Version Control

Code management with Git and GitHub

Moritz Wunderwald (they / them)
2.-3.12.2025

Agenda

Day 1

- Git Basics - Tracking and Branching

- GitHub - Synchronising, Sharing, Collaborating

Day 2

- Git and GitHub in RStudio Projects

- Best practices for collaborative coding

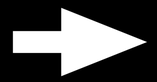
Git Basics

Tracking and Branching

Version Control

Why?

- 📂 my_project
  - main_script
  - main_script_backup
  - main_script_fixed
  - main_script_fixed_final
- 📂 my_project_SAVE_january2025
- 📂 my_project_RELEASE



- 📂 my_project
  - .git
  - main_script

Version Control

Why?

Without version control:

… changes are irreversible

… implementing new features affects stable codebase

… manual backups can get confusing

… work progress is hidden

… collaboration makes everything worse

Version Control

Why?

Safety

Code is always **backed up**

Breaking changes can be **reverted**

Parallelisation

Separation of concerns:

Project **branches** per work package without manual copies

Progress Tracking

What has been done **when** and **which lines of code** are affected?

Version Control

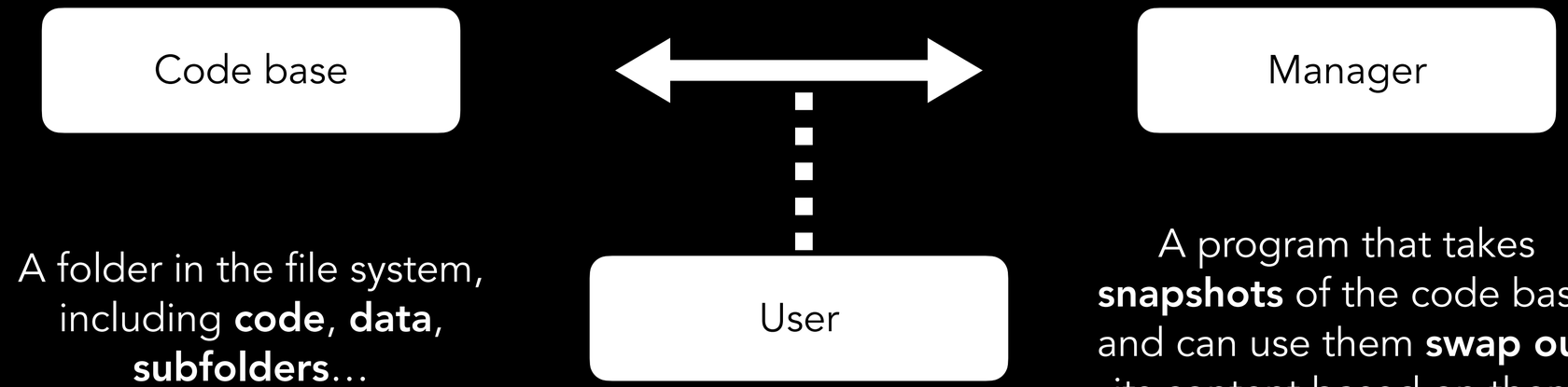
Core concept

A folder in the file system, including **code**, **data**, **subfolders**…

A program that takes **snapshots** of the code base and can use them **swap out** its content based on them

Version Control

Core concept



A folder in the file system, including **code**, **data**, **subfolders**…

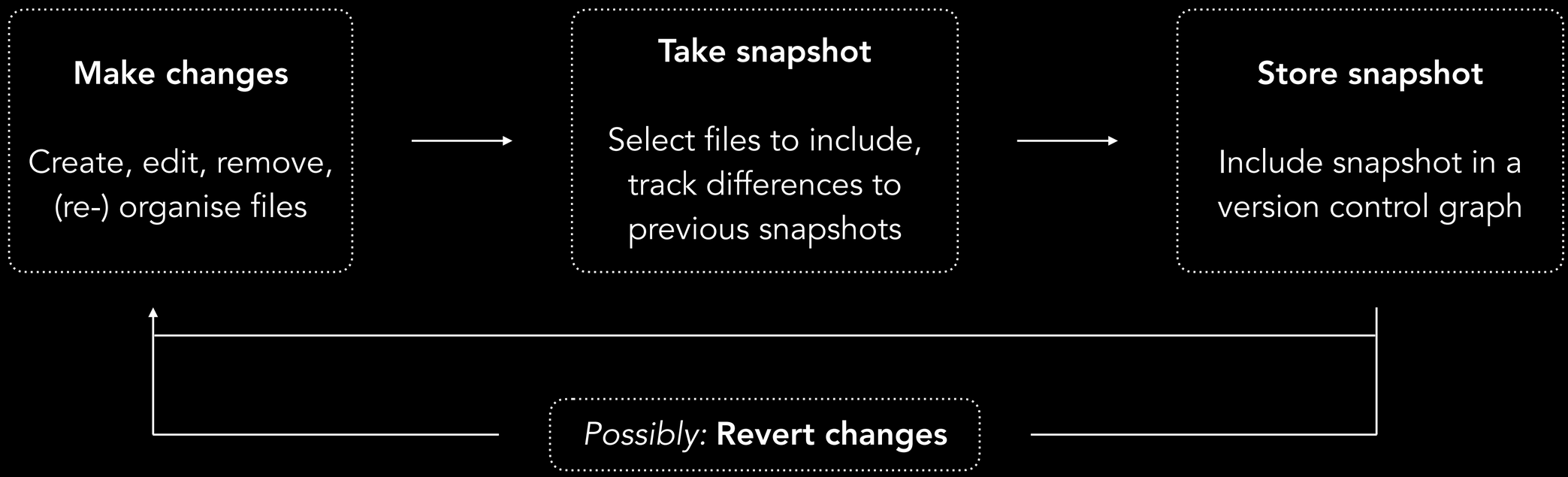
A program that takes **snapshots** of the code base and can use them **swap out** its content based on them

Works on code base

Tells manager **which files to keep track of** and **when to take snapshots**

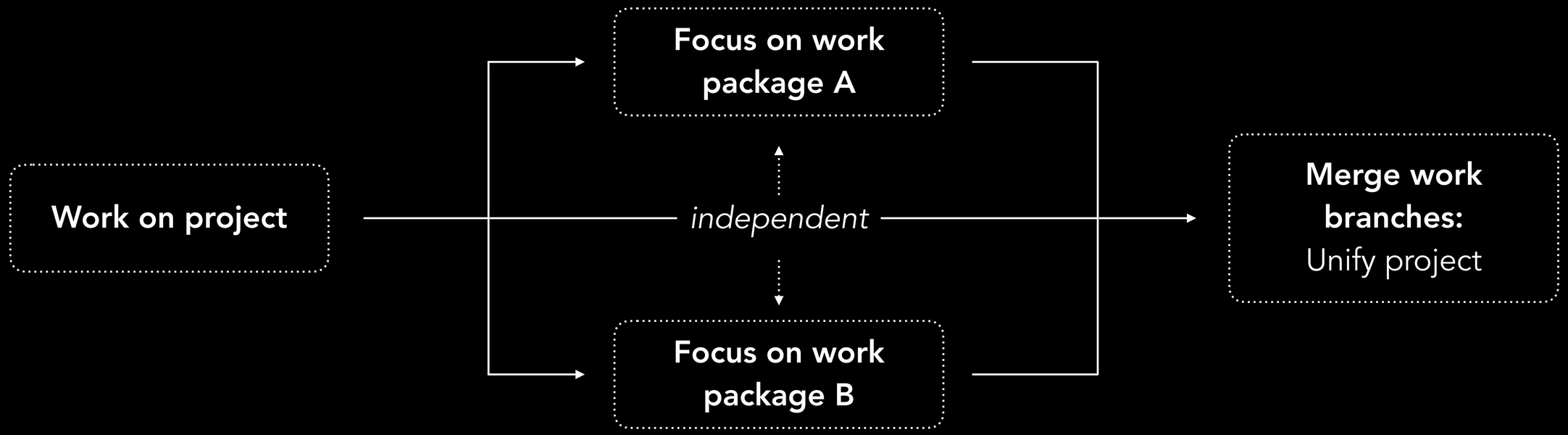
Version Control

General workflow: Tracking



Version Control

General workflow: Branching



Git

Background

Released 2005 by Linus Torvalds

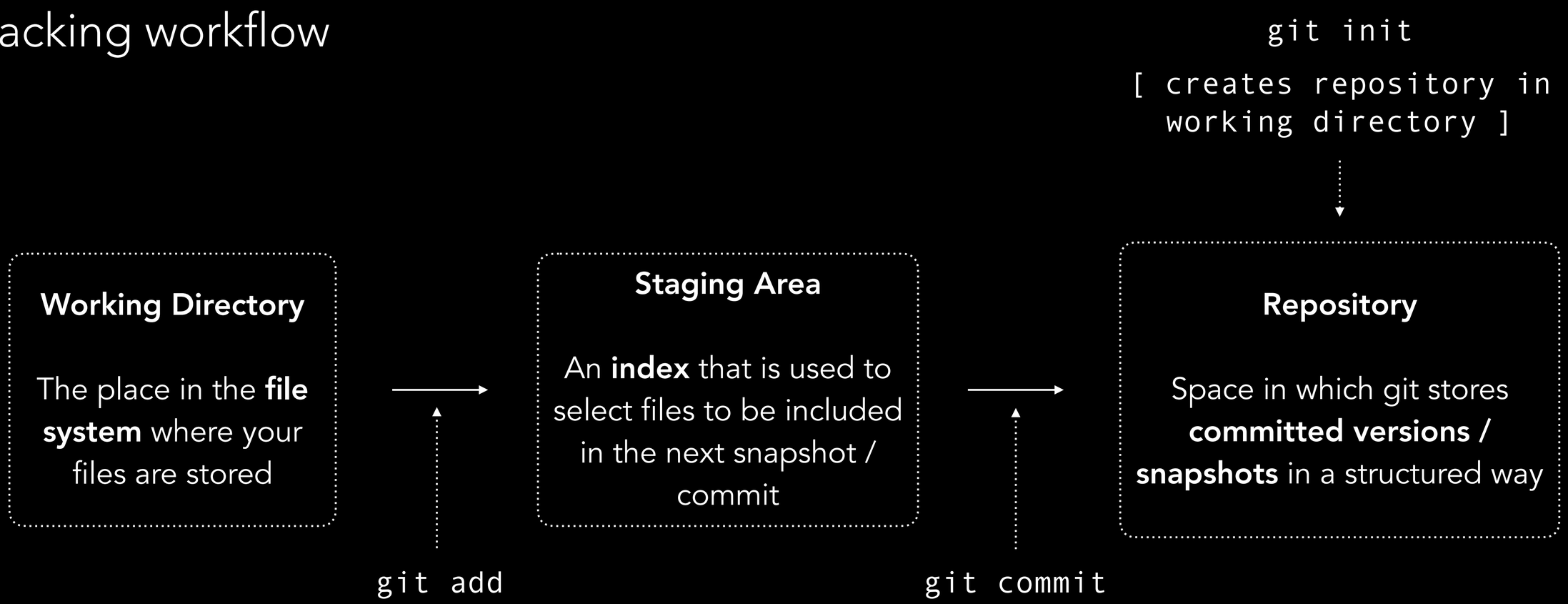
Free, open-source, industry-standard version control software

Runs locally - can be used via command line, desktop application, IDE integration

Collaboration and hosting via online platforms: GitHub, GitLab…

Git

Tracking workflow



Git

Tracking workflow: .gitignore

Especially in research, it is good practice to use GitHub **only for code**, **not for data.**

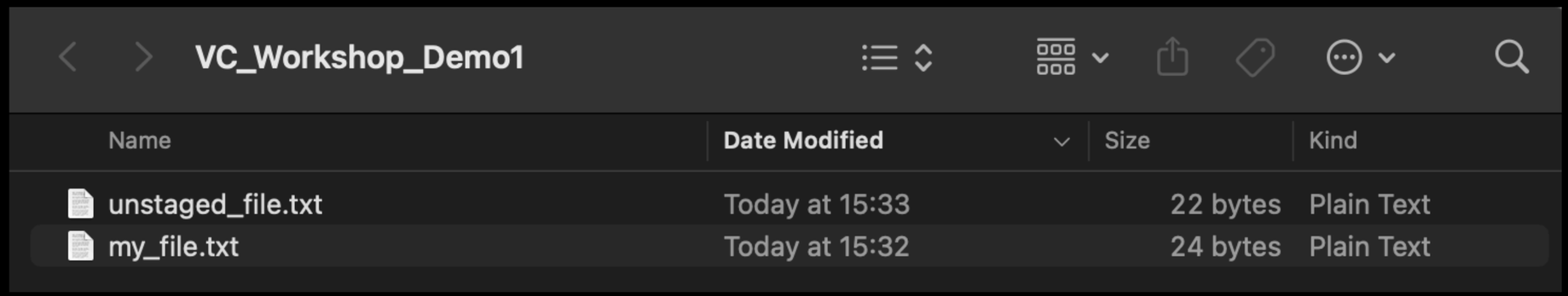
A **.gitignore** file tells git file types and subdirectories that are excluded from staging and commits.

Ideally, do not commit any files of ignored types before setting up .gitignore!

Git

Tracking practice

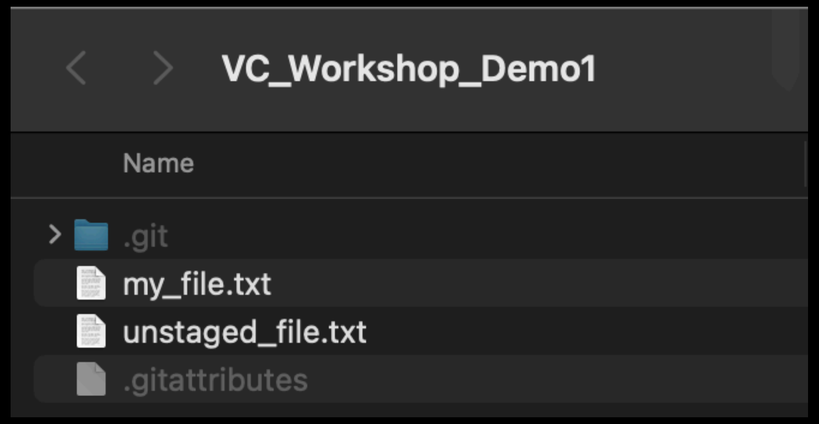
Working Directory



Two (newly created) files

Git

Tracking practice



When initialising a git repository, git creates `.git` (the repository) inside the working directory.

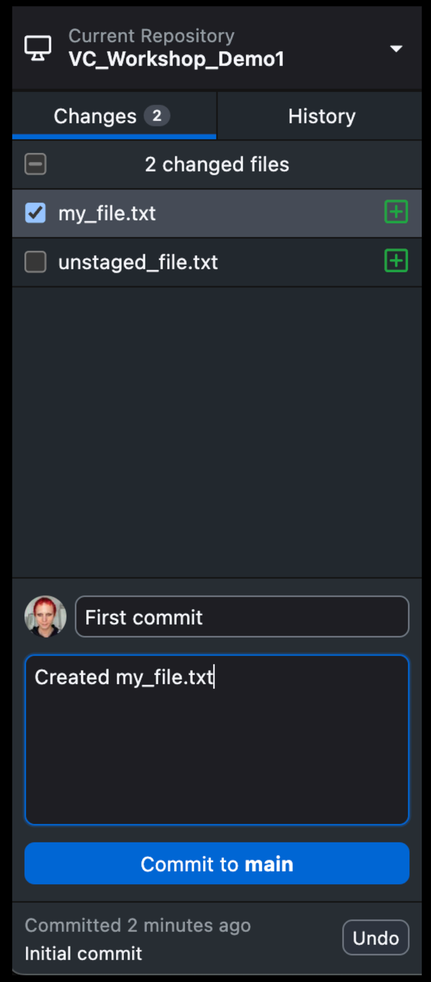
Files and folders starting with a dot are usually hidden in Mac Finder and Windows explorer.

Use `cmd + shift + .` (Mac Finder) or `View > Show, then select Hidden items` (Win Explorer) to see them.

Git

Tracking practice

Changed files show up here and can be added to the **staging area**

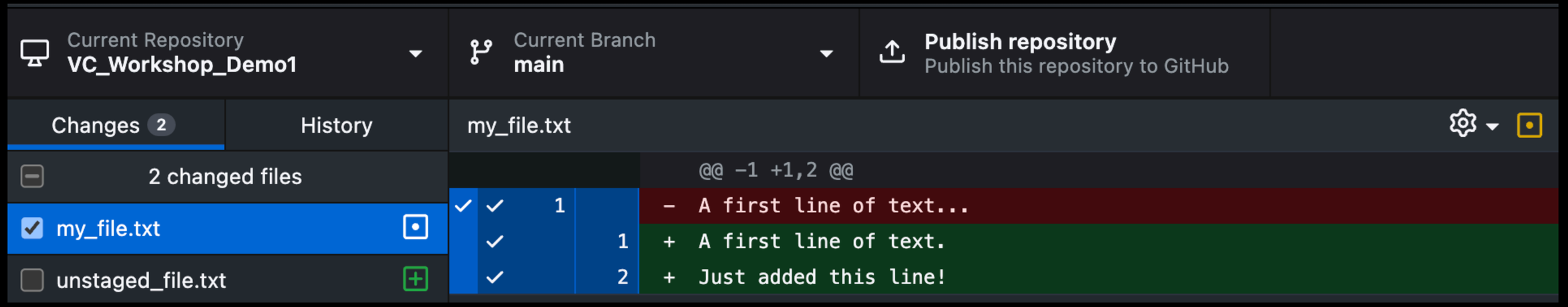


Repository opened in Git Desktop Application

Staged files are part of the next **commit**. Write a concise title (and description) before committing.

Git

Tracking practice



Differences in the file compared to the version in the last commit

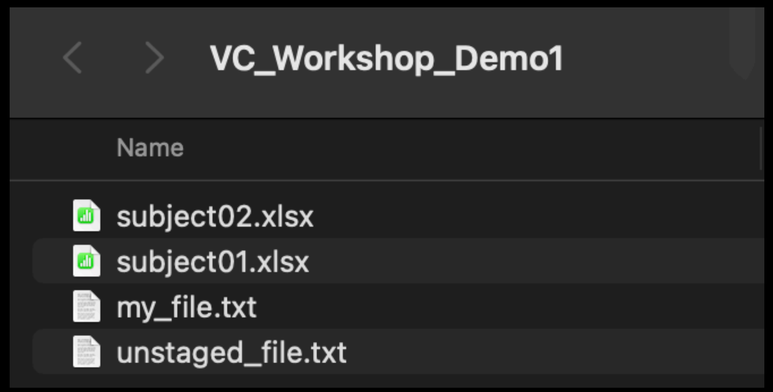
Git

Tracking practice: .gitignore

Text editor

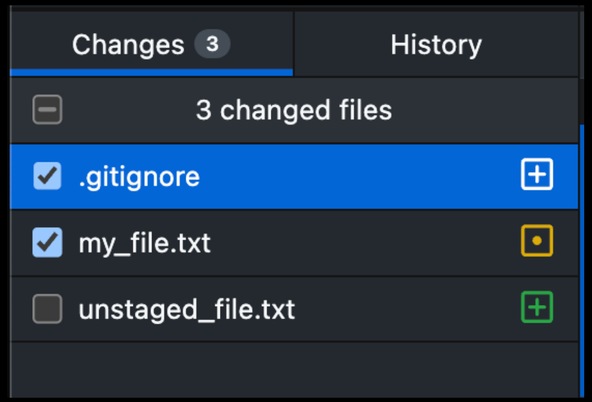
.gitignore is a plain text file with no file extension

Finder / Explorer



If files of the ignored types are in your working directory…

GitHub Desktop



… they will not be considered for staging

Git

Tracking - your turn!

Open **GitHub Desktop** and a **simple text editor**.

Download the cheat sheet from: github.com/wunderwald/workshop_materials

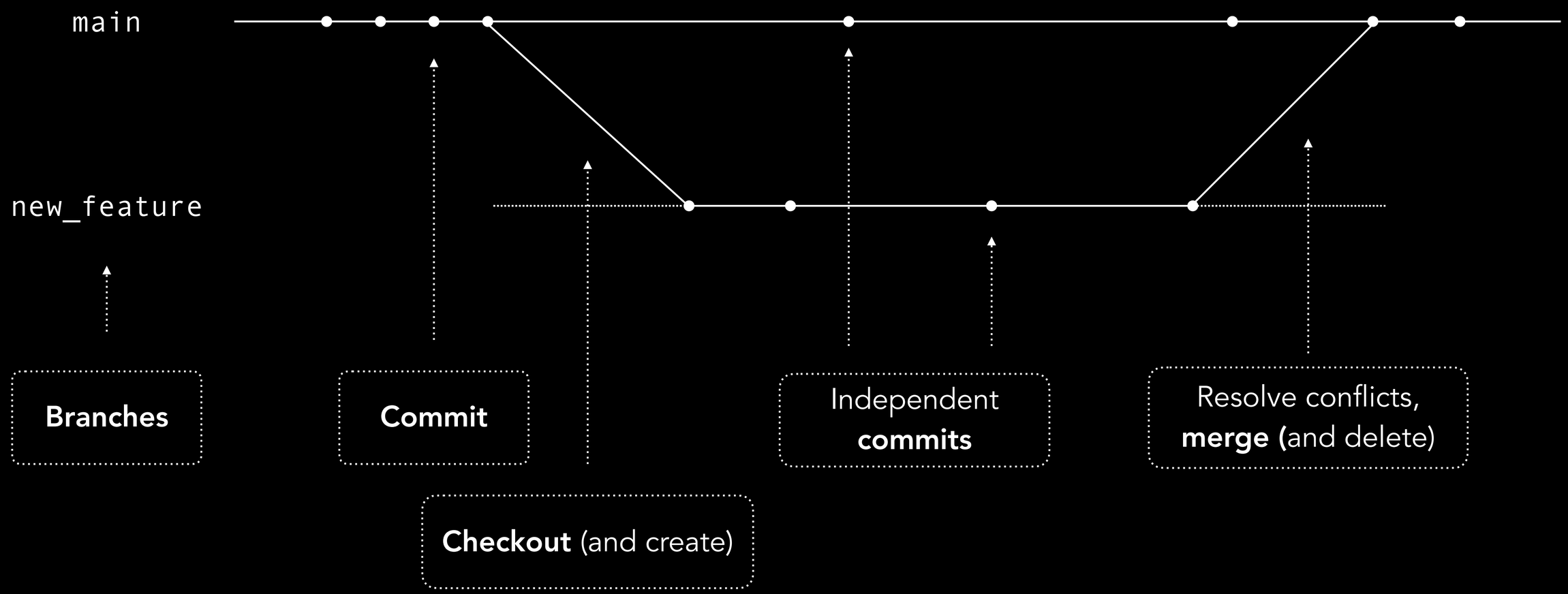
Git

Tracking - your turn!

1. Set up an empty **working directory** and **initialise a git repository** in it using GitHub Desktop.
2. Create some **.txt files** with some random content in the directory.
   [use TextEdit on Mac or Editor on Windows]
3. Create a **.gitignore** file to exclude some types of data files (.xlsx, .csv…).
   Unhide dot-files in finder / explorer.
   Put some **files of the ignored types** into the working directory for testing.
4. Make a first **commit**! Be sure that all of your .txt files have been added to the **staging area**.
   Write a **commit message** and **description**.
5. **Change your files** (add to them, change lines, delete/rename files) and make **2 more commits**. Look at the diff view before committing and check the history.

Git

Branching workflow



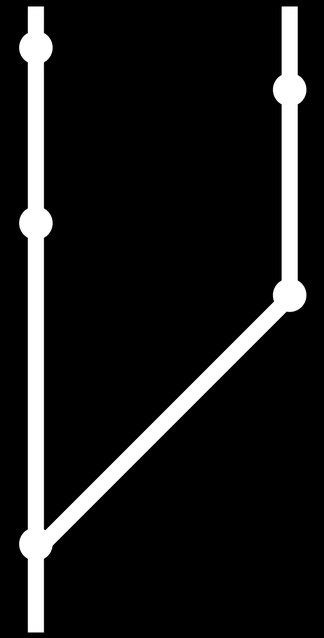
Git

Branching workflow: merging

`git merge` := combine the history of two branches

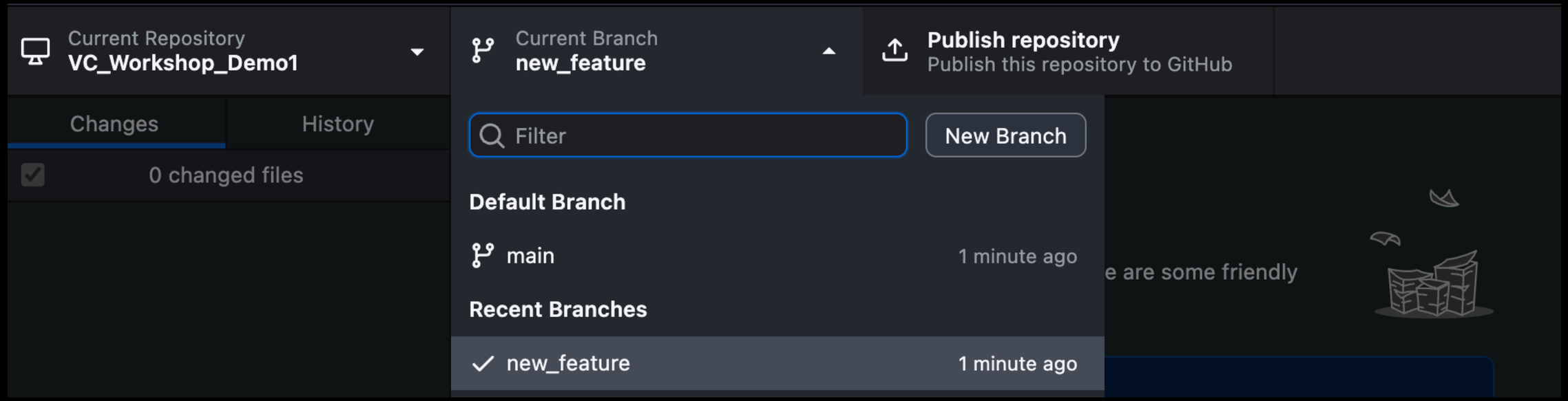
If merged branches have **independent histories**, git can merge **automatically.**

If the same lines of code have been changed on the branches, there is a **merge conflict** that needs to be **resolved manually.** *[this sounds a lot worse than it is]*



Git

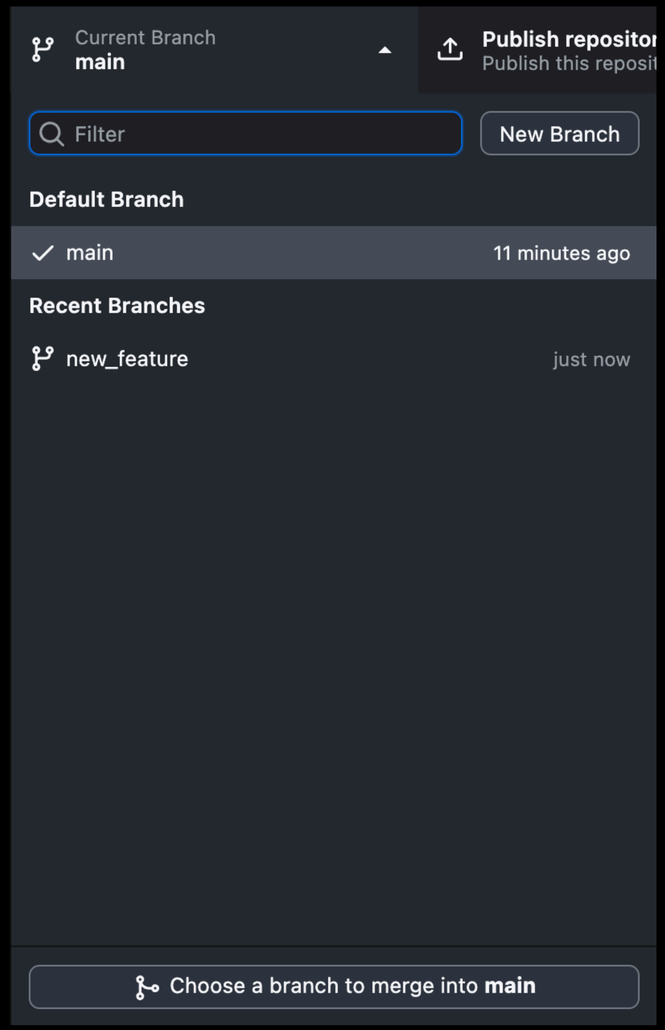
Branching practice



Create and switch branches in GitHub Desktop

Git

Branching practice: merging



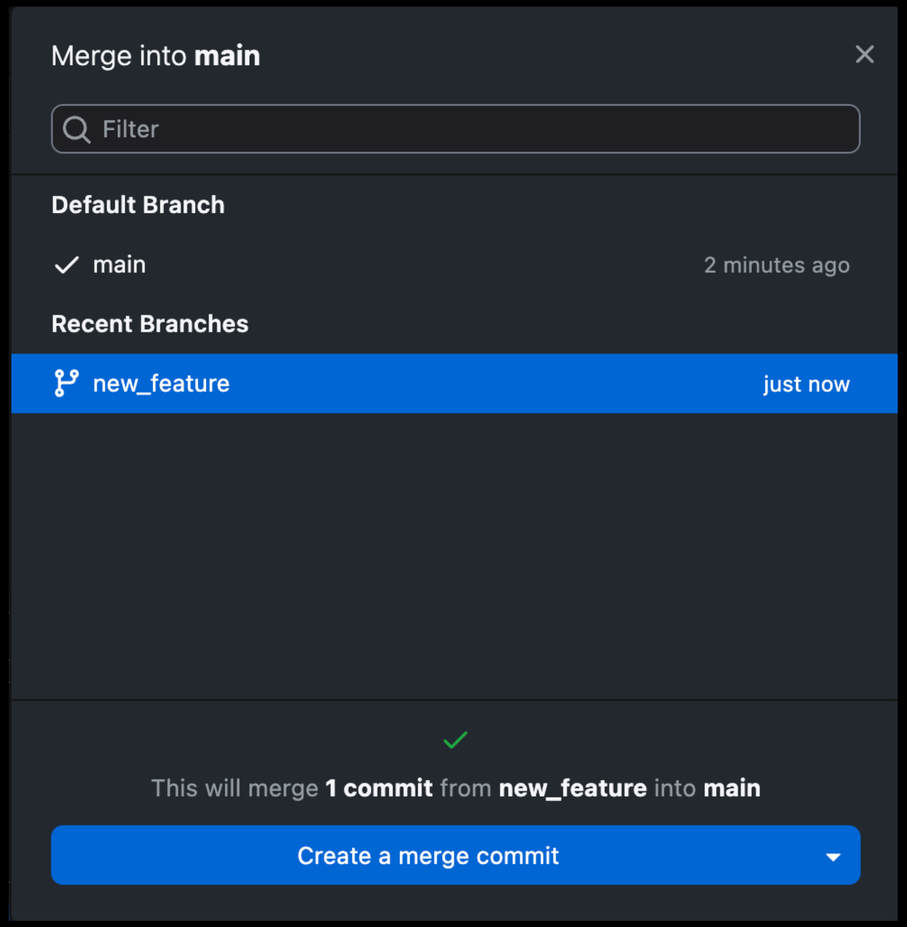
Switch to branch that should be merged into

Select a branch to merge into selected branch

Git

Branching practice: merging

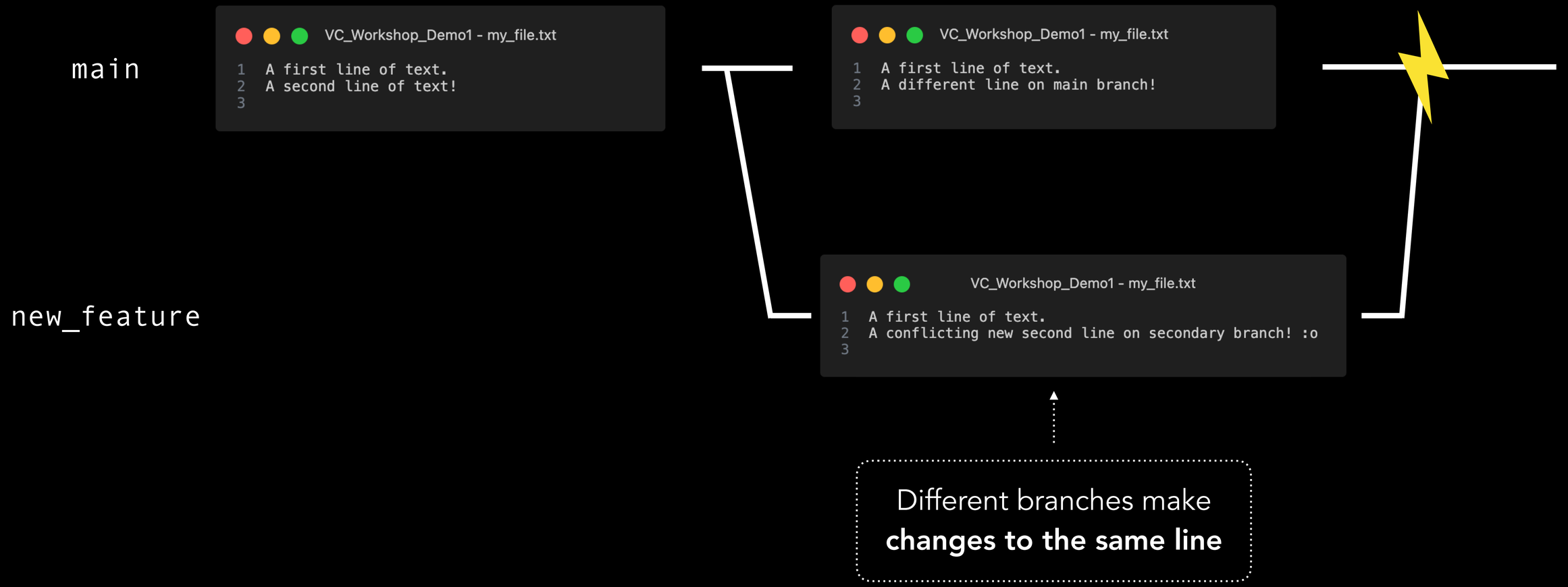
After selecting, git will tell you **if there are conflicts**



Here, there are none - simply **commit the merge**

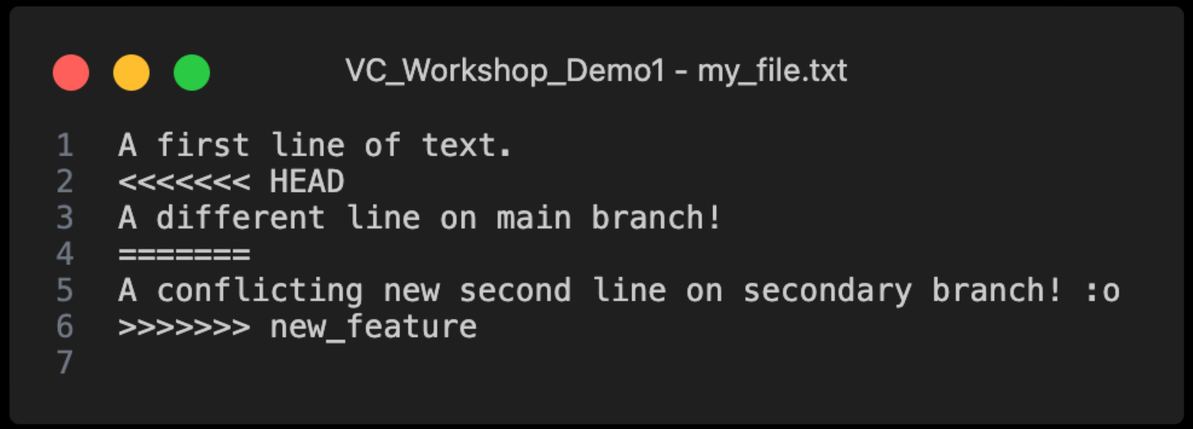
Git

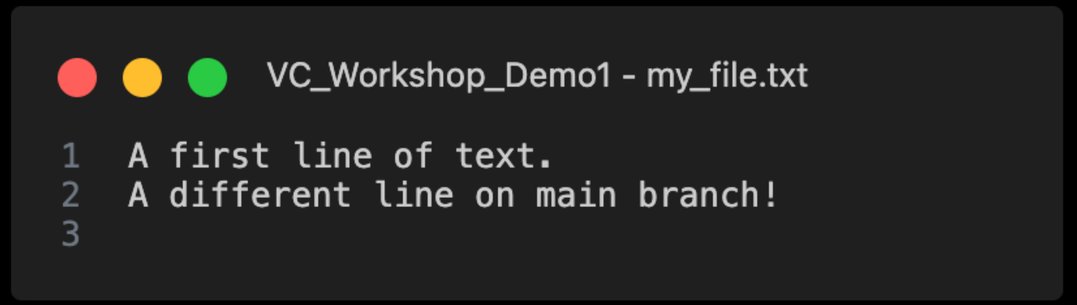
Branching practice: creating conflicts



Git

Branching practice: resolving conflicts





If there is a conflict, git puts a conflict marker into the affected file:

To resolve, simply delete everything apart from the content to keep.

Git

Branching - your turn!

1. Create a **new branch** and **switch** to it.
2. Get a feeling for branches: Create a **new file and commit**. Switch to the main branch and check your finder/explorer. Then switch back to the new branch, check your finder/explorer again.
3. Conflict-free merge: On the new branch, **make changes** to a file you created earlier. **Commit** the change and **merge** the new branch into the main branch.
4. Merge conflict: **change the same line** of text in a file on **both branches**. Merge the new branch into main and get a conflict. **Resolve** the conflict and commit the merge.

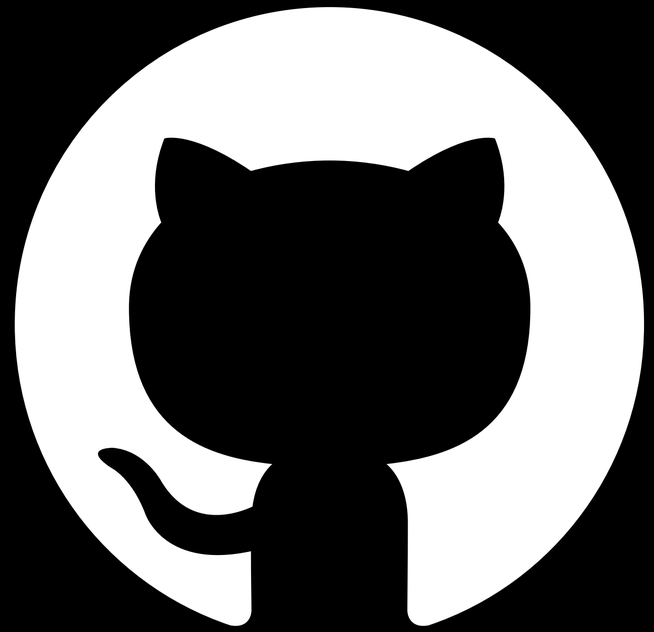
GitHub

Synchronising, Sharing, Collaborating

GitHub

Background

- Git manages local repositories (on your computer) - **GitHub hosts remote repositories** (online)
- Local repositories can be **published** to GitHub
- Remote repositories can be **cloned** or **forked**
- Changes between a local and remote repository can be synchronised using **fetch**, **pull** and **push**



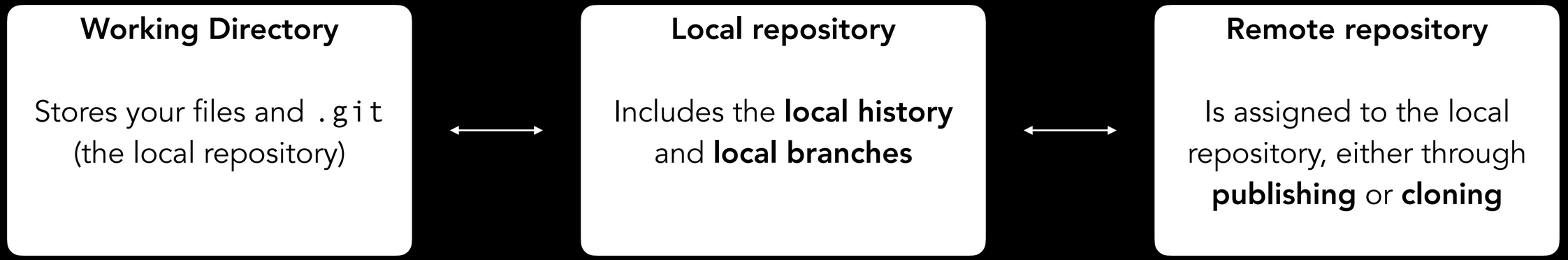
GitHub

Terminology: Synchronising

- `clone` := download a remote repository to your computer
- `fork` := copy a remote repository
- `origin` := the default name of the remote repository assigned to a local one
- `fetch` := download new changes from a remote repository
- `pull` := download new changes from a remote and synchronise the local repository
- `push` := synchronise local changes to the remote repository

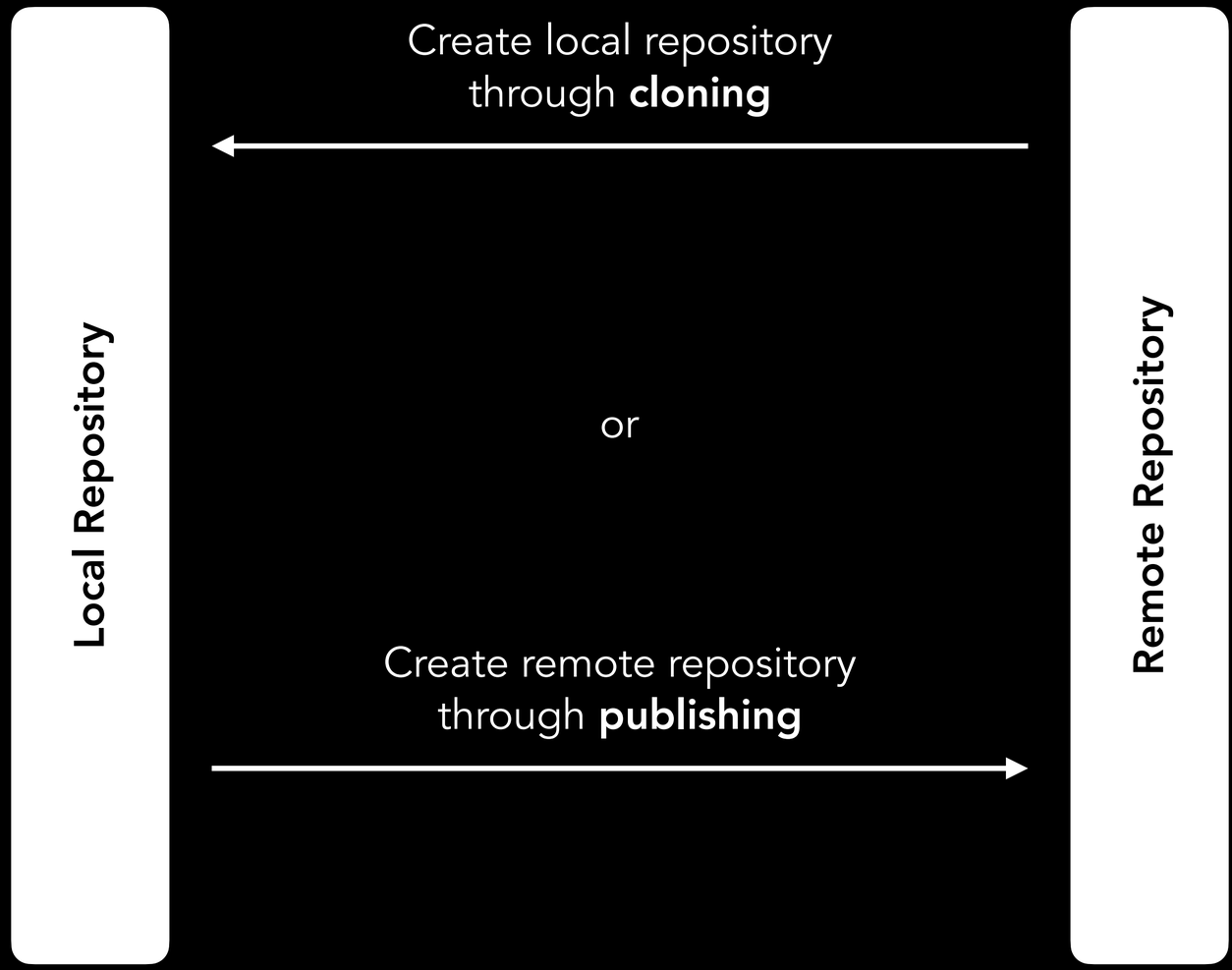
GitHub

Synchronising workflow: wd, local, remote



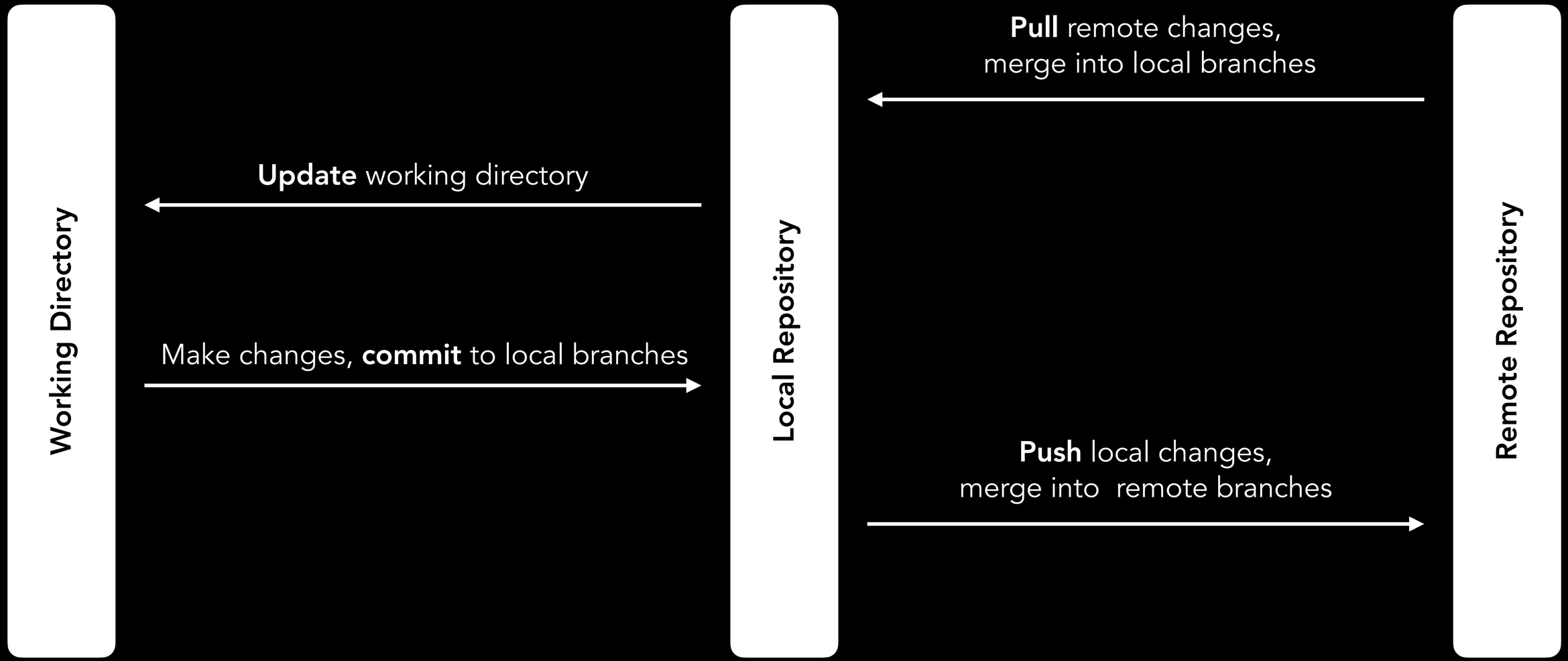
GitHub

Synchronising workflow: init



GitHub

Synchronising workflow: updates

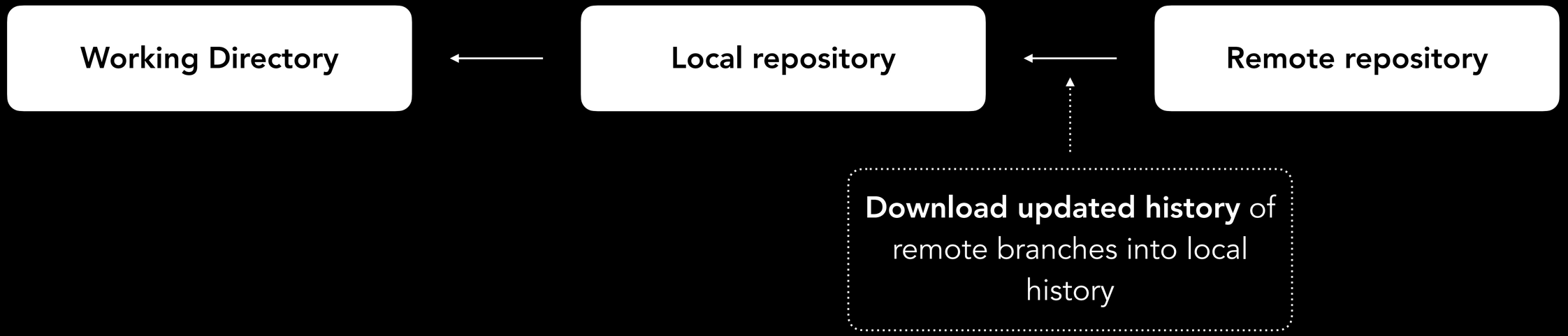


GitHub

Synchronising workflow: `push`

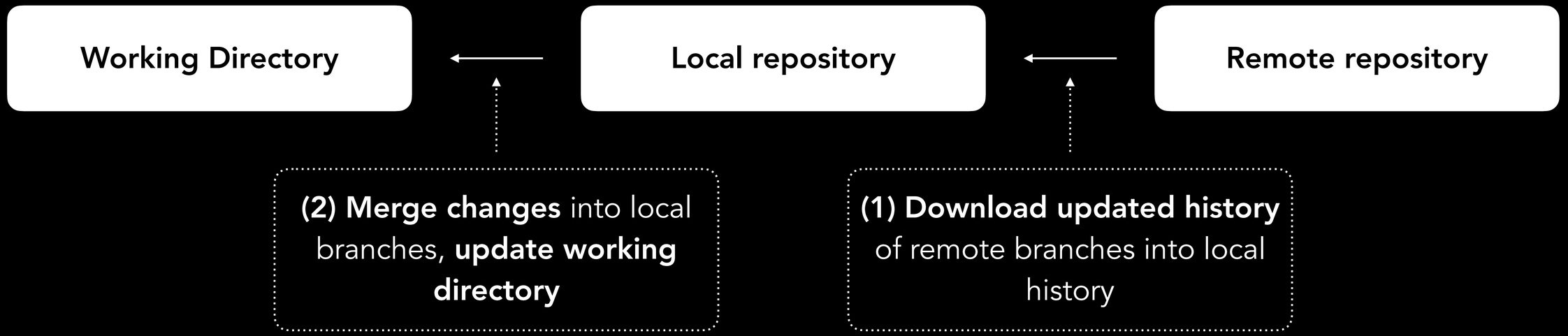
GitHub

Synchronising workflow: `fetch`



GitHub

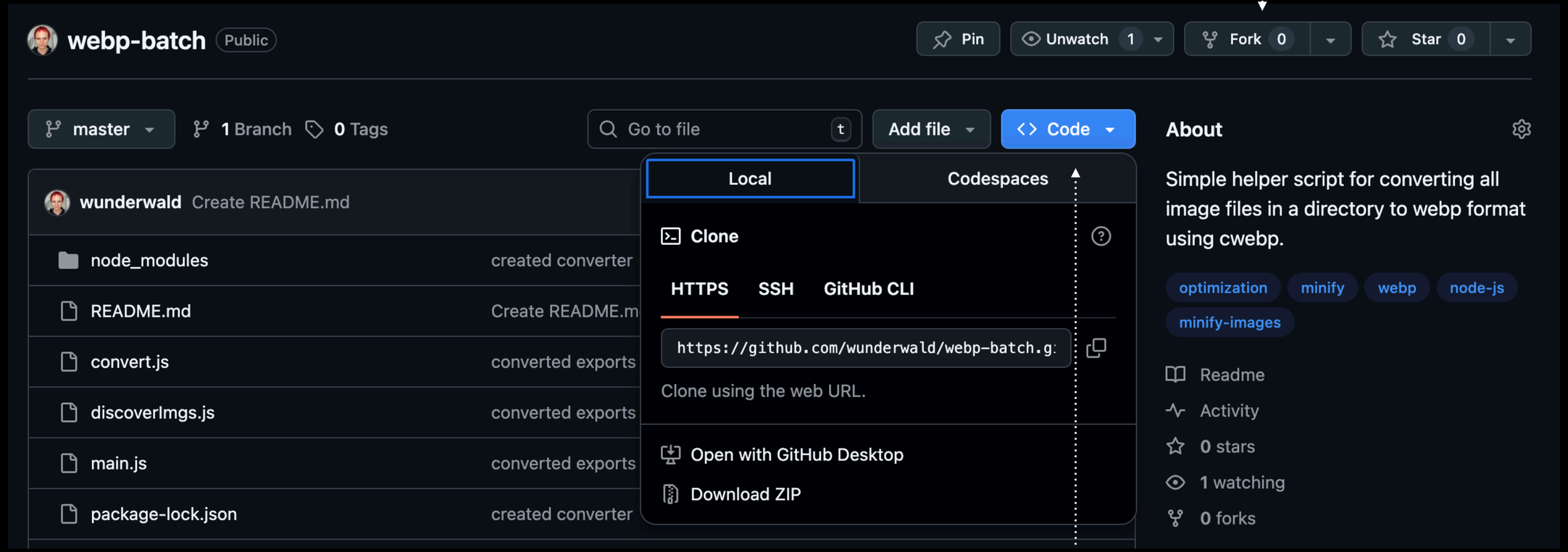
Synchronising workflow: `pull`



GitHub

Synchronising practice: `clone & fork`

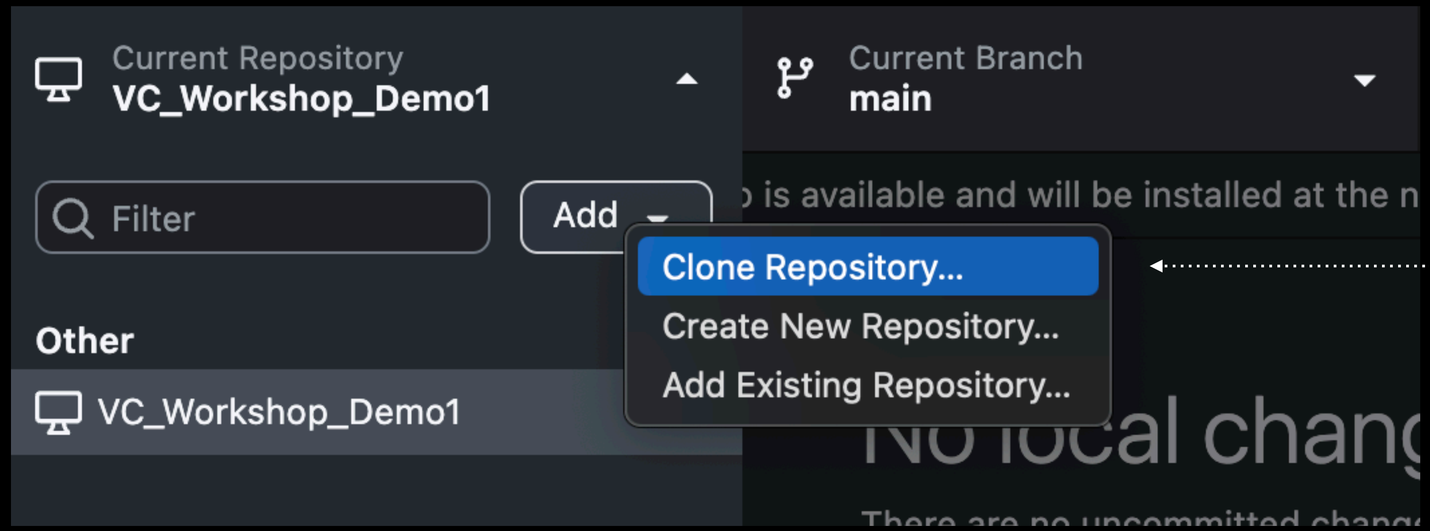
Fork via GitHub:
click Fork Button



Clone via GitHub: click <> Code and select “Open with GitHub Desktop”

GitHub

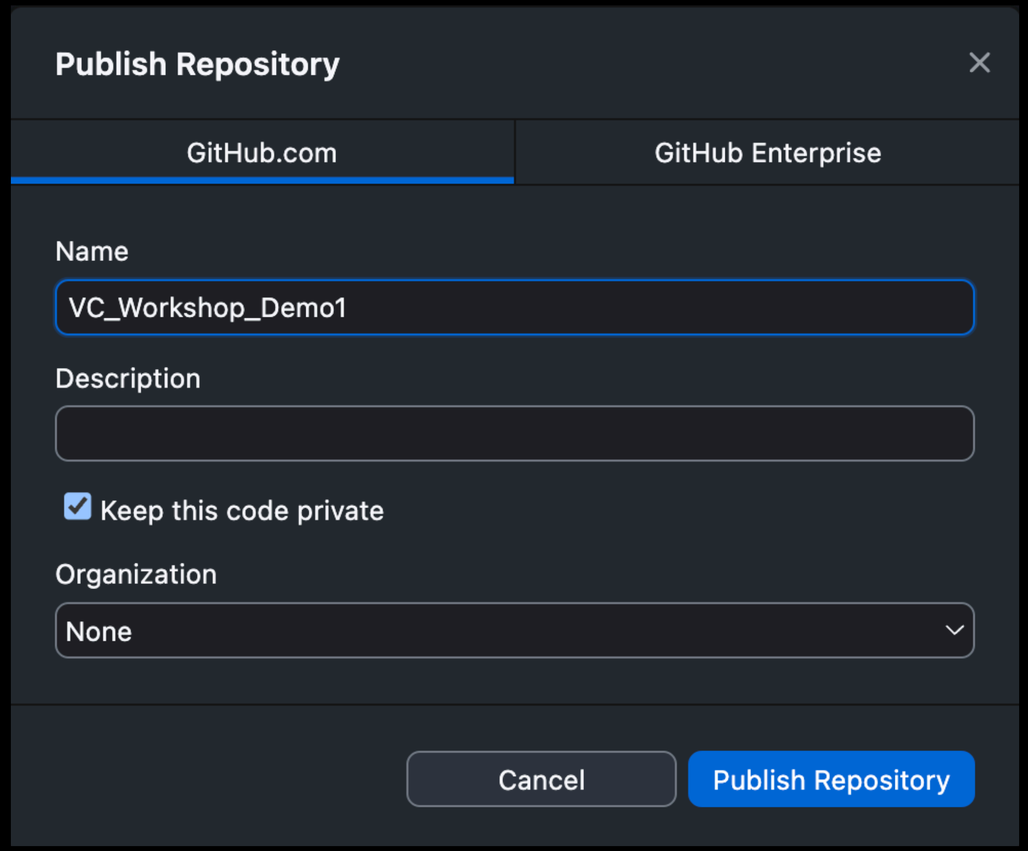
Synchronising practice: `clone`



Clone vie GitHub Desktop: add repository, select clone repository

GitHub

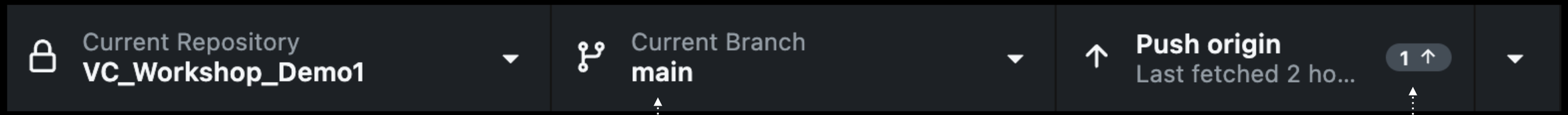
Synchronising practice: `publish`



In GitHub Desktop, **publish** repository to **create a remote repository**

GitHub

Synchronising practice: `push`



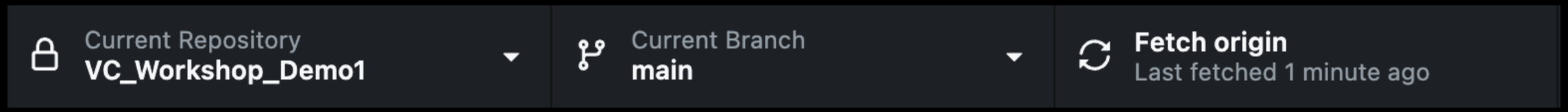
Pushing only affects the selected branch in local and remote repositories

Number of commits that local main branch is ahead of remote

Push local changes to remote

GitHub

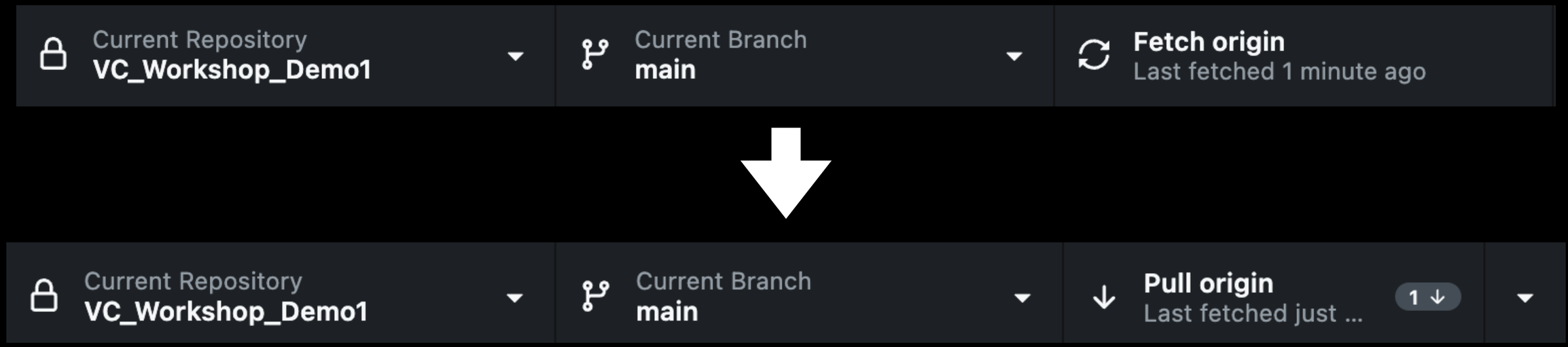
Synchronising practice: `fetch` & `pull`



Click fetch to get info about new changes on remote / origin

GitHub

Synchronising practice: `fetch` & `pull`



There is one new change on origin! Click pull to integrate it into local branch and to update files in working directory.

GitHub

Synchronising: Your turn!

1. Log in to github.com, **fork** the repo `sync-practice` from my GitHub profile (github.com/wunderwald/sync-practice/).
2. **Clone** the forked repository using GitHub Desktop (you will find it under “Your Repositories”).
3. On the **main branch**, commit some changes and **push them** to remote.
4. **Edit** a file of your repository on github.com and **commit** the change there.
5. Fetch and pull the change using GitHub Desktop. Check the file in the working directory.
6. Create a new **local branch**, commit something to it and **push** it to origin. Look at the different branches on GitHub in the browser.
7. Checkout to main.

GitHub

Working as a team

The administrator of a remote repository can add other GitHub users as **collaborators**.

For a clean, parallel work process, it is very important to:

- use **fine-grained branches** that a limited amount of users is working on (one branch per work package or feature)
- use **pull requests** to incorporate branches into the main branch when a feature or work package is done
- **regularly fetch / pull** changes made by collaborators
- **regularly push** new changes to remote **(!)**

GitHub

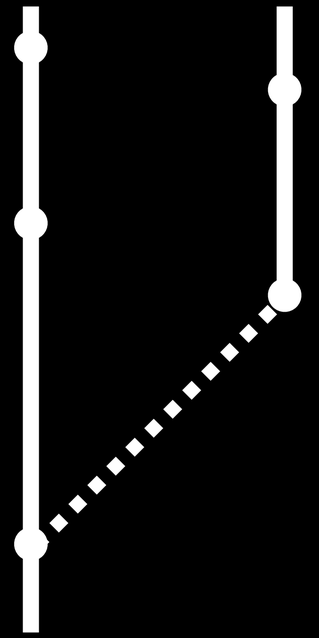
Collaborating and sharing workflow: pull requests

First of all: pull != pull request

A **pull request** is a “safety check” **before merging** a branch into the main branch in a remote repository.

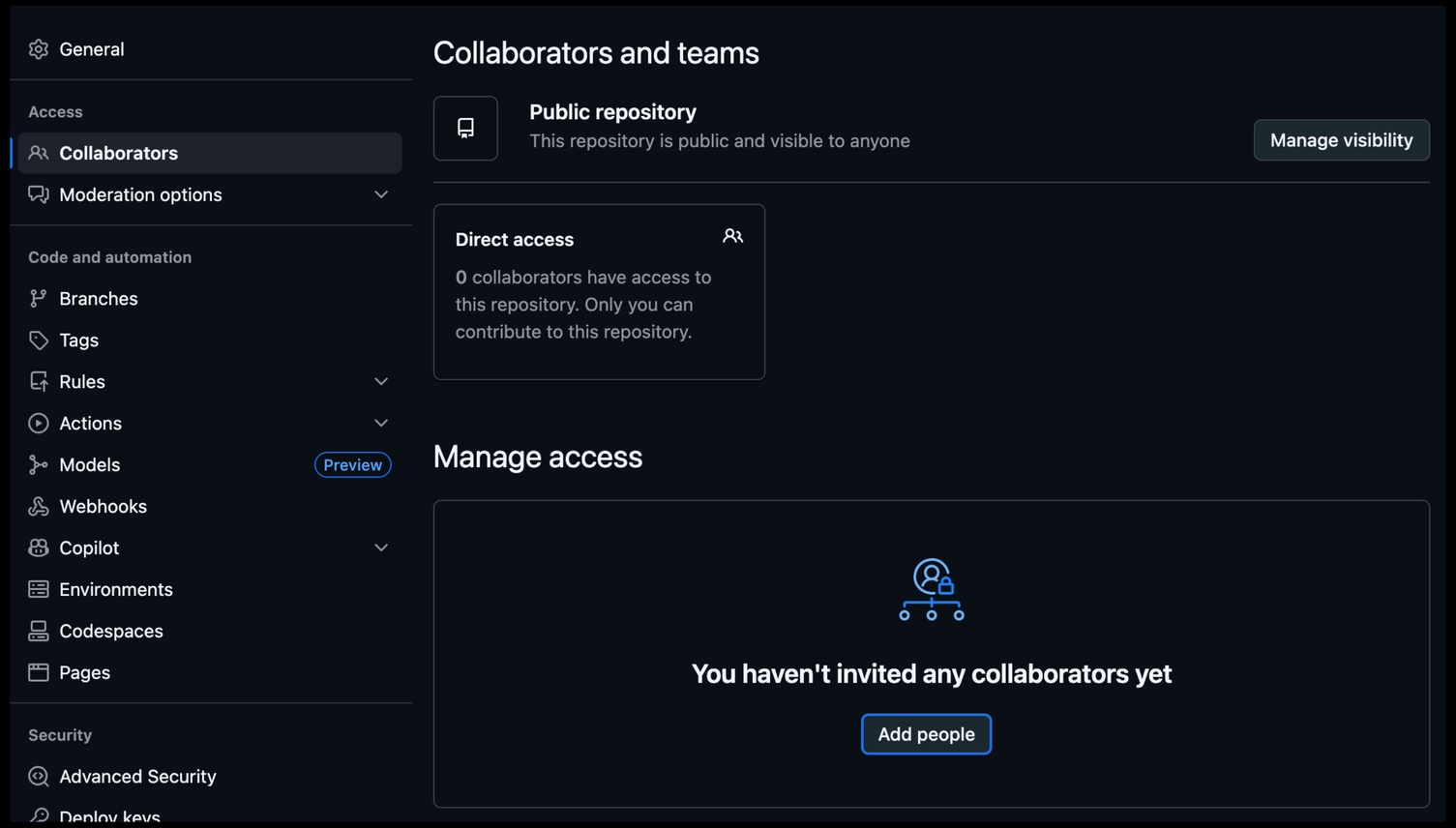
A collaborator (often the administrator) can **inspect differences** between main and the merged branch and choose to **accept or reject** them.

It might be necessary to **resolve merge conflicts** when accepting a pull request.



GitHub

Collaborating and sharing practice: collab setup



Add team members by user name:

In your repository on GitHub.com, navigate to Settings > Collaborators and add team members as **collaborators**. They will receive an e-mail to accept the invitation.

Collaborators have **permission** to make commits, create and rename branches, manage pull requests, issues and more.

Permissions can not be changed on free / private accounts.

GitHub

Collaborating and sharing practice: pull requests

When changes have been made to another branch than main (such as “conflict”), GitHub automatically suggests to create a pull request.

GitHub

Safety and privacy

Especially when working in a team, **NEVER commit**:

- Personal / sensitive data
- Passwords / keys
- Large data sets and large files
  - use .gitignore to enforce this!
  - Exclude subdirectories that store only data as well as potentially large container types such as .mp4, wav…

.gitignore

```
# exclude subject data
 subject_data/*

# exclude video and audio
*.mp4
*.wav
```

GitHub

Collaborating and sharing practice: Your turn!

1. Create a local repository `your_name_collab` with a simple text file with some random content and publish it to GitHub.
2. Become a **team** with the person next to you. Invite each other as **collaborators**.
3. **Clone** your teammate's repository and **commit** something to the **main** branch. Look at the changes in your own repository.
4. In your teammate's repository, create your own **branch**. Commit changes (edit, add…). When you are done, create a **pull request**.
5. In your own repository, **accept** the pull request and **merge** your teammate's branch into main.
6. In one of the repositories, **cause a conflict** by editing the same line of a file on main and another branch. Create a **pull request** for that branch and **resolve the conflict**.