Lecture 5 — Version control with Git

C1 Research Computing — MPhil in Data Intensive Science, Michaelmas 2026

Contents

1 Repositories and history **2**
- 1.1 Repositories and commits . . . 2
- 1.2 Working tree, index and history . . . 3

2 Branches and merges **6**
- 2.1 Branches . . . 7
- 2.2 Merges and conflicts . . . 8

3 Sharing a repository **11**
- 3.1 Remotes, forks and pull requests . . . 11
- 3.2 Ignored files, credentials and large data . . . 13

4 Correcting and marking history **14**
- 4.1 Reversing changes . . . 15
- 4.2 Tags and analysis provenance . . . 16

Troubleshooting **17**

Key points **18**

Course environment **18**

Exercises **19**

Solutions to the exercises **20**

The `attractor` project of the previous chapter estimates the cycle period of ten chemostat runs from the peaks of the algae density [1, 2]. Its script `analyse.py` has changed several times: NumPy 2 removed the name `np.NaN` and the script had to be edited, and the constant `MIN_GAP`, the smallest spacing allowed between two peaks, was changed while the method was tested. Without a record of these edits, a figure made last month cannot be traced to the version of the script that produced it, a test that made the periods worse cannot be undone with confidence, and two people editing the script cannot combine their changes. Git keeps that record. Every saved state of the project is a commit, every commit names its predecessor, and the differences between any two commits can be printed.

The outputs shown were produced with Git 2.39.5 as shipped by Apple on macOS, in a scratch clone of the public repository `github.com/borisbolliet/attractor` named `attractor-git-demo`. The commits made in the chapter are signed by a fictitious author, Ada Lovelace, and their identifiers are those of one run on 20 October 2026. A second run produces other identifiers, because the identifier of a commit depends on its time of creation (Section 1.1). The table of periods in Section 3.2 was produced with Python 3.12 and NumPy 2.4. Nothing in the chapter is pushed to the public repository.

1 Repositories and history

1.1 Repositories and commits

Definition 1.1: Repository, clone and commit

A **repository** is a project directory together with a `.git/` subdirectory that holds the complete history of the project. A **clone** is a copy of a repository, history included, made on another machine or in another directory. A **commit** is a saved snapshot of the tracked files of the project, together with a message, the name and email of its author, the date, and the identifiers of its parent commits. An initial commit has no parent, an ordinary commit has one, and a merge commit has two [3].

Git is a distributed version control system: every clone holds the whole history and can record new commits without a network connection. Linus Torvalds started it in April 2005, when the Linux kernel developers lost the free use of BitKeeper, the proprietary system they had used until then [4, 5]. It was designed for speed, for a large code base and for development by many people in parallel. Git is the version control program. GitHub, GitLab and Bitbucket are services that host remote copies of Git repositories and add access control, code review and issue tracking on top of them. The commands of this chapter are the same whichever service holds the remote copy.

Git records the contents and paths of the files it is told to track. It does not record which Python interpreter ran a script, which packages were installed, or the contents of data that stay outside the repository. Those belong in files of their own, a `requirements.txt`, a lock file, a data citation, which Git then versions like any other file, as the previous chapter set out. A commit therefore preserves the environment records that existed at that point of the history, not the environment itself.

Listing 1 clones the repository. The last argument of `git clone` names the local directory. A fresh clone has nothing to report in `git status`, and `git log` lists the five commits of the project, the newest first. The option `--oneline` prints one line per commit, `--graph` draws the parent links as a column of asterisks, and `--decorate` adds the names that point at a commit: the local branch `main`, the remote branch `origin/main`, and `HEAD`, the commit currently checked out [6]. A new repository without a remote is started with `git init -b main` instead of `git clone`.

```
$ git clone https://github.com/borisbolliet/attractor.git attractor-git-demo
Cloning into 'attractor-git-demo'...
$ cd attractor-git-demo
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
$ git log --oneline --graph --decorate
* 63e94ea (HEAD -> main, origin/main, origin/HEAD) Use np.nan
* 09a7ce2 Snapshot wording, two-line LLM answer with rounded periods, setup comment
* 6b672dd Ignore the lecture's second venv and result folders
* 30470f1 Kernel attractor-py312, connection-error guard in ask_llm.py, README setup
    details
* e5eb8bb Reproducible-environment demo: chemostat cycle periods
```

Listing 1: A clone of the `attractor` repository and its history. The commit at the top, "Use np.nan", is the fix made during the lecture.

The seven characters in front of each message are the beginning of the **commit identifier**, a 40-character hexadecimal string (Listing 2). Git computes it as the SHA-1 hash of the commit object, that is of the snapshot, the message, the author, the dates and the parent identifiers together [3]. Two commits with the same content made at different times therefore have different identifiers, and a commit cannot be altered without changing its identifier and those of every commit that descends from it. The abbreviated form is accepted wherever a commit is named, as long as it is unique in the repository. Git shows at least seven characters and more when seven would be ambiguous.

> A commit identifier is a hash, not a sequential number. There is no commit 1, 2, 3, and the identifiers of two consecutive commits share nothing. The order of commits is read from the parent links, which `git log` follows from the newest commit backwards.

```
$ git log -1
commit 63e94eafdaf744dcefa244b2079aaf78885c0d18
Author: Ada Lovelace <ada@example.org>
Date:   Fri Oct 16 09:41:53 2026 +0100

    Use np.nan
$ git rev-parse HEAD
63e94eafdaf744dcefa244b2079aaf78885c0d18
```

Listing 2: The commit at `HEAD` in full and its 40-character identifier.

`git show` prints a commit with the changes it made (Listing 3). The commit "Use np.nan" replaced `np.NaN` by `np.nan` in the two places where `analyse.py` used it. A line that begins with a minus sign is the old version and a line that begins with a plus sign the new one. The numbers in `@@ -46,7 +46,7 @@` locate the changed lines in the old and the new file: seven lines starting at line 46 in both. Git stores the full snapshot of each commit and computes such differences on demand.

> NumPy 2.0 removed the alias `np.NaN`, in capitals. The lower-case `np.nan` is the surviving name and works in NumPy 1 and 2 [7]. The fix replaces the capitalised spelling by the lower-case one, not the other way round.

Every commit records an author name and email. Listing 4 sets them for the demo clone with `git config`, which writes two lines into the file `.git/config` of this repository [8]. The same file holds the URL of the remote and the branch that `main` follows. With the option `--global` the setting is written to `~/.gitconfig` instead and applies to every repository of the user.

> `.git/config` is a plain text file in the INI format, not a script, and it belongs to one clone. A name set without `--global` applies to that clone only, which is why a fictitious author in a demo clone leaves the other repositories on the lecture laptop unchanged.

1.2 Working tree, index and history

Definition 1.2: The three areas of a repository

The **working tree** is the set of project files on disk, the ones an editor opens. The **index**, also called the staging area, holds the content proposed for the next commit. The **history** is the set of commits and the file contents they refer to, stored inside `.git/`. `git add` copies the current version of a file from the working tree into the index, and `git commit` records the index as a new commit [9].

```
$ git show --stat 63e94ea
commit 63e94eafdaf744dcefa244b2079aaf78885c0d18
Author: Ada Lovelace <ada@example.org>
Date:   Fri Oct 16 09:41:53 2026 +0100

    Use np.nan

 analyse.py | 4 ++--
 1 file changed, 2 insertions(+), 2 deletions(-)
$ git show 63e94ea
[the header above, then]
@@ -46,7 +46,7 @@ def cycle_period(t, algae):
     """Mean peak-to-peak spacing in days, and the peak times."""
     p = peaks(t, algae)
     if len(p) < 3:
-        return np.NaN, t[p]
+        return np.nan, t[p]
     return (t[p[-1]] - t[p[0]]) / (len(p) - 1), t[p]


@@ -63,7 +63,7 @@ def main():
         peak_times.append(tp)
     # peak times padded to a common length, one row per run
     width = max(len(tp) for tp in peak_times)
-    table = np.full((len(files), width), np.NaN)
+    table = np.full((len(files), width), np.nan)
     for i, tp in enumerate(peak_times):
         table[i, : len(tp)] = tp
```

Listing 3: The changes recorded by the commit "Use np.nan": two lines of `analyse.py`, each replaced by its lower-case version.

```
$ git config user.name "Ada Lovelace"
$ git config user.email ada@example.org
$ git config --local --list
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
remote.origin.url=https://github.com/borisbolliet/attractor.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
branch.main.merge=refs/heads/main
user.name=Ada Lovelace
user.email=ada@example.org
```

Listing 4: The author recorded on the commits of the demo clone, and the local configuration file it is written to. The two `remote` and `branch` entries were written by `git clone`.

The index exists so that a commit can contain a chosen subset of the edits on disk. A session of work often touches several files for different reasons, and the index lets each reason become its own commit. Figure 1 shows the three areas and the commands that move content between them. `git status` compares the three areas and reports, for each file, whether the working tree differs from the index and whether the index differs from the last commit [10]. `git diff` prints the first difference, the edits not yet staged, and `git diff --staged` the second, the content of the next commit [11].

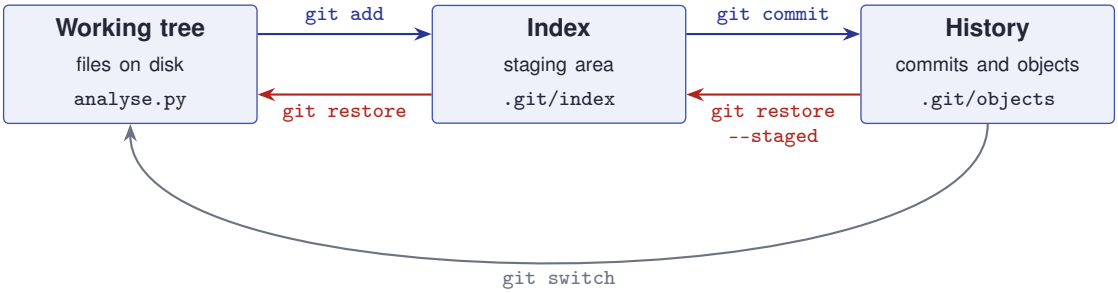


Figure 1: The three areas of a repository and the commands that copy content between them. `git add` and `git commit` move content to the right. `git restore` and `git restore --staged` copy it back to the left (Section 4.1). `git switch` rewrites both the index and the working tree from a commit (Section 2.1).

Listing 5 changes `MIN_GAP` from 3.0 to 3.5 days in `analyse.py` with an editor and stages the edit. Before `git add`, the difference is between the working tree and the index. After it, `git diff` prints nothing and the same difference appears under `--staged`.

```
$ git diff -- analyse.py
diff --git a/analyse.py b/analyse.py
index 86b74f9..cd88f34 100644
--- a/analyse.py
+++ b/analyse.py
@@ -16,7 +16,7 @@ RESULTS = Path(os.environ.get("RESULTS_PATH", "results"))
 os.environ.setdefault("MPLBACKEND", "Agg")
 import matplotlib.pyplot as plt  # noqa: E402  (after the backend is set)

-MIN_GAP = 3.0  # minimum spacing between two peaks, in days
+MIN_GAP = 3.5  # minimum spacing between two peaks, in days


 def load_run(path):
$ git add analyse.py
$ git diff -- analyse.py
$ git diff --staged -- analyse.py
[the same difference as above]
```

Listing 5: An edit to `MIN_GAP` before and after staging. `git diff` compares the working tree with the index and `git diff --staged` the index with the last commit.

The index holds the version of the file at the time of `git add`, not the file itself. Listing 6 edits `MIN_GAP` a second time, to 4.0 days, after the first edit was staged. `git diff` now shows the step from 3.5 to 4.0 and `git diff --staged` still shows the step from 3.0 to 3.5. `git status` lists the file twice, once under each heading, and the short form prints the two-letter code `MM`: the first column describes the index and the second the working tree (Table 1). A commit at this point records the 3.5-day version. In Listing 7, `git restore analyse.py` copies the index back to the working tree and drops the 4.0-day edit. The commit "Test peak spacing" then records the 3.5-day version, and `git log` shows `main` one commit ahead of `origin/main`, which still points

at the commit fetched from GitHub.

> An edit made after `git add` is not in the next commit. The index holds the version of the file at the moment it was added, and `git status --short` marks the situation with `MM`. A second `git add` of the file stages the latest version.

```
$ git diff -- analyse.py
[...]
-MIN_GAP = 3.5  # minimum spacing between two peaks, in days
+MIN_GAP = 4.0  # minimum spacing between two peaks, in days
$ git diff --staged -- analyse.py
[...]
-MIN_GAP = 3.0  # minimum spacing between two peaks, in days
+MIN_GAP = 3.5  # minimum spacing between two peaks, in days
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   analyse.py

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

$ git status --short
MM analyse.py
```

Listing 6: A second edit after staging: the index keeps the 3.5-day version while the working tree holds 4.0, and `git status` lists the file under both headings.

```
$ git restore analyse.py
$ grep -n "^MIN_GAP" analyse.py
19:MIN_GAP = 3.5  # minimum spacing between two peaks, in days
$ git status --short
M  analyse.py
$ git commit -m "Test peak spacing"
[main 02a6234] Test peak spacing
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate
* 02a6234 (HEAD -> main) Test peak spacing
* 63e94ea (origin/main, origin/HEAD) Use np.nan
* 09a7ce2 Snapshot wording, two-line LLM answer with rounded periods, setup comment
[...]
```

Listing 7: The unstaged edit discarded with `git restore`, and the staged version committed. `main` is now one commit ahead of `origin/main`.

2 Branches and merges

Table 1: Codes of `git status --short`. The first column compares the index with the last commit and the second the working tree with the index. A space means no difference. The codes `M`, `MM`, `A`, `D` and `??` appear in the listings of this chapter.

code	columns	meaning
`M`	index	modified and staged, the working tree agrees with the index
` M`	working tree	modified on disk, not staged
`MM`	both	staged once, then edited again
`A`	index	new file staged
`D`	index	removal staged, by `git rm` or `git rm --cached`
`??`	neither	untracked, never added and not ignored
`UU`	both	both branches changed the file in a merge that has not been resolved

2.1 Branches

Definition 2.1: Branch and `HEAD`

A **branch** is a name that points at a commit. A commit made while the branch is checked out becomes the new target of the name. The default branch is called `main`. **HEAD** is the name of the branch currently checked out, and so of the commit that the next commit will have as its parent [12].

A branch costs one line in a file under `.git/refs/heads/`, which is why branches are created freely: one per experiment, one per fix. Figure 2 shows a history in which two branches have diverged. Commits A to C are shared, D was made on `main` and E on `feature`, and `HEAD` marks `feature` as the branch checked out. Each arrow points from a commit to its parent, which is the direction stored in the commits themselves.

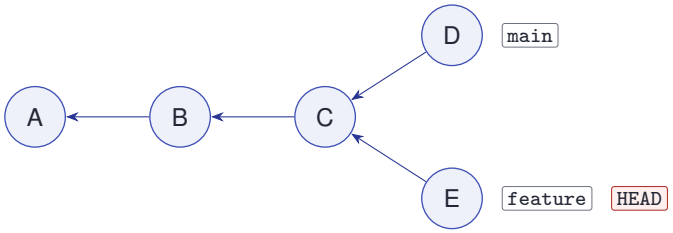


Figure 2: A commit graph with two branches. Each arrow points from a commit to its parent. `main` and `feature` are names that point at the commits D and E. `HEAD` names the checked-out branch, here `feature`, so the next commit will have E as its parent.

Listing 8 lists the branches, creates one and deletes it. `git switch -c feature` creates `feature` at the current commit and checks it out. Both names point at the same commit until a commit is made on one of them, as the decorated log line shows. `git switch main` returns to `main` and rewrites the working tree and the index to the files of that commit. Git refuses to switch when an uncommitted edit would be overwritten by the change of files, and the edit must be committed or discarded first [13, 14].

The case of the letter in `git branch -d` matters. The lower-case `-d` deletes a branch only when its commits are already merged into the current branch and refuses otherwise, as in Listing 9. The upper-case `-D` forces the deletion and abandons the unmerged commits, which stay reachable only through `git reflog` for a time, 30 days by default [14, 15].

```
$ git branch
* main
$ git switch -c feature
Switched to a new branch 'feature'
$ git branch
* feature
  main
$ git log --oneline --decorate -1
02a6234 (HEAD -> feature, main) Test peak spacing
$ git switch main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
$ git branch -d feature
Deleted branch feature (was 02a6234).
```

Listing 8: A branch created at the current commit and deleted again. The asterisk marks the checked-out branch.

```
$ git switch -c scratch
Switched to a new branch 'scratch'
$ printf 'step_size = 0.1\n' > settings.py
$ git commit -am "Try a huge step"
[scratch 710aa6f] Try a huge step
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git switch main
Switched to branch 'main'
$ git branch -d scratch
error: The branch 'scratch' is not fully merged.
If you are sure you want to delete it, run 'git branch -D scratch'.
$ git branch -D scratch
Deleted branch scratch (was 710aa6f).
```

Listing 9: A branch with an unmerged commit cannot be deleted with `-d`. The option `-a` of `git commit` stages every tracked file that was modified before committing.

`git checkout`, the older command, does two unrelated jobs: changing branch and overwriting files from the index or from a commit. Git 2.23 of 2019 split them into `git switch` for branches and `git restore` for files, so that `git switch -c feature` equals `git checkout -b feature` and `git restore analyse.py` equals `git checkout -- analyse.py` [13]. The older spelling still works and appears in most tutorials.

2.2 Merges and conflicts

Definition 2.2: Fast-forward, merge commit and conflict

`git merge other` integrates the commits of the branch `other` into the checked-out branch. When the checked-out branch is an ancestor of `other`, Git moves its name forward to the commit of `other` and creates nothing. This is a **fast-forward**. When both branches have commits after their common ancestor, Git creates a **merge commit** whose two parents are the two branch tips. A **conflict** arises when the two branches changed the same lines of a file, or when one deleted a file that the other modified. Git then stops and leaves the decision to the author [16, 17].

Figure 3 draws the two cases with the commits of this chapter. Listing 10 produces the fast-forward. The branch `small-step` adds a file `settings.py` holding a step size of 0.01 and commits it. The file is absent from the working tree once `main` is checked out again, because the commit that created it is not in the history of `main`. The merge moves `main` to the commit of `small-step`, and the graph stays a single line.

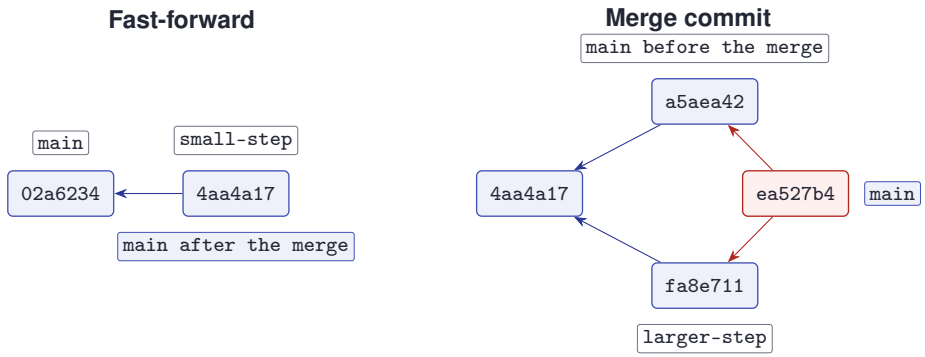


Figure 3: The two outcomes of `git merge`, with the commits of Listings 10 and 13. Left: `main` is an ancestor of `small-step`, so the merge moves the name `main` forward and creates no commit. Right: both branches have advanced since 4aa4a17, so the merge creates the commit ea527b4 with two parents.

```
$ git switch -c small-step
Switched to a new branch 'small-step'
$ printf 'step_size = 0.01\n' > settings.py
$ git status --short
?? settings.py
$ git add settings.py
$ git commit -m "Add step-size setting"
[small-step 4aa4a17] Add step-size setting
 1 file changed, 1 insertion(+)
 create mode 100644 settings.py
$ git switch main
Switched to branch 'main'
$ ls settings.py
ls: settings.py: No such file or directory
$ git merge small-step
Updating 02a6234..4aa4a17
Fast-forward
 settings.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 settings.py
$ git log --oneline --graph --decorate
* 4aa4a17 (HEAD -> main, small-step) Add step-size setting
* 02a6234 Test peak spacing
* 63e94ea (origin/main, origin/HEAD) Use np.nan
[...]
$ git branch -d small-step
Deleted branch small-step (was 4aa4a17).
```

Listing 10: A fast-forward merge. `settings.py` does not exist on `main` until the merge moves `main` to the commit that created it.

Listing 11 makes the same setting diverge. The branch `larger-step` sets the step size to 0.02 and `main` sets it to 0.005, both from the common ancestor 4aa4a17. The graph with `--all` shows the fork. In Listing 12 the merge stops with a conflict in `settings.py`, and `git status` lists the file under "Unmerged paths". The file on disk now holds both versions between markers:

the lines between `<<<<<<< HEAD` and `=======` come from the checked-out branch and the lines between `=======` and `>>>>>>> larger-step` from the branch being merged.

```
$ git switch -c larger-step
Switched to a new branch 'larger-step'
$ printf 'step_size = 0.02\n' > settings.py
$ git add settings.py
$ git commit -m "Test larger step"
[larger-step fa8e711] Test larger step
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git switch main
Switched to branch 'main'
$ printf 'step_size = 0.005\n' > settings.py
$ git add settings.py
$ git commit -m "Test smaller step"
[main a5aea42] Test smaller step
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git log --oneline --graph --decorate --all
* fa8e711 (larger-step) Test larger step
| * a5aea42 (HEAD -> main) Test smaller step
|/
* 4aa4a17 Add step-size setting
[...]
```

Listing 11: The same line changed on two branches since their common ancestor 4aa4a17.

```
$ git merge larger-step
Auto-merging settings.py
CONFLICT (content): Merge conflict in settings.py
Automatic merge failed; fix conflicts and then commit the result.
$ git status
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   settings.py

no changes added to commit (use "git add" and/or "git commit -a")
$ cat settings.py
<<<<<<< HEAD
step_size = 0.005
=======
step_size = 0.02
>>>>>>> larger-step
```

Listing 12: The merge stops with a conflict and leaves both versions in the file between markers.

The resolution is written into the file with an editor: the markers are removed and the lines that should survive are kept (Listing 13). `git add` marks the file as resolved and `git commit` completes the merge. Git composes the message "Merge branch 'larger-step'" when none is

given. The graph now has a merge commit with two parents, which the format `%p` of `git log` prints as two identifiers. `git merge --abort` abandons the merge instead and returns the branch to its state before `git merge`. Git can detect that the two branches changed the same line and can keep both proposals, but it cannot know whether a step size of 0.005 or 0.02 is the right one for the analysis. The value kept is a modelling decision, and the commit message should say why it was taken.

```
$ printf 'step_size = 0.005\n' > settings.py
$ git add settings.py
$ git commit -m "Resolve step-size conflict"
[main ea527b4] Resolve step-size conflict
$ git log --oneline --graph --decorate
*   ea527b4 (HEAD -> main) Resolve step-size conflict
|\
| * fa8e711 (larger-step) Test larger step
* | a5aea42 Test smaller step
|/
* 4aa4a17 Add step-size setting
* 02a6234 Test peak spacing
* 63e94ea (origin/main, origin/HEAD) Use np.nan
[...]
$ git log --pretty="%h %p %s" -4
ea527b4 a5aea42 fa8e711 Resolve step-size conflict
a5aea42 4aa4a17 Test smaller step
fa8e711 4aa4a17 Test larger step
4aa4a17 02a6234 Add step-size setting
$ git branch -d larger-step
Deleted branch larger-step (was fa8e711).
```

Listing 13: The conflict resolved in favour of the smaller step. The merge commit ea527b4 has the two branch tips as parents.

3 Sharing a repository

3.1 Remotes, forks and pull requests

Definition 3.1: Remote and remote-tracking branch

A **remote** is a named URL of another copy of the repository. `git clone` names the repository it copied from `origin`. A **remote-tracking branch** such as `origin/main` is a local name that records where the branch `main` of the remote was at the last exchange with it. It is updated by `git fetch` and `git push`, never by a commit [18, 19].

Listing 14 inspects the remote of the demo clone. `git remote -v` prints the URL used to fetch and the URL used to push, here the same. `git fetch origin` downloads the commits that the remote has and the clone does not, and moves the remote-tracking branches. It does not touch the local `main`, the index or the working tree, so the downloaded commits can be inspected before they are integrated. The range `main..origin/main` lists the commits on `origin/main` that `main` lacks. It is empty here, since nothing was pushed to GitHub since the clone. The reverse range `origin/main..main` lists the five commits made in this chapter, the ones `git status` counts as "ahead".

```
$ git remote -v
origin  https://github.com/borisbolliet/attractor.git (fetch)
origin  https://github.com/borisbolliet/attractor.git (push)
$ git branch -a
* main
  remotes/origin/HEAD -> origin/main
  remotes/origin/main
$ git fetch origin
$ git log --oneline main..origin/main
$ git log --oneline origin/main..main
ea527b4 Resolve step-size conflict
a5aea42 Test smaller step
fa8e711 Test larger step
4aa4a17 Add step-size setting
02a6234 Test peak spacing
$ git branch -vv
* main ea527b4 [origin/main: ahead 5] Resolve step-size conflict
```

Listing 14: The remote of the demo clone, a fetch that finds nothing new, and the two commit ranges between `main` and `origin/main`.

`remotes/origin/HEAD` in the output of `git branch -a` is not the local `HEAD`. It is a reference set at clone time that records the default branch of the remote, which is why it prints as `-> origin/main`. The local `HEAD`, the checked-out branch, is the asterisk before `main` in the same listing [18].

Integrating the fetched commits is a merge of `origin/main` into `main`, with the same two outcomes as in Section 2.2. `git pull origin main` runs the fetch and the merge in one command, so it is the one command of the pair that changes the working tree, and it can stop with a conflict like any merge [20]. Git merges by default and can be configured to rebase the local commits onto the fetched ones instead. `git push -u origin feature` sends the local branch `feature` to `origin` and records `origin/feature` as its upstream, so that a later `git push` on that branch needs no arguments [21]. Listing 15 gathers these commands. They were not run in the demo clone, which has no permission to push to the public repository.

```
$ git pull origin main          # fetch, then merge into the checked-out branch
$ git push -u origin feature    # publish feature and record its upstream
$ git push                      # later pushes of the same branch
```

Listing 15: The commands that exchange commits with a remote. They were not run in the demo clone and no output is shown.

`user.name` and `user.email` are labels written into the commit and play no part in authentication. A push is accepted or refused according to the credentials of the clone: a personal access token or the GitHub CLI login for an HTTPS remote, a key registered with the account for an SSH remote [22]. A commit signed "Ada Lovelace" pushes without objection from a clone whose owner has write access.

A **fork** is a copy of a repository hosted under another account [23]. It is the usual route to contribute to a project that one has no permission to push to. A **pull request**, called a merge request on GitLab, asks the maintainers of the source repository to review and integrate a branch [24, 25]. The request page shows the difference between the branch and the target, collects comments, and is updated by every push to the branch. The sequence is the following.

- Fork the source repository on the hosting service.

- Clone the fork. Its URL becomes `origin`, and the source repository is often added as a second remote named `upstream`.
- Create a branch, commit the change, and push the branch to the fork with `git push -u origin branch`.
- Open a pull request from the branch of the fork to the default branch of the source repository.

3.2 Ignored files, credentials and large data

Definition 3.2: Tracked, untracked and ignored files

A **tracked** file is one recorded in the index or in `HEAD`. An **untracked** file exists in the working tree and has never been added. An **ignored** file is an untracked file whose path matches a pattern in a `.gitignore` file. It is left out of `git status` and of `git add` with a directory argument, so that it is never committed by accident [26].

The `.gitignore` of `attractor` (Listing 16) excludes the virtual environments, the `.env` file of local configuration, the `results` directories and the bytecode caches. These are the files that are either generated from the tracked ones or specific to one machine. Running `analyse.py` writes `periods.csv` and `periods.png` into `results/`, and `git status` stays silent, which `git check-ignore -v` explains by naming the rule and its line [27]. The file `.gitignore` is itself tracked, so that every clone applies the same rules.

```
$ cat .gitignore
.venv*/
.env
results*/
__pycache__/
.ipynb_checkpoints/
$ python analyse.py
run    samples  peaks  period (days)
C1         359     50          7.408
C2         189     31          5.800
C3         182     24          7.079
C4         139     20          7.159
C5          74     12          6.671
C6         231     24          9.653
C7         234     24          7.306
C8         184     21          8.452
C9         141     17          8.059
C10        197     24          7.783
written: results/periods.csv, results/periods.png
$ git status --short
$ git check-ignore -v results/periods.csv
.gitignore:3:results*/    results/periods.csv
```

Listing 16: The ignore rules of `attractor`, the output of `analyse.py`, and the rule that hides the results from `git status`.

`.gitignore` hides untracked paths from `git status` and `git add`. It does not decide what is pushed: everything already committed travels with the history whatever the rules say, and a rule added after a file was committed has no effect on that file. To stop tracking a committed file, `git rm --cached path` removes it from the index while the copy on disk stays [28].

Listing 17 shows the mistake and its repair. Git refuses the plain `git add` of an ignored path and accepts it with `-f`, so the results table is committed. `git rm --cached` then stages its removal, which `git status --short` marks with `D` in the index column, and the file is still on disk. After the commit the file is untracked again and the ignore rule hides it. The earlier commit still contains it, as every commit keeps its snapshot.

```
$ git add results/periods.csv
The following paths are ignored by one of your .gitignore files:
results
hint: Use -f if you really want to add them.
$ git add -f results/periods.csv
$ git commit -m "Add results table"
[main 5537177] Add results table
 1 file changed, 11 insertions(+)
 create mode 100644 results/periods.csv
$ git rm --cached results/periods.csv
rm 'results/periods.csv'
$ git status --short
D  results/periods.csv
$ ls results
periods.csv
periods.png
$ git commit -m "Stop tracking generated results"
[main da4dd83] Stop tracking generated results
 1 file changed, 11 deletions(-)
 delete mode 100644 results/periods.csv
$ git status --short
```

Listing 17: A generated file committed by force, then removed from the index while it stays on disk.

Credentials such as API keys are never committed. The previous chapter placed them in an ignored `.env` file, and `attractor` tracks a `.env.example` that documents the variable names with harmless values. A credential that has been pushed is compromised, because the history of a public repository is readable by anyone and is copied by every clone. Deleting the file in a later commit leaves the earlier commit in place, as the third exercise shows. The only repair is to revoke the credential and issue a new one. GitHub scans pushed commits for known key formats and notifies the owner [29].

Data follow a different rule. A small, redistributable file that the analysis needs is committed with the code, as the ten CSV files of `attractor` are. Large or restricted data stay in an archive such as figshare or Zenodo and are cited from the project documentation, as the previous chapter did for the chemostat runs. At the time of writing, October 2026, GitHub refuses a push that contains a file larger than 100 MiB and warns from 50 MiB [30]. Git Large File Storage, LFS, is the extension for files that must be versioned despite their size [31].

> Git LFS does not store the large files beside the project. It replaces each one in the history by a pointer file of a few lines, holding the hash and size of the content, and keeps the content on a separate LFS server. The pointers travel with the ordinary history and the content is fetched on checkout, so a clone without the LFS extension receives the pointers only [31].

4 Correcting and marking history

4.1 Reversing changes

Three situations call for three commands, according to where the change sits in Figure 1. A change staged by mistake is removed from the index with `git restore --staged path`, which copies the version of `HEAD` back into the index and leaves the working tree alone. A change on disk that should be dropped is overwritten with `git restore path`, which copies the index back into the working tree. This second command discards work and cannot be undone, so `git diff` is run before it [32]. A change that has been committed is undone with `git revert commit`, which creates a new commit applying the inverse of the old one [33]. Listing 18 shows the first two on an edit of `MIN_GAP` to 2.5 days. After `git restore --staged`, the status code moves from the index column to the working-tree column, and after `git restore` the file is back at its committed version.

```
$ git add analyse.py
$ git status --short
M  analyse.py
$ git restore --staged analyse.py
$ git status --short
 M analyse.py
$ git diff -- analyse.py
[...]
-MIN_GAP = 3.5  # minimum spacing between two peaks, in days
+MIN_GAP = 2.5  # minimum spacing between two peaks, in days
$ git restore analyse.py
$ git status --short
$ grep -n "^MIN_GAP" analyse.py
19:MIN_GAP = 3.5  # minimum spacing between two peaks, in days
```

Listing 18: An edit unstaged, then discarded. The code `M` moves from the first column to the second and then disappears.

A plain `git reset` does not erase changes. It is the `--mixed` mode, which unstages everything and leaves the working tree untouched, as Listing 19 shows. The command that discards every staged and unstaged change and returns the files to the last commit is `git reset --hard`, and the uncommitted edits it removes cannot be recovered [34].

```
$ git add analyse.py
$ git reset
Unstaged changes after reset:
M       analyse.py
$ git status --short
 M analyse.py
$ git restore analyse.py
```

Listing 19: A plain `git reset` unstages the edit and keeps it on disk.

The commit “Test peak spacing” of Section 1.2 raised `MIN_GAP` to 3.5 days. Listing 20 reverts it. Git writes a new commit whose message names the reverted one, and the constant is back at 3.0 days. The commit of 3.5 days stays in the history, so a collaborator who already fetched it sees a consistent record. A revert can itself conflict when later commits changed the same lines, and the conflict is resolved as in Section 2.2. Rewriting the history to make a commit disappear is possible but is reserved for commits that nobody else has fetched.

```
$ git log --oneline -8
da4dd83 Stop tracking generated results
5537177 Add results table
ea527b4 Resolve step-size conflict
a5aea42 Test smaller step
fa8e711 Test larger step
4aa4a17 Add step-size setting
02a6234 Test peak spacing
63e94ea Use np.nan
$ git revert --no-edit 02a6234
[main 577cdfc] Revert "Test peak spacing"
 Date: Tue Oct 20 23:06:45 2026 +0100
 1 file changed, 1 insertion(+), 1 deletion(-)
$ grep -n "^MIN_GAP" analyse.py
19:MIN_GAP = 3.0  # minimum spacing between two peaks, in days
$ git show --stat HEAD
commit 577cdfc6adb2344336c47176d2e1017d74ec6580
Author: Ada Lovelace <ada@example.org>
Date:   Tue Oct 20 23:06:45 2026 +0100

    Revert "Test peak spacing"

    This reverts commit 02a62348a4d9297ed312cb194e9e0016aa11b9c4.

 analyse.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Listing 20: A committed change undone by a new commit. `--no-edit` accepts the message composed by Git.

4.2 Tags and analysis provenance

A branch moves with every commit. An **annotated tag** is a name that stays on one commit and carries a message, the name of its creator and a date [35, 36]. It is the device for naming the version of the code that produced a reported result. Listing 21 tags the current commit `analysis-v1` with the message “Code used for Figure 2”. `git show` prints the tag object followed by the commit it points at [37], and `git log --decorate` shows the tag beside the branch name. The tagged commit fixes the code. The data and the environment that the figure also depends on must be recorded in the same commit, as the tracked CSV files and the lock file of `attractor` are, or linked from it through a stable archive.

> A tag is created locally and `git push` does not send it on its own. The tag reaches the remote only with `git push origin analysis-v1`, or `git push origin --tags` for all tags [35]. The last line of Listing 21 asks the remote for its tags and finds none.

```
$ git tag -a analysis-v1 -m "Code used for Figure 2"
$ git tag
analysis-v1
$ git show --stat analysis-v1
tag analysis-v1
Tagger: Ada Lovelace <ada@example.org>
Date:   Tue Oct 20 23:06:45 2026 +0100

Code used for Figure 2

commit 577cdfc6adb2344336c47176d2e1017d74ec6580
Author: Ada Lovelace <ada@example.org>
Date:   Tue Oct 20 23:06:45 2026 +0100

    Revert "Test peak spacing"
[...]
$ git log --oneline --decorate -1
577cdfc (HEAD -> main, tag: analysis-v1) Revert "Test peak spacing"
$ git ls-remote --tags origin
```

Listing 21: An annotated tag on the commit at `HEAD`. The remote has no tags, since none was pushed.

Troubleshooting

Six symptoms point back to the areas and commands of this chapter.

symptom	cause	remedy
The commit holds an older version of a file than the one on disk, and `git status --short` prints `MM`.	The file was edited after `git add`, and the index kept the earlier version (Definition 1.2).	`git add` the file again before committing.
`git switch` refuses with "Your local changes … would be overwritten".	An uncommitted edit touches a file that differs between the two branches (Section 2.1).	Commit the edit on the current branch, or discard it with `git restore`.
`git merge` stops with `CONFLICT (content)`.	Both branches changed the same lines (Definition 2.2).	Edit the file, remove the markers, `git add` it and `git commit`, or `git merge --abort`.
`git branch -d` answers "not fully merged".	The branch has commits that the current branch does not contain (Listing 9).	Merge the branch first, or `git branch -D` if its commits are not wanted.
A file in the working tree never appears in `git status`.	A `.gitignore` rule matches it (Definition 3.2).	`git check-ignore -v path` names the rule. `git add -f` overrides it when the file is meant to be tracked.
A credential appears in a pushed commit.	The history keeps every committed file, whatever a later commit removes (Section 3.2).	Revoke the credential and issue a new one. Then remove the file and add it to `.gitignore`.

Key points

- A commit is a snapshot of the tracked files with a message, an author, a date and its parents, named by the SHA-1 hash of all of these (Definition 1.1).
- `git add` copies a file from the working tree into the index, and `git commit` records the index. An edit made after the add is not in the commit, which `git status --short` reports as `MM` (Definition 1.2, Table 1).
- A branch is a name on a commit. A merge is a fast-forward when the checked-out branch is an ancestor of the other, and a merge commit with two parents otherwise. A conflict on the same lines is resolved by hand and committed (Definition 2.2, Figure 3).
- `git fetch` moves `origin/main` only, `git pull` fetches and merges, and the name recorded by `git config` is a label without any role in authentication (Definition 3.1).
- `.gitignore` hides untracked files only, a committed credential stays in the history until it is revoked, and a tag reaches the remote only when pushed by name (Definition 3.2, Section 4.2).

Course environment

The chapter uses the command line. The source-control view of VS Code runs the same commands behind buttons. GitHub Desktop, used in the lecture, exists for macOS and 64-bit Windows only [38]. Course repositories live on GitHub and on the University's GitLab.

Exercises

Solutions are given from page 20.

Two edits to one file

In a clone of `attractor`, `MIN_GAP` is changed from 3.0 to 3.5 in `analyse.py`, the file is staged with `git add`, and the constant is then changed again to 4.0 without a second `git add`. (a) What does `git status --short` print? (b) If `git commit -m "Widen the peak gap"` is run now, which value of `MIN_GAP` does the commit contain, and which value is on disk afterwards? (c) Which single command brings the working tree back to the committed version?

Parents of a merge commit

The history of the demo clone at the end of Section 4.2 contains the commits of Listings 10 and 13. (a) How many parents has the commit that ended the fast-forward merge of Listing 10, and how many has the merge commit ea527b4? (b) Which option of `git log` prints the parents of each commit? (c) In the raw commit object, printed by `git cat-file -p`, which lines hold the parents?

A committed credential

A file `.env` holding `API_KEY=sk-demo-1234` is committed by mistake with `git add -f` and pushed. The author then runs `git rm --cached .env`, commits “Remove .env from the repository” and pushes again. (a) Is the key still readable from the repository, and with which command? (b) What must be done about the key? (c) Which command lists every commit that touched `.env`?

Solutions to the exercises

Two edits to one file

(a) `MM analyse.py`: the index differs from the last commit and the working tree differs from the index (Table 1). (b) The commit contains the staged version, 3.5, because `git commit` records the index. The file on disk still holds 4.0, and `git status --short` prints `M` after the commit. (c) `git restore analyse.py` copies the index, which now agrees with the commit, back to the working tree.

Answer check

```
$ git status --short
MM analyse.py
$ git commit -m "Widen the peak gap"
[exercise-one e758ca3] Widen the peak gap
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git show HEAD:analyse.py | grep -n "^MIN_GAP"
19:MIN_GAP = 3.5  # minimum spacing between two peaks, in days
$ grep -n "^MIN_GAP" analyse.py
19:MIN_GAP = 4.0  # minimum spacing between two peaks, in days
$ git status --short
 M analyse.py
$ git restore analyse.py
```

The check was run on a branch `exercise-one` of the demo clone, which was deleted afterwards with `git branch -D`.

Parents of a merge commit

(a) A fast-forward creates no commit. The commit 4aa4a17 that `main` moved to is an ordinary commit with one parent, 02a6234. The merge commit ea527b4 has two parents, a5aea42 and fa8e711, the tips of the two branches. (b) `--pretty="%h %p %s"` prints the abbreviated identifier, the parents and the subject of each commit. (c) The lines that begin with `parent`, one per parent, after the `tree` line that names the snapshot.

Answer check

```
$ git log --pretty="%h %p %s" -5
577cdfc da4dd83 Revert "Test peak spacing"
da4dd83 5537177 Stop tracking generated results
5537177 ea527b4 Add results table
ea527b4 a5aea42 fa8e711 Resolve step-size conflict
a5aea42 4aa4a17 Test smaller step
$ git cat-file -p ea527b4
tree 49751677ccf8cfc8fd2a8d657afc676c7cf108a6
parent a5aea422c462ae5a16829402cbda3e4f1a217334
parent fa8e7111178024ae3e400d0ca3eff222bbf4d21f
author Ada Lovelace <ada@example.org> 1792534004 +0100
committer Ada Lovelace <ada@example.org> 1792534004 +0100

Resolve step-size conflict
```

A committed credential

(a) Yes. Every commit keeps its snapshot, so `git show HEAD~1:.env` prints the file from the commit before the removal, in every clone and on the hosting service. (b) The key must be revoked and a new one issued. Removing the file and adding `.env` to `.gitignore` prevents the next mistake but does not repair this one. (c) `git log --all -- .env` lists the commits that added, changed or removed the path on any branch.

Answer check

```
$ git add -f .env
$ git commit -m "Add local config"
[secret-demo 9f8c804] Add local config
 1 file changed, 1 insertion(+)
 create mode 100644 .env
$ git rm --cached .env
rm '.env'
$ git commit -m "Remove .env from the repository"
[secret-demo d4ee6fd] Remove .env from the repository
 1 file changed, 1 deletion(-)
 delete mode 100644 .env
$ git show HEAD~1:.env
API_KEY=sk-demo-1234
$ git log --oneline --all -- .env
d4ee6fd Remove .env from the repository
9f8c804 Add local config
```

The check was run on a branch `secret-demo` of the demo clone, never pushed, and the branch was deleted afterwards.

References

[1] Boris Bolliet. `attractor`: cycle periods of the chemostat runs. GitHub, https://github.com/borisbolliet/attractor. accessed 20 October 2026.

[2] Bernd Blasius, Lars Rudolf, Guntram Weithoff, Ursula Gaedke, and Gregor F. Fussmann. Long-term cyclic persistence in an experimental predator–prey system. *Nature*, 577:226–230, 2020. Data: https://doi.org/10.6084/m9.figshare.10045976, CC BY 4.0.

[3] Scott Chacon and Ben Straub. What is Git? Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F. accessed 20 October 2026.

[4] Scott Chacon and Ben Straub. A short history of Git. Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git. accessed 20 October 2026.

[5] Linus Torvalds. Wikipedia, https://en.wikipedia.org/wiki/Linus_Torvalds. accessed 20 October 2026.

[6] Git developers. `git-log`. Git reference manual, https://git-scm.com/docs/git-log. accessed 20 October 2026.

[7] NumPy developers. NumPy 2.0 migration guide. NumPy documentation, https://numpy.org/doc/stable/numpy_2_0_migration_guide.html. accessed 20 October 2026.

[8] Git developers. `git-config`. Git reference manual, https://git-scm.com/docs/git-config. accessed 20 October 2026.

[9] Scott Chacon and Ben Straub. Recording changes to the repository. Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Git-Basics-Recording-Changes-to-the-Repository. accessed 20 October 2026.

[10] Git developers. `git-status`. Git reference manual, https://git-scm.com/docs/git-status. accessed 20 October 2026.

[11] Git developers. `git-diff`. Git reference manual, https://git-scm.com/docs/git-diff. accessed 20 October 2026.

[12] Scott Chacon and Ben Straub. Branches in a nutshell. Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell. accessed 20 October 2026.

[13] Git developers. `git-switch`. Git reference manual, https://git-scm.com/docs/git-switch. accessed 20 October 2026.

[14] Git developers. `git-branch`. Git reference manual, https://git-scm.com/docs/git-branch. accessed 20 October 2026.

[15] Git developers. `git-reflog`. Git reference manual, https://git-scm.com/docs/git-reflog. accessed 20 October 2026.

[16] Scott Chacon and Ben Straub. Basic branching and merging. Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Git-Branching-Basic-Branching-and-Merging. accessed 20 October 2026.

[17] Git developers. `git-merge`. Git reference manual, https://git-scm.com/docs/git-merge. accessed 20 October 2026.

[18] Git developers. `git-remote`. Git reference manual, https://git-scm.com/docs/git-remote. accessed 20 October 2026.

[19] Git developers. `git-fetch`. Git reference manual, https://git-scm.com/docs/git-fetch. accessed 20 October 2026.

[20] Git developers. `git-pull`. Git reference manual, https://git-scm.com/docs/git-pull. accessed 20 October 2026.

[21] Git developers. `git-push`. Git reference manual, https://git-scm.com/docs/git-push. accessed 20 October 2026.

[22] GitHub. About authentication to GitHub. GitHub Docs, https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/about-authentication-to-github. accessed 20 October 2026.

[23] GitHub. Fork a repository. GitHub Docs, https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/working-with-forks/fork-a-repo. accessed 20 October 2026.

[24] GitHub. Collaborating with pull requests. GitHub Docs, https://docs.github.com/en/pull-requests/collaborating-with-pull-requests. accessed 20 October 2026.

[25] GitLab. Merge requests. GitLab Docs, https://docs.gitlab.com/user/project/merge_requests/. accessed 20 October 2026.

[26] Git developers. `gitignore`. Git reference manual, https://git-scm.com/docs/gitignore. accessed 20 October 2026.

[27] Git developers. `git-check-ignore`. Git reference manual, https://git-scm.com/docs/git-check-ignore. accessed 20 October 2026.

[28] Git developers. `git-rm`. Git reference manual, https://git-scm.com/docs/git-rm. accessed 20 October 2026.

[29] GitHub. About secret scanning. GitHub Docs, https://docs.github.com/en/code-security/secret-scanning/introduction/about-secret-scanning. accessed 20 October 2026.

[30] GitHub. About large files on GitHub. GitHub Docs, https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-large-files-on-github. accessed 20 October 2026.

[31] GitHub. About Git large file storage. GitHub Docs, https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-git-large-file-storage. accessed 20 October 2026.

[32] Git developers. `git-restore`. Git reference manual, https://git-scm.com/docs/git-restore. accessed 20 October 2026.

[33] Git developers. `git-revert`. Git reference manual, https://git-scm.com/docs/git-revert. accessed 20 October 2026.

[34] Git developers. `git-reset`. Git reference manual, https://git-scm.com/docs/git-reset. accessed 20 October 2026.

[35] Scott Chacon and Ben Straub. Tagging. Pro Git, 2nd edition, https://git-scm.com/book/en/v2/Git-Basics-Tagging. accessed 20 October 2026.

[36] Git developers. `git-tag`. Git reference manual, https://git-scm.com/docs/git-tag. accessed 20 October 2026.

[37] Git developers. `git-show`. Git reference manual, https://git-scm.com/docs/git-show. accessed 20 October 2026.

[38] GitHub. Supported operating systems for GitHub desktop. GitHub Docs, https://docs.github.com/en/desktop/overview/supported-operating-systems-for-github-desktop. accessed 20 October 2026.

Index

annotated tag, 16

branch, 7

clone, 2
commit, 2
commit identifier, 3
conflict, 8

fast-forward, 8
fork, 12

HEAD, 7
history, 3

ignored, 13
index, 3

merge commit, 8

pull request, 12

remote, 11
remote-tracking branch, 11
repository, 2

tracked, 13

untracked, 13

working tree, 3