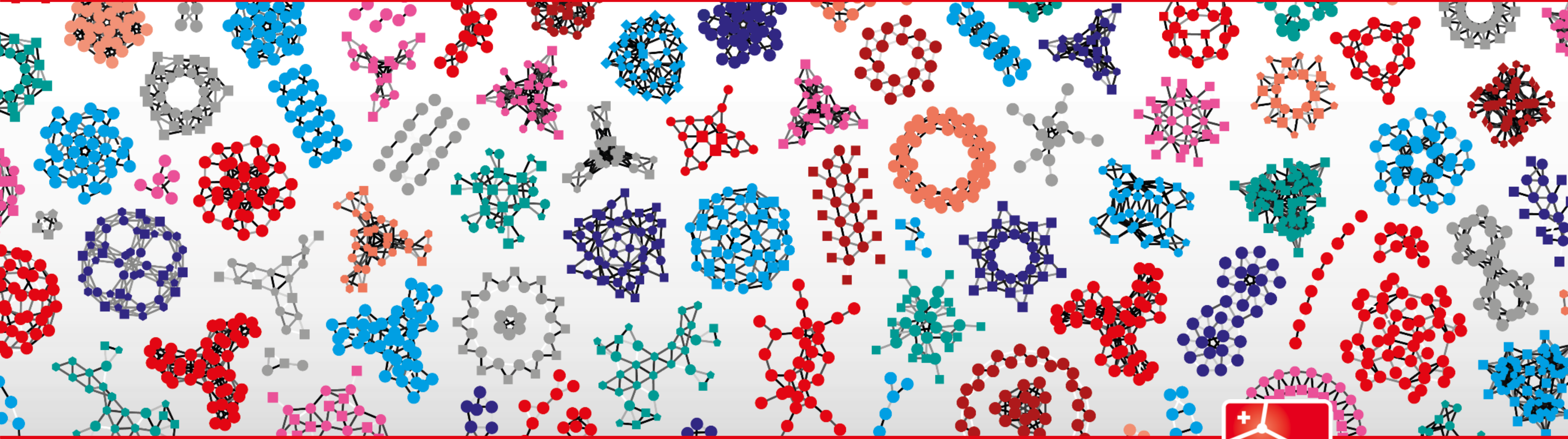


Version control with Git – optional modules

- Git submodules
- Git LFS
- Run automated pipelines with GitHub Actions and GitLab CI/CD

Swiss Institute of Bioinformatics

www.sib.swiss

Robin Engler
Vassilios Ioannidis

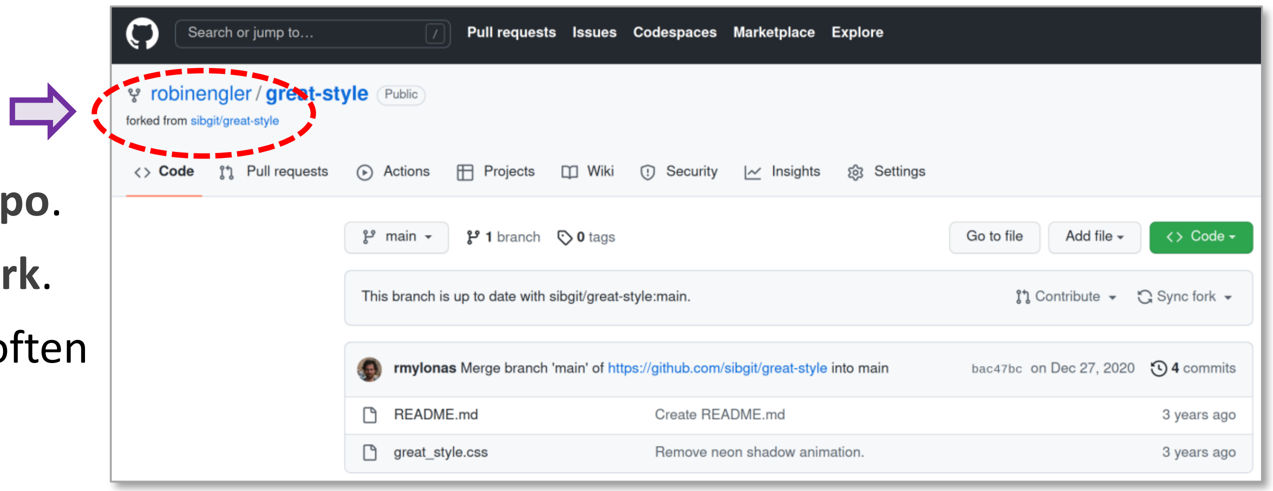
Lausanne, 11-13 Oct 2023

Working with **forks**

Get your own copy of a public repo,
contribute code without access to a project's repo

What are repository forks ?

- A **fork** is simply a **copy of an existing (original) Git repo**.
- Unlike the original repo, **you are the owner of the fork**.
- The original repo (from which the fork is derived) is often referred to as the **upstream**.



This branch is up to date with sibgit/great-style:main.

rmylonas Merge branch 'main' of https://github.com/sibgit/great-style into main bac47bc on Dec 27, 2020 4 commits

README.md	Create README.md	3 years ago
great_style.css	Remove neon shadow animation.	3 years ago

Why is forking useful ?

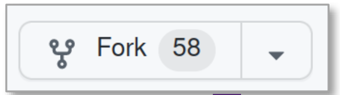
- Allows to create a copy of the repo of which **you are the owner**, and on which you therefore have **write access** (so you can push commits).
- Forking a repo and then making a pull request in GitHub (or a merge request in GitLab) is the **standard way of contributing to open source projects** (since the owners of the project don’t know you and will not give you write access to their repo).

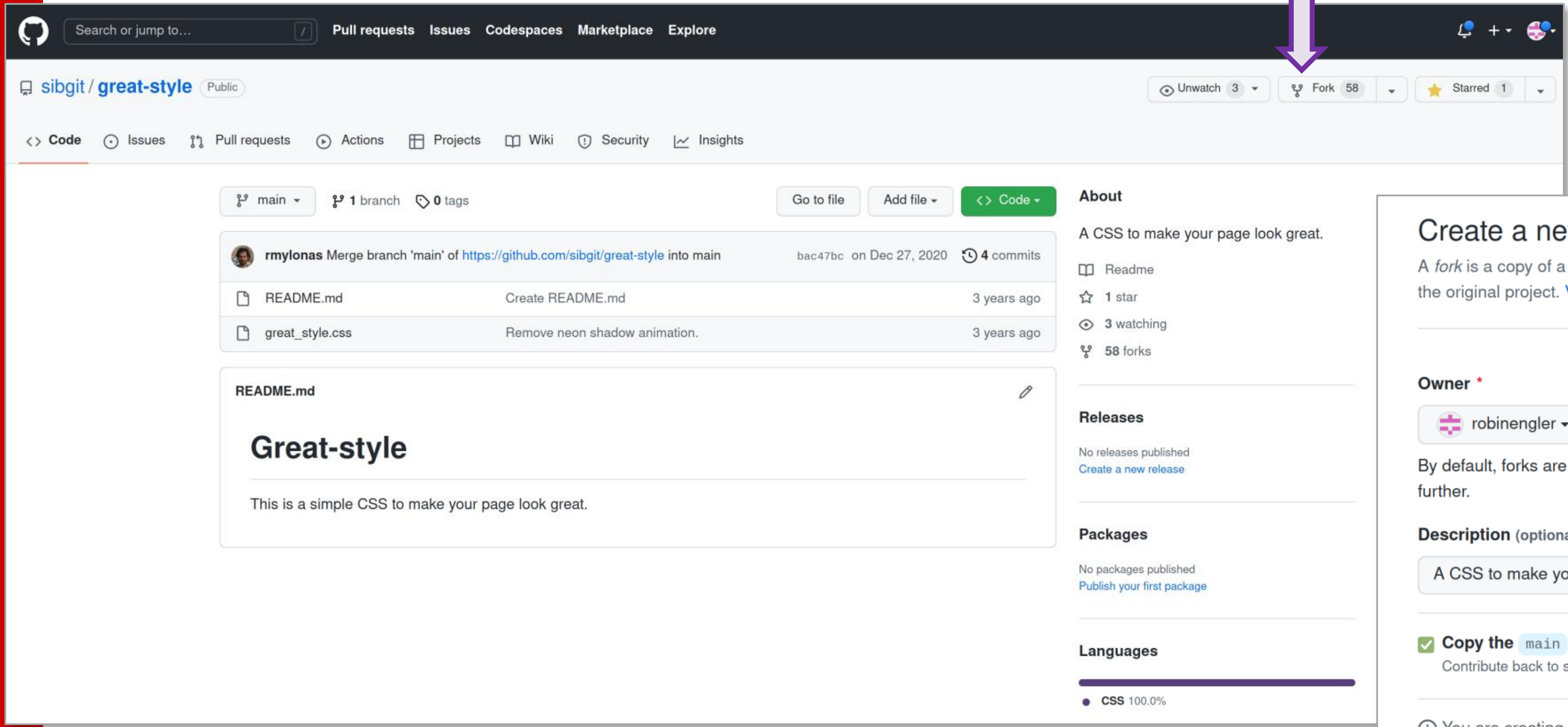
Forking on GitHub

You will need to do this in the optional exercise "submodules"

To create a fork of someone's project:

1. Go to their repo on GitHub and click on the **Fork** button:





rmylonas Merge branch 'main' of https://github.com/sibgit/great-style into main bac47bc on Dec 27, 2020 4 commits

README.md	Create README.md	3 years ago
great_style.css	Remove neon shadow animation.	3 years ago

README.md

Great-style

This is a simple CSS to make your page look great.

About

A CSS to make your page look great.

Readme
1 star
3 watching
58 forks

Releases

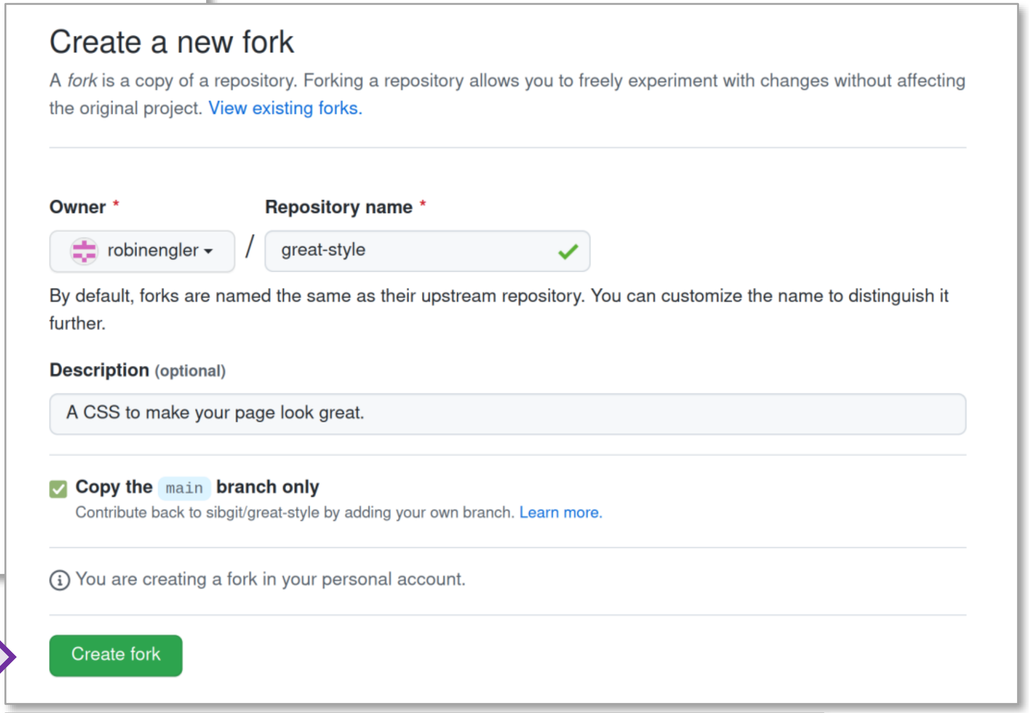
No releases published
Create a new release

Packages

No packages published
Publish your first package

Languages

CSS 100.0%



Create a new fork

A *fork* is a copy of a repository. Forking a repository allows you to freely experiment with changes without affecting the original project. View existing forks.

Owner * robinengler / Repository name * great-style ✓

By default, forks are named the same as their upstream repository. You can customize the name to distinguish it further.

Description (optional)

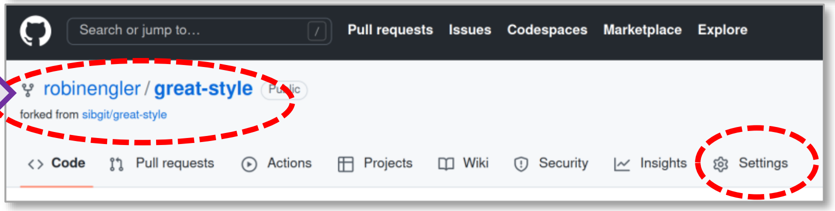
A CSS to make your page look great.

✓ Copy the `main` branch only
Contribute back to sibgit/great-style by adding your own branch. Learn more.

You are creating a fork in your personal account.

Create fork

2. A new page will open, where you can leave all values to their default and simply click on **Create fork**.



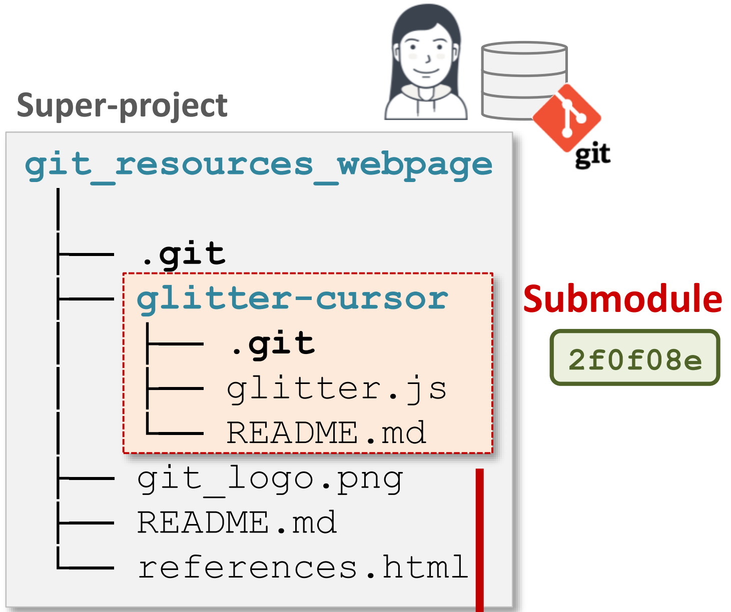
3. **Done**: you now have a copy of the repo under your own username and are the owner of that copy.

git submodules

The "symlink" of Git repositories

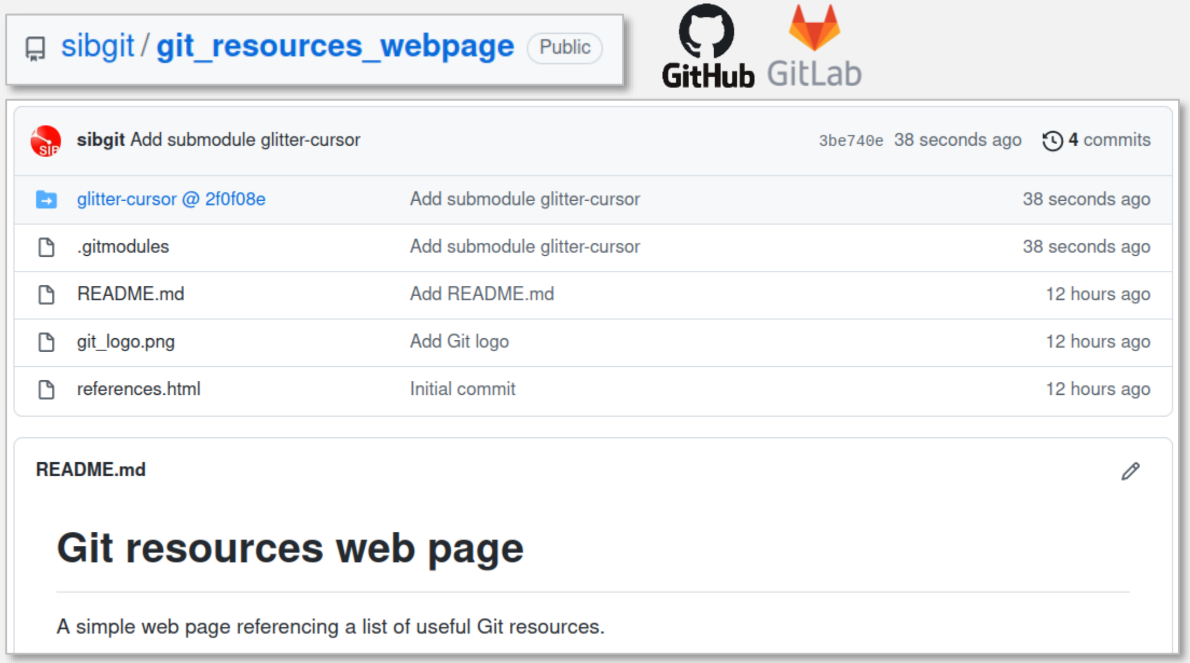
What are submodules ?

- Git submodules allow keeping a Git repository (the "submodule") as a **subdirectory** of another Git repository (the "super-project") **while version controlling the version of the submodule**.
- The "super-project" and the submodule remain independent repos, and have independent remotes.



```
git_resources_webpage
 │
 ├── .git
 ├── glitter-cursor
 │   ├── .git
 │   ├── glitter.js
 │   └── README.md
 ├── git_logo.png
 ├── README.md
 └── references.html
```

Main repository / super-project (repo containing the submodule)



sibgit / **git_resources_webpage** Public

GitHub GitLab

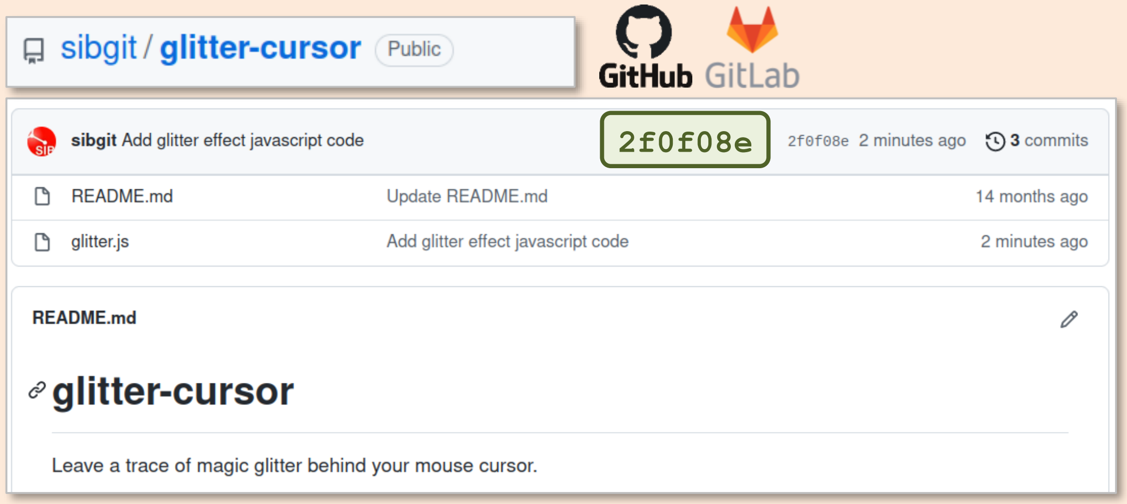
sibgit Add submodule glitter-cursor		3be740e 38 seconds ago 4 commits
glitter-cursor @ 2f0f08e	Add submodule glitter-cursor	38 seconds ago
.gitmodules	Add submodule glitter-cursor	38 seconds ago
README.md	Add README.md	12 hours ago
git_logo.png	Add Git logo	12 hours ago
references.html	Initial commit	12 hours ago

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.

Subproject (project used as submodule in the super-project)



sibgit / **glitter-cursor** Public

GitHub GitLab

sibgit Add glitter effect javascript code	2f0f08e	2f0f08e 2 minutes ago 3 commits
README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

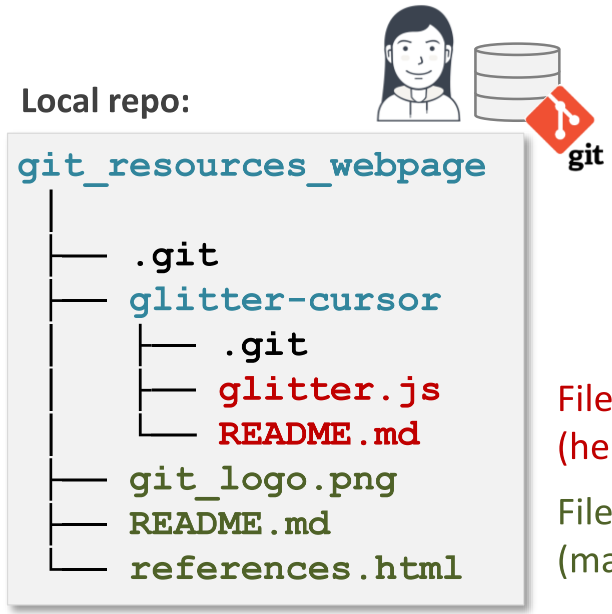
What are submodules (continued)

- A Git submodules is a **reference to another repository** at a **specific commit**.

  Important: the super-project does not keep track of individual files inside the submodule.

- Because the submodule is **fixed at a specific commit** (unless explicitly changed), the maintainer of the super-project has **full control of which revision of the submodule's code they are using**.

Local repo:

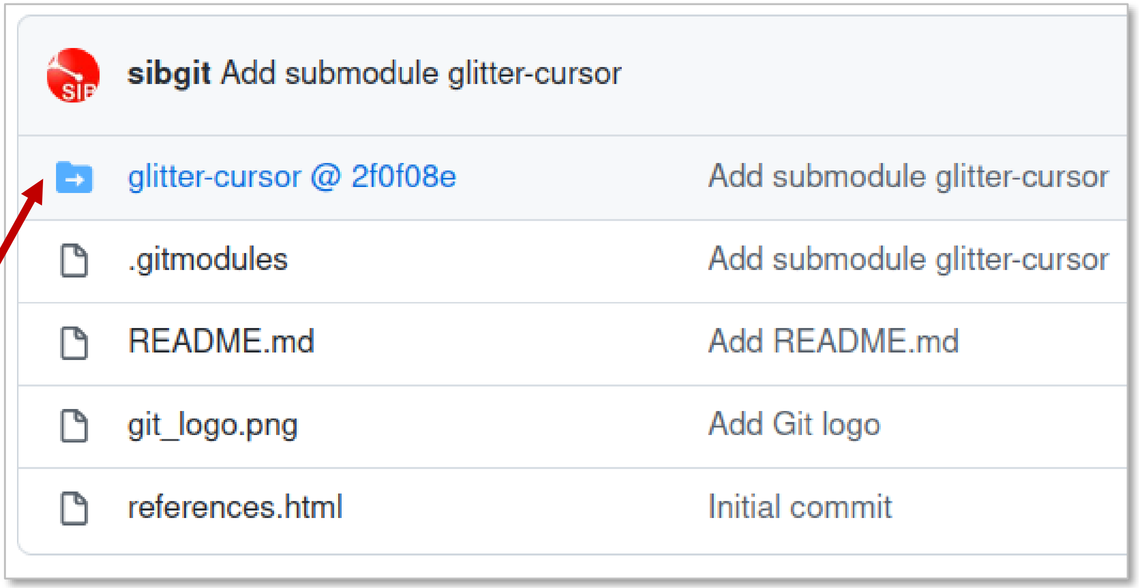


```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── references.html
```

Files tracked by the **subproject** (here used as a submodule)

Files tracked by the **super-project** (main project)

On GitHub/GitLab, submodules are shown with the syntax:
<submodule dir name>@<commit hash>



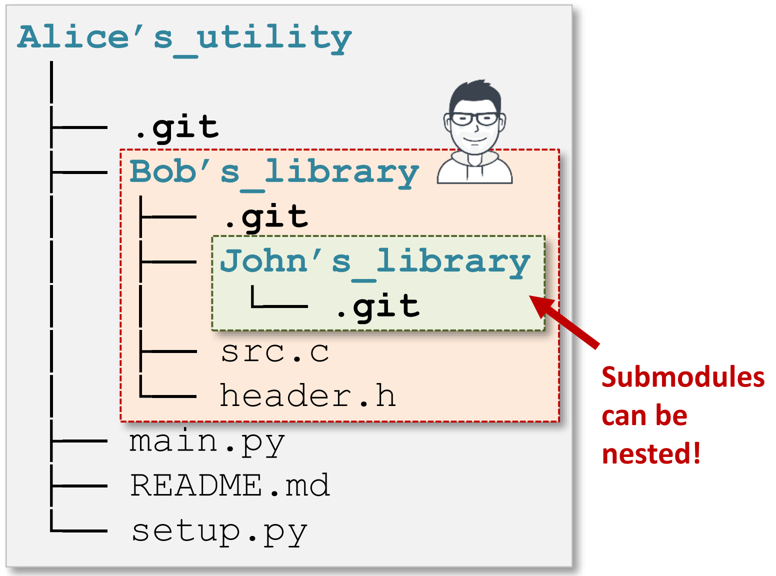
sibgit Add submodule glitter-cursor

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

Use cases: when to use submodules

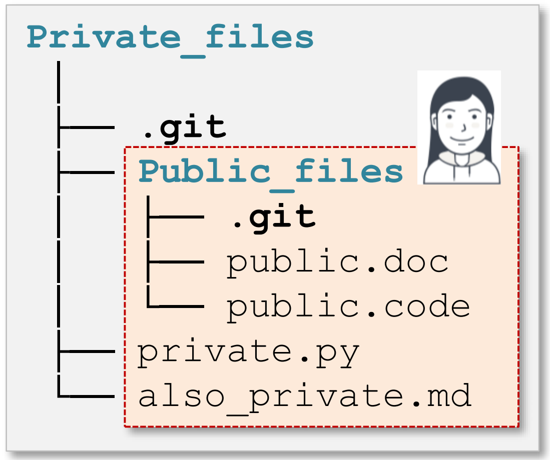
- **To include external code**, i.e. code maintained by someone else (e.g. on GitHub/GitLab), into your project. With Git submodules you can easily integrate external code, get updates from the upstream, and stay in control of when the external code should be updated. Can also be used to re-use one of your own repos in multiple projects.
- **To make public only a part of a project**. You can put the part of your code/files that you want to make public in a submodule (with public access), and keep the rest of the code in a private repository.
- Large project that uses multiple subprojects maintained independently.

Alice uses a library maintained by Bob as a submodule



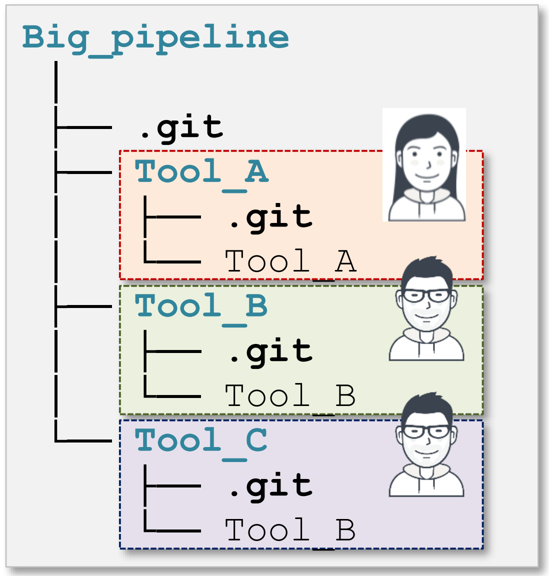
```
Alice's_utility
│
├── .git
├── Bob's_library
│   ├── .git
│   ├── John's_library
│   │   └── .git
│   ├── src.c
│   └── header.h
├── main.py
├── README.md
└── setup.py
```

Alice wants to mix public and private files in a project.



```
Private_files
│
├── .git
├── Public_files
│   ├── .git
│   ├── public.doc
│   └── public.code
├── private.py
└── also_private.md
```

Large pipeline with multiple collaborators.



```
Big_pipeline
│
├── .git
├── Tool_A
│   ├── .git
│   └── Tool_A
├── Tool_B
│   ├── .git
│   └── Tool_B
└── Tool_C
    ├── .git
    └── Tool_B
```

When NOT to use submodules

- **Don’t use submodules unless really needed**, monolithic repositories are simpler to maintain.
- If you have a sub-project that you want to use in multiple projects, it might be more efficient to create a package instead. Most programming languages have a dedicated package managers/repositories (CRAN for R, npm for javascript, PyPI for Python, etc).
- If you simply want to have a nested Git repos on your local machine (but with no link between them), you can simply add the nested repo to the `.gitignore` file of the higher-level repo.

```
git_resources_webpage
│
├── .git
├── glitter-cursor
│   ├── .git
│   ├── glitter.js
│   └── README.md
├── git_logo.png
├── README.md
└── .gitignore
```

```
glitter-cursor/
test_outputs.tmp
```

If all you want is keeping a Git repo inside another one on your local computer with no link between them… you don’t need submodules – *save yourself the hassle*!

Adding/registering a submodule

To add/register a new submodule inside a Git repo:

```
git submodule add <URL of submodule repository>
```

This will:

- Add a new directory named after the submodule's repo name.
- Download the content of the submodule corresponding to the latest commit (on the default branch) into that directory.
- Create a `.gitmodules` file at the root of the super-project.

.gitmodules

```
[submodule "my-submodule"]
    path = my-submodule
    url = https://github.com/some-user/my-submodule.git
```

Local path of submodule

URL of submodule

- Initialize the submodule in the `.git/config` file.

.git/config

```
[submodule "my-submodule"]
   url = https://github.com/some-user/my-submodule.git
   active = true
```

"active = true" --> module is initialized

- If you add multiple submodules, you will have multiple entries in `.gitmodules`.
- `.gitmodules` should be version controlled, so that other people who clone the project know where the submodule projects are from (Git stages this file by default when adding a new submodule).

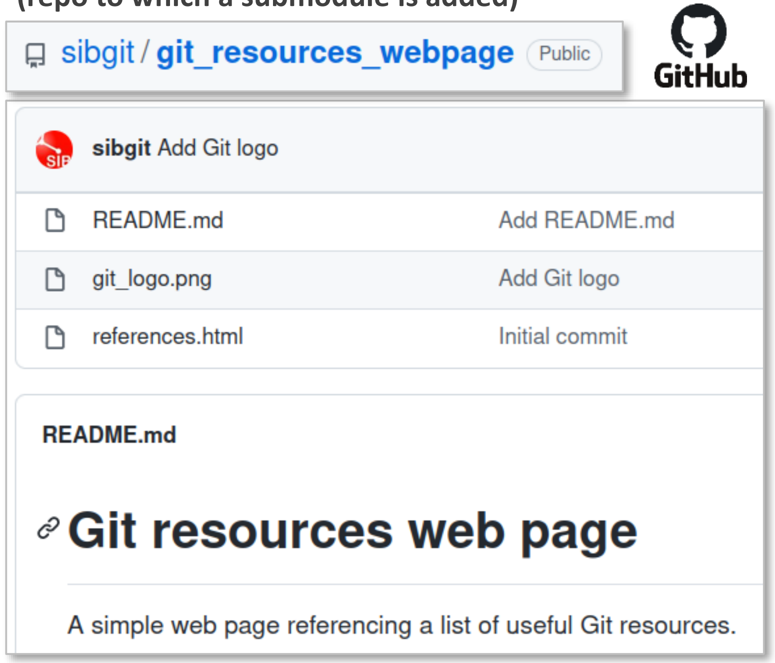
Submodule with custom name:

- Set custom name when adding submodule:
  `git submodule add <URL> <name>`
- Rename an exiting submodule:
  `git mv <submodule name> <submodule new name>`

Adding a submodule: example

Adding *glitter-cursor* as a submodule to *git_resources_webpage*

Main repository / super-project
(repo to which a submodule is added)



sibgit / git_resources_webpage Public

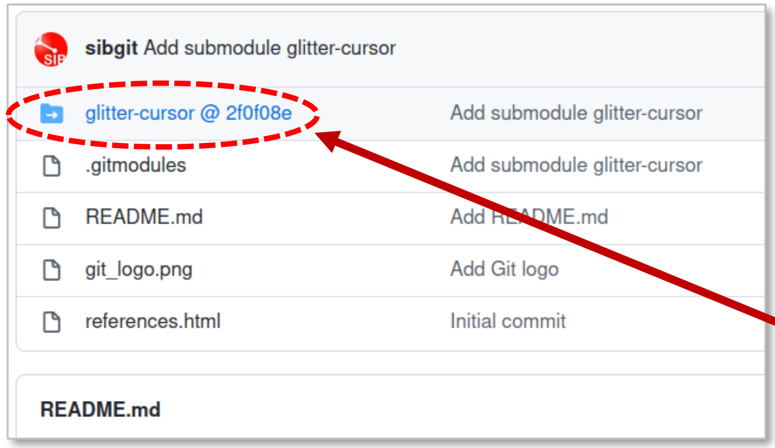
sibgit Add Git logo

README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.



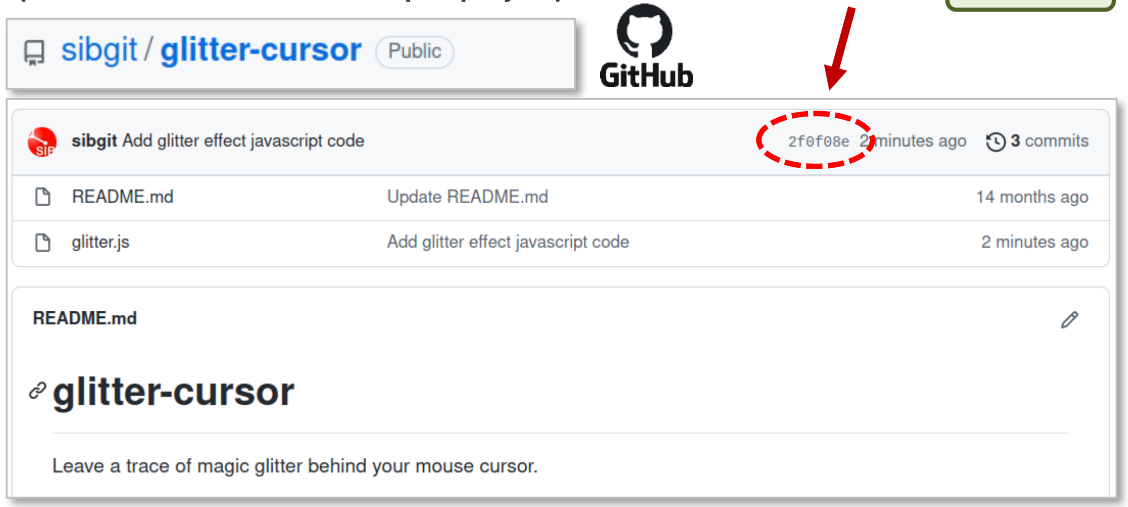
sibgit Add submodule glitter-cursor

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Subproject
(used as submodule in the super-project)

Repo is currently at commit `2f0f08e`



sibgit / glitter-cursor Public

sibgit Add glitter effect javascript code 2f0f08e 2 minutes ago 3 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

```
git submodule add https://.../glitter-cursor.git
git commit -m "Add submodule glitter-cursor"
git push
```

Icon and syntax indicating a submodule, which is pointing at `2f0f08e`

When a new submodule is added, it points at the latest commit of the submodule's online repository.

How Git keeps track of the submodule's version: some more details.

Adding "glitter-cursor" as a submodule to "git_resources_webpage"

```
$ cd git_resources_webpage
$ git submodule add https://github.com/sibgit/glitter-cursor.git
Cloning into '/home/.../git_resources_webpage/glitter-cursor'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (9/9), done.
```

`Git submodule add` does the following:

- Create a new directory named "glitter-cursor".
- Download the content of "glitter-cursor" corresponding to the latest commit (on the default branch).
- Create a `.gitmodules` file.

```
[submodule "glitter-cursor"]
    path = glitter-cursor
    url = https://github.com/sibgit/glitter-cursor.git
```

Local path of submodule

URL of submodule

- Initialize the submodule in the `.git/config` file.

```
[remote "origin"]
  url = https://github.com/sibgit/git_resources_webpage.git
  fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
  remote = origin
  merge = refs/heads/main
[submodule "glitter-cursor"]
  url = https://github.com/sibgit/glitter-cursor.git
  active = true
```

Section that was added

"active = true" --> module is initialized

How does Git keep track of the submodule's version ?

```
$ git status
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitmodules
        new file:   glitter-cursor
```

```
$ git diff --cached
diff --git a/.gitmodules b/.gitmodules
--- /dev/null
+++ b/.gitmodules
@@ -0,0 +1,3 @@
+[submodule "glitter-cursor"]
+       path = glitter-cursor
+       url = https://github.com/sibgit/glitter-cursor.git
diff --git a/glitter-cursor b/glitter-cursor
--- /dev/null
+++ b/glitter-cursor
@@ -0,0 +1 @@
+Subproject commit 2f0f08e991d828dd27cf399c0b88edaaa48a2bf9
```

2f0f08e

- The submodule is tracked/added as a "virtual file" to the index.
- This "virtual file" contains the **commit ID** (SHA-1 checksum) to which the submodule is pointing (and nothing else).
- Individual files in the submodule are <u>not</u> tracked by the super-project.

Clone a repository with submodules

```
git clone <repository>
git submodule init
git submodule update
```

or

```
git clone <repository>
git submodule update --init --recursive
```

or

```
git clone --recurse-submodules <repository>
```

Shortcut to clone, initialize and update all submodules.

This is what you will want to use in most situations.

- After cloning a repository that contains submodules, there will only be an empty directory for the submodules: their content is not automatically downloaded!
- You have to initialize* the local configuration files with: `git submodule init`
- Now the content of submodule(s) can be retrieved** with: `git submodule update`
- `--recursive` / `--recurse-submodules` means that the command also applies to nested submodules (submodules within submodules).

Notes:

- By default, the commands `git submodule init/update` apply to all submodules of a project. To apply them only to a specific submodule, the name of the submodules can be passed: e.g. `git submodule init <submodule name>`
- * What does **initialize** a submodule mean, and what exactly does `git submodule init` do?
  When Git initializes a submodule, it creates an entry for it in the `.git/config` file of the superproject repo and marks it as "active = true".
  When working on a large project with many submodules, this makes it e.g. possible to only initialize those submodules that are really needed for your work.

`.git/config`

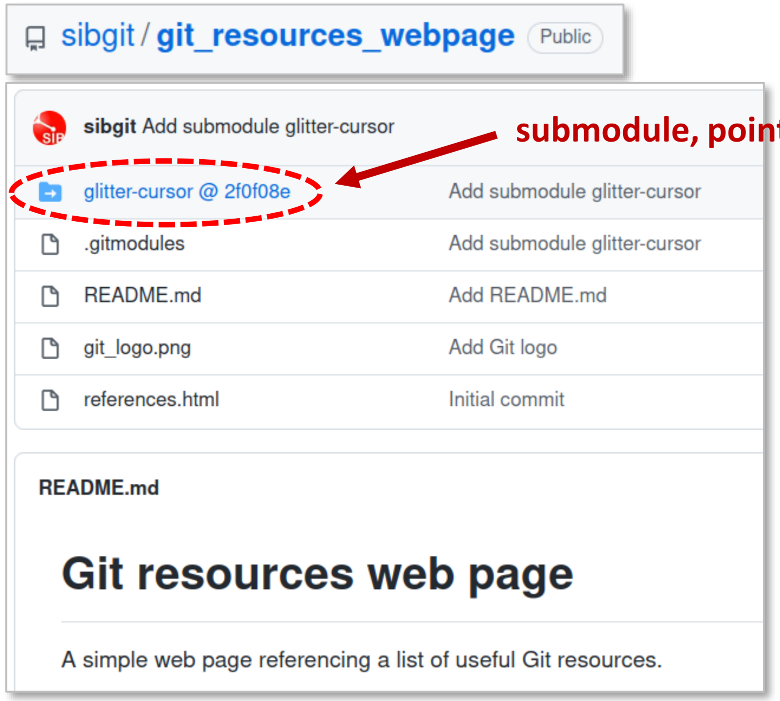
```
[submodule "glitter-cursor"]
    active = true
    url = https://github.com/sibgit/glitter-cursor.git
```

- ** The meaning of **update** in `git submodule update` is to fetch updates in submodules and update the working tree of the submodules to the revision expected by the superproject. It does not mean to update the submodules to their latest version.

Clone a repository with submodules: example

Cloning *git_resources_webpage* that contains the submodule *glitter-cursor*.

Online main repository / super-project
(repo that contains a submodule)



sibgit / **git_resources_webpage** Public

sibgit Add submodule glitter-cursor	
glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

README.md

Git resources web page

A simple web page referencing a list of useful Git resources.

submodule, pointing at `2f0f08e`

```
git clone
https://.../git_resources_webpage.git
```

Local copy of repository

```
git_resources_webpage
  │
  ├── glitter-cursor
  ├── git_logo.png
  ├── README.md
  └── references.html
```

Directory is empty !

```
git submodule init
```

Initializes/activates the submodule(s) in `.git/config`

```
git submodule update
--init --recursive
```

```
git submodule update
```

Downloads submodule content

```
git_resources_webpage
  │
  ├── glitter-cursor
  │     ├── glitter.js
  │     └── README.md
  ├── git_logo.png
  ├── README.md
  └── references.html
```

Now the files of the submodule are locally available.

Shortcut !

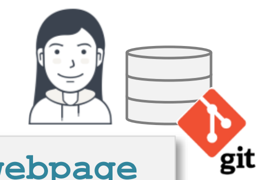
```
git clone --recurse-submodules
https://.../git_resources_webpage.git
```

Cloned submodules are (by default) in detached HEAD state

- After cloning a repo (superproject) with submodules, the submodules are in **detached HEAD** state.
- To make it point to a branch you have to explicitly checkout (switch to) that branch.

```
$ cd glitter-cursor
$ git status
HEAD detached at 2f0f08e
$ git switch main
```

Commit the submodule is currently pointing at.



```
git_resources_webpage
  │
  ├── .git
  ├── glitter-cursor
  │     ├── .git
  │     ├── glitter.js
  │     └── README.md
  ├── git_logo.png
  ├── README.md
  └── references.html
```

To display the revision of the submodule to which a super-project is currently pointing:

```
$ git submodule status
2f0f08e991d828dd27cf399c0b88 glitter-cursor (heads/main)
```

Update a repository with submodules (git pull on the super-project)

Similarly to `git clone`, running `git pull` in the super-project (the main project that hosts the submodule) does *not* automatically update the submodules' content. You need to either:

```
git pull
git submodule update --init --recursive
```

Downloads the submodule's updated content

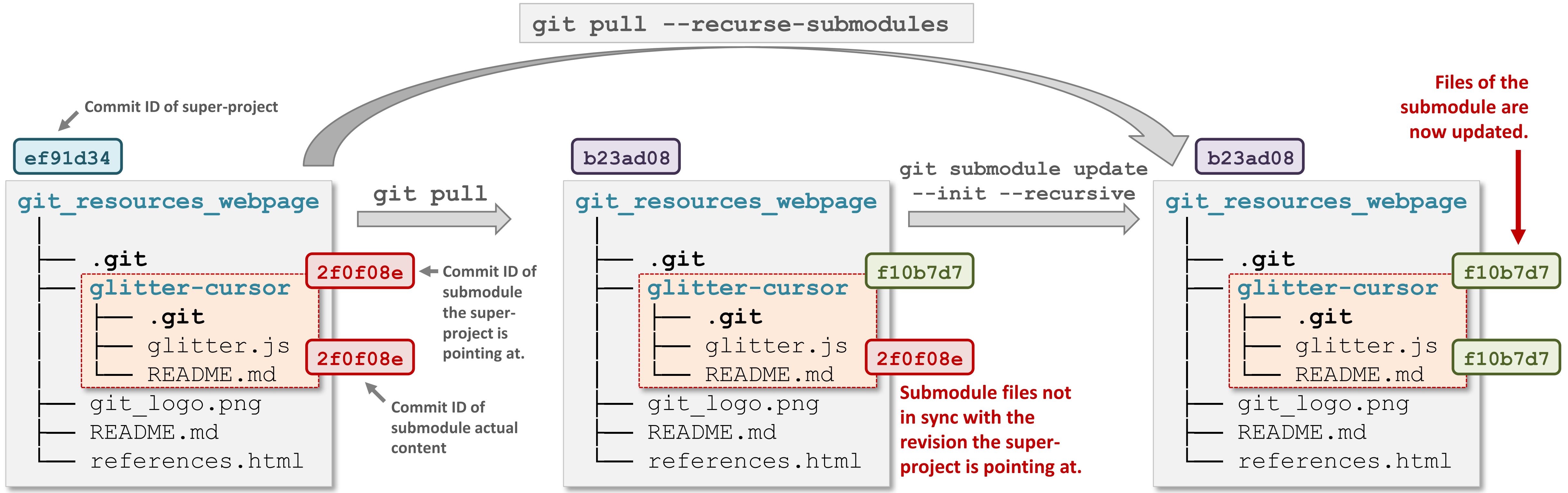
or

```
git pull --recurse-submodules
```

Shortcut !

This is what you will want to use in most situations.

Important: these are commands to run in the super project!



Working with submodules

- Submodules are regular Git repos. Once inside, you can run the same Git commands as you would on any repo.

Example:

```
$ cd glitter-cursor

# We are now in the submodule directory.
$ git status
$ git add ...
$ git commit ...
$ git push
```

- The super-project does not keep track of individual files in the submodule: it only keeps track of the commit to which it points.

  However, the super-project will detect when changes are made inside a submodule (but not exactly which changes).

Example: files were added/modified in the submodule.

```
$ git status   # run in the super-project's root!

Changes not staged for commit:
      modified: glitter-cursor (modified content,
                                untracked content)
```

Example: one or more new commits in submodule.

```
$ git status   # run in the super-project's root!

Changes not staged for commit:
      modified: glitter-cursor (new commits)
```

- To run the same tasks on multiple submodules, there is the handy command:

```
git submodule foreach "git command"
```

Example:

```
$ git submodule foreach "git status"
$ git submodule foreach "git log --oneline"
Entering 'glitter-cursor'
2f0f08e (HEAD -> main) Add glitter effect code
841e83a Update README.md
b0b66f8 Initial commit
```

Making changes to a submodule (modifying the content of the submodule)

Let’s assume we want to **modify the content** of a submodule, for instance:

- Update the submodule’s content to a newer version.
- Make changes to files in the submodule.
- Point the submodule at an older version.

We proceed as follows:

1. Make the desired changes in the submodule.
   If needed, pull/push the changes from/to the submodule’s remote.

Commands run in the **submodule**:

```
$ cd glitter-cursor
```

```
$ git pull
```

```
$ git add ...
$ git commit ...
$ git push
```

```
$ git checkout ...
```

2. The commit ID (hash) of the submodule has now changed, so we must update the super-project by making a new commit that will indicate the update in commit ID of the submodule.

Commands run in the **super-project**:

```
$ git status
On branch main
Changes not staged for commit:
      modified: glitter-cursor (new commits)
```

```
$ git diff
diff --git a/glitter-cursor b/glitter-cursor
--- a/glitter-cursor
+++ b/glitter-cursor
-Subproject commit 2f0f08e991d828dd27cf399c0b88edaaa48a2bf9
+Subproject commit f10d7b772342c6a9f31390af4f8a16f71c440777
```

New commit to which the submodule is now pointing

```
$ git add glitter-cursor
$ git commit -m "Update submodule glitter-cursor"
$ git push
```

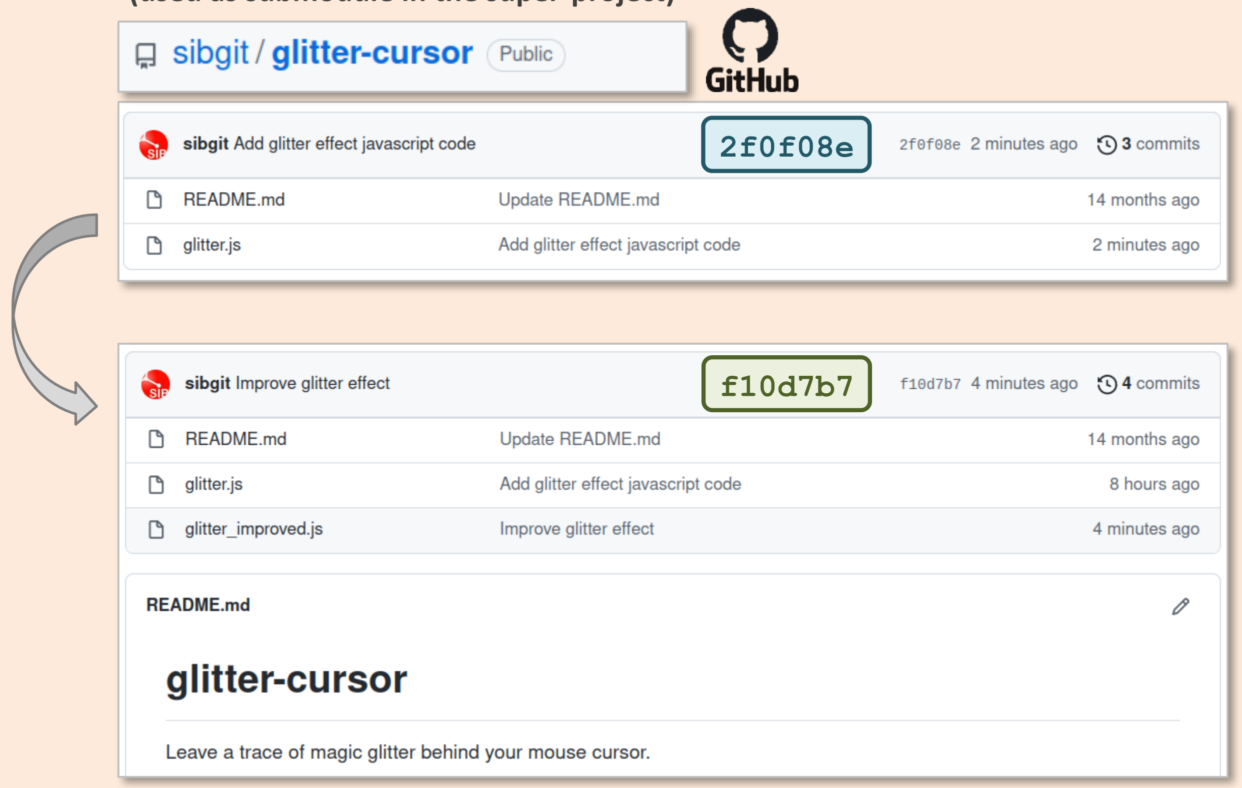
Making a new commit in the super-project

Making changes to a submodule

How things look on the GitHub pages of the remotes

```
git_resources_webpage
  │
  ├── .git
  ├── glitter-cursor
  │   ├── .git
  │   ├── glitter.js
  │   └── README.md
  ├── git_logo.png
  ├── README.md
  └── references.html
```

Subproject
(used as submodule in the super-project)



sibgit / **glitter-cursor** Public

sibgit Add glitter effect javascript code — **2f0f08e** — 2f0f08e 2 minutes ago — 3 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	2 minutes ago

sibgit Improve glitter effect — **f10d7b7** — f10d7b7 4 minutes ago — 4 commits

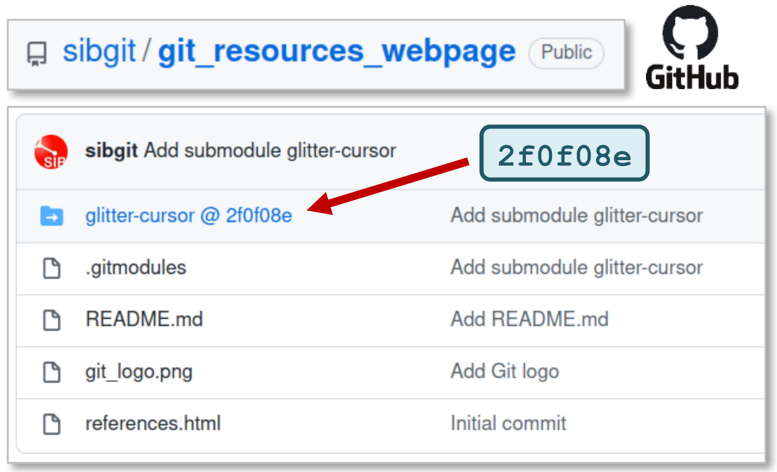
README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	8 hours ago
glitter_improved.js	Improve glitter effect	4 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

Main repository / super-project (repo containing the submodule)

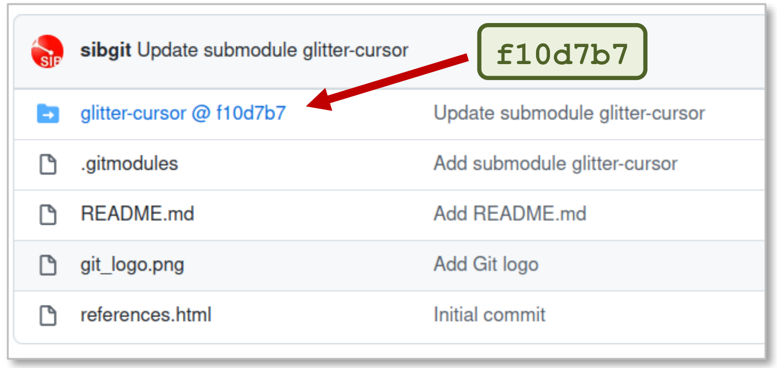


sibgit / **git_resources_webpage** Public

sibgit Add submodule glitter-cursor — **2f0f08e**

glitter-cursor @ 2f0f08e	Add submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

git push



sibgit Update submodule glitter-cursor — **f10d7b7**

glitter-cursor @ f10d7b7	Update submodule glitter-cursor
.gitmodules	Add submodule glitter-cursor
README.md	Add README.md
git_logo.png	Add Git logo
references.html	Initial commit

`--recurse-submodules` option: automated submodules push

To avoid accidentally forgetting to push changes in a submodule when pushing in the super-project:

- `git push --recurse-submodules=check` : safeguard that will make your push fail is there are any "non-pushed" changes in submodules.
- `git push --recurse-submodules=on-demand` : automatically push all submodules when pushing the super-project.

- These options can also be permanently set in the Git configuration of the super-project:

```
$ git config push.recurseSubmodules check
# or
$ git config push.recurseSubmodules on-demand
```

Note: we are not using the `--global` option, so this setting only affects the current repo.

- **Important:** all these commands must be run in the context (directory) of the super-project, not of the submodule!

Examples:

```
$ git push --recurse-submodules=check
The following submodule paths contain changes that
cannot be found on any remote:
    submodule-name

Please try
git push --recurse-submodules=on-demand
```

```
$ git push --recurse-submodules=on-demand
Pushing submodule 'submodule-name'
...
Pushing super-project (main project)
...
```

Pulling updates for a submodule

```
git submodule update --remote <submodule name>
```

If no submodule is specified, all submodules are updated

Updating a submodule to its latest commit

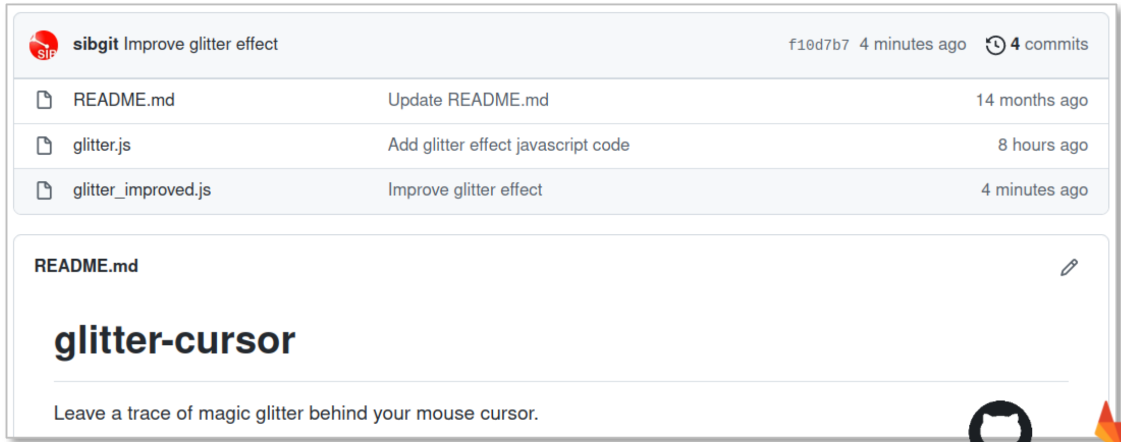
1. Bob, the maintainer of the *glitter-cursor* repo, pushes a new update.
2. Alice updates her submodule in the *git_resources_webpage* project with Bob's new update.

`f10d7b7`

```
glitter-cursor
 |
 ├── .git
 ├── glitter.js
 └── README.md
```

```
$ git push
```

`2f0f08e` → `f10d7b7`


sibgit Improve glitter effect f10d7b7 4 minutes ago 4 commits

README.md	Update README.md	14 months ago
glitter.js	Add glitter effect javascript code	8 hours ago
glitter_improved.js	Improve glitter effect	4 minutes ago

README.md

glitter-cursor

Leave a trace of magic glitter behind your mouse cursor.

GitHub GitLab

```
git_resources_webpage
 |
 ├── .git
 ├── glitter-cursor
 |   ├── .git
 |   ├── glitter.js
 |   └── README.md
 ├── git_logo.png
 └── references.html
```

`2f0f08e`

`2f0f08e` → `f10d7b7`

```
$ git submodule update --remote
Submodule path 'glitter-cursor': checked out f10d7b77...
```

To complete the update, Alice updates the super-project with a new commit that will make it point to the submodule commit: `f10d7b7`

```
$ git status
      modified: glitter-cursor (new commits)
$ git add glitter-cursor
$ git commit -m "Update submodule to latest version"
```

Pulling updates for a submodule (command details)

Updating a submodule to its latest commit

To pull the latest changes for a submodule:

```
git submodule update --remote <submodule name>
```

- If no submodule is specified, all submodules are updated.
- If the local submodule has diverged from its remote (e.g. you made some commits), `--merge`/`--rebase` must be added to the command to either merge or rebase.

```
$ git submodule update --remote --merge
```

- By default Git will try to pull the changes from the main branch. To pull from another branch, you have to specify it in `.gitmodules` by setting the parameter **branch**.

`.gitmodules`

```
[submodule "glitter-cursor"]
    path = glitter-cursor
    url = https://.../glitter-cursor.git
    branch = master
```

- After the content of the submodule is updated, the update in its version (commit hash) must still be committed.

```
$ git status
      modified: my-submodule (new commits)
$ git add my-submodule
$ git commit -m "Update submodule to latest version"
```

Alternatively, the pull in the submodule can also be done **manually**:

```
$ cd my-submodule
$ git switch main     # If in DETACHED HEAD state.
$ git pull
```

Supplementary material...

exercise 5

The Git reference web page gets better with submodules

git LFS

large file storage

Tracking large files can be useful…

Tracking large files together with code is an attractive proposition, e.g. in scientific applications:

- Data analysis/processing pipeline.
- Machine learning applications (training data and code in the same place).

… but Git does not work well with large files

- Git was designed for tracking code – i.e. relatively small text files.
- Adding large files to a Git repo is technically possible, however:
  - Since Git is a distributed VCS (version control system), **each local copy of a repository will contain a full copy of all versions of all tracked files**. Therefore, **adding large files will quickly inflate the size of everyone's repository**, resulting in higher disk space usage (on local hosts).
  - Git's internal data compression (i.e. packfiles) is **not optimized to work with binary data** (e.g. image or video files). Each change to a binary file will (more or less) add the full size of the file to the repo, taking disk space and slowing down operations such as repo cloning or update fetching.
  - Commercial **hosting platforms impose limits on the size of files** that can be pushed to hosted Git repos (GitHub: 100 MB, GitLab: no file limit but 10 GB repo limit).

The solution*: Git LFS

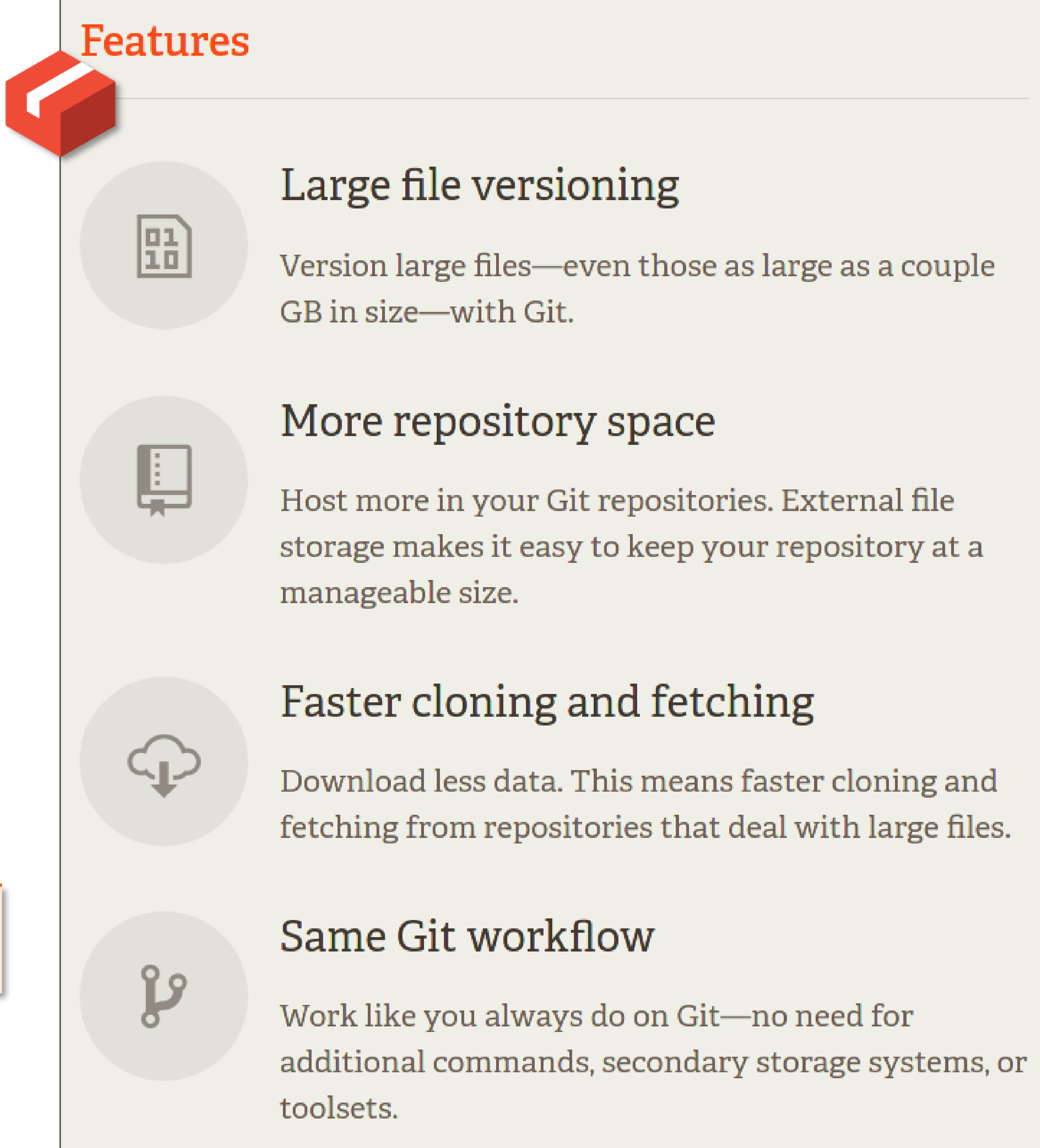
Git LFS (Large File Storage) is an extension for Git, specifically **designed to handle large files**.

Basic principle: large files are not stored in the Git database (the `.git` directory), instead:

- Only a **reference/pointer to large files** is stored in the Git database.
- The actual **files are stored in a separate repository** or "object store".

Open source project: https://git-lfs.github.com
First released in 2015.

Not all hosting services support Git LFS, and when they do, storage space is limited (additional space may be purchased).



Features

Large file versioning

Version large files—even those as large as a couple GB in size—with Git.

More repository space

Host more in your Git repositories. External file storage makes it easy to keep your repository at a manageable size.

Faster cloning and fetching

Download less data. This means faster cloning and fetching from repositories that deal with large files.

Same Git workflow

Work like you always do on Git—no need for additional commands, secondary storage systems, or toolsets.

* Alternatives to Git LFS exist, but Git LFS is the most popular.
Example: DVC – Data Version Control - https://dvc.org

GitHub and GitLab disk quotas, file size limit and pricing

- If your institution is running their own instance of GitLab, you can check with them if they offer LFS support (and how much space you can have their.
- Here are limits for 2 popular commercial Git hosting providers:

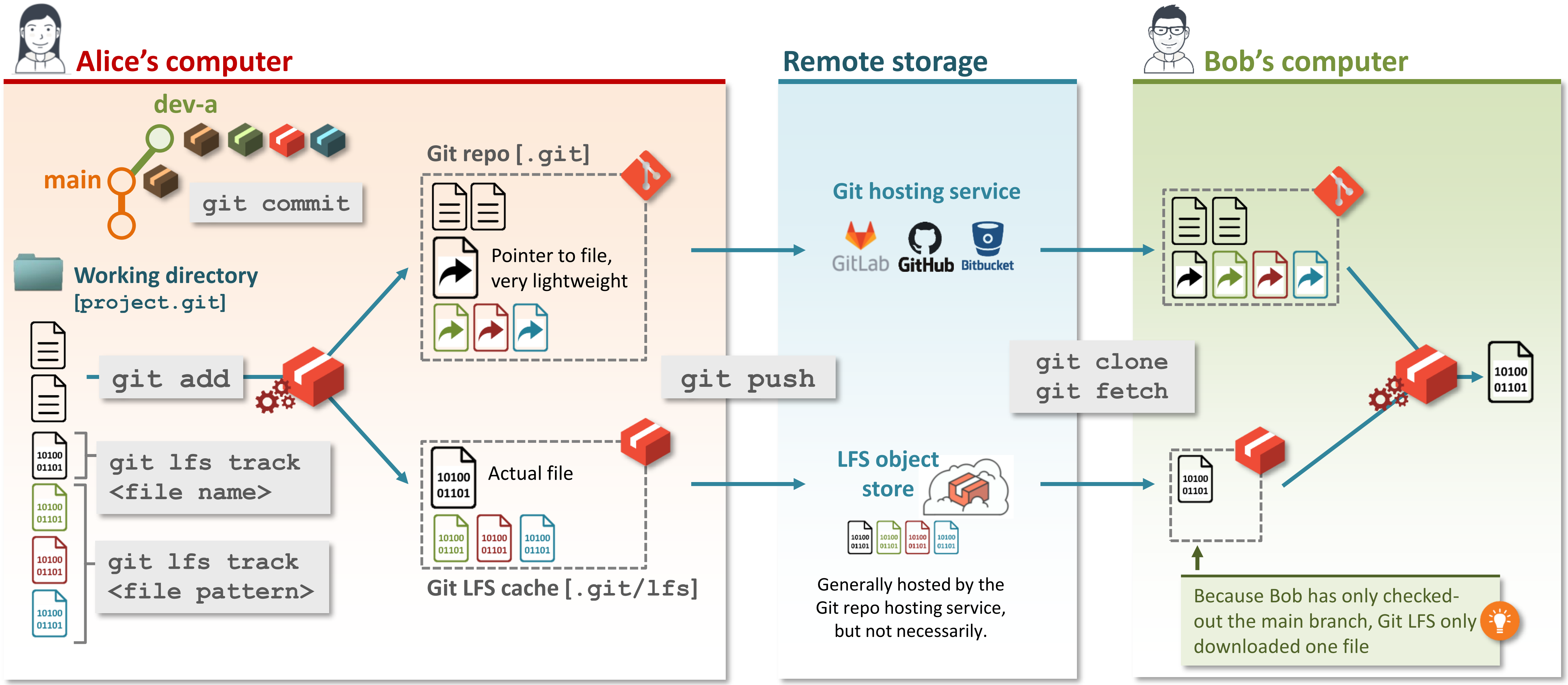
	GitHub.com	GitLab.com
Max file size	100 MB	No size limit
Max repo size	1 GB (recommended) 2 GB to 5 GB (max)	10 GB
LFS max file size	2 GB	No size limit (not sure)
LFS object store	1 GB storage for free 1 GB/month free bandwidth (download) 5 USD/month for each additional "pack" of 50 GB storage + 50 GB bandwidth	60 USD/year per 10GB

last updated on Feb 2021

- You can also setup a Git LFS object store on third-party storage provider - but you need to set it up yourself and it is *not* a trivial task:
  - SWITCHengines (220 CHF/TB*year) – no backup (need to organize your own).
  - AWS (amazon web services).

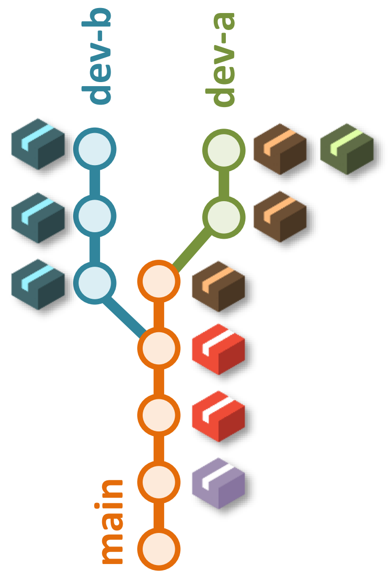
Git LFS workflow overview

- **Only a reference/pointer** to large files **is stored** in the Git database.
- The large files themselves are stored in a separate repository known as the “LFS object store”.
- Large files are downloaded only when needed.
- Transparent: **only 1 extra command** is needed for this workflow (`git lfs track`).



Alice’s computer
dev-a
main
git commit
Git repo [.git]
Pointer to file, very lightweight
Working directory [project.git]
git add
10100 01101
git lfs track <file name>
git lfs track <file pattern>
Actual file
Git LFS cache [.git/lfs]
Remote storage
Git hosting service
GitLab
GitHub
Bitbucket
git push
LFS object store
Generally hosted by the Git repo hosting service, but not necessarily.
Bob’s computer
git clone
git fetch
Because Bob has only checked-out the main branch, Git LFS only downloaded one file

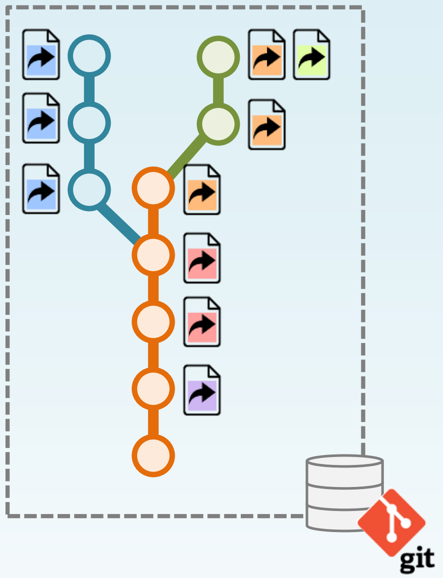
Complete Git history of project



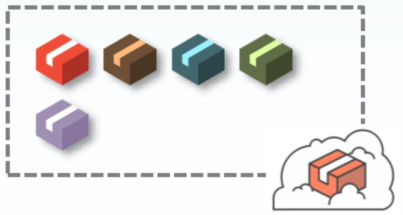
■ Large file. Colors represent different versions or different files.

Remote storage

Git database content



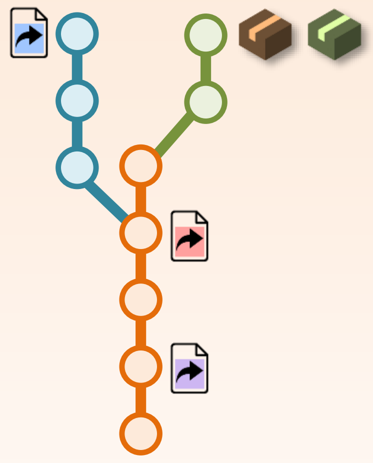
LFS object store content



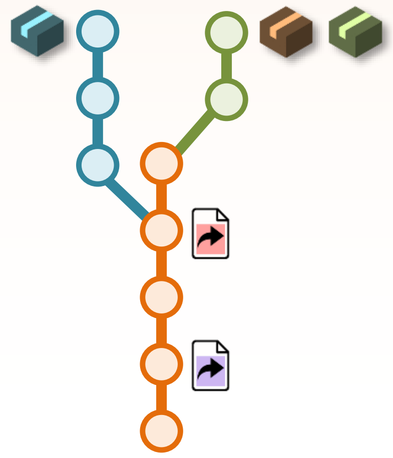
Local Git repositories

Alice's local repo

Alice just started to work on the project. She cloned the repo and created the "dev-a" branch.

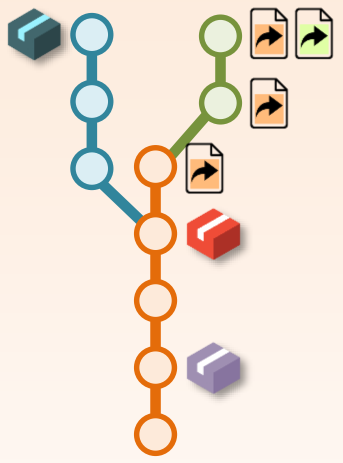


```
git checkout dev-b
```

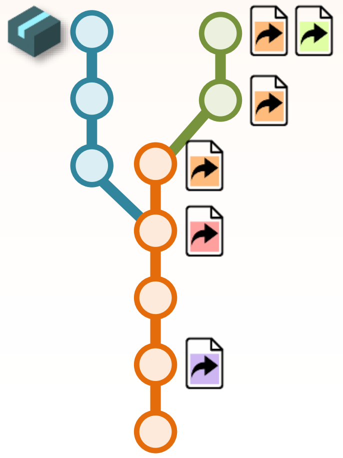


Bob's local repo

Bob contributed to the project since a while. He's currently working on "dev-b".



```
git lfs purge
```



Git LFS: initial setup

- One time setup: to be executed only once per user/machine, after Git LFS was installed.
  (this adds LFS Git filters to your global configuration file `~/.gitconfig`)

```
git lfs install
```

Git LFS: tracking files

- Adding files to Git LFS:

```
git lfs track <file name or pattern>
```

  - When using a file pattern (glob pattern), all files matching the pattern are tracked.
  - Each call to `git lfs track` creates a new entry in the `.gitattributes` file.

- Examples:

```
$ git lfs track file_1.csv
$ git lfs track file_2.csv file_3.csv
$ git lfs track "*.fasta"
$ git lfs track "*.img"
$ git lfs track "large_file_?.txt"
$ git lfs track "subdir/*.jpg"
```

Track the file named exactly “file_1.csv”

Track the files named exactly “file_2.csv” and “file_3.csv”

Track all files ending in “.fasta”

Track all files ending in “.img”

Track all files whose name are of the form “large_file_” + any single character + “.txt”

Track all files ending in “.jpg” in sub-directory “subdir”

Content of `.gitattributes`

```
file_1.csv filter=lfs diff=lfs merge=lfs -text
file_2.csv filter=lfs diff=lfs merge=lfs -text
file_3.csv filter=lfs diff=lfs merge=lfs -text
*.fasta filter=lfs diff=lfs merge=lfs -text
*.img filter=lfs diff=lfs merge=lfs -text
large_file_?.txt filter=lfs diff=lfs merge=lfs -text
subdir/*.jpg filter=lfs diff=lfs merge=lfs -text
```

It is also possible to edit directly the `.gitattributes` file instead of using the `git lfs track` command.

Do not forget "quotes" when using the `git lfs track` command with a file pattern, otherwise the pattern expands when the command is run and the matching files in your current working directory (rather than the pattern) are added to `.gitattributes`.

```
git lfs track "*.img"
```

✓

```
git lfs track *.img
```

✗

content of `.gitattributes` assuming that "file1.img" and "file2.img" are present in the working directory.

```
*.img filter=lfs diff=lfs merge=lfs -text
```

```
file_1.img filter=lfs diff=lfs merge=lfs -text
file_2.img filter=lfs diff=lfs merge=lfs -text
```

if we add a new file "file_3.img" at a later point in time...

✓ **File "file_3.img" is tracked** because it matches the *.img pattern.

✗ **File "file_3.img" is not tracked** because it matches neither file_1.img nor file_2.img.

- Recursively tracking an entire directory

```
git lfs track "directory_path/**"
```

Using `/**` is important.
Using `/` or `/*` will ***not*** work.

Content of `.gitattributes`

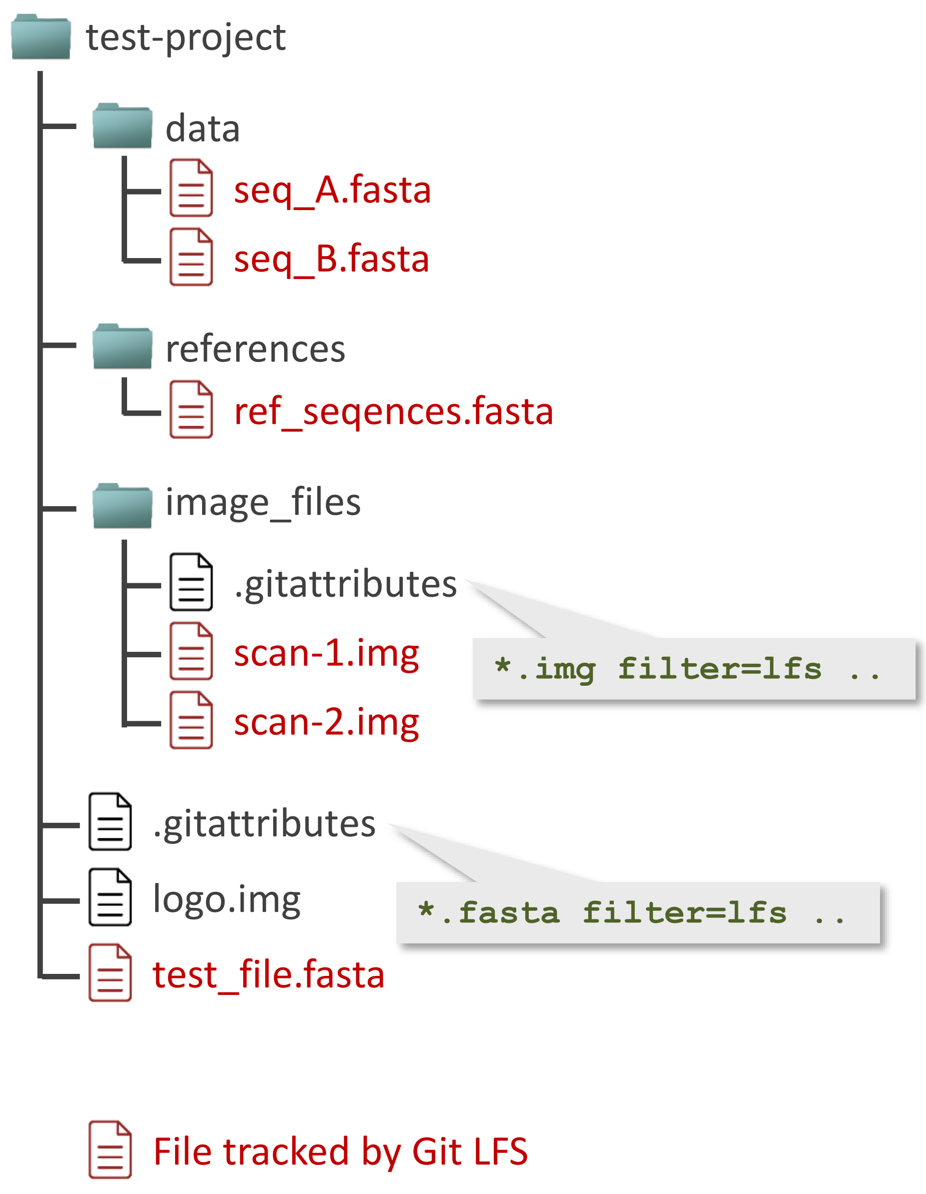
```
dir_to_track/** filter=lfs diff=lfs merge=lfs -text
```

Git LFS file tracking: fine-grained control

- For fine-grained control, `git lfs track <file name/pattern>` can be run in sub-directories. This places `.gitattributes` files in sub-directories (similar to how `.gitignore` files behave).
- The scope of each `.gitattributes` file is its current directory and sub-directories.
- Running `git lfs track <file name or pattern>` inside a sub-directory, creates the `.gitattributes` file inside that sub-directory

The `.gitattributes` file(s) in your repo should be tracked - just like `.gitignore` file(s).

Don't forget to commit them.



Negative pattern matching

- Unlike `.gitignore` files, `.gitattributes` files **do <u>not</u>** support the `!pattern` for negative matching (to tell Git LFS to not track a file).
- It is best to write `.gitattributes` files so that no negative matching is needed.
- If unavoidable, a workaround is possible by adding a line with the file/pattern that should not be tracked followed by `!filter !diff !merge` **after** the general pattern to track.

Example of `.gitattributes` file for tracking all ".jpg" files except "small_logo.jpg"

```
*.jpg filter=lfs diff=lfs merge=lfs -text
small_logo.jpg !filter !diff !merge
```

File that should not be tracked

Git LFS: untracking files

- Removing files from Git LFS:

```
git lfs untrack <file name or pattern>
```

  - Calls to `git lfs untrack` remove entries from the `.gitattributes` file.
  - The same result can be obtained by manually deleting lines from the `.gitattributes` file.

Git LFS: adding and committing files

- Nothing special to do!
- Once files are tracked by LFS, adding them to the git repo and committing them is done as usual.

```
git add ...
```

```
git commit ...
```

```
git push ...
```

Git LFS: updating files

- Nothing special to do!
- Files tracked by Git LFS can be updated, staged and committed like any file under Git control.

```
$ git add sequence_db.fasta
$ git commit -m “updated sequence database file”
$ git push
```

The new version of the file is added to the local Git LFS cache. The pointer file is updated.

The new version of the file is pushed to the remote LFS object store.

- After commits are pushed, the remote Git LFS object store contains a copy of each version of all LFS-tracked files.

Data backup

The idea behind Git LFS is to ***avoid replicating large data files*** across local copies of a Git repository. This has implications for data-backup:

- For LFS-tracked files, local repos *cannot* be relied-upon to contain a full copy of all data.
- Therefore the remote repository has to be backed-up.

In addition, keep in mind that, depending on the data you are working with, there might be legal aspects to consider (e.g. data might have to be stored encrypted, or be stored within the country)

Using Git LFS: diff-ing files

- For LFS-tracked files, `git diff` will only show the difference between pointer files, not between actual file content (even for text files).

```
git diff HEAD~1 sequences_A.fasta
diff --git a/sequences_A.fasta b/sequences_A.fasta
index a33c8a7..01f8d67 100644
--- a/sequences_A.fasta
+++ b/sequences_A.fasta
@@ -1,3 +1,3 @@
 version https://git-lfs.github.com/spec/v1
-oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
-size 6010948
+oid sha256:fc51c1860c4341e175dcfc24fc2c653f75c5e8b3bae6cf80d3632788ccaf4379
+size 6011029
```

size of file in bytes

checksum (SHA-256) of file content.

Listing files tracked by Git LFS

- List LFS-tracked files of **HEAD** commit (i.e. currently checked-out files).

```
git lfs ls-files
```

Example:

```
git lfs ls-files
b04f62c7a1 * large_file_1.txt
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
```

- List files associated with any reference (commit).

```
git lfs ls-files <ref>
```

Example:

```
git lfs ls-files HEAD~1
b04f62c7a1 * large_file_1.txt
fc51c1860c - sequences_A12.fasta
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
```

*** = file is present in working tree**

- = file is absent in working tree

```
git lfs ls-files origin/dev
b04f62c7a1 * large_file_1.txt
e82048e6d3 - sequence_C34.fasta
e6aa57987e * subdir/logo_image.img
```

- List all LFS-tracked files in the entire repo history.

```
git lfs ls-files --all
```

Example:

```
git lfs ls-files --all
b04f62c7a1 * large_file_1.txt
efdc76ef2a * sequences_B23.fasta
e6aa57987e * subdir/logo_image.img
e82048e6d3 - sequence_C34.fasta
fc51c1860c - sequences_A12.fasta
c1d5ab0faf - sequences_A12.fasta
```

Clearing the local Git LFS cache

- Deleting files from the Git LFS local cache [`.git/lfs/objects`] can be done using:

```
git lfs prune
```

Files that are deleted by the `prune` command are those that:

- Are not currently checked-out.
- Are not part of the latest commit of a “recent” branch or tag (“recent” defaults to 10 days and can be customized via `lfs.fetchrecentcommitsdays` and `lfs.pruneoffsetdays`).
- Are not part of a commit that was never pushed to the remote (since in this case there is not yet a copy of the file in the remote object store, and hence deleting it would amount to permanently losing the file).

- `lfs prune` command options:

```
git lfs prune --dry-run
```

- Lists the number of files that would be deleted, without actually deleting them.

```
$ git lfs prune --dry-run
prune: 6 local object(s), 4 retained, done.
prune: 2 file(s) would be pruned (12 MB), done.
```

```
git lfs prune --verify-remote
```

- A safety options that explicitly verifies that files are present on the remote LFS object store before deleting them.

Pulling LFS content from a remote

- Nothing special to do!
- Just use the regular Git commands and Git LFS will download content as needed.

```
git clone ...
```

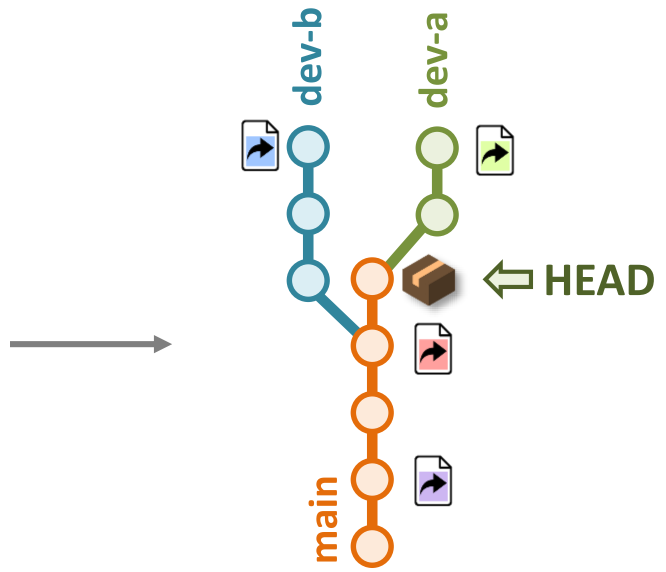
```
git fetch ...
```

```
git pull ...
```

```
git switch ...
```

- By default, only the LFS-tracked files needed for the currently checked-out branch are downloaded.

Example: if we `git clone` a new repository, only the LFS-tracked files needed for the latest commit of the *main* branch are downloaded.



Pulling additional LFS content from a remote (files from older commits or files from other branches)

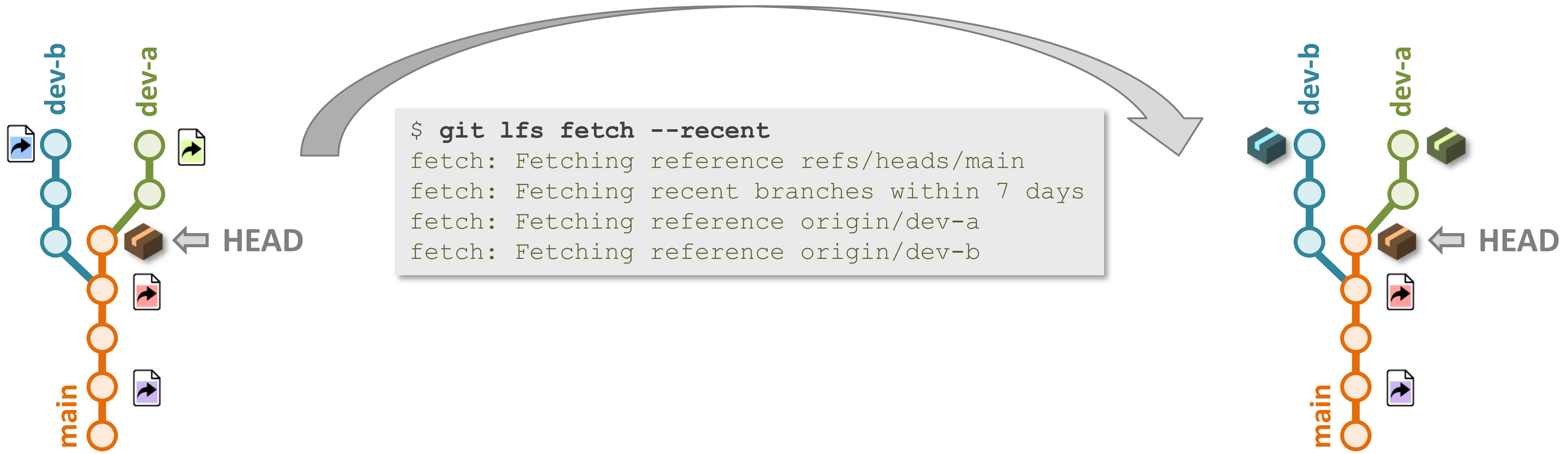
Sometimes, it can be useful to download LFS-tracked files to the local LFS cache
(e.g. when anticipating off-line time).

```
git lfs fetch --recent
```

- Downloads the LFS-tracked files of *the last* commit of all branches or tags that are considered "recent".
  - By default, "recent" is defined as no more than 7 days old.
  - The definition of "recent" can be customized via the `git config lfs.fetchrecentcommitsdays <days>` configuration option (where `<days>` = number of days).

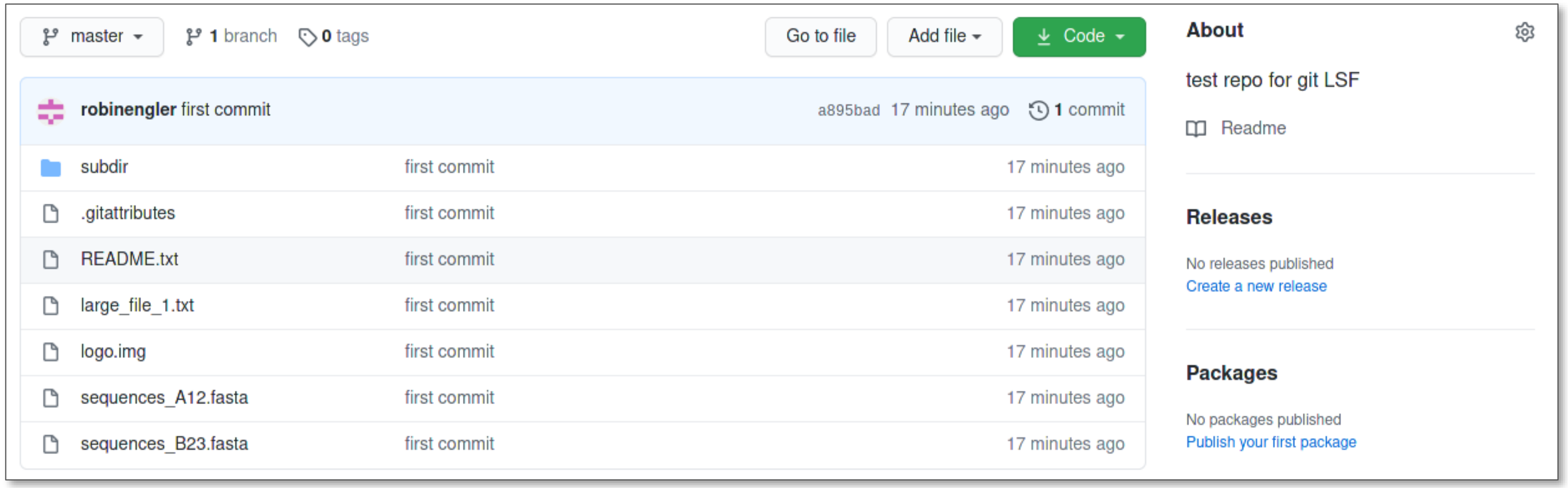
```
git lfs fetch --all
```

- Downloads all LFS-tracked files for all commits.



```
$ git lfs fetch --recent
fetch: Fetching reference refs/heads/main
fetch: Fetching recent branches within 7 days
fetch: Fetching reference origin/dev-a
fetch: Fetching reference origin/dev-b
```

- On Git hosting platforms like GitHub or GitLab, LFS-tracked files are listed just like regular files…



robinengler first commit a895bad 17 minutes ago 1 commit

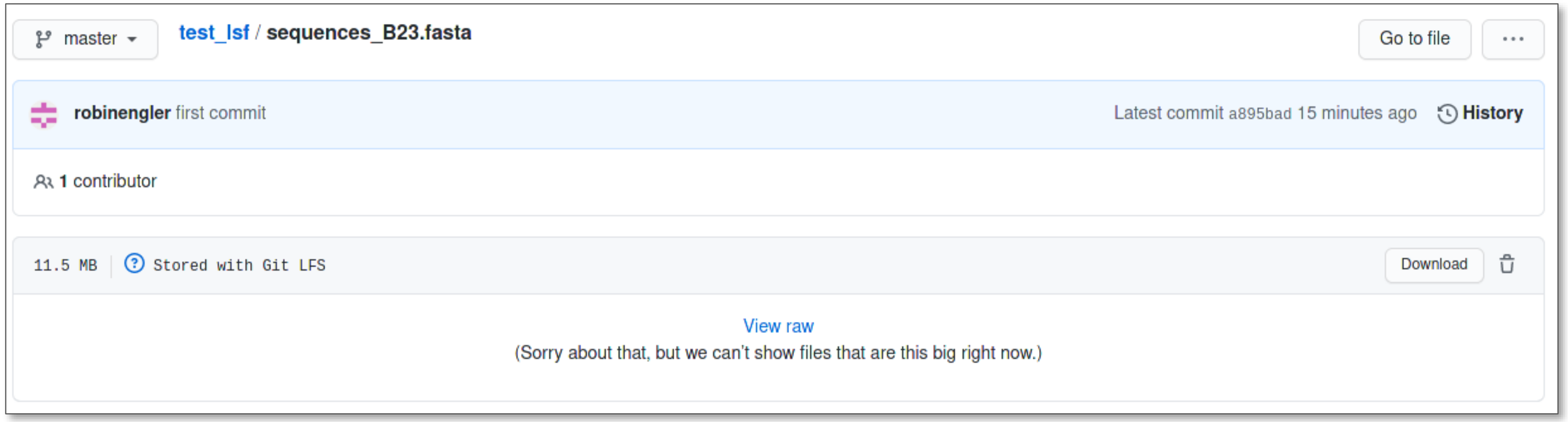
subdir	first commit	17 minutes ago
.gitattributes	first commit	17 minutes ago
README.txt	first commit	17 minutes ago
large_file_1.txt	first commit	17 minutes ago
logo.img	first commit	17 minutes ago
sequences_A12.fasta	first commit	17 minutes ago
sequences_B23.fasta	first commit	17 minutes ago

test repo for git LSF

No releases published
Create a new release

No packages published
Publish your first package

- … however, when selecting an LFS-tracked file, the content is not shown - because it's not there! Instead a "Stored with Git LFS" mention is listed:



test_lsf / sequences_B23.fasta

robinengler first commit Latest commit a895bad 15 minutes ago

1 contributor

11.5 MB | Stored with Git LFS

View raw
(Sorry about that, but we can't show files that are this big right now.)

exercise Git LFS 1

Tracking files already in Git

When a set of files are already part of a Git repository's history, there are two options to start tracking them with Git LFS:

1. Add the files (or file patterns) as tracked files with `git lfs track`. In this case however, the versions of the files associated with already made commits will remain in the Git database.

2. Remove the files' entire history from the Git repo, and have them tracked by Git LFS instead (over all of the repo's history). This can be done using `git lfs migrate` command.

Option 1

Keep files to track history in the Git repo up to the current commit.

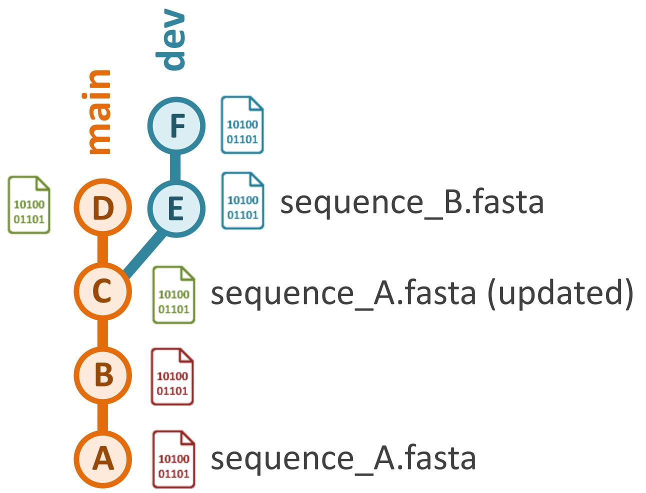
```
git lfs track "*.fasta"
git add *.gitattributes
git add *.fasta
git commit
```

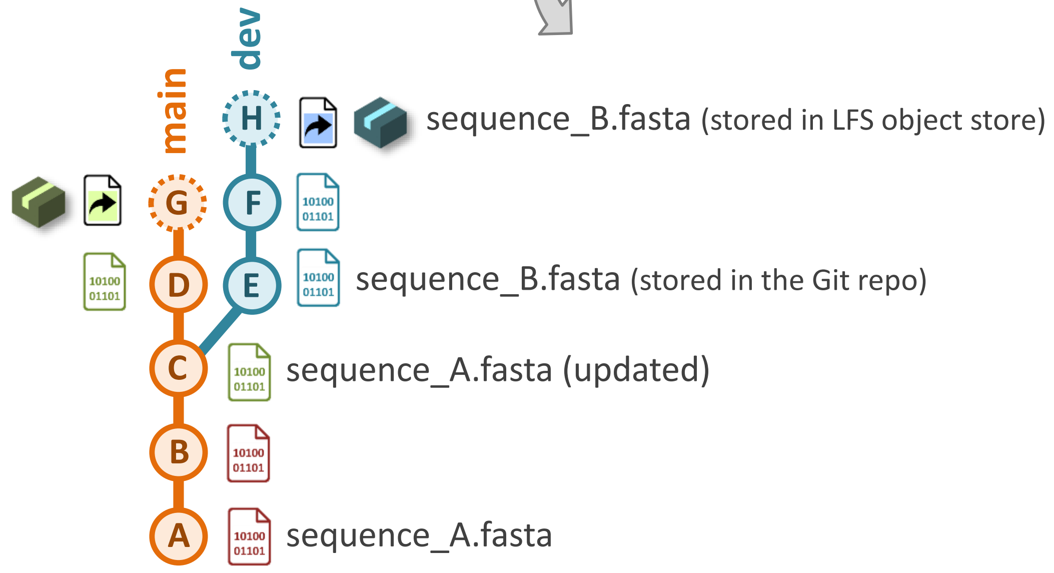
... now do the same for branch dev

Option 2

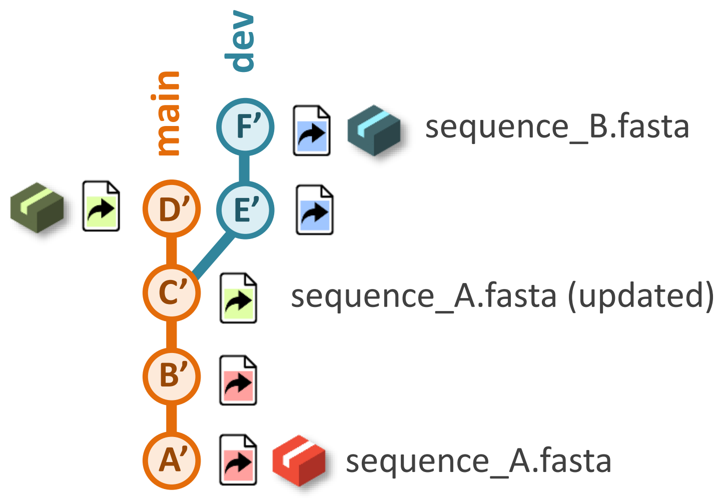
Remove files from entire Git repo history and rewrite history with files stored in LFS.

```
git lfs migrate import \
    --include="*.fasta" \
    --everything
git lfs checkout
```





main
dev
H
G
F
E
D
C
B
A
sequence_B.fasta (stored in LFS object store)
sequence_B.fasta (stored in the Git repo)
sequence_A.fasta (updated)
sequence_A.fasta



+ The repo's history remains the same.
- Git repo size possibly still too large to push to GitHub/GitLab
- Mix of files being stored in Git repo and LFS object store = not a clean solution.

+ Large files have now their entire history saved in Git LFS.
+ Size of Git database [.git/objects] truly reduced.
- Complete history rewrite: everyone has to reset their copy of the Git repo.

The `git lfs migrate` command

```
git lfs migrate import --include=<file name or pattern> --everything
```

- List of files or file patterns to "import" into Git LFS.
- Entries in `.gitattributes` will be automatically created.
- Multiple patterns/files can be specified by separating them with a comma, e.g.: `--include="*.fasta,*.img"`

This options tells git LFS to process all (local) branches of the repository.

Example:

```
git lfs migrate import --include="*.fasta,*.img" --everything

git lfs ls-files
702c4c3a56 - logo.img
6f0a4add2f - sequences_A.fasta
```

After the migrate import command completes, LFS-tracked files in the working directory are replaced with their pointer (indicated by the " – ").

```
git lfs checkout
git lfs ls-files
702c4c3a56 * logo.img
6f0a4add2f * sequences_A.fasta
```

The content of the files can be restored with `git lfs checkout`.

The `git lfs migrate` command

A couple of warnings…

History overwrite warning !

The `git lfs migrate import` command **rewrites the entire history** of your repository!

- Updating a remote repo with the changes requires a `git push --force`.
- Coordinate this operation with other people working on the repo.

Data loss warning !

- Never run `git lfs migrate import` with a non-clean working directory. All your uncommitted changes will be lost (true story)!
- To be on the safe side, it's best to **make a full copy/backup of your Git repository before running the migrate command**. In this way, should anything go wrong, you can restore your repository from your copy.

Behind the scenes...

- Git LFS stores the tracked files in the LFS cache [`.git/lfs/objects`] rather than in the Git repo [`.git/objects`].
- A lightweight “pointer” file is saved in the git repository.

Example of “pointer” blob objects stored in the Git repo [`.git/objects`]

```
.git/objects/d4/c3cf36a1c6865ba5e4d6e82e937dc835006231
```
126 bytes

```
.git/objects/a3/3c8a78275c0763d964b3a2b0facdf5909b58c3
```
125 bytes

```
git cat-file -p d4c3cf36
```

```
version https://git-lfs.github.com/spec/v1
oid sha256:e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4
size 21852324
```

```
git cat-file -p a33c8a78
```

```
version https://git-lfs.github.com/spec/v1
oid sha256:c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
size 6010948
```

The actual files are stored in the Git LFS cache [`.git/lfs/objects`]

```
.git/lfs/objects/e6/aa/e6aa57987e7b8340dbf0ed1f4e5f90cf58a1a98de2d7a860aeed178ea4e734b4
```
21.8 MB

```
.git/lfs/objects/c1/d5/c1d5ab0faf552cdb3a365347093abc42a4e65718348e17eaad1584d650ae7aa6
```
6 MB

exercise Git LFS 2

GitHub Actions
GitLab CI/CD

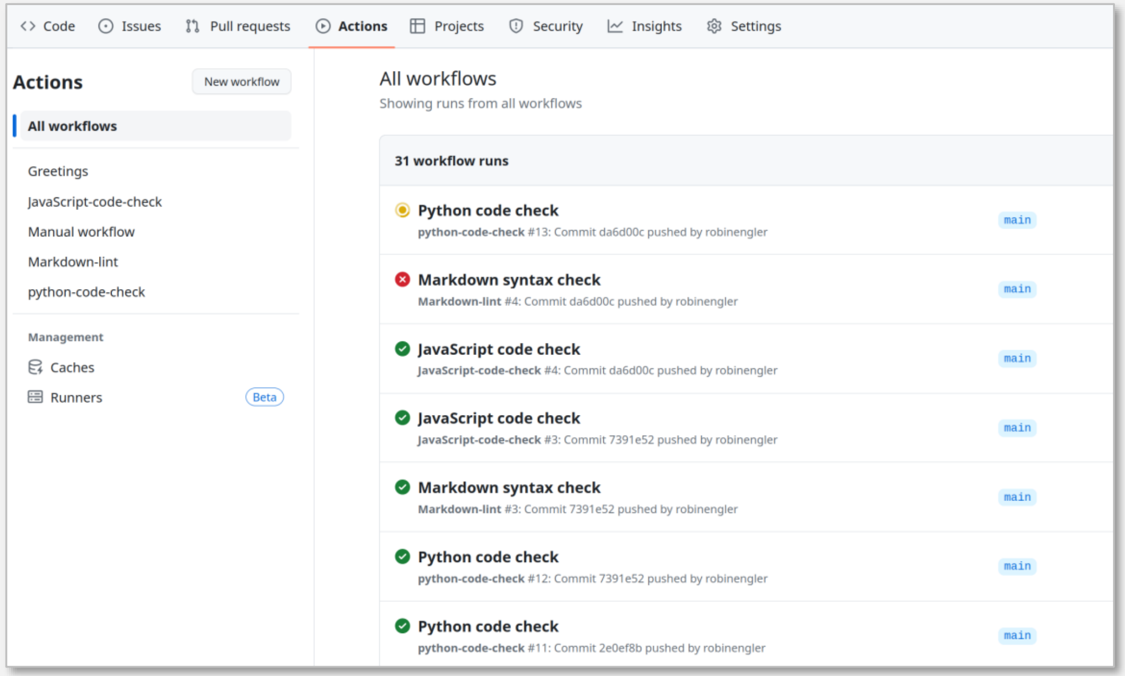
Automate your testing and delivery

Continuous integration, continuous delivery/deployment (CICD)

- The objective of CICD is to automate the testing/monitoring (integration) and delivery/deployment of code.
- Examples:
  - Run format and syntax checkers each time a new commit is pushed to the repo on GitHub/GitLab.
  - Run tests (e.g. unit tests, integration tests) each time a new commit is pushed to the repo on GitHub/GitLab.
  - Create a new release on GitHub/GitLab or a new Docker container each time a new version tag is pushed to the repo.

- **GitHub Actions** and **GitLab CI/CD** are both providing a (more or less) equivalent service: perform automated tasks/actions when a given condition is triggered, e.g. when a new commits is pushed to the repo on GitHub/GitLab.

On **GitHub**, the CICD pipelines are called **Action** "**Workflows**".



Actions

All workflows

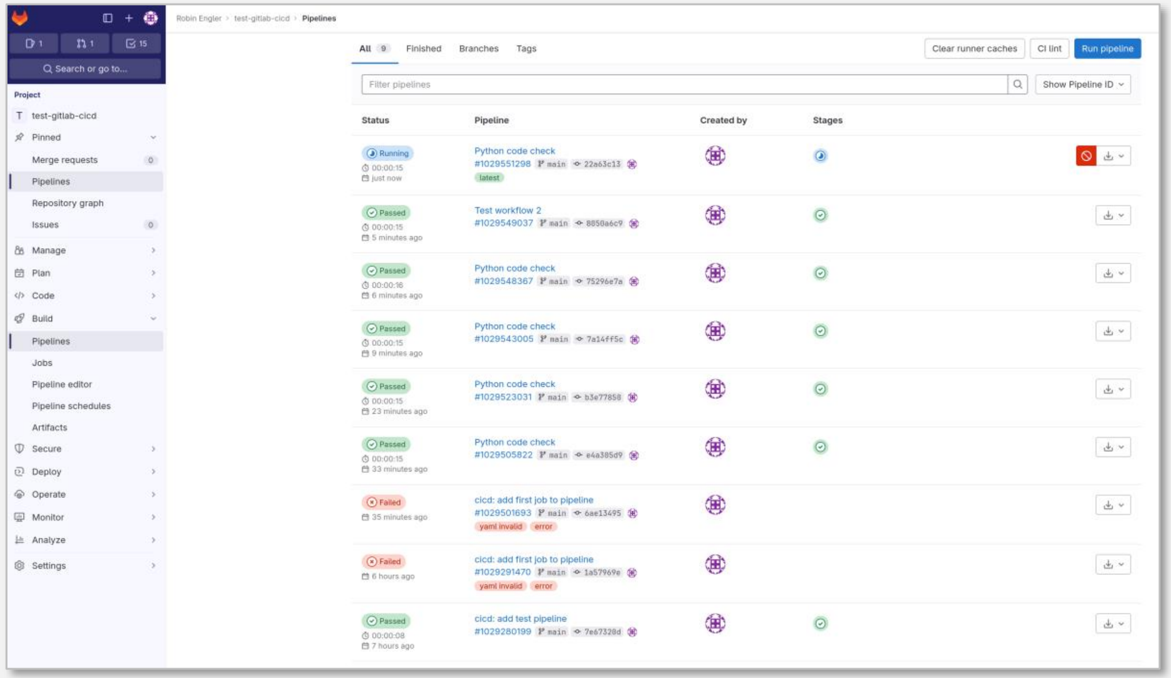
Greetings
JavaScript-code-check
Manual workflow
Markdown-lint
python-code-check

All workflows
Showing runs from all workflows

31 workflow runs

Python code check python-code-check #13: Commit da6d00c pushed by robinengler	main
Markdown syntax check Markdown-lint #4: Commit da6d00c pushed by robinengler	main
JavaScript code check JavaScript-code-check #4: Commit da6d00c pushed by robinengler	main
JavaScript code check JavaScript-code-check #3: Commit 7391e52 pushed by robinengler	main
Markdown syntax check Markdown-lint #3: Commit 7391e52 pushed by robinengler	main
Python code check python-code-check #12: Commit 7391e52 pushed by robinengler	main
Python code check python-code-check #11: Commit 2e0ef8b pushed by robinengler	main

On **GitLab**, the CICD pipelines are called "**Pipelines**".



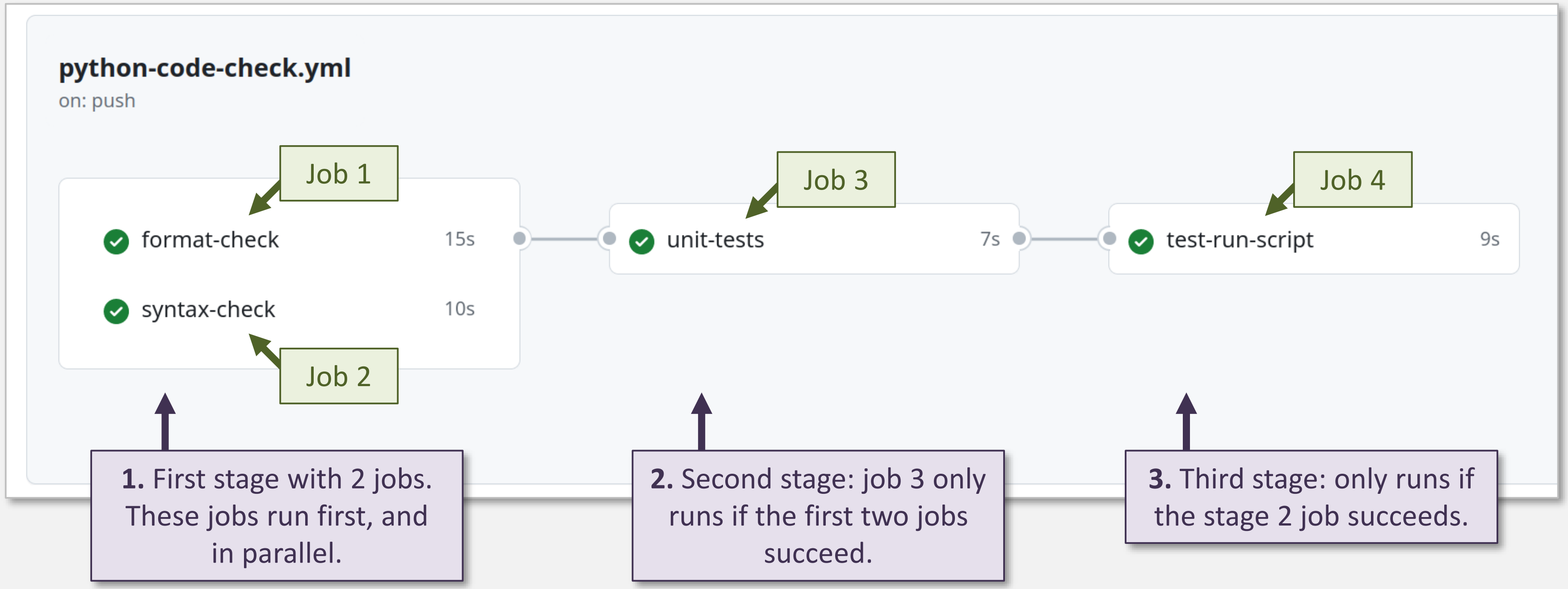
Robin Engler > test-gitlab-cicd > Pipelines

Status	Pipeline	Created by	Stages
Running 00:00:15 just now	Python code check #1029551298 main 22a63c13 latest		
Passed 00:00:15 5 minutes ago	Test workflow 2 #1029549037 main 8850a6c9		✓
Passed 00:00:16 6 minutes ago	Python code check #1029548367 main 75296e7a		✓
Passed 00:00:15 9 minutes ago	Python code check #1029543005 main 7a14ff5c		✓
Passed 00:00:15 23 minutes ago	Python code check #1029523031 main b3e77858		✓
Passed 00:00:15 33 minutes ago	Python code check #1029505822 main e4a385d9		✓
Failed 35 minutes ago	cicd: add first job to pipeline #1029501693 main 6ae13495 yaml invalid error		
Failed 6 hours ago	cicd: add first job to pipeline #1029291470 main 1a57969e yaml invalid error		
Passed 00:00:08 7 hours ago	cicd: add test pipeline #1029280199 main 7e67320d		✓

Basic principles

- The tasks to run are called **jobs**.
- Jobs are grouped together in **pipelines** (in GitLab) or **workflows** (in GitHub) *.
- In a pipeline, dependencies between jobs can be specified, to run them in a specific order.

Example of a **GitHub Workflow**, with 4 jobs distributed in 3 stages.



python-code-check.yml
on: push

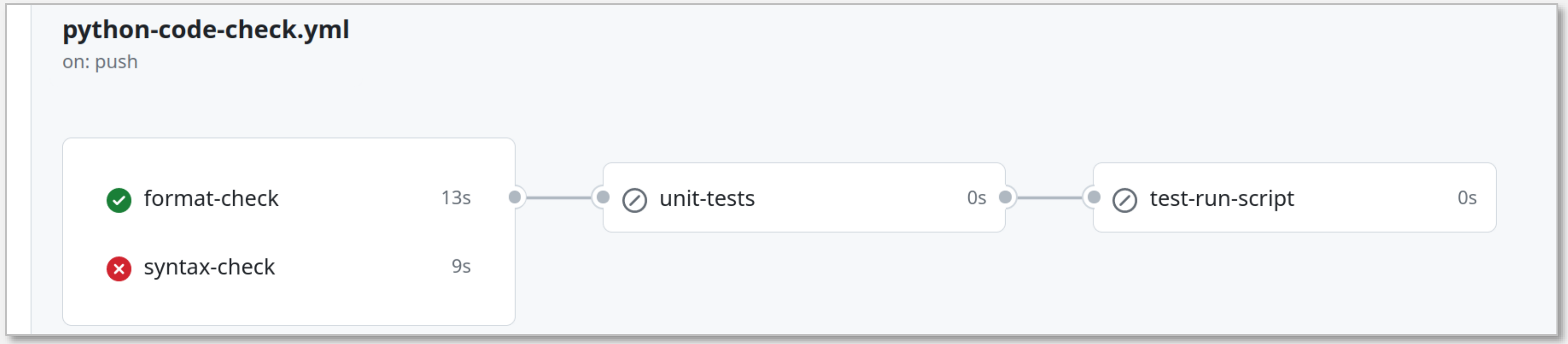
Job	Name	Duration
Job 1	format-check	15s
Job 2	syntax-check	10s
Job 3	unit-tests	7s
Job 4	test-run-script	9s

1. First stage with 2 jobs. These jobs run first, and in parallel.

2. Second stage: job 3 only runs if the first two jobs succeed.

3. Third stage: only runs if the stage 2 job succeeds.

Example of a **workflow that failed**: the jobs 3 (unit-tests) and 4 (test-run-script) did not run because a job they depend on (syntax check) has failed.



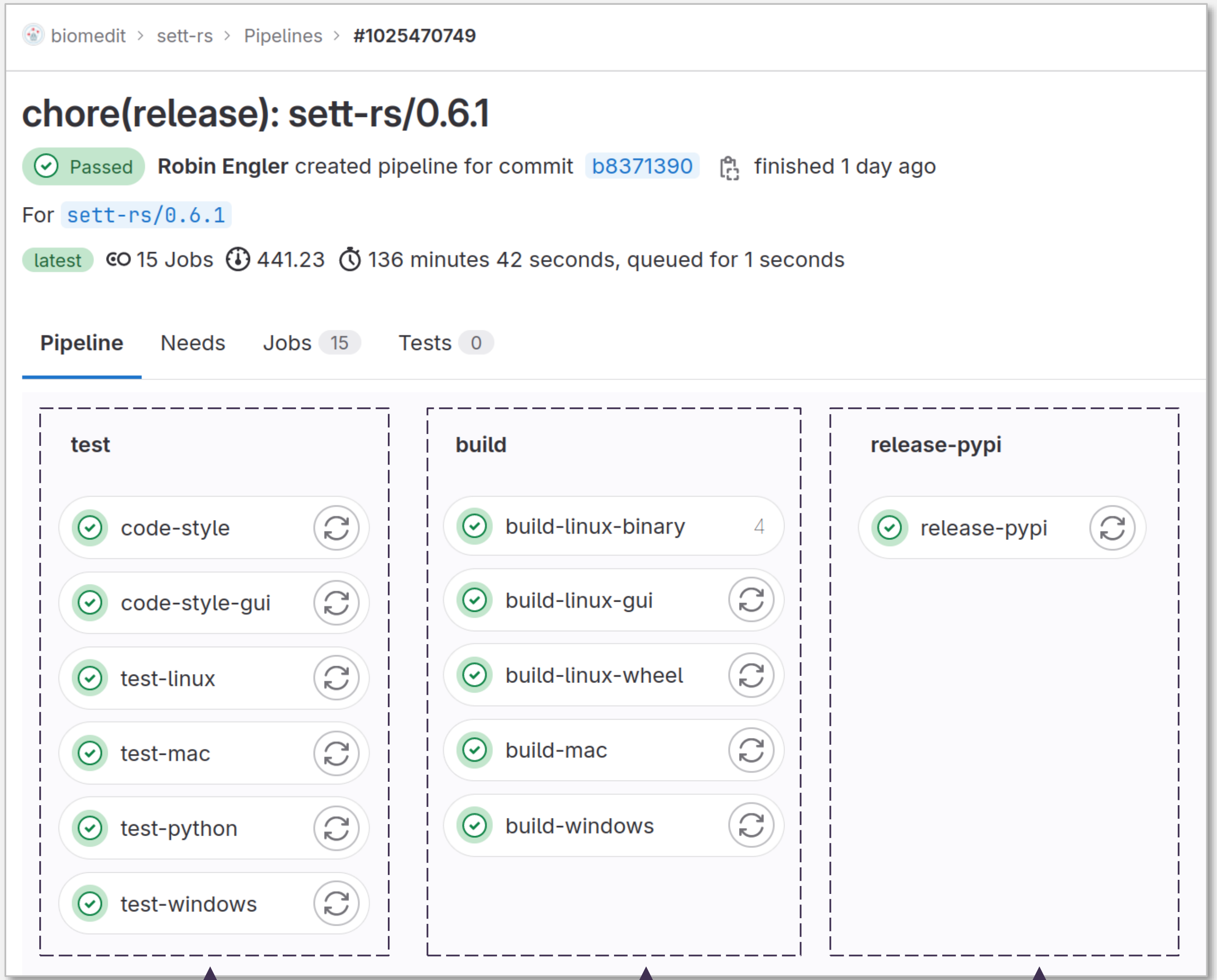
python-code-check.yml
on: push

✓	format-check	13s
✗	syntax-check	9s
⊘	unit-tests	0s
⊘	test-run-script	0s

Basic principles: GitLab CI/CD

- In a **GitLab pipeline**, jobs are grouped in **stages**.
- All jobs in a stage run in parallel.
- The next stage only runs if all jobs in the previous stage succeed (unless more fine-grained dependency relationships are defined).

Example of a **GitLab pipeline**, with 12 jobs distributed in 3 stages.



biomedit › sett-rs › Pipelines › #1025470749

chore(release): sett-rs/0.6.1

Passed **Robin Engler** created pipeline for commit b8371390 finished 1 day ago

For `sett-rs/0.6.1`

latest 15 Jobs 441.23 136 minutes 42 seconds, queued for 1 seconds

1. First stage

2. Second stage: only runs if all jobs from stage 1 succeed.

3. Third stage: only runs if all jobs from stage 2 succeed.

How to setup a pipeline: general principle

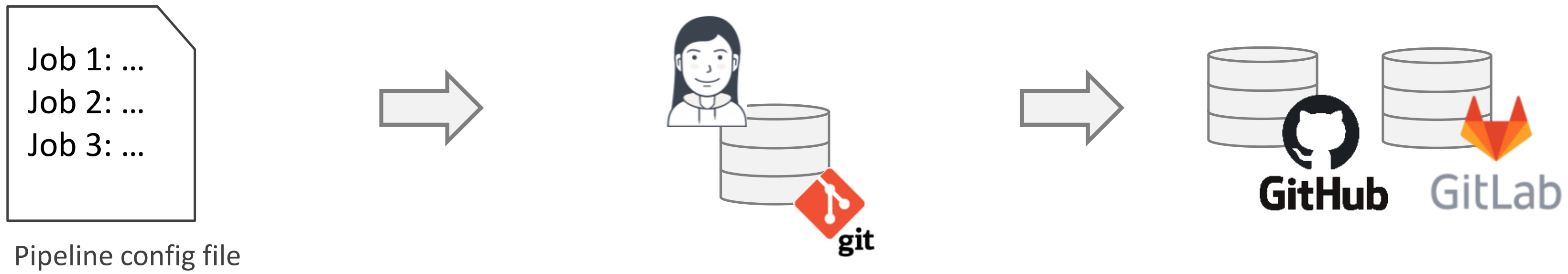
Setting-up a pipeline follows a similar procedure on both GitHub and GitLab:

- **Write an instruction file** that defines the jobs to run to your Git repository.
- **Commit the instruction file** to your repo, and **push the commit** to the GitHub/GitLab remote.
- GitHub/GitLab automatically detect your CICD instruction files and, from now on, will run the jobs as specified.

1. Write a configuration file with instructions on how and when to run each job.

2. Add the configuration file to your local repo and **commit it.**

3. Push the commit to the remote on GitHub/GitLab.



Pipeline config file

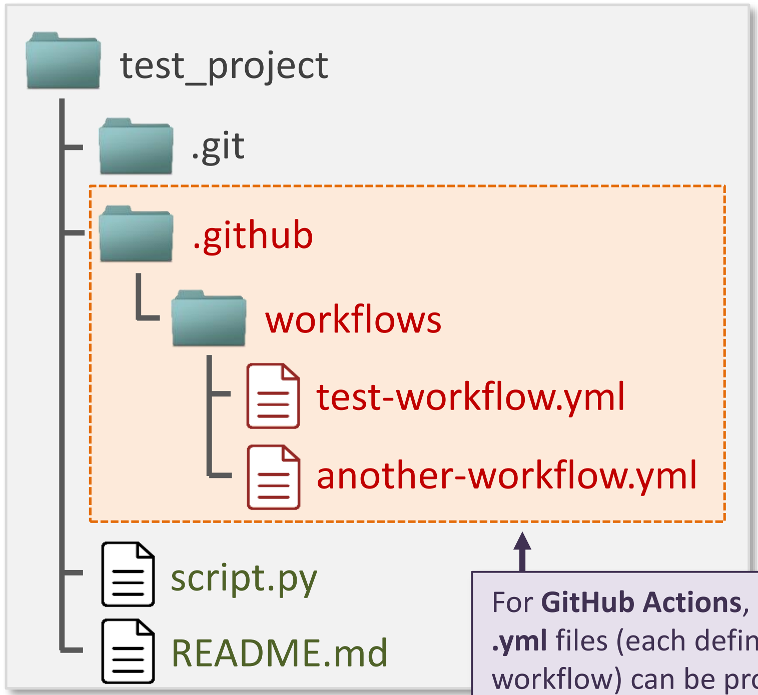
Done ! Your jobs will now run automatically whenever their trigger condition is met.

Pipelines configuration files: location and naming conventions

For GitHub/GitLab to detect your CICD configuration files, they must strictly follow these conventions.

Naming conventions for GitHub Actions

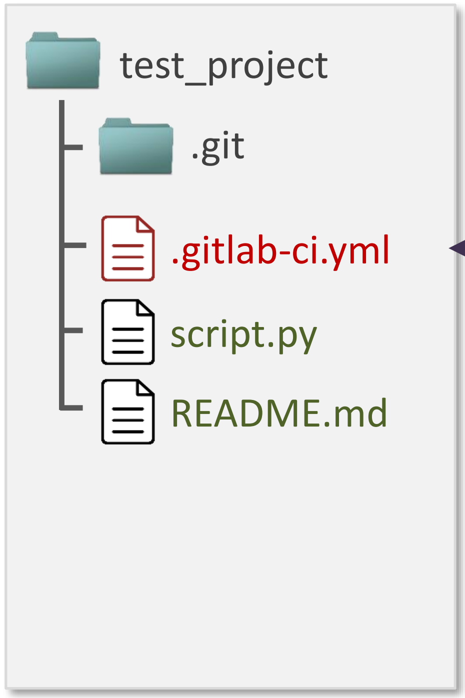
- Workflow (pipelines) configuration files must be stored in **.github/workflows**.
- One or more **.yml** workflows can be defined.



For **GitHub Actions**, multiple **.yml** files (each defining a workflow) can be provided. They must be in a directory named exactly **.github/workflows**.

Naming conventions for GitLab CI/CD

- All jobs are defined in a **single config file**.
- The config file must be named exactly **.gitlab-ci.yml**
- The config file must be placed at the **root of the repo**.



For **GitLab CI/CD** to work, a single file named **.gitlab-ci.yml** must be provided.

Pipelines configuration files: GitHub Actions syntax

- CICD configuration files for GitHub must be written in **YAML** (Yet Another Markup Language - https://yaml.org).
- **YAML** is a "human-readable data serialization language", which roughly means that it's a way to write some key-value configurations in an easy way using plain text.

GitHub workflow example:

A basic (and useless) workflow with a single job that prints "Hello World".

test-workflow.yml

workflow name →
display name →
Trigger condition →

```yaml
name: test-workflow
run-name: Test workflow
on: push

jobs:
  # Comments can be added to the file like this.
  print-hello-world:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout git repo
        uses: actions/checkout@v4
      - name: Say hello...
        run: echo "Hello World"
```

The **push** condition means that the job will run each time a commit is pushed to the GitHub remote.

Jobs to run in workflow →

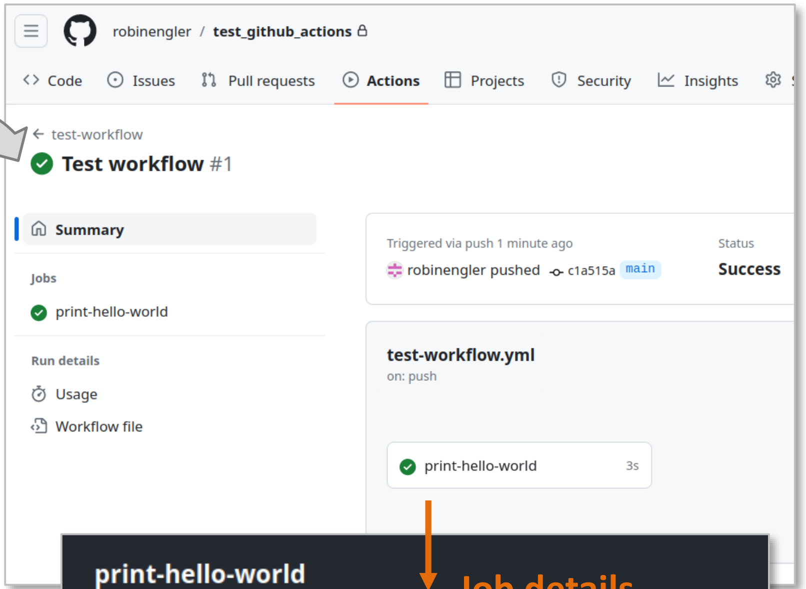
job name →

Important: nested levels must be indented properly.

Instructions for the first (and only) job of the workflow.

This only works on GitHub

Commit and push to GitHub... the workflow runs.



test-workflow

Test workflow #1

print-hello-world

Triggered via push 1 minute ago

robinengler pushed c1a515a main

Status

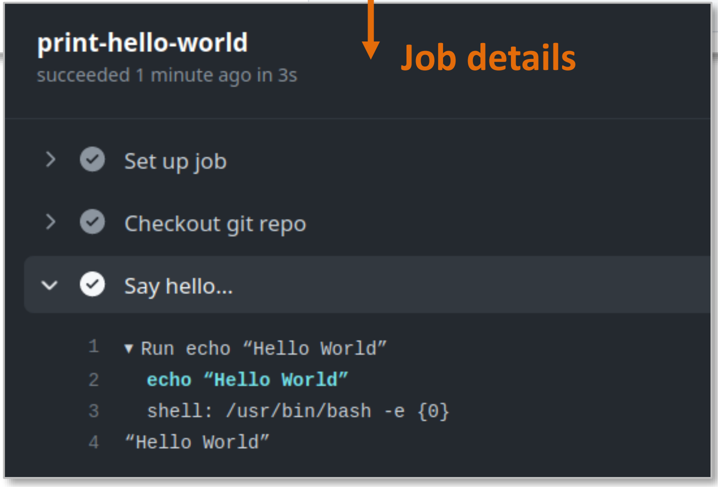
Success

test-workflow.yml

on: push

print-hello-world 3s

Job details



print-hello-world

succeeded 1 minute ago in 3s

Set up job

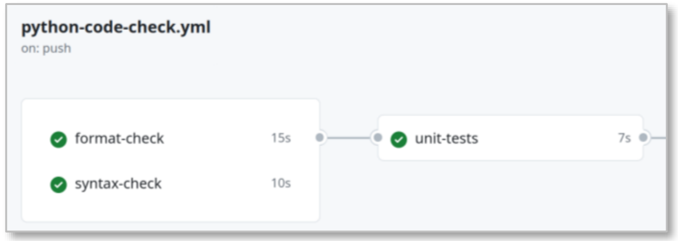
Checkout git repo

Say hello...

```
▾ Run echo "Hello World"
  echo "Hello World"
  shell: /usr/bin/bash -e {0}
"Hello World"
```

GitHub workflow example

A python code quality checking workflow with 3 jobs in 2 stages.


python-code-check.yml
on: push

- ✓ format-check 15s
- ✓ syntax-check 10s
- ✓ unit-tests 7s

python-code-check.yml

```yaml
name: python-code-check
run-name: Python code check
on: [push, pull_request]

jobs:
  # Run the python black formatter.
  format-check:
    runs-on: ubuntu-latest
    steps:
    - name: Checkout git repo
      uses: actions/checkout@v4
    - name: Install Python
      uses: actions/setup-python@v4
    - name: Install black
      run: |
        python -m pip install --upgrade pip
        pip install black
    - name: Run black
      run: black --check .

  # Run pylint, a python code linter (checks for syntax errors).
  syntax-check:
    # Content not shown to save space on the slide...

  # Run unit-tests for our code.
  unit-tests:
    needs: [format-check, syntax-check]
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - uses: actions/setup-python@v4
      with:
        python-version: '3.11'
    - name: Install pytest
      run: |
        python -m pip install --upgrade pip
        pip install pytest
    - name: Run pytest
      run: pytest test_*.py
```

Multiple trigger conditions can be specified.

Each job runs on a separate VM (virtual machine). Here we indicate that the VM should be a Linux Ubuntu machine. Other operating systems can be chosen.

Step 1 of job: here we clone the content of our repo to the VM.

Step 2 of job: here we install python.

Step 3 of job: we install the tool that checks code format.

Step 4 of job: we run the code formatter on the content of our repo.

Our job has 4 steps. Each step starts with a line prefixed with "-".

The `needs:` keyword is used to indicate dependencies between jobs. This job will only run if both the "format-check" and "syntax-check" jobs complete successfully.

- Giving a name to a step is optional. Here we skip naming and directly tell what the step should do.
- The **actions/preset@version** indicates to run a preset action available from GitHub, e.g. checkout@v4 => check-out the repo, setup-python@v4 => install python in the VM.

To write commands on multiple lines, start the `run:` command with **|** .

This only works on GitHub

Pipelines configuration files: GitLab CI/CD syntax

This only works on GitLab

- GitLab CI/CD configuration files are also written in YAML, but the file structure is different from GitHub Actions.
- Not possible to use the same files for GitLab and GitHub.

.gitlab-ci.yml

```yaml
# GitLab CI/CD configuration file.

workflow:
  name: "Test workflow"

stages:
  - test
  - deploy

test-job:
  stage: test
  image: alpine:latest
  before_script:
    - echo "This runs first"
  script:
    - echo "Hello World!"
  rules:
    - if: $CI_COMMIT_TAG
      when: never
    - if: $CI_COMMIT_BRANCH

deploy-application:
  stage: deploy
  image: python:slim
  script:
    - echo "here we would build and push our \
            application to DockerHub"
```

Optional: the **`workflow:`** section allows to set values at the pipeline level. E.g. give a name to the pipeline.

The **`stages:`** a stage is a group of job that are run at the same time (i.e. they do not depend on each other). Stages run in the listed order, and jobs from a given stage only start running if all jobs from the previous stage have completed successfully.

Job name →

Stage that the job belongs to →

Definition of a job ("test-job")

`image:` container image (e.g. for DockerHub) to use to run the job.

`before_script:` Optional. Commands that will run before the "**`script`**" commands.

`script:` commands that the job should run.

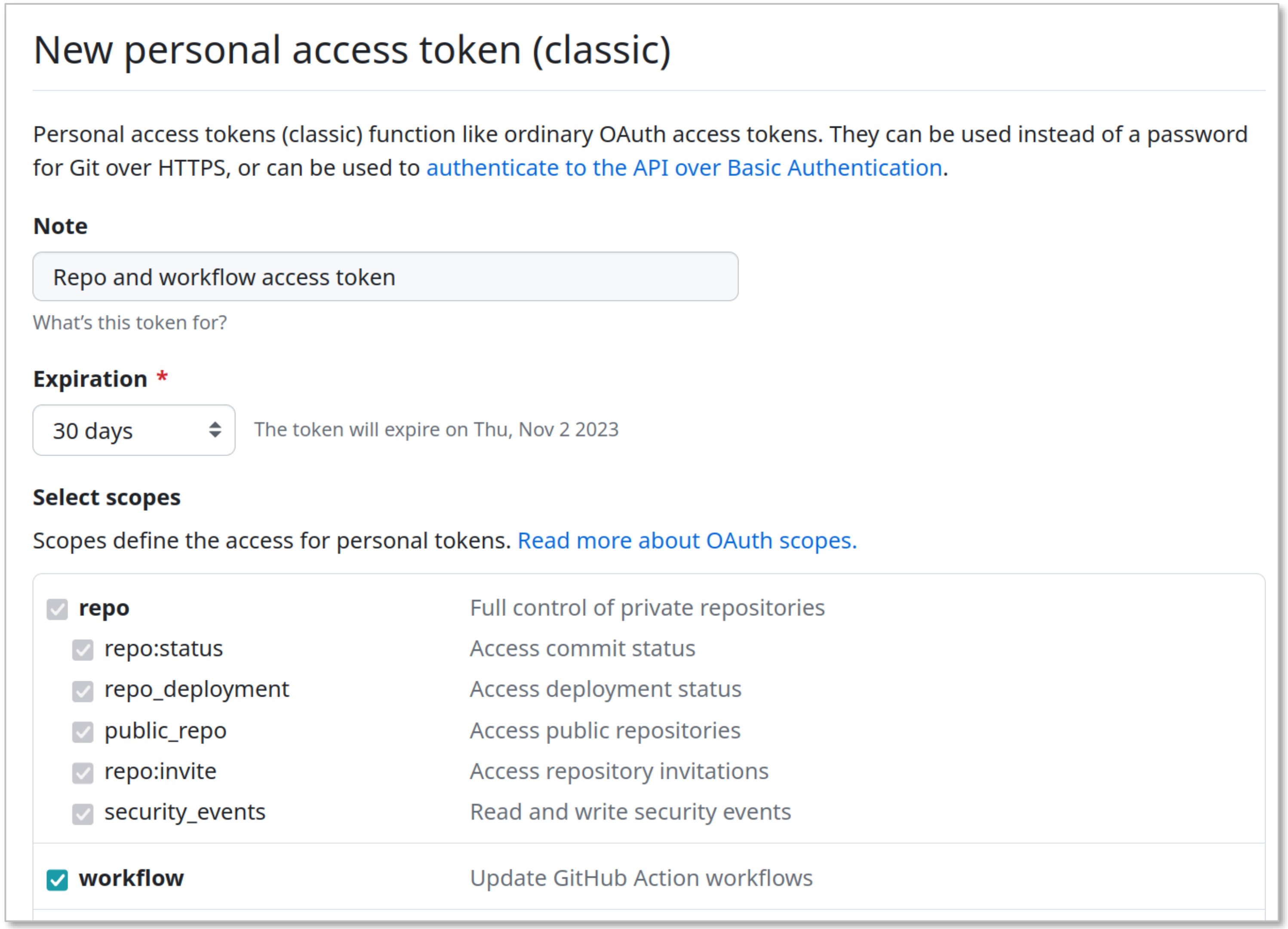
`rules:` can be used to specific the conditions under which a job should run (by default jobs run on every commit).

Job name →

Another job

GitHub Actions requires an access token with "workflow" scope:

- To push a commit that contains a workflow configuration file, the authentication token needs to have the "workflow" scope enabled.
- You can create a new token, or add this scope to an existing token.



New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to authenticate to the API over Basic Authentication.

Note

Repo and workflow access token

What's this token for?

Expiration *

30 days — The token will expire on Thu, Nov 2 2023

Select scopes

Scopes define the access for personal tokens. Read more about OAuth scopes.

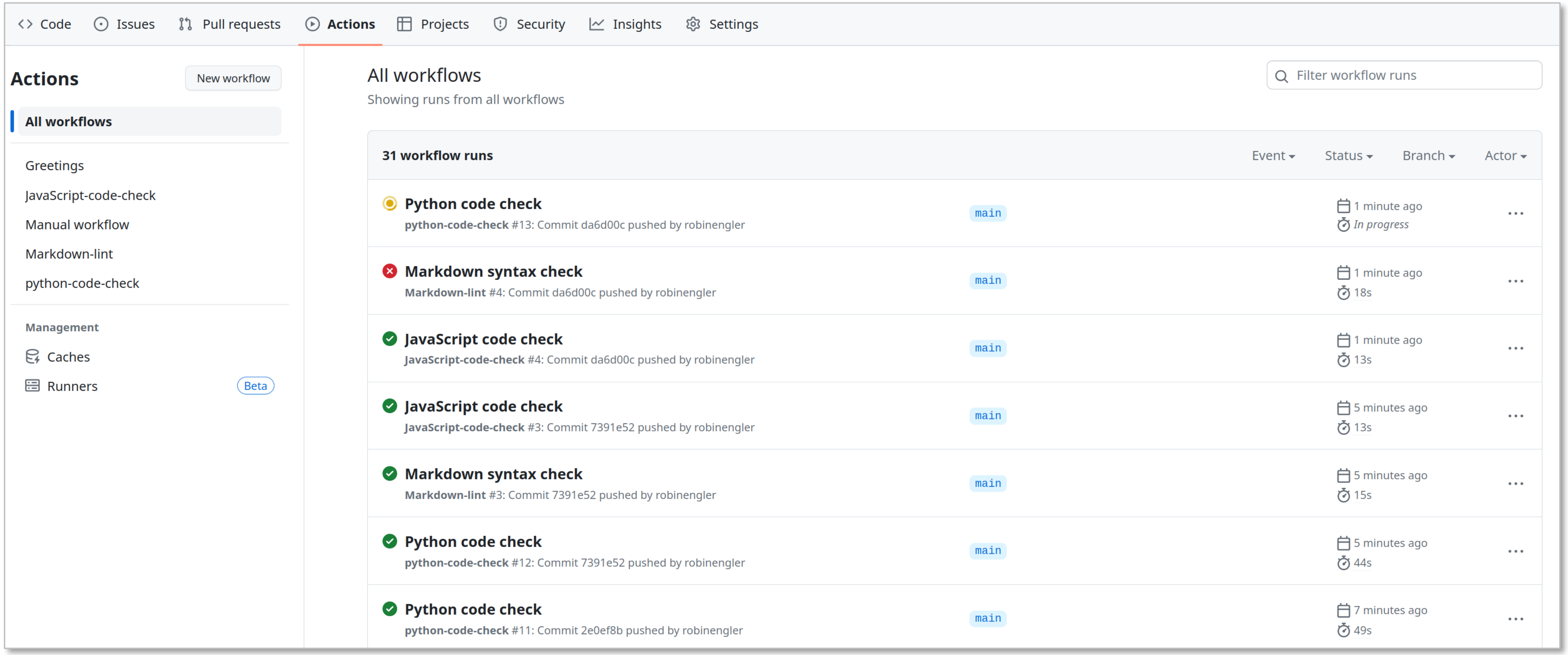
✓ **repo**	Full control of private repositories
✓ repo:status	Access commit status
✓ repo_deployment	Access deployment status
✓ public_repo	Access public repositories
✓ repo:invite	Access repository invitations
✓ security_events	Read and write security events
✓ **workflow**	Update GitHub Action workflows

This only needed for GitHub

Running the pipelines

- **All workflows/pipelines run automatically** – there is nothing to do (exception: workflow that requires manual triggering).
- GitHub/GitLab will send you a notification email if a workflow/pipeline fails.

GitHub: to view your workflows, go to the **Actions** tab.



All workflows
Showing runs from all workflows

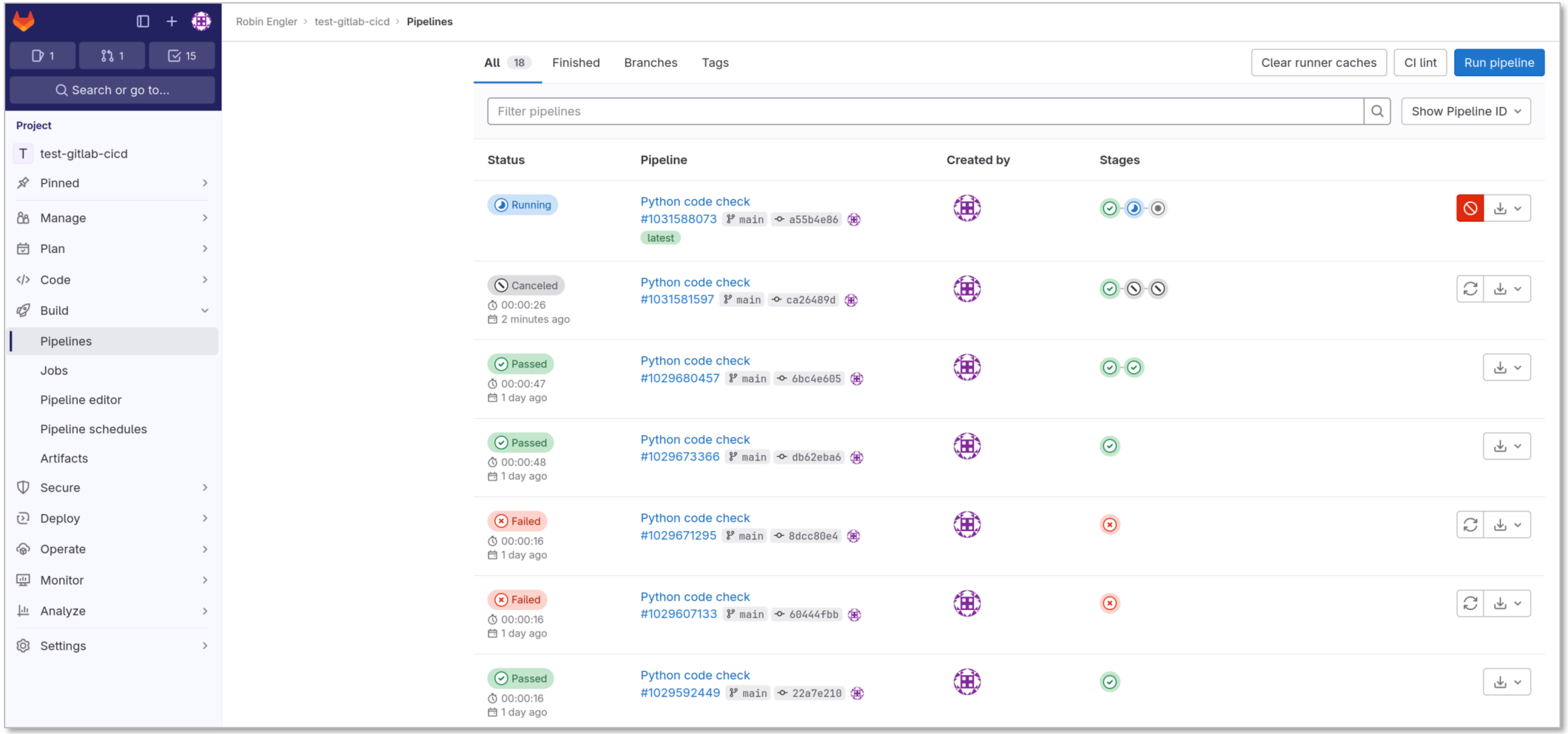
31 workflow runs

Workflow	Run	Branch	When	Duration
Python code check	python-code-check #13: Commit da6d00c pushed by robinengler	main	1 minute ago	In progress
Markdown syntax check	Markdown-lint #4: Commit da6d00c pushed by robinengler	main	1 minute ago	18s
JavaScript code check	JavaScript-code-check #4: Commit da6d00c pushed by robinengler	main	1 minute ago	13s
JavaScript code check	JavaScript-code-check #3: Commit 7391e52 pushed by robinengler	main	5 minutes ago	13s
Markdown syntax check	Markdown-lint #3: Commit 7391e52 pushed by robinengler	main	5 minutes ago	15s
Python code check	python-code-check #12: Commit 7391e52 pushed by robinengler	main	5 minutes ago	44s
Python code check	python-code-check #11: Commit 2e0ef8b pushed by robinengler	main	7 minutes ago	49s

Running the pipelines

- **All workflows/pipelines run automatically** – there is nothing to do (exception: workflow that requires manual triggering).
- GitHub/GitLab will send you a notification email if a workflow/pipeline fails.

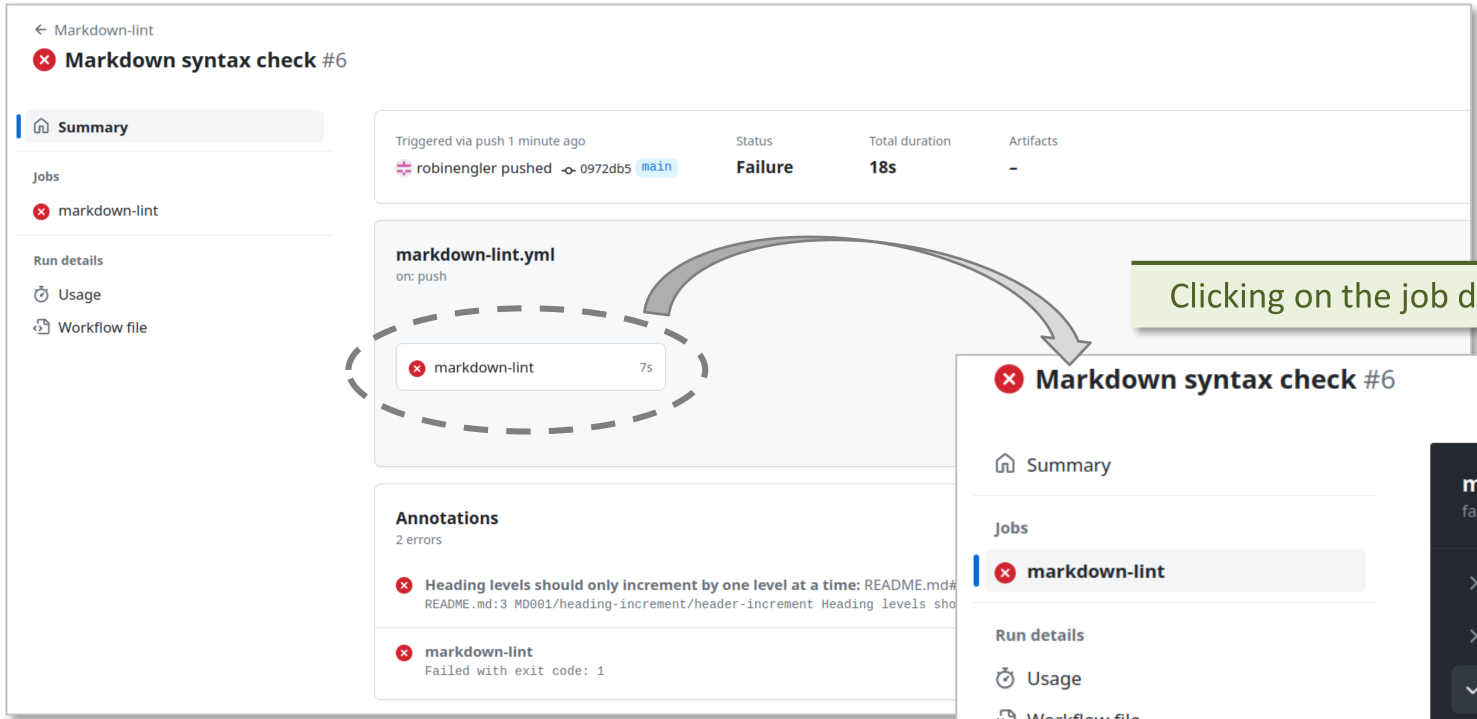
GitLab: to view your pipelines, go to the **Pipelines** tab.



Status	Pipeline	Created by	Stages
Running	Python code check #1031588073 main a55b4e86 latest		
Canceled 00:00:26 2 minutes ago	Python code check #1031581597 main ca26489d		
Passed 00:00:47 1 day ago	Python code check #1029680457 main 6bc4e605		
Passed 00:00:48 1 day ago	Python code check #1029673366 main db62eba6		
Failed 00:00:16 1 day ago	Python code check #1029671295 main 8dcc80e4		
Failed 00:00:16 1 day ago	Python code check #1029607133 main 60444fbb		
Passed 00:00:16 1 day ago	Python code check #1029592449 main 22a7e210		

Investigating failed jobs

- Clicking on a job that failed displays details on the reason for failure… so that we can fix problems.



← Markdown-lint

Markdown syntax check #6

Summary

Jobs

markdown-lint

Run details

Usage

Workflow file

Triggered via push 1 minute ago	Status	Total duration	Artifacts
robinengler pushed 0972db5 main	Failure	18s	–

markdown-lint.yml
on: push

markdown-lint 7s

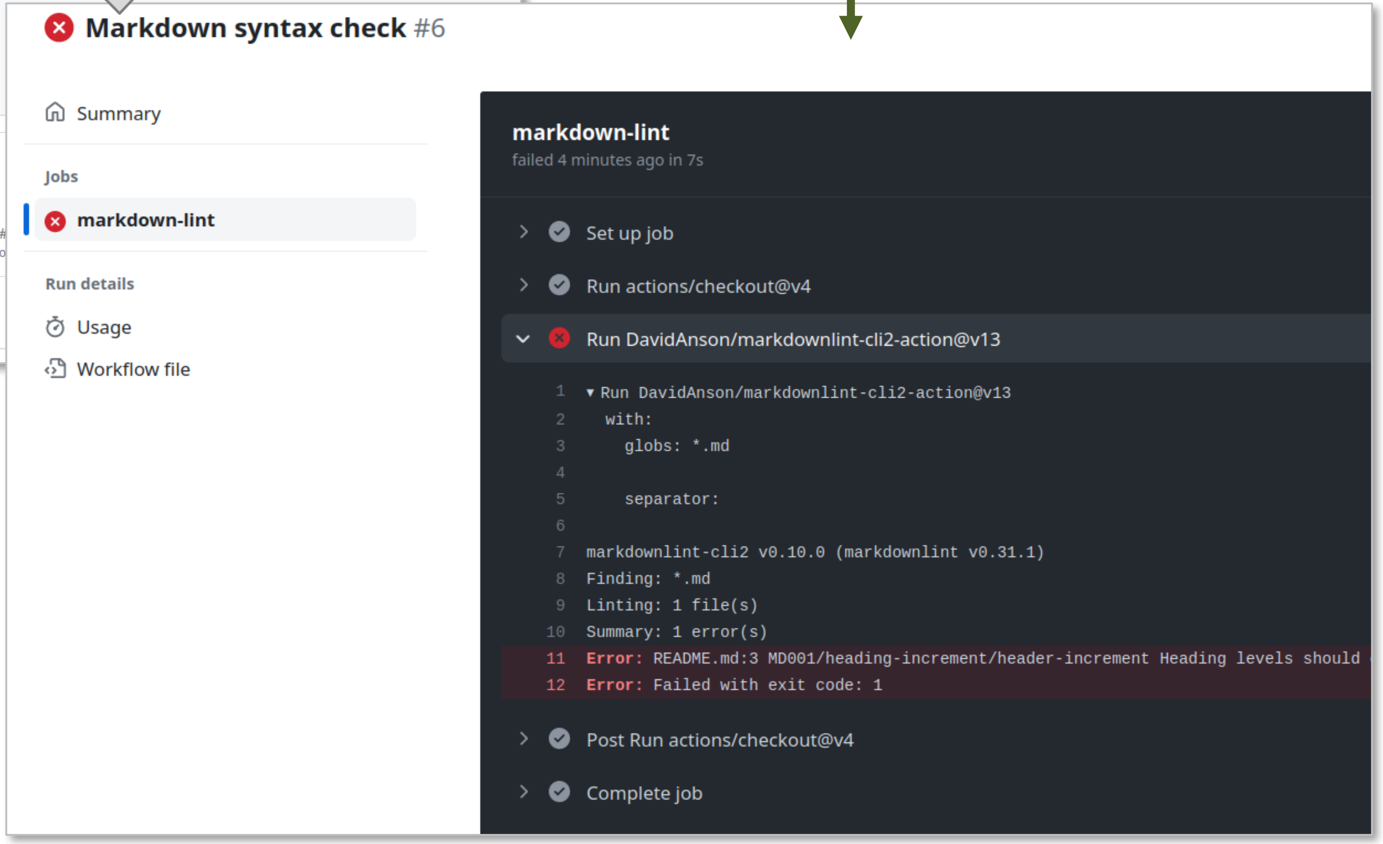
Annotations
2 errors

Heading levels should only increment by one level at a time: README.md#
README.md:3 MD001/heading-increment/header-increment Heading levels sho

markdown-lint
Failed with exit code: 1

Example: this job performs a check on markdown document syntax and found an error in the README.md file.

Clicking on the job displays additional details.



Markdown syntax check #6

Summary

Jobs

markdown-lint

Run details

Usage

Workflow file

markdown-lint
failed 4 minutes ago in 7s

Set up job

Run actions/checkout@v4

Run DavidAnson/markdownlint-cli2-action@v13

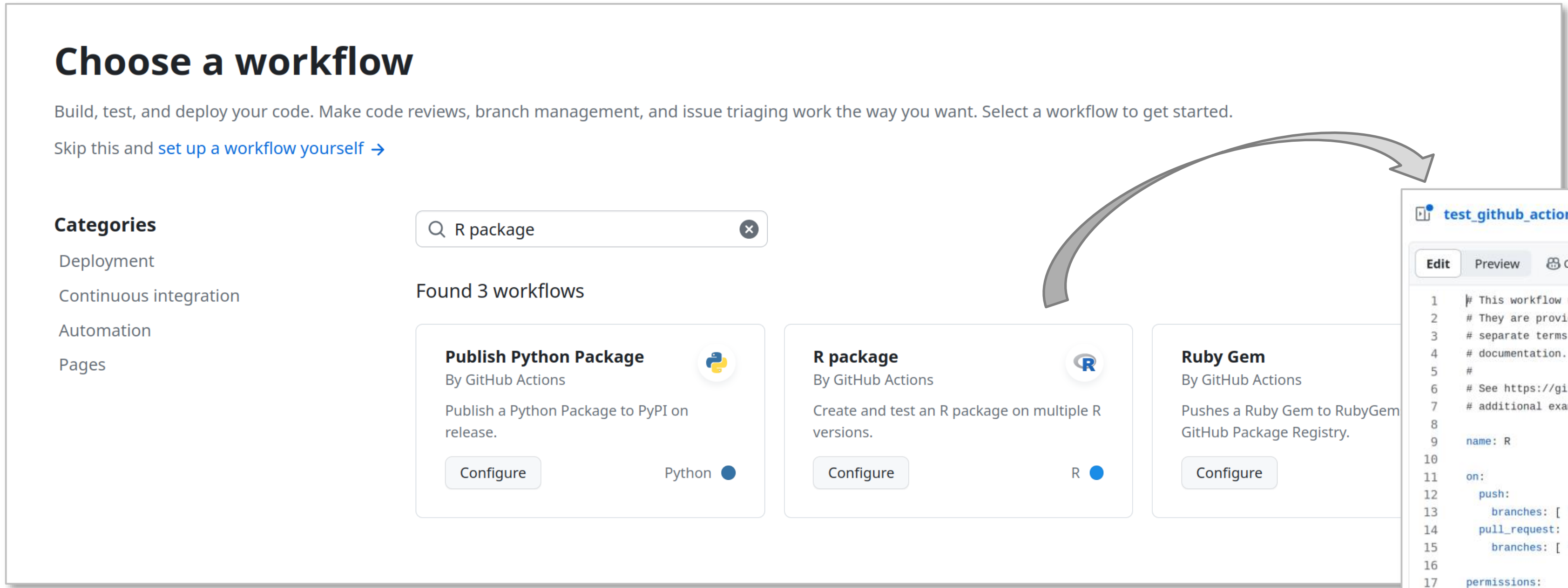
```
▾ Run DavidAnson/markdownlint-cli2-action@v13
  with:
    globs: *.md

    separator:

markdownlint-cli2 v0.10.0 (markdownlint v0.31.1)
Finding: *.md
Linting: 1 file(s)
Summary: 1 error(s)
Error: README.md:3 MD001/heading-increment/header-increment Heading levels should
Error: Failed with exit code: 1
```

Post Run actions/checkout@v4

Complete job

Finding workflow presets

- GitHub provides a number of workflow presets under the **Actions** tab > **New workflow**.



Choose a workflow

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and set up a workflow yourself →

Categories

- Deployment
- Continuous integration
- Automation
- Pages

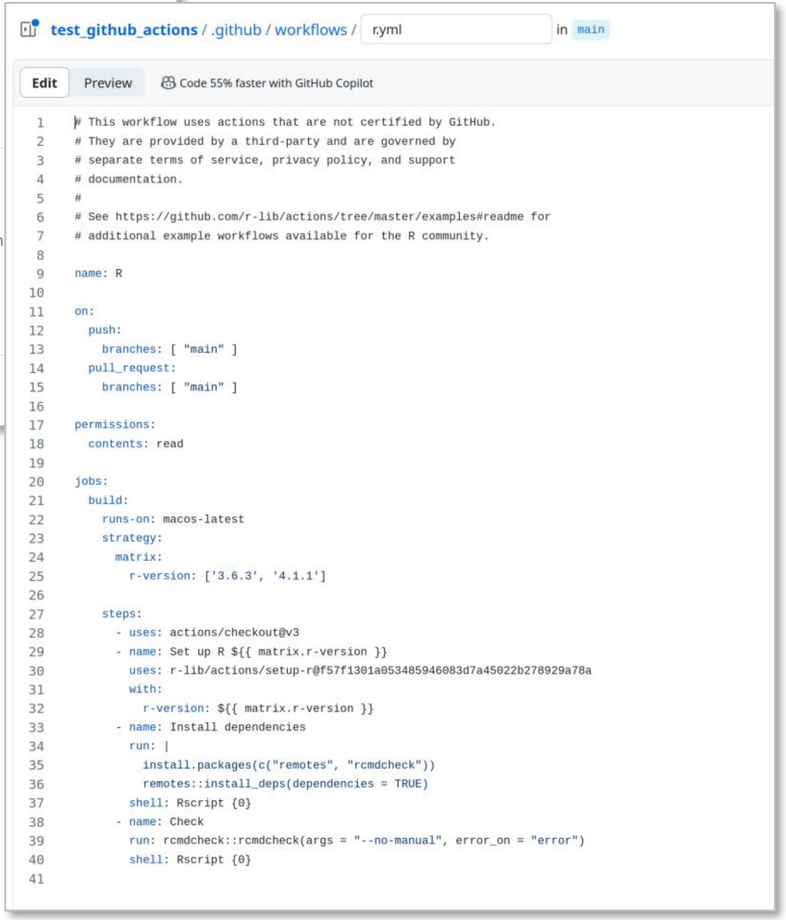
R package

Found 3 workflows

Publish Python Package
By GitHub Actions
Publish a Python Package to PyPI on release.
Configure — Python

R package
By GitHub Actions
Create and test an R package on multiple R versions.
Configure — R

Ruby Gem
By GitHub Actions
Pushes a Ruby Gem to RubyGem
GitHub Package Registry.
Configure



test_github_actions / .github / workflows / r.yml in main

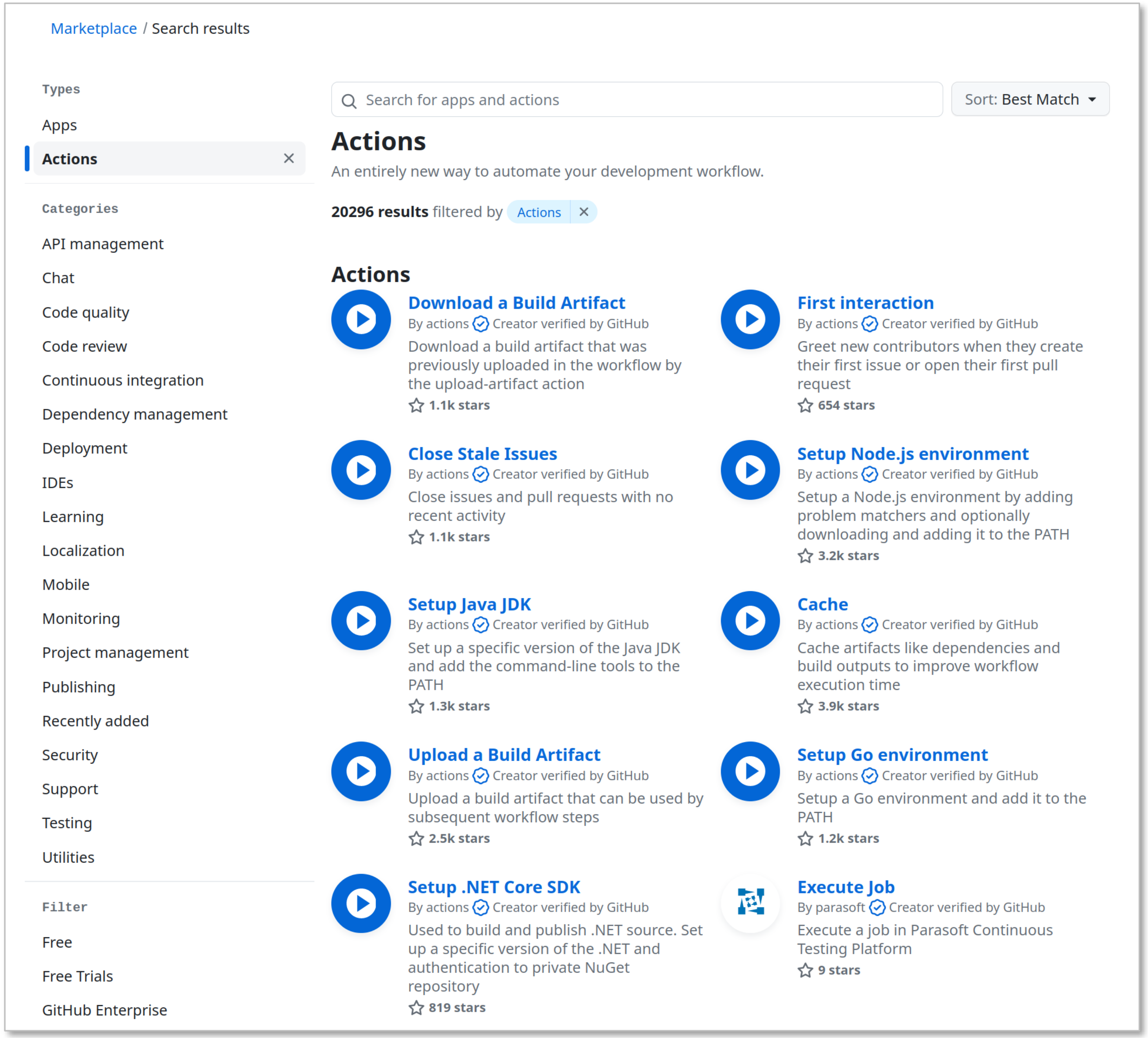
Edit | Preview | Code 55% faster with GitHub Copilot

```yaml
# This workflow uses actions that are not certified by GitHub.
# They are provided by a third-party and are governed by
# separate terms of service, privacy policy, and support
# documentation.
#
# See https://github.com/r-lib/actions/tree/master/examples#readme for
# additional example workflows available for the R community.

name: R

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]

permissions:
  contents: read

jobs:
  build:
    runs-on: macos-latest
    strategy:
      matrix:
        r-version: ['3.6.3', '4.1.1']

    steps:
      - uses: actions/checkout@v3
      - name: Set up R ${{ matrix.r-version }}
        uses: r-lib/actions/setup-r@f57f1301a053485946083d7a45022b278929a78a
        with:
          r-version: ${{ matrix.r-version }}
      - name: Install dependencies
        run: |
          install.packages(c("remotes", "rcmdcheck"))
          remotes::install_deps(dependencies = TRUE)
        shell: Rscript {0}
      - name: Check
        run: rcmdcheck::rcmdcheck(args = "--no-manual", error_on = "error")
        shell: Rscript {0}

```

The workflow code can be copied, directly added to the repo, or used for inspiration

Finding workflow presets

- More workflows can be found on the **GitHub Marketplace** https://github.com/marketplace
- But these are provided by third-parties so their **quality and trustworthiness might vary**.
- **Use with caution.**



Marketplace / Search results

Types
Apps
Actions ×

Categories
API management
Chat
Code quality
Code review
Continuous integration
Dependency management
Deployment
IDEs
Learning
Localization
Mobile
Monitoring
Project management
Publishing
Recently added
Security
Support
Testing
Utilities

Filter
Free
Free Trials
GitHub Enterprise

Search for apps and actions | Sort: Best Match

Actions

An entirely new way to automate your development workflow.

20296 results filtered by Actions ×

Actions

Download a Build Artifact
By actions Creator verified by GitHub
Download a build artifact that was previously uploaded in the workflow by the upload-artifact action
1.1k stars

First interaction
By actions Creator verified by GitHub
Greet new contributors when they create their first issue or open their first pull request
654 stars

Close Stale Issues
By actions Creator verified by GitHub
Close issues and pull requests with no recent activity
1.1k stars

Setup Node.js environment
By actions Creator verified by GitHub
Setup a Node.js environment by adding problem matchers and optionally downloading and adding it to the PATH
3.2k stars

Setup Java JDK
By actions Creator verified by GitHub
Set up a specific version of the Java JDK and add the command-line tools to the PATH
1.3k stars

Cache
By actions Creator verified by GitHub
Cache artifacts like dependencies and build outputs to improve workflow execution time
3.9k stars

Upload a Build Artifact
By actions Creator verified by GitHub
Upload a build artifact that can be used by subsequent workflow steps
2.5k stars

Setup Go environment
By actions Creator verified by GitHub
Setup a Go environment and add it to the PATH
1.2k stars

Setup .NET Core SDK
By actions Creator verified by GitHub
Used to build and publish .NET source. Set up a specific version of the .NET and authentication to private NuGet repository
819 stars

Execute Job
By parasoft Creator verified by GitHub
Execute a job in Parasoft Continuous Testing Platform
9 stars

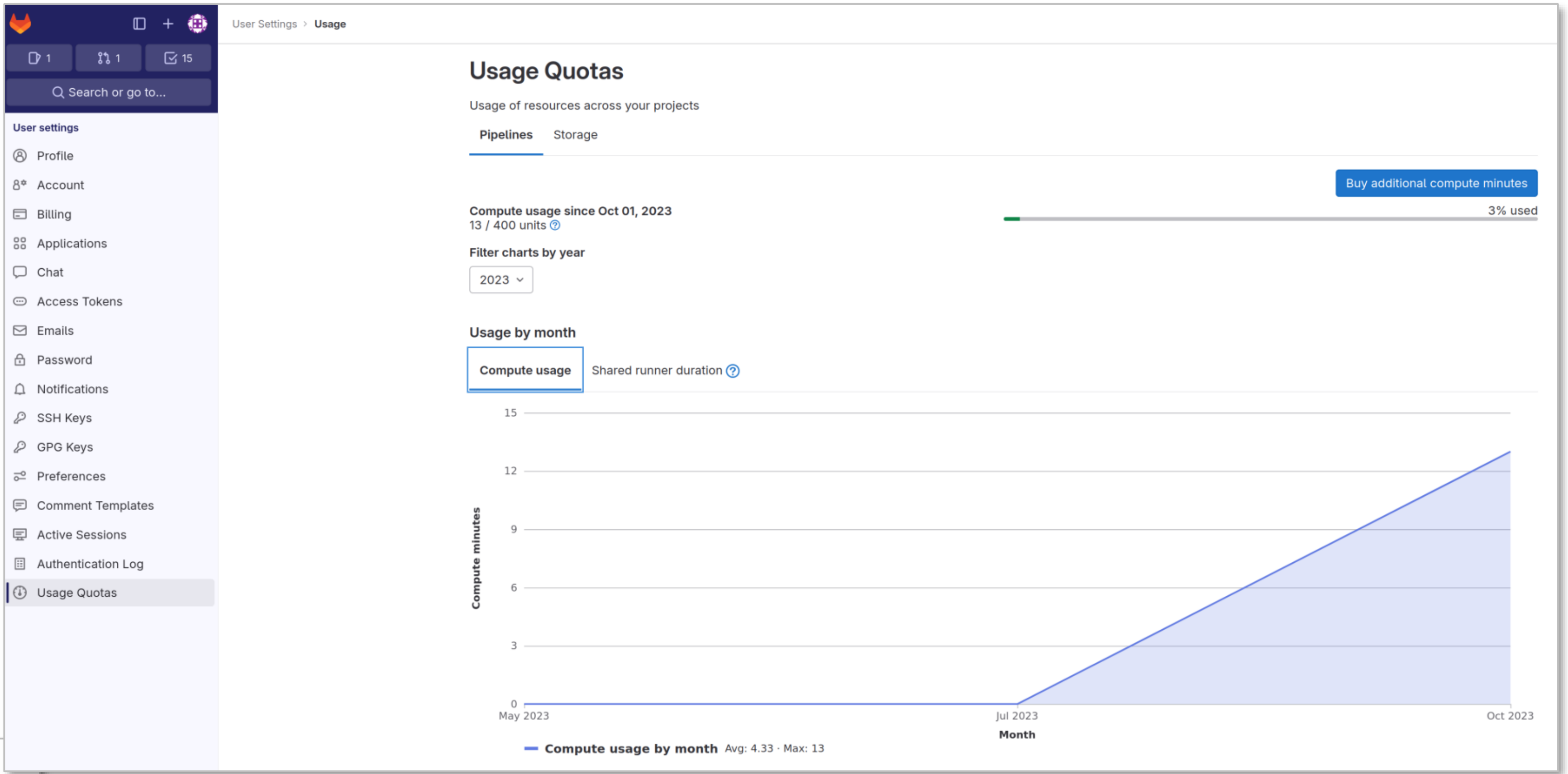
Usage quotas

There are monthly limits for using CICD pipelines (as of 2023):

- GitHub Actions: 2000 min/month
- GitLab CI/CD : 400 min/month

What to do if you need more compute time?

- Buy compute minutes from GitHub/GitLab
- Setup your own runner machine (e.g. on SWITCHengines)


User Settings > Usage

Usage Quotas

Usage of resources across your projects

Pipelines | Storage

Buy additional compute minutes

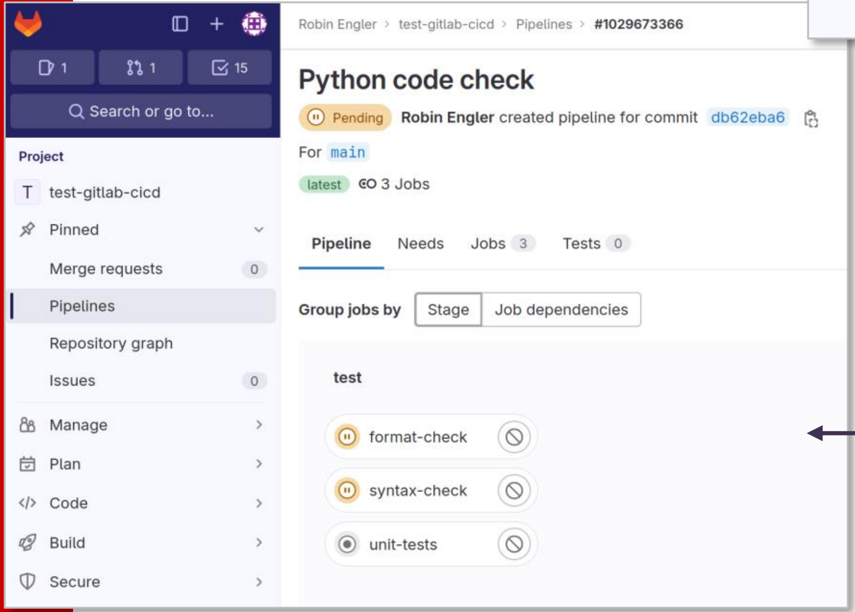
Compute usage since Oct 01, 2023
13 / 400 units

3% used

Filter charts by year: 2023

Usage by month

Compute usage | Shared runner duration


Robin Engler > test-gitlab-cicd > Pipelines > #1029673366

Python code check

Pending Robin Engler created pipeline for commit db62eba6

For main

latest 3 Jobs

Pipeline | Needs | Jobs 3 | Tests 0

Group jobs by Stage | Job dependencies

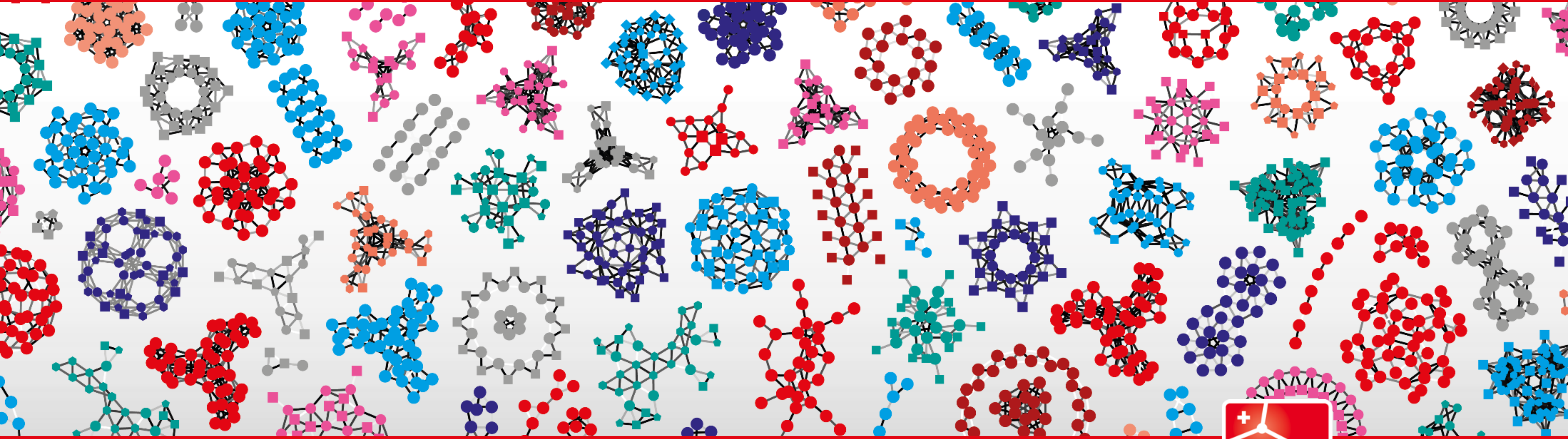
test

- format-check
- syntax-check
- unit-tests

When using the "free" runners, a job can sometimes be "pending" for a while, waiting for a free slot on the compute infrastructure.

exercise CI/CD

Exercise 1A -> GitHub Actions
Exercise 1B -> GitLab CI/CD



Thank you for attending this course

Swiss Institute of Bioinformatics