Git Course

Flavio Copes

Updated August 3, 2026

Contents

Learn version control from the command line: repositories, commits, branches, merges, remotes, GitHub, and safe ways to recover from mistakes.

What you'll learn: Track a project with Git, develop work on a branch, share it through GitHub, and undo common mistakes without losing work.

Git fundamentals

Version control, repositories, snapshots, and configuration.

Why version control

See why a project needs a durable history instead of copies, backups, and editor undo.

You change a file, then realize yesterday's version was better.

An editor's undo history might help. A folder named `project-final-2` might help too. But neither one gives you a reliable history of the whole project.

A version control system records meaningful project states. In Git, each recorded state is a **commit**. You can inspect a commit, compare it with another one, or start new work from it.

This is different from a backup. A backup protects files from loss. Git explains how the project changed and why.

Git also lets people work on separate branches. Each branch can move forward without changing the others. You combine the work when it is ready.

The important word is *deliberate*. Git does not record every edit automatically. You choose which changes belong together, inspect them, and create a commit.

Try this thought exercise. Imagine you changed the homepage, upgraded a dependency, and added a debug log. Would you want one commit or three? My advice is to make separate commits when each change can be understood and reversed on its own.

What Git is

Understand Git as a distributed version control system that stores a complete local project history.

Git tracks a project inside a directory we call a **repository**.

The repository contains two kinds of state. Your visible files are the **working tree**. Git's private data lives in the `.git` directory.

Inside `.git`, Git stores file contents, directory trees, commits, and references. A branch is one of those references: a readable name that points to a commit.

This is why Git works without a network connection. Your local repository contains its own history. You can inspect commits, create branches, and record new work while offline.

Git is **distributed** because every normal clone gets a repository, not just a working copy. A server can disappear and your local history still exists.

Git and GitHub are different things. Git is the version control tool. GitHub hosts Git repositories and adds pull requests, issues, permissions, and other collaboration features.

Run this inside any Git project:

```
git status
```

The command reads local repository state. It does not contact GitHub.

As a quick exercise, disconnect from the network and run `git log --oneline`. Your project history is still there.

Install Git

Install the Git command-line tool and verify that your terminal can run it.

Git is a command-line program, so installing it means getting the `git` command working in your terminal. Install Git with the package recommended for your operating system. The official Git website at git-scm.com lists current installers and package-manager options.

On macOS, the quickest route is Apple's command line tools, which include Git:

```
xcode-select --install
```

If you use Homebrew, `brew install git` gives you a more recent version than the Apple one. On Windows, download the Git for Windows installer from the official site; it also provides Git Bash, a terminal where the commands in this course work as written. On Linux, use your distribution's package manager, for example `sudo apt install git` on Debian and Ubuntu.

Whichever route you took, check the command from the terminal:

```
git --version
```

You should see output similar to:

```
git version 2.50.1
```

The exact number will change. What matters is that your shell finds one Git installation. Version differences rarely matter for this course; anything from recent years covers everything we use.

If you see `command not found`, close and reopen the terminal first. An installer may have changed your shell's command path, and open terminals keep the old one. If the problem remains, check the installation instructions for your operating system instead of installing a second copy at random. Two competing installations cause the confusing situation where the version you update is not the version your shell runs.

You can also find which executable your shell selected:

```
command -v git
```

This prints a path such as `/usr/bin/git` or `/opt/homebrew/bin/git`, telling you exactly which

installation answers the `git` command.

Exercise: record the version and executable path. This information is useful when two computers behave differently.

Configure your Git identity

Set the name and email Git records inside new commits.

Git records an author name and email inside every commit. This identifies the author in project history. It is not a login: Git never checks these values against any account, it just writes them into each commit you create.

Why does this come before your first commit? Because Git refuses to commit without an identity. On a fresh installation, `git commit` stops with *"Please tell me who you are"* and prints the exact commands to fix it.

Set them once for your user account:

```
git config --global user.name "Flavio Copes"
git config --global user.email "flavio@example.com"
```

Use your own name and email. The `--global` flag writes to a configuration file in your home directory, `~/.gitconfig`, so the values apply to every repository you work on as this user. Then verify the stored values:

```
git config --global --get user.name
git config --global --get user.email
```

Each command should print back exactly what you set. Pick the email deliberately: hosting platforms such as GitHub use it to connect commits to your profile, so an email your account does not know about produces commits that look like a stranger wrote them.

A project can override the global value with local configuration. Say your employer wants work commits under a work address. Run this inside that repository, without `--global`:

```
git config user.email "flavio@work.example"
```

The value lands in that repository's `.git/config` file and wins over the global one there. Run the same command with `--get` but without `--global` to inspect the value Git will use in that repository.

Configuration changes affect future commits. They do not rewrite authors stored in existing commits, so fix a wrong identity as soon as you notice it.

Exercise: create a disposable repository and run `git config --show-origin --get user.email`. Notice that Git shows both the value and the configuration file that supplied it. That one command settles every "which email will this commit use?" doubt.

Initialize a repository

Turn an existing project folder into a Git repository with git init.

Move into an existing project folder, then run:

```
git init
```

Git creates a hidden `.git` directory. This turns the current folder into a repository, but it does not

commit any files.

Check the state:

```
git status
```

Git will show the current branch and any untracked files. Your working files have not moved. Git is now ready to track selected versions of them.

The `.git` directory holds the object database, references, configuration, and staging information. Deleting it removes the repository metadata and local history while leaving the working files behind.

Do not edit files inside `.git` by hand.

Running `git init` again in the same folder is normally safe. Git reinitializes the repository instead of erasing its history. Still, always check `pwd` and `git status` first. Initializing the parent folder by mistake can make Git see far more files than you intended.

Exercise: initialize an empty practice folder. Run `git status`, create `notes.txt`, then run `git status` again. Explain why the file is visible but not yet part of a commit.

Working tree, staging area, and repository

Build the three-part mental model behind everyday Git commands.

Most everyday Git commands make sense when you picture three project states.

The **working tree** contains the files you see and edit.

The **staging area**, also called the **index**, holds the exact snapshot prepared for the next commit.

The **repository** stores completed commits. `HEAD` normally points to your current branch, and that branch points to its latest commit.

Suppose `notes.txt` matches the current commit. You edit it, then run:

```
git status
git diff
```

The working tree changed. The staging area and current commit did not.

Now stage the file:

```
git add notes.txt
git status
git diff --staged
```

`git add` copies the current file content into the index. It does not freeze the working file. If you edit `notes.txt` again, Git can show one staged version and another unstaged change at the same time.

Finally, commit the index:

```
git commit -m "Add project notes"
git status
```

Git creates a commit from the staged snapshot. The current branch reference moves to that commit. If the working tree still matches it, `git status` reports a clean state.

My advice is to ask this before every command: *Which state will this command read, and which*

state will it change?

Exercise: stage a file, edit it again, and inspect both `git diff` and `git diff --staged`. Predict which version a commit would contain before you run it.

Check your understanding: Git fundamentals

Check repositories, snapshots, staging, commits, ignored files, diffs, history, and distributed version control.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Your daily Git workflow

Status, staging, commits, diffs, history, and ignored files.

Check the repository status

Use git status before and after every important operation to see what Git knows about your files.

`git status` is the command that keeps you oriented.

Run it before and after an operation:

```
git status
```

Git compares three states: `HEAD`, the staging area, and the working tree.

It reports changes in useful groups:

- **Changes to be committed** are in the staging area.
- **Changes not staged for commit** exist only in the working tree.
- **Untracked files** are in the working tree but not in `HEAD` or the index.

Status also shows the current branch. When a branch tracks a remote branch, it can show whether your local reference is ahead or behind the last fetched remote-tracking reference.

Notice that `git status` does not show every changed line. Use `git diff` for that.

My advice is to run `git status` often. It is the safest way to stay oriented.

You can use the shorter format once you understand the full one:

```
git status --short
```

Exercise: create one untracked file, modify one tracked file, and stage a third file. Run both status formats and identify where each change currently lives.

Stage changes

Select the files and changes that should be part of the next commit.

The staging area lets you decide exactly what the next commit will contain.

Stage one file with:

```
git add README.md
```

Staging is not saving the file. Your editor already did that. `git add` copies the file's current content into the index.

Verify the staged change:

```
git status
git diff --staged
```

If you edit `README.md` again, the new edit remains in the working tree. The index still contains the version you staged earlier. Run both commands:

```
git diff
git diff --staged
```

This is useful when one file contains two unrelated changes. You can stage only selected parts with `git add -p`, review each hunk, and leave the rest for another commit.

Avoid `git add .` until you have checked `git status`. It is easy to include generated files or unrelated edits.

Exercise: make two edits to one file. Stage it after the first edit, make the second edit, then predict which content the next commit would record.

Create a commit

Record one coherent staged change with a short message that explains its purpose.

A commit records the current staging area. Unstaged and untracked changes stay outside it.

Before committing, inspect the snapshot:

```
git status
git diff --staged
```

Then create the commit:

```
git commit -m "Add project instructions"
```

A commit stores a project snapshot, parent commit, author, committer, timestamps, and message. Git gives it an object ID based on that content.

Your current branch reference moves from the parent to the new commit. `HEAD` still points to the same branch.

Verify the result:

```
git status
git log -1 --stat
```

If `git status` still shows changes, they were not in the staged snapshot. This is not an error. It means your working tree contains more work.

Keep each commit focused. It should be easy to explain and safe to review.

Exercise: stage one of two modified files and commit it. Confirm that the other file remains modified after the branch moves to the new commit.

Write useful commit messages

Describe the result of a commit so future readers can understand the history quickly.

A commit message should help someone understand the purpose without opening the diff first.

Use a short command-style subject:

```
Add password reset form
```

This reads naturally after the phrase: "If applied, this commit will…"

Avoid messages such as `changes`, `stuff`, or `fix`. The diff shows the changed lines. The message should name the result or intent.

For a change that needs more context, let Git open your editor:

```
git commit
```

Write a concise subject, leave a blank line, then explain why the change exists and any decision that is not obvious from the code.

Before committing, inspect `git diff --staged`. A good message describes that exact snapshot, not all the work you did during the day.

Exercise: compare these messages: `Update files`, `Fix login`, and `Reject expired password reset links`. Which one will help you most six months from now, and why?

Inspect differences

Compare working and staged changes before recording them.

`git diff` compares project states. The command you choose decides which states Git compares.

See working-tree changes that are not staged:

```
git diff
```

This compares the working tree with the index.

See the snapshot prepared for the next commit:

```
git diff --staged
```

This compares the index with `HEAD`.

To inspect everything changed since the current commit, including staged and unstaged work, use:

```
git diff HEAD
```

Untracked files do not appear in a normal diff because Git has no recorded version to compare. Find them with `git status`.

Review both ordinary and staged diffs before committing. This catches debug output, accidental edits, and changes staged from an older version of a file.

Exercise: stage one edit, make another edit in the same file, then run all three commands. Explain why each diff is different.

Read the project history

Use git log to inspect commits and understand how the project reached its current state.

The commit history tells you which snapshots exist and how they connect.

Start with a compact view:

```
git log --oneline
```

Each line shows an abbreviated commit ID and its subject. The newest reachable commit appears first.

To see branch and tag names beside their commits, add decorations:

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

`--all` asks Git to start from all local references, not only `HEAD`. `--graph` draws the parent relationships. It does not change history.

Inspect one commit in detail with:

```
git show --stat <commit-id>
```

Then use `git show <commit-id>` to inspect its patch.

History is read-only, so these are safe commands when you feel lost. If a change seems to have disappeared after switching or resetting, inspect `git log --all` before trying another modifying command.

Exercise: find the commit that introduced one file. Read its message, parent, and patch. Can you explain the project state before and after it?

Ignore generated and private files

Use .gitignore for files that should not enter the repository.

Some files belong in your working tree but not in project history. Examples include generated output, local settings, dependencies, and secret files.

Create a `.gitignore` file and list patterns:

```
node_modules/
.env
dist/
```

Check the result:

```
git status
```

Matching untracked files should disappear from the status output.

Ignore rules affect untracked files. If `.env` is already tracked, adding it to `.gitignore` does not remove it from the index or earlier commits.

You can inspect which rule matches a path:

```
git check-ignore -v .env
```

Be careful with secrets. Ignoring a file prevents a future accidental `git add`, but it cannot revoke a credential already committed. Rotate an exposed secret first, then clean up the repository history if necessary.

Commit the `.gitignore` file when the rules apply to everyone on the project. Use repository-local excludes for personal files that should not become shared policy.

Exercise: ignore `notes.local`, verify the matching rule, then confirm that `git add notes.local` refuses to stage it unless you explicitly force the operation.

Practice with a small repository

Build a short history that includes new, modified, staged, ignored, and committed files.

Create a disposable `notes-project` folder with a README and two text files.

Initialize Git. Before each command, predict the working-tree, index, and `HEAD` state. Then verify with `git status` and the appropriate diff.

Build this history:

1. Add the README in the first commit.
2. Add one notes file in the second commit.
3. Improve both files, but stage and commit only one.
4. Add `notes.local` to `.gitignore` and commit the rule.
5. Finish the remaining change in a final commit.

After every commit, run:

```
git status
git log --oneline --decorate
```

At least once, stage a file and edit it again. Use `git diff` and `git diff --staged` to explain which version the next commit will contain.

Do not rush. The goal is not to memorize commands. The goal is to predict which state each command reads and changes.

When you finish, draw the commits as boxes connected by parent arrows. Mark the commit your current branch points to and where `HEAD` points.

Branches and merges

Create branches, combine work, and resolve conflicts.

What a branch is

Understand a branch as a movable name that points to the latest commit in one line of development.

A branch is a readable name that points to a commit.

Suppose `main` points to commit `C`:

```
A---B---C  main
```

Creating a branch named `add-search` creates another reference to the same commit:

```
A---B---C  main, add-search
```

No files are copied. A branch is lightweight because Git only needs another reference.

`HEAD` normally points to the branch you currently have checked out. If `HEAD` points to `add-search` and you commit, Git creates a child of `C` and moves only `add-search`:

```
A---B---C  main
```

```
        \
         D  add-search <- HEAD
```

The `main` reference remains at `C`. The commits are not "inside" folders called branches. References give Git starting points for walking the commit graph.

Inspect the current branch with:

```
git branch --show-current
git log --oneline --decorate --graph --all
```

Exercise: draw three commits and two branch references. Move `HEAD` to one branch, add a commit, and predict which reference moves.

Create and switch branches

Start a feature branch and move your working tree to it.

Before creating a branch, inspect your current state:

```
git status
git branch --show-current
```

Create a branch at the current commit and switch to it:

```
git switch -c add-search
```

Git creates the `add-search` reference, then points `HEAD` to it. The working tree still represents the same commit, so the files might not visibly change yet.

Verify the result:

```
git branch --show-current
git log -1 --oneline --decorate
```

Now create a commit. Only `add-search` moves to the new commit.

Later, return to the main branch:

```
git switch main
```

Switching updates the working tree and index to match the destination commit. Git refuses when that update would overwrite conflicting local changes.

Do not solve that refusal by discarding work. Commit the change if it is coherent, or store it deliberately before switching.

Exercise: create a branch, commit a new file, switch back to `main`, and confirm the file disappears from the working tree but remains in the branch history.

Merge a branch

Bring a completed branch into the current branch.

A merge brings another line of history into the current branch.

First, make sure the working tree is clean and switch to the branch that should receive the work:

```
git status
git switch main
```

```
git merge add-search
```

The direction matters. This command merges `add-search` **into** `main` because `main` is the current branch.

Git finds the common ancestor and compares both histories. If `main` has not moved, Git can fast-forward its reference. If both branches have new commits, Git usually creates a merge commit after combining compatible changes.

Verify the result:

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

After a successful merge, `main` reaches the feature commits. The `add-search` reference still exists until you delete it.

If Git reports a conflict, stop and inspect `git status`. You can resolve the files and continue, or return to the pre-merge state with `git merge --abort` before making unrelated changes.

Exercise: merge a one-commit branch into an unchanged `main`. Draw the graph before and after, and explain why no merge commit was necessary.

Fast-forward and merge commits

Recognize the two common results of merging a branch.

When you merge a branch, Git produces one of two results. Which one you get depends on a single question: did the receiving branch move since the other branch started? Once you can read both shapes in the history, merge output stops being mysterious.

Suppose `add-search` started at commit `B` and added commit `C`:

```
A---B  main
     \
      C  add-search
```

If `main` did not move, merging can **fast-forward** it. Git only moves the `main` reference to `C`:

```
A---B---C  main, add-search
```

No merge commit is needed because `C` already contains `B` in its history. Nothing new is created; no files are combined. Git checks that `B` is an ancestor of `C` and slides the branch pointer forward. The message `Fast-forward` in the merge output tells you this is what happened.

Now imagine both branches moved:

```
A---B---D  main
     \
      C  add-search
```

Neither tip is an ancestor of the other, so a pointer move cannot represent both lines of work. A normal merge combines both histories in a new commit with two parents:

```
A---B---D---M  main
     \     /
      C---'  add-search
```

The merge commit `M` records that the histories joined. It does not mean a conflict occurred. Compat-

ible changes merge automatically, and the output says something like `Merge made by the 'ort' strategy`.

You can also state your expectation up front. `git merge --ff-only add-search` refuses to run unless a fast-forward is possible, which is useful when you want history to stay a straight line. `git merge --no-ff add-search` forces a merge commit even when fast-forwarding was possible, so the branch remains visible as a grouped unit in history.

Use this read-only view to see which result happened:

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

A straight line means fast-forward. A diamond with a two-parent commit means a real merge.

Exercise: draw a graph where fast-forwarding is possible, then add one commit to `main`. Explain why the same merge now needs a merge commit.

Resolve a merge conflict

Read conflict markers, choose the correct content, and complete a merge deliberately.

A conflict means Git cannot choose a correct result automatically. It is not repository damage. It is a decision Git needs you to make.

Start with:

```
git status
```

Git lists the unmerged paths. Open one and look for markers like these:

```
<<<<<<< HEAD
Search our notes
=======
Find saved notes
>>>>>>> add-search
```

The first section came from the current branch. The second came from the branch being merged. Read the surrounding code and write the final content you actually want. Do not keep the marker lines.

Then stage the resolved file:

```
git add README.md
git status
git diff --staged
```

Staging tells Git that this path is resolved. When every conflict is resolved, complete the merge:

```
git commit
```

If you are not ready to resolve the merge, abort it before doing unrelated work:

```
git merge --abort
```

This attempts to restore the pre-merge state. Existing uncommitted changes can make restoration harder, which is why a clean working tree matters before merging.

Exercise: create the same line differently on two branches, merge them, resolve the conflict, and inspect the resulting merge commit with `git show --stat`.

Git, detached HEAD

Understand the Git detached HEAD state, when HEAD points to a commit instead of a branch, and how to create a branch so you do not lose new commits.

One of the states in which your Git repository can end up is "detached HEAD".

Sounds scary.

Two words: detached (not attached?) and `HEAD`.

`HEAD` is the current commit.

When you create a repository, HEAD points to the first commit. If you do a second commit, HEAD points to the new commit.

If you switch branch, HEAD points to the latest commit in that branch.

That's what's normal 90% of the time: `HEAD` points to the latest commit in the currently selected branch. To Git, that's what it means `HEAD` being *attached*.

Our problem is `HEAD` being *detached*.

This happens when you checkout a commit that's not the latest in the branch, and basically means `HEAD` is not pointing to a branch, but at a commit.

It's not a situation that's too unusual. Perhaps you are debugging an issue and you're trying to figure out in which commit the issue was introduced, so you check out individual commits (using `git checkout <commit>`) until you find where the code was working.

In this case, when you're done, you can checkout a branch, for example `main`, using `git checkout main`

One thing you can do however is end up in this situation: you are in a detached HEAD state (checked out a commit) and you create a new commit. One or more.

In this case, you need to create a branch before you switch to another branch, otherwise your changes might be lost.

Do so by creating a branch at this commit using:

```
git branch <new-branch-name>
```

and then switching to that branch (important):

```
git checkout <new-branch-name>
```

Or, with a single command:

```
git checkout -b <new-branch-name>
```

If you're in a Git situation that's messier than this, I built a free Git recovery tool: describe the mess, and it gives you the recovery steps.

Delete a finished branch

Remove a local branch name after its useful commits have been merged.

After merging a feature, verify that the current branch reaches its commits:

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

Then delete the local branch name:

```
git branch -d add-search
```

The lowercase `-d` is deliberately cautious. Git refuses to delete the branch when its tip is not merged into an appropriate upstream or `HEAD`.

When the branch was merged, its commits remain reachable through `main`. You are deleting one reference, not the completed work.

Do not reach for uppercase `-D` just because deletion was refused. It forces the reference deletion and can leave commits without an ordinary branch name. Inspect the graph and decide whether the work is truly disposable.

After deletion, check the remaining references:

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

Exercise: create an unmerged practice branch and try `git branch -d`. Read the refusal, merge the branch, then repeat the safe deletion.

Check your understanding: branches and merges

Check HEAD, branch creation, fast-forward merges, merge commits, conflicts, detached HEAD, updates, and branch deletion.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Undoing and recovery

Unstage, restore, amend, revert, and recover safely.

Unstage a file

Remove a file from the next commit without discarding its edits.

Sometimes the index contains a file you do not want in the next commit. Maybe you ran `git add .` and it swept in a file that belongs to a different change, or you staged an experiment you are not ready to record. Unstaging changes the index, not the working file, so your edits are never at risk.

Inspect the current state first:

```
git status
git diff --staged -- README.md
```

`git status` lists the file under *"Changes to be committed"*, and the staged diff shows exactly what would enter the commit. Notice that `git status` already prints the answer in its hint line: *"use git restore --staged ... to unstage"*.

Then unstage the file:

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

By default, Git restores the index entry from `HEAD`, the current commit. In plain words: the staging

area's copy of `README.md` goes back to matching the last commit, as if you had never run `git add` on it. Your edited `README.md` stays in the working tree, untouched.

Verify both states:

```
git status
git diff -- README.md
git diff --staged -- README.md
```

The ordinary diff now shows the edit. The staged diff no longer does. The file moved from *"Changes to be committed"* to *"Changes not staged for commit"*.

This is a recoverable action because the working content remains available. It is different from `git restore README.md`, which targets the working tree and can overwrite unstaged content permanently. The `--staged` flag is the whole difference between a safe command and a destructive one, so read twice before pressing Enter.

You may also see `git reset HEAD README.md` in older tutorials. It has the same effect; `git restore --staged` is the clearer modern spelling of the same operation.

Exercise: stage two files, unstage one, and predict exactly which file the next commit will contain. Then run `git commit` and check the result with `git show --stat`.

Discard working changes carefully

Restore a tracked file from the current commit when you truly do not need its uncommitted edits.

Discarding an uncommitted edit is destructive. Git cannot recover content that never reached the index or a commit.

First, inspect the file in both places:

```
git status
git diff -- config.js
git diff --staged -- config.js
```

If the staged diff is empty, the index matches `HEAD`. Restoring the working file will then bring it back to the current commit:

```
git restore config.js
```

By default, `git restore` copies from the index into the working tree. That is why checking the staged diff matters.

If you want to state the source explicitly, use:

```
git restore --source=HEAD --worktree config.js
```

Both commands overwrite working-tree edits in this scenario. If you might need the content, copy the file elsewhere or commit it on a temporary branch first.

Run `git status` and `git diff -- config.js` afterward. The working-tree change should be gone.

Exercise: make a disposable edit, inspect it, save a copy outside the repository, then restore the tracked file. Compare the restored file with your copy.

Git, what if you forgot to add a file to a commit?

Forgot to add a file to your last Git commit? Learn how to use git commit --amend to stage the missing file and optionally fix the commit message.

This happens often. You create a commit, then notice that one file stayed in the working tree.

Start by checking exactly what the last commit contains and what remains outside it:

```
git status
git show --stat --oneline HEAD
git diff -- file-forgotten.txt
```

If the missing file belongs to that same change, stage it:

```
git add file-forgotten.txt
git diff --staged
```

Then replace the last commit while keeping its message:

```
git commit --amend --no-edit
```

Amending does not open and modify the old commit. Git creates a new commit from the current index and gives it a new object ID. The current branch moves to the replacement.

Verify the result:

```
git show --stat --oneline HEAD
git status
```

Only amend work that is still private to your local branch. If other people may already have the old commit, create a new commit instead. That avoids rewriting shared history.

Exercise: create a commit that intentionally misses one file, record its ID, amend it, then compare the new ID and file list.

For this and other Git mistakes, I made a free Git recovery tool that gives you step-by-step commands to get out of trouble.

Amend the last local commit

Replace the latest commit when its message or selected files need a small correction.

`git commit --amend` replaces the latest commit on the current branch.

Before using it, inspect both the latest commit and the index:

```
git show --stat --oneline HEAD
git diff --staged
```

To change only the commit message, run:

```
git commit --amend
```

To include a correction, stage it first, then amend. The new commit uses the complete current index, not only the last file you staged.

```
git add README.md
git diff --staged
git commit --amend --no-edit
```

Amending creates a new commit with a new ID. The parent normally stays the same, but the message, snapshot, author metadata, or committer metadata can differ.

Use amend on local work. Avoid replacing a commit that other people may already have. Their history still contains the old ID, so a later push can require history rewriting and create unnecessary recovery work.

If the commit is already shared, make a normal follow-up commit instead. It is less elegant, but it preserves the common history.

Exercise: amend a disposable commit, then run `git log --oneline -2`. Explain why the replacement appears as one commit rather than an extra child.

Revert a shared commit

Undo a published change by adding a new commit instead of rewriting shared history.

When a bad commit is already shared, preserve history and add a new commit that reverses its changes.

Start with a clean working tree. Inspect the target commit before changing anything:

```
git status
git show --stat 4f72a1c
```

Then revert it:

```
git revert 4f72a1c
```

Git applies the inverse patch and opens a commit message. The original commit remains in history, and the current branch moves to a new revert commit.

Verify both commits:

```
git log -2 --oneline
git show --stat HEAD
```

A revert can conflict when later work changed the same lines. Resolve it like a merge conflict, stage the result, then run `git revert --continue`. Use `git revert --abort` to return to the pre-revert state.

Revert is safer for shared history because collaborators can fetch a normal new commit. It does not require everyone to replace an existing commit ID.

Exercise: commit a line, commit another unrelated line, then revert the first commit. Confirm that the unrelated later change remains.

Understand reset before using it

Know how reset moves a branch and why hard reset can permanently discard uncommitted work.

When you reset a branch to a commit, Git moves the branch reference. The mode decides what happens to the index and working tree.

Before any reset, inspect and protect the current state:

```
git status
git log --oneline --decorate -5
git branch backup-before-reset
```

The backup branch gives the current commit a name. This makes recovery much easier.

A soft reset moves the branch but leaves the index and working tree unchanged:

```
git reset --soft HEAD~1
```

The old commit's changes now appear staged.

The default **mixed** reset moves the branch and resets the index, but preserves working-tree files:

```
git reset HEAD~1
```

The changes now appear unstaged.

A hard reset moves the branch and makes both the index and tracked working files match the target:

```
git reset --hard HEAD~1
```

This can permanently discard uncommitted tracked changes. It is not a first response to a confusing repository.

Reset is best kept to local history. If a commit is shared, prefer `git revert` so the correction becomes a normal new commit.

Exercise: in a disposable repository, create a backup branch, run a soft reset, inspect `status`, then recover by switching back to the backup branch. Do not practice `--hard` with real work.

git reflog: how to recover lost commits

git reflog is your safety net for recovering lost commits, deleted branches, and bad resets. Learn how to read it and restore your work.

You can recover lost commits with `git reflog`, even after a bad `reset --hard` or a deleted branch. Git keeps a local log of where `HEAD` has been. That log is your safety net.

What the reflog records

The reflog stores every time `HEAD` moves in your local repo. A commit, a reset, a checkout, a rebase. Each entry gets a short hash and a message.

This log lives only on your machine. It is not pushed to the remote. Your teammates cannot see your reflog.

Run this to see the last moves:

```
git reflog
```

Example output:

```
a1b2c3d HEAD@{0}: reset: moving to HEAD~3
e4f5a6b HEAD@{1}: commit: add payment form
c7d8e9f HEAD@{2}: commit: fix checkout bug
```

`HEAD@{0}` is the most recent move. `HEAD@{1}` is one step back. The hash on the left is the commit you can recover.

Recover after a bad reset --hard

Say you ran `git reset --hard HEAD~3` and lost three commits. The commits still exist. Git just moved `HEAD` away from them.

Find the commit before the reset in the reflog:

```
git reflog
```

Look for the line right before `reset: moving to`. That hash is your work.

Reset back to it:

```
git reset --hard e4f5a6b
```

Your commits are back. If you only want one commit, cherry-pick it instead:

```
git cherry-pick e4f5a6b
```

Recover a deleted branch

You deleted a branch with `git branch -D feature/payments`. The commits are still in the reflog if you worked on that branch recently.

Check the reflog for that branch:

```
git reflog show feature/payments
```

Or search the main reflog for the last commit on that branch. Then recreate the branch:

```
git branch feature/payments e4f5a6b
```

You are back where you left off.

Cherry-pick or reset to a reflog entry

You have two main options once you find the right hash.

`git reset --hard <hash>` moves your current branch to that commit. Use this when you want the whole branch back.

`git cherry-pick <hash>` copies one commit onto your current branch. Use this when you only need specific work.

You can also checkout a detached state to inspect first:

```
git checkout e4f5a6b
```

Look around. When you are happy, create a branch or reset your main branch to that point.

Reflog expiry

Reflog entries expire. The default is 90 days for reachable commits. Unreachable commits may expire after 30 days.

After expiry, Git may garbage-collect those commits. Recovery gets harder or impossible. Do not treat reflog as a backup strategy. Push your work to the remote regularly.

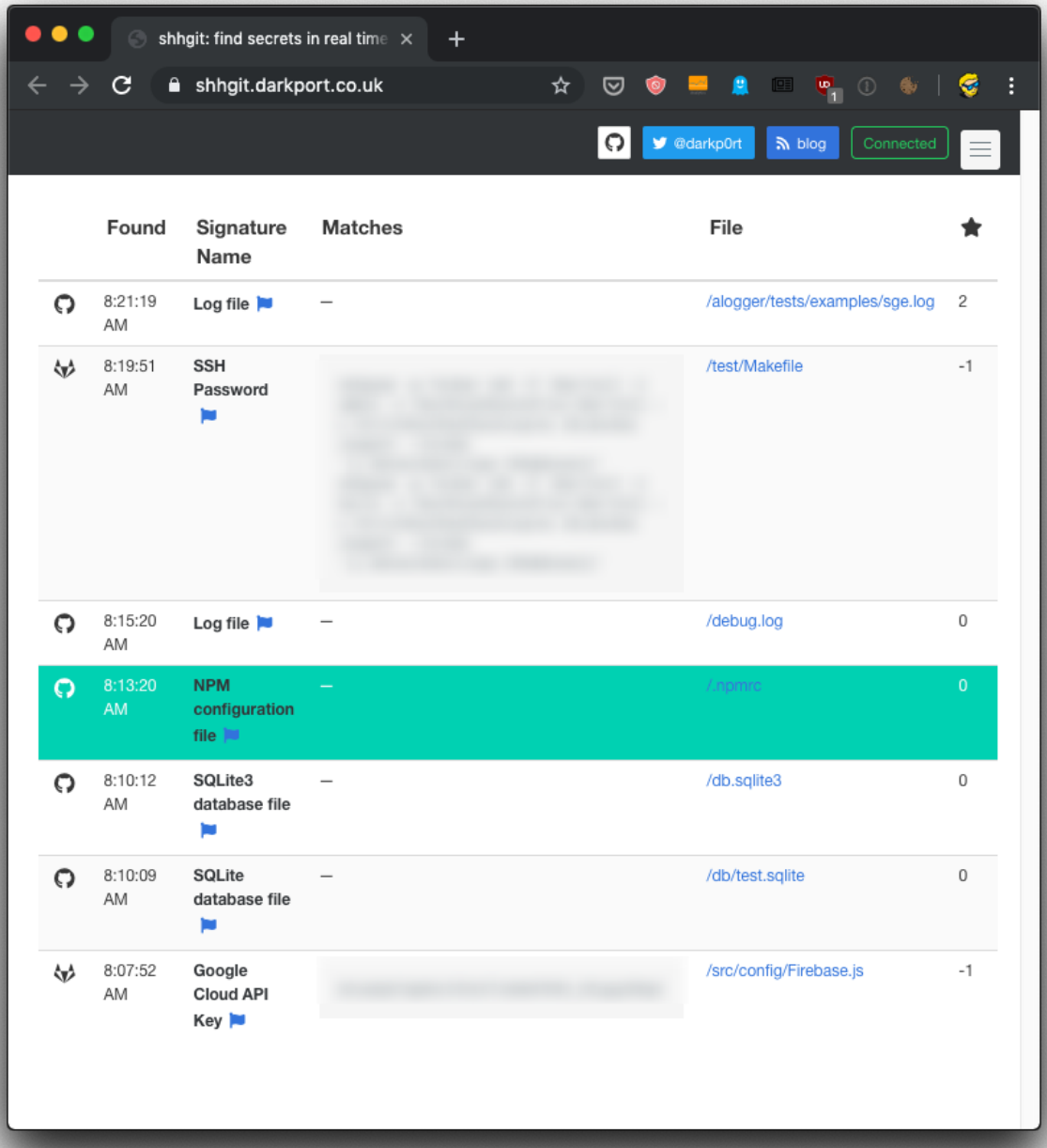
My advice: run `git reflog` the moment you think you lost something. The entry is almost always there if it happened today.

For more Git basics, see the Git guide. The Git cheat sheet is handy when you need a quick command reference.

I posted my password / API key on GitHub

Posted a password or API key to GitHub? Invalidate it now since rollback keeps it in history, then prevent leaks with .env files and the git-secrets hook.

I stumbled on a website that continuously scans GitHub, GitLab and BitBucket, the 3 most common places to host source code publicly, and shows you committed SSH passwords, API keys for common services, databases and so on.



It's scary, right?

Raise your hand if it never happened to you. We can make mistakes. And when this happens, there's no other way than quickly invalidating the password or API key that was exposed to the public.

For people new to Git: you can't just rollback the commit, because it will still be kept in the history

of the repository.

Your reputation, the reputation of your project, the security of your users is at stake.

After you fix the emergency, the issue is: how to prevent the problem? What's the answer? What's the solution that can help us avoid commit secrets to a publicly available Git repository?

The answer is: workflow and tooling.

First, never add your API keys or passwords inside source code. They can hide in there, quietly. Instead, always add them to a `.env` file in the project root folder, and add `.env` to your `.gitignore` file, so it will never be committed. Use a tool like dotenv to access them. If you need a solid `.gitignore` for your stack, I built a free gitignore generator that creates one for you.

Use `git-secrets`, a tool that will help you avoid committing secrets to Git.

In macOS you install it using Homebrew:

```
brew install git-secrets
```

then go inside the repository you want to activate it on, and run

```
git secrets --install
```

to install the Git pre-commit hook. This will ensure the tool runs before Git makes the commit to the repo.

If you use Amazon Web Services (AWS), run this command to add the set of patterns used by that services credentials:

```
git secrets --register-aws
```

You can immediately scan for issues using

```
git secrets --scan
```

Ideally the tool should not print anything. But if you have issues, it will give you plenty of details.

Check your understanding: undoing things

Check unstaging, restoring files, reverting shared changes, reset modes, reflog recovery, amending, and leaked-secret response.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Remotes and GitHub

Clone, fetch, pull, push, forks, and pull requests.

What a remote is

Understand a remote as a named connection to another Git repository.

A **remote** is a named set of URLs for another Git repository.

Inspect the configured remotes with:

```
git remote -v
```

You might see a name such as `origin` beside fetch and push URLs. `origin` is a convention created by `git clone`; it is not a special server. A project can have no remotes, one remote, or several.

A remote name is configuration. Remote-tracking references are local names such as `origin/main` that record the remote branch state from your last network operation.

This distinction matters. `main` is a local branch that you can commit on. `origin/main` is Git's local record of the remote branch. It changes when you fetch or successfully push, not when someone changes the server while you are offline.

Local commits stay local until you push them. Remote commits do not appear locally until you fetch them.

Use these read-only commands to inspect both:

```
git remote get-url origin
git branch --all
git log --oneline --decorate --graph --all
```

Exercise: point to `main`, `origin/main`, and `origin` in the output. Explain which one is a branch, which one is a remote-tracking reference, and which one is configuration.

Clone a repository

Download a remote repository, its branches, and its history into a new local folder.

Cloning creates a new local repository from an existing one.

Run:

```
git clone git@github.com:flaviocopes/notes.git
```

Git creates a `notes` folder, copies reachable repository data, configures the source as `origin`, and checks out the remote's default branch as a local branch.

Move into the repository and inspect the state:

```
cd notes
git status
git remote -v
git branch --all
```

The working tree contains one checked-out branch. Other remote branches usually appear as remote-tracking references such as `origin/add-search`; Git does not create a local branch for every one.

A clone is independent. New local commits do not change the source repository until you push and have permission to do so.

If cloning stops midway, remove only the incomplete destination folder after confirming its exact path, then retry. Do not run `git init` inside the partial clone and hope it repairs missing objects.

Exercise: clone a small repository, disconnect from the network, and run `git log`. Explain why the history remains available.

How to add a Git remote

Learn how to add a Git remote to your repository with git remote add origin, pointing it at your GitHub repo so you can push your code there.

An existing local repository does not automatically know where you want to publish it.

First, check whether a remote already exists:

```
git remote -v
```

If there is no remote named `origin`, add one:

```
git remote add origin git@github.com:flaviocopes/myrepo.git
```

This stores the URL under the name `origin`. It does not contact GitHub and it does not push any commits.

Verify the configuration:

```
git remote get-url origin
git remote -v
```

If Git says `remote origin already exists`, stop and inspect its URL. Do not delete it blindly. If the repository moved and you intend to change the existing URL, use:

```
git remote set-url origin git@github.com:flaviocopes/myrepo.git
```

After adding a remote, `git status` still shows the same working tree and local branch. Network state changes only when you fetch or push.

Exercise: add a remote to a disposable repository, inspect `.git/config`, then use `git remote remove origin`. Confirm that removing the configuration does not delete local commits.

Fetch and pull changes

Choose whether to inspect remote work first or download and integrate it immediately.

`git fetch` downloads objects and updates remote-tracking references without changing your current branch, index, or working files.

Run:

```
git fetch origin
git status
```

Now inspect commits that the remote branch has and your branch does not:

```
git log --oneline HEAD..origin/main
```

You can review the incoming work before integrating it.

`git pull` performs a fetch, then integrates the selected remote branch into your current branch. The integration can merge, rebase, or require an explicit choice, depending on the command options and configuration.

If you only want a pull that can fast-forward, say so:

```
git pull --ff-only
```

Git stops instead of creating a merge commit when the histories diverged.

Before pulling, check `git status`. Local uncommitted work can conflict with files the integration needs to update. Preserve that work deliberately instead of discarding it to make the command pass.

Fetch first when you want to inspect the incoming work before combining it.

Exercise: fetch a repository with an updated remote branch. Compare `HEAD` with `origin/main`, then predict whether `git pull --ff-only` can succeed.

Push a branch

Publish local commits and connect a new local branch to its remote counterpart.

Before pushing, inspect the local commits and remote configuration:

```
git status
git log --oneline --decorate -3
git remote -v
```

Publish a new local branch with:

```
git push -u origin add-search
```

Git sends objects the remote needs and asks it to update its `add-search` branch. A successful push also updates your `origin/add-search` remote-tracking reference.

The `-u` option records the upstream relationship. Later, plain `git status`, `git push`, and `git pull` can relate the local branch to that remote branch.

Verify it:

```
git status
git branch -vv
```

A push can be rejected when the remote branch contains commits your local branch does not have. Do not immediately force-push. Fetch, inspect the graph, and integrate or coordinate the remote work first.

Pushing does not make uncommitted or merely staged changes available remotely. Only commits are transferred.

Exercise: modify a file without committing it, then push. Confirm that the remote branch does not receive the working-tree edit.

Authenticate with SSH

Use an SSH key so GitHub can identify your computer without asking for a password on every operation.

SSH authentication lets a Git host identify your computer through a key pair.

The pair contains a private key and a public key. The private key stays on your computer. You add the public key to the account on the Git host.

Create a modern key with a label that identifies it:

```
ssh-keygen -t ed25519 -C "flavio@example.com"
```

Follow the prompts and protect the private key with a passphrase. Your operating system or SSH

agent can remember the unlocked key for a session.

Files ending in `.pub` contain the public key. Inspect the names carefully before copying anything:

```
ls -la ~/.ssh
```

Add the public key content to GitHub, then test the SSH connection using GitHub's documented test command. A successful authentication does not grant access to every repository; repository permissions still apply.

Use an SSH remote URL:

```
git remote set-url origin git@github.com:flaviocopes/myrepo.git
git remote -v
```

The file ending in `.pub` is safe to copy to a service. Never share the private key.

If a key is exposed, remove its public counterpart from services and replace the pair. Do not commit keys to a repository.

Exercise: identify the public and private files in a disposable key pair without printing the private key content.

A developer's introduction to GitHub

A developer's introduction to GitHub: repositories, issues, branches, pull requests, Actions, Projects, releases, integrations, and account security.

GitHub is a platform for hosting Git repositories and collaborating on software.

Git does the version control work on your computer. GitHub stores a remote copy of the repository and adds tools for discussing, reviewing, automating, and publishing the work.

You do not need GitHub to use Git, but GitHub is often where a team shares its Git repository.

Repositories

A GitHub repository contains your files and their Git history. It can be public or private and can belong to a person or an organization.

A repository can also contain:

- a README that introduces the project
- issues for bugs and tasks
- pull requests for proposed changes
- discussions for questions and announcements
- releases for downloadable versions
- automated workflows

GitHub's repository documentation explains the terminology and visibility options.

To get a repository onto your computer, clone it:

```
git clone https://github.com/OWNER/REPOSITORY.git
```

If you use SSH with GitHub, use the SSH URL instead:

```
git clone git@github.com:OWNER/REPOSITORY.git
```

Branches and forks

A branch is a separate line of work inside a repository. You normally create a branch, make commits, push it to GitHub, and open a pull request.

```
git switch -c add-search
git add .
git commit -m "Add search"
git push -u origin add-search
```

A fork is a separate repository connected to an upstream repository. Forks are useful when you want to contribute to a project but do not have permission to create branches in its repository.

Team members with write access usually work from branches in the shared repository. External contributors often work from forks. Both approaches can lead to a pull request.

Issues

Issues track bugs, tasks, ideas, and user feedback.

An issue can have labels, assignees, milestones, sub-issues, and dependency relationships. A pull request can also close an issue automatically when its description includes a supported keyword such as:

```
Closes #42
```

Use an issue when the work needs discussion or tracking. Use a discussion when the conversation is more open-ended and does not yet represent a concrete task.

See GitHub's guide to issues.

Pull requests

A pull request proposes merging changes from one branch into another.

It shows the commits and file differences, and gives collaborators one place to:

- discuss the change
- comment on specific lines
- suggest edits
- approve the change or request changes
- inspect automated checks
- merge the branch

A pull request is not limited to forks. It can compare two branches in the same repository, which is the common workflow for a team with write access.

Draft pull requests are useful when you want early feedback but the change is not ready to merge. GitHub's pull request documentation covers both regular and draft pull requests.

GitHub Actions

GitHub Actions runs automated workflows in response to repository events.

A workflow is a YAML file in `.github/workflows/`. It can run tests when a pull request opens, build a site after a push, publish a package, or deploy a release.

Here is a small workflow that installs dependencies and runs tests:

```yaml
name: Test

on:
  pull_request:
  push:

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v6
      - uses: actions/setup-node@v6
        with:
          node-version: lts/*
          cache: npm
      - run: npm ci
      - run: npm test
```

Actions referenced with `uses:` execute code in your workflow. Review third-party actions before using them and pin them according to your project's security policy.

Start with GitHub's workflow documentation.

GitHub Projects

GitHub Projects helps you plan and track work across issues and pull requests.

A project can display the same work as a table, board, or roadmap. You can add custom fields, filters, charts, automation, and draft items. Projects can belong to a user or an organization and are not limited to a single repository.

See About Projects for the current feature set.

Releases

A Git tag marks a point in the repository history. A GitHub release adds a page around a tag with release notes and optional downloadable files.

Releases are useful for publishing application builds, command-line binaries, or source archives. You can create them in the web interface, with GitHub CLI, through the API, or from an automated workflow.

Read GitHub's release documentation.

Webhooks and GitHub Apps

Webhooks send an HTTP request to another service when an event happens, such as a push, issue update, or release.

Use a webhook when an external system only needs event notifications. Use a GitHub App when an integration needs to authenticate, request narrowly scoped permissions, and act on GitHub data.

GitHub recommends GitHub Apps over OAuth apps for many organization-wide integrations because their permissions can be more precise. See the documentation for webhooks and GitHub

Apps.

Keep your account secure

Enable two-factor authentication or use a passkey, and save your recovery codes somewhere safe.

For command-line Git access, use SSH or HTTPS with GitHub CLI, a credential manager, or a personal access token. GitHub no longer accepts your account password for Git operations over HTTPS.

Give tokens the smallest permissions they need, set an expiration when possible, and never commit tokens or private keys to a repository.

GitHub's authentication guide describes the supported methods.

Forks and pull requests

Use a fork to propose work across repository boundaries and a pull request to review a branch before merging.

A **fork** is a repository hosted under another GitHub account. It gives you a place to push branches when you cannot or should not push to the original repository.

Your local clone can keep both repositories as remotes:

```
git remote -v
```

A common convention is `origin` for your fork and `upstream` for the original repository. These names are conventions, so inspect the URLs instead of assuming.

A **pull request** asks maintainers to review and merge one branch into another. It is a GitHub collaboration feature, not a core Git command.

Before opening one, fetch the target repository, inspect the graph, and make sure your branch contains only the intended commits:

```
git fetch upstream
git log --oneline upstream/main..HEAD
git diff upstream/main...HEAD
```

The commit range shows work reachable from your branch but not from `upstream/main`. The three-dot diff compares your branch with their merge base.

Keep the pull request focused and explain what changed, why, and how you tested it.

Review comments may lead to more commits on the same branch. The pull request updates when you push them.

Exercise: inspect a branch against its intended base. Can every listed commit and changed file be explained by one purpose?

Publish a small project

Practice the complete fundamentals workflow from an empty folder to a reviewed GitHub branch.

Create a disposable project with a README and two content files. You will take it from an empty folder to a reviewed remote branch.

Build the local history first:

1. Initialize the repository and verify the current state.
2. Commit the README by itself.
3. Add one content file in a second focused commit.
4. Create `improve-navigation`, change the second content file, and commit it.
5. Switch to `main`, inspect the graph, and merge the branch.

Use `git status`, `git diff`, `git diff --staged`, and the graph view throughout. Before every commit, name the exact snapshot in the index.

Now create an empty remote repository. Add it as `origin`, verify the URL, and push `main` with an upstream relationship.

Create another branch, make one focused commit, push it, and open a pull request. Review the commit range and three-dot diff before asking anyone else to review it.

Finally, practice recovery in a separate disposable branch:

- Stage two files and unstage one without losing its edits.
- Amend a private commit and observe its new ID.
- Revert a shared practice commit with a normal new commit.
- Attempt to delete an unmerged branch with safe `-d` and read the refusal.

Finish by drawing the commit graph. Label `HEAD`, local branches, and remote-tracking references. If you can explain why each reference points where it does, you understand the Git fundamentals behind the commands.

Check your understanding: remotes and GitHub

Check cloning, remote names, fetch and pull, upstream branches, SSH keys, forks, pull requests, and repository connections.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/git/.