Good practices II

Raul P. Pelaez

March 11, 2026

Contents

1 Introduction

Today we will be learning about version control with `git`. Let me start by defining some basic terminology.

Info

Version control system (VCS)
A category of software tools that helps in recording changes made to files by keeping a track of modifications done in the code.

Info

Repository
A repository (or simply repo) is a collection of files accompanied by a database of changes. This database contains all the edits and historical versions (snapshots) of the project. Github is an example of a platform that hosts repositories.

Info

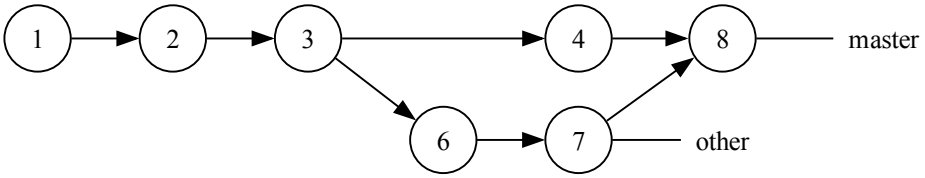
Commit: a snapshot of the project
A repository is "represented" or "encoded" as a collection of commits. A commit is a snapshot of the project at a certain point in time. Commits contain information about a set of changes in addition to metadata like the author, the date, and a message describing the changes.

Internally, the totality of the project is not stored at every commit, rather the first version of the project is stored and then a list of changes (commits) that take it from one snapshot to the next.

The commit representing the current state of the project is called the `HEAD` commit.

Commits are named by a unique alphanumeric *hash*, like:
`a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0`.

We can represent the history of a repository using a list of connected nodes (representing commits):



A continuous line of commits is called a branch. The default branch in a git repository is called `master` or `main`. Branches are used to develop features isolated from each other, and they can be merged back into the main branch when the feature is ready. Branches can be created from any commit, and can also be merged with others. The diagram above shows a simple example of a repository with two branches, `master` and `other`. At some point, the `other` branch was created from a commit in `master`, and it was merged back into the `master` branch at a later time.

1.1 Git and version control

Git's own webpage starts with:

> Git is a free and open source distributed version control system [...]

We already know what VCS means. The other term, **distributed**, means "collaborative", in the sense that it can understand several copies of the project existing and contributing changes to each other at the same time. There are other types of VCS, but they are basically reduced versions of distributed ones and we will not cover them. There are many VCS tools, but our choice is going to be `git`, the de-facto standard [1].

Advice

Usually I try to imbue into you that the ideas and workflows are more important than the tool itself, which is subject to change with time. With `git`, however, I bet it is going to keep being the standard for many many years to come.

[1] There is also subversion (svn), mercurial, ...

Info

git
A distributed version control system that serves two main purposes:

- Keep track of changes in the codebase.
- Facilitate collaboration between developers.

Use git correctly and your peace of mind will reach nirvana levels.

Warning

Git's power is only surpassed by its dangerousness
Think of git as a chainsaw; hand it to an experienced lumberjack and you will be warm next winter, but let a drunken monkey take care of it and see what happens... It is akin to when we learned about the `rm` or `mv` commands, you can do a lot of damage with them if you are not careful.

Git is infamous for its obscure syntax, with command names that transmit no information or are downright misleading[2] and lots of contextual behavior[3]. Many times the same action can be performed in several ways and other times the same command can be used to deal with completely orthogonal situations.
These situations may leave you wondering about the strange design choices of git. The key here is that there was (kinda) not a design choice involved. Git's command line interface (CLI) evolved organically over the years to accommodate new necessities[4] and most of the time this explains its oddities.
You are not forced, though, to use git's CLI to leverage git. There are several interfaces, textual and graphical, which use git under the hood, calling it for you when dealing with the typical workflows in a version-controlled project. Most IDEs have one (like VSCode or emacs[5]), and there are also standalone ones (like Github Desktop).

Advice

Online resources
There are countless resources on git online, GitHub provides a good one, git-scm is also really good.

We will learn the basics of git, which will greatly improve your personal workflow and the way you collaborate with others when dealing with text-based files, such as latex papers, reports and software. Come with me to the depths of git hell (an actual concept).

Advice

Why do you need a distributed version control system (VCS)?
Some benefits of a VCS:

1. Enables efficient collaboration (multiple people can work simultaneously on a single project).
2. Allows one developer to work on the same project from multiple computers.
3. Enables traceability of every small change.
4. Informs us about Who, When, What, Why changes have been made.
5. Each developer keeps and maintains a local copy, which are only merged after validation.
6. Allows to visit a snapshot of the project at any point in time.

[2] `git cherry-pick` is an actual command.
[3] The command `git checkout` can do things like transport the entire project to a different point in time, delete a file, resurrect a file and more depending on the name we give it as next argument.
[4] Git was created by Linus Torvalds to version control the Linux kernel codebase.
[5] The one in emacs is called magit, and it is life-changing

2 Git basics

Before we move on, it is useful to show an explicit example of what a git repository (let me just call this a "project") looks like. If you remember from our previous lesson, we created the skeleton for a project:

```
myproject/
|-- src/
|-- tests/
|-- environment.yml
|-- setup.py
|-- README.md
|-- .gitignore
|-- .git/
```

where I have omitted stuff below the root level and I have sneakily added the `.git` folder. This folder marks the `myproject` folder as a git repository.

Info

`git` is a commandline tool
Which means that we interact with it via our terminals. We invoke the various `git` functionalities by providing it with arguments, like so: `git <command> <options>`. For instance:

```
(base) raul@laptop:~$ git help
```

Lets start by *cloning* a repository to have as a reference. For example, to clone a repository called `torchmd-net` from the username `torchmd` from GitHub, you would run:
`git clone https://github.com/torchmd/torchmd-net`
(This is a research project I have worked on in the past). Feel free to clone any other repository you find interesting instead.

A new folder called `torchmd-net` will be created in your current directory, containing the project files (and the `.git` folder). `cd` into it and inspect it.

Info

Remote
A copy of the repository stored somewhere that is not the local copy (e.g., GitHub, some server) is called a remote. The default remote when you clone a repository is called `origin`. A local copy of a repository can have many remotes, but usually, you only have one. The `git remote` command is used to manage remotes. Try running `git remote show origin` from inside a repo.

Info

Cloning a repository
To clone a repository means to create a local copy of a remote. This is done with the `git clone` command.

Advanced

What is in the `.git` folder
In git, the "database of changes" that constitutes a repository is stored in a folder called .git in the root directory of the project. If you remove this folder you would still have the project's files at the current point in time, but you would have lost all information about its history or about the location of any remote copy of the repo.

We do not create or modify the `.git` folder directly, but we interact with it through the git CLI.

Info

Querying the status of the repo: `git status`
`git status` shows all relevant info about the current state of the repo, like which files have been modified, which files are staged for the next commit, and which files are untracked.

Info

The history of the project: `git log`
`git log` shows a list of all the commits in the project. It can be customized a lot, I usually like to do:

```
$ git log --oneline --graph
```

Info

Showing information about a commit: `git show`
`git show <commit>` will print all the information about a commit, including the changes made in it.

Advice

Getting help from git
If you are ever unsure about how to use a command, you can always ask git for help. For example, `git help clone` will show you the documentation for the `clone` command.

2.1 Adding commits to a repository

Thus far, we have only seen how to inspect the state of a repository. Let us now see how to create and add new commits to a repository.

Info

Creating a new repository
To create a new repository from scratch, you can run `git init` in the root directory of your project. This will create a new `.git` folder in the project directory. You can see this by running `ls -a`.

Once a repository is initialized, you can start adding files to it. Remember that repositories are a series of concatenated commits. When the state of the repository has been modified in some way, we need to somehow "pack" these changes into a new commit and add it on top of the repository's history. This is done in two steps: *Staging* and *committing*

Info

Staging: `git add`, `git rm`
When you make changes to a file, git does not automatically track them. You need to tell git to track the changes by *staging* them. This is done with the `git add` command. Similarly, `git rm` can be used to mark the removal of a file. For example:

```
$ git add file.txt
# or, to remove a file:
$ git rm file.txt
# or, to stage all changes in pwd and below:
$ git add .
```

Info

Committing: `git commit`
Once the changes have been staged, you can create a new commit with the `git commit` command, followed by a message describing the changes. For example:

```
$ git commit -m "Add a new feature"
```

Warning

You cannot rewrite history
Once a commit has been created, it is immutable. You can undo the changes by adding an "inverse" commit, but you cannot change the original commit. This means that once a file is in there, it is going to "pollute" the history of the project forever. Be careful not to commit spurious files, like the `.DS_Store` files that macOS creates. (You can technically travel to the past and rewrite history, but doing that will render the repository incompatible with any other copy of it, so better to assume it is not possible. Luckily, `git` will fight you at every step of the way if you try.)

2.2 Synchronizing with a remote

The seamless integration of git with remote repositories is its enabler of collaboration. We can push our changes to a remote repository and pull changes from it.

Info

Pushing changes to a remote: `git push`
Once you have created a commit, you can push it to a remote repository with the `git push` command. For example:

```
$ git push origin master
```

Will push the changes in the local copy of the `master` branch to the remote copy of the `master` branch in the `origin` remote.

Info

Bringing changes from a remote: `git pull`
If you are working in a team, you will need to bring changes from the remote repository to your local copy. This is done with the `git pull` command. For example:

```
$ git pull origin master
```

Will bring the commits in the remote copy of the `master` branch to the local copy of the `master` branch. If you have commits that are not in the remote, you will need to merge them (see below).

2.3 Traveling through time

Info

The `HEAD` commit
The commit that represents the current state of the project is called the `HEAD` commit. This is the commit that is currently checked out in the repository. You can see where `HEAD` is pointing by running `git log`.

Info

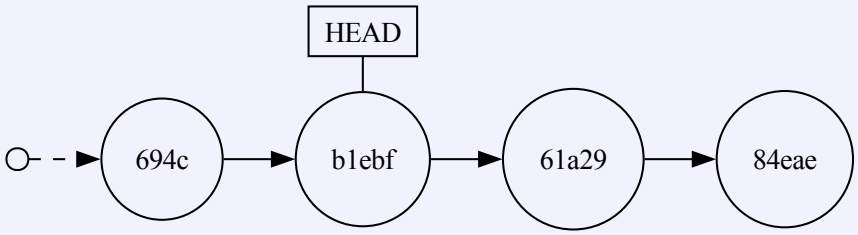
Visiting a commit
The `git checkout` command can be used to take the repo to the state it was just after the application of a certain commit.
Get into the repo you cloned before and use the log command to choose a particular commit, then run:

```
$ git checkout <commit_hash>
```

You will probably see git warning you about being in a "detached HEAD state", which means that you are not at the tip of a branch anymore
See where HEAD is pointing now by using `git log`. Try to go to the previous commit using

```
$ git checkout HEAD~1
```

Look for `HEAD` in `git log` again. You will end up with something like:



2.4 Branching and merging

Branching is a powerful feature of git that allows you to work on different features of a project in isolation. Branches are lightweight and can be created and deleted easily. Merging is the process of combining the changes from one branch into another.

Info

Creating a branch
To create and switch to a new branch use the `git checkout -b` command:

```
$ git checkout -b new-feature
```

In general, the `git branch` command deals with branches. Try to run `git branch` to see a list of branches.

After creating a branch, you can work on it as if it were the main branch. You can make changes, stage them, and commit them. When you are done with the feature, you can merge the branch back into the main branch (see the diagram at the beginning of the lesson).

Info

Merging a branch
To merge a branch into another, you need to be in the branch you want to merge into. For example, to merge the `new-feature` branch into the `master` branch, you would run:

```
$ git checkout master
$ git merge new-feature
```

If there are no conflicts, the changes in the `new-feature` branch will be added to the `master` branch. If there are conflicts, you will need to resolve them manually.

Info

Basic workflow with Git

1. `git init` or `git clone`: Initialize a new repository or clone an existing one.
2. `git status`: Check the status of your working directory and staging area.
3. `git add <files>`: Stage changes for the next commit.
4. `git commit -m "message"`: Commit the staged changes.
5. `git push <remote> <branch>`: Push the changes to a branch in a remote repository.
6. `git log`: See a list of commits.

Recommended practice: commit often, with small changes accompanied by clear commit messages.
Often, instead of committing directly to the `master` branch, developers create a new branch to work on a feature or a bug fix. This way, the `master` branch remains clean and stable. When the feature is ready, the branch is merged back into the `master` branch. In the above, we would add a step 2.5: `git checkout -b new-branch`, and a step 4.5: `git checkout master` followed by `git merge new-branch`.

2.5 The `.gitignore` file

The `.gitignore` file is a file that tells git to ignore certain files or directories. This is useful for files that are generated by the project, like compiled code, or for files that are specific to your local environment, like editor configuration files. You can create a `.gitignore` file in the root directory of the project and add the files you want to ignore, some level of wildcarding is allowed. For example:

```
*.DS_Store
*pycache*
.vscode/
*build*
```

2.6 Advanced functionality

Let me put some advanced commands in your radar. We do not have time to cover them today, but luckily you can get away with not knowing them during your first months of using git.

- `git cherry-pick`: Apply a commit from another branch.
- `git rebase`: Reapply commits on top of another base tip. This is the dangerous cousin of `git merge`.
- `git stash`: Temporarily store changes.

3 Exercises

3.1 Your first repo

Goal

Take your project from the previous lesson's exercises and turn it into a git repository. Make a first commit with its current state. Be careful about not committing files that are not necessary (like the `.DS_Store` files) by adding a `.gitignore` file.
Get familiar with the different `git` commands by inspecting the state of the repo, creating several commits, etc.

hint

- You can use the `git init` command to initialize a new repository in the project's root directory.
- Use the `git add` and `git commit` commands to add and commit the files in the project.

Milestone

Move around the project's history by using `git log` and the `git checkout` command to visit different commits.

hint

Your own repo might not have many commits, so you can use the `torchmd-net` repo (or any other) you cloned before to practice.

Milestone

Create a new branch and make some changes to the project. Merge the branch back into the main branch.

hint

- Use the `git checkout -b` command to create a new branch.
- Use the `git merge` command to merge the branch back into the main branch.
- Use `git log --oneline --graph` often to get a sense of the project's history.