



Advanced Workflows with Git and Version Control

Branching and integration strategies for large-scale collaboration, and the CI/CD automation that makes them work

Daniel García Hernández · DevOps & Research Engineer, Barcelona Supercomputing Center

European Open Source Academy · Summer School 2026 · Day 2, 13:30–15:00

Where we are going

- 13:30 Framing and a warm-up poll
- 13:42 1 · Branching, integration, and the people problem
- 14:05 One project, three ways to run it
- 14:13 2 · A history your tools can read
- 14:28 Live demo: a regression, 220 commits deep
- 14:40 3 · Automating it: CI/CD for software and hardware
- 14:58 Wrap-up and questions

Why me

● Day job

DevOps & Research Engineer at the Barcelona Supercomputing Center. I build and maintain the CI/CD behind our RISC-V hardware verification work.

● Why that is useful here

Hardware testing pushes every CI assumption to its limit: jobs that run for hours, licensed tools, enormous artefacts. The lessons transfer downwards easily.

● What it taught me

Almost every painful engineering week I have had traces back to a workflow decision someone made, or never made, early on.

Everything in this session is something that has either saved me, or bitten me.

This is not a Git tutorial

We skip

- clone, add, commit, push
- "what is a branch"
- GUI clients and IDE integrations
- any single vendor's product tour

We dig into

- how projects decide who may merge what
- why that is a people problem more than a tooling one
- keeping a history you can search and test
- CI/CD that stays useful when tests take hours

ONE QUESTION BEFORE WE START

Which of these is most true of your project right now?

- A We ship releases that people install and keep
- B Our tests are too slow to run on every push
- C We work across many repositories at once
- D None of these, I am here to look around

One click. It tells me which half of this session to slow down on.

Your history is not a backup.

It is the record of every decision the project made,
and it is the input your automation reads.

Branch names, tags and commit messages are not paperwork. They are what your pipeline reads to decide what to do.

01



Branching, integration, and the people problem

Who is allowed to merge what, and why the hard part turns out not to be the tooling.

Before pull requests: patches by email

Worth knowing, because it explains why everything looks the way it does today.

● How it worked

You produced a patch and emailed it to a mailing list. The maintainer replied inline with comments. You sent v2, then v3.

git format-patch and git send-email still do exactly this, and Linux and Git themselves are still developed this way.

● What the web platforms changed

The discussion moved next to the code, and the cost of a first contribution dropped enormously.

What did not change: someone still has to read your patch and decide. The review is still where everything happens.

The tooling changed completely. The bottleneck did not move at all.

The branching models you will meet

● Trunk-based

Everyone commits to main, with very short-lived branches. Needs strong tests and a team that trusts each other.

● Pull request flow

One branch per change, review, merge into main. Simple, and it assumes only one version is alive at a time.

● Release branches

main keeps moving while a branch per supported version receives only fixes. Necessary once people install your releases.

● Forks and pull requests

Contributors have no write access at all. The default of the open source world.

The same four, drawn

Each dot is a commit. Time runs left to right.

Trunk-based



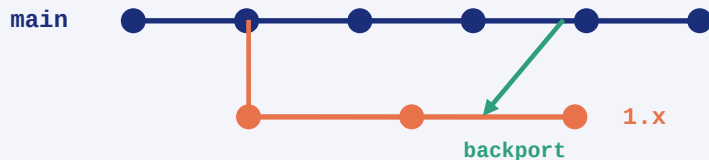
Branches last hours. Everything meets on main.

Pull request flow



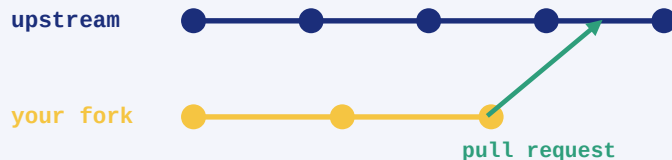
One branch per change, reviewed, then merged.

Release branches



Fixes land on main first, then get copied back.

Forks and pull requests



You cannot write upstream. You ask, and someone merges.

Why forks and pull requests won

- **Write access stops being the gate**
A stranger can propose a change without you trusting them first.
- **Review becomes the contract**
Licensing, style, tests and design all get settled in one place.
- **The maintainer becomes the bottleneck**
Every scaling problem in open source eventually shows up right here.

● The uncomfortable part

Contributions scale beautifully.
Review does not.

Most open source projects in trouble
are not short of contributors. They
are short of people with the time to
read what arrives.

Which means the interesting failure is not a technical one. Here is what that looks like.

When the bottleneck is one person: xz, 2024

Before

xz is a compression library that ships in nearly every Linux system. It was maintained, unpaid and in his spare time, by one person: Lasse Collin.

2021

A new contributor, "Jia Tan", starts sending small, useful, correct patches. For over a year they are genuinely good contributions.

2022

Other accounts appear on the mailing list complaining the project is too slow and demanding a new maintainer. Collin, who had mentioned long-term health problems, gives in.

2023

Jia Tan is co-maintainer, with commit access and the right to sign releases.

Feb 2024

A backdoor targeting SSH ships in xz versions 5.6.0 and 5.6.1. It is found by accident a month later, by an engineer who noticed SSH logins were half a second slower than usual.

No branching model would have stopped this. The attack was aimed at the maintainer, not at the repository.

What the tooling can and cannot do

Two settings are worth turning on in any project. The third column is the one that matters.

● Protected branches

Nobody pushes straight to main, nobody force-pushes it, nobody deletes it. The rule lives in the platform instead of in people's memory.

● Required checks

A merge stays blocked until the tests pass and someone has approved. A gate, rather than a convention.

● What neither can do

Tell you whether the person approving has the time, the energy or the good intentions to do it properly. That is a staffing and funding problem.

Turn the first two on anyway. They cost ten minutes and they catch the ordinary mistakes, which are most mistakes.

So which model do you pick?

Do not start from the model. Start from these facts about your project.

If your project...	...this is what usually fits
Deploys continuously, one live version	Trunk-based, or a simple pull request flow
Ships releases that people install and keep	A release branch for each supported version
Receives changes from people you do not know	Forks and pull requests, with required checks
Has more contributors than reviewers	Fewer required approvals, more automated checks

Most projects match more than one row. That is normal, and it is why the answer is a combination rather than a brand name.

One project, three ways to run it

40 external contributors. A release every six months. Two versions supported in production. Three maintainers, all part-time.

Before a change may be merged, this project requires:

A One approving review, and the tests passing.

B Two approving reviews, and the tests passing.

C Only the tests passing. No review required.

There is no trick answer. All three are running real projects right now.

What each of them costs

None of them is free. The question is which cost this project can actually carry.

● A · one review

Fastest, and with three part-time maintainers it may be the only sustainable choice. You are relying on one person's attention per change.

● B · two reviews

Safer on paper. With three part-time maintainers it can mean nothing merges for a week. Safety you cannot staff is not safety.

● C · tests only

Honest for a small internal tool. On a public project it is how a change reaches your users with nobody having read it.

Underneath all three sits the constraint xz had: hours of honest attention per week.

02



A history your tools can read

Commits are not save points. They are what review, debugging and your pipeline all work on.

Merge or rebase? Wrong question.

Ask instead: what is this history for, and who will read it?

● Merge keeps what happened

True to reality: you can see when branches split and when they came back.

Cost: on a busy project, main turns into a braid that nobody reads.

● Rebase keeps what matters

A straight line, where each commit is a step that works.

Cost: the history is a tidied version of events, and rewriting is risky once other people have your branch.

A useful test: a commit should be a step you could ship, not a moment you happened to save.

Two ways to keep a branch tidy

Both simple, both used constantly, and between them they cover almost everything.

● **git commit --amend**

Fix the commit you just made instead of stacking a "typo" commit on top of it. --amend --no-edit keeps the message and only changes the content.

```
$ git commit --amend --no-edit  
$ git push --force-with-lease
```

● **Squash when you merge**

Every platform offers squash-and-merge in the web interface. The whole branch becomes a single commit on main.

Your messy working history stays in the pull request, where it is still readable. main gets one clear entry per change.

This is what I use in practice, and it needs nobody on the team to learn anything.

The goal either way: main should read as a list of decisions, not a list of Tuesdays.

What not to rewrite, and the way back

● The only real rule

Rewrite freely on a branch only you are using. Never rewrite anything other people have already built on.

● The reassuring part

Almost nothing in Git is actually destructive. Commits you have made stay reachable for about ninety days.

```
# "I amended and lost my work"
$ git reflog
a1b2c3d HEAD@{0}: commit (amend)
9f8e7d6 HEAD@{1}: commit: the work

$ git branch rescue 9f8e7d6
```

The exception: changes you never committed. That is the one thing Git cannot give back.

Who wrote this, and can you prove it?

Where Git meets the licensing conversation from Day 1.

● The author field is just text

Anyone can set any name and email in their Git config. On its own it proves nothing at all.

● Sign-off and signatures

`git commit -s` adds a line saying you have the right to contribute the code. Signing a release tag proves the tag came from a specific key.

● And the honest limit

In the xz case the poisoned releases were signed correctly, by a maintainer who had every right to sign them. Signatures prove who, never whether to trust them.

Still worth signing your release tags. It is what users, packagers and compliance tooling all point at.

Three commands that search your history

● **git bisect**

Binary search across commits to find the one that broke something. 220 candidates become 8 tests. We are about to do this live.

● **git log -S**

Find the commit where a piece of text entered or left the codebase. The answer to "where did this number come from".

● **git blame**

Who last changed each line, and in which commit. Useful mostly for finding the discussion, not for assigning fault.

bisect tells you when it broke, log -S tells you where it came from, blame tells you who to go and ask.

LIVE DEMO

A regression, 220 commits deep

Everything worked at version 1.0.0. Something since then broke one function. Nobody knows what. Nobody remembers touching it.

- Reproduce it with one command
- Let Git find the commit that did it
- Fix it, and keep the history clean

The whole demo, in six lines

```
$ python3 tests/test_median.py    # fails

$ git bisect start HEAD v1.0.0
$ git bisect run python3 tests/test_median.py
5359858 is the first bad commit

$ git bisect reset
```

● Now look again

bisect run needs one thing: a command that exits 0 when things are fine and non-zero when they are not.

That is exactly what a CI job needs too. The little script you just watched is already a pipeline job.

Which is where we go next.

220 commits, 8 checkouts, under a minute. The search was the easy part. Writing the one-line reproducer was the work.

03



Automating it: CI/CD for software and hardware

Turning all of that into pipelines people actually trust, even when the tests take hours.

Anatomy of a pipeline

Four ideas. Everything else is vendor syntax.

```
stages: [build, test]

unit-tests:
  stage: test
  artifacts:
    when: always      # keep the logs even on failure
    paths: [reports/]
  script:
    - pytest -q --junitxml=reports/tests.xml
```

- **Runner**
The machine that executes the job.
- **Stage**
A group of jobs. Later stages wait for earlier ones.
- **Artifact**
A file the job produces and keeps.
- **Cache**
Scratch space reused between runs.

Same project. The tests take two hours.

What runs automatically when someone opens a pull request? Three answers, all of them real.

- A** The full two-hour suite. Correctness first, and the machines can wait.
- B** A ten-minute subset. The full suite runs on main and overnight.
- C** Nothing automatic. A maintainer starts it once the change looks worth it.

C is the honest answer of every project whose CI costs real money.

Branch and tag names are an interface

Your pipeline reads them, which makes them a contract rather than a habit.

```
rules:
- if: $CI_COMMIT_TAG
  # a release: run everything, then publish
- if: $CI_COMMIT_BRANCH == "main"
  # integration: the big suite, overnight
- if: $CI_PIPELINE_SOURCE == "merge_request_event"
  # a proposal: the fast subset, under 10 minutes
```

● The answer you just voted for

Option B, written down. A fast check on proposals, the heavy suite on main, everything on a release tag.

The policy lives in the repository, so it gets reviewed like any other change.

A pipeline nobody trusts is worse than none

Slow and flaky pipelines are how a green tick quietly stops meaning anything.

- **Split by cost, not by type**

A fast check gates the merge. The expensive suite runs on main and overnight.

- **Run only what changed**

A change to the documentation should not start a two-hour run.

- **Keep the failure readable**

Logs and reports as artefacts, kept on failure. A red job nobody can debug gets ignored.

- **Deal with flaky tests**

A test that fails at random teaches everyone to press retry. Fix it, or take it out of the gate.

The failure here is social, not technical. Once people stop reading the results, you pay for the compute and get none of the safety.

What changes when the target is hardware

Same principles, different numbers, and the numbers are brutal.

● Jobs run for hours

A simulation is not a unit test. The merge gate has to be a genuinely different, smaller set of checks.

● Machines are not disposable

Licensed tools and boards live on specific machines. Self-hosted runners, limited capacity, real queues.

● Artifacts are enormous

Waveforms and logs run to gigabytes. Keep them for failures, expire them quickly, never commit them.

● Random tests need fixed seeds

If the seed is not recorded with the result, a failure you cannot reproduce is a rumour rather than a bug report.

Long test suites, GPU jobs, licensed tooling and shared lab machines all have this shape. It is not only a chip problem.

When it is not one repository

A short war story, and the general lesson underneath it.

- **The situation**

Several repositories, each with its own pipeline, all expecting the same set of CI variables.

- **The failure**

One variable renamed in one place. Everything else keeps working, until a pipeline dies an hour in for no visible reason.

- **The fix**

One shared check, included by every repository, that verifies the contract before any real work starts.

- **The general lesson**

Anything duplicated across repositories will drift apart. The only question is whether you find out in review, in CI, or in front of a user.

Make the shared thing a real dependency, and make it fail loudly and early.

Where this sits in the rest of the school

● Licensing and legal frameworks

Authorship and sign-off are the evidence layer underneath every licence claim.

● Standardisation and trust

A signed release tag is what packagers and compliance tooling point at.

● Sustainable business models

Review capacity, not contributor count, is what limits an open source project.

● Digital sovereignty

Controlling your history and your build infrastructure is the concrete version of controlling your software.

And one thread running through all of it: the hard problems in open source are usually about people, not tools.



Thank you. Questions?

`git help <command>` is genuinely good, and so is `git help -g` for the tutorials nobody knows exist.

Daniel García Hernández · daniel.garcia1@bsc.es · Barcelona Supercomputing Center