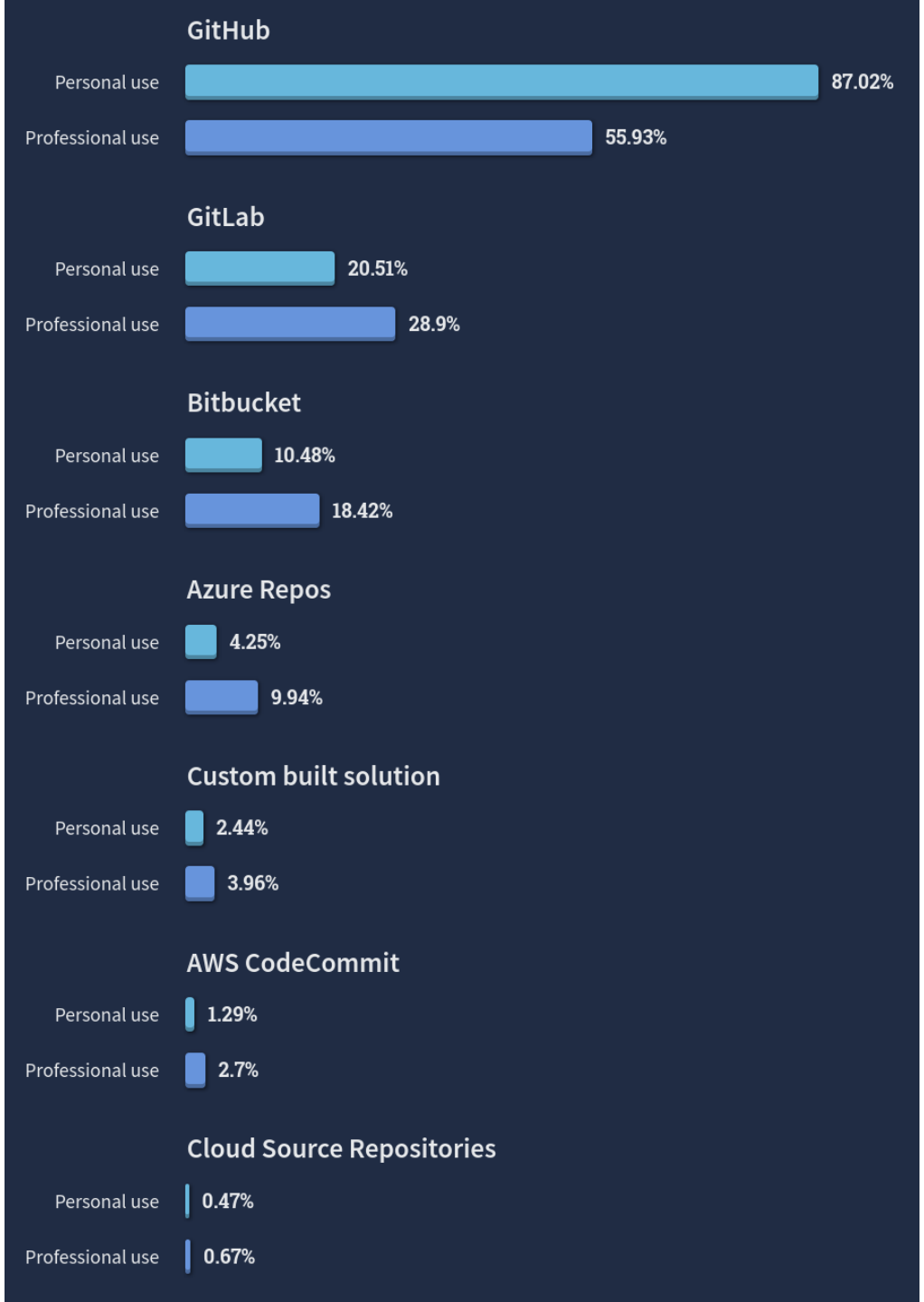
- Choosing a repository, accesible through a network, to act as the central one
- That each member of the team has a clone of this central repository in their computer
- Knowing the Git commands to synchronize local changes with the central repository and viceversa
- A workflow that organizes all this

GitHub

The simplest, and most common way to have a central repository accesible through a network is creating one in an online platform, such as GitHub

There are other alternatives, but GitHub is the most popular choice among software developers

- The popularity of Git owes **a lot** to the popularity of GitHub



https://survey.stackoverflow.co/2022#version-control-platforms

Centralized workflow with GitHub

The centralized workflow is the most straightforward approach to organize a team using Git

- It is a good choice for small or medium sized teams and a necessary foundation for advanced devops practices such as Continuous Delivery

Once every person in the team has a GitHub account, the first step is creating a central repository there (https://github.com), and providing read and write permissions to the team members

- That repository will have a single branch named main

Centralized workflow with GitHub

Every team member clones the central GitHub repository with the `git clone <url>` command to their computer

- The `<url>` is in the web of the repository in GitHub (green button labeled "Code")

This command:

- Creates a local repository (the clone) with all the history of commits in the GitHub repository
- Adds the GitHub repository as a *remote* (named `origin`) of the local one
- Creates a local main branch which *tracks* the main branch in the remote

Centralized workflow with GitHub

The `git pull` command will allow then to download and integrate locally the changes in the remote

- It combines two commands: `git fetch`, which downloads data from the remote, and `git merge` that ingrates the changes in the remote with local changes
  - As with any other merge, there may be conflicts

And the `git push` command will allow to upload local changes to the remote

- We can only push changes that have been merged first locally
  - "Pull before you push"

Exercises

Click to open the exercises

To know more

There are other workflows https://www.atlassian.com/git/tutorials/comparing-workflows

- The *forking workflow* was popularized by GitHub
- The *feature branch workflow* can be combined with others

The *rebase* command is an alternative to merge that rewrites history https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase

- It is often useful to use both for different things in the same project https://www.atlassian.com/git/tutorials/merging-vs-rebasing

To know more

Almost everything that you do with Git can be undone, although it is not always easy https://www.atlassian.com/git/tutorials/undoing-changes

We have just scratched the surface regarding the work with remotes

- For example, sharing other branches besides the main one is useful often, and even necessary in some workflows https://www.atlassian.com/git/tutorials/syncing

To know more

There are other commands to explore:

- `git stash` to put away temporarily some changes
- `git tag` to give a proper name to certain commits
- `git blame` to know who wrote every line of code in a commit
- `git diff` to see the differences between commits or branches
- `git bisect` to help you find the first commit where a bug was introduced
- ...

To know more

The Git official documentation is difficult to read, and most commands have lots of options which may radically change what they do

Chatbots based on large language models (such as ChatGPT, Copilot, Gemini...) can help you find the right options for a git command, or to find out what a given command with certain options is going to do

- As usual with these chatbots, you should verify the answers