Version control with git

PYOOP Workshop 2025

David Koch

Gefördert durch:

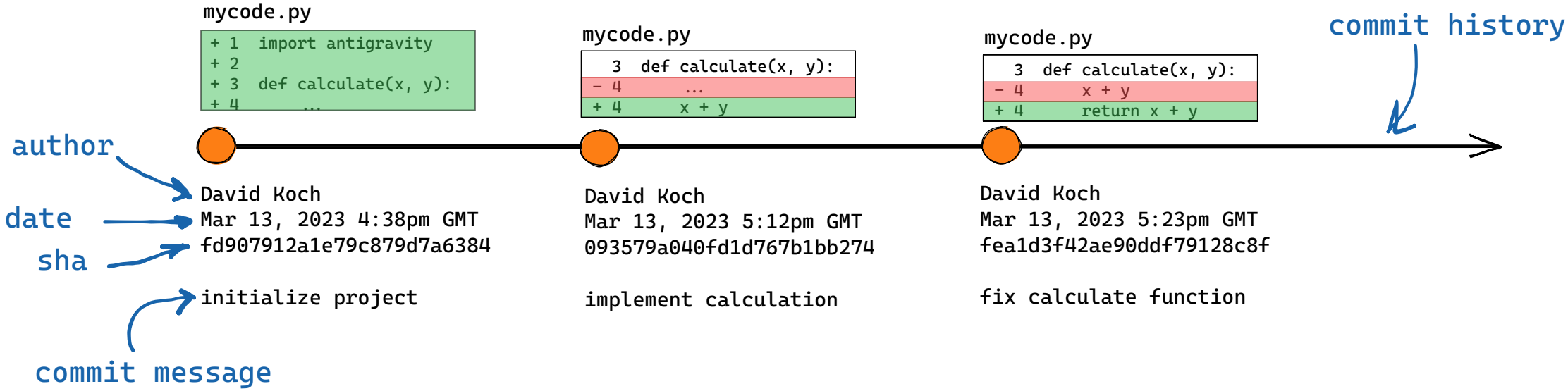
Bundesministerium
für Forschung, Technologie
und Raumfahrt

When you're dead but remember you forgot to git commit git push your last code iterations

Website & Documentation:

https://git-scm.org

- A project (= folder) managed by git is called a **repository**
- The "history" of the repository consists of **commits**
- A commit is like a snapshot of the repository state; you decide *when* to commit, *which* changes to include, and *why*



- Each commit includes author, date, commit message, changes (**diffs**), and pointer(s) to one or more parent commits

Create a commit

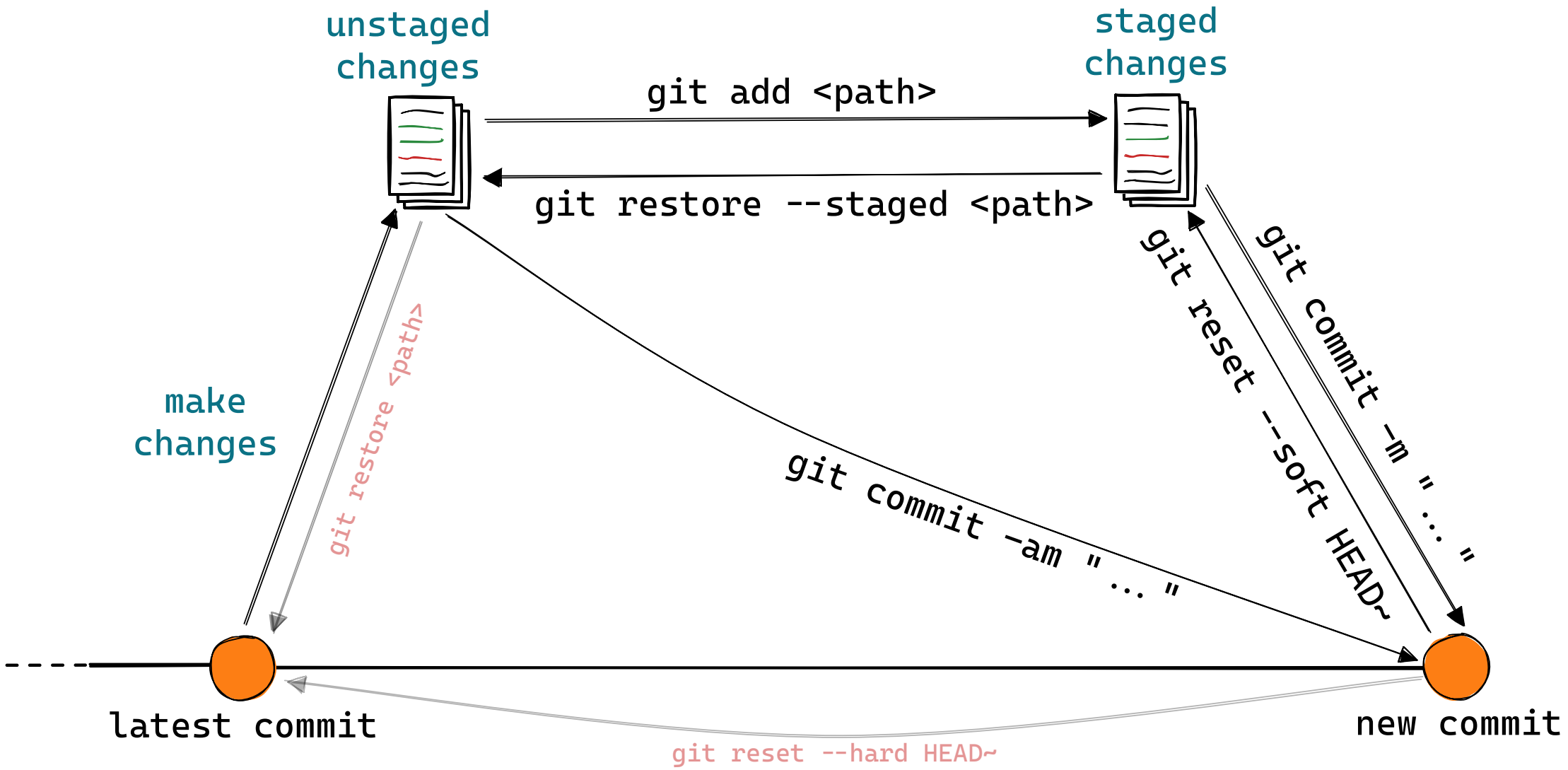
Commits are created with the `git commit` command. Git only commits changes that are in the **staging area**. Stage changes with `git add`.

```
git add <path>   # add whatever you want to include in the commit
git commit -m "meaningful commit message"
```

To get additional info on your repository's status:

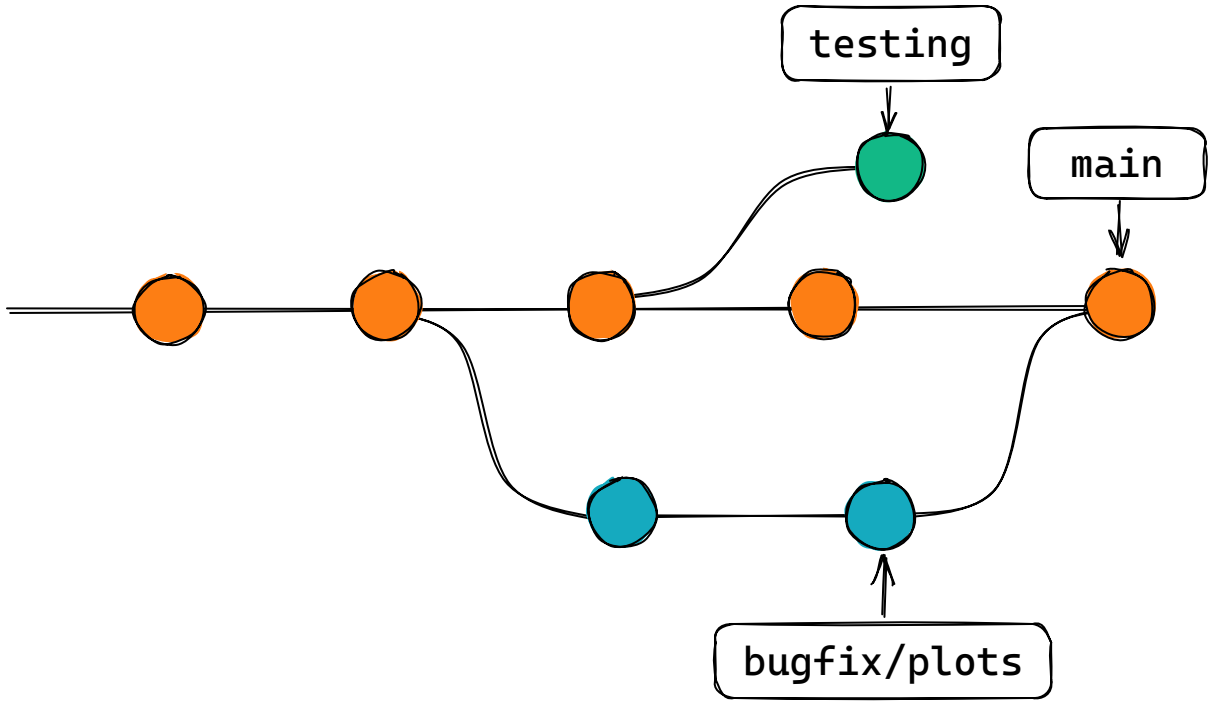
```
git status            # list changed files since last commit
git diff              # see what changes are not staged
git diff --staged     # see what changes are staged
git show <SHA>        # show information about a specific commit
git log               # show commit history of current branch
```

Create a commit



Branches

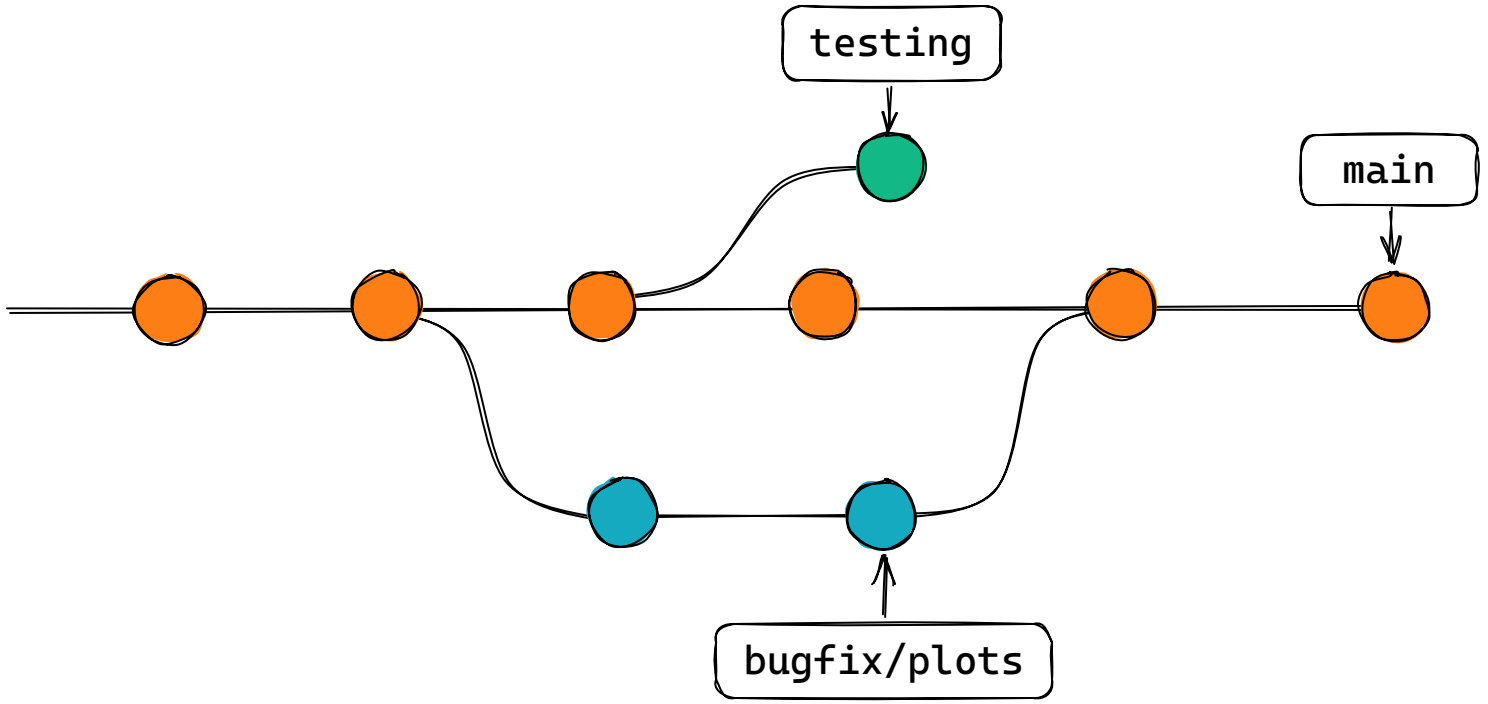
The commit history (commit tree) does not have to be linear!



A **branch** points to a commit and is automatically updated when a new commit is added.

Branches

The commit history (commit tree) does not have to be linear!



A **branch** points to a commit and is automatically updated when a new commit is added.

Branches

Create a new branch based on the latest commit and switch to it:

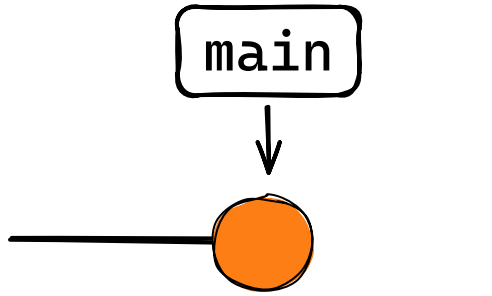
```
git checkout -b <new-branch-name>
```

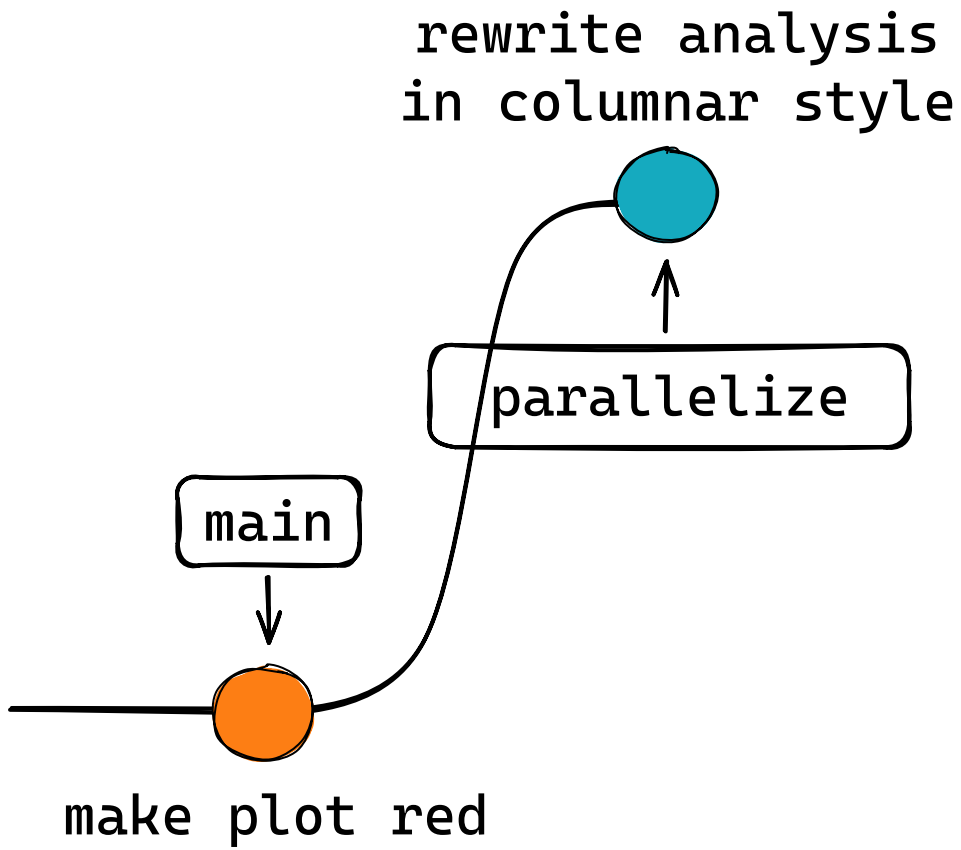
Switch to an existing branch (only works if the working directory is clean — no unstaged files):

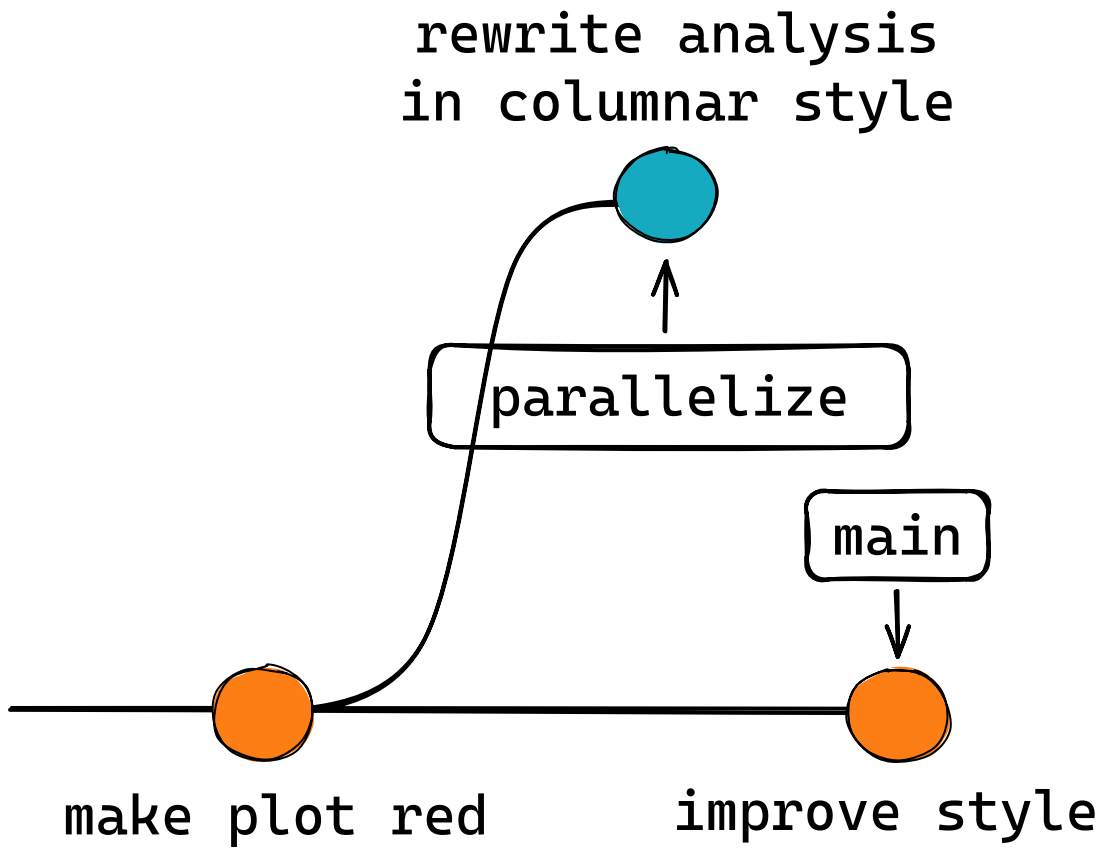
```
git checkout <branch-name>
```

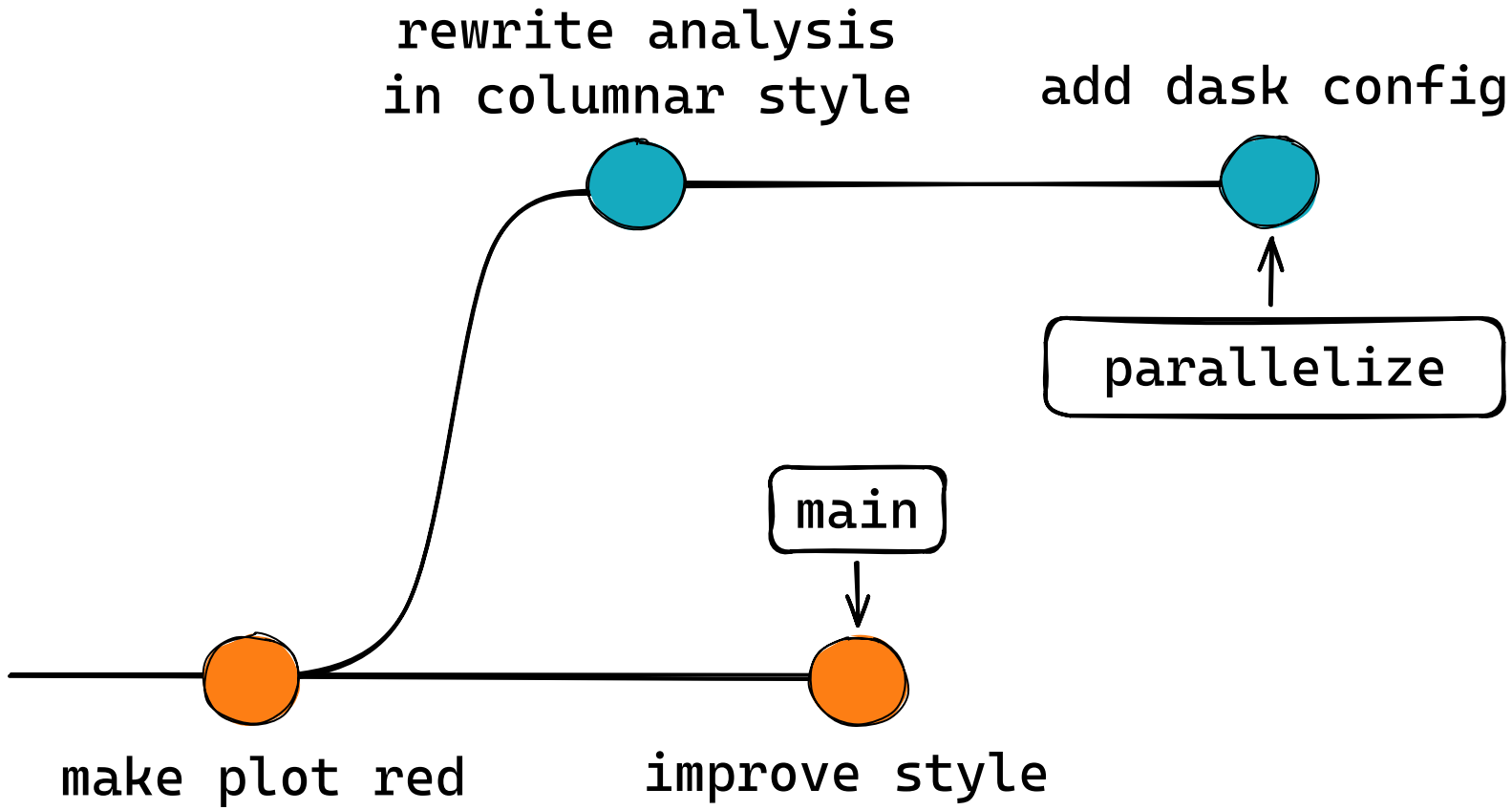
List existing branches and see which branch you're currently on:

```
git branch
```


main
make plot red

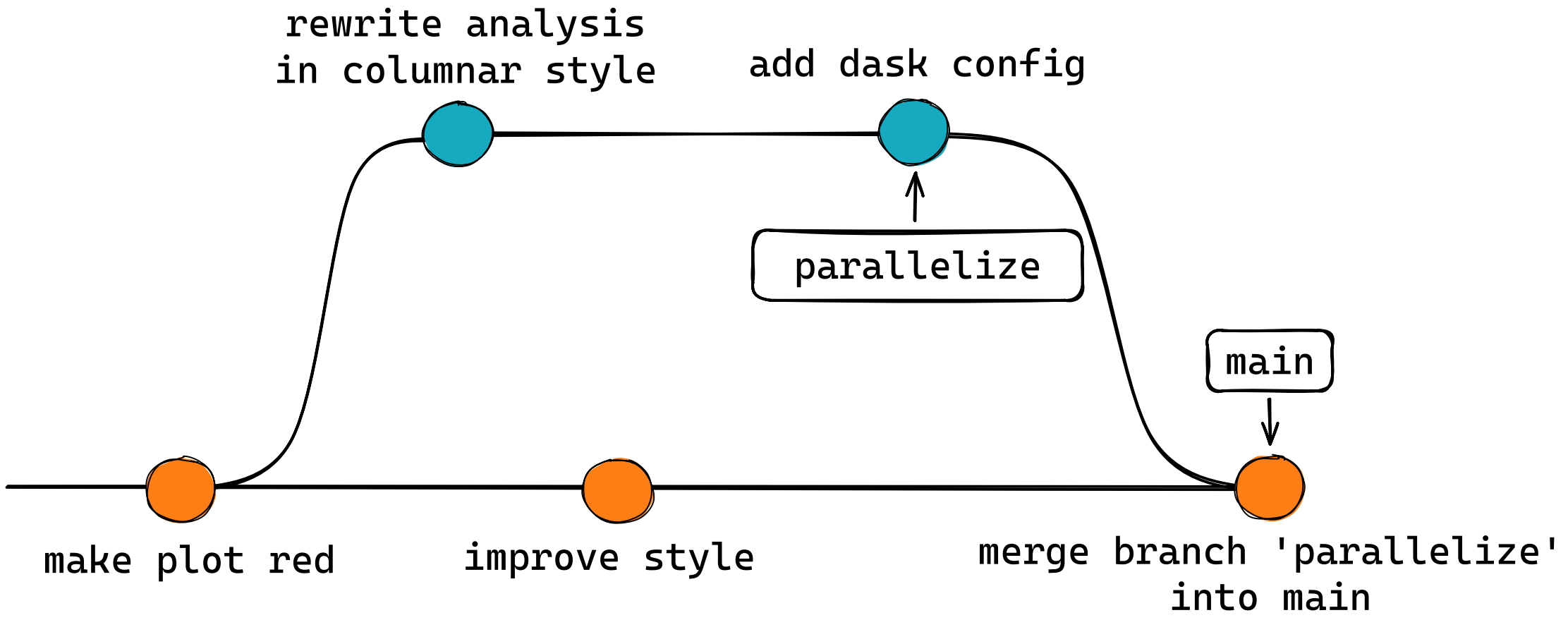

rewrite analysis
in columnar style
parallelize
main
make plot red


rewrite analysis
in columnar style
parallelize
main
make plot red
improve style


rewrite analysis
in columnar style
add dask config
parallelize
main
make plot red
improve style

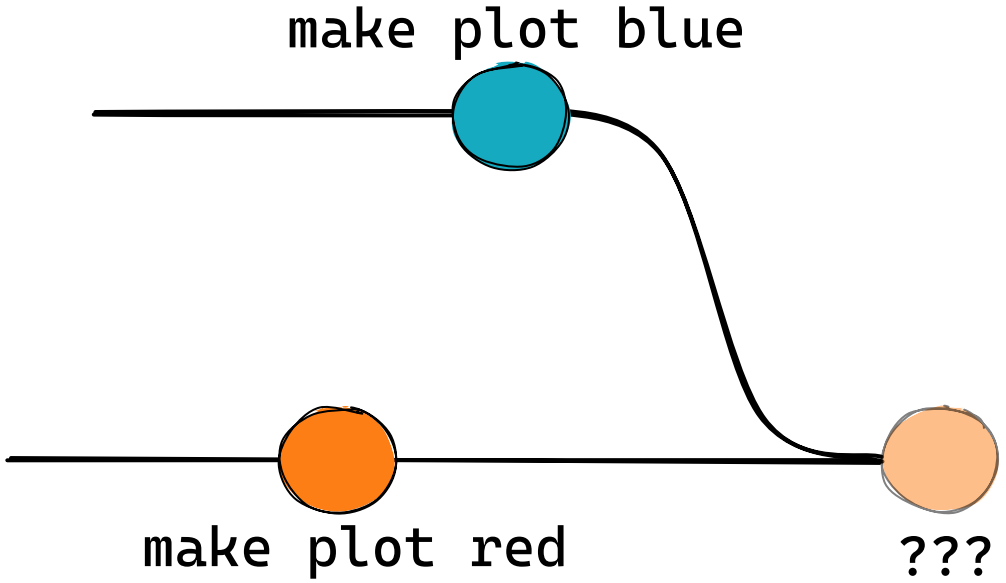
Merging

```
git checkout <branch-to-merge-into>
git merge <branch-to-merge> --no-edit
```



Merge Conflicts

In some situations git can't automatically merge → Merge Conflict!




```
CONFLICT (content): Merge conflict in analsis.py
Automatic merge failed; fix conflicts and then commit the result.
```

Resolving Merge Conflicts

Git marks the spots where conflicting changes need to be merged manually:

```
<<<<<<< HEAD
plt.hist(df["B0_mbc"], range=(5.2, 5.3), color="red", bins=20)
=======
plt.hist(df["B0_mbc"], range=(5.2, 5.3), color="blue", bins=20)
>>>>>>> styling
```

To resolve the conflict, remove the markers (`<<< HEAD`, `===`, `>>> other-branch`) and decide which change to keep. Then:

```
git add .   # stage all changes
git commit  # finish the merge
# or abort the merge
git merge --abort
```

Resolving Merge Conflicts

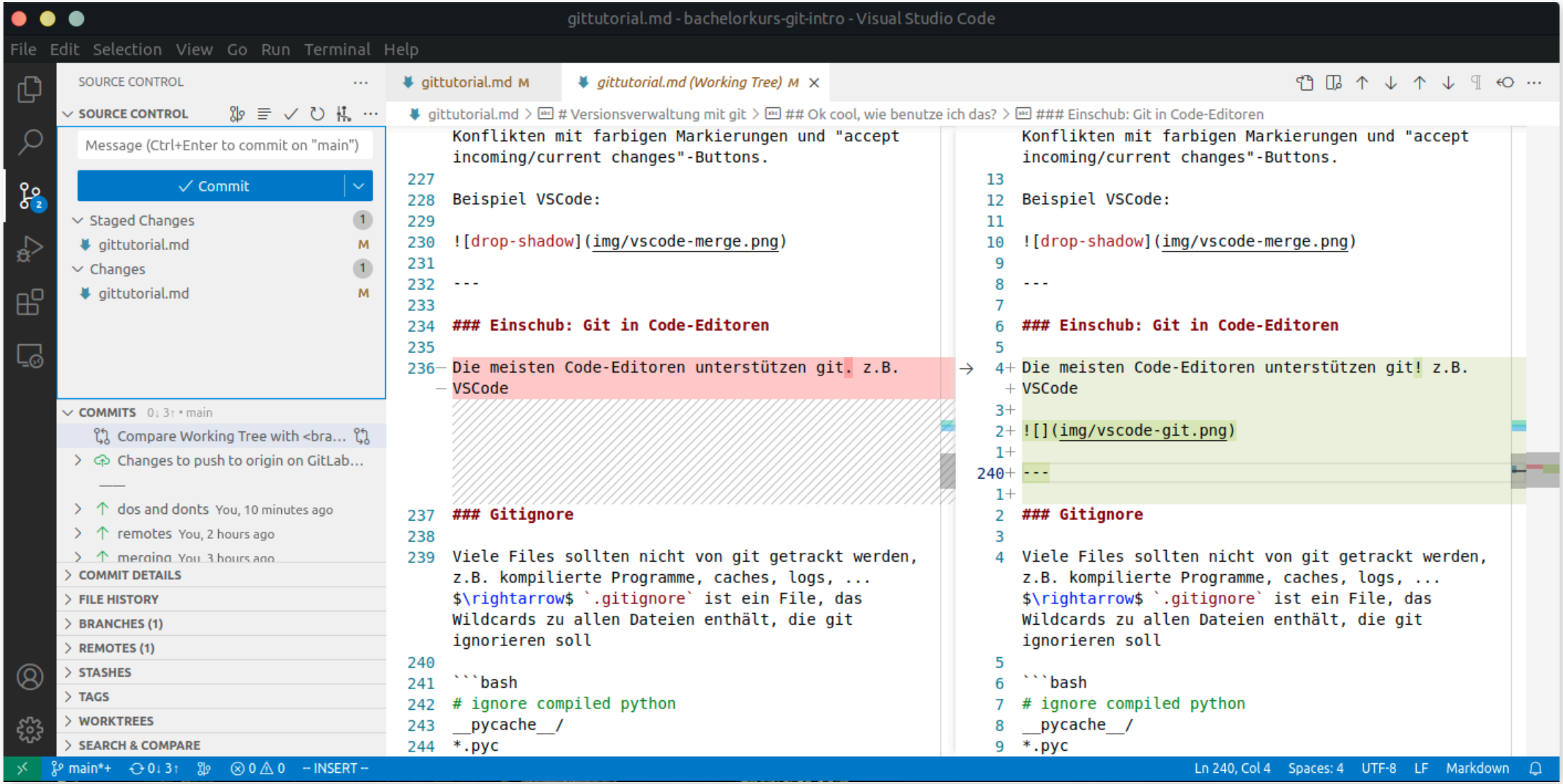
Many code editors assist in resolving merge conflicts with color highlights and "accept incoming/current changes" buttons.

Example VSCode:

```
15    df = ak.concatenate(data)
16
      Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
17    <<<<<<< HEAD (Current Change)
18    plt.hist(df["B0_mbc"], range=(5.2, 5.3), color="red", bins=20)
19    =======
20    plt.hist(df["B0_mbc"], range=(5.2, 5.3), color="green", bins=20)
21    >>>>>>> colors (Incoming Change)
22    plt.title("B0_mbc")
```

Insert: Git in Code Editors

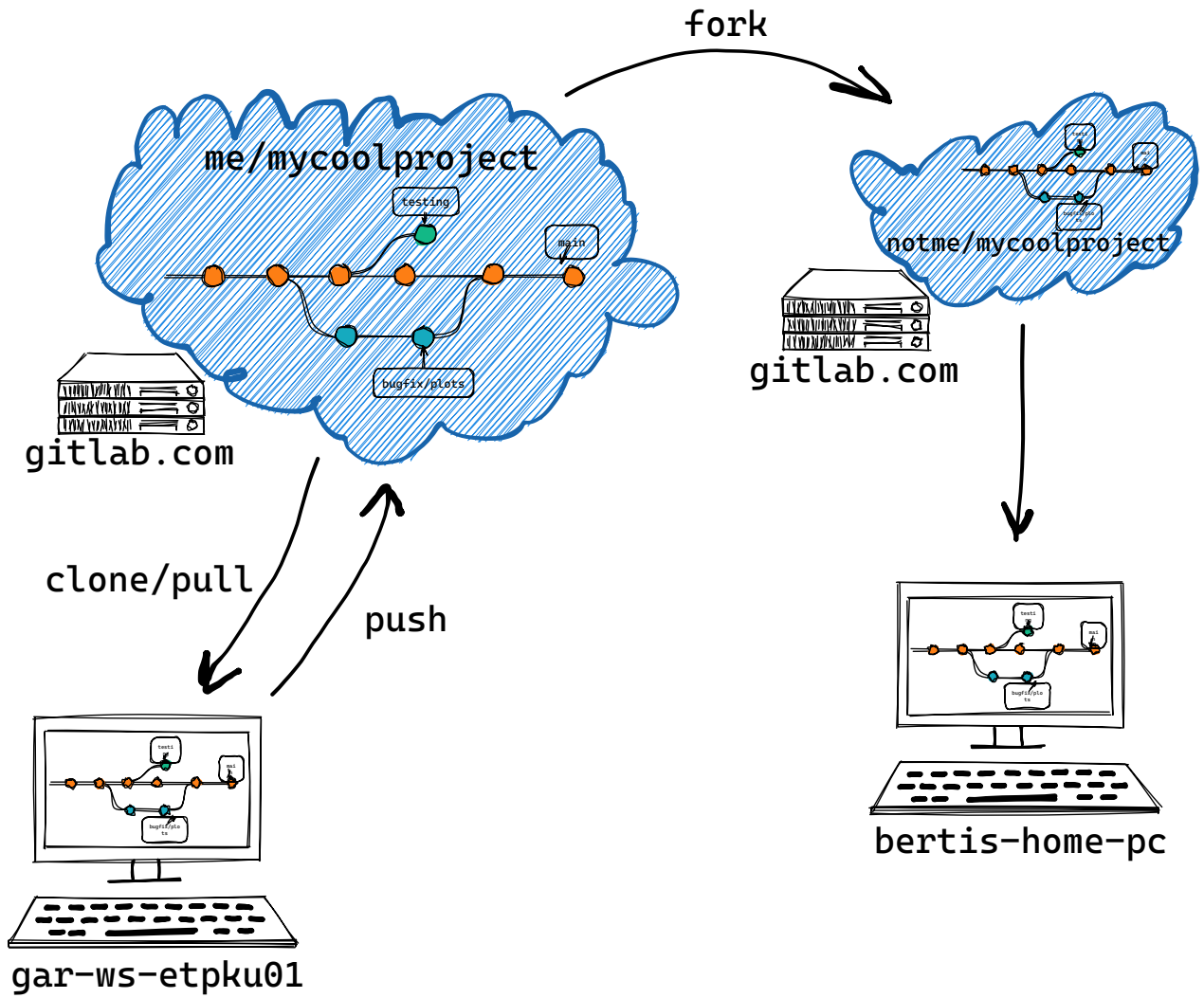
Most code editors support git! e.g., VSCode



Remotes

A git repository can have one or more **remotes**, identical copies on a server like

- https://github.com
- https://gitlab.com
- https://gitlab.physik.uni-muenchen.de
- …



Remotes

A remotely existing repository can be copied using `git clone`.

```
git clone <url>
```

In the local copy, the remote is called `origin` per default.

Remotes

If you already have a local repository and want to create a remote for it, go to your chosen platform, create a blank repository there, copy the URL (*not* from the browser, use the one ending in `.git` that appears in the instructions), then:

```
git remote add origin <url>
git remote -v   # list remotes
```

`origin` is the local name of the remote repository. The name is arbitrary.

It is possible to have multiple remotes, e.g., `git remote add fork other-url`

Remotes

To keep the local and remote repositories in sync, you use push and pull:

```
# get most recent changes from the remote main branch
# and merge them into the local main branch
git pull origin branch
git pull   # shortcut
# push my recent changes in some branch
# to the remote 'origin'
git push [--set-origin] origin branch
```

`git push` only works if there are no newer changes on the remote! $\longrightarrow$ always `git pull` first

get remote changes without merging them into your local branch: `git fetch`

Remotes - Authentication

Services like GitHub or GitLab require authentication to push (and to pull/clone private repos). The remote URL determines the method:

- `git@...` → SSH authentication: upload your public SSH key to the server once
- `https://...` → HTTPS authentication: enter your password every time

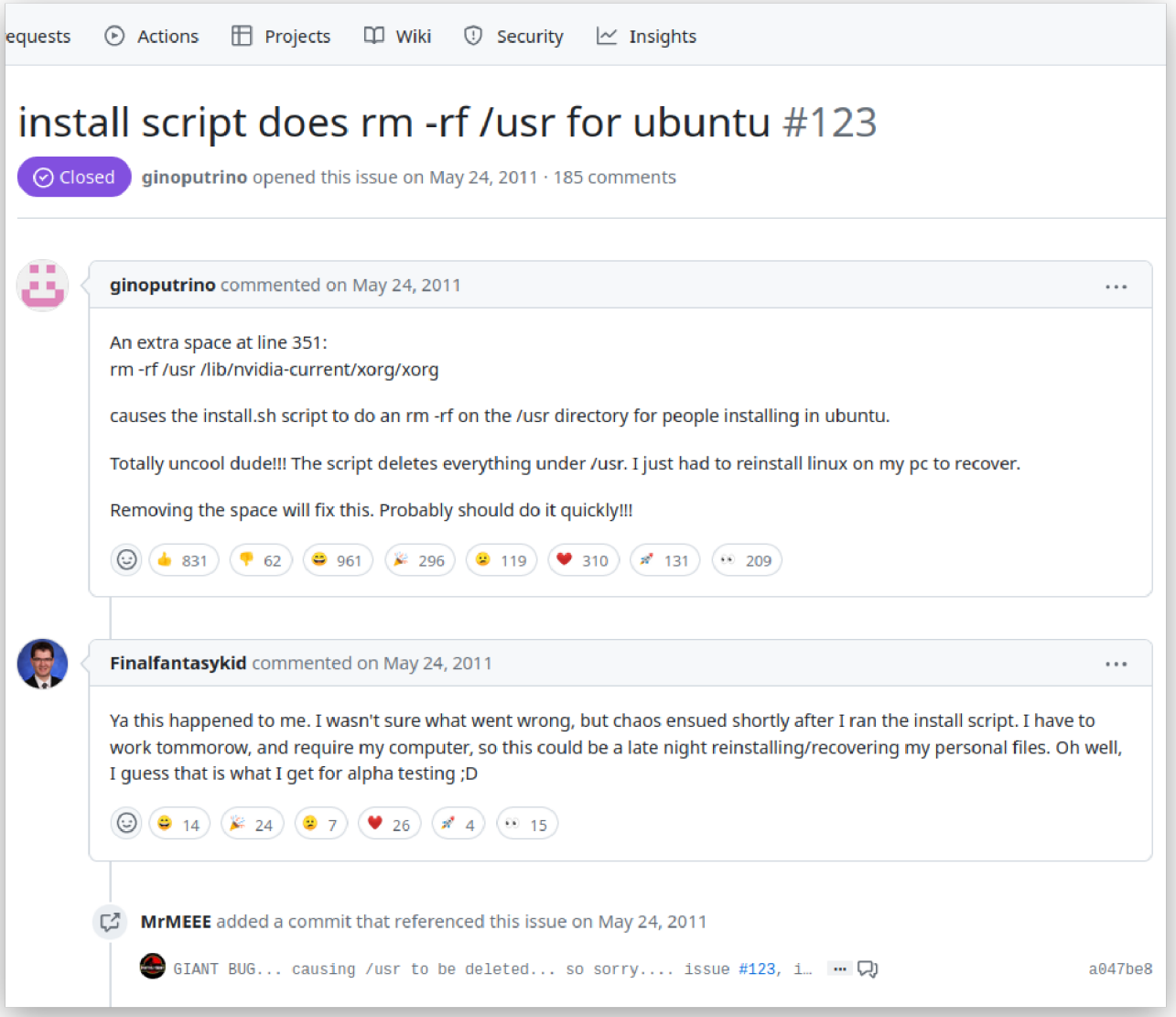
SSH setup instructions:

- gitlab: https://docs.gitlab.com/ee/ssh/index.html
- github: https://docs.github.com/en/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account

Remotes - Not Just a Backup

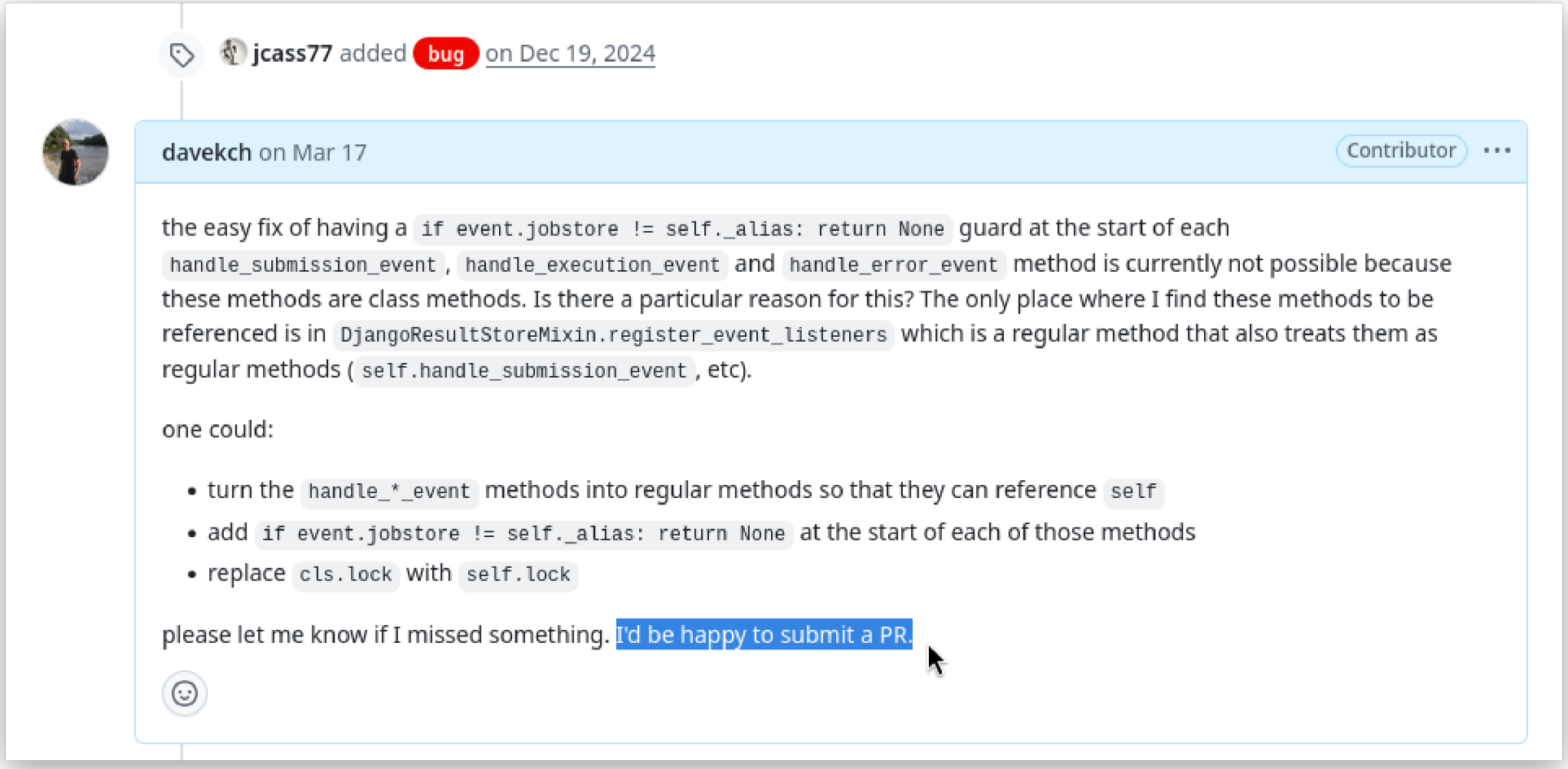
social coding

- Create bug reports
- Create pull requests (/merge requests)
- Review code
- Comment, like, react, …



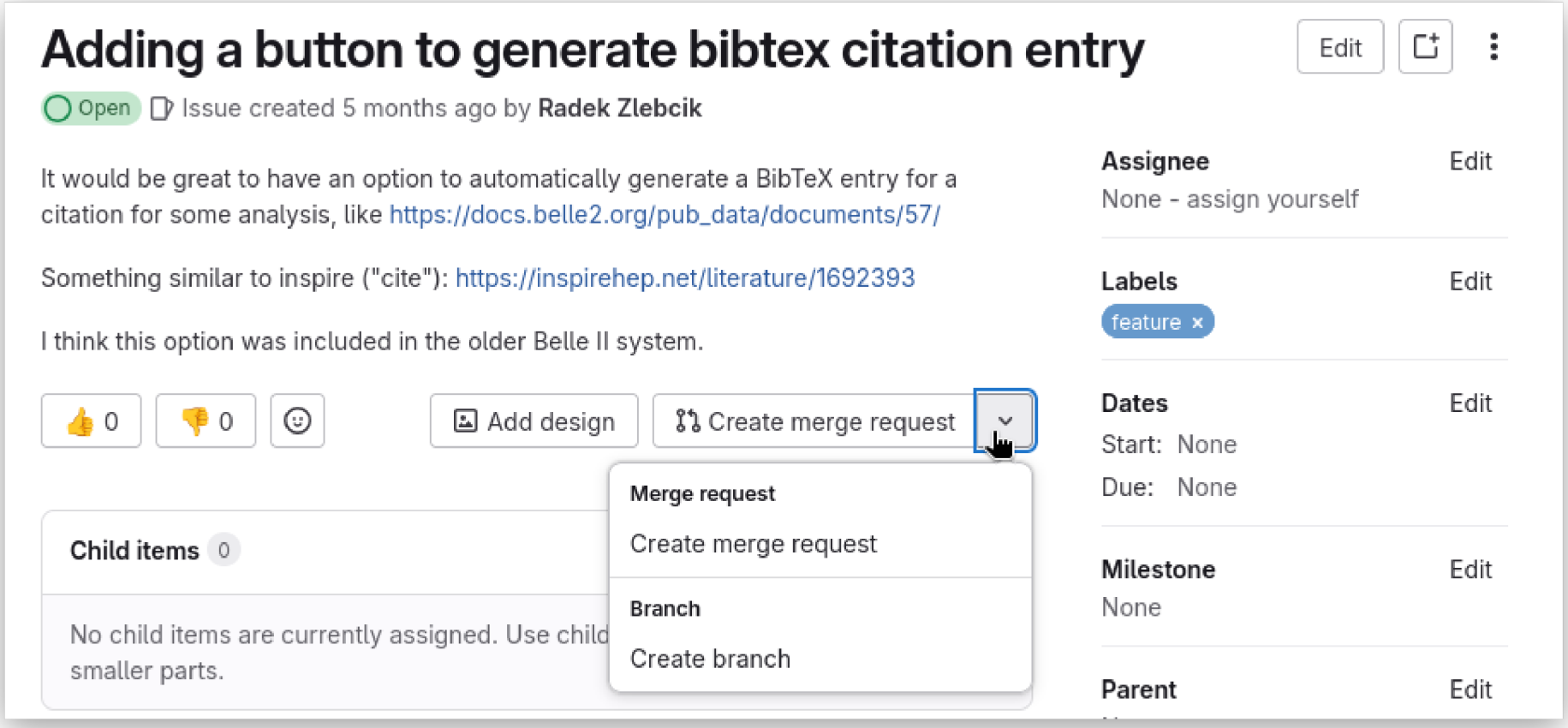
Collaboration -- how to contribute to a project (Github / Gitlab)

1. Create an issue to report a bug or describe a feature request
2. Indicate that you would like to work on it! Best also describe what you plan to do. *example: github*

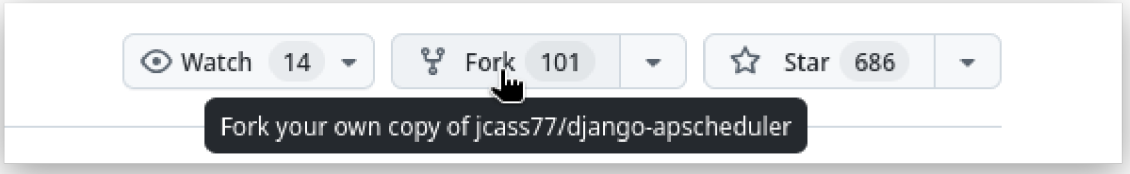


3. If you have the required permissions, create a branch + PR directly on that remote repository

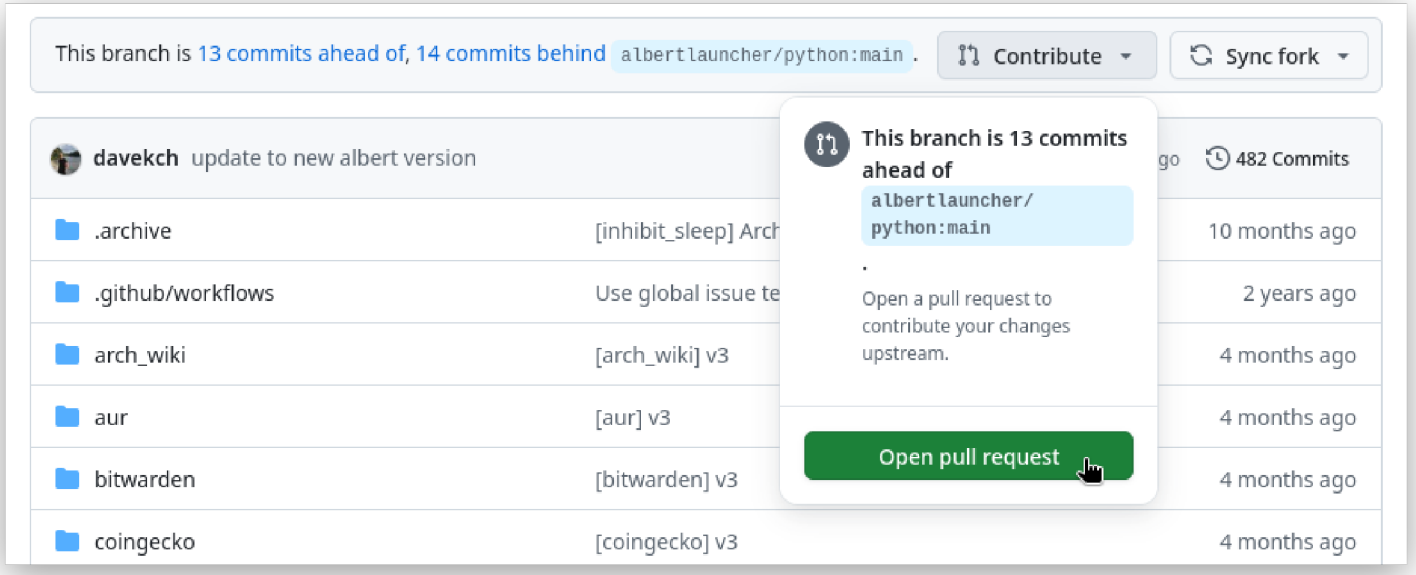
   example: gitlab



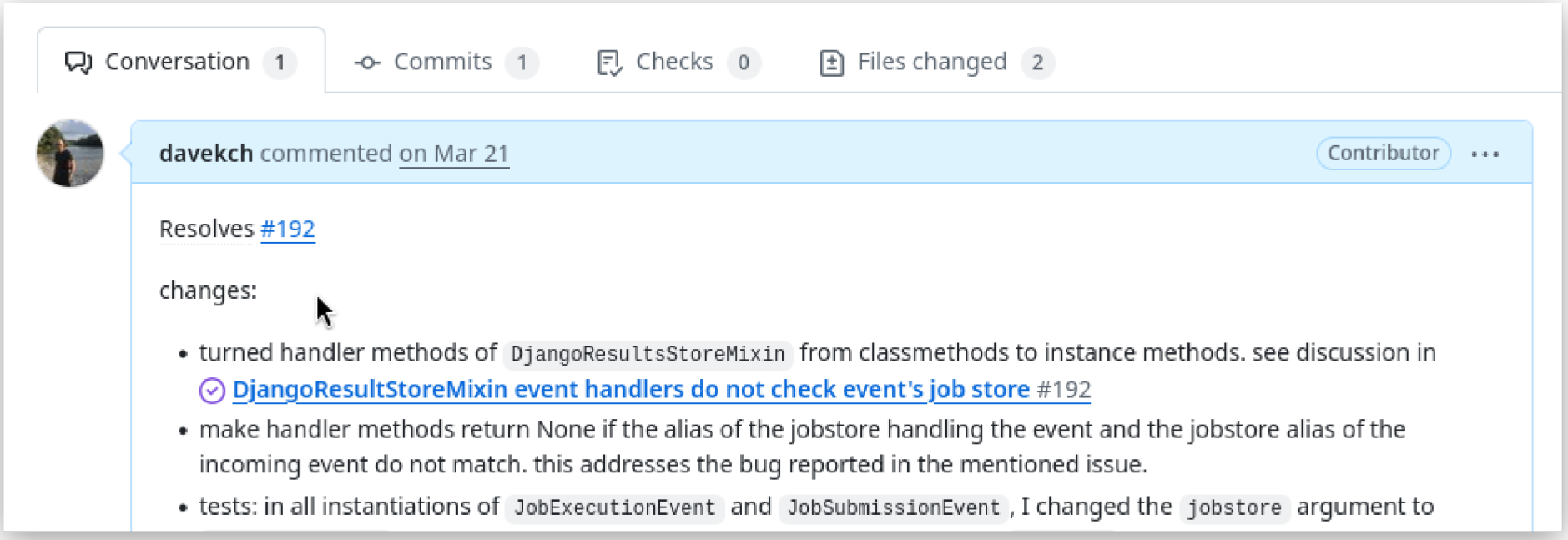
3. If you don't have permissions to create a branch on that remote, fork the project first and create a branch there. *example: github*



4. Fix the bug / implement the feature on your branch. Push your changes.
5. Create a merge request (MR, Gitlab language) / pull request (PR, Github language) if there is none yet. *example: github*

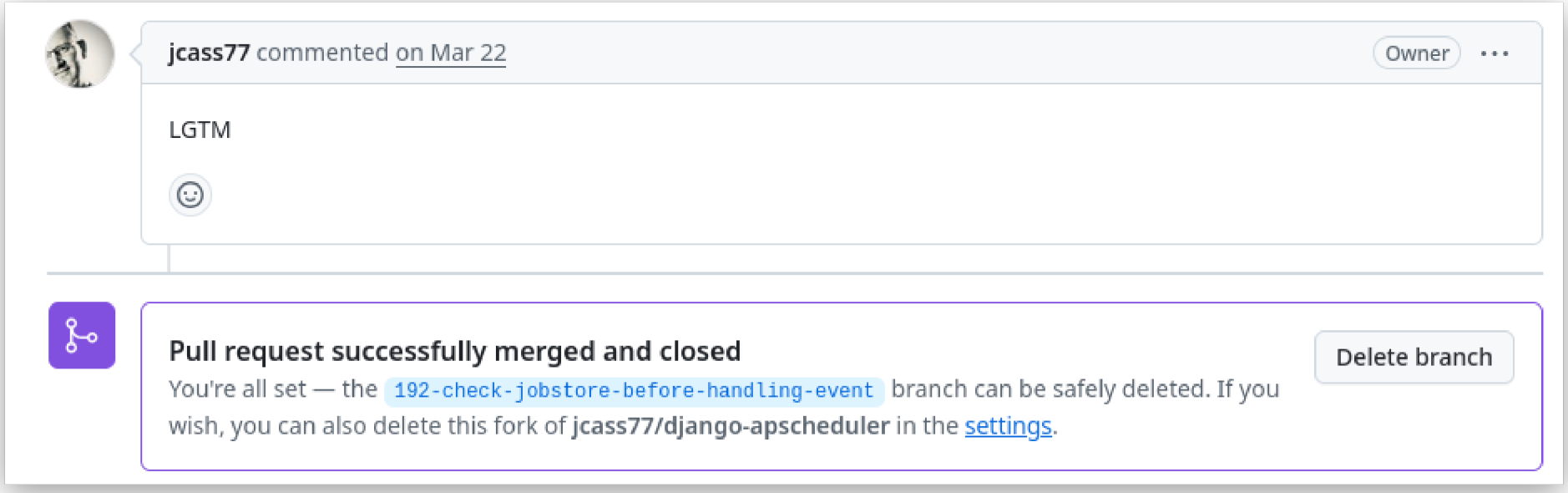


5.2 Make sure to link the related issue by mentioning it (`#<issue-number>`).
Describe what you did. *example: github*



6. Wait for your changes to be reviewed. Implement requested changes.

7. Eventually, a maintainer may or may not approve and merge your changes.



8. Don't be discouraged if your changes don't get approved in the end! You can always maintain your own fork.

If it's mainly you working on your own project, it can still be helpful to follow this process to make full use of Github's / Gitlab's project management features.

Tags

give a name to a specific commit; commonly used to mark releases

```
git tag v0.3.0  # current HEAD is now also called v0.3.0
git push --tags
```

`checkout` also recognises tags:

```
git checkout v0.2.1
```

list all commits between version 0.2.11 and the current HEAD (useful for writing changelogs!):

```
git log --oneline v0.2.11..
```

Git LFS

- git diffs are **line based** $\rightarrow$ works only well for text
- when tracking a binary file with git and it changes, it has to store the new version as a separate blob $\rightarrow$ `.git` folder size increases each time!

$\Rightarrow$ git **L**arge **F**ile **S**torage to the rescue

https://git-lfs.com/

start using git lsf: `git lfs install`

tell git to track files with a certain ending with LFS:

```
git lfs track "*.root"
# these settings are saved in .gitattributes
git add .gitattributes
# continue using git like usual
git add somefile.root   # ← will be tracked with git lfs
```

want to switch to using git lfs? `git lfs migrate` **attention**: this will rewrite history

git lfs footguns / limitations

- if you clone a repo with LFS tracked files without having git lfs installed, you will only see "pointer-files"
- remote hosting service must support git lfs
- on Github: max 2GB per file

Submodules

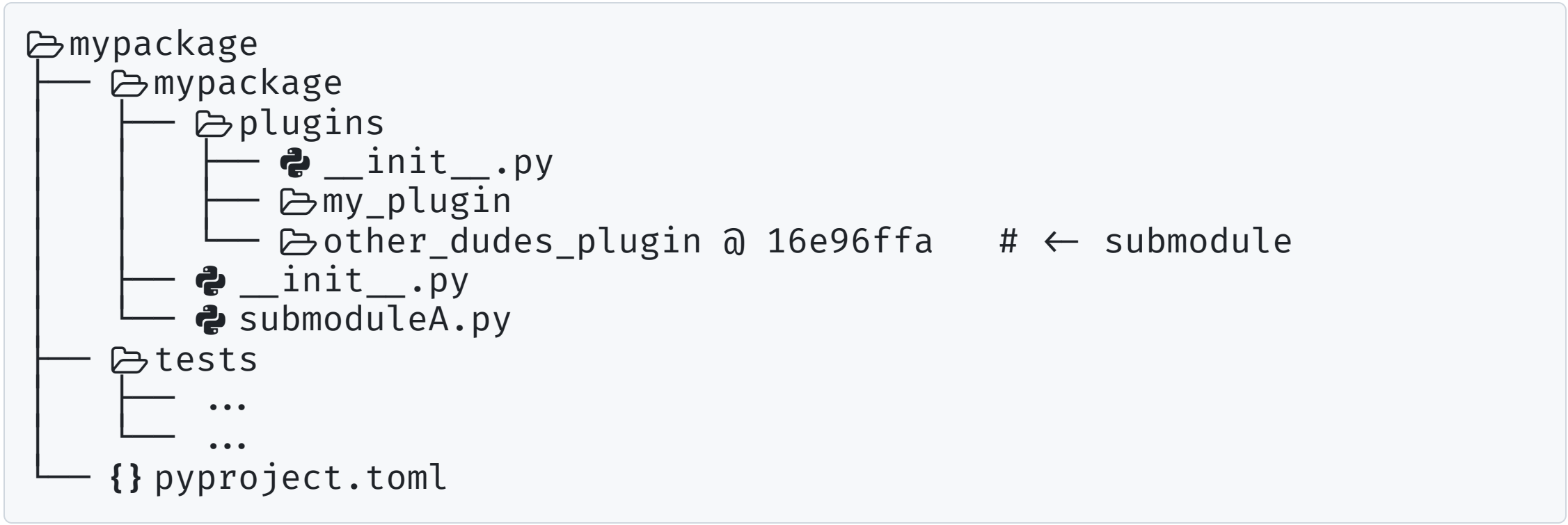
"git repository inside a git repository"

why?

- dependency on some fork of a library / need to pin to a commit
- sharing a common codebase in multiple projects
- "pseudo mono-repo": managing multiple dependent repos in a parent-repo
- no trust in package repository to be available in the long future

$\rightarrow$ if the dependency / common codebase is properly packaged and published in some package registry, there often is no need to use submodules

Examples


mypackage
mypackage
plugins
__init__.py
my_plugin
other_dudes_plugin @ 16e96ffa # ← submodule
__init__.py
submoduleA.py
tests
...
...
{} pyproject.toml

using submodules

cloning a repository that has submodules:

```
git clone --recursive <url>
```

pulling changes in submodules:

```
git submodule update
```

adding a submodule to a repository:

```
git submodule add <url> [<path>]
git add .gitmodules
git commit ...
```

making changes to a submodule without tripping up

```
cd my-submodule
git add; git commit; git push
cd ..
git add my-submodule
git commit ...
```

footguns:

- forgetting to commit changes inside the submodule
- forgetting to push changes inside the submodule (`git push` in the parent repo does *not* push changes in the submodule to the submodule's remote)

Notable mentions

- undo all unstaged changes without throwing them away: `git stash`; bring them back: `git stash pop`
- cloning only part of a (large) repository: sparse checkouts

```
git clone --filter=blob:none --no-checkout <url> && cd <repo>
git sparse-checkout init --cone
git sparse-checkout set <path-i-want>
```

- cloning only part of the history, eg only the latest commit: shallow checkouts

```
git clone --depth 1 <url>
```

- see "where you've been": `git reflog`

`~/.gitconfig`

random neat little tricks for your configuration

aliases: define your own git commands

```
[alias]
    unstage = restore --staged
    # show a visual graph
    graph = log --graph --full-history --all --color
    latest-tag = describe --tags --abbrev=0
    # show all remote commits that are not yet merged into local
    incoming = "!f() { git fetch && git log ..origin/$(git rev-parse --abbrev-ref HEAD); }; f"
    # show all local commits that are not yet pushed
    outgoing = "!f() { git fetch && git log origin/$(git rev-parse --abbrev-ref HEAD)..; }; f"
```

allows you to type the command `git unstage`, `git graph`, …

`~/.gitconfig`

random neat little tricks for your configuration

conditional includes:

```
[includeIf "gitdir:/home/davekoch/"]
    path = ~/.gitconfig-personal
[includeIf "gitdir:/home/davekoch/Documents/arbeit/"]
    path = ~/.gitconfig-work
```

```
# ~/.gitconfig-work
[user]
    name = David Koch
    email = david.koch@physik.uni-muenchen.de
```

What have we learned today?

- how to use the basics of git: creating commits, inspecting diffs and the log
- why and how to use branches, merging with and without conflicts
- how to keep your local repository in sync with a remote repository: clone, push & pull
- how to make use of Github's / Gitlab's project management features to follow the development cycle: issue $\rightarrow$ fork or branch $\rightarrow$ merge request $\rightarrow$ review $\rightarrow$ merge
- how to use tags to mark releases
- how to use git lfs
- how to use submodules
- ...

Exercise

https://github.com/davekch/PYOPP-2025-git-tutorial

Backup

Brief Insert: SHA

The "sha" of a commit is the output of a cryptographic hash function called SHA-1. All information defining the commit (including parent commit!) is used as input.

- small change in input $\rightarrow$ completely different output
- one-way: input can't be derived from output
- no collisions: (almost) impossible to find two different inputs with same output

mycode.py

```
  3  def calculate(x, y):
- 4      ...
+ 4      x + y
```

David Koch
Mar 13, 2023 5:12pm GMT

implement calculation

SHA-1 → 093579a040fd1d767b1bb274

mycode.py

```
  3  def calculate(x, y):
- 4      ...
+ 4      x + y
```

David Koch
Mar 13, 2023 5:12pm GMT

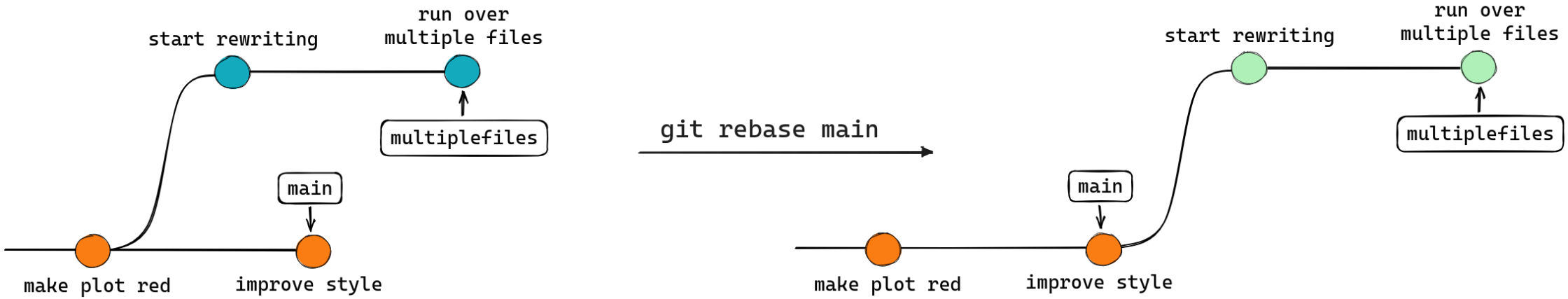
implement calculate

SHA-1 → 3072717fc8cb51229bbe4424

→ the SHA / commit ID *uniquely* identifies a commit

rebase

If for some reason you prefer to have a linear history, you can use `git rebase`

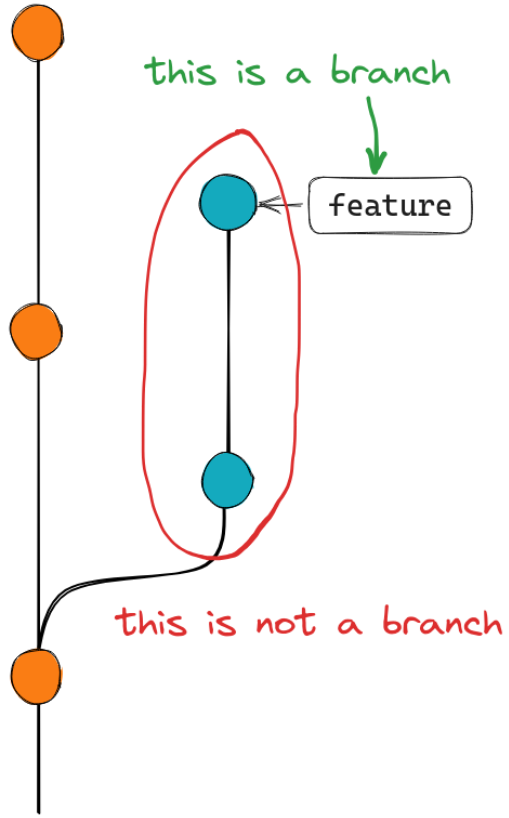


`rebase` creates new commits on top of main (in this example) and deletes the old ones. This is *rewriting history*. Git will not allow you to push a branch with an altered history unless you do `git push --force`.

Never rewrite history on branches other people work on too unless it's coordinated. Rewriting history can cause lots of friction.

The intuition that a branch is a literal "branch" that branched off some other branch works well in many scenarios but it has its limitations and it is technically false.

But what *is* a branch?



In git, a branch is just a **reference to a commit** that gets automatically updated when new commits are added on top.

This means:

- a branch not only contains the "offshoot" commits but the entire history that came before its most recent commit
- there is no such thing as a base or a parent of a branch
- instead: common ancestor of two or more branches

That's for example the reason why you can't `git rebase` without specifying a target (e.g., `git rebase main`): git does not know your branch "branched-off" of `main`.

Inspect branches by looking into `.git/refs/heads`.

read more

Gitignore

Many files shouldn't be tracked by git, e.g., compiled programs, caches, logs, ... → `.gitignore` is a file containing wildcards for files git should ignore.

```
# ignore compiled python
__pycache__/
*.pyc

# ignore pdfs
*.pdf
# but not this one
!super-important.pdf
```

If a file is already tracked and you now want git to ignore it:

```
git rm --cached <file>
```