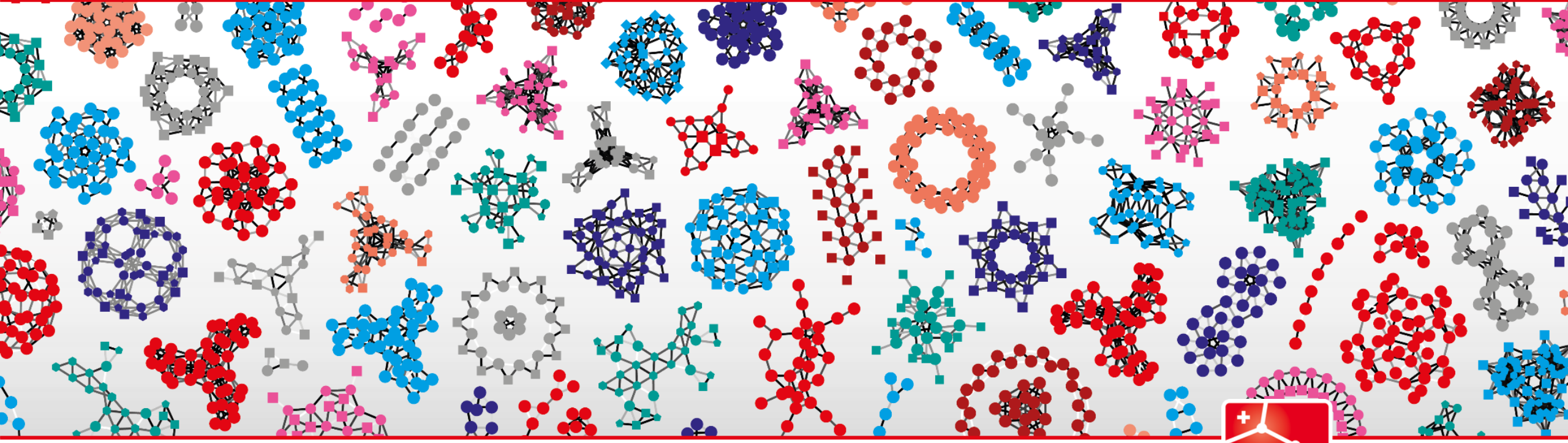


Version control with Git - first steps

Swiss Institute of Bioinformatics

www.sib.swiss

Robin Engler
Vassilios Ioannidis

Lausanne, 11-13 Oct. 2023

First steps with Git: course outline

- **Introduction** to Version Control Systems and Git.
- **Git basics**: your first commit.
- **Git concepts**: commits, the HEAD pointer and the Git index.
- **Git branches**: introduction to branched workflows and collaborative workflow examples.
- **Branch management:** merge, rebase and cherry-pick.
- **Retrieving data from the Git database:** git checkout.
- **Working with remotes**: collaborating with Git.
- **GitHub:** an overview.

Course resources

Course home page: Slides, exercises, exercise solutions, command summary (cheat sheet), setting-up your environment, link to feedback form, links to references.

https://gitlab.sib.swiss/rengler/git_course_public

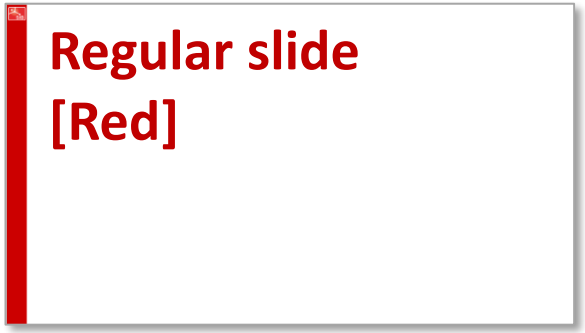
Google doc: Register for collaborative exercises (and optionally for exam), FAQ, ask questions.

https://docs.google.com/document/d/1EX72NInz-eA2d2GOa5aTB8D88GWb91Sk-sCNHwQYXqE

Questions: feel free to interrupt at anytime to ask questions, or use the Google doc.

Course slides

- 3 categories of slides:

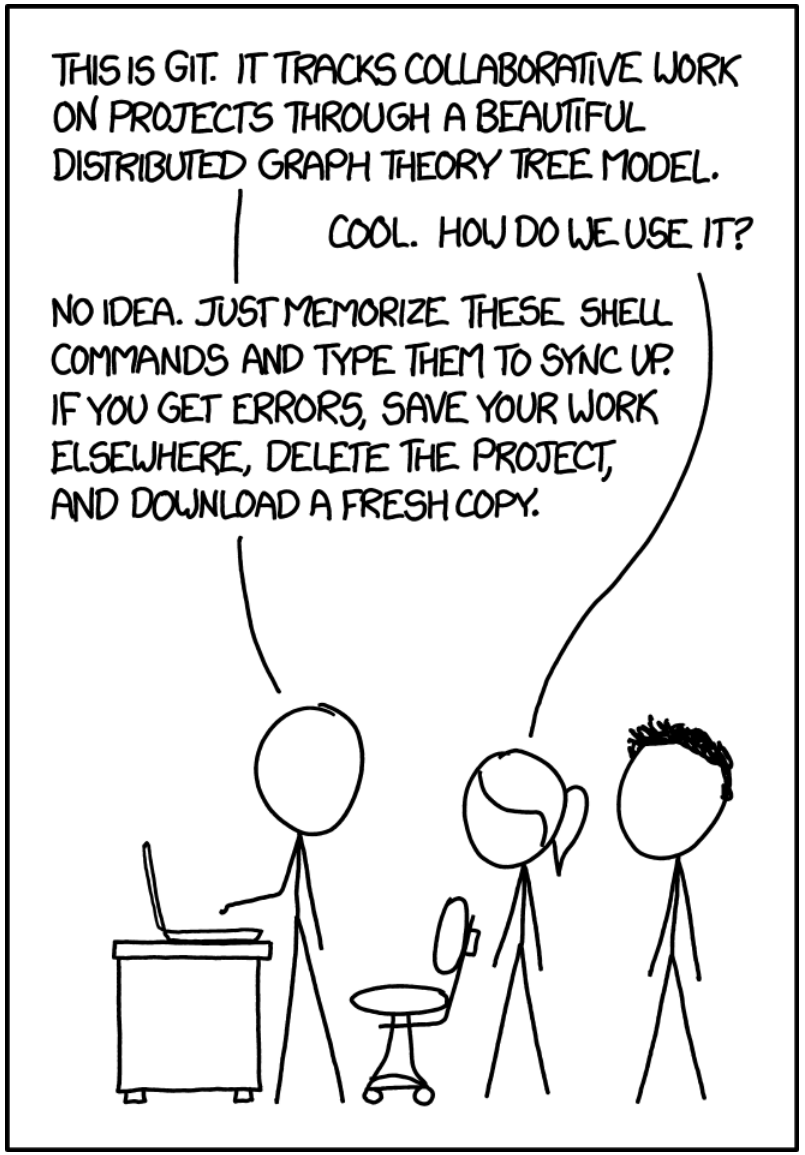


Slide covered in detail during the course.

Material available for your interest, to read on your own.
Not formally covered in the course.
We are of course happy to discuss it with you if you have questions.

Material we assume you know.
Covered quickly during the course.

Learning objective



source: https://xkcd.com/1597

Command line vs. graphical interface (GUI)

- This course focuses exclusively on **Git concepts** and **command line** usage.
- Many GUI (graphical user interface) software are available for Git, often integrated with code or text editors (e.g. Rstudio, Visual Studio Code, PyCharm, …), and it will be easy for you to start using them (if you wish to) once you know the command line usage and the concepts of Git.

version control

a (very) brief introduction

Why use version control ?

Version control systems (VCS), often also referred to as *source control/code managers* (SCM), are software designed to:

- Keep a **record of changes** made to (mostly) text-based content by **recording specific states** of a repository's content.
- **Associate metadata to changes**, such as author, date, description, tags (e.g. version).
- **Share** files among several people and allow **collaborative, simultaneous, work** on the repository's content.
- **Backup** strategy:
  - Repositories under VCS can typically be mirrored to more than one location.
  - The database allows to retrieve older versions of a document: if you delete something and end-up regretting it, the VCS can restore past content for you.
- In the case of Git, entire ecosystems such as GitHub or GitLab have emerged to offer **additional functionality**:
  - **Distribute** software and **documentation**.
  - **Run automated pipelines** for code testing and deployment (CI/CD).
  - Team and **project management tool** (e.g. issue tracking, continuous integration).

A (very brief) history of Git

- Created by **Linus Torvald** (who also wrote the first Linux kernel in his spare time…).
- Created to support the development of the Linux kernel code (> 20 million lines of code).
- **First release in 2005** - in a self-hosting Git repository… of course :-).

The first commit of Git's own repository by Linus Torvalds in 2005.

```
commit e83c5163316f89bfbde7d9ab23ca2e25604af29
Author: Linus Torvalds <torvalds@ppc970.osdl.org>
Date: Thu Apr 7 15:13:13 2005 -0700

Initial revision of "git", the information manager
from hell
```

(some of) The principles that guided the development of Git

Linus wasn't satisfied with existing version control software, so he wrote his own…
He had the following objectives (among others) in mind:

- **Distributed development:** allow parallel, asynchronous work in independent repositories that do not require constant synchronization with a central database. **Each local Git repo is a full copy of the project** so users can work independently and offline.

- **Maintain integrity and trust:** Since Git is a distributed VCS, maintaining integrity and trust between the different copies of a repositories is essential. **Git uses a blockchain-like approach to uniquely identify each change to a repository**, making it impossible to modify the history of a Git repo without other people noticing it.

- **Enforce documentation:** in Git, **each change to a repo must have an associated message**. This forces users to document their changes.

- **Easy branching/merging:** Git makes it easy to create new "lines of development" (a.k.a. branches) in a project. This encourages good working practices.

- **Free and open source:** users have the freedom to run, copy, distribute, study, change and improve the software.

Supplementary material…

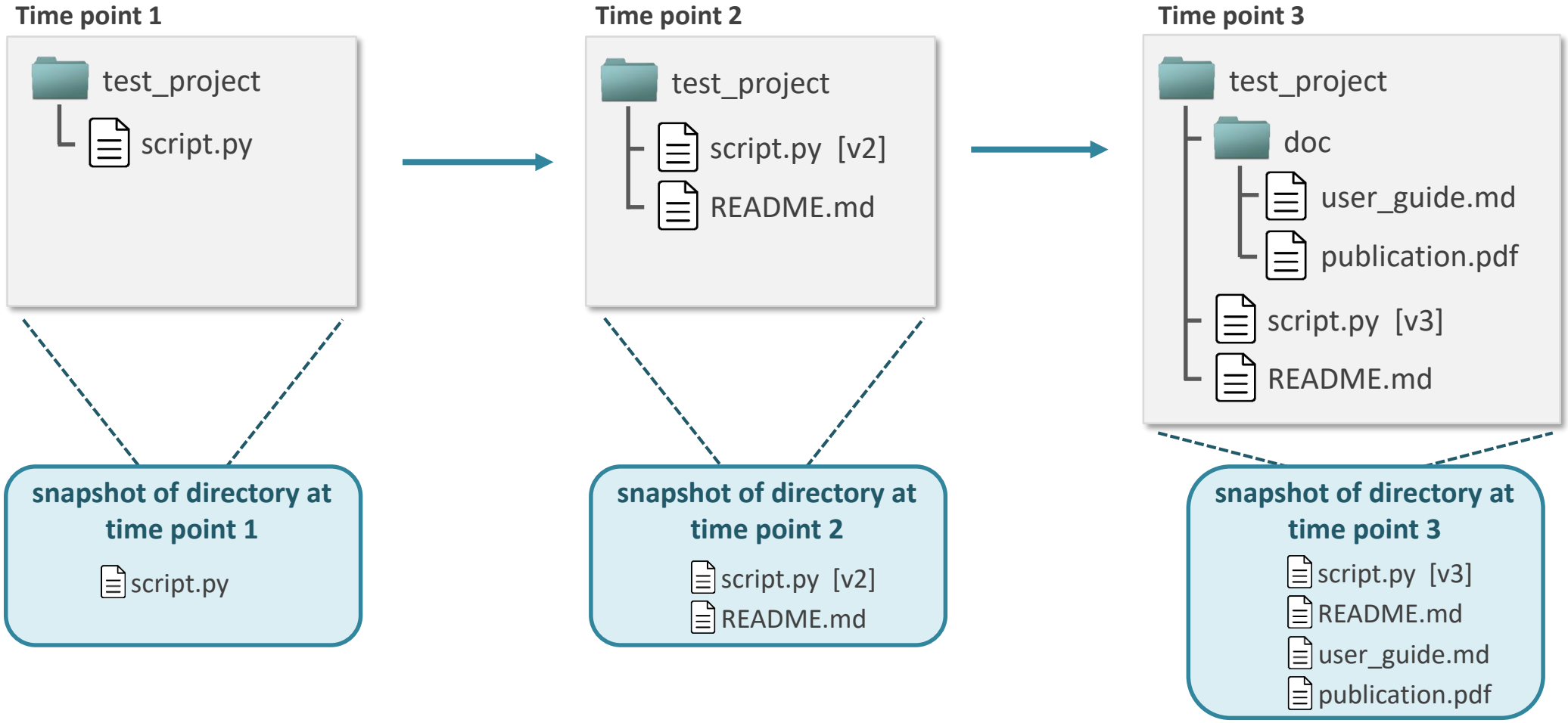
Git basics

Working principle and definitions

Basic principle of Git (and VCS in general)

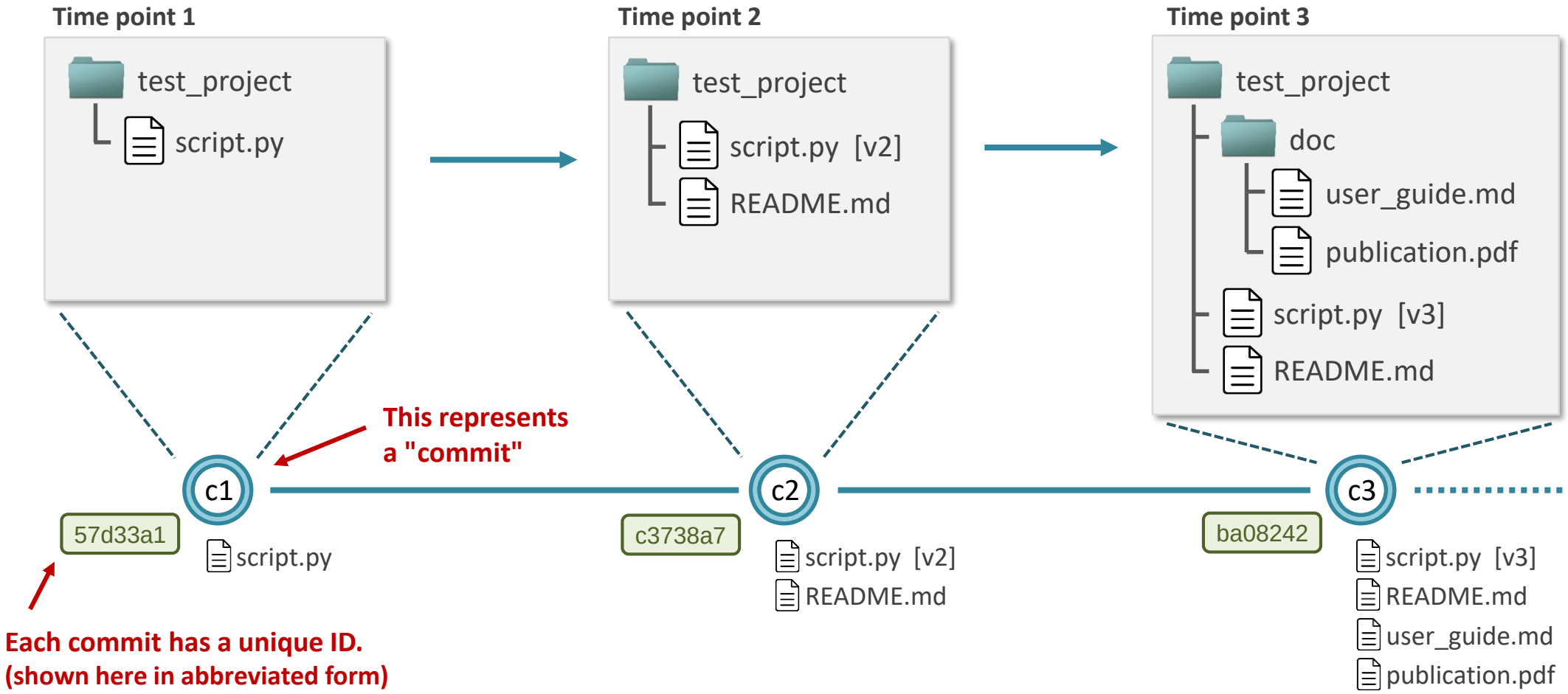
Our objective: **record the changes made to the content of a directory** on our **local machine**.
How we proceed:

- Take **snapshots** (current content of files) at ***user defined*** **time points – they are *not* taken automatically.**
- Keep track of **the order of snapshots** so their history can be recreated.
- Associate **metadata** with each snapshot: who made it, when, what does it container, …



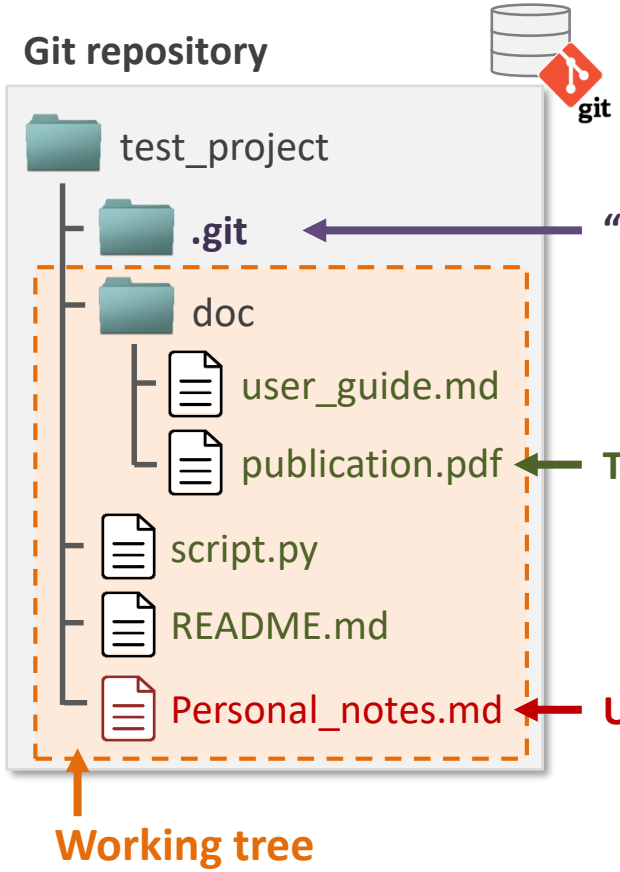
Definitions: snapshots are called “commits”

- **Commit = snapshot + metadata** (author, time, commit message, parent commit ID, etc. …).
- Create a new commit = record a new state of the directory’s content.
- Each commit has a unique **ID number** / **hash** (40 hexadecimal characters): commit ID `3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204`



Definitions: commits are stored in a repository (or “repo”)

- **Repository/repo**: a directory under Git control (a collection of commits).
  - Not all files in a directory under Git control have to be tracked.
  - There can be a mix of **tracked** and **untracked** files.
- **Working Tree**: current content (on your computer) of a Git repository.



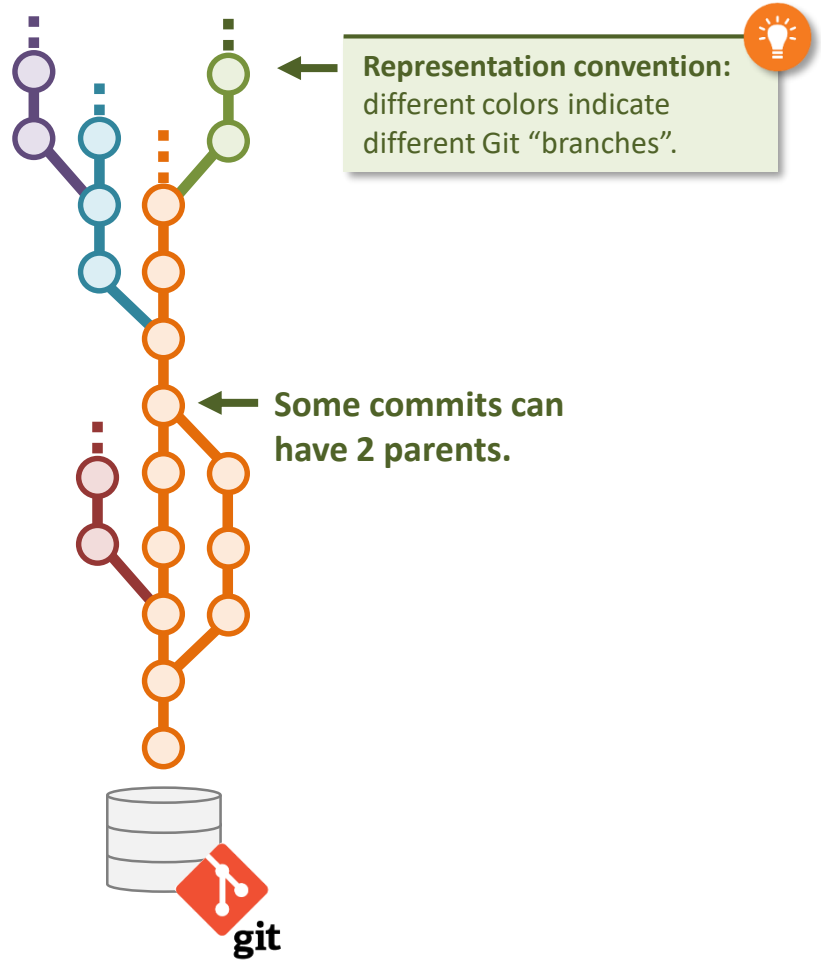
“database” of the Git repository.
- Contains the history of the repo and all other repo-related files.
- Each Git repo has its own, separate, “database”.

Tracked file: file under Git version control.

Untracked file: file present in the git repo directory, but *not* under version control.

Definitions: branches

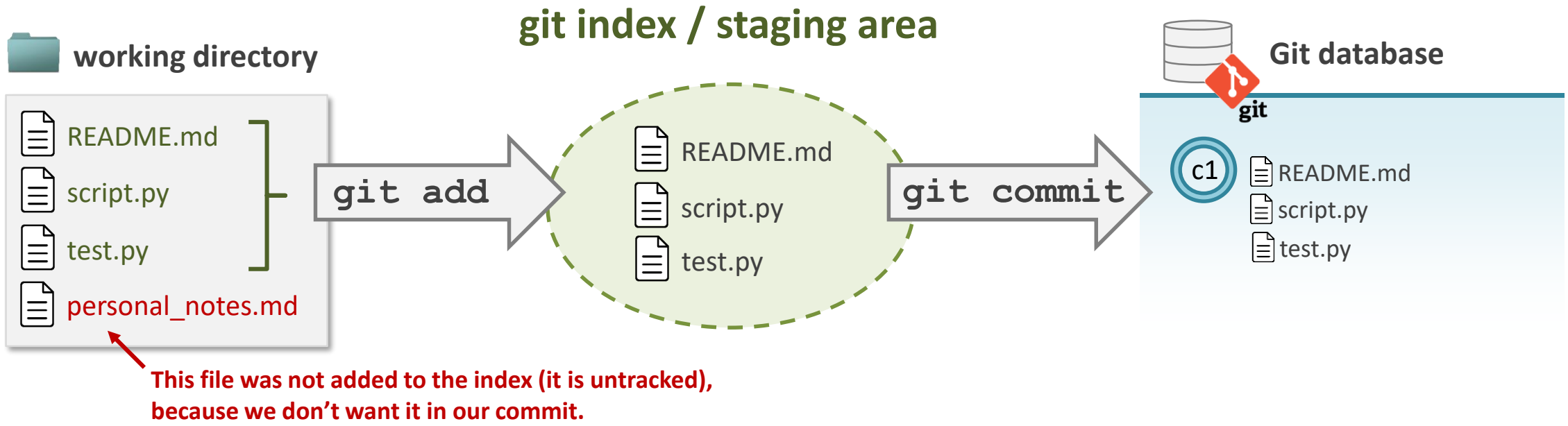
- **Repository history**: history of commits (chronology or commits).
- **Branch**: refers to a "**line of development**" within the commit history.
  (technically a branch is simply a reference to a commit)



Definitions: the Git index (or “staging area”)

In Git, committing content is a 2-step process:

1. ***Staging***: new content that should be part of the next commit must first be added to the **git index** (sometimes also called the **staging area**). This process is referred to as ***staging***.
2. ***Committing***: a new commit containing the content of the index is added to the repository (a new snapshot of the staged files is made).

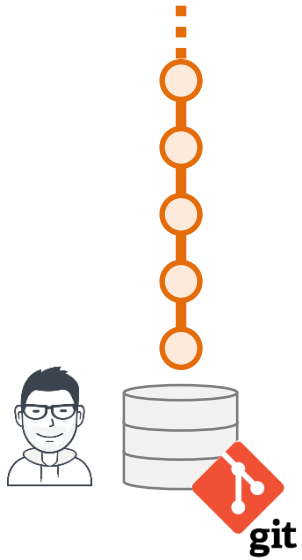


git index = content of your next commit.
commit = snapshot of the git index at a given time.

Examples of Git use cases

Exercise 1

Local repo, single branch

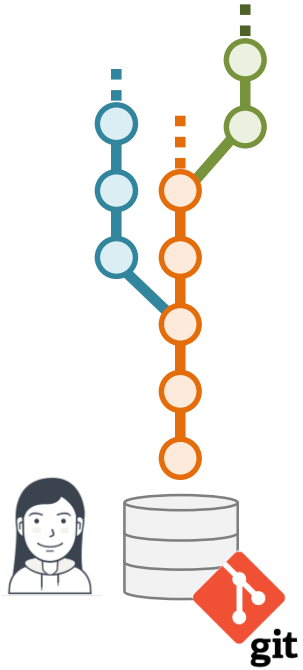


Use case

- Keep a documented log of your work.
- Go back to earlier versions.

Exercises 2 and 3

Local repo, branched workflow (multiple development lines)

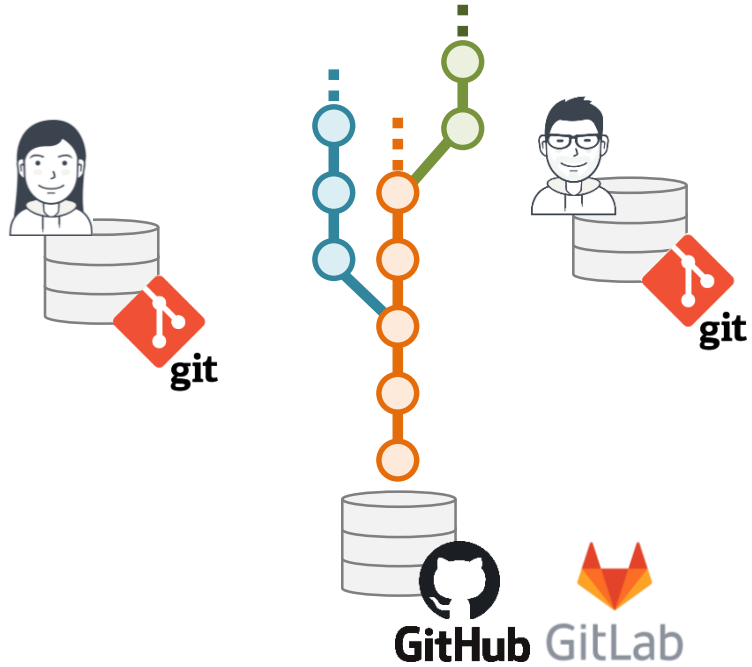


Use case

- Service in production with continuing development in parallel (e.g. new feature).

Exercise 4

Collaboration with distributed and central repos.



Use case

- Collaborate with others.
- Distributed development.

These two cases provide no backup !! only versioning.

Each user has a full copy of the data*.

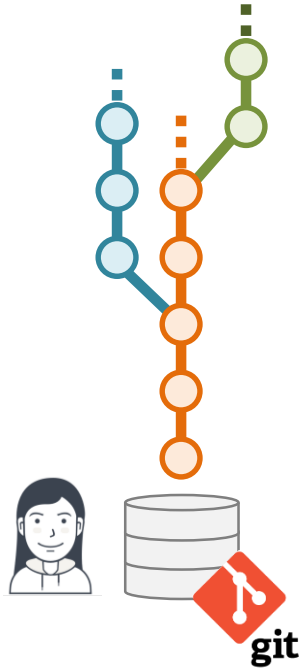
*** Provided they regularly sync their local repo.**

Local vs. Remote repository

- When creating a new Git repository on your computer, **everything is only local**.

- To get a copy of your repository online, you must take the active steps of:
  - Creating a new repository on a hosting service (e.g. GitHub, GitLab, Bitbucket, …).
  - Associate the online repository with your local repo.
  - Push your local content to the remote.

- By design, Git **does not automatically synchronize** a local and remote repo.
  Download/upload of data must be triggered by the user.



By default, everything stays local.

Supplementary material…

Git basics

your first commit

Configuring Git

- The minimum configuration is setting a **user name** and **email**. These will be used as default author for each commit.
- Setting user name and email:

```
git config --global user.name <user name>
git config --global user.email <email>
```

The `--global` option/flag tells Git to store the setting at the "global" (user wide) scope. Global settings apply to all Git repos on your machine.

If you don't add the `--global` option, then the setting will only apply to the current Git repo.

Global settings are stored in the following file:

- Linux: `/home/$USER/.gitconfig`
- Windows: `C:/Users/<user name>/.gitconfig`
- Mac OS: `/Users/<user name>/.gitconfig`

- Config values can be retrieved by using the `--get` option.
- Examples:

```
# Set user name and email at the global (user-wide) scope:
[alice@local ~]$ git config --global user.name "Alice"
[alice@local ~]$ git config --global user.email alice@redqueen.org

# Retrieve setting values:
[alice@local ~]$ git config --get user.name
 Alice
[alice@login1 ~]$ git config --get user.email
 alice@redqueen.org
```

Git config: changing the default text editor

- On most systems, the default editor that Git uses is "**vim**".
  However, this can be configured with the following `git config` command:

```
git config --global core.editor <editor cmd>
git config --global --get core.editor
```

- **Example**: changing the default editor to "nano" (another command line editor).

```
[alice@local ~]$ git config --global core.editor nano
[alice@local ~]$ git config --global --get core.editor
 nano
```

Supplementary material...

Git config: scopes and their config file locations

Depending on their scope, Git configurations apply to all Git repositories of a user, or only to a specific repository.

The main 3 scopes are:

- **Global (user wide):** settings apply to all Git repositories controlled by the user.
  - To save a setting as part of the global scope, add the `--global` flag to the `git config` command: `git config --global ...`
  - Stored in `/home/<user name>/.gitconfig` (Linux), `C:\Users\<user name>\.gitconfig` (Windows) or `/Users/<user name>/.gitconfig` (Mac OS).
- **Local (repo specific):** settings apply only to a specific Git repo.
  - Stored in the `.git/config` file of the repository.
- **System (system wide):** settings apply to all users and all repos on a given machine. This can only be modified by a system administrator.

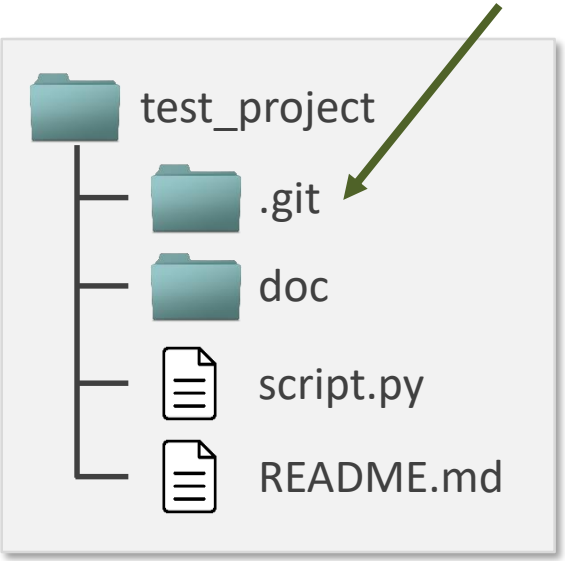
To show the list of all Git configurations, along with their scope and the location of the file they are stored-in:

```
git config --list --show-origin --show-scope
```

Supplementary material...

Creating a new Git repository

The Git database is stored in this "hidden" directory.



- Typing `git init` in any directory **initializes a Git database** in the directory, turning it into a **Git repository**.

- This creates a hidden `.git` directory - i.e. an empty Git database - at the root of the directory.

```
$ cd /home/alice/test_project
$ git init
Initialized empty Git repository in /home/alice/test_project/.git/
$ ls -a
./ ../ .git/ doc/ src/ README.md
```

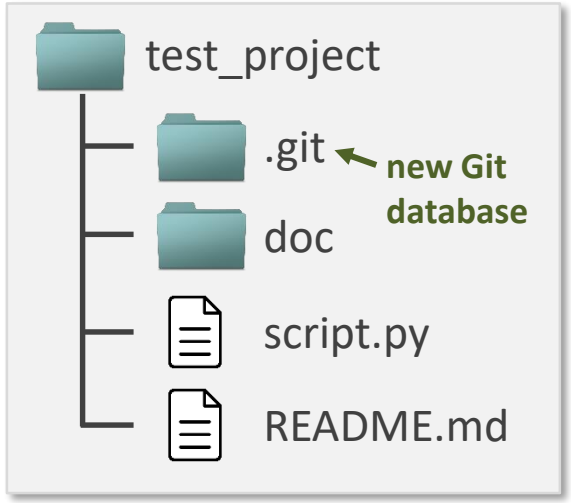
- **Everything** is stored in this single `.git` directory:
  - Content of all tracked files.
  - Complete versioning history.
  - All other data associated to the Git repository (e.g. branches, tags).

Never delete the `.git` directory

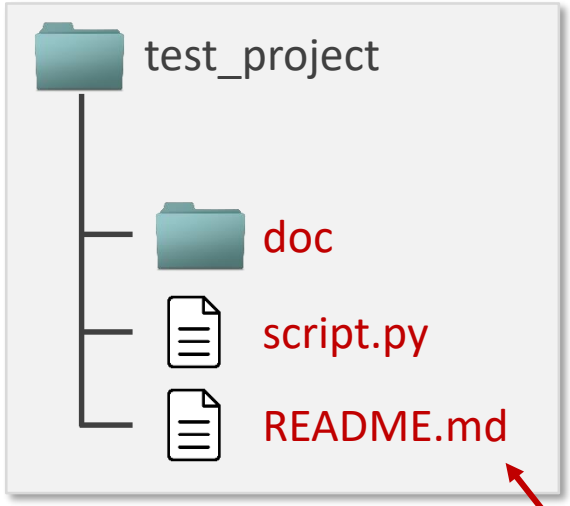
- The content of the `.git` database can re-create the exact state of all your files at any versioned time - e.g. if you delete a file accidentally or want to go back to an earlier version.

State of the working directory just after `git init`

How it look on your file system



How Git sees it



List of files tracked by Git

```
$ git ls-files
<empty output>
```

Commit history

```
$ git log
fatal: your current branch
'main' does not have any
commits yet
```

```
$ git status
On branch main
No commits yet
Untracked files:
      doc/
      README.md
      script.py
```

"main" is the default branch name.

red = untracked file

`git status`

Show status of files in project directory.

`git ls-files`

List files tracked in the Git repo.

`git log`

Show log of commits (i.e. history of repo).

Summary: when creating new Git repo…

- It does not matter whether the directory is empty or already contains files/sub-directories.
- **Files** in your Git repo (project directory) **are not automatically tracked** by Git. They must be manually added.
- Only files located in the Git repo (or one of its sub-directories) can be tracked.
- You can have both tracked and untracked files in a project directory.
- You can (should) have multiple Git repositories on your system – typically one per project or per code/script you develop - don't use a single Git repo to track the entire content of your computer!
- Git repos are self-contained – you can rename them or move them around on your file system.
- The ensemble of all files that are under Git control in a given git repository is generally referred to as the repository's **working tree**.

Never delete the `.git` directory, you would lose the entire versioning history of your repository (along with all files not currently present in the working tree).

“Bare” Git repositories

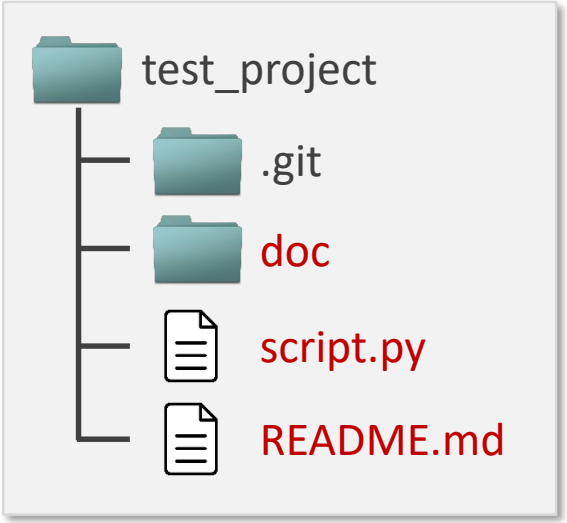
A **bare repo** is a repo that has **no working tree**: it does not contain any instance of the files that are under Git version control, but only the content of the `.git` directory/database.

This type of repo is found on remote servers used to share and sync changes across multiple Git repositories. They can be initialized with the command: `git init --bare`

Supplementary material...

Adding content to a Git repository (staging files)

- By default, files in a directory under Git control are **untracked**.
- To add a new file – or a change in file content – to the Git repository, the file **must be explicitly added** with the `git add` command. This is often referred to as ***staging* a file** (or file content).
- This allows to **separate important files** of your project - that you want to be tracked by Git - **from unimportant ones** that should not be tracked or shared (e.g. a test file of your own).
- After a file has been added once, it is considered as **tracked** by Git (unless you manually remove it).
- **Each time a file is modified, it must be added/staged again** for the new content to get added to the next commit.

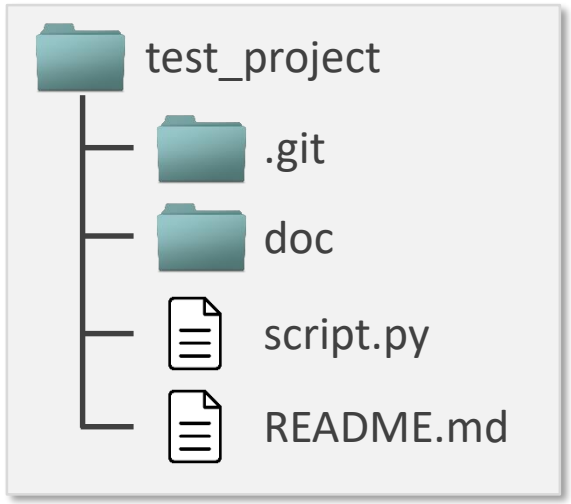


Only files/directories located inside the project's directory can be tracked.

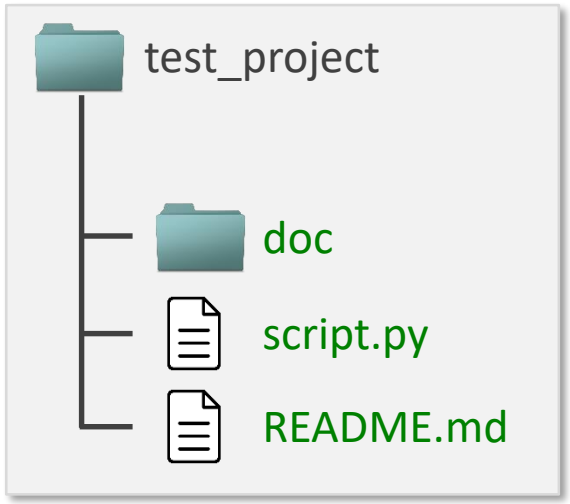
State of the project directory after content is added with `git add`

```
$ git add script.py README.md doc
```

How it look on your file system



How Git sees it



List of files tracked by Git

```
$ git ls-files
README.md
script.py
doc/quick_start.md
```

Commit history

```
$ git log
fatal: your current branch
'main' does not have any
commits yet
```

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

Changes to be committed:
    new file: README.md
    new file: script.py
    new file: doc/quick_start.md
```

green = new or modified file

Files/changes are added, but not committed yet.

Committing content

```
git commit -m/--message "your commit message"
git commit
```

If no commit message is given, Git will open its default editor and ask you to enter it interactively.

Example

```
$ git commit -m "Initial commit for test_project"
[main (root-commit) 8190787] Initial commit for test_project
3 files changed, 6 insertions(+)
create mode 100644 README.md
create mode 100644 script.py
create mode 100644 doc/quick_start.md
```

6 insertions = 6 lines added in total (across all files).

README.md

```
# Quick-start guide for the test_project software
```

+ 1

script.py

```
#! /usr/bin/env python3
```

+ 1

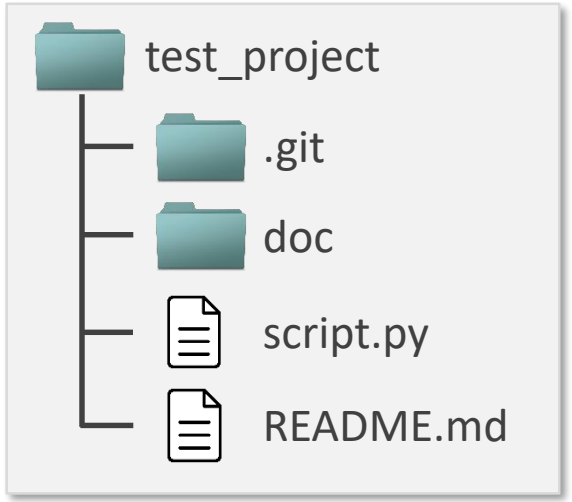
doc/quick_start.md

```
# Test project: a project to test version control with git

This is a small test project to illustrate the use of git.
Maybe I will add more content to it later.
```

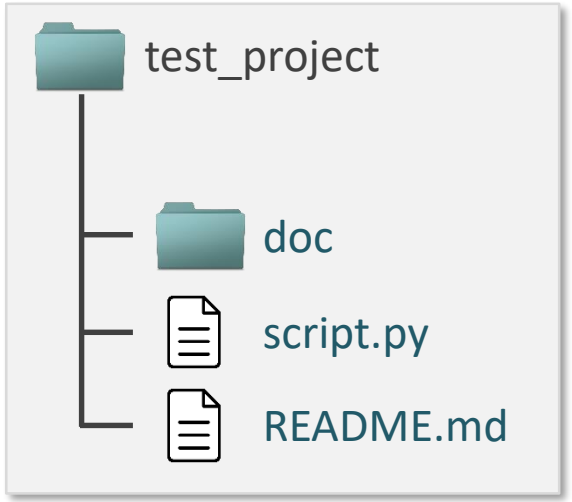
+ 4 (empty lines also count)

State of the project directory after `git commit`

How it look on your file system



How git sees it




```
$ git status
On branch main
Nothing to commit, working
tree clean
```

Clean working tree = current state of working tree matches exactly with the latest commit.

List of files tracked by Git

```
$ git ls-files
README.md
script.py
doc/quick_start.md
```

Commit history

```
$ git log
commit 8190787daa6fca93f5f25b819716d50c31bf5c26
Author: Alice <alice@redqueen.org>
Date:   Sun Feb 9 15:07:56 2020 +0100

    Initial commit for test_project
```

Now `git log` has finally something to display (just 1 commit, for now).

Committing content: interactive commit message with the “vim” editor

```
$ git commit
```

When no commit message is specified, Git automatically opens a text editor. By default, this editor is “vim”.

```
Initial commit for test_project
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       new file:   README.md
#       new file:   script.py
#       new file:   doc/quick_start.md
#
```

- In the “vim” editor, press on the key “**i**” to enter edit mode
- In edit mode, you can now type your commit message.

Supplementary material...

Committing content: interactive commit message with the “vim” editor

```
Initial commit for test_project

This is the very first commit in this Git repo.
Way to go!

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch main
# Changes to be committed:
#       modified:   README.md
#       new file:   script.py
#       new file:   doc/quick_start.md
#
~
~
:wq
```

- Commit message can be entered over multiple lines.
- By convention, try to keep lines reasonably short (<= 80 chars)

- Press “**Esc**” to exit “edit” mode.
- Type “**:wq**” in the vim “command” mode.

Press “**Enter**” to exit vim and save your commit message.

```
[main (root-commit) 8190787] Initial commit for test_project
3 files changed, 6 insertions(+)
create mode 100644 README.md
create mode 100644 script.py
create mode 100644 doc/quick_start.md
```

- You are now back in the shell and your commit is done.

Supplementary material...

Live demo

- Initializing a new Git repo.
- Adding content to the Git repo.
- Making a commit with interactive commit message.

Making commits: some basic advice.

Git does not impose any restrictions on what and when things can be committed.
(the only exception being that you cannot commit zero changes)

However, it's best if you:

- Make **commits at meaningful points of** your code/script **development**.
  For instance:
  - A new function/feature was added (or a few related functions).
  - A bug was fixed.

- **Make multiple small commits** instead of a large one if you are making changes that affect different functionalities of your code (this can make it easier to e.g. revert changes).

- **Don't commit broken code on your *main/master* branch** (i.e. the branch that others might use to get the latest version of your code). If you have partial work, you can commit it to a *temporary/feature* branch, and later merge it into *main*/*master* (more on branch management will follow later).

Ignoring files

- By default, files that are not added to a Git repo are considered **untracked**, and are always listed as such by `git status` .
- To stop Git from listing files as **untracked**, they can be added to one of the following "ignore" files:

`.gitignore`

- For files to be **ignored by every copy of the repository**.
- `.gitignore` is meant to be tracked: `git add .gitignore`
- Examples:
  - outputs of tests
  - .Rhistory, .RData
  - .pyc (compiled version of python code)

Most of the time, this is the method you will want to use to ignore files.

`.git/info/exclude`

- For files that should be **ignored only by your own local copy of the repository**.
- Not versioned and not shared.
- Examples:
  - Files with some personal notes.
  - Files specific to your development environment (IDE).

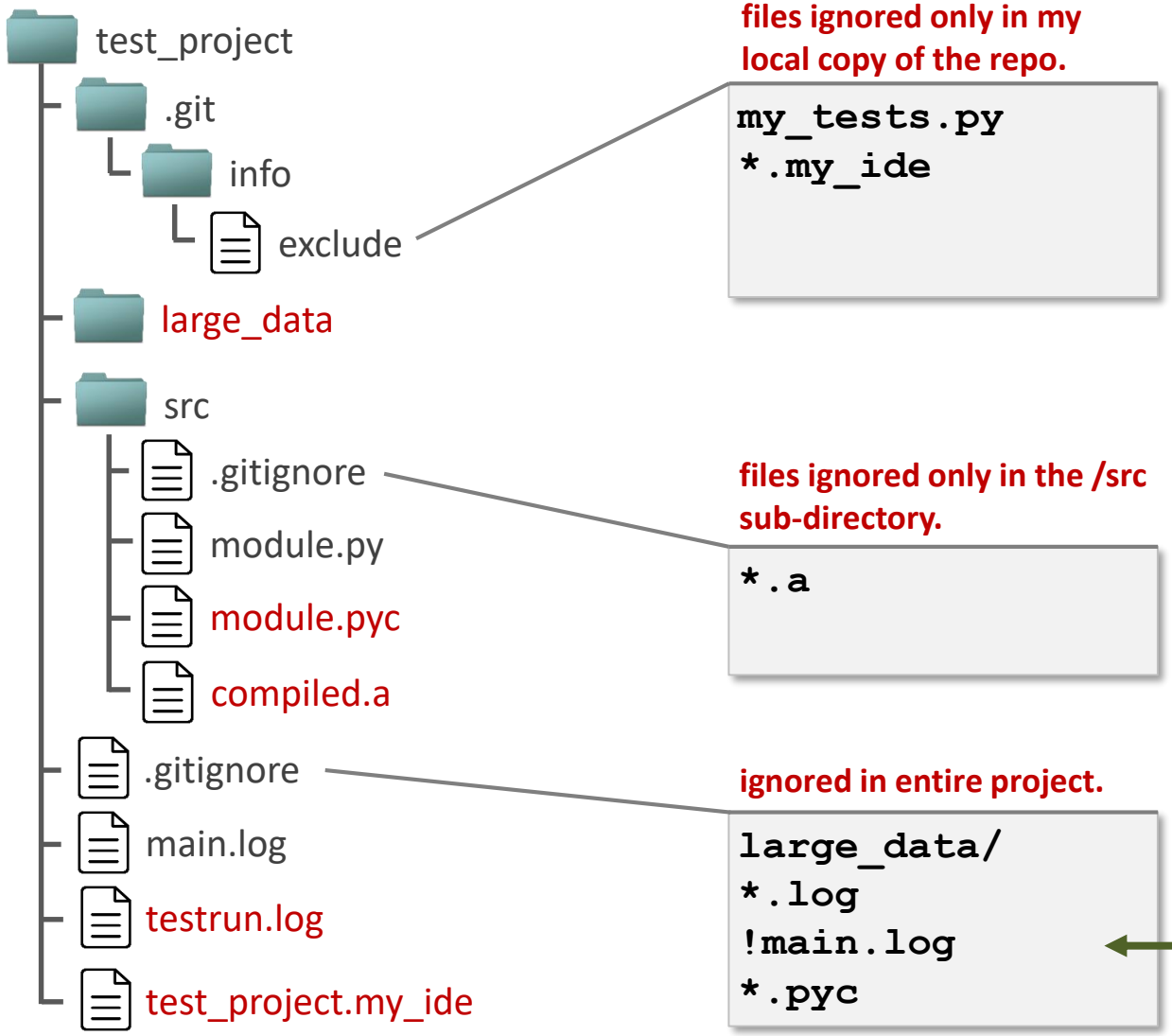
Use this method for **special cases** where a file should **only be ignored in your local copy of the repo.**

Example of a `.gitignore` file

```
my_tests.py
.Rdata
.Rhistory
*.pyc
test_outputs/
```

- Files to ignore are added by manually editing the two above-mentioned files.
- Files can be ignored based on their full name, or based on glob patterns (see next slide for examples).
  - `*.txt` ignore all files ending in ".txt"
  - `*.[oa]` ignore all files ending either in ".o" or ".a"
  - `logs/` appending a slash indicates a directory. The entire directory and all of its content are ignored.
  - `!dontignorethis.txt` adding a **!** In front of a file name means it should not be ignored (exception to rule).

Ignoring files: example



- There can be multiple `.gitignore` files per project, to create custom per-directory ignore rules.
- Ignore rules in sub-directories are inherited from the `.gitignore` of their parent directory(ies).
- The `.gitignore` files themselves should not be ignored: add them to the Git repo so they are tracked.

- **Order (sometimes) matters**: here the rule to not ignore main.log must be placed after the general rule to ignore *.log files.

red = ignored file.

Live demo

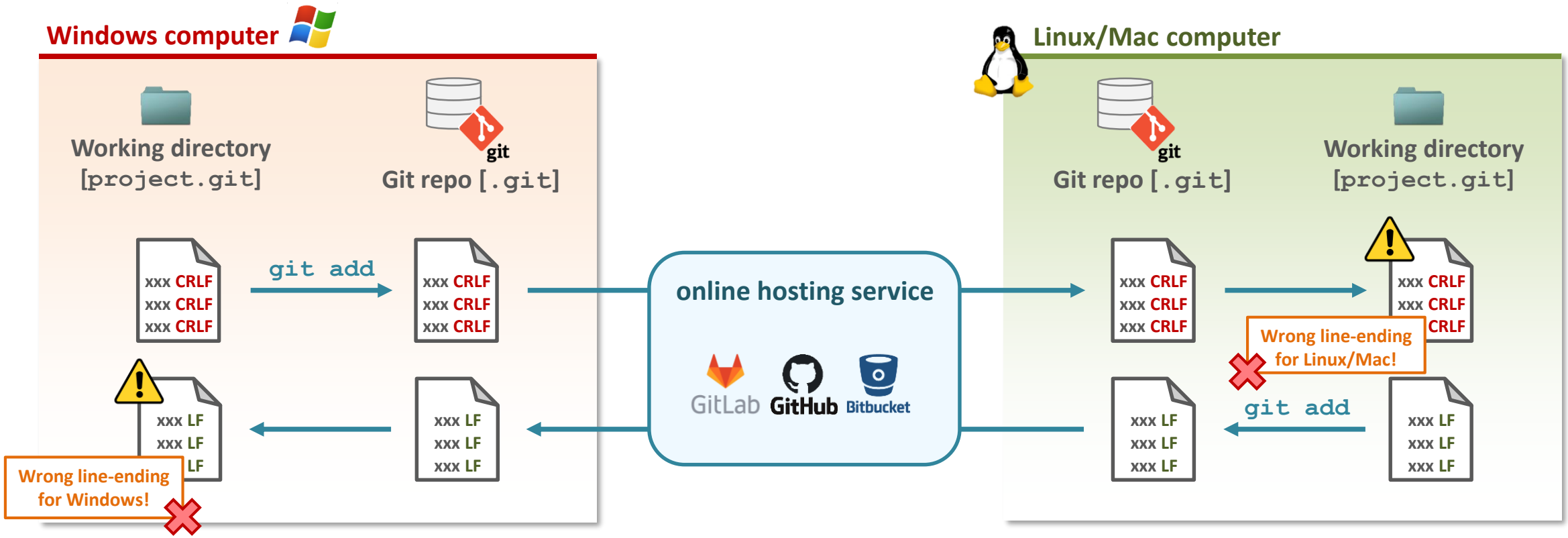
- Adding files to `.gitignore`

Cross-platform collaboration: the line-end problem

Linux/Mac and Windows do not use the same “line-end” characters: this can cause problems when collaborating with people who use a different operating system.

- **Linux/Mac:** uses **LF** (linefeed; \n) as line-ending character.
- **Windows:** uses **CRLF** (carriage-return + linefeed; \r\n) as line-ending character.

→ **Text files created on Windows will not work well on Linux/Mac and vice versa.**



Supplementary material...

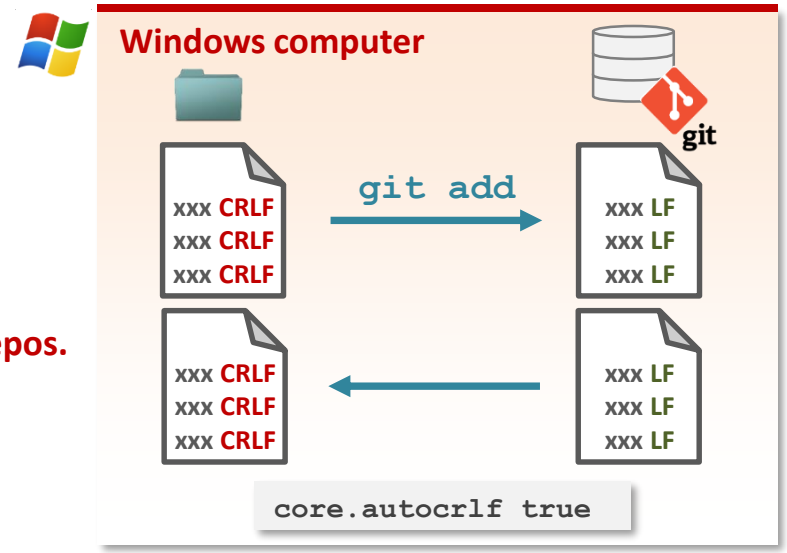
Cross-platform collaboration: solution -> setting `git config core.autocrlf`

The solution is to ask Git to **automatically convert between LF and CRLF** during add/checkout operations.

- On **Windows** computers: `core.autocrlf true` should be set so that LF are automatically changed to CRLF each time a file is checked-in or checked-out.

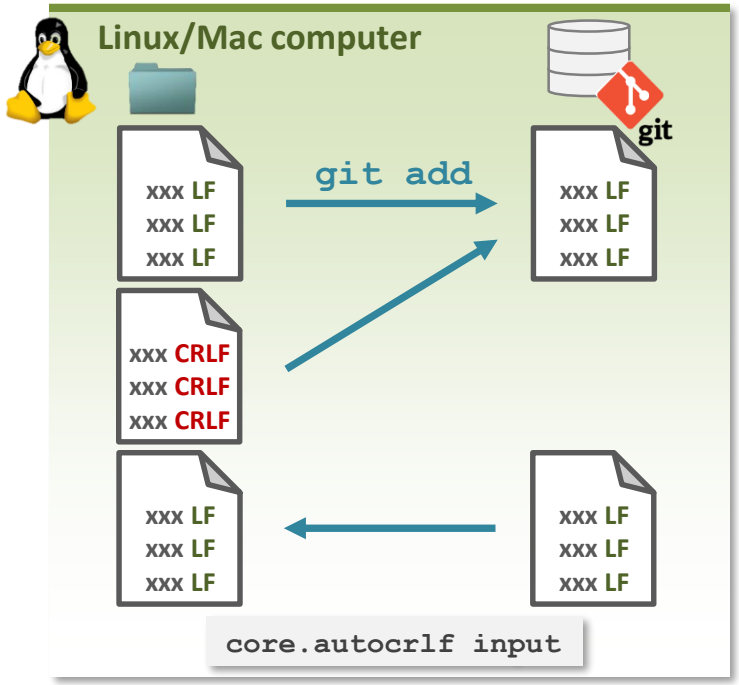
```
git config core.autocrlf true
git config --global core.autocrlf true
```

← Change setting for current repo.
← `--global` = change setting for all repos.



- On **Linux/Mac** computers: `core.autocrlf input` should be set so that LF line-endings (LF) are left untouched, and that CRLF are converted to LF when a file is added (this will only be useful in the rare cases when a file with CRLF ending is somehow present on the machine, e.g. because it was sent via email by a Windows user).

```
git config core.autocrlf input
git config --global core.autocrlf input
```



- `core.autocrlf false` to disable LF/CRLF auto-modifications (this is the default):

```
git config core.autocrlf false
git config --global core.autocrlf false
```

Supplementary material...

`core.autocrlf` warnings

When `core.autocrlf` is set to `True` (this is in principle only for windows users), a warning is displayed when files are added/checked-out to/from the git repo:

```
$ git add test_file.py
warning: LF will be replaced by CRLF in test_file.py
The file will have its original line endings in your working directory
```

Somehow the message is the same during adding and check-out of files… so when adding files to the index (`git add`), the message is actually the wrong way round: it should be something like “CRLF will be changed to LF in checked-in file”.

Supplementary material…

Displaying a repository's state and history

git status, **git show** and **git log**

git status

- Display the status of files in the working directory.

```
git status
```

- **Tracked files that have not been modified** since the last commit are not listed (but they are still in the index and will be part of the next commit).
- **Ignored files** are also not listed.

Green = new content in this file has been staged and will be part of the next commit.

Red = this file contains new* content, but it is not staged. The new content will not be part of the next commit.

* new = which differs from the latest commit.

```
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   LICENSE.txt
        modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   README.md

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

        untracked_file.txt
```

tracked files.

untracked files.

Note: new content in a file can be **partially committed**: i.e. it's possible to have some changes in the file staged (added to the index), and some unstaged…

This is the case in the example above for the `README.md` file. **Only the staged content will become part of the next commit**.

File status in Git

There are 4 possible statuses for files in Git:

- **Tracked**
  File that is currently under version control by Git (i.e. it is in the Git index).
  - **unmodified** - the content of the file is the same as in latest commit.
    more precisely: the content is the same as in the commit to which HEAD is currently pointing.
  - **modified** - the content of the file differs from the latest commit.
    more precisely: it differs from the commit to which HEAD is currently pointing.
- **Untracked**
  File that is in the working directory, but not under version control by Git.
- **Ignored**
  Untracked file, but where Git is aware it should not be tracked.

Supplementary material...

git show

- Display the change in file content introduced by a commit.

```
git show <commit reference>
```

```
git show
```

→ **with no argument, the latest commit on the current branch is shown (i.e. HEAD).**

Examples of commit references:

- A commit ID (hash): **89d201f**
- A branch name: **develop**
- A tag name: **1.0.7**
- The **HEAD** pointer.
- A relative reference: **HEAD~3**

If no commit reference is given, **HEAD** is used as default.

Example:

```
$ git show 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100

    Add stripe color option to class Cheshire_cat

diff --git a/script.sh b/script.sh
index d7bfdc8..fa99250 100755
--- a/script.sh
+++ b/script.sh
@@ -7,13 +7,28 @@

# def Cheshire_cat():
-   def __init__(self, name, owner="red queen"):
+   def __init__(self, name, owner="red queen", stripe_color="orange"):
+       self.stripe_color = stripe_color
```

```
git show --name-only <ref>
```

Only display file names (without the changes)

```
$ git show --name-only 89d201f
commit 89d201fd01ead6a499a146bc6da5aa078c921ecf
Author: Alice <alice@redqueen.org>
Date:   Wed Feb 19 14:00:02 2020 +0100

    Add stripe color option to Cheshire_cat

script.sh
```

git log: display the commit history of a Git repo

```
git log
git log --oneline
git log --all --decorate --oneline --graph
```

`git log` has many options to format its output.

See `git log --help`

Example: default view (detailed commits of current branch).

```
$ git log
commit f6ceaac2cc74bd8c152e11b9c12ada725e06c8b9 (HEAD -> main, origin/main)
Author: Alice alice@redqueen.org
Date:   Wed Feb 19 14:13:30 2020 +0100

    Add stripe color option to class Cheshire_cat

commit f3d8e2280010525ba29b0df63de8b7c2cd7daeaf
Author: Alice alice@redqueen.org
Date:   Wed Feb 19 14:11:56 2020 +0100

    Fix off_with_their_heads() so it now passes tests

commit cfd30ce6e362bb4536f9d94ef0320f9bf8f81e69
Author: Mad Hatter mad.hatter@wonder.net
Date:   Wed Feb 19 13:31:32 2020 +0100

    Add .gitignore file to ignore script output
```

Example: compact view of current branch

```
$ git log --oneline
f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
f3d8e22 peak_sorter: display name of highest peak when script completes
cfd30ce Add gitignore file to ignore script output
f8231ce Add README file to project
821bcf5 peak_sorter: add +x permission
40d5ad5 Add input table of peaks above 4000m in the Alps
a3e9ea6 peak_sorter: add first version of peak sorter script
```

Example: compact view of entire repo (all branches)

```
$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: add authors as comment to script
* 6c0d087 peak_sorter: improve code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> main, origin/main) peak_sorter: add authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy, dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```

Adding custom shortcuts to Git

Some git commands can be long and painful to type, especially when you need them often!
To shorten a command, you can create **custom aliases**:

```
git config --global alias.<name of your alias> "command to associate to alias"
```

Example:

```
git config --global alias.adog "log --all --decorate --oneline --graph"
```

With the alias set, you can now simply type:

```
git adog
```

exercise 1

Your first commit

This exercise has helper slides

Exercise 1 help: bash (shell) commands you may need during this course

`cd <directory>`	Change into directory (enter directory).
`cd ..`	Change to parent directory.
`ls -l`	List content of current directory.
`ls -la`	List content of current directory including hidden files.
`pwd`	Print current working directory.
`cp <file> <dest dir>`	Copy a file to directory “dest dir”.
`mv <file> <new name>`	Rename a file to <new name>.
`mv <file> <directory>`	Move a file to a different directory.
`cat <file>`	Print a file to the terminal.
`less <file>`	Show the content of a file (type “q” to exit).
`vim <file>`	Open a file with the “vim” text editor.
`nano <file>`	Open a file with the “nano” text editor.

Reminder...

Git concepts

commits, the **git index**, and the **HEAD** pointer

Git commits

Git’s immutable, atomic, units of change

Introducing SHA-1

- SHA-1 stands for **S**ecure **H**ashing **A**lgorithm 1.
- This algorithm turns any binary input into an (almost*) unique 40 character hexadecimal **hash/checksum value** (hexadecimal = base 16 number, 0-9 + a-f).

  `e83c5163316f89bfbde7d9ab23ca2e25604af290`

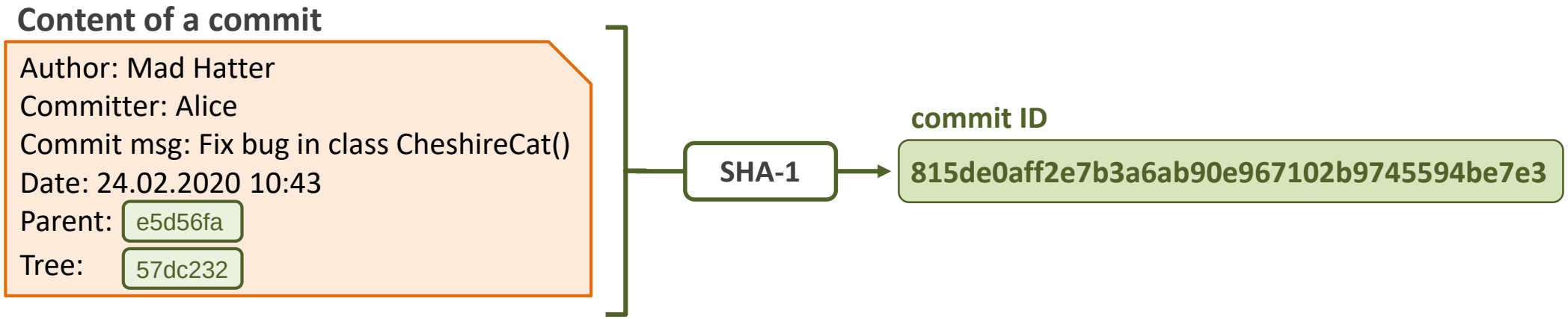
- Important: for a given input, SHA-1 always computes the exact same and (almost*) unique hash.
- Example: running "This is a test" through the SHA-1 algorithm, will always produce the hash shown below:

```
echo "This is a test" | openssl sha1   →   3c1bb0cd5d67dddc02fae50bf56d3a3a4cbc7204

echo "This is a Test" | openssl sha1   →   7500c6645cb9cdb20b32002cb82bbe067cc77d6e
```

* With current hardware, SHA-1 collisions can be reasonably easily created. SHA-1 is no longer considered secure for cryptographic purposes, but is good enough for usage in Git. It is also fast to compute.

Commits: Git's atomic, immutable, units of change

- A commit is the **smallest unit of change** in a Git repository.
- A commit is **the only way to enter a change** into a Git repository.
  (enforces accountability as you cannot have untraceable modifications)
- Each commit has an associated author, committer, commit message and date.
  (enforces documentation)



- Commits are lightweight: they **do *not* contain the tracked files' data**, only a **reference to the data**
  (specifically, a **Tree*** object that represents the content of the Git index at the time the commit was made).
- Commits contain a **reference to their parent** commit.
- Each commit is uniquely identified by a **commit ID**: a SHA-1 hash/checksum computed on all its metadata.

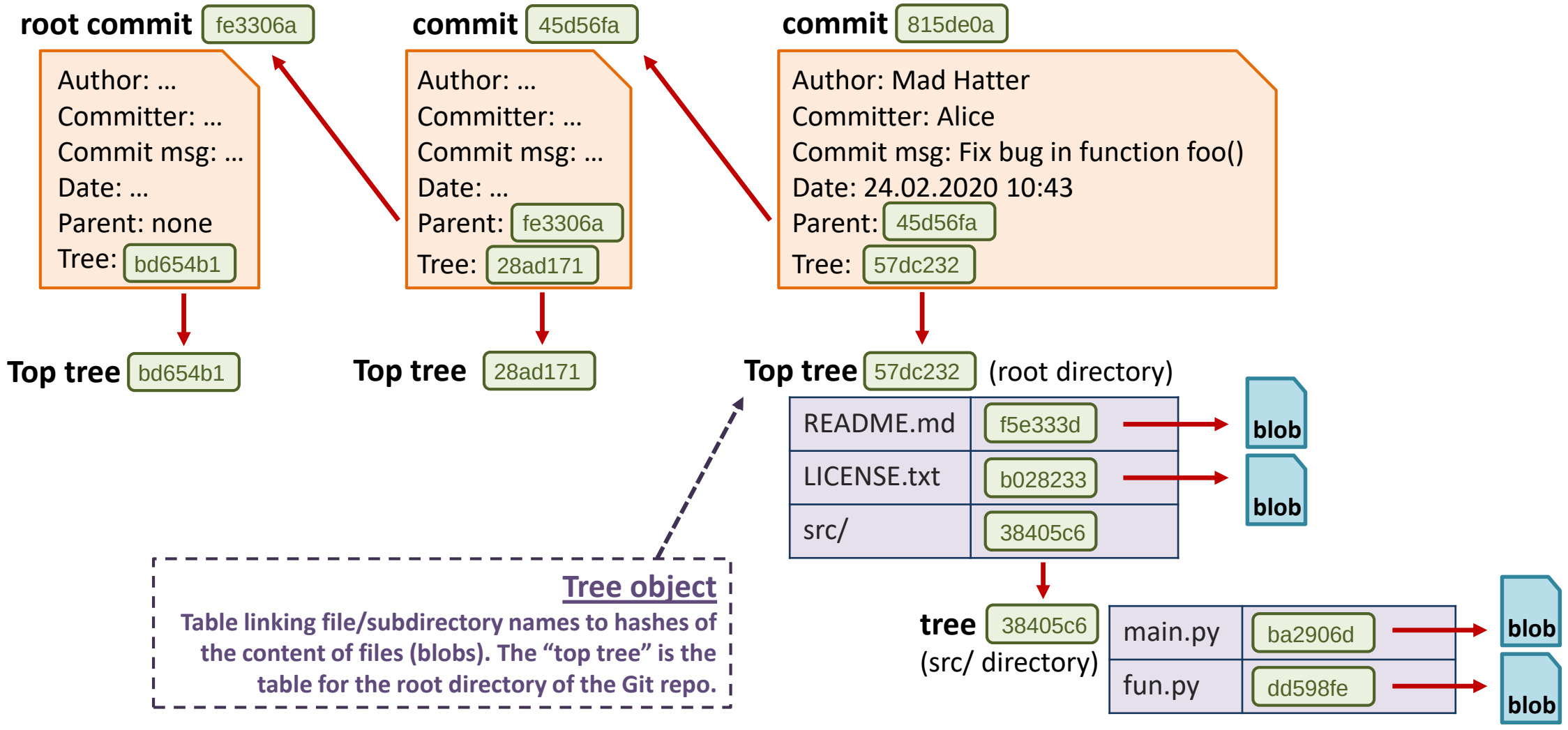
* Tree = reference to the content of all files at a given time point.

- **Commits contain a reference to the top "Tree object"** – a table linking file names and hashes of the Git index at the time the commit was made. This is how Git can retrieve the state of every file at a given commit.

If two commits have the same ID, their content is identical !

- **Commits point to their direct parent** – forming a DAG (directed acyclic graph) where no commit can be modified without altering all of its descendants.

If two commits have the same ID, their entire history is identical !



Supplementary material…

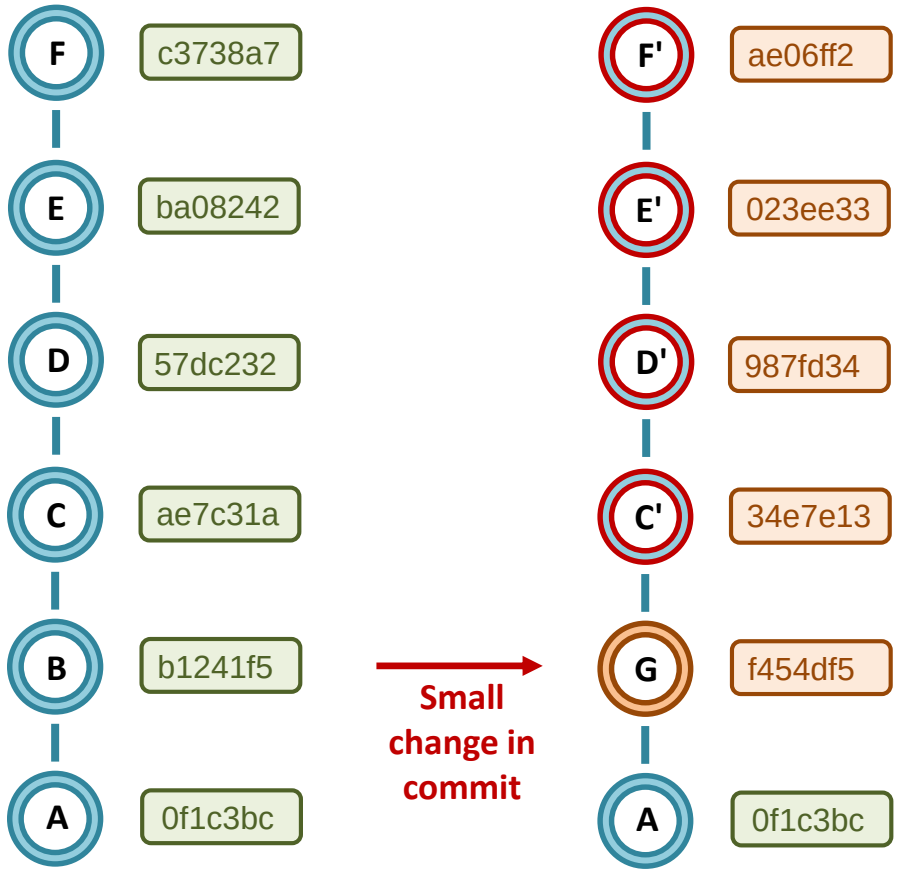
- Because of how their commit ID (SHA-1 hash) is computed, **commits are immutable**: once a commit is made, it cannot be modified without its commit ID being modified too - which would then make it a different commit !

- **Modifying a commit** will modify all of its descendants. It **creates a completely new history** of the Git repo.

- This ensures the **integrity of a Git repository's history**, something that **is important due to the distributed nature of Git**. It can be seen as a sort of blockchain.

Examples of things that change a commit's ID:

- **Changing the content** of a file.
- **Changing the time** a commit was made.
- **Changing the parent commit** of a commit.



Git versioning

- Git stores a complete copy of each file's version*.
- Optimized for speed rather than disk space preservation.
- Sub-optimal for tracking large files, as they will quickly inflate the size of the `.git` repo.

As counter-intuitive as it may sound, Git stores a complete copy of each file version. Not just a diff.

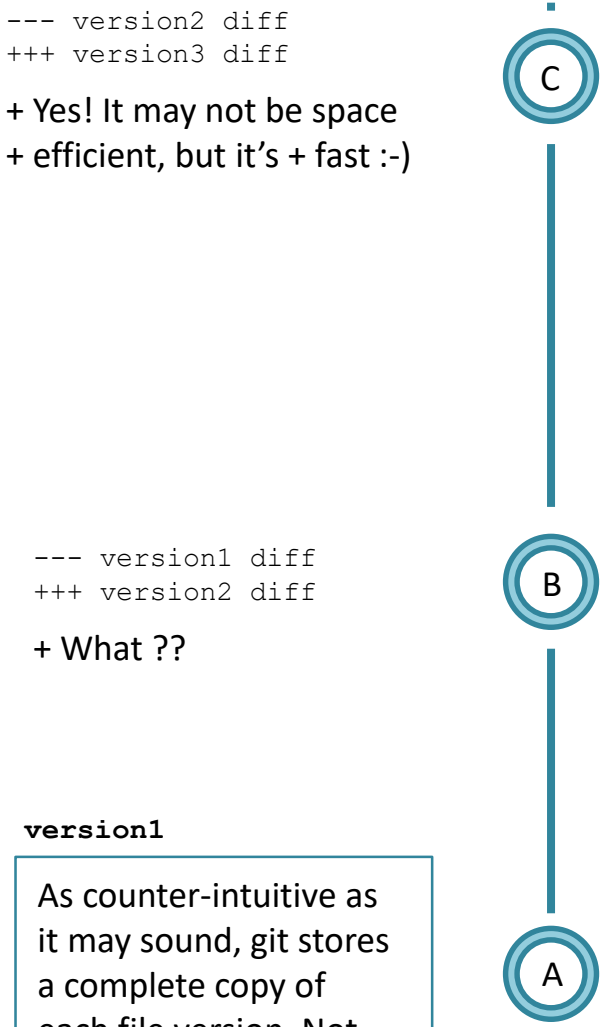
What ??

Yes! It may not be space efficient, but it's fast :-)

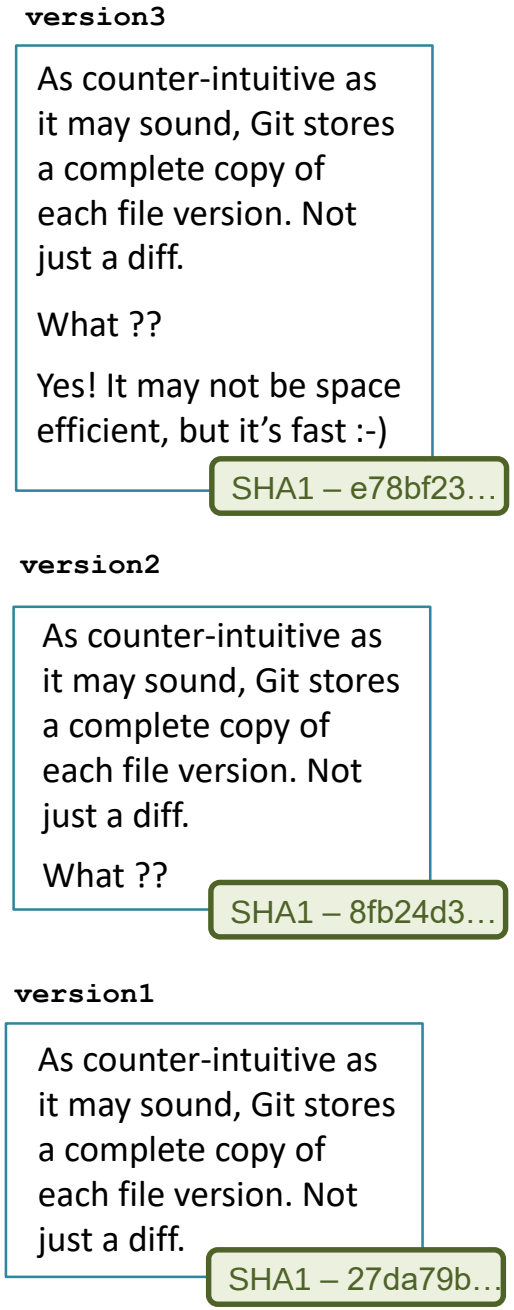
* At least for a while - at some point Git also stores things as diffs, see "packfiles".

Supplementary material...

most VCS versioning



Git versioning



Git packfiles: compressing old history

- For older commits, Git uses a few tricks to decrease disk space usage:
  - Differences between similar files are stored as diffs.
  - Multiple files are compressed into a single “packfile” (`.pack` extension).
  - Each packfile has an associated packfile index (`.idx` extension), that associates filenames to blobs.

Supplementary material...

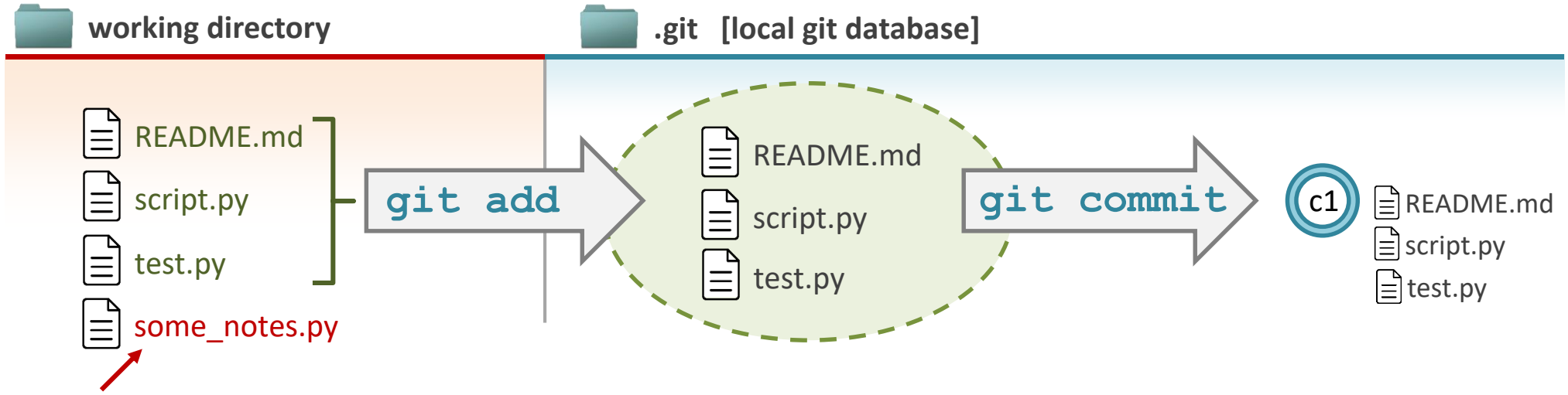
the Git index

(staging area)

A preview of your next commit

Committing new content: a two-step process…

- Any new content to be committed must first be added to the **git index**, or **staging area**. This process is referred to as ***staging***. The objective of this 2-step procedure is to help create "well defined" commits
- **New commit = snapshot of the Git index**. The Git index can be thought of as a sort of "virtual stage" where the content of the next commit is prepared.
- **Staged files remain staged**, unless removed or overwritten by a newer version.
- When files are added to the git index (staged), their content is already copied to the git database.



This file is not added (untracked), because we don't want it in our commit.

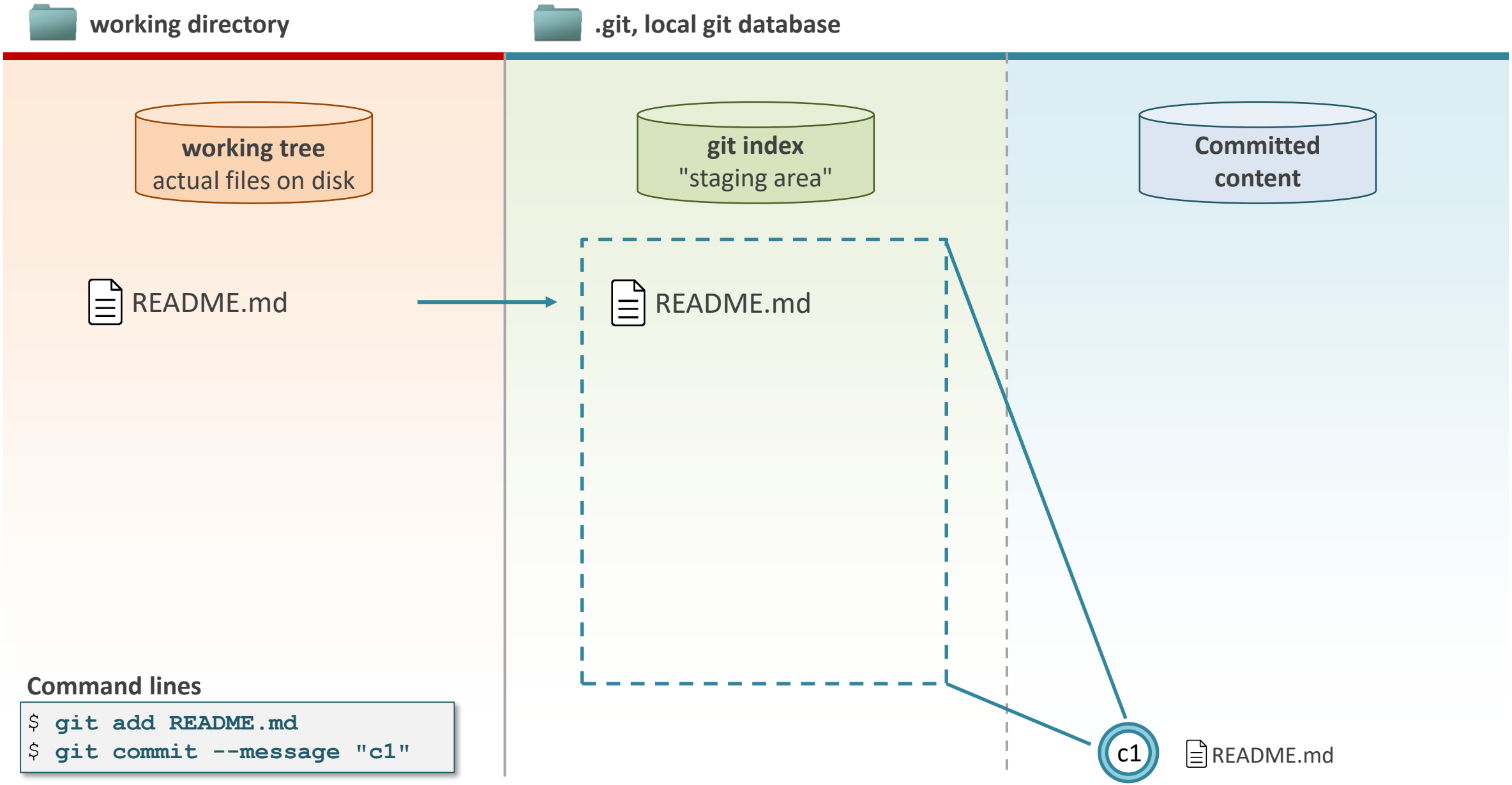
git index = content of your next commit.
commit = snapshot of the git index at a given time.

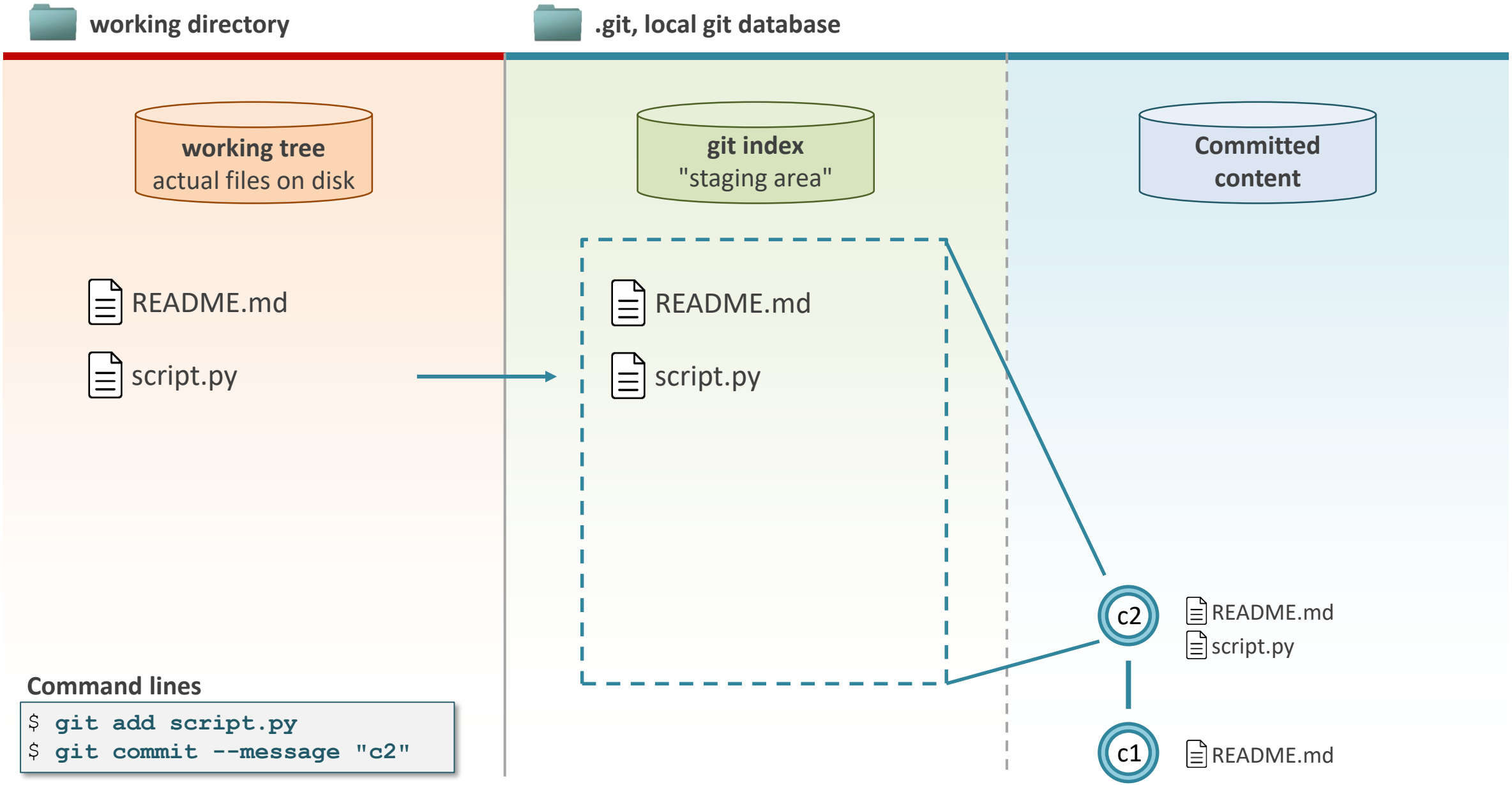
Committing new content: a two-step process…

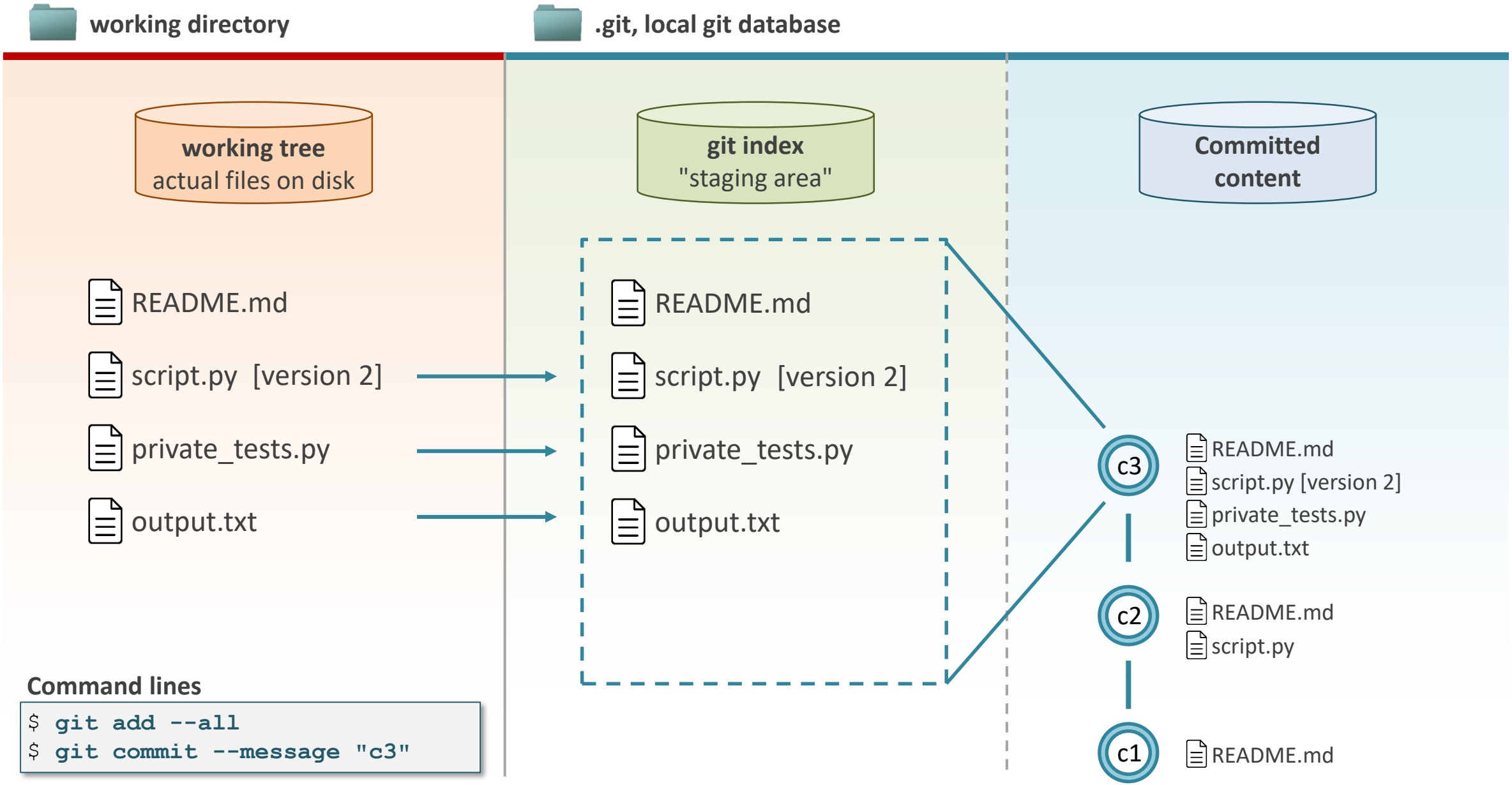
- Why do we need the git index ?
- Why not simply commit the entire content of the directory directly ?

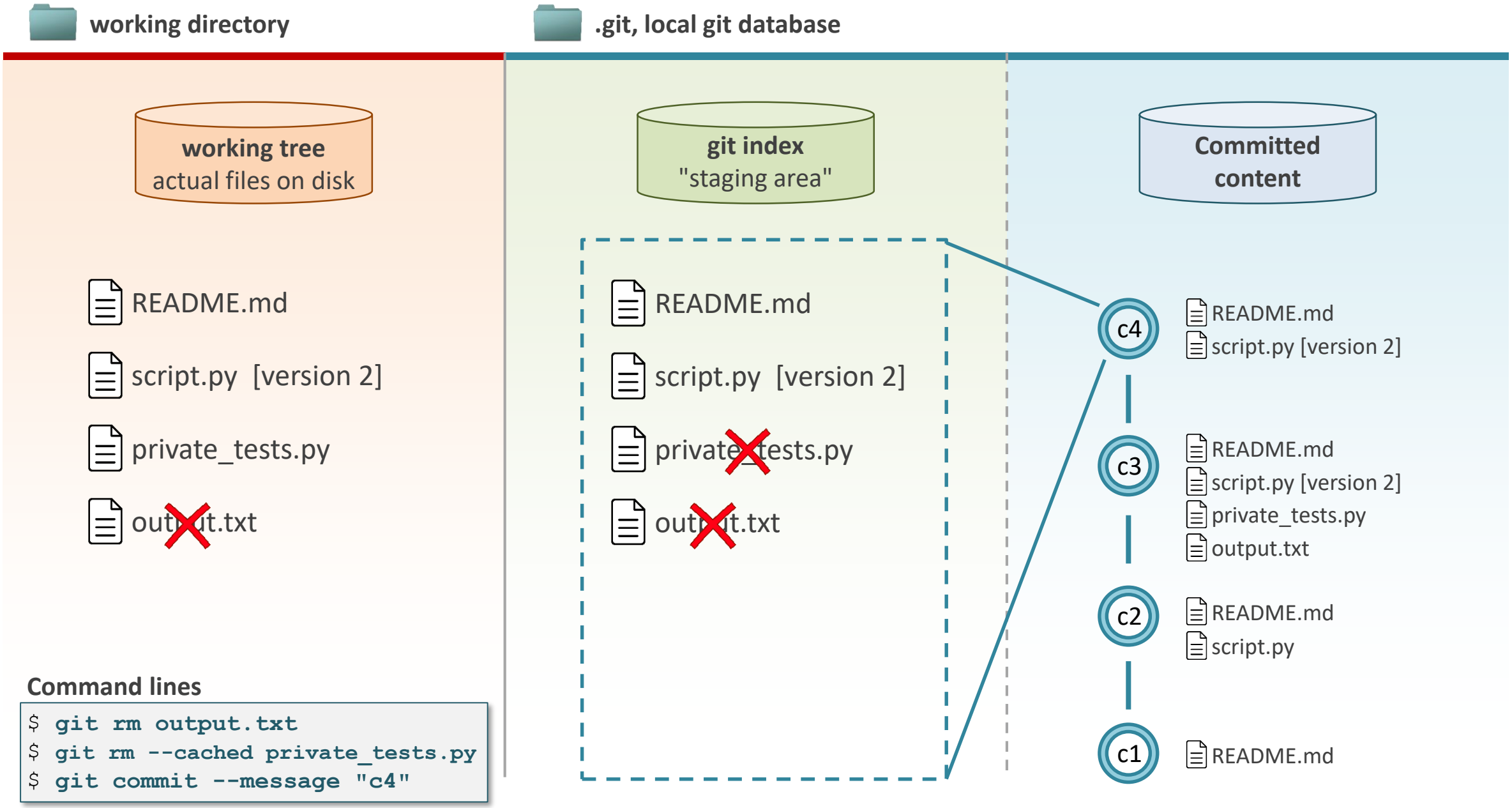
⇨ The objective of this 2-step procedure is to let users craft **"well thought-out" commits**.

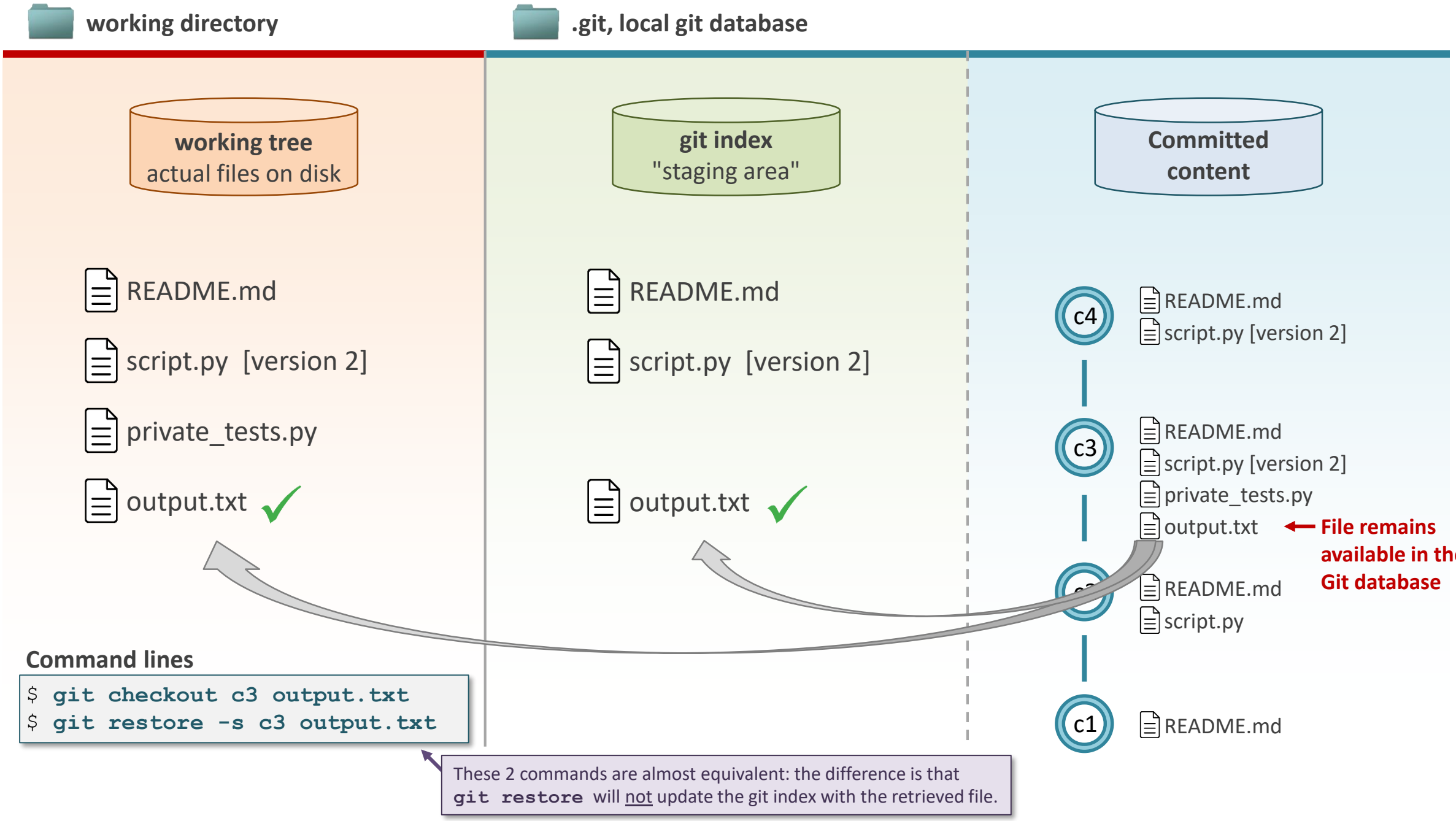
- **Commits are** meant to be **meaningful units of change** in your code base (or the content you track).
- Not all current changes in the working directory need to be part of the next commit.


working directory
.git, local git database
working tree
actual files on disk
git index
"staging area"
Committed
content
README.md
README.md
c1
README.md
Command lines
$ git add README.md
$ git commit --message "c1"


working directory
.git, local git database
working tree
actual files on disk
git index
"staging area"
Committed
content
README.md
script.py
README.md
script.py
c2
README.md
script.py
c1
README.md
Command lines
$ git add script.py
$ git commit --message "c2"


working directory
.git, local git database
working tree
actual files on disk
git index
"staging area"
Committed content
README.md
script.py [version 2]
private_tests.py
output.txt
README.md
script.py [version 2]
private_tests.py
output.txt
c3
README.md
script.py [version 2]
private_tests.py
output.txt
c2
README.md
script.py
c1
README.md
Command lines
$ git add --all
$ git commit --message "c3"


working directory
.git, local git database
working tree
actual files on disk
git index
"staging area"
Committed content
README.md
script.py [version 2]
private_tests.py
output.txt
README.md
script.py [version 2]
private_tests.py
output.txt
c4
README.md
script.py [version 2]
c3
README.md
script.py [version 2]
private_tests.py
output.txt
c2
README.md
script.py
c1
README.md
Command lines
$ git rm output.txt
$ git rm --cached private_tests.py
$ git commit --message "c4"


working directory
.git, local git database
working tree
actual files on disk
git index
"staging area"
Committed content
README.md
script.py [version 2]
private_tests.py
output.txt
README.md
script.py [version 2]
output.txt
c4
README.md
script.py [version 2]
c3
README.md
script.py [version 2]
private_tests.py
output.txt
File remains available in the Git database
README.md
script.py
c1
README.md
Command lines
$ git checkout c3 output.txt
$ git restore -s c3 output.txt
These 2 commands are almost equivalent: the difference is that git restore will not update the git index with the retrieved file.

Adding content to the index (staging content)

```
git add <file/directory>        # add the selected file/directory to the git index.
```

- Adds the file content to the Git index (*stages* a file).
- Multiple files/directories can be added in a single command (by passing multiple file/directory names)
- By default, the entire content of a file is added.
  (adding only part of a file is possible with the --edit or --patch options)
- **Each time a file is modified, it must be added again** so that the new version of the file gets added to the git index (unless we don't want the change in the file to be part of the next commit of course).

- Useful `git add` options

```
git add -u/--update     # Stages all already tracked files, but ignore untracked files.
git add -A/--all        # Stages all files/dirs in the working directory (except ignored files), including file deletions.
git add .               # Stages entire content of working directory, except file deletions.
```

Removing content from the index

- **git restore --staged / git reset HEAD**: remove newly staged content from the index.

```
git restore --staged <file>  # remove newly staged content of specified file.
```

Note: this is a specific use of the reset command, which has a wider scope.

```
git reset HEAD <file>    # remove newly staged content of specific file.
git reset HEAD           # remove all newly staged content.
```

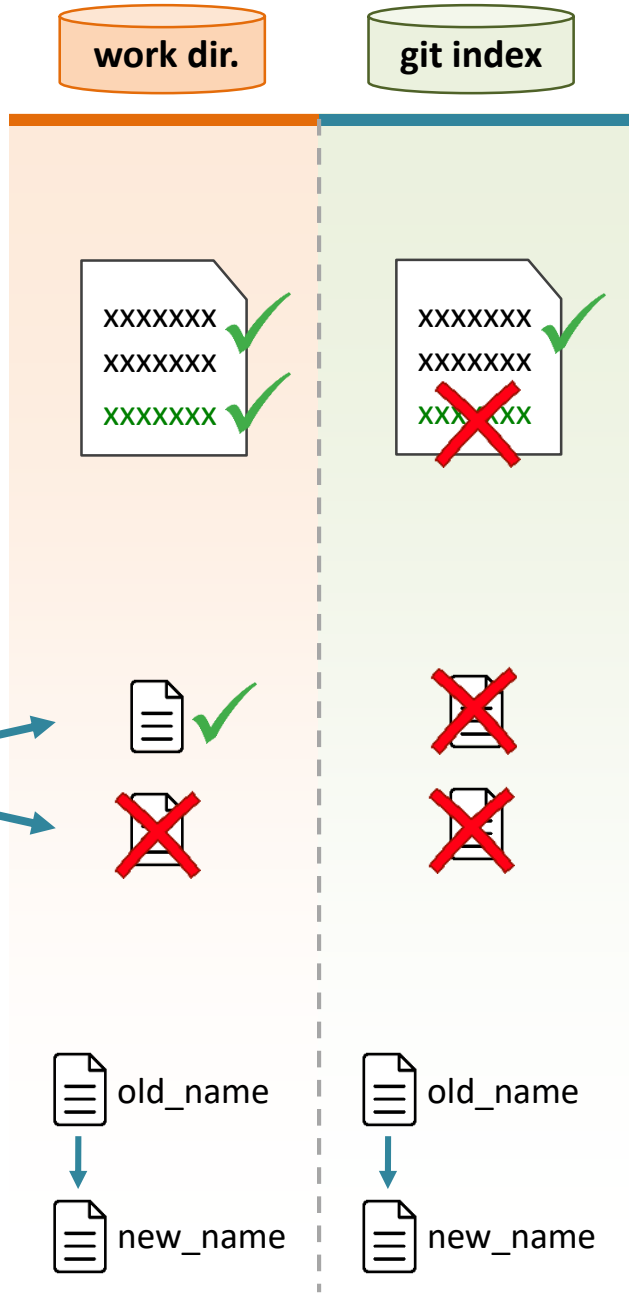
- **git rm**: remove entire files from the index and the working tree.

```
git rm --cached <file>   # remove file from index only.
git rm <file>            # remove file from both index and working tree.
```

With no `--cached` option => deletes file on disk !

- **git mv**: rename and/or move files both in the working tree and the index.

```
git mv <file> <new location/new name>
```



How do I know which *files* are staged? use git status!

Green = new content in this file has been staged and will be part of the next commit.

Red = this file contains new content, but it is not staged and will not be part of the next commit.

git status

```
$ git status
On branch main
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   LICENSE.txt
        modified:   README.md

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working
directory)

        modified:   README.md

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

        untracked_file.txt
```

tracked files.

untracked files.

- **Tracked files that have not been modified** since the last commit are not listed (but they will still be part of the next commit).
- **Ignored files** are also not listed.

Reminder...

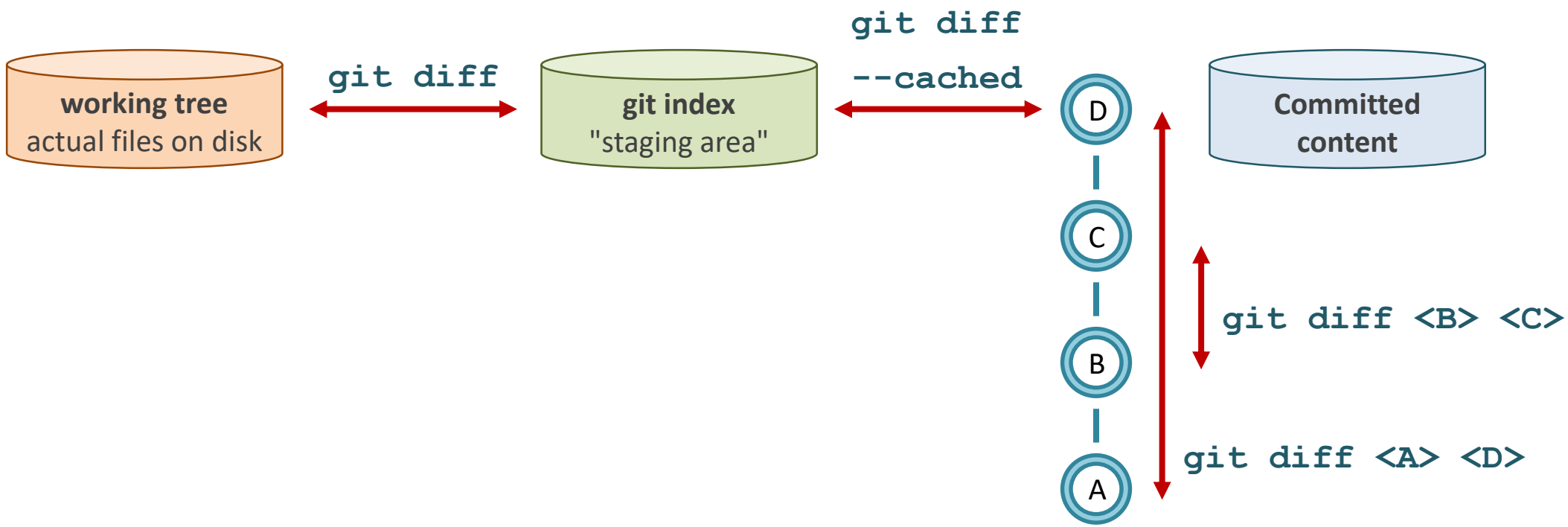
How do I know which *changes* are staged? use git diff

- Show differences between two states of the git repo.

```
git diff
git diff <file>                    # show diff only for a specific file.
git diff --cached
git diff <commit 1 (older)> <commit 2 (newer)>
git diff --name-only               # show only file names, not the changes.
```

Example:

```
$ git diff
diff --git a/README.md b/README.md
index f5e333d..844d178 100644
--- a/README.md
+++ b/README.md
@@ -1,2 +1,3 @@
Project description:
-This is a test
+This is a demo project
+and it's pretty useless
```



Shortcuts: add + commit in a single command

- Stage all changes in ***tracked*** files (does not add untracked files).

```
git add -u
```

- Stage all changes in ***tracked*** files and commit them.
  This will not commit untracked files (unlike `git add --all` that also adds untracked files).

```
git commit --all --message "log message"
git commit -am "log message"                    # short options version.
```

- Stage + commit all changes in the specified files:

```
git commit -m "log message" <file to commit>
```

This only works for files that are already tracked.

Example

```
$ git commit -m "README: updates project description" README.md

Is the same as:
$ git add README.md
$ git commit -m "README: updates project description"
```

Supplementary material...

the **HEAD** pointer

The tip of your current branch

HEAD: a pointer to the most recent commit on the currently active branch

Looking at the output of `git log`, we see a **HEAD ->** label: this **shows the position of the HEAD pointer**.

Commit ID (SHA1 hash)
Here shown in a shortened form (7 first chars).

HEAD pointer

Local branch name

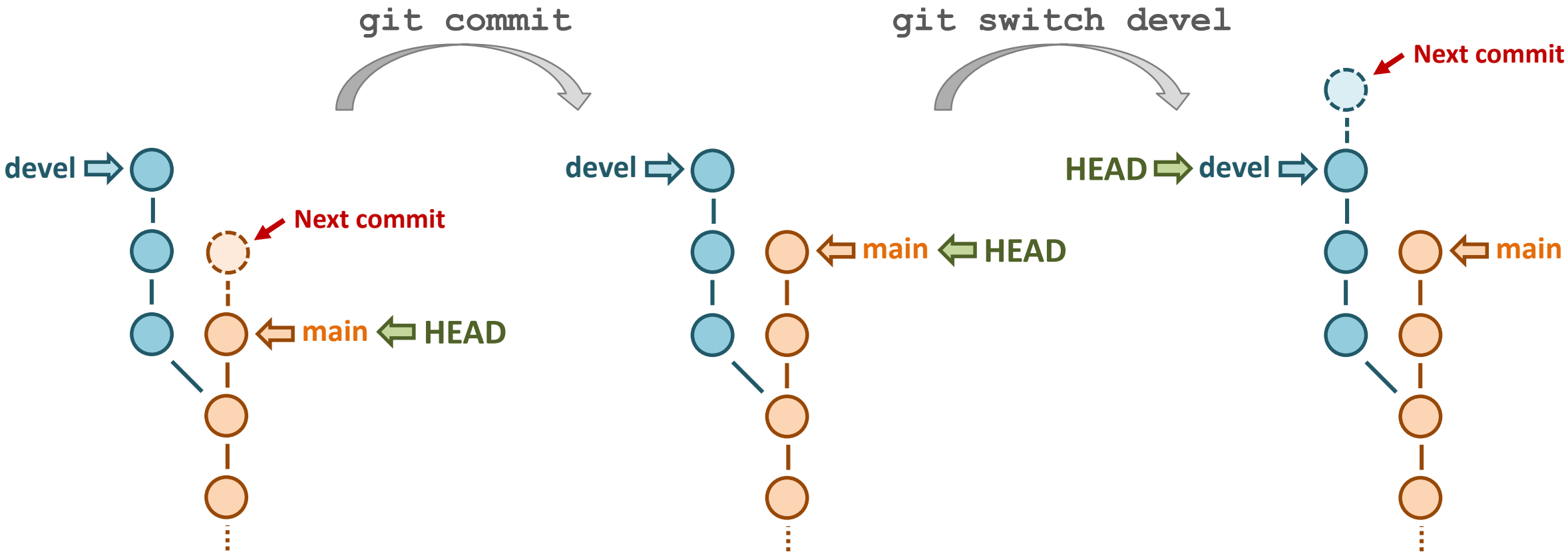
Remote branch name

First line of commit message

```
❯ git log --all --decorate --oneline --graph
* 351dca6 (HEAD -> main, origin/main, origin/HEAD) peak_sorter: added authors to script
* f3d8e22 peak_sorter: display name of highest peak when script completes
| * 076aa80 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
| * d29958d peak_sorter: added authors as comment to script
| * 6c0d087 peak_sorter: improved code commenting
| * 89d201f peak_sorter: add Dahu observation counts to output table
| * 9da30be README: add more explanation about the added Dahu counts
| * 58e6152 Add Dahu count table
|/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 8e0d4fe (origin/dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```

HEAD: a pointer to the currently checked-out branch/commit

- **HEAD** is – most of the time – a pointer to the latest commit on your current branch.
  (Sometimes it is also described as a pointer to the current branch – which is itself a pointer to the latest commit on the branch)
- The **HEAD** position is how Git knows what is the currently "active" branch.
- New commits are added "under" the current **HEAD**, i.e. a new commit is the "child" of the commit pointed-to by **HEAD**.
- When a new commit is added, **HEAD** is automatically moved by Git to point to that new commit.



Another way to look at it, is that **HEAD always points to the parent of your next commit**.

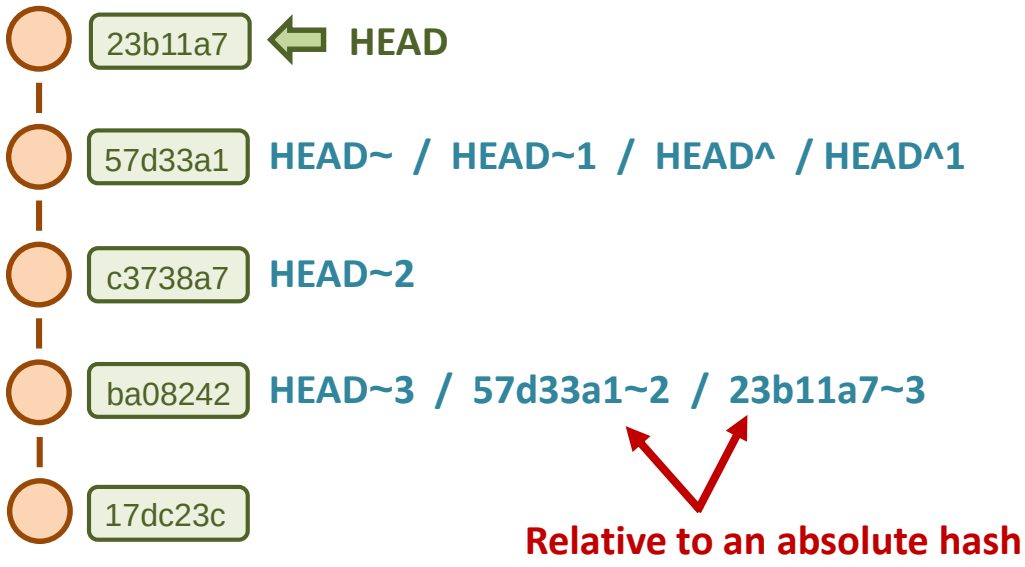
Relative references to commits

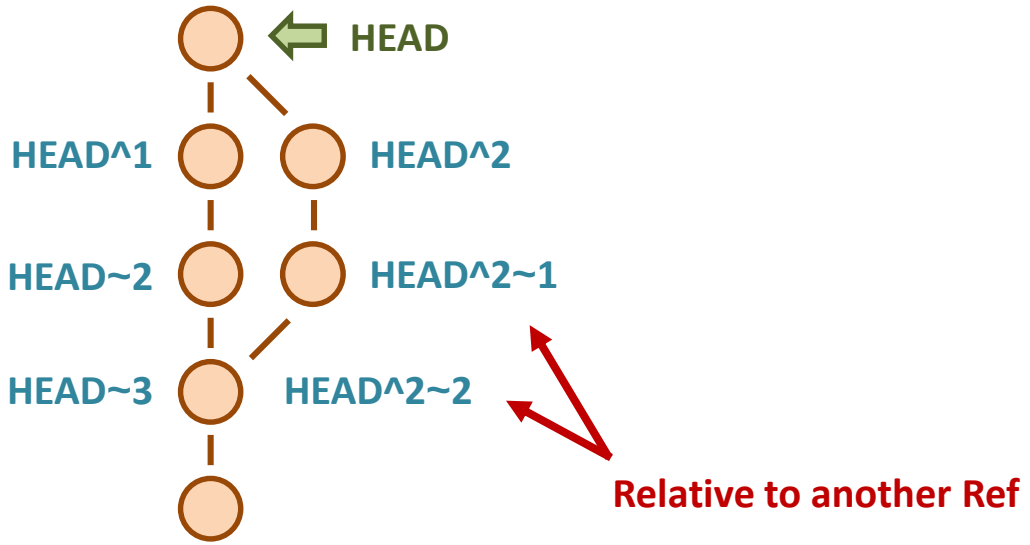
- Using `~` and `^` symbols, Git allows to refer to a commit by its position relative to another commit, rather than by its absolute hash.

- `Ref` can be any reference, such as **HEAD**, a commit hash, a branch name, or even another Ref.

`Ref~X`	refers to the **Xth generation before** the commit: ~1 = parent, ~2 = grand-parent, etc.
`Ref~`	is a shortcut for `Ref~1`

`Ref^X`	refers to the **Xth direct parent** of the HEAD commit (but most commits have only a single parent).
`Ref^`	is a shortcut for `Ref^1`





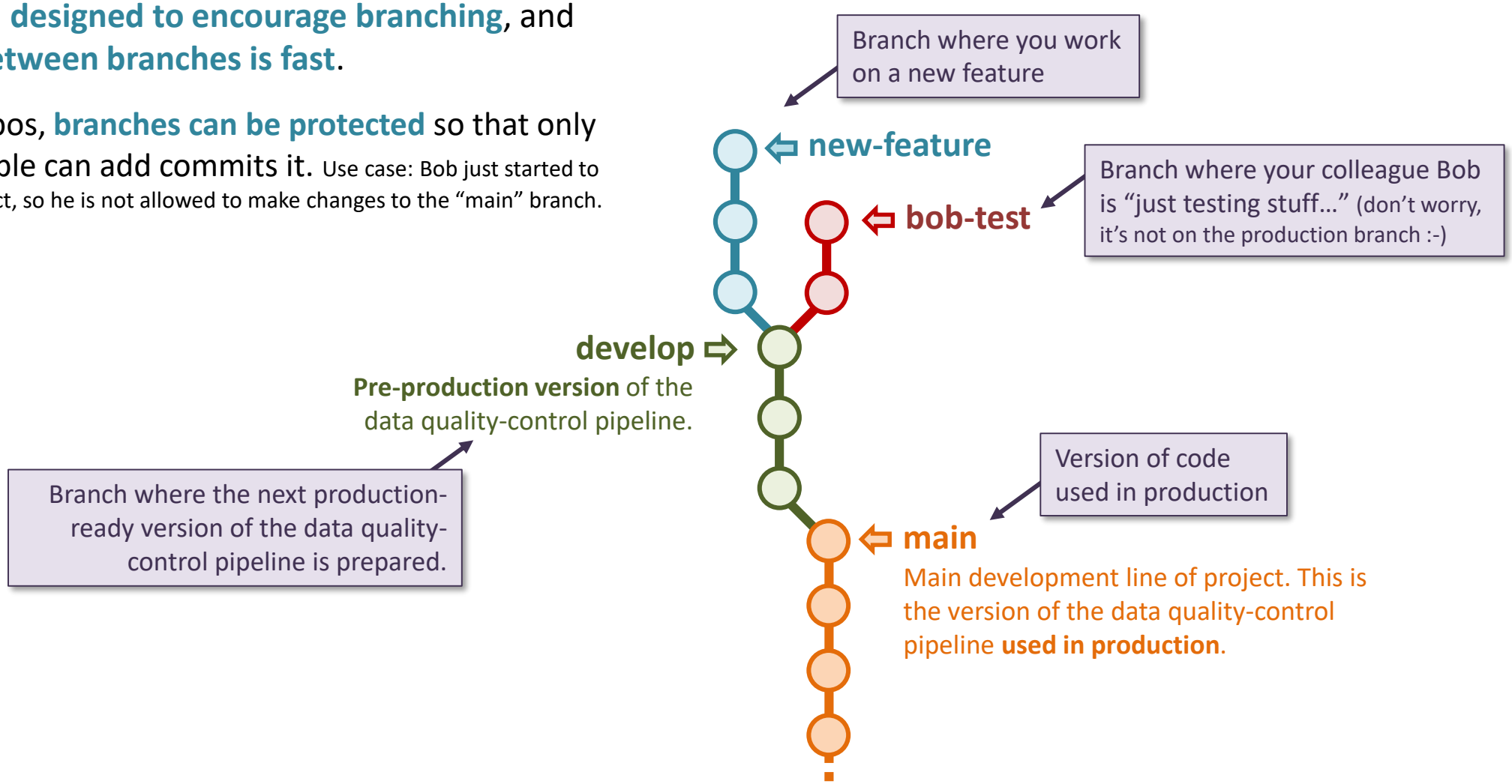
Supplementary material...

Git branches

run multiple lines of development

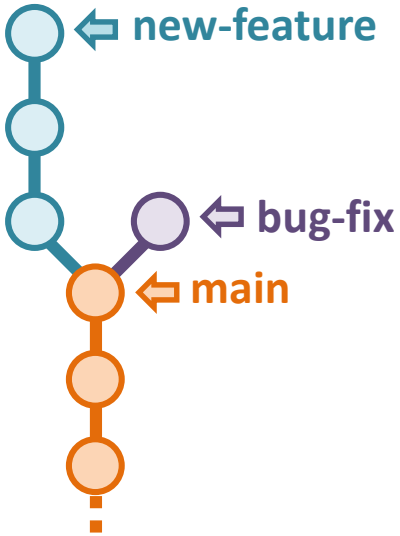
Why branches? An example of a data quality-control pipeline project

- Branches are **a great way to isolate new changes you are working on** from the main line of development.
- Git has been **designed to encourage branching**, and **switching between branches is fast**.
- On online repos, **branches can be protected** so that only selected people can add commits it. Use case: Bob just started to work on our project, so he is not allowed to make changes to the "main" branch.



What are branches?

- A branch is just a pointer (to a commit).
- A branch is **very lightweight** (41 bytes).
- By convention, the **main/master** branch is the branch representing the stable version of your work.
- Git is **designed to encourage branching**: branches are "cheap" to create (they use little disk space), and switching between them is fast.



The main/master branch

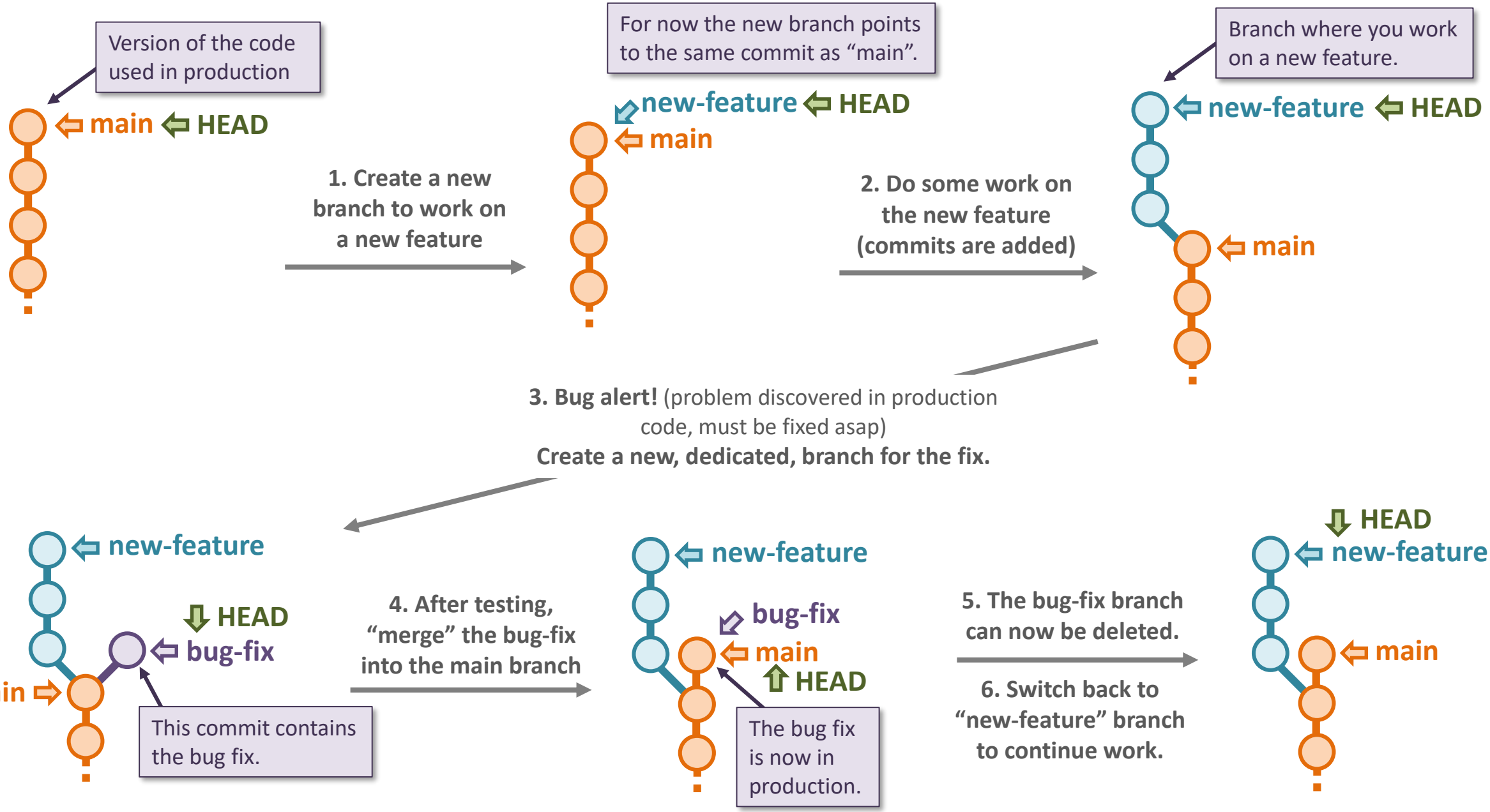
The **main/master** branch **is no special branch**. It is simply the default name given to the branch created when initializing a new repo [`git init`]. It has become a convention to use this branch as the stable version of a project.

main or master ?
Since October 2020, repositories created on GitHub use **main** instead of **master** as their default branch (because "master" carries a references to slavery). Recent versions of Git have now also adopted this convention.

Illegal characters within branch names

Spaces and some characters such as `,~^:?*[]\` are not allowed in branch names. It is strongly recommended to stick to lowercase letters, numbers and the "dash" character [**–**].

Example of branched workflow: adding a new feature to an application and fixing a bug

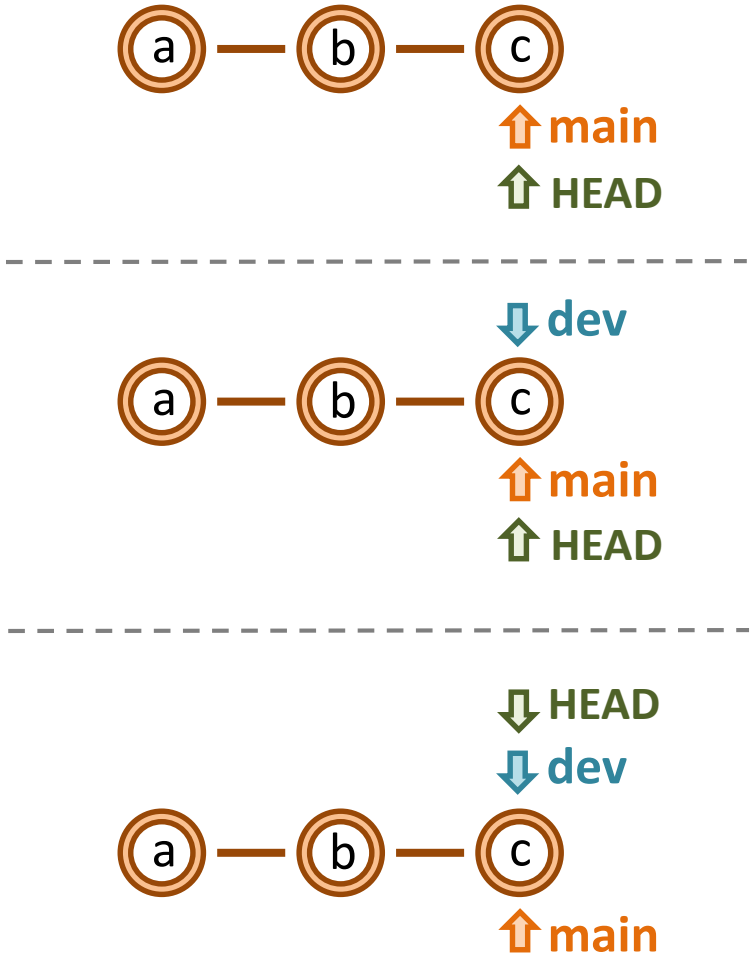


Switching and creating new branches

Create a new branch: `git branch <branch name>`

Create a new branch **and switch** to it: `git switch -c <branch name>`

Same as above, for older Git versions: `git checkout -b <branch name>`



`git branch dev`

`git switch dev`
or
`git checkout dev`

`git switch -c dev`

The `-c` option is to create and switch to the new branch immediately.

checkout vs. switch

The `git switch` command was introduced in Git version 2.23 as an alternatively/replacement to `git checkout` when switching branches. This is because the `checkout` command already has other uses (e.g. to revert files to a given version), and it was deemed confusing that a same command would have multiple usages.

List branches and identify the currently active branch

`git branch` List **local** branches

`git branch -a` List local **and remote** branches

Examples

```
$ git branch
devel
* main
new-feature
```

The * denotes the currently checkout-out (active) branch. Generally it is also displayed in green.

```
$ git branch -a
devel
* main
new-feature
remotes/origin/main
remotes/origin/devel
```

Remote branches (to be precise, pointers to remote branches) are shown in red and are named **remotes/<remote name>/<branch name>**

As a handy alternative, “git adog” (`git log --all --decorate --oneline --graph`) will also show all branches.
The currently active branch can be identified as it has the **HEAD** pointing to it.

```
* 351dca6 (HEAD -> main, origin/main, origin/HEAD) peak_sorter: added authors to script
* f3d8e22 peak_sorter: display name of highest peak when script completes
| * 076aa80 (origin/feature-dahu, feature-dahu) peak_sorter: display highest peak at end of script
| * d29958d peak_sorter: added authors as comment to script
| * 6c0d087 peak_sorter: improved code commenting
| * 89d201f peak_sorter: add Dahu observation counts to output table
| * 9da30be README: add more explanation about the added Dahu counts
| * 58e6152 Add Dahu count table
|/
* cfd30ce Add gitignore file to ignore script output
```

git merge

get branches back together

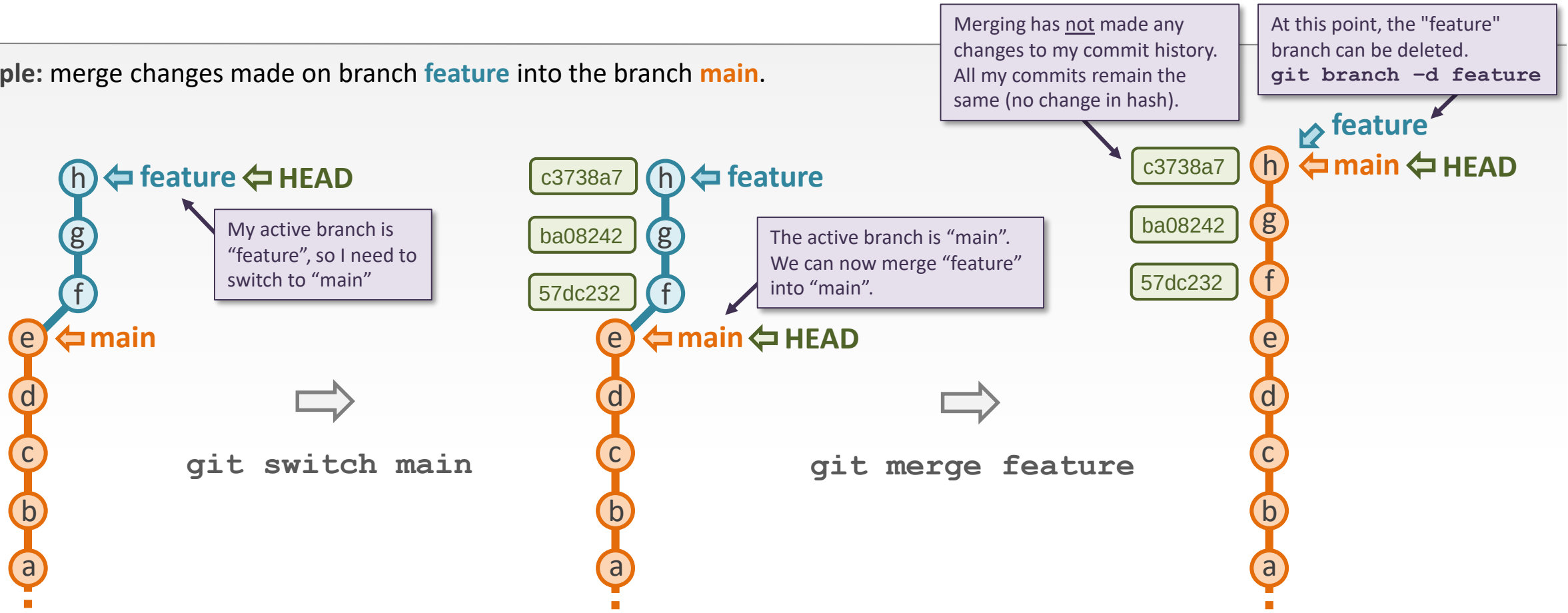
Branch merging

- **Merge:** incorporate changes from the specified branch into the currently active (checked-out) branch.

```
git merge <branch to merge into the current branch>
```

Before running the command, make sure that the branch into which the changes should be merged is the currently active branch.
If not, use `git switch <branch>` to checkout the correct branch.

Example: merge changes made on branch **feature** into the branch **main**.

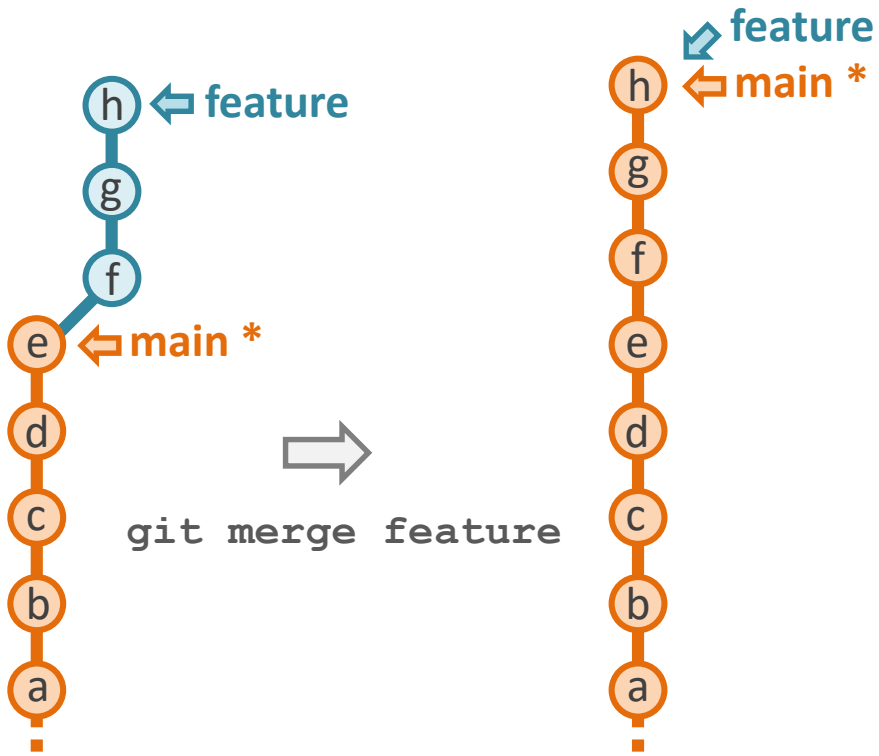


Two types of merges

- **Fast-forward merge:** when branches **have <u>not</u> diverged.**
  The branch that is being merged (here **feature**) is rooted on the latest commit of the branch that it is being merged into (here **main**).
- **3-way merge:** when branches **have diverged**. This introduces an extra "**merge commit**".
  The **common ancestor** of the 2 branches is not the last commit of the branch we merge into (here **main**).

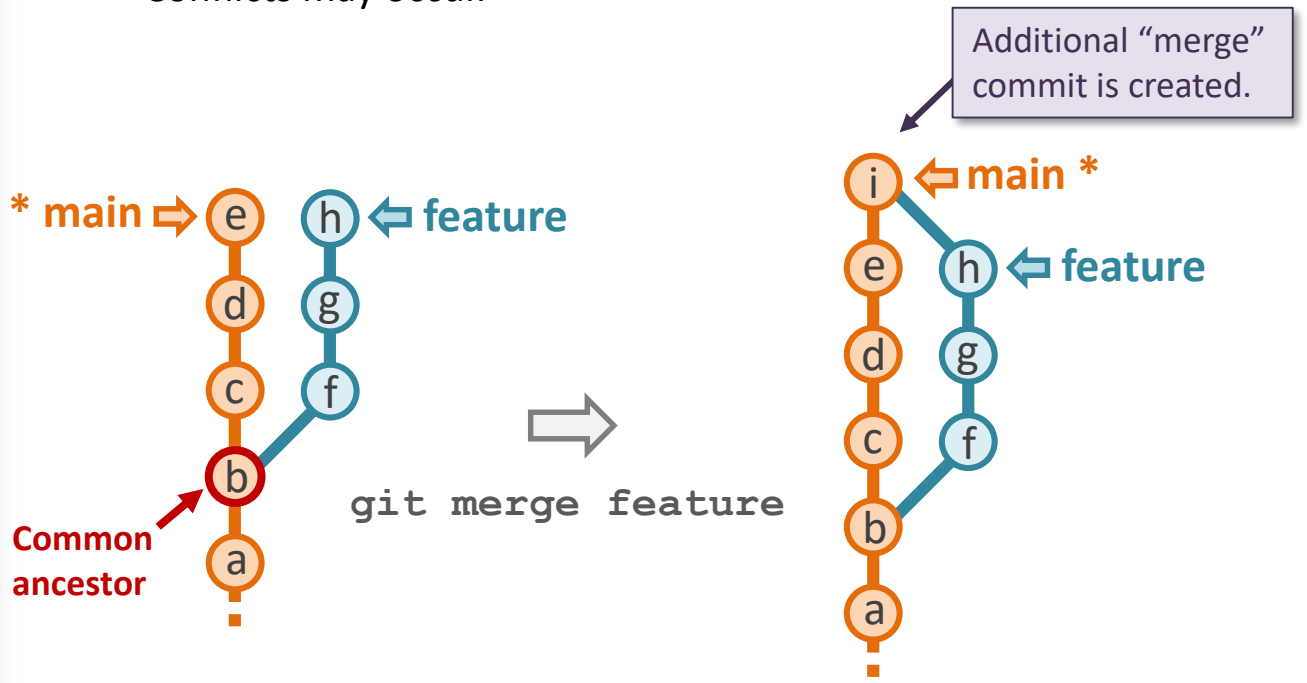
Fast-forward merge

- Guaranteed to be conflict free.



3-way merge (non-fast-forward)

- Creates an additional "merge commit" (has 2 parents).
- Conflicts may occur.



* denotes the currently active (checkout-out) branch.

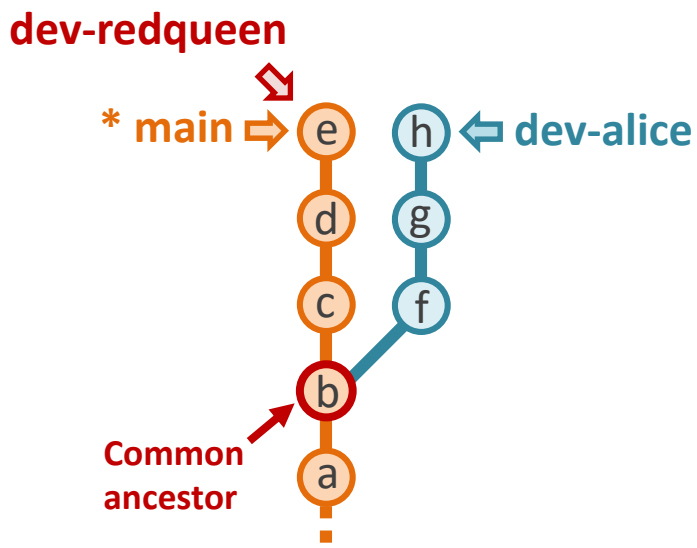
Conflicts in 3-way merges (non fast-forward)

If **a same file is modified at (or around) the same place** in the two branches being merged, Git cannot decide which version to keep. There is a conflict, and **you need to manually resolve** it.

Story background: the Red Queen has just merged changes from her branch **dev-redqueen** into **main**.
Now Alice wants to merge her branch **dev-alice** into **main**.

README.md version of main branch.

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.

Authors: Mad Hatter, Red Queen
Date modified: 2022 Oct 10

## Step 1: physical integrity check
* Check exterior for cracks and uneven
  painting.
* Check for mice inside of pot.
* Verify the Mad Hatter is on time.

## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
```



README.md version of dev-alice branch.

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.

Authors: Mad Hatter, Alice
Date modified: 2022 Oct 11

## Step 1: physical integrity check
* Check exterior for cracks and uneven
  painting.
* Check for mice inside of pot.

## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
* Make sure we still have no idea why
  a raven is like a writing desk.
```

Let's merge dev-alice into main...

```
$ git merge dev-alice
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

← File with conflicts that need to be manually solved.

Resolving conflicts

```
$ git merge dev-alice
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

File with conflicts

1. **Open the conflicting files** in the text editor of your choice.
2. **Look for the text between `<<<<<<<` and `>>>>>>>` .**

   There can be more than one of such sections, if there is more than one conflict in the file.

   - The text between `<<<<<<<` and `=======` is the version of the current branch, i.e. the branch into which we merge (**main**, in this example).
   - The text between `=======` and `>>>>>>>` is the version from the branch we are merging (**dev-alice**, in this example).

Version from the current branch (here **main**).

Version from branch being merged into the current branch (here **dev-alice**).

Note: there is no conflict for these 2 lines, because the edits were made at different locations in the file. Git is able to auto-merge such changes.

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.

<<<<<<< HEAD
Authors: Mad Hatter, Red Queen
Date modified: 2022 Oct 10
=======
Authors: Mad Hatter, Alice
Date modified: 2022 Oct 11
>>>>>>> dev-alice

## Step 1: physical integrity check
* Check for mice inside of pot.
* Verify the Mad Hatter is on time.

## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
* Make sure we still have no idea why a
  raven is like a writing desk.
```

3. Manually edits the file(s)…

```
# Tea pot quality-control pipeline
Check and approve tea pots for use in
unbirthday parties.

Authors: Mad Hatter, Red Queen, Alice
Date modified: 2022 Oct 11

## Step 1: physical integrity check
* Check for mice inside of pot.
* Verify the Mad Hatter is on time.

## Step 2: tea-brewing integration test
* Brew tea for 7 min.
* Add 2 cubes of sugar.
* Probe tea.
* Make sure we still have no idea why a
  raven is like a writing desk.
```

4. Stage the conflict-resolved file(s).
5. Commit

```
$ git add README.md
$ git commit
[main a317d38] Merge branch 'dev-alice'
```

An editor will open with a pre-set commit message. You can accept it as is, or modify it.

Hash of the added "merge" commit.

Resolving conflicts: if you get lost…

- If you are lost at some point, run `git status` and it will give you some hints and commands.
- A merge can be aborted at anytime with `git merge --abort`
- Completed merges can be reverted (with the `git reset` commands – see the "git advanced" slides).

Examples

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   README.md
```

Git tells you what to do and reminds you of commands.

Running `git status` before conflicts are resolved in the file.

```
$ git status
On branch main
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   README.md
```

Git tells you what to do and reminds you of commands.

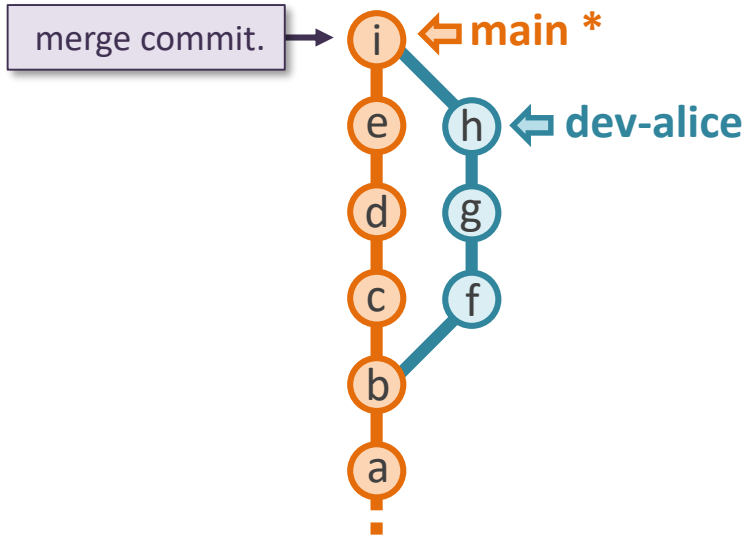
Running `git status` after conflicts are resolved in the file and the file was staged.

What's in a merge commit ?

If there was no conflict, the **merge commit** contains nothing but the commit message (and other metadata).

```
$ git show HEAD
commit 10fa3ad505821b0ea628b811143af47343a4d8dc (HEAD -> main)
Merge: 7446b3e b4fb462
Author: Red Queen <off.with.their.heads@wonder.org>
Date:   Tue Oct 11 15:16:39 2022 +0200

    Merge branch 'dev-redqueen'
```

If there was a conflict, the **merge commit** contains the conflict resolution changes made to the conflicted file(s).




```
$ git show HEAD
commit a317d38448dae4e6bd9b4862dcaccf4e416cc46c (HEAD -> main)
Merge: 10fa3ad 7999c7c
Author: Alice <alice@redqueen.org>
Date:   Tue Oct 11 15:27:35 2022 +0200

    Merge branch 'dev-alice'

diff --cc README.md
index 647be0c,74edef5..3ce8aa7
--- a/README.md
+++ b/README.md
@@@ -1,8 -1,8 +1,8 @@@
  # Tea pot quality-control pipeline
  Check and approve tea pots for use in unbirthday parties.

- - Authors: Mad-Hatter, Red Queen
- - Date modified: 2022 Oct 10
- - Authors: Mad-Hatter, Alice
++Authors: Mad-Hatter, Red Queen, Alice
+ Date modified: 2022 Oct 11

  ## Step 1: physical integrity check
  * Check exterior for cracks and uneven
```

Supplementary material...

demo: branch merging

fast-forward and 3-way merge

Deleting branches

Branches that are merged and are not used anymore can (should) be deleted.

`git branch -d <branch name>` ← **safe option:** only lets you delete branches that are fully merged.

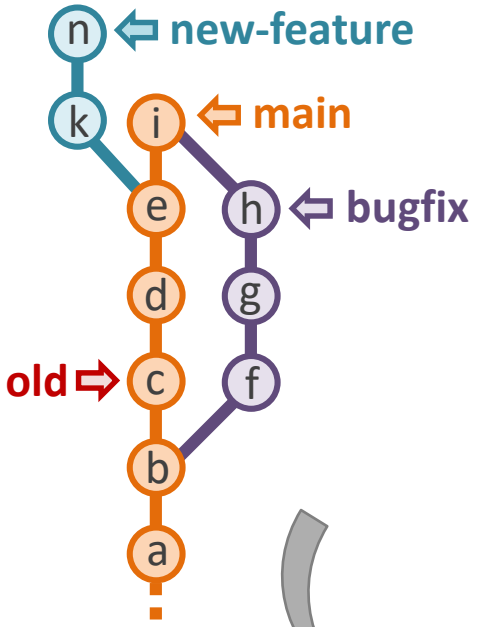
`git branch -D <branch name>` ← **YOLO option:** lets you delete any branch.

- Note: A currently active (checked-out) branch cannot be deleted.
  You must switch to another branch before deleting it.

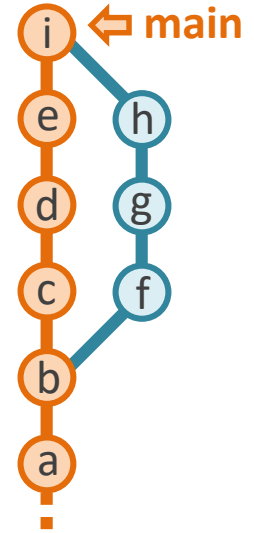
Example

```
# The 'bugfix' and 'old' branches are fully merged.
$ git branch -d bugfix
Deleted branch bugfix (was bd898dc)
$ git branch -d old
Deleted branch old (was 75d3fed)

# Trying to delete a non-merged branch with -d will fail:
$ git branch -d new-feature
error: The branch 'testing' is not fully merged.
If you are sure you want to delete it, run 'git branch -D testing'.

# Using -D will allow deletion of a non-merged branch:
$ git branch -D new-feature
Deleted branch new-feature (was f2a898b)
```

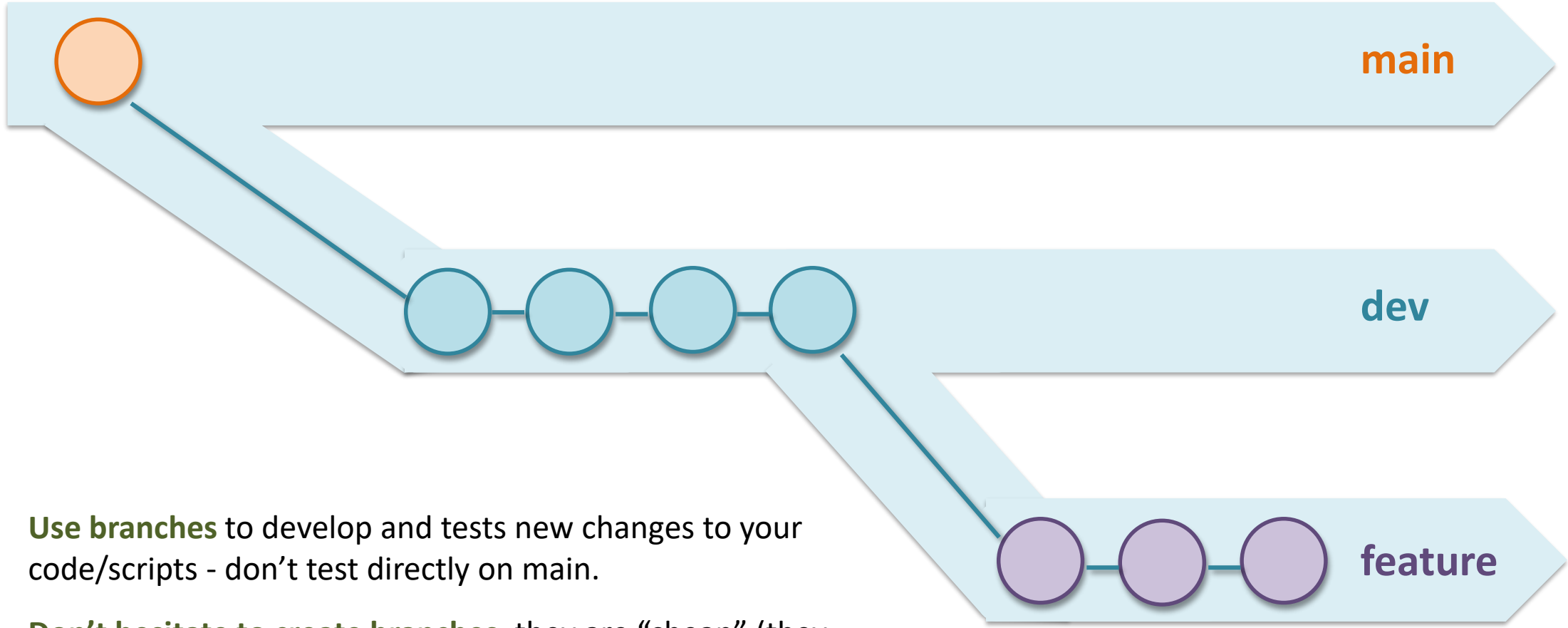
Deleted a branch by mistake ?
This hash can be used to re-create it:
`git branch new-feature f2a898b`



Commands from "Example" box



Branch management: best practices



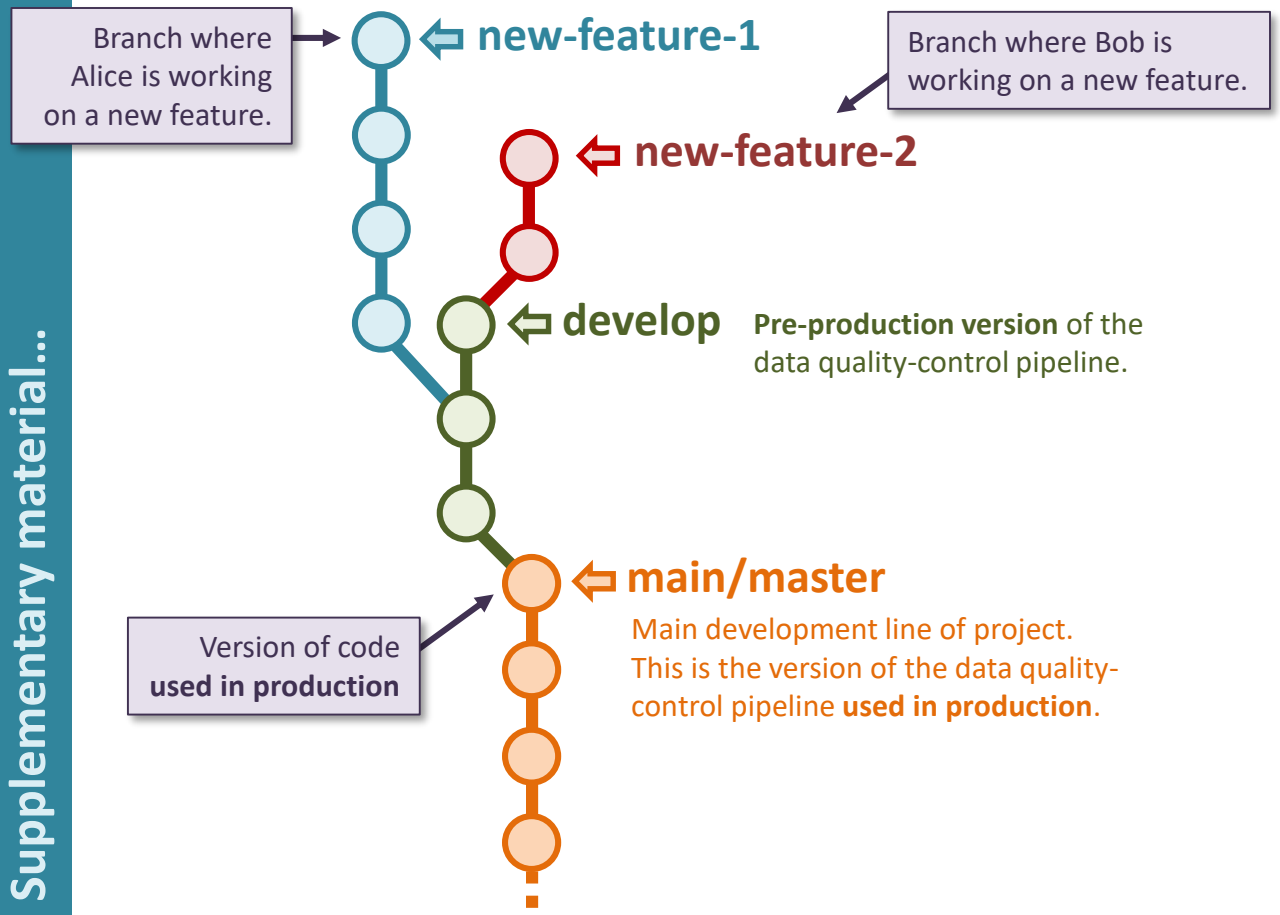
- **Use branches** to develop and tests new changes to your code/scripts - don't test directly on main.
- **Don't hesitate to create branches**, they are "cheap" (they don't add any overhead to the git database).
- Delete branches that are no longer used.

⚠ **Don't change the history on the main branch** if your project is used by others.

Branch management strategies: GitFlow vs. trunk-based development

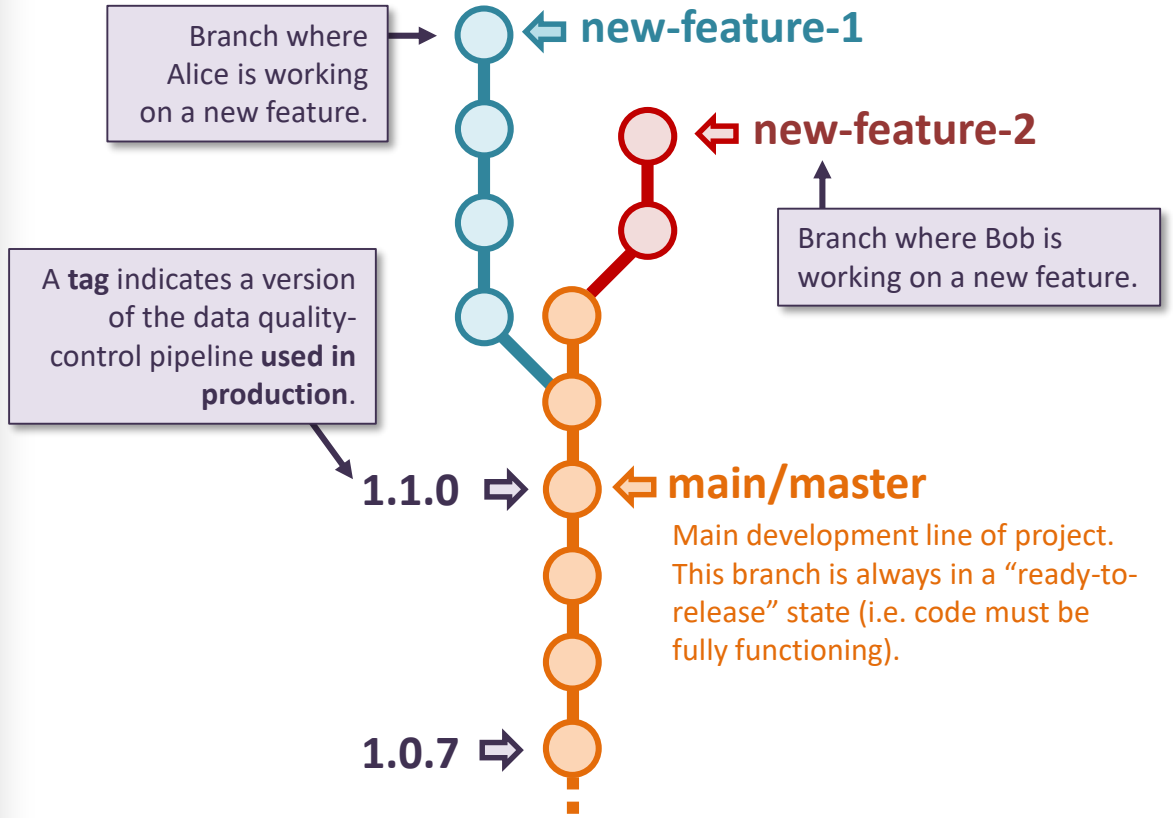
GitFlow: the idea is to have a long-lived **pre-production branch** (here "**develop**"), on which new features are added until ready for a new release, at which point the pre-production branch is merged into **main**.

- Useful if you distribute your code via the **main** branch of the Git repo, without making formal releases, i.e. your end-users use the latest version of **main** in production.



Trunk-based development: there is no long-lived branch outside of the **main** branch. All feature branches are directly merged into **main** once they are completed, and **main** should always be "production-ready". **Tags** are generally added to denote commits corresponding to versions used in production.

- If you distribute your code via formal releases, then this strategy makes more sense as it avoids the overhead of managing an extra long-lived branch (the pre-release breach in GitFlow).



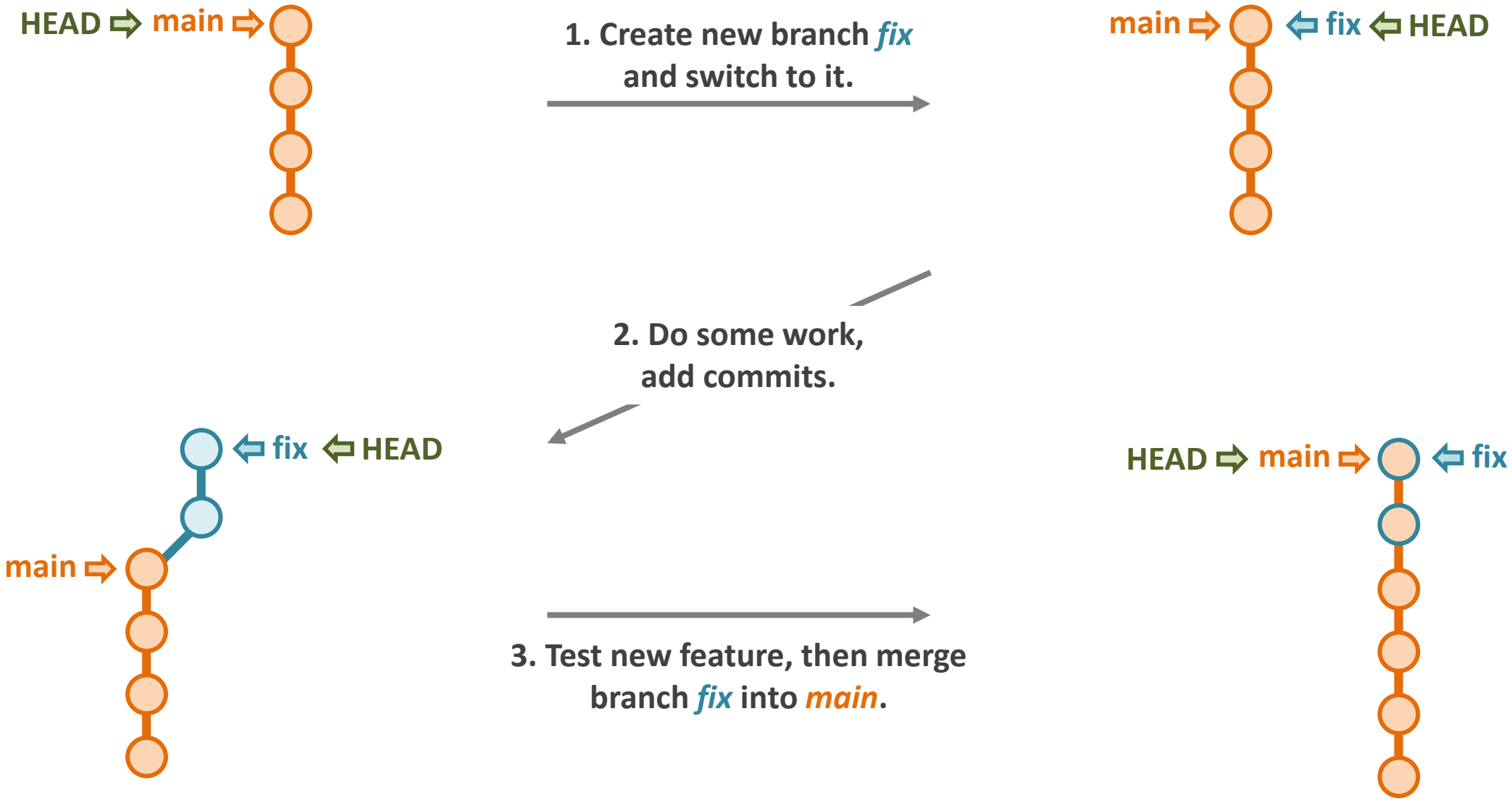
Supplementary material...

exercise 2

The Git reference webpage

This exercise has helper slides

Exercise 2 help: workflow example



git rebase

make a linear history

git rebase: replay commits onto a different base

- `git rebase`: move/re-root a branch to a different base commit.
- **Important**: the rebase command must be executed when on the branch to rebase, not the branch you rebase on.

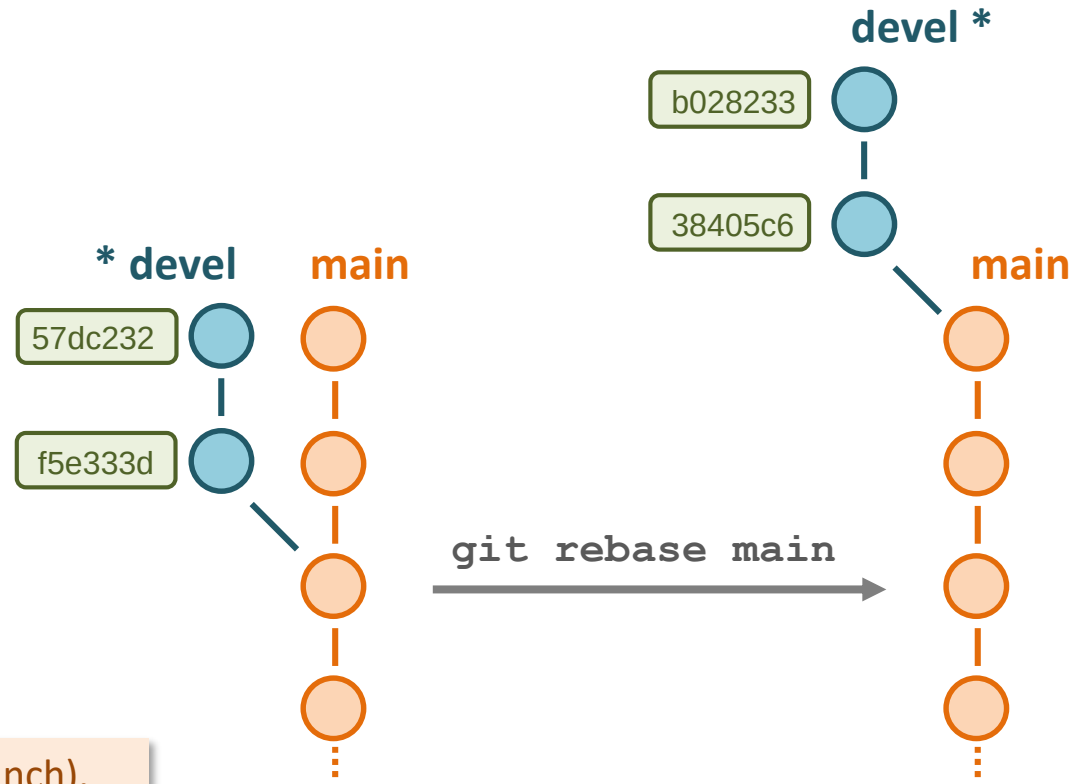
```
git rebase <branch to rebase on>
```

Example:

```
$ git branch
* devel
main

$ git rebase main
```

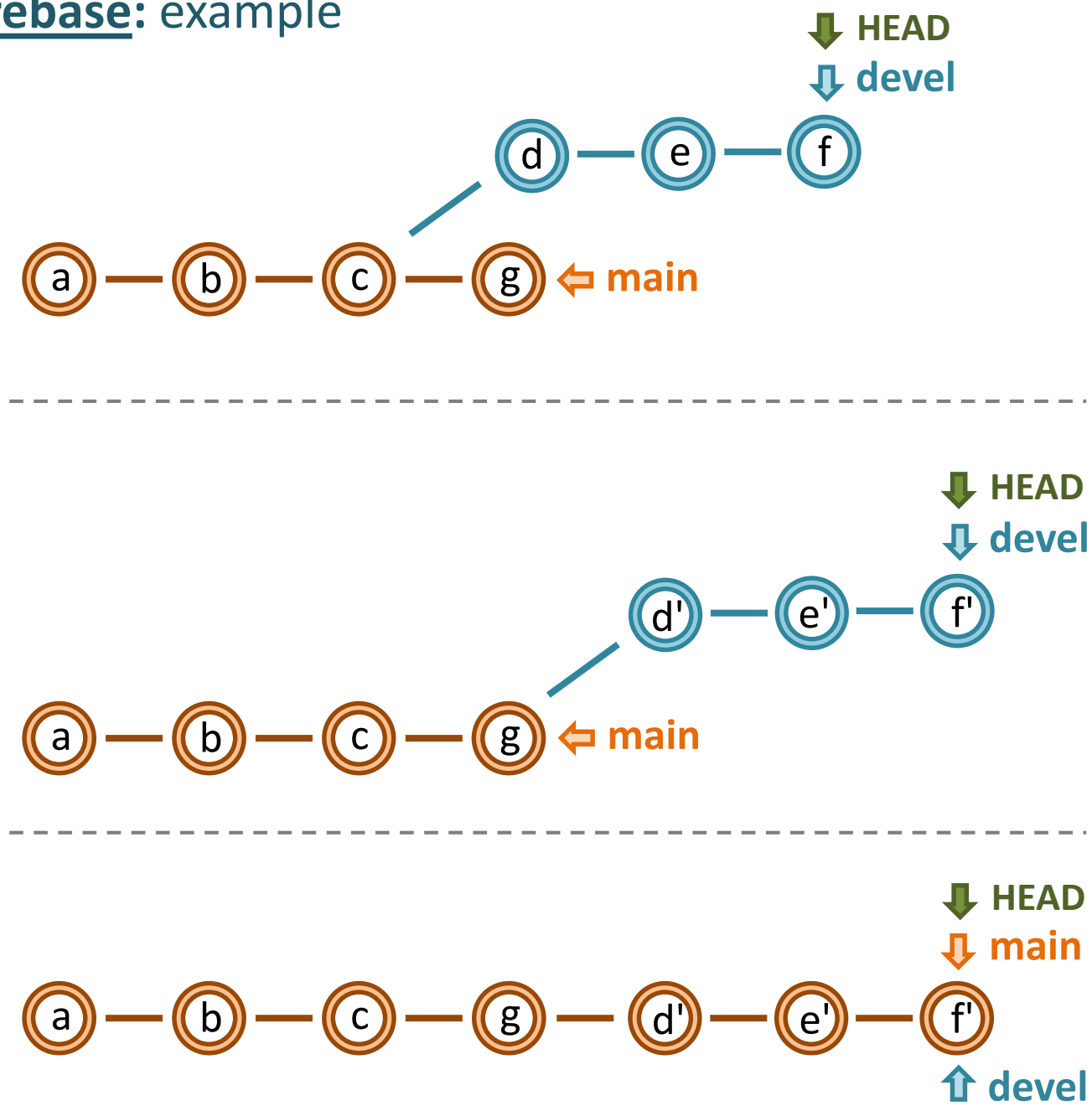
Make sure you are on the branch you want to rebase !

The branch you want to rebase on.



Rebase will modify your commit ID values (history of the rebased branch).
It's best to **only rebase commits that have never left your own computer.**

git rebase: example



Before starting the rebase: make sure you are on the branch to rebase!
In this case, if we are not on *devel*:

```
git switch devel
```

```
git rebase main
```

We can now fast-forward merge.
Guaranteed to be conflict free :-)

```
git switch main
git merge devel
```

Resolving conflicts with rebase

- Rebase re-applies all commit to rebase sequentially: **at each step** there is a potential for conflict…
- To resolve conflicts, you will have to (same as for conflict resolution during merges):

1. **Edit the conflicting files**, choose the parts you want to keep, then remove all lines containing `<<<<<<<`, `=======` and `>>>>>>>`.

2. **Mark the files as resolved** with `git add <file>`

1. **Continue the rebase** with `git rebase --continue`

When a conflict arises, Git will provide guidance:

```
$ git rebase main
First, rewinding head to replay your work on top of it...
Applying: first commit on new branch
Using index info to reconstruct a base tree...
 M       new.txt
Falling back to patching base and 3-way merge...
Auto-merging new.txt

CONFLICT (content): Merge conflict in new.txt
error: Failed to merge in the changes.
Patch failed at 0001 first commit on new branch
Use 'git am --show-current-patch' to see the failed patch

1. → Resolve all conflicts manually,
2. → mark them as resolved with "git add/rm <conflicted_files>"
3. → , then run "git rebase --continue".

You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase",
run "git rebase --abort".
```

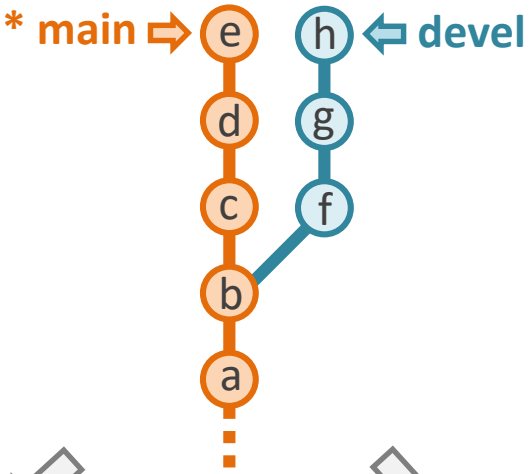
Branch reconciliation strategies when history has diverged: merge vs. rebase

merge (3-way merge)

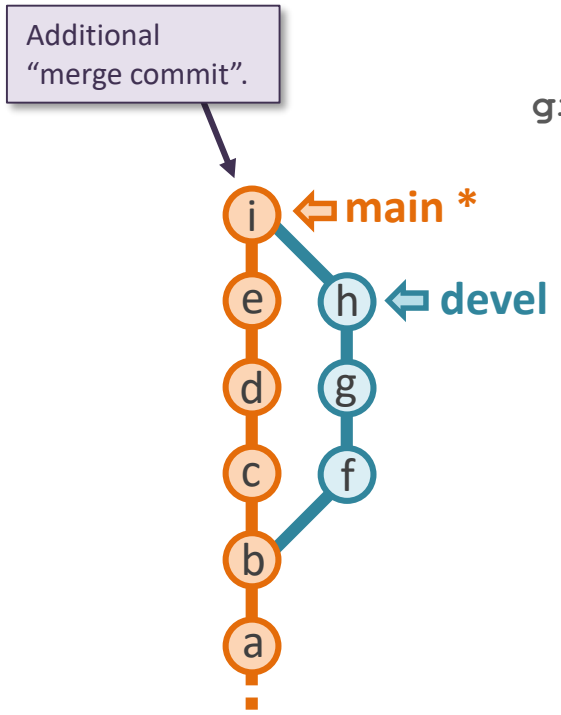
+ **Preserves history perfectly**.
+ Potentials conflicts must be solved only once.
- Creates an additional merge commit.
- Often leads to a "messy" history.

rebase + fast-forward merge

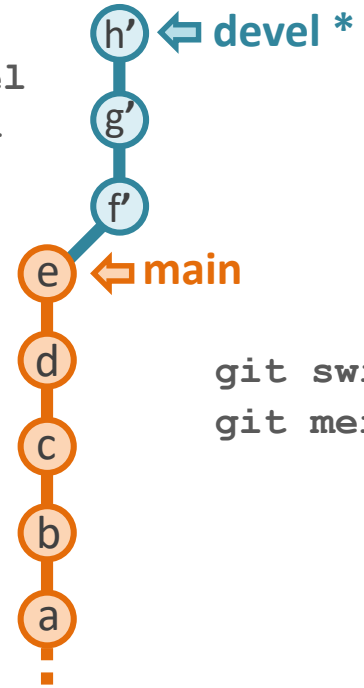
+ **Cleaner history** = easier to read and navigate.
- Conflicts may have to be solved multiple times.
- Loss of branching history.
  History of rebased branch is rewritten, not a problem in general.



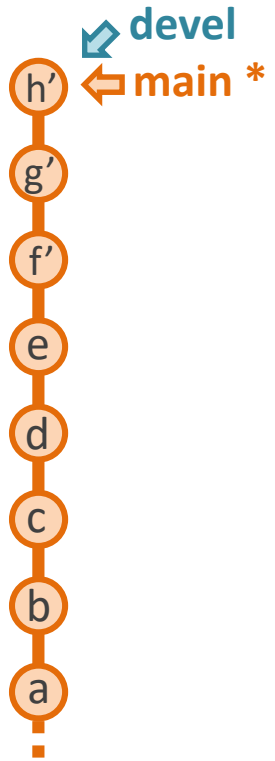
`git merge devel`



`git switch devel`
`git rebase main`



`git switch main`
`git merge devel`



Spoiler-alert: the end result is the same, i and h’ have the same content.

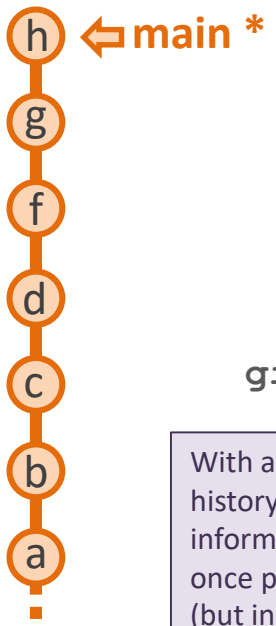
Ultimate history preservation: force the addition of a merge commit with `--no-ff`

If keeping an **exact record** of how the history of a Git repo came into existence is of prime importance, some people like to **add a merge commit even if a fast-forward merge is possible**.

This is possible by adding the `--no-ff` option ("no fast-forward") to git merge.

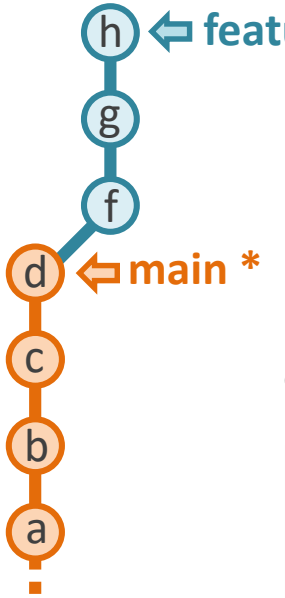
```
git merge --no-ff <branch to merge>
```

```
$ git show 10fa3ad
commit 10fa3ad505821b0ea628b8
Merge: 7446b3e b4fb462
Author: Alice <alice@redqueen.org>
Date: Tue Oct 11 15:16:39 2022 +0200

    Merge branch 'feature'
```



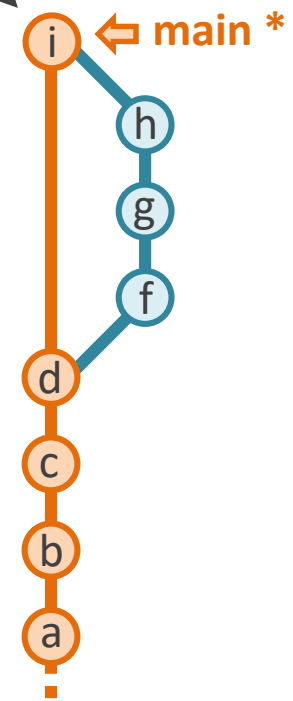
`git merge feature`

With a regular fast-forward merge, the history is cleaner. However, the information that "f", "g" and "h" were once part of a different branch is lost (but in most cases this doesn't matter).



`git merge --no-ff feature`

The merge commit " i " is added for the sole purpose of allowing us to reconstruct the exact history of the repo: it tells us that commits "f", "g" and "h" were once part of a different branch, which was then merged into "main".



Supplementary material...

Readability vs. history preservation tradeoff

Screenshots of two versions of a **same repository** (in the sense that it contains the exact same content with mostly the same commits).

```
*   a317d38 (HEAD -> main) Merge branch 'dev-alice'
|\
| * 7999c7c (dev-alice) improvement: add success message to QC pipeline
| * da96caa fix: update README
| * ccea24b improvement: better tea-brewing checks
* |   10fa3ad Merge branch 'dev-redqueen'
|\ \
| |/
|/|
| * b4fb462 (dev-redqueen) update: add timing module
| * d37df05 improvement: check that Mad Hatter is on time
|/
* 7446b3e update: add tea-brew integration test
* b82c9c9 update: add physical integrity check to pipeline
* 96d19d4 Initial commit
```

Here, history has been fully preserved, by always using merges and forcing extra merge commits (--no-ff) when needed.

Here, having a linear history has been prioritized (better readability), by rebasing branches before (fast-forward) merging them.

```
* 77d8354 (HEAD -> main, dev-alice) improvement: add success message to QC pipeline
* e48c71a fix: update README
* 51ae05e improvement: better tea-brewing checks
* b4fb462 (dev-redqueen) update: add timing module
* d37df05 improvement: check that Mad Hatter is on time
* 7446b3e update: add tea-brew integration test
* b82c9c9 update: add physical integrity check to pipeline
* 96d19d4 Initial commit
```

Supplementary material...

Supplementary material...

Never rebasing your changes before merging can lead to a hard to read history...

demo: branch rebase

feat. manual conflict resolution

git cherry-pick

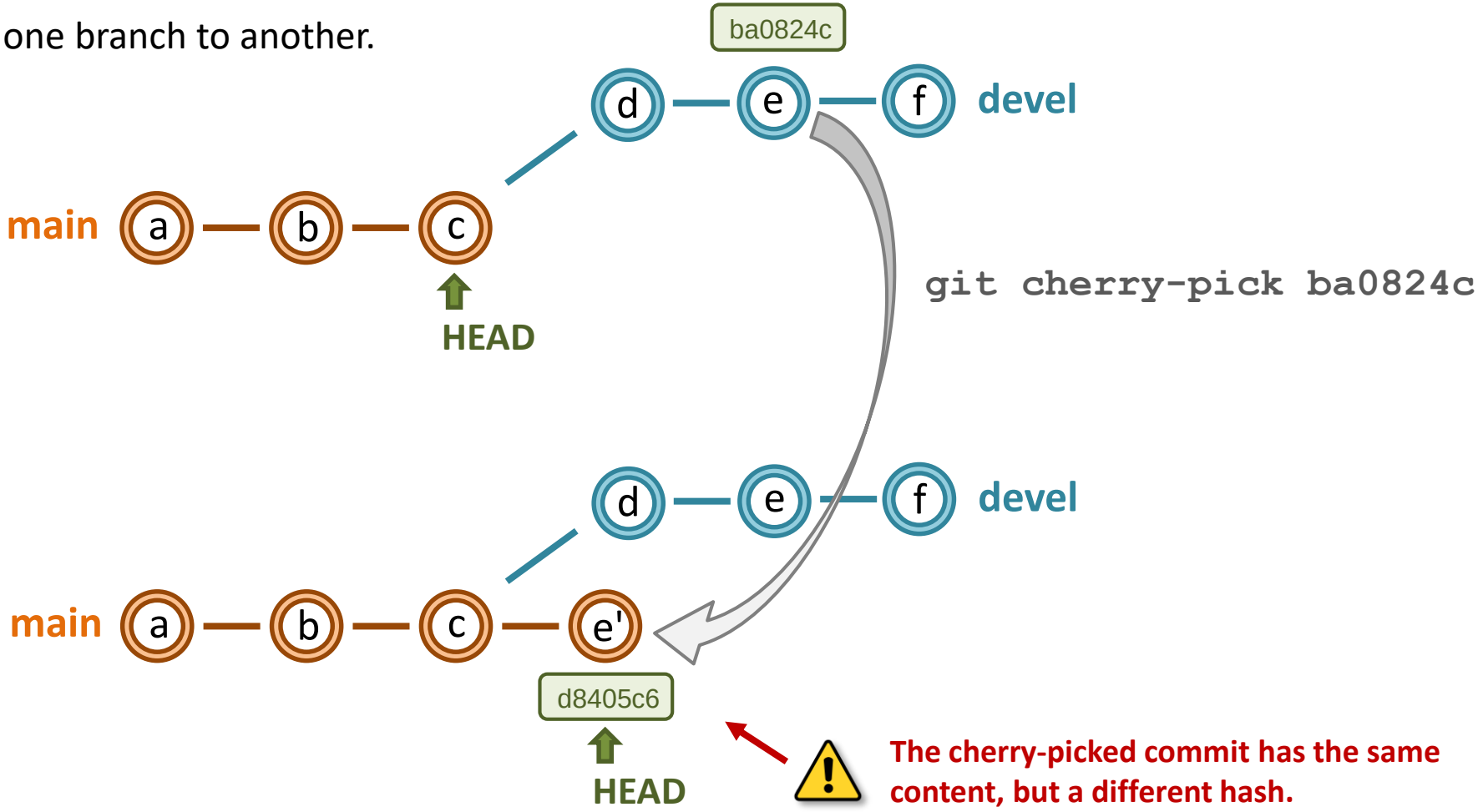
the "copy/paste" for commits

Cherry-pick: merge a single commit into the current branch

- `git cherry-pick`: "copy" a commit (or several) to the current branch.

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

Example:
"copy" a fix from one branch to another.



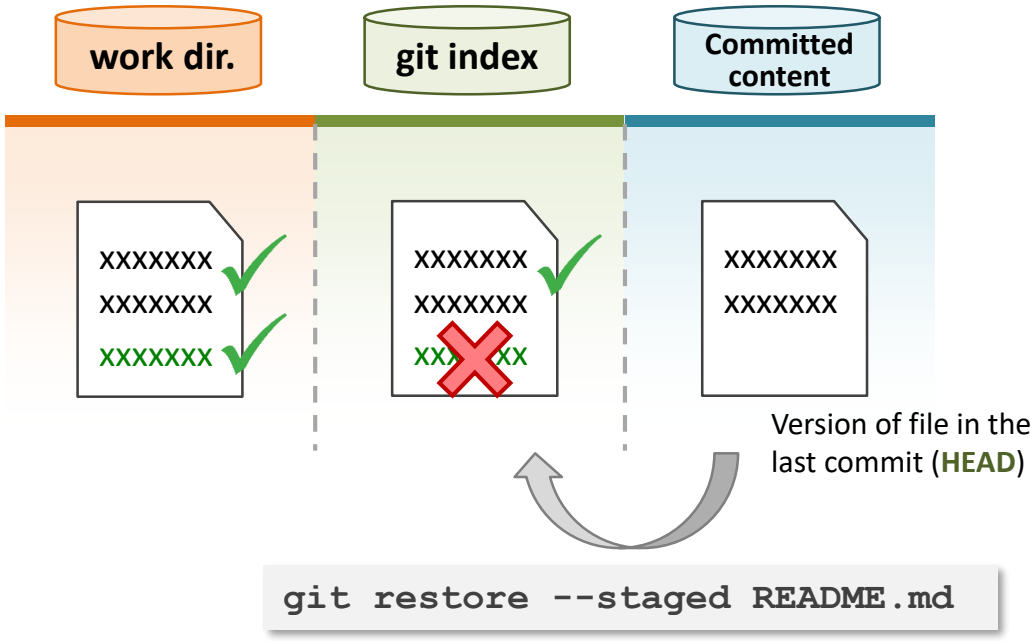
git restore / checkout

retrieve data from earlier commits

Un-stage file modifications (restore file in index)

```
git restore --staged <file name>
```

- Restores the content of a file in the Git index back to the latest commit (**HEAD** commit).
- Does not modify files in the working directory.



Example: un-stage changes to README.md file.

```
$ git status
On branch main
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
          modified:   README.md
```

```
$ git restore --staged README.md
```

```
$ git status
On branch main
Changes not staged for commit:
(use "git add <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
          modified:   README.md
```

The file is still modified in the working directory, but the changes are no longer staged.

Note: the restore command is available from Git >= 2.23

Reminder...

Restore / checkout of individual files

Warning: these commands will overwrite existing versions of the retrieved file in your working directory. Make sure you don't have uncommitted changes you want to keep.

Retrieving the content of a file from an earlier commit can be done with either:

```
git restore -s/--source <commit reference> <file name>
```

or

If no commit references is specified, the file is retrieved from the index.

```
git checkout <commit reference> <file name>
```

Examples: the **<commit reference>** can be e.g. a commit ID, a relative reference, a tag or a branch name.

```
$ git restore -s ba08242 output.txt
$ git restore -s HEAD~10 output.txt
$ git restore -s v2.0.5  output.txt
$ git restore -s devel-branch output.txt
```

using a branch name, implicitly refers to the latest commit on the branch.

```
$ git checkout ba08242 output.txt
$ git checkout HEAD~10 output.txt
$ git checkout v2.0.5  output.txt
  Updated 1 path from 2a7fac8
$ git checkout devel-branch output.txt
  Updated 1 path from e55fa6f
```

A small difference between these two commands is that `restore` updates the file only in the working tree (i.e. the files in your working directory), while `checkout` updates both the working tree and the index.

```
$ git restore --source ad26560 README.md
$ git status
Changes not staged for commit:
(use "git restore <file>..." to discard changes
in working directory)
          modified:   README.md
```

```
$ git checkout ad26560 README.md
Updated 1 path from e55fa6f
$ git status
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
          modified:   README.md
```

Checkout of the entire repo state at an earlier commit

- Checking out a commit will restore both the working tree and the index to the exact state of the specified commit.
- It will also move the **HEAD** pointer to that commit.

```
git checkout <commit reference>
```

Examples:

```
$ git checkout ba08242
$ git checkout HEAD~10
$ git checkout v2.0.5
```

Make sure to have a clean working tree before doing a checkout!

```
$ git checkout ad26560
error: Your local changes to the following files would be
overwritten by checkout:
        README.md
Please commit your changes or stash them before you switch branches
```

- After a checkout, you enter a "detached HEAD" state….


```
$ git checkout ba08242
Note: checking out 'ba08242'.

You are in 'detached HEAD' state. You can look
around, make experimental changes and commit them,
and you can discard any commits you make in this
state without impacting any branches by performing
another checkout.
```

- To get back to a “normal” state you should go back to a regular branch:

`git switch <branch>` or `git checkout <branch>`

exercise 3

The crazy peak sorter script

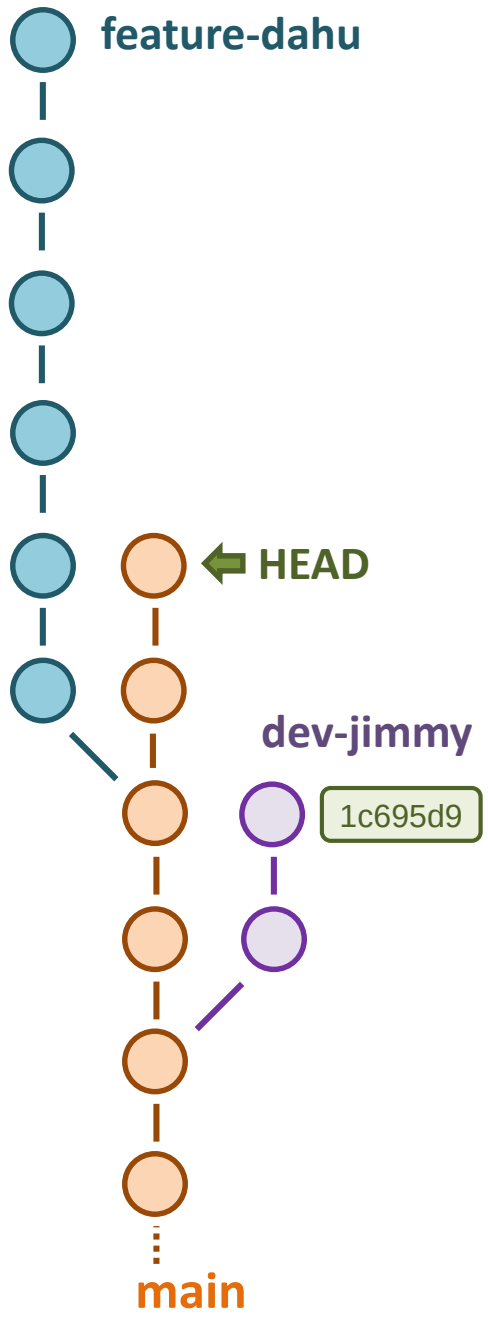
This exercise has helper slides

Exercise 3 help: history of the peak-sorter repo

This slide shows the history of the repo for exercise 3, both as the command line output and as a schematic representation (on the right).

This can help you understand the command line representation of a repo's history.

```
[rengler@local peak_sorter]$ git log --all --decorate --oneline --graph
* fc0b016 (origin/feature-dahu) peak_sorter: display highest peak at end of script
* d29958d peak_sorter: added authors as comment to script
* 6c0d087 peak_sorter: improved code commenting
* 89d201f peak_sorter: add Dahu observation counts to output table
* 9da30be README: add more explanation about the added Dahu counts
* 58e6152 Add Dahu count table
| * f6ceaac (HEAD -> master, origin/master, origin/HEAD) peak_sorter: added authors to script
| * f3d8e22 peak_sorter: display name of highest peak when script completes
|/
* cfd30ce Add gitignore file to ignore script output
* f8231ce Add README file to project
| * 1c695d9 (origin/dev-jimmy) peak_sorter: add check that input table has the ALTITUDE and PEAK columns
| * ff85686 Ran script and added output
|/
* 821bcf5 peak_sorter: add +x permission
* 40d5ad5 Add input table of peaks above 4000m in the Alps
* a3e9ea6 peak_sorter: add first version of peak sorter script
```



Working with **remotes**

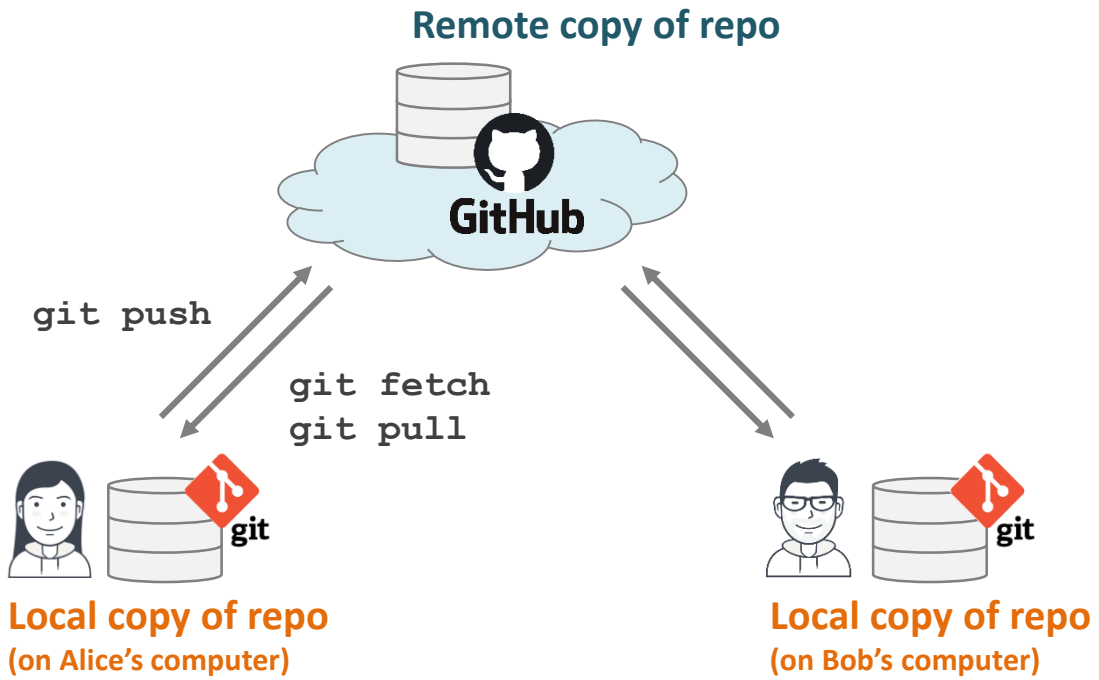
Linking your local repo with an online server

What is a “remote” ?

A **remote** is a copy of a Git repository that is stored on a server (i.e. online).

Remotes are very useful, as they allow you to:

- **Backup** your work.
- **Collaborate** and synchronize your repo with other team members.
- **Distribute** your work – i.e. let other people clone your repo (e.g. like the repo of this course).



Good to know:

- Each copy of a Git repo (local or online) is a **full copy of the entire repo’s history** (provided it has been synced).
- Git does not perform any automatic sync between the local and remote repos. All **sync operations must be manually triggered**.

Remotes are generally hosted on dedicated servers/services, such as GitHub, GitLab (either gitlab.com or a self-hosted instance), BitBucket, ...

Add a remote to an existing project (or update a remote's URL)

- **Case 1:** your local repo was cloned from a remote – *nothing to do* (the remote was automatically added by Git).
- **Case 2:** your local repo was created independently from the remote – it must be linked to it.

Add a new remote: `git remote add <remote name> <remote url>`

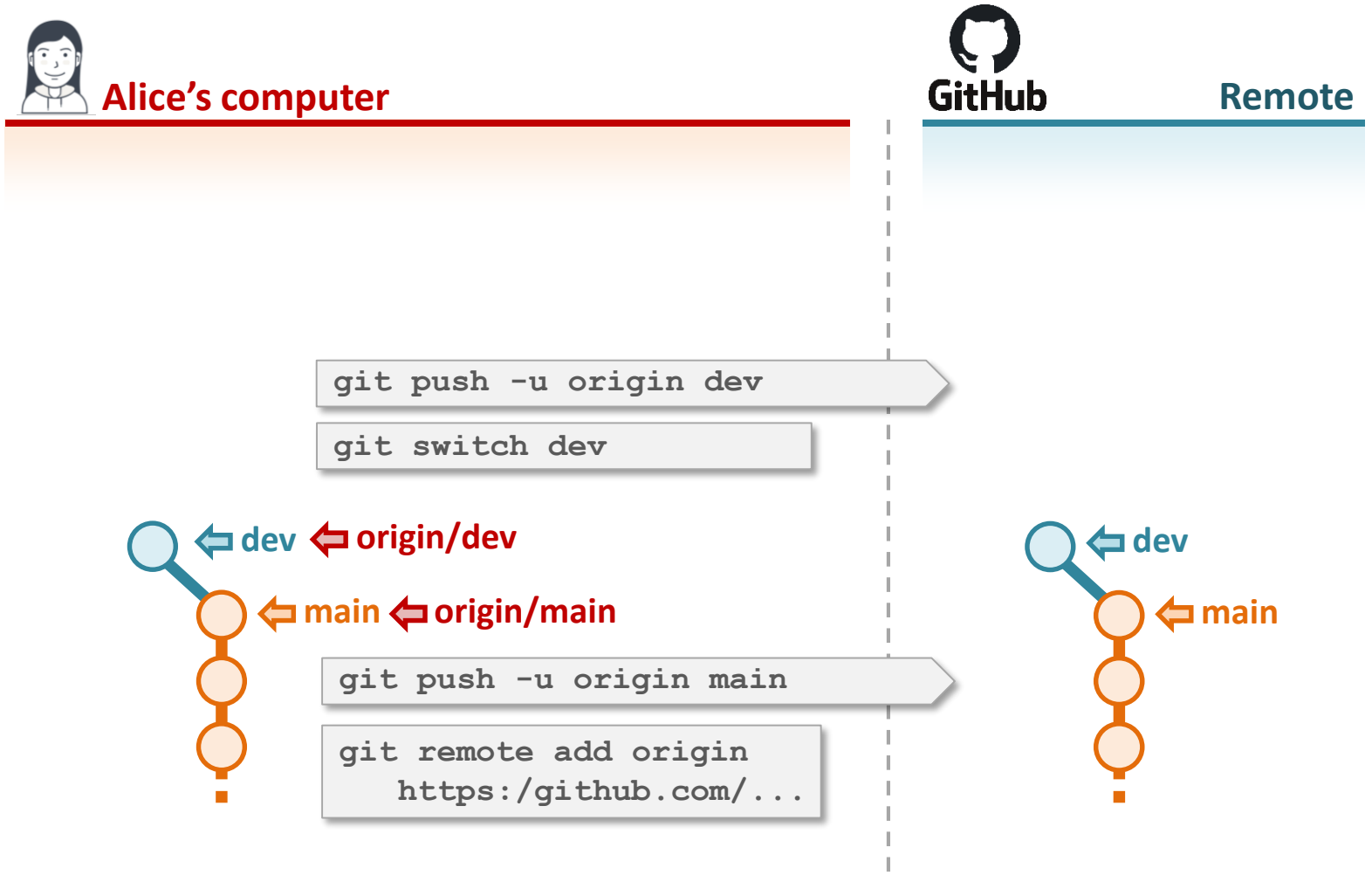
Change URL of remote: `git remote set-url <remote name> <remote url>`

Note: by convention, the `<remote name>` is generally set to `origin` .

Examples

```
# Add a new remote (named origin) to the local repo:
$ git remote add origin https://github.com/sibgit/test.git


# Update the URL of the existing origin remote.
# In this example, the remote was moved GitLab.
$ git remote set-url origin https://gitlab.sib.swiss/sibgit/test.git
```

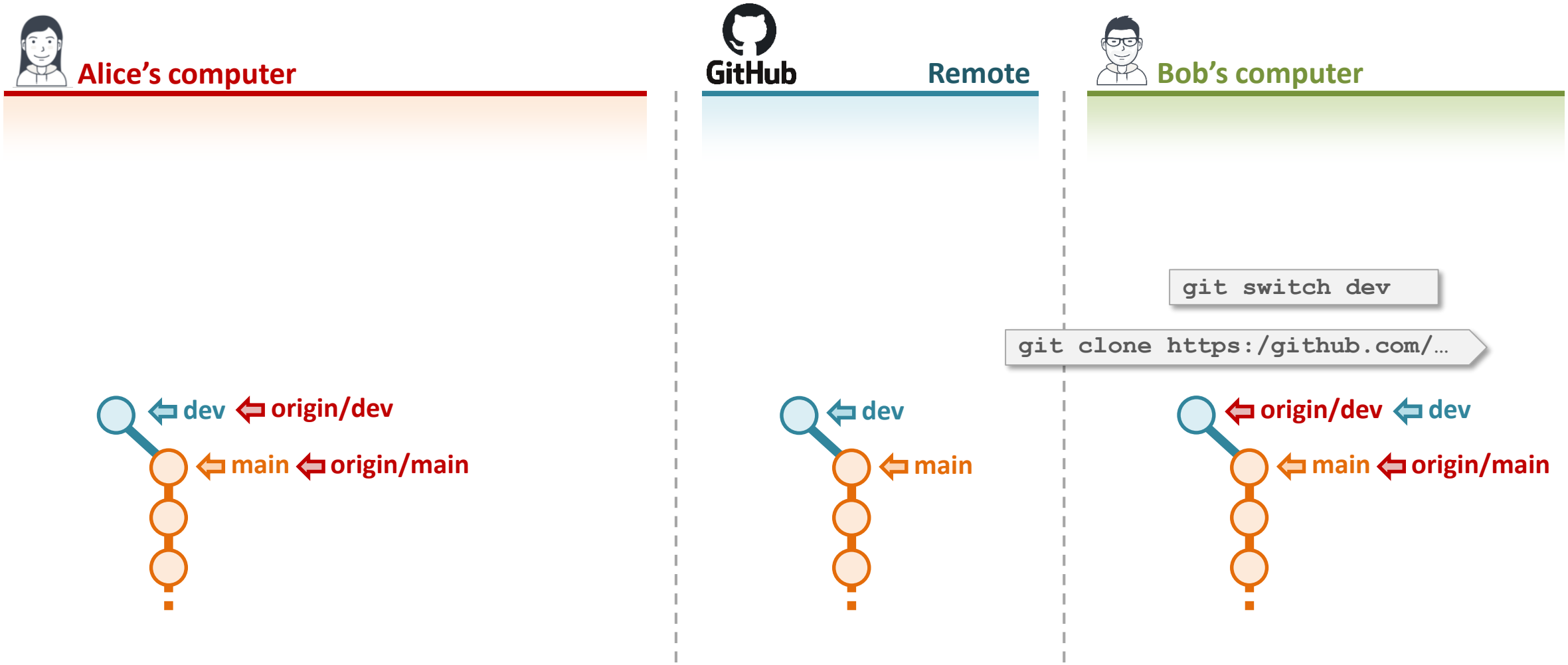
Example – part 1: creating a new remote and pushing new branches



Alice has a Git repo with 2 branches: **main** and **dev**. She now wants to store her work on GitHub, to collaborate and have a backup.

1. She creates a remote on GitHub and links it to her local repo using `git remote add origin <URL of remote>`
2. She pushes her branch **main** to the remote using `git push -u origin <branch name>`
   (at this point the branch has no upstream, so the `-u/--set-upstream` option must be used).
3. She pushes her branch **dev** to the remote.

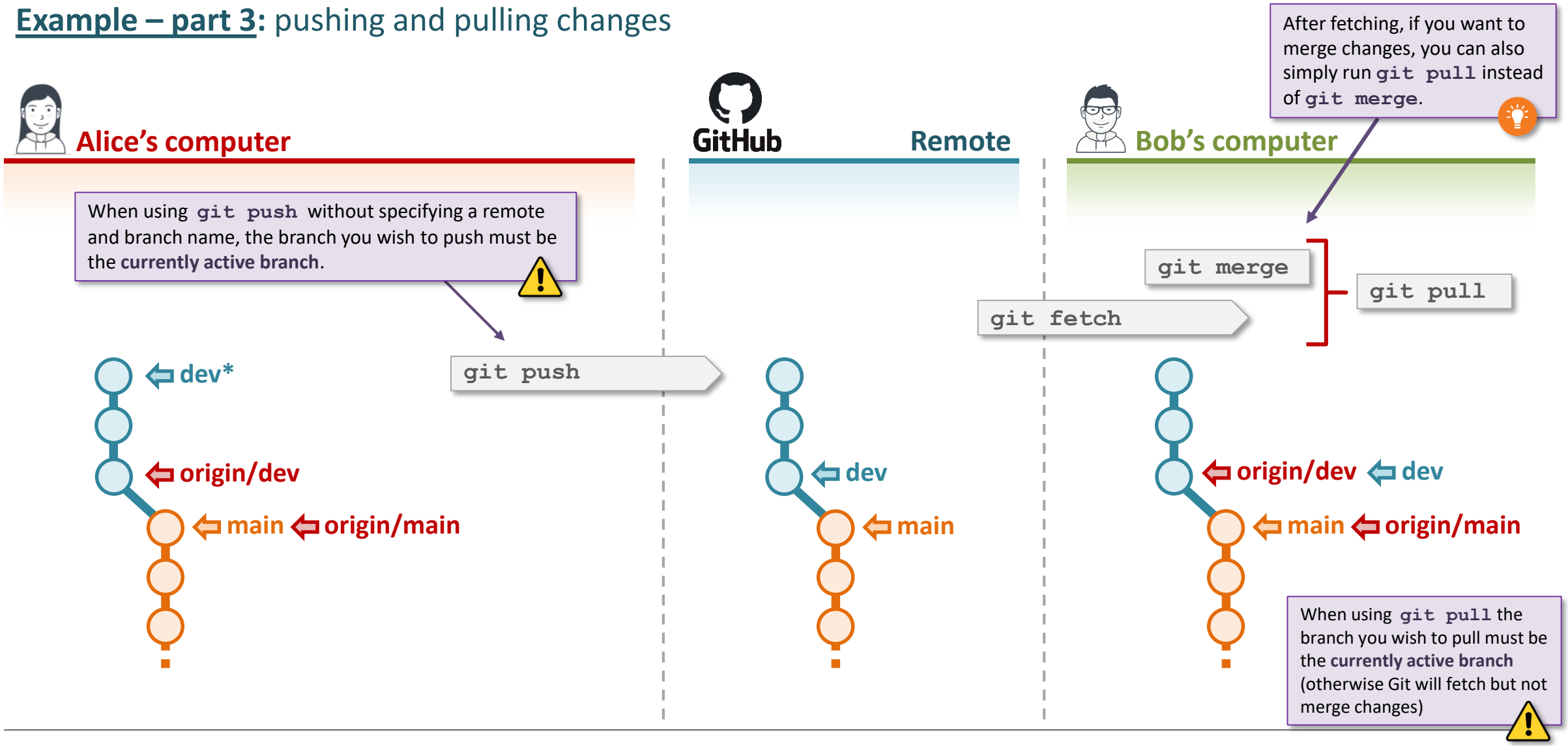
Example – part 2: cloning a remote and checking-out branches



Bob has now joined the team to work with Alice.

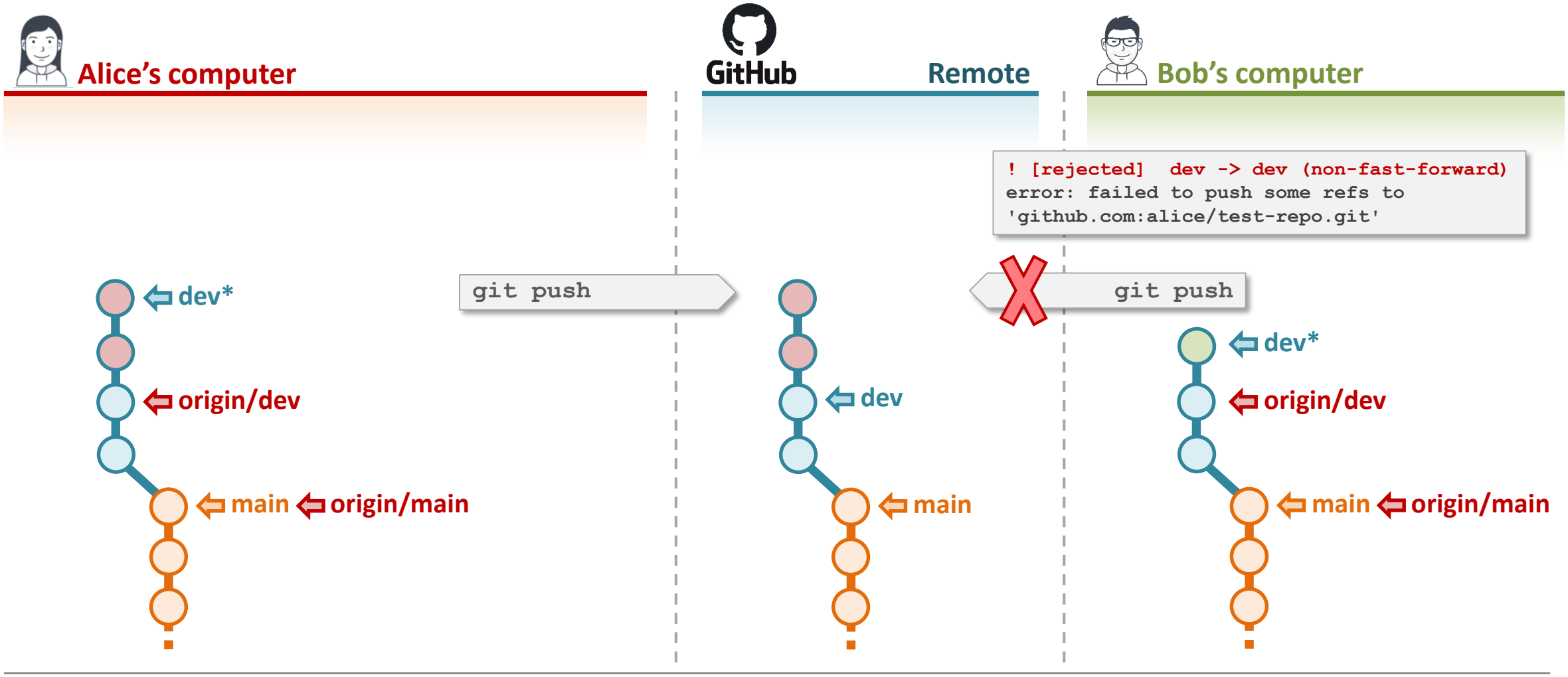
1. He **clones** the repo from GitHub using `git clone <URL of remote>`. At this point, Bob has no local **dev** branch - only a pointer to **origin/dev**.
2. Bob checks-out the **dev** branch to work on it. Because there is already a remote branch **origin/dev** present, Git automatically creates a new local branch **dev** with **origin/dev** as upstream (no need add the `--create/-c` option to `git switch`).

Example – part 3: pushing and pulling changes



1. In the mean time, Alice added 2 new commits to **dev**. She pushes her changes to the remote using `git push` (since her **dev** branch already has an upstream, there is no need to add the `-u/--set-upstream` option this time).
2. To get Alice's updates from the remote, Bob runs `git pull`, which is a combination of `git fetch` + `git merge`.
   Important: `git fetch` downloads all new changes/updates from the remote, but does not update your local branches.

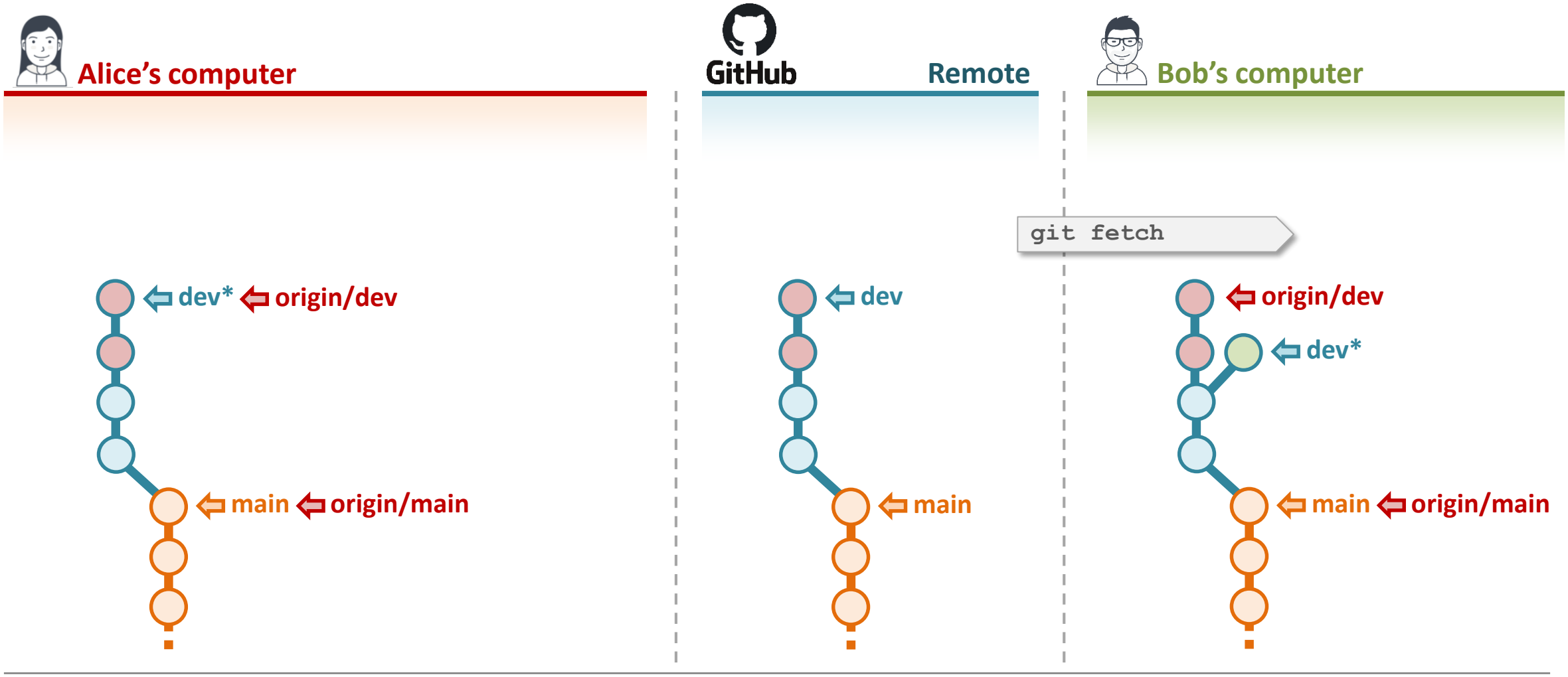
Example – part 4: reconciliation of a diverging history



Both Alice and Bob have now both added some commits to their local **dev** branch. As a result, **the history of their branches has diverged**.

1. Alice pushes her changes to the remote with `git push`, as usual.
2. When Bob tries to `git push`, his changes are **rejected** because the history between his local **dev** branch and the remote have **diverged**!

Example – part 4: reconciliation of a diverging history (continued)

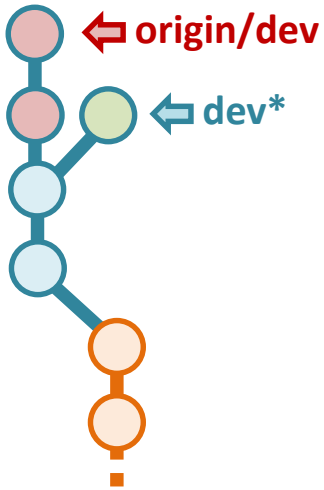


In order to be able to push his changes to the remote, Bob must first reconcile his local **dev** branch with the remote…

1. Bob starts by performing a `git fetch`, just to get the new commits from the remote and see how his local branch diverges from the remote (**important:** this operation does not impact/update his local **dev** branch).

Example – part 4: reconciliation of a diverging history (continued)

To reconcile his local **dev** branch with the remote, Bob must decide to either perform a merge or a rebase.



In this situation, a regular pull raises an error *

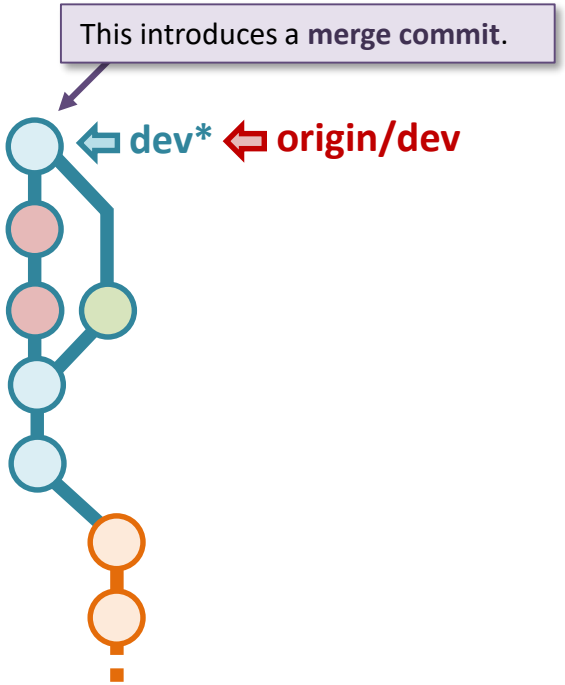
```
$ git pull
fatal: Need to specify how to
reconcile divergent branches
```

Option 1 - reconciliation using merge.

This is equivalent to:

```
git fetch
git merge origin/dev
```

`git pull --no-rebase`

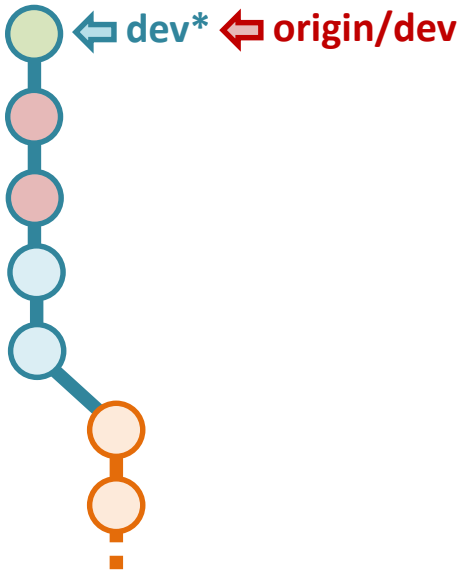


`git pull --rebase`

Option 2 - reconciliation using rebase.

This is equivalent to:

```
git fetch
git rebase origin/dev
```



If you don't remember the `--no-rebase` and `--rebase` options of `git pull`, simply `fetch` and then `merge` or `rebase` on **origin/dev**.

* On recent Git versions (>= 2.33), the default pull behavior is to abort if history diverged. On older versions, the default behavior is to merge (as in `git pull --no-rebase`).

`git pull`: a shortcut for fetch + merge

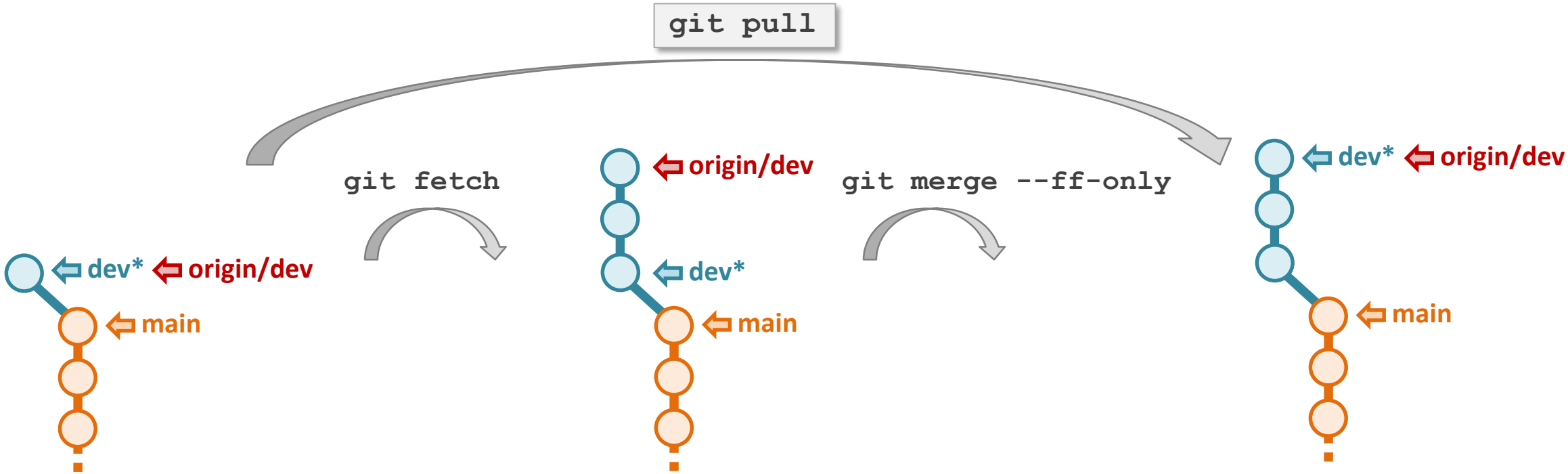
Having the **`git pull`** command use **`--ff-only`** as default merge option is a recent behavior (Git >= 2.33). In older versions, to force **`git pull`** to only allow fast-forward merges, the following option must be set:

```
git config --global pull.ff only
```

The `git pull` command is a shortcut for:

1. `git fetch` : fetches all updates from the remote.
2. `git merge –ff-only` : merge the currently active branch with its upstream branch (`origin/<branch>`).

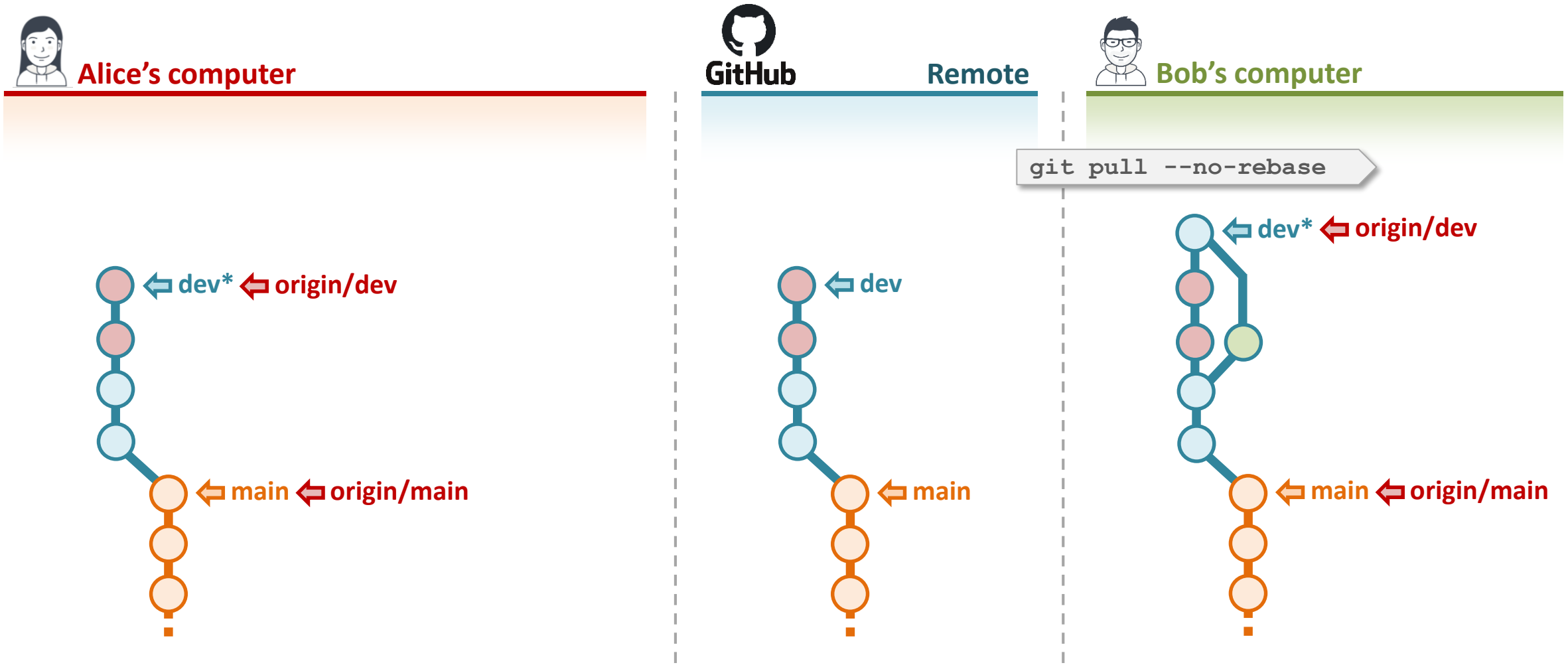
Fast-forward only -> any divergence in history will cause the command to fail and report an error.



By default, git merges a branch with its upstream branch, so `git merge` is the same as `git merge origin/<branch>` .

Supplementary material...

Example – part 4: reconciliation of a diverging history (continued)



Bob decides to merge without rebase and runs `git pull --no-rebase`.

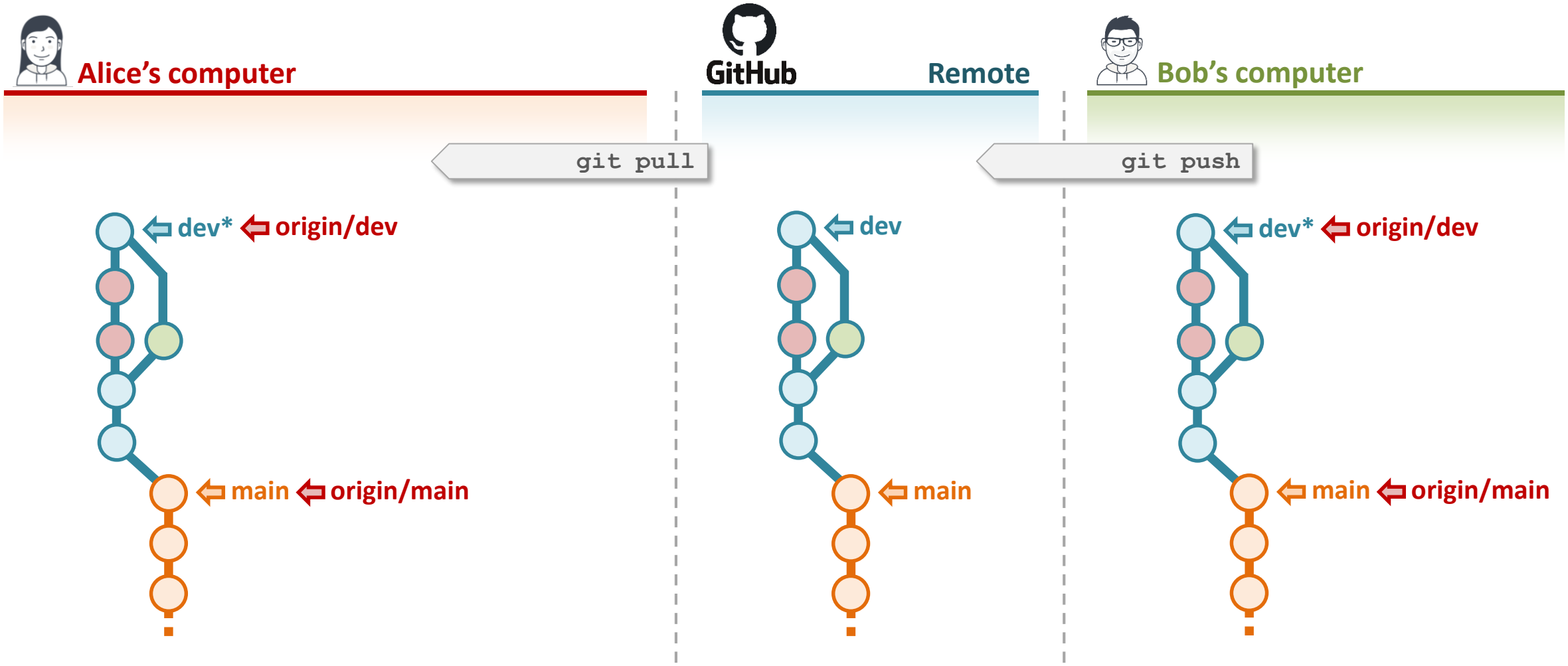
Note: depending on the version of Git, the default behavior of `git pull` is different:

- Newer versions default to `git pull --ff-only` (i.e. raise an error if a fast-forward merge is not possible)
- Older versions default to `git pull --no-rebase` (i.e. the automatically merge)

The default behavior can be modified in the git config.

```
git config pull.rebase false   # merge
git config pull.rebase true    # rebase
git config pull.ff only        # fast-forward only
```

Example – part 4: reconciliation of a diverging history (the end!)

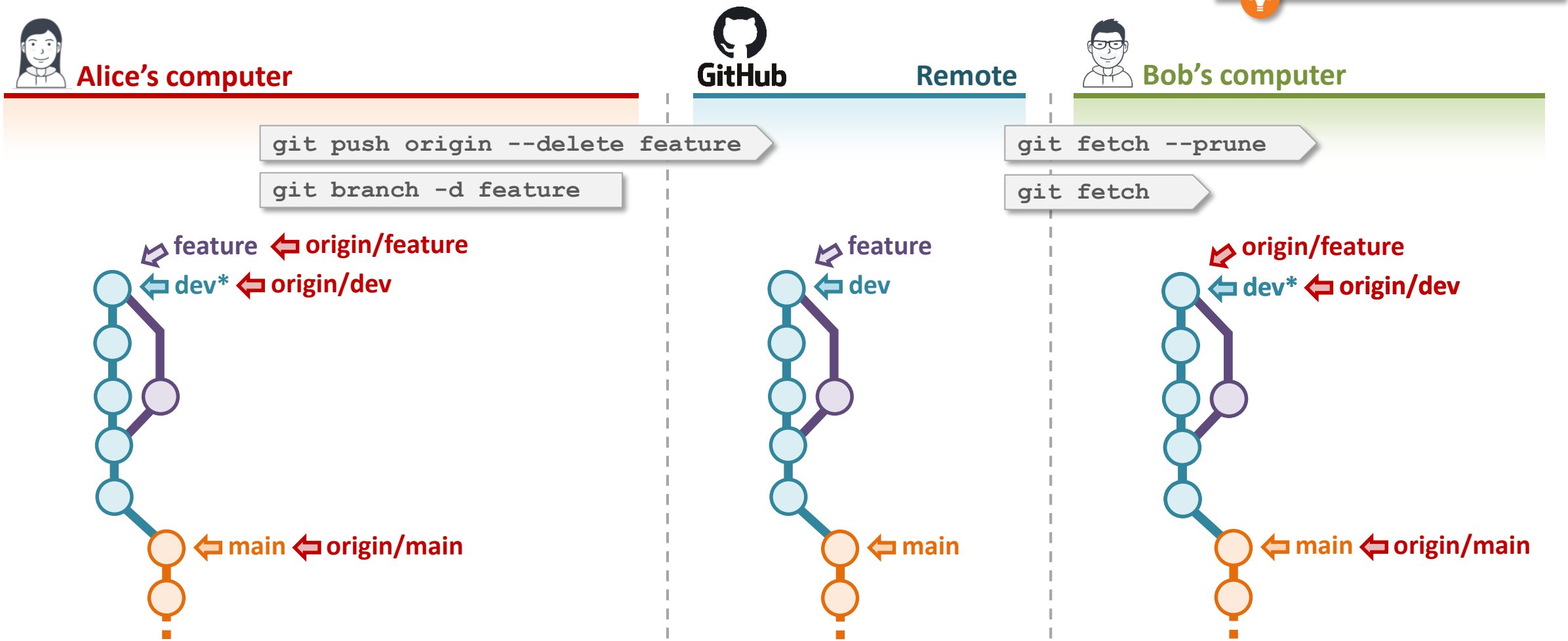


Finally, Bob can `git push` his changes to the remote - there are no more conflicts.

Alice can then `git pull` them.

Example – part 5: deleting branches on the remote

The `--prune` option also works with `git pull --prune`.

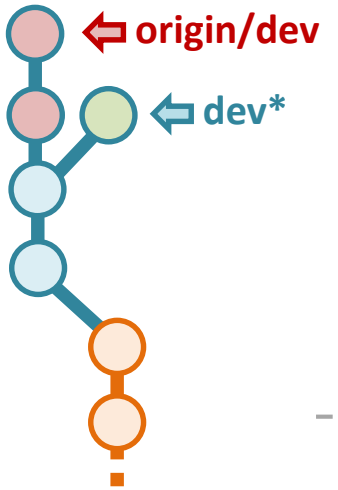


We are now at a later point in the development... Alice has just completed a new feature on her branch **feature**, and merged it into **dev**. She now wants to delete the **feature** branch both locally and on the remote.

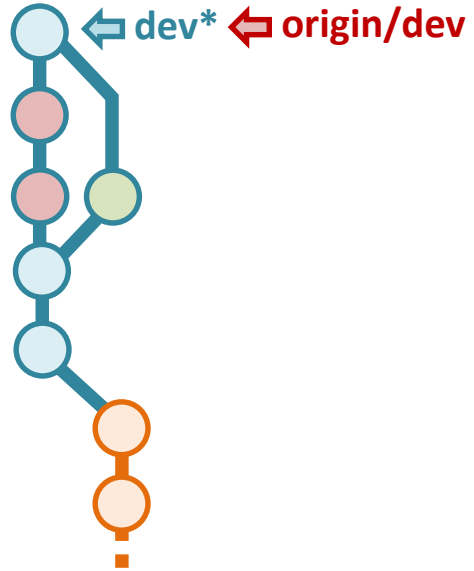
1. Alice deletes her local branch with `git branch -d <branch name>`.
2. Alice deletes the feature branch on the remote with `git push origin --delete <branch name>`. This also deletes her **origin/feature** pointer.
3. Bob runs `git fetch`, but this does not delete references to remote branches, even if they no longer exist on the remote.
4. To delete his local reference to the remote feature branch (**origin/feature**), Bob has to use `git fetch --prune`.

Example – part 6: overwrite history on the remote

Example, if you made some history-rewriting change locally, typically a rebase of a branch.

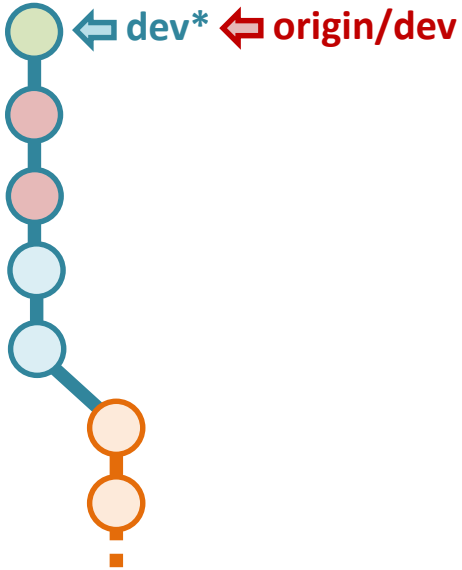


`git pull --no-rebase`

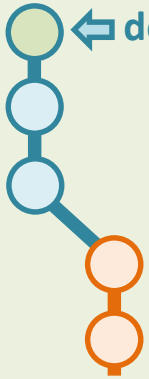


`git push --force`

`git pull --rebase`



Option 3 – overwrite the remote with `git push --force`



⚠ **This will permanently delete data on the remote !!**

Interacting with remotes: commands summary

Command	What it does	Where to run and comments
`git push`	**push new commits** on the current branch to the remote.	**Run on the branch that you wish to push.** (only changes on the active branch are pushed)
`git push -u origin <branch-name>`	Same as git push, but additionally sets the upstream branch to `origin/branch-name`. Only needed if no upstream is set.	`-u` option is only needed when pushing a branch to the remote for the very first time. It is not needed if you initially created the local branch from a remote branch.
`git push origin <branch-name>`	Push new commits on the specified branch to the remote.	When the remote (here `origin`) and branch names are specified, the push command **can be run from anywhere**.
`git push --force`	**Overwrite the branch on the remote** with the local version.	**Warning:** this **deletes data** on the remote!
`git fetch`	**Download all updates** from the remote to your local repo (even for non-active branches or branches for which there is no local version). Does not update your local branch pointer to `origin/branch-name`.	**Can be run from any branch.**
`git pull`	**Download all updates** and **merge changes** the upstream `origin/branch-name` into the active branch (i.e. update the active branch to its version on the remote).	**Run on the branch that you wish to update.** `git pull` is a shortcut for `git fetch + git merge origin/branch-name`
`git pull --no-rebase`	Fetch + 3-way merge active branch with its upstream `origin/branch-name`.	
`git pull --rebase`	Fetch + rebase active branch on its upstream `origin/branch-name`.	
`git pull --ff-only`	Fetch + fast-forward merge active branch with its upstream `origin/branch-name`. If a fast-forward merge is not possible, an error is generated.	

On recent versions of Git (>= 2.33), the default pull behavior is to abort the pull if a branch and its upstream are diverging.
On older versions, the default behavior is to merge them (same as `git pull --no-rebase`).

Interacting with remotes: commands summary

Command	What it does
`git clone <URL>`	**Create a local copy** from an existing online repo. Git automatically adds the online repo as a remote.
`git remote add origin <remote url>`	**Add a new remote** to an existing local repo.
`git remote set-url origin <remote url>`	**Change/update the URL of a remote** associated to a local repo.
`git remote -v`	**Display the remote(s)** associated to a repo.

By convention, the `<remote name>` is generally set to `origin`, but it could be anything.

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

The fetch and push URLs should be the same.
To use different URLs (different remotes) for push and fetch, add two different remotes.

`git branch -vva` — **List branches of repo and their associated upstream** (if any).

```
$ git branch -vva
  manta-dev 18d8de0 [origin/manta-dev] manta ray: add animal name
  main      6c8d731 [origin/main] Merge pull request #44 from sibgit/dahu-dev
* sunfish   18d8de0 manta ray: add animal name
```

We can see that the branches **main** and **manta-dev** have an upstream branch. The **sunfish** branch does not.

GitHub

collaborate and share your work

GitHub – an online home for Git repositories

- GitHub [github.com] is a hosting platform for Git repositories.
- 73+ million users, 200+ million repositories (as of 2022).
- Very popular to share/distribute open source software.
- Allows to host public (anybody can access) and private (restricted access) repos.
- Hosting of projects is free, with some paid features.
- Popular alternatives include:
  - **GitLab** [gitlab.com], which can also be installed as a local instance.
  - **BitBucket** [bitbucket.org].

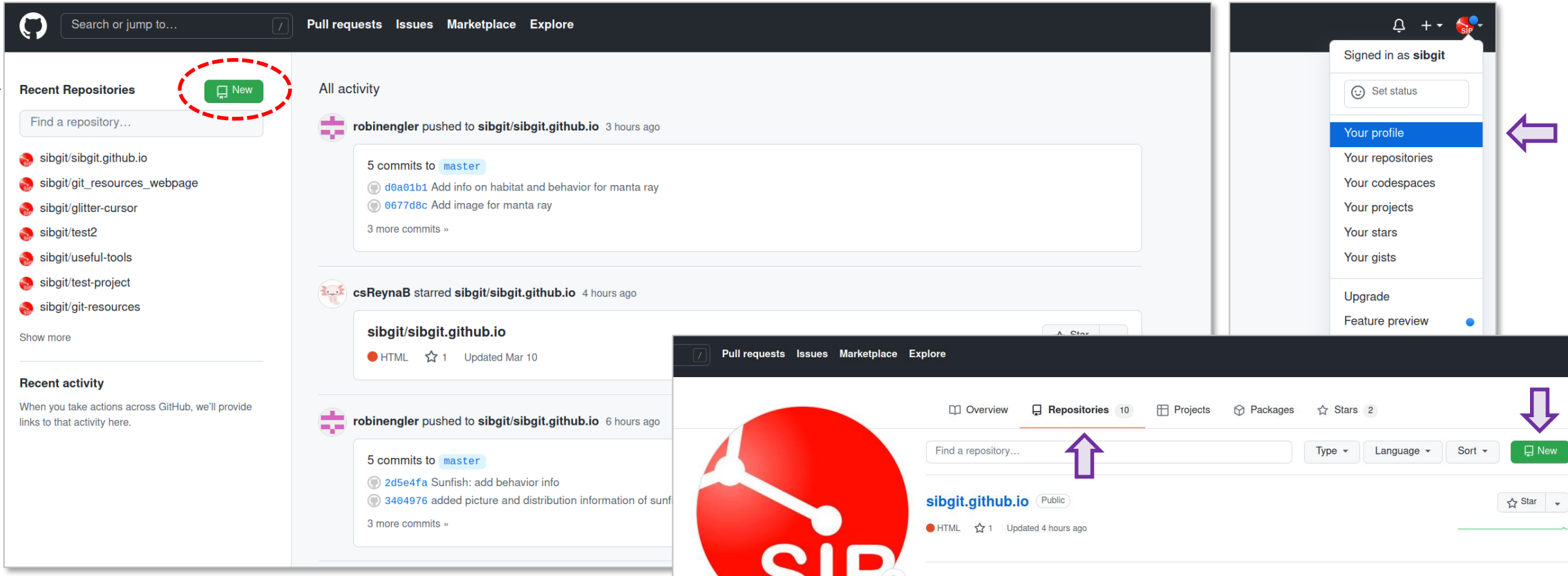
Creating a new project on GitHub

You will need to do this in tomorrow's exercise 4 !

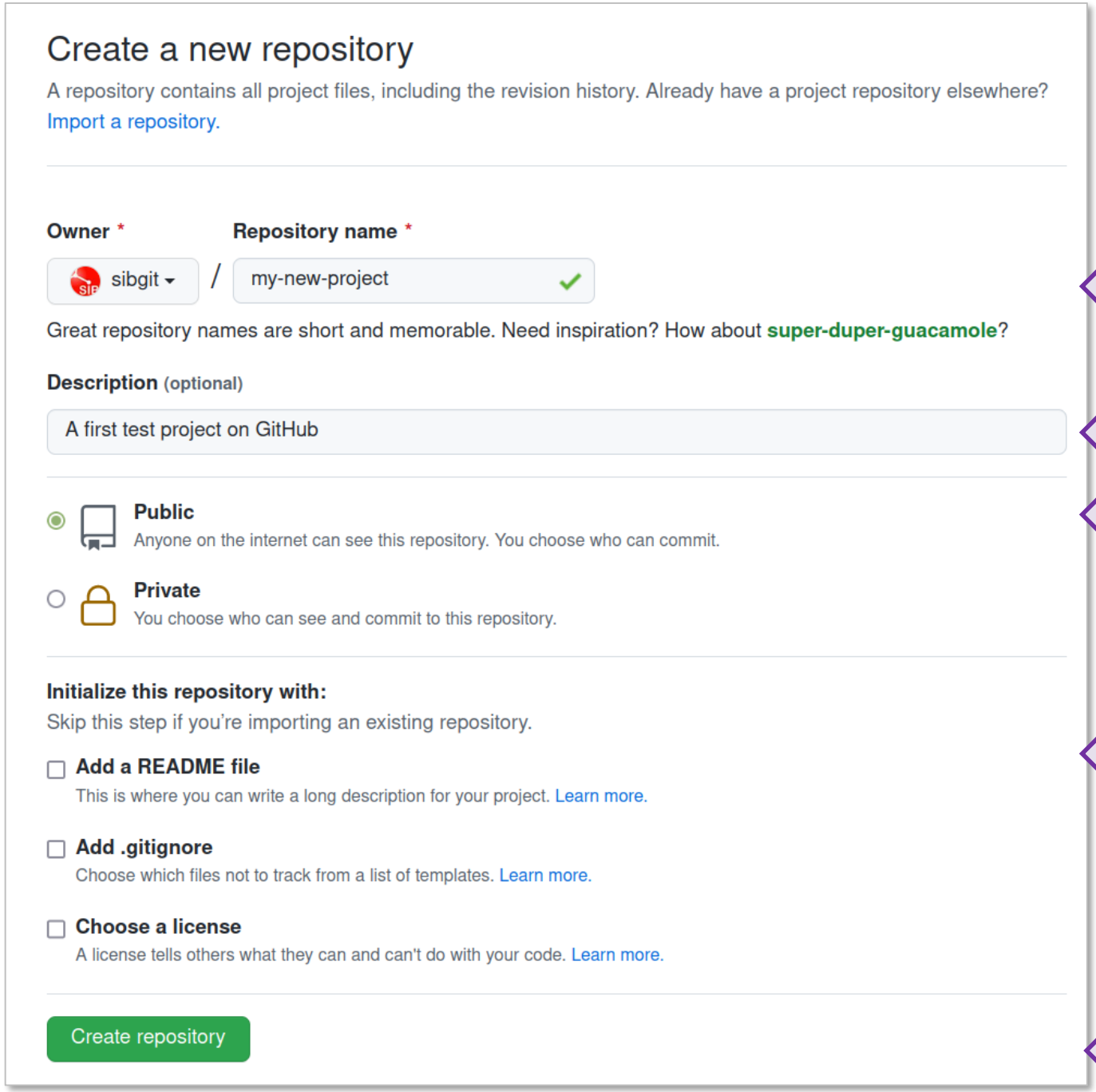
To create a new repo, click on 

… or click on your user icon (top right), then **Your profile** …

Either on the welcome screen at https://github.com (after signing-in)…



… and the **Repositories** tab.



Enter a **name** for your new repository.

Project **description**.

Select whether your repo should be:

- **Public** - anyone can access it (read from it).
- **Private** - only people you authorize.

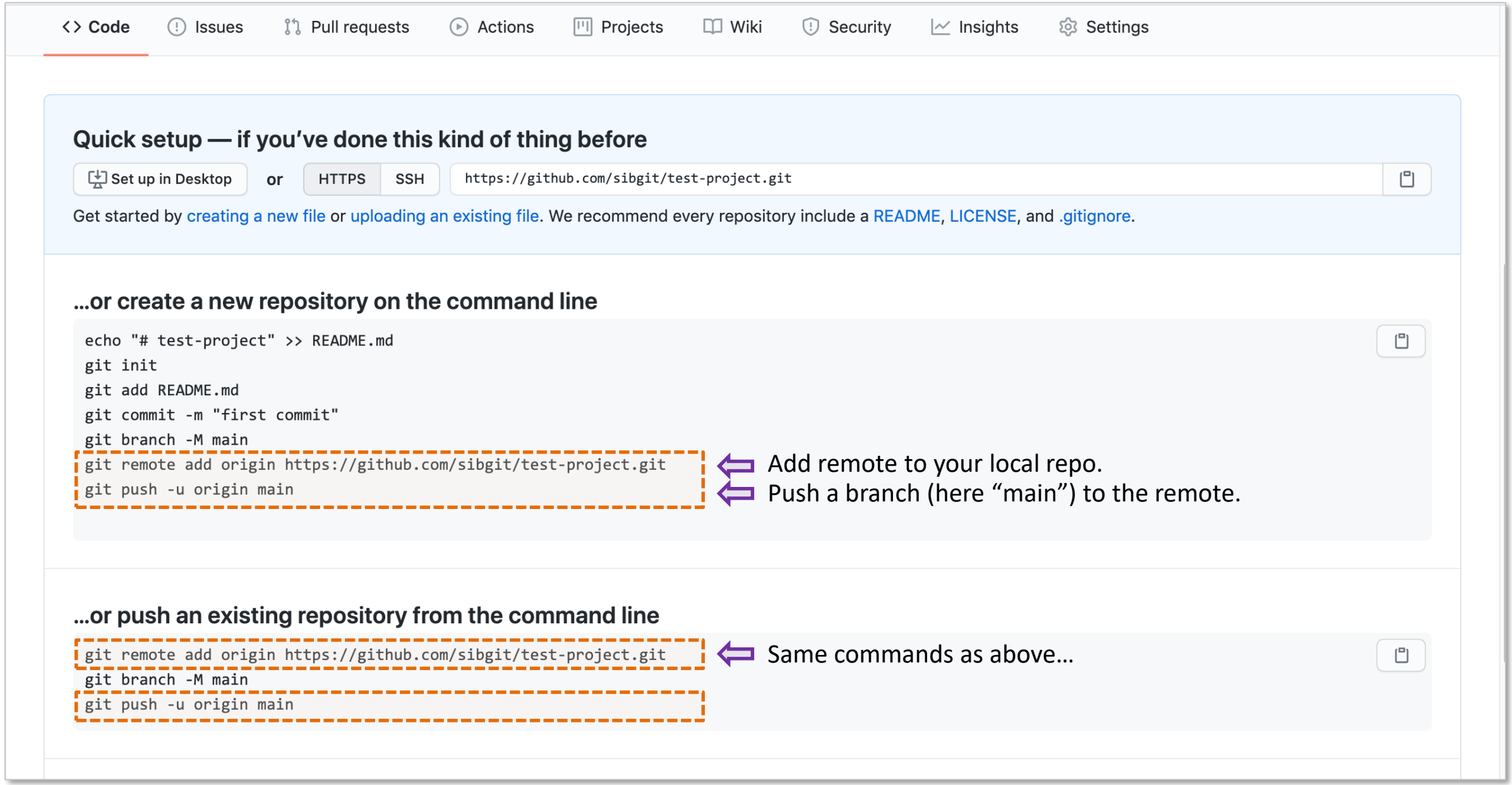
Note: even if a repo is public, only authorized members can push changes to it.

Pre-fill the repository with some files (don't do this if you already have a local repo you want to push):

- **README** – A text file that is displayed on the homepage of your repo (with markdown rendering).
- A **.gitignore** file selected from a list of templates.
- A **license** file selected from a set of standard licenses (e.g. GPL, MIT, …).

Click **Create repository**.

The **home page** of an empty repository provides instructions to get started…



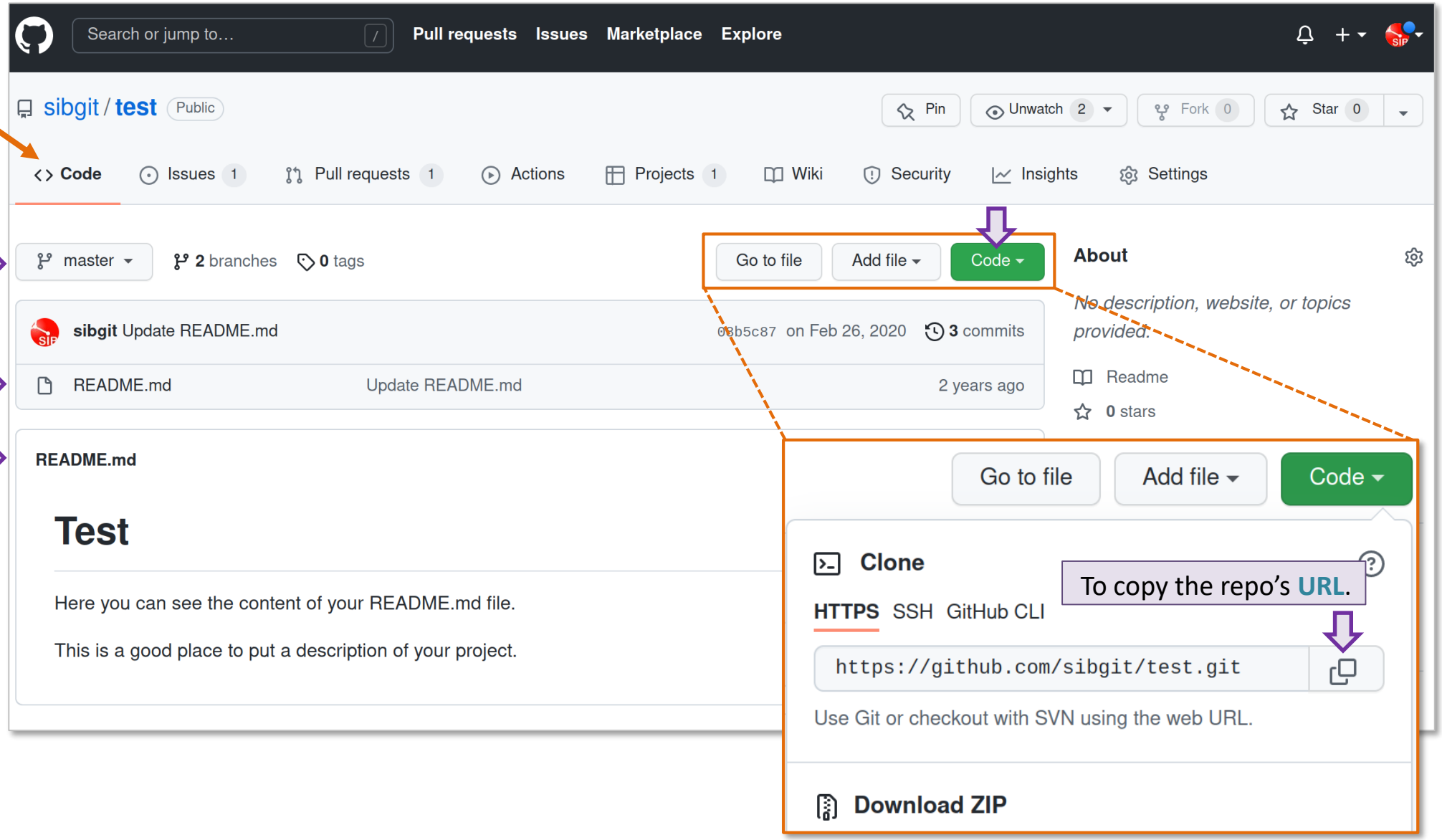
When at least 1 file is present in the repo, the **home page** of your Git repo looks like this:

Code tab: the "home" page of your repo.

Branch you are currently viewing

List of files present in the repo.

If you have a **README.md** file, it is displayed here (with markdown rendering).

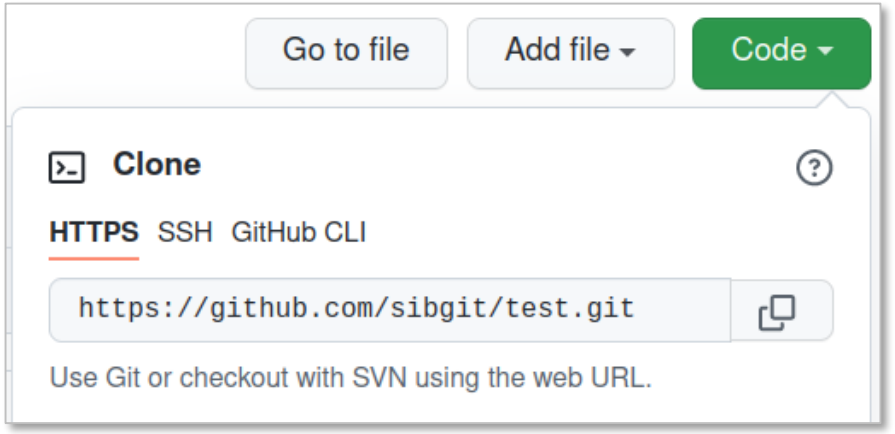


Cloning a repo: HTTPS vs. SSH

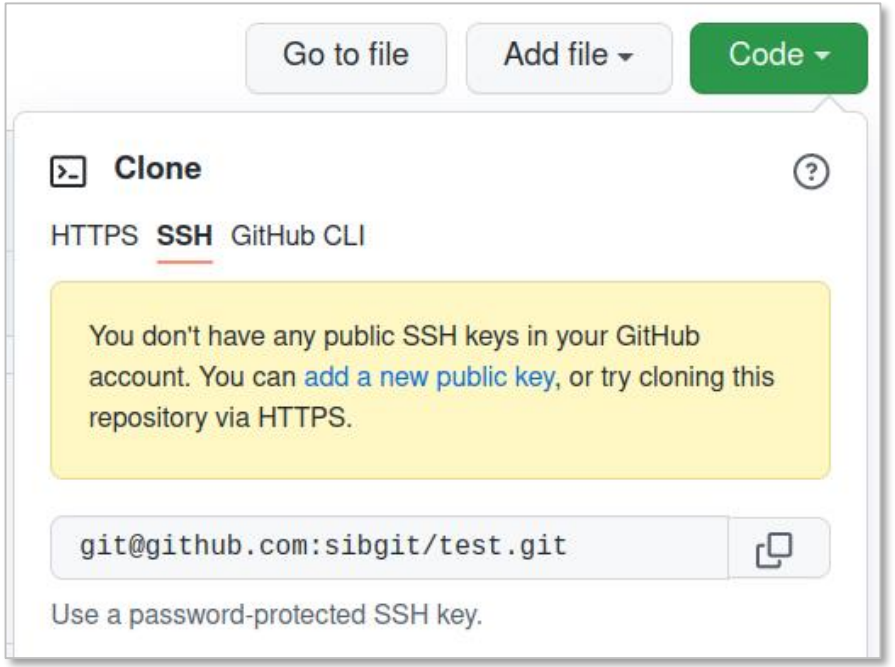
HTTPS and SSH are two different network protocols that machines can use to communicate.

When cloning (or adding a remote) via:

- **HTTPS**, you will need to provide a **personal access token** (PAT) as authentication credential.
  - If the repo is public, credentials are only needed to push data to the remote (not to pull).
  - Your local Git repo will in principle store the login credentials, so you need to provide them only once.
  - Instructions on **how to generate a PAT** can be found in the *helper slides of exercise 4*.



- **SSH**, you will need to add your **public SSH key** to your GitHub account.



Reminder: command to clone a repo (here via https)

```
$ git clone https://github.com/sibgit/test.git
```

Supplementary material...

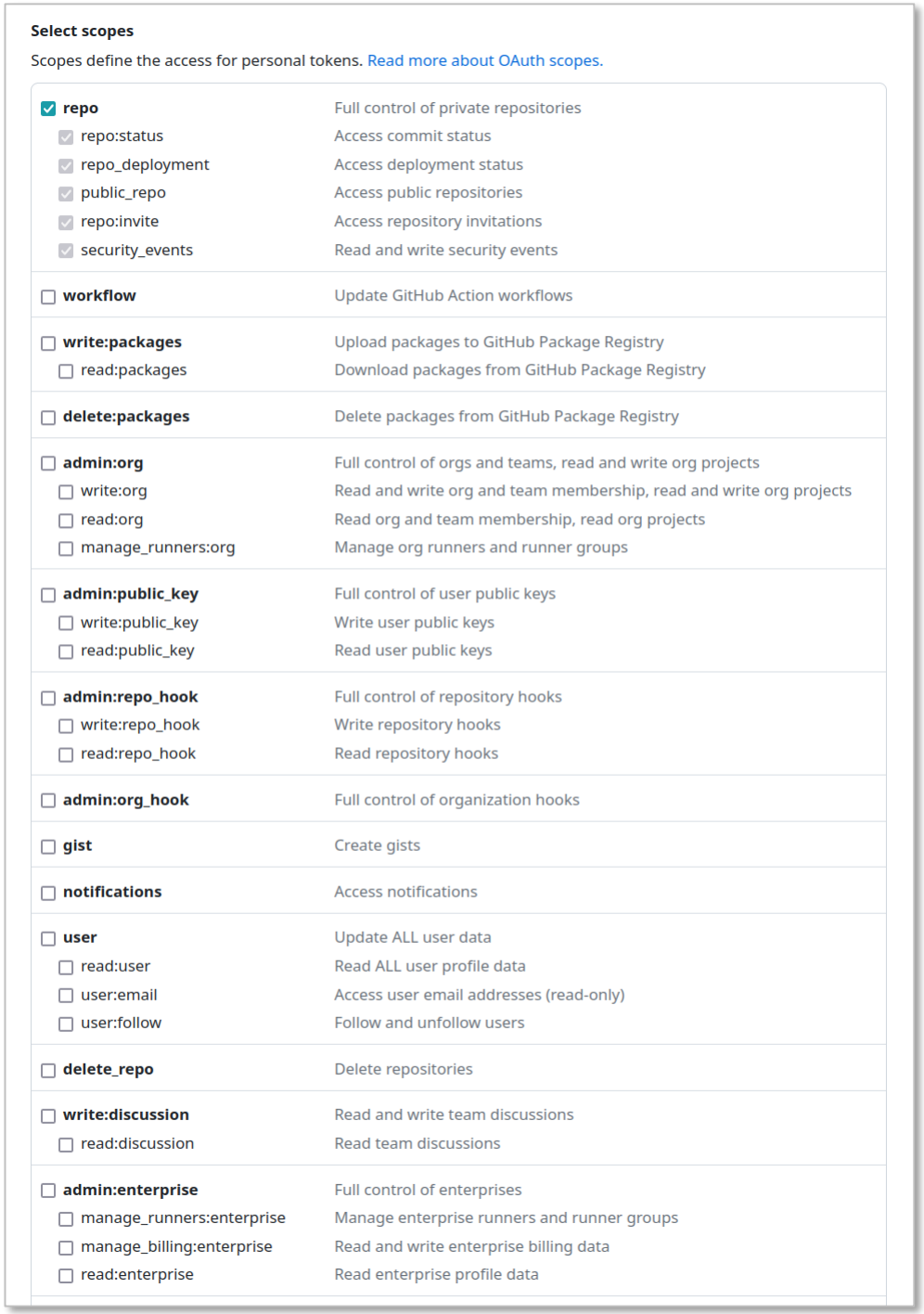
Personal access tokens (PAT) on GitHub

Pushing data to a remote requires **some form of authentication**…
… otherwise anyone could push anything to your remotes!

For security reasons, GitHub does not allow using your user name and password for authentication when running a git push command. Instead you need to use a **personal access token (PAT)**.

In **exercise 4** you will need to push commits to GitHub.

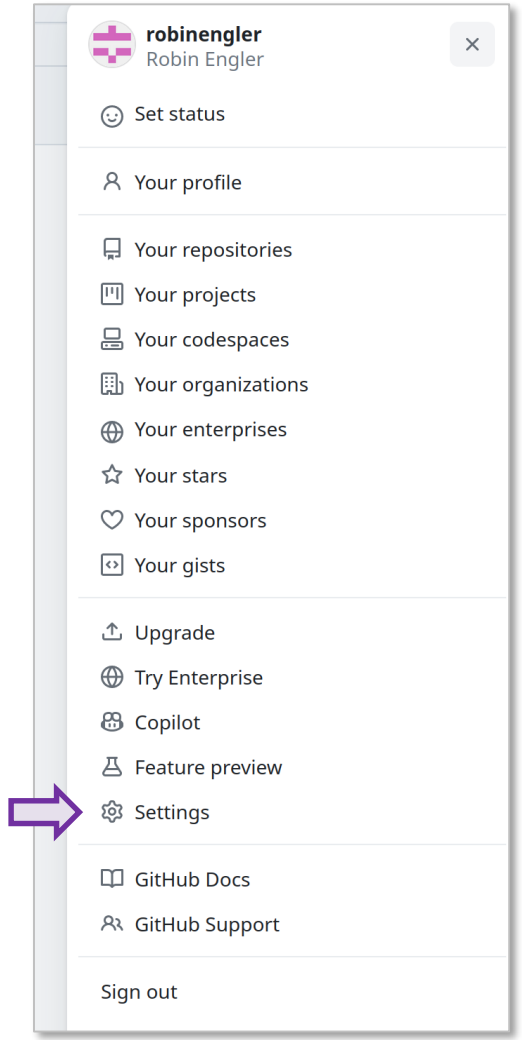
Let's generate a PAT together now…



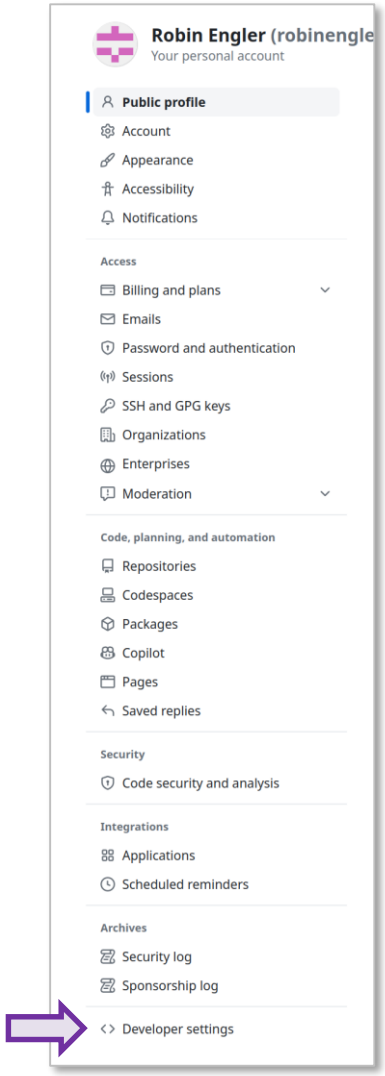
Generating a “personal access token” on GitHub

In order to push data (commits) to GitHub, you will need a **personal access token (PAT)**.

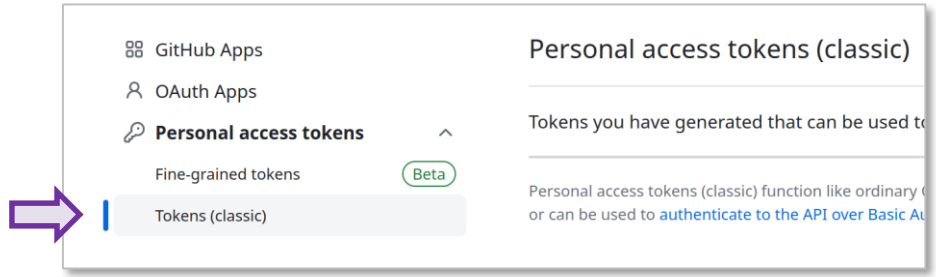
1. In your user profile (top right), click on **Settings**.



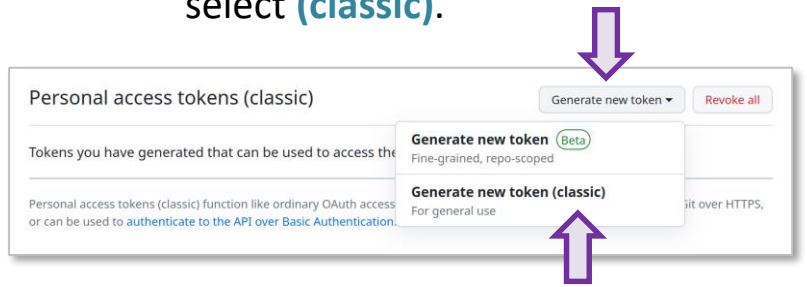
2. In your Account settings, click on **Developer settings** (at the very bottom of the list)



3. In **Developer settings**, click on **Personal access tokens**, and select **Tokens (classic)**.



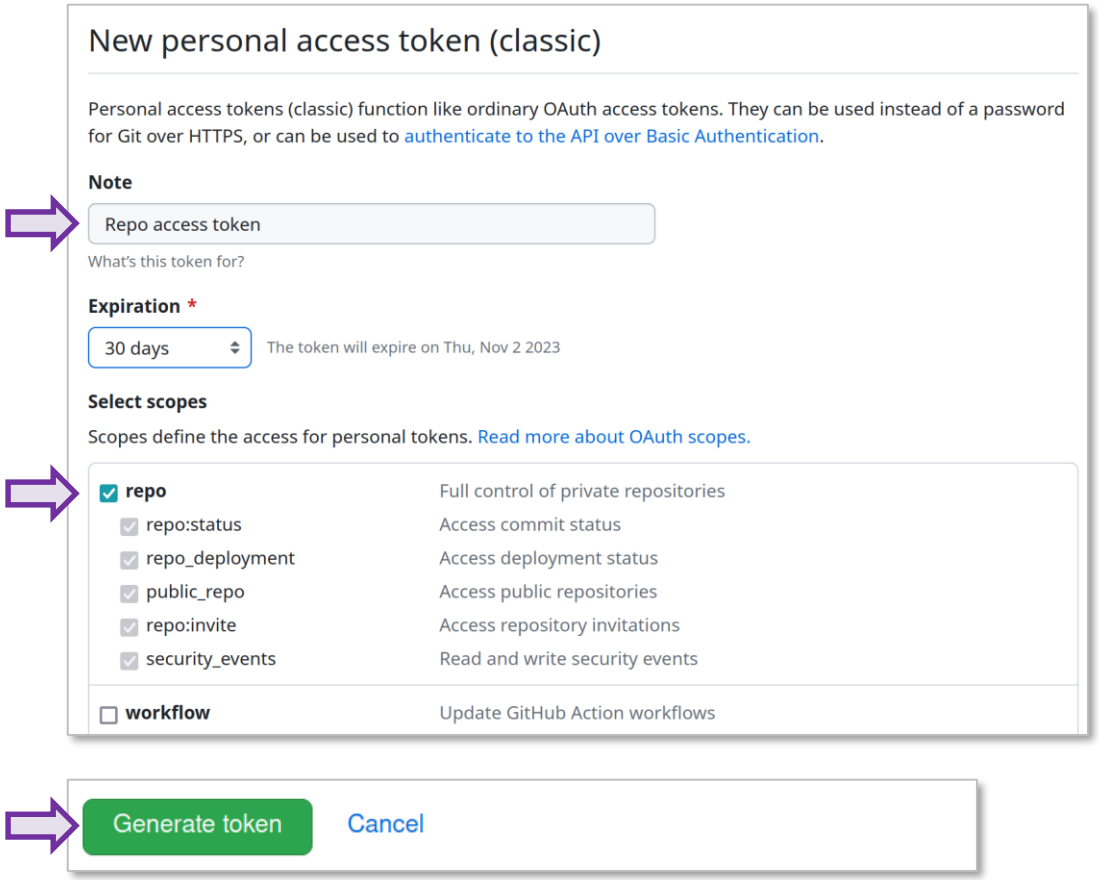
4. Click on **Generate new token**, and select **(classic)**.



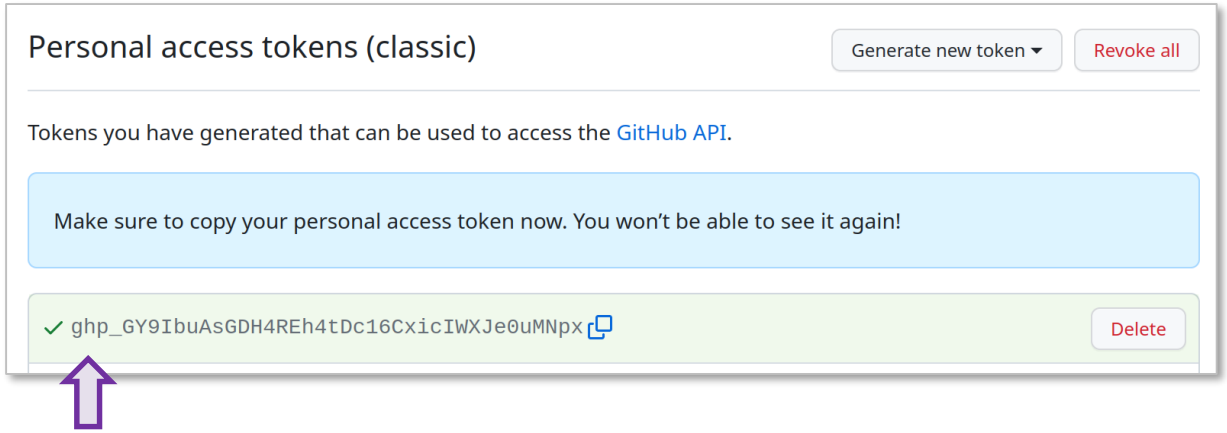
Go to next page

Generating a “personal access token” on GitHub (continued)

5. Add a **Note** (description) to your token and select the **repo** scope checkbox. The click **Generate token**.



6. **Copy the personal access token** to a safe locations (ideally in a password manager). You will **not be able to access it again** later.



7. When you will push content to GitHub for the first time in the project, you will be asked for your user name and password. **Instead of the password**, enter the **personal access token** you just created.

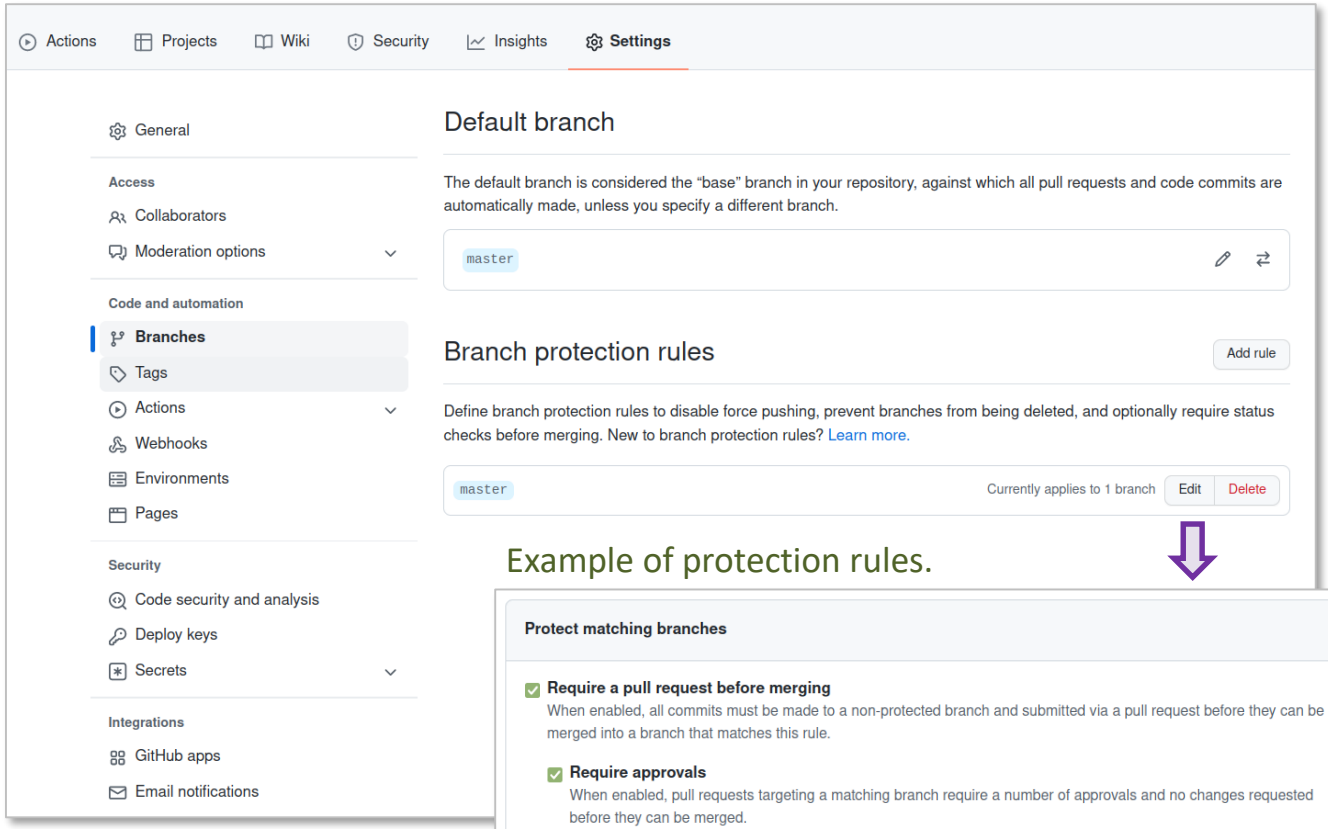
GitHub Pull Requests (PR)

Pull Requests * (PR) are a way to ask someone to integrate your changes (i.e. merge your branch) into another branch.

- PRs perform a branch merge operation on the GitHub remote (rather than on your local copy).
- Typically, a PR is created to merge a feature branch into the *main* branch on the remote.

Why use PRs instead of a local merge (and push)?

- The branch you want to merge into (e.g. main/master) is **protected** **.
- Gives the opportunity to the repository owner(s) to **review changes** before merging them.
- Makes it easy to merge changes from a **forked** *** repository.



Example of protection rules.

* On **GitLab**, pull requests are called **Merge Requests** (MR), but it's the exact same thing.

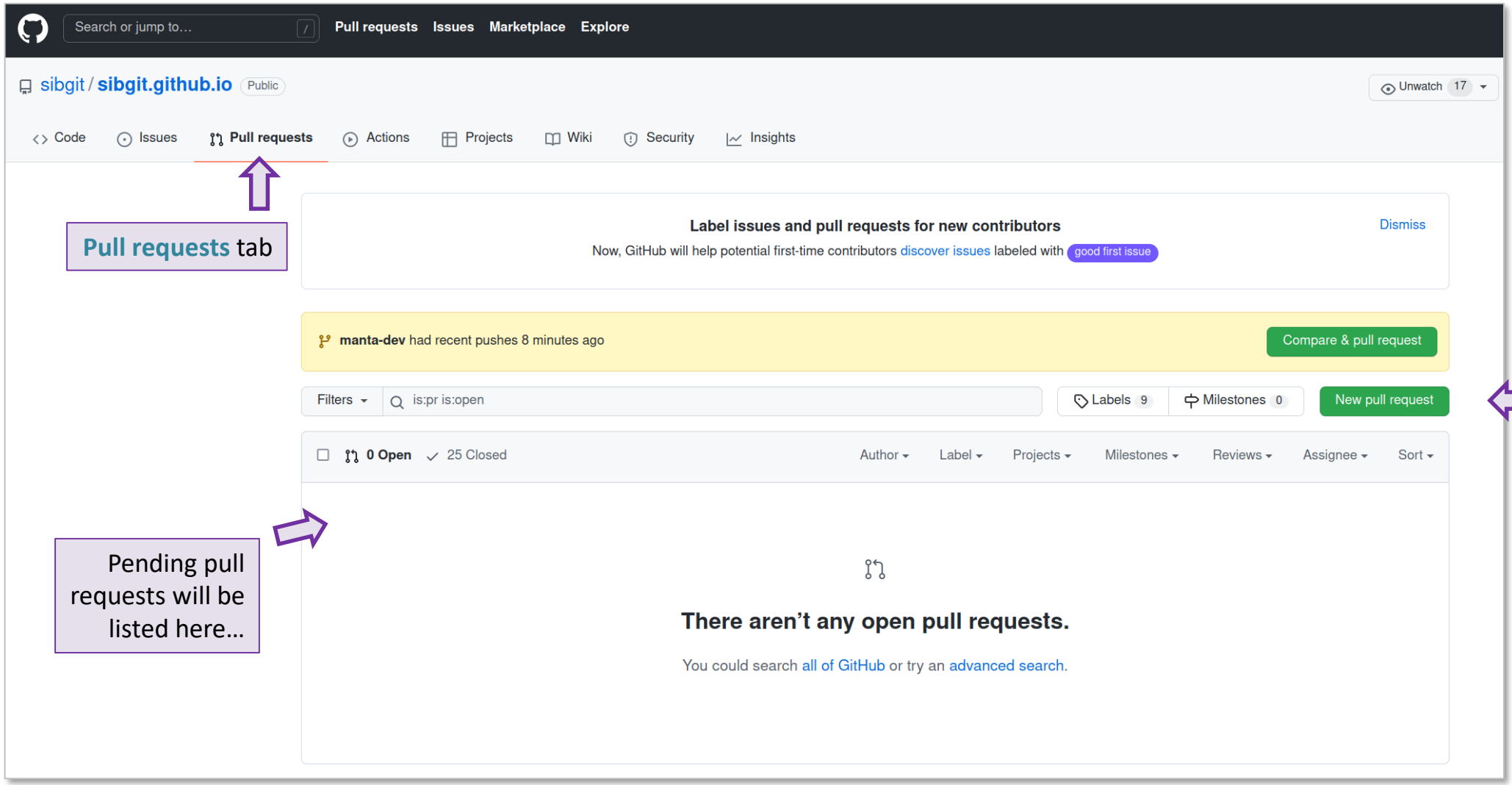
** **Protected** branches are branches where push operations are limited to users with enough privileges.

*** A **fork** is a copy of an entire repository under a new ownership.

How to open a Pull Request on GitHub: step-by-step

You will need to do this in exercise 4 !

1. On the project's page on GitHub, go to the **Pull requests** tab.

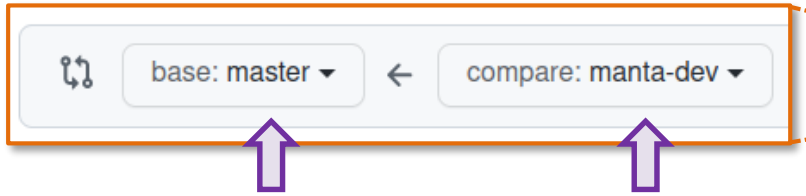


Pull requests tab

Pending pull requests will be listed here…

2. Click on **New pull request**.

3. Select the branches to merge:



Branch to merge into

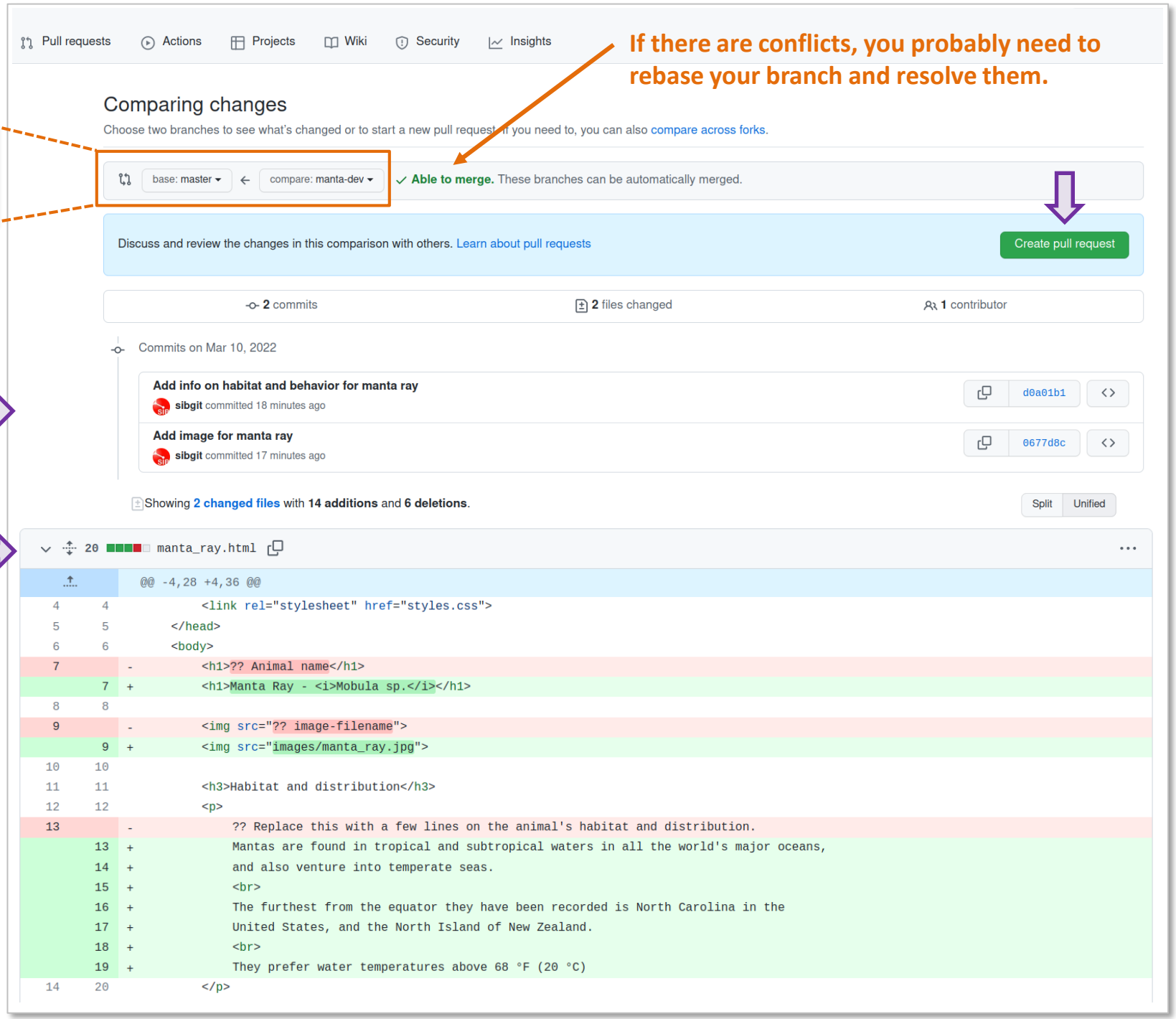
Branch to merge (your contribution)

List of commits that will be merged

In this example, there are 2 commits on branch “manta-dev” that will be merged into “master”.

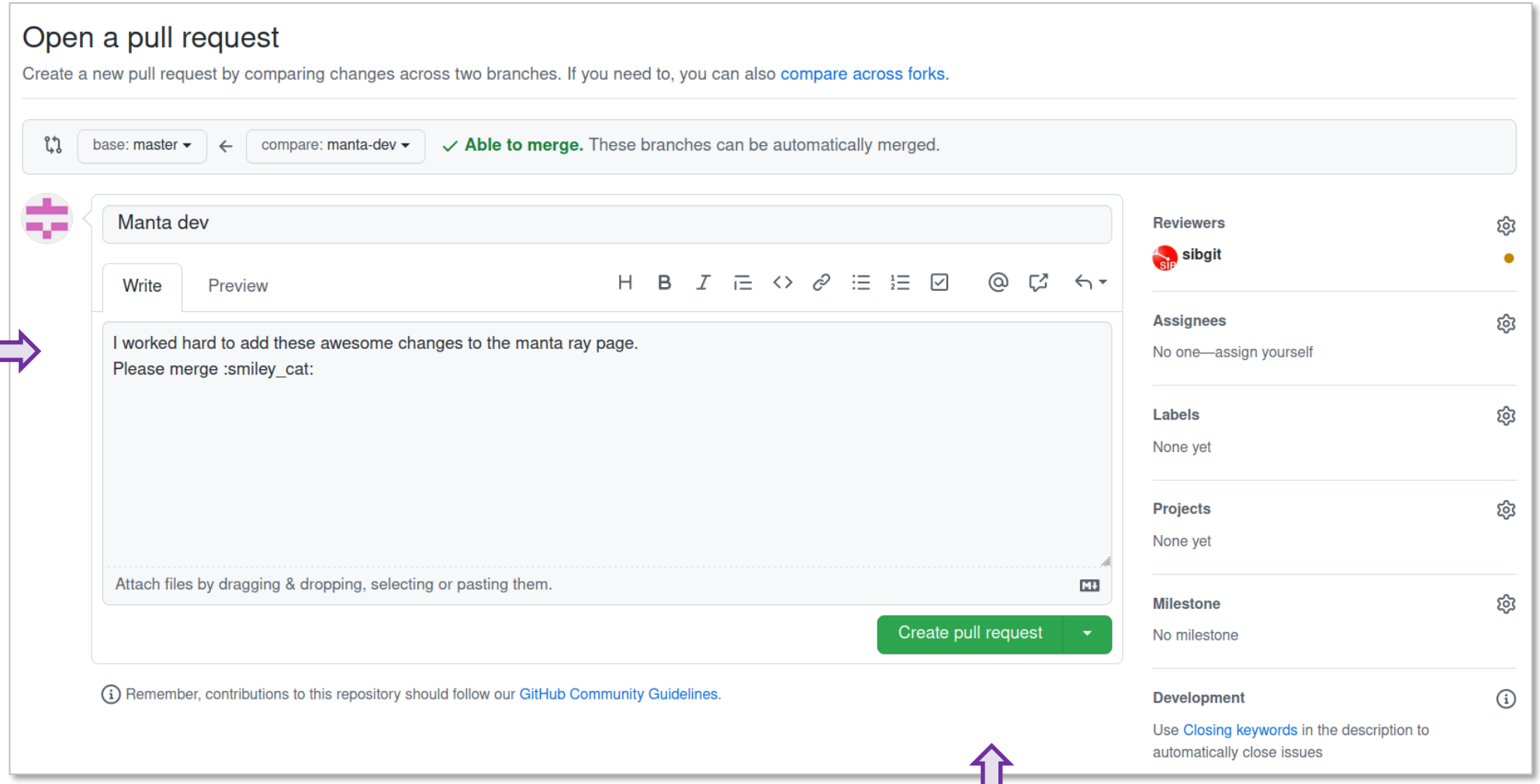
Summary of changes introduced by the pull request.

Green lines = new content.
Red lines = deleted content.



4. Click on Create pull request.

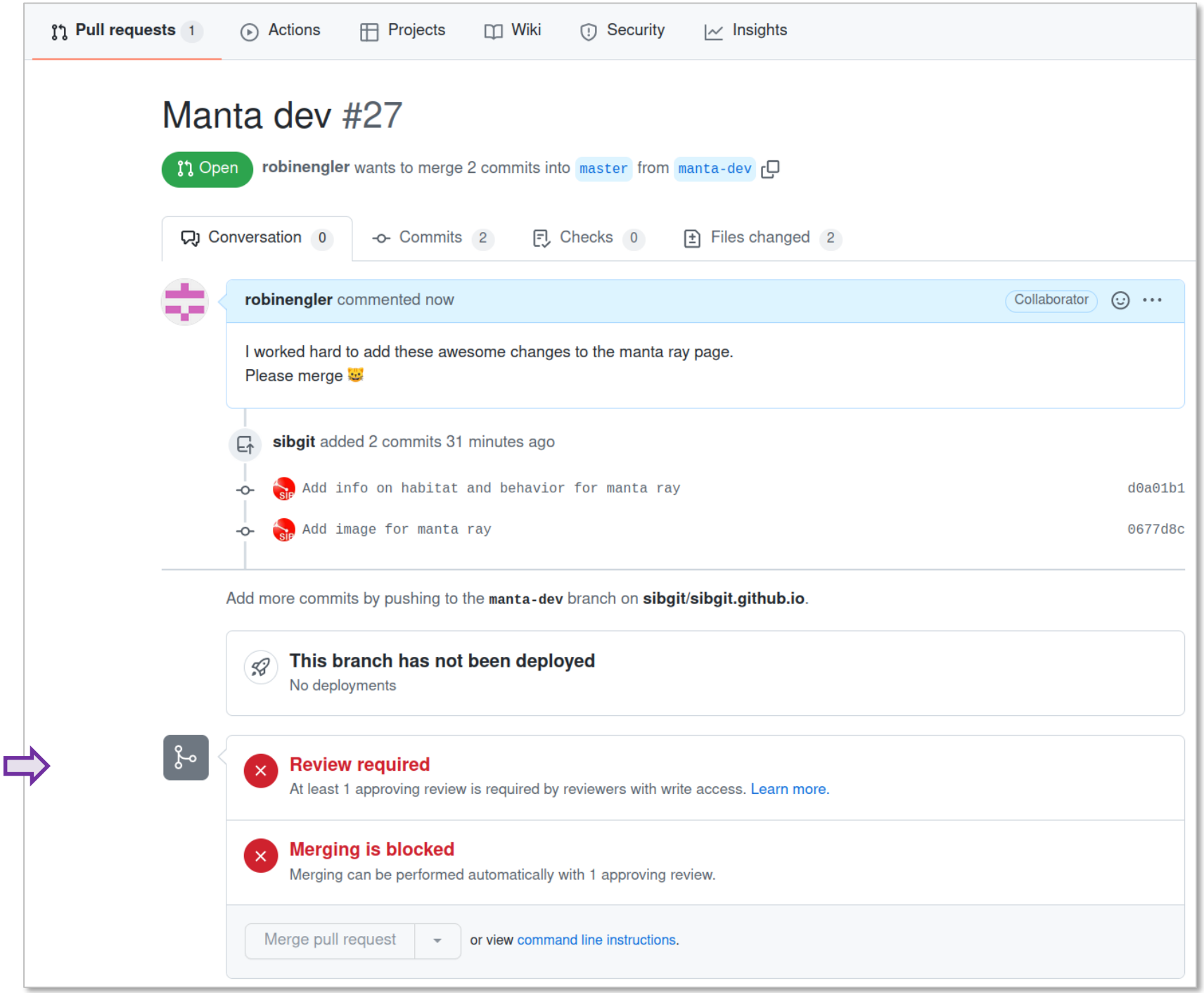
5. Optionally, enter a **message** for the people that will review your pull request.



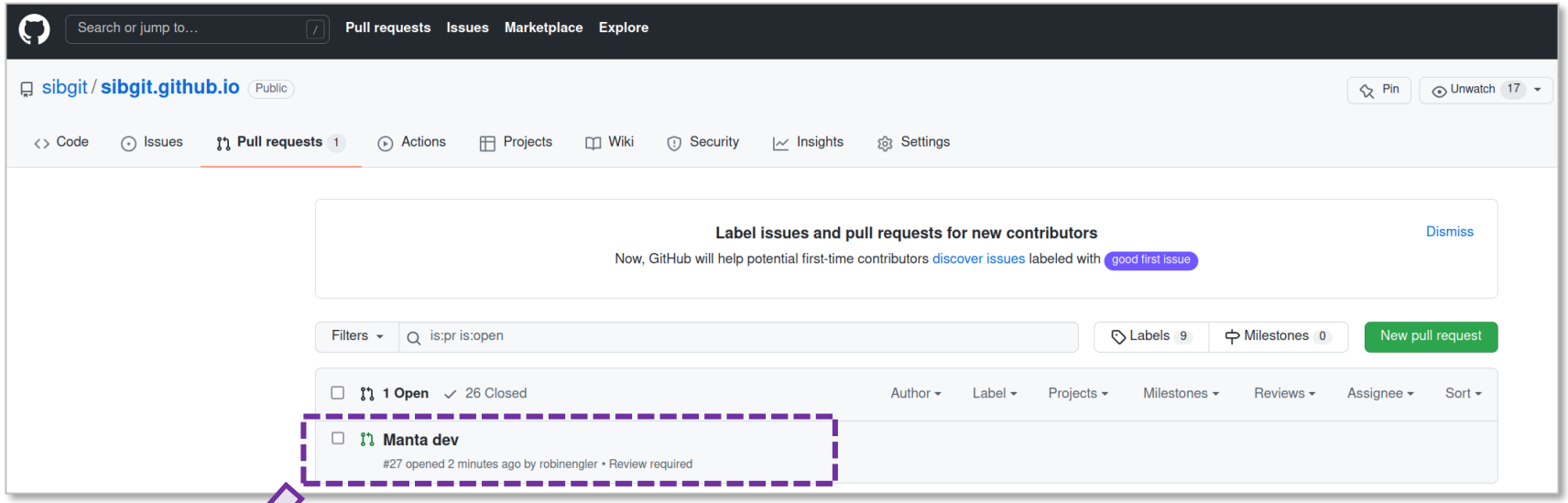
6. Submit your pull request by clicking **Create pull request**.

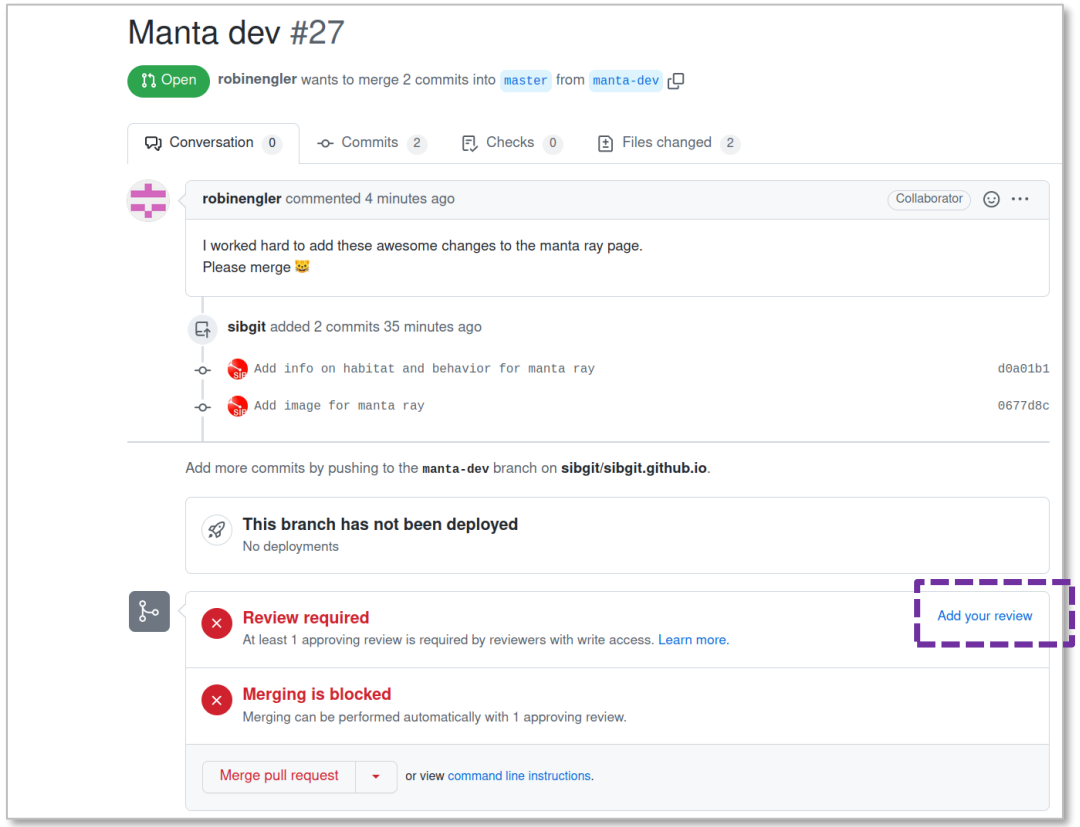
The pull request is now **created**, and **awaiting approval** from an authorized person.
(e.g. the repo owner or a colleague)

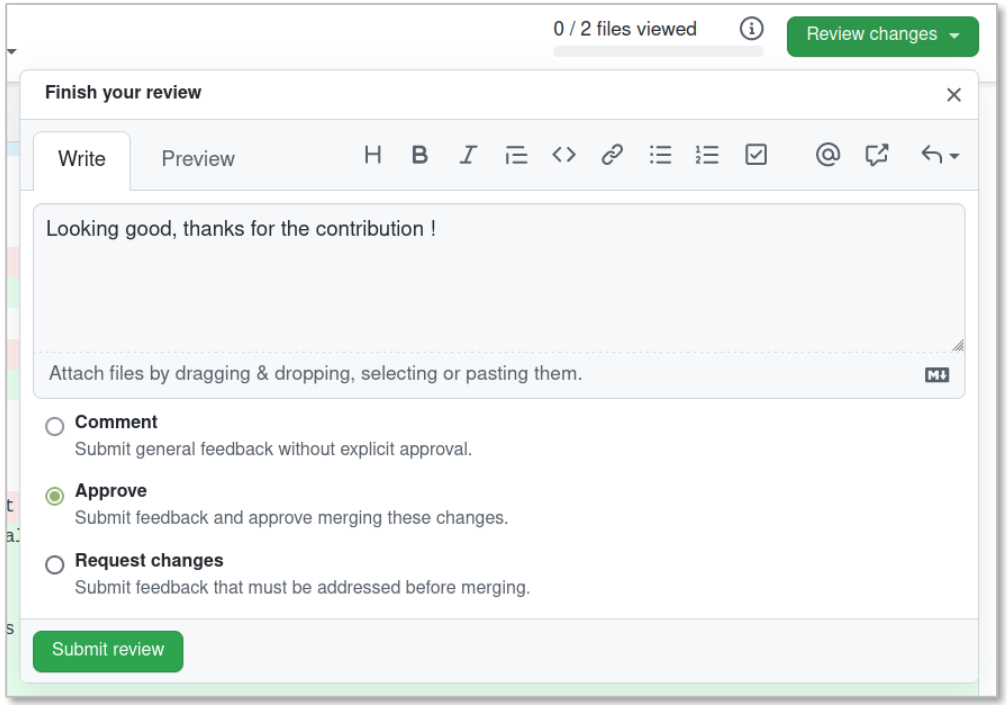
Merging is **blocked**, because someone has to approve your PR.

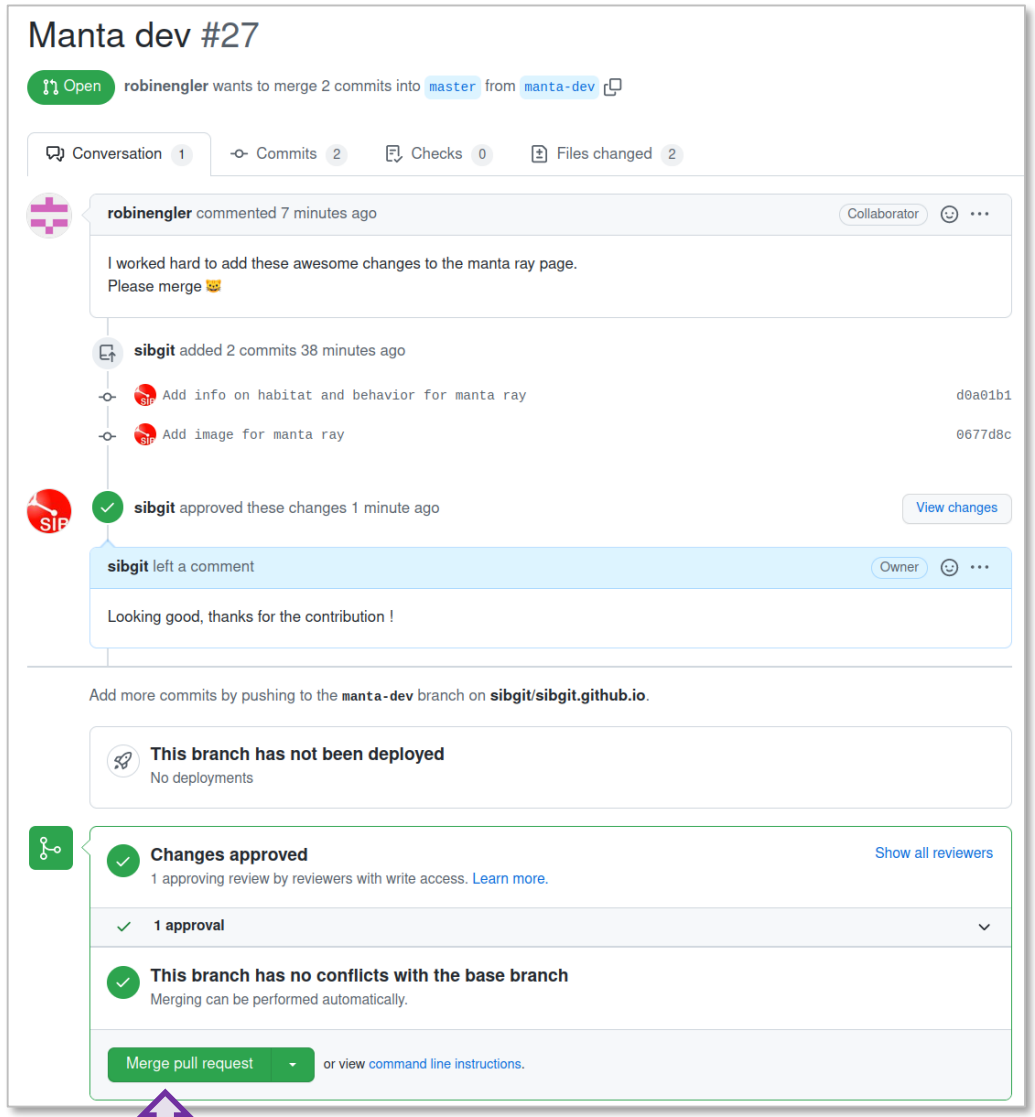


The **reviewer** of your PR will then have a look at your changes (the modifications introduced with your commits) and **approve them or request changes**.

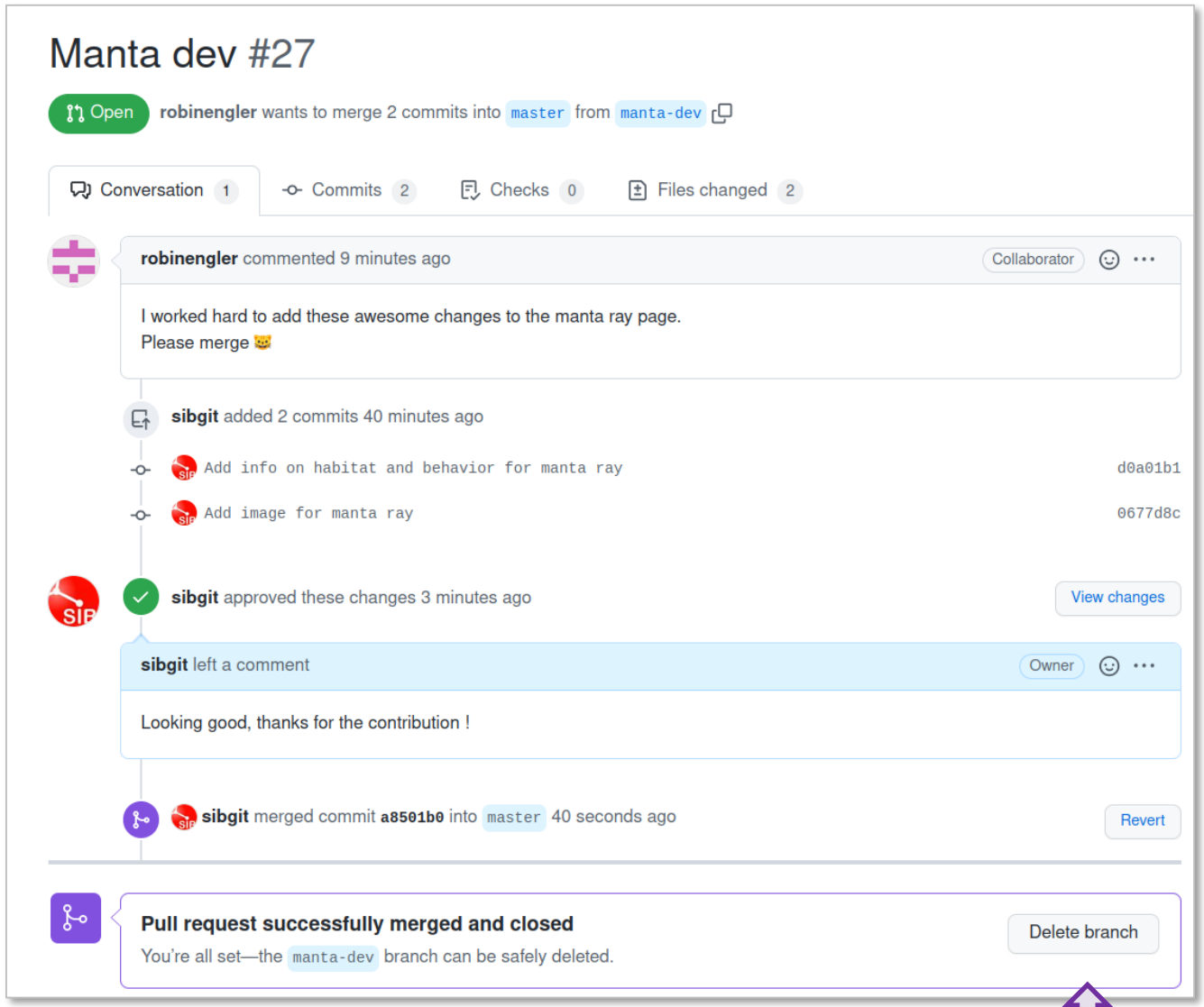








Now that the pull request is approved, it can be merged (either by the reviewer or by you) by clicking **Merge pull request**.



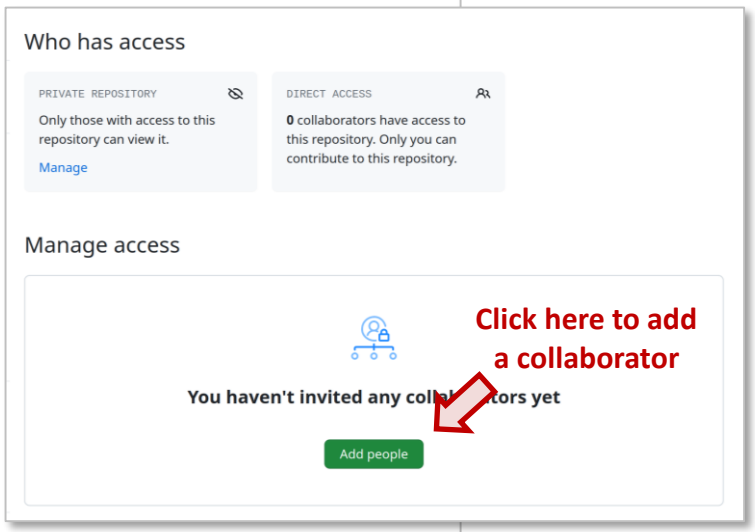
Completed ! Optionally, you can **delete your branch** on the remote (this will not delete it locally).

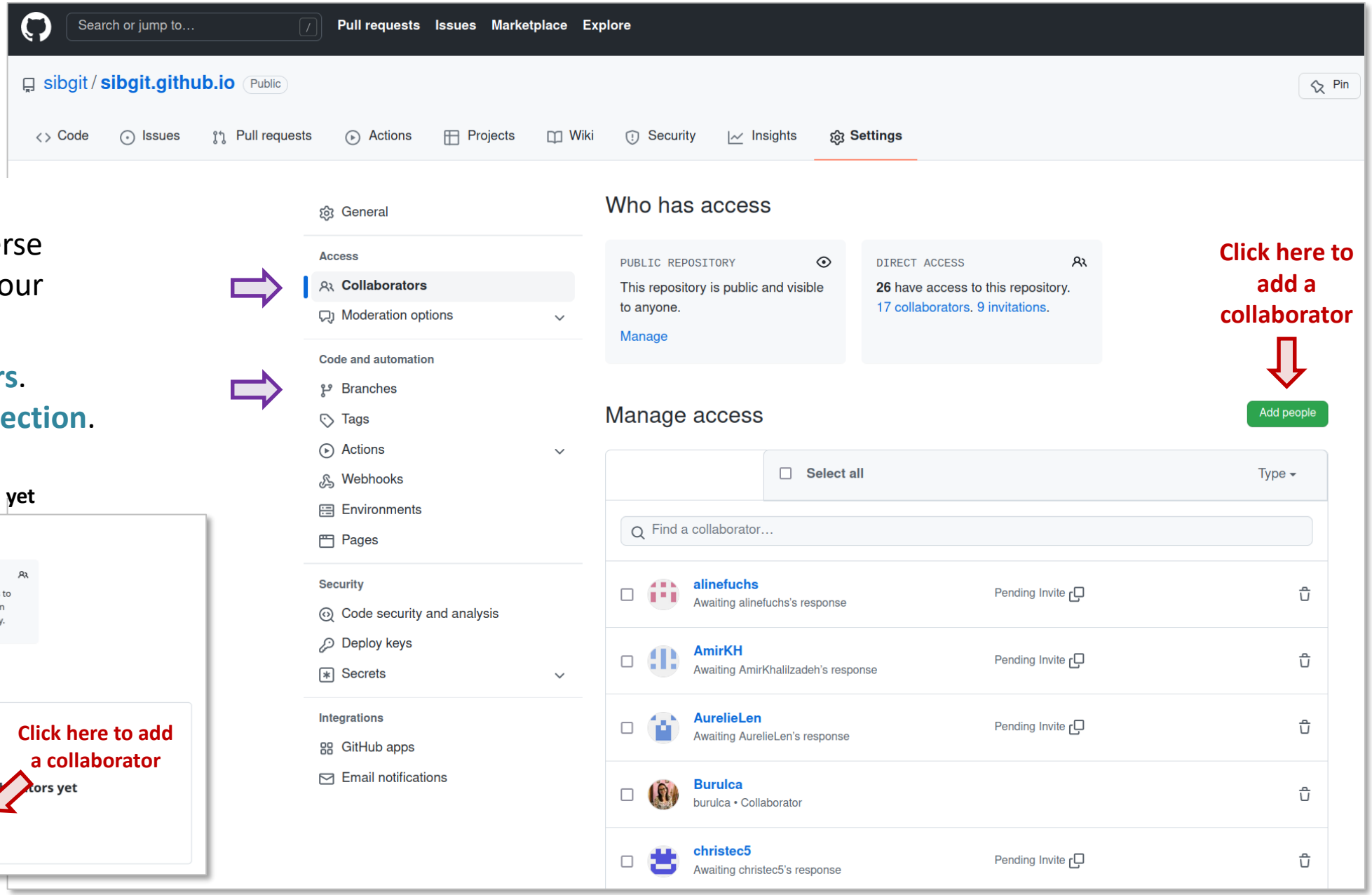
Repository settings (only available if you are the owner)

Here you can set diverse settings concerning your repository, e.g. :

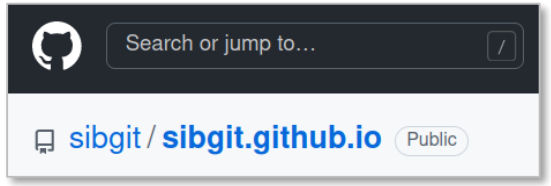
- Invite **collaborators**.
- Setup **branch protection**.

View with no collaborator added yet





Other GitHub features (some of them)



"Home" of your repo (repo content)

Issue tracker

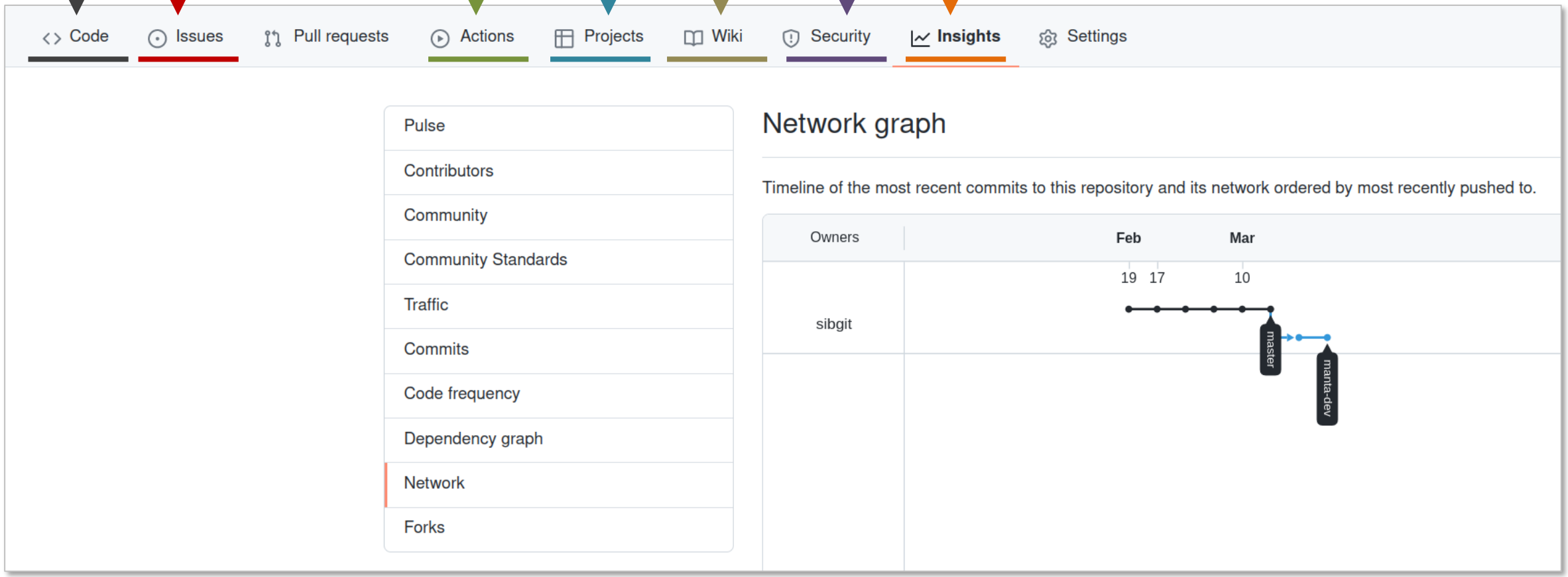
Continuous integration (automated testing)

Group issues and PR by topics.

Add a wiki for your project.

Setup automated security scanning for your code (vulnerability check).

Statistics about your repo's activity.

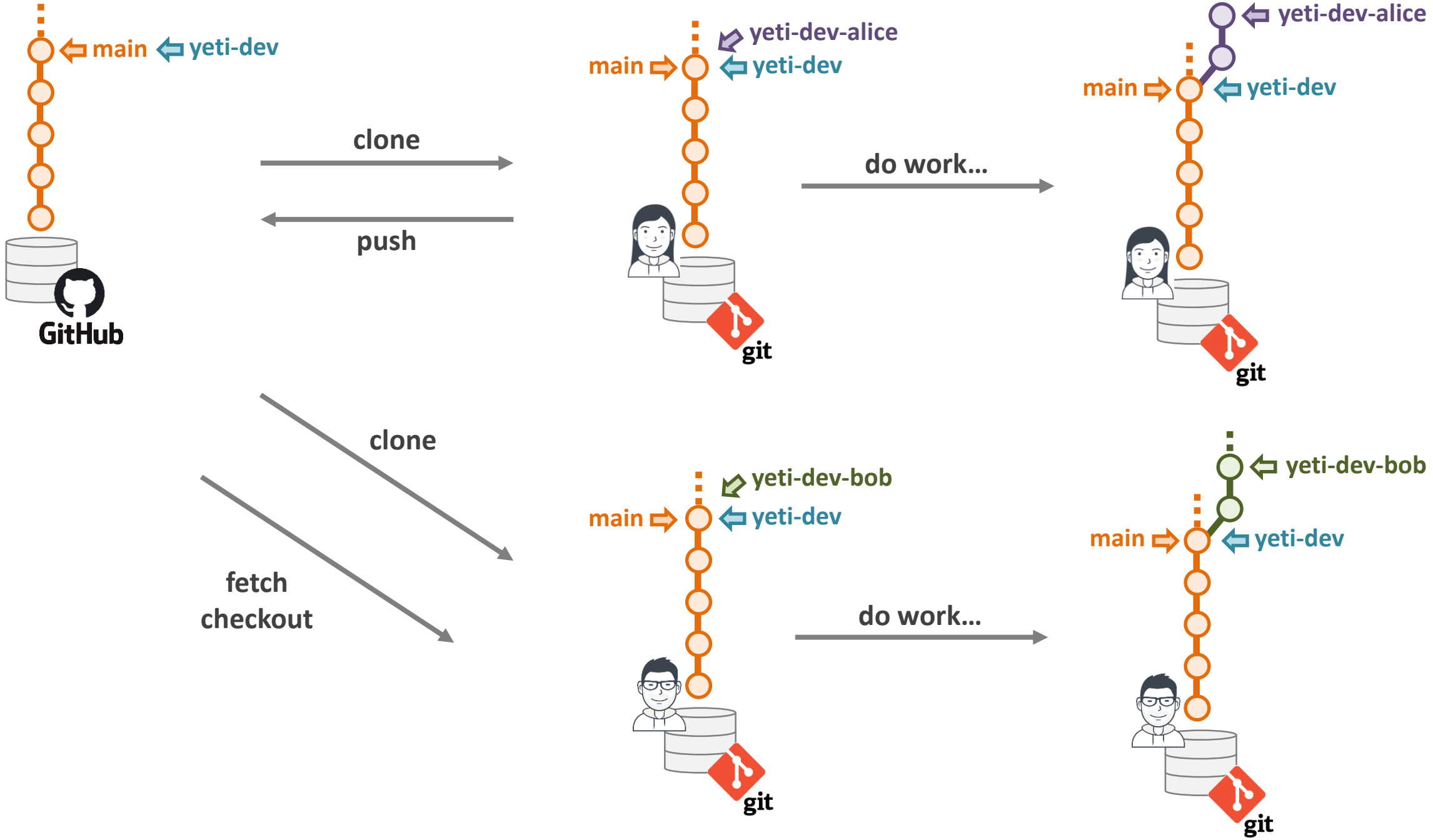


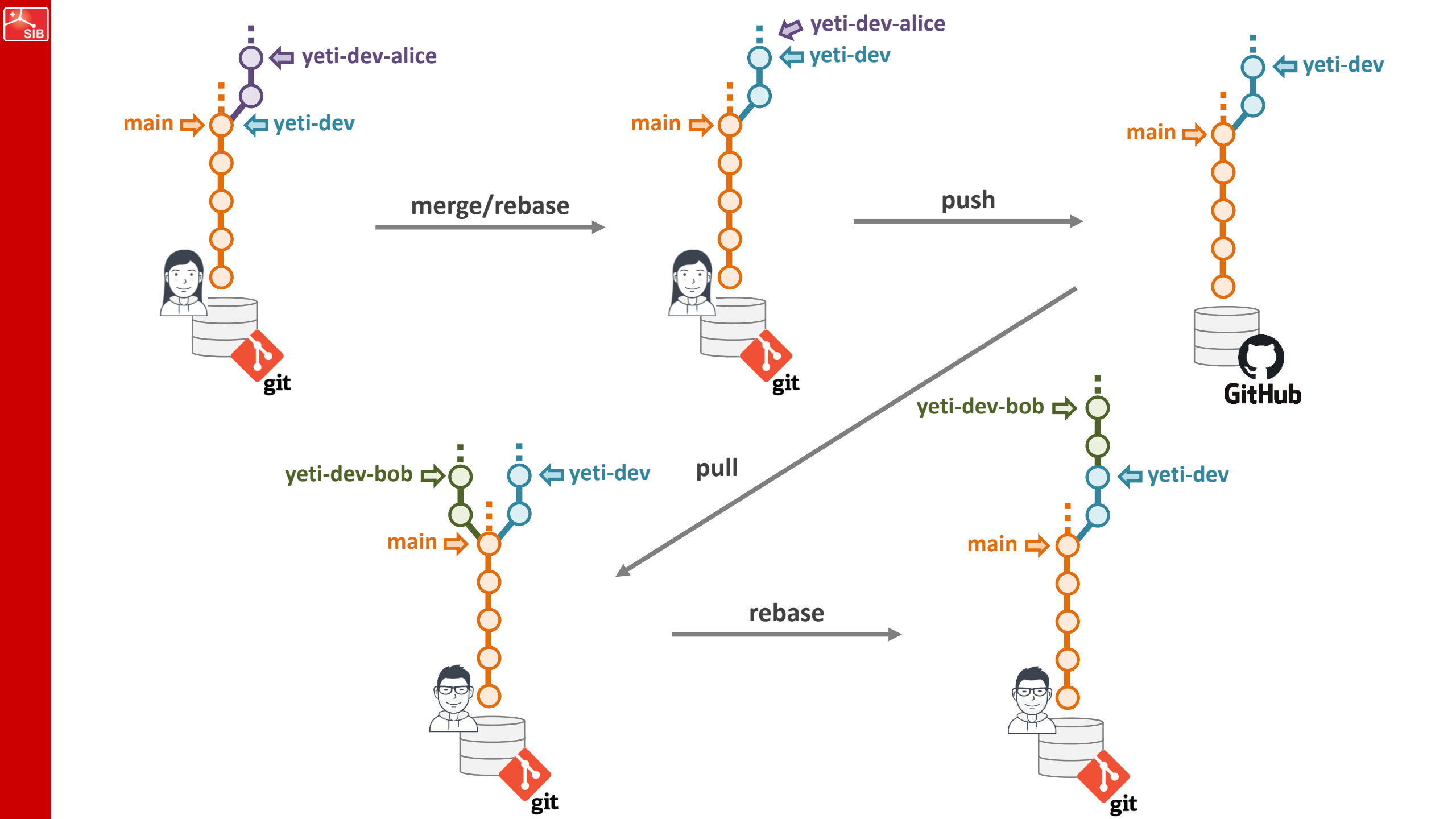
exercise 4

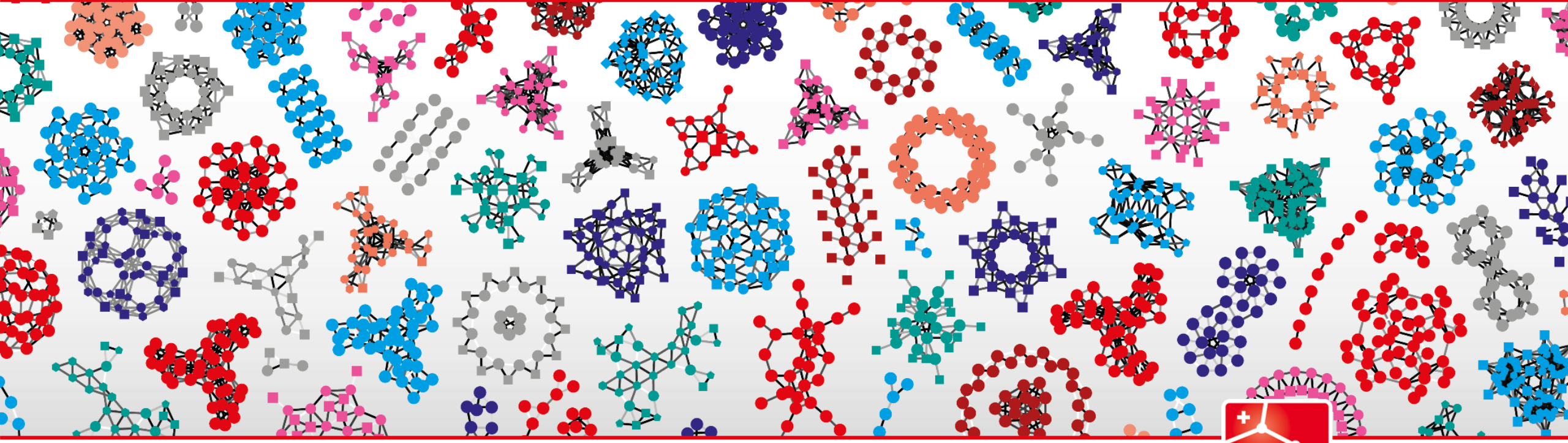
The Awesome Animal Awareness Project

This exercise has helper slides

Exercise 4 help: branch – rebase – merge sequence




yeti-dev-alice
main
yeti-dev
git
merge/rebase
yeti-dev-alice
yeti-dev
main
git
push
yeti-dev
main
GitHub
pull
yeti-dev-bob
yeti-dev
main
git
rebase
yeti-dev-bob
yeti-dev
main
git



Thank you for attending this course

SIB
Swiss Institute of Bioinformatics