CI/CD

Version control systems

Aleix Llusà Serra

Master's degree in Machine Learning and Cybersecurity for Internet Connected Systems
Universitat Politècnica de Catalunya
https://epsem.upc.edu

28 d'octubre de 2026



This work is licensed under a Creative Commons "Attribution-ShareAlike 4.0 International" license.

1 Forge

2 Version control systems

3 Git

4 References

Forge

Manage all the project's lifecycle:

- revision control system
- code review and history
- documentation
- bug tracking system
- release management
- workflow discussion
- continuous integration
- project and task management

Examples: Github, Gitlab, Openproject...

GitOps

Definition

Git as the single source of truth.

DevOps trough Git-based workflows with containerization technologies.
`https://about.gitlab.com/topics/gitops/`

Version control systems (VCS)

Definition

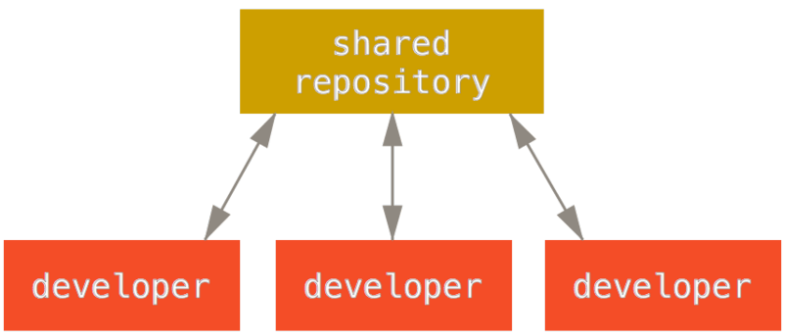
Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later (Chacon i Straub 2014)

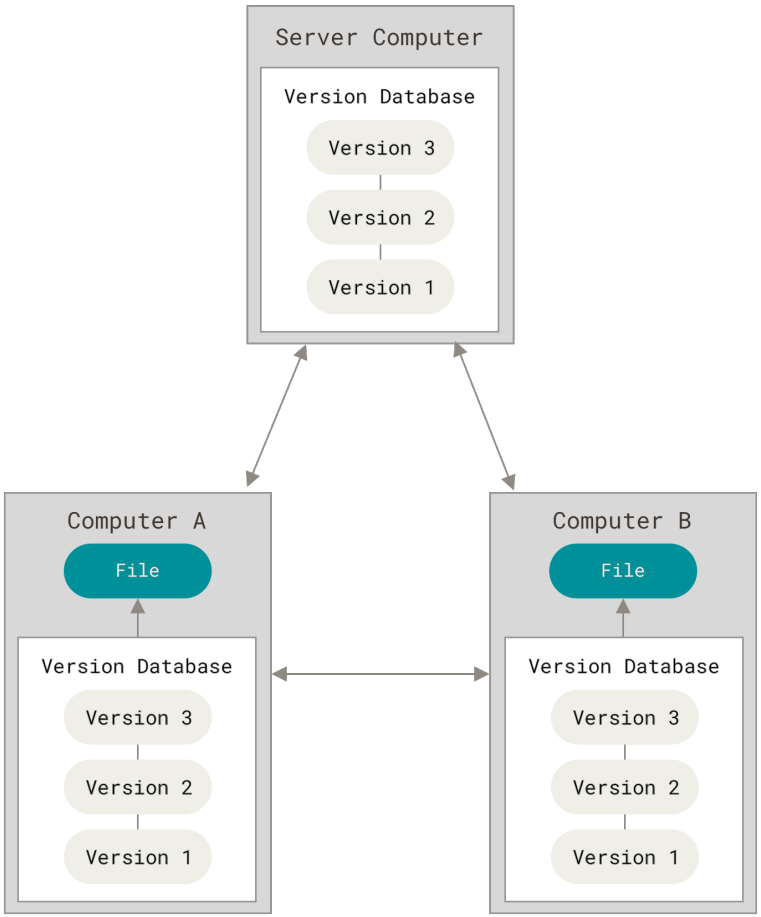
Revert files, compare changes over time, blame on someone's modifications...

Centralised vs distributed

Centralised: server contain all, client has a working copy

Distributed: fully mirror the repository to each client, including history





Source: Chacon i Straub 2014

Working reference

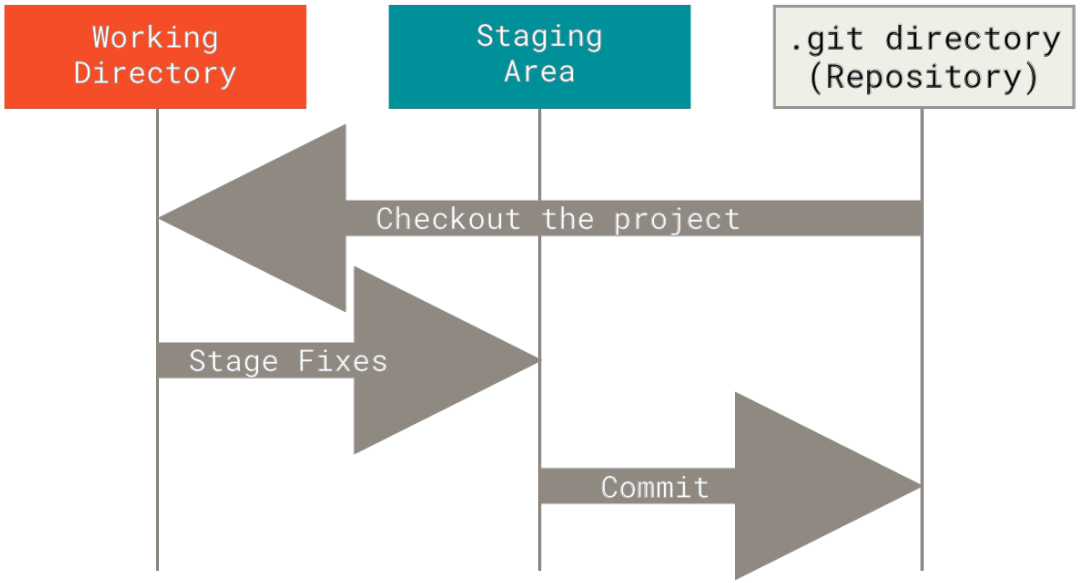
- Version control: repository/master storage. A central reference is needed, where CI/CD happens.
- Concurrent model (working simultaneously) better than locking model.

Git: remote repositories

- Remote repository in Git actually means another/elsewhere (for clarity's sake you can simplify as a central reference)
- origin is the default remote from where you have cloned
- Multiple remotes can be regarded as forks
- `git fetch` downloads from remotes to your local
- Although `git pull` (fetch+merge) is more common
- `git push <remote> <branch>` uploads your local to a remote if nobody has pushed after your last fetch!

Git: the three states

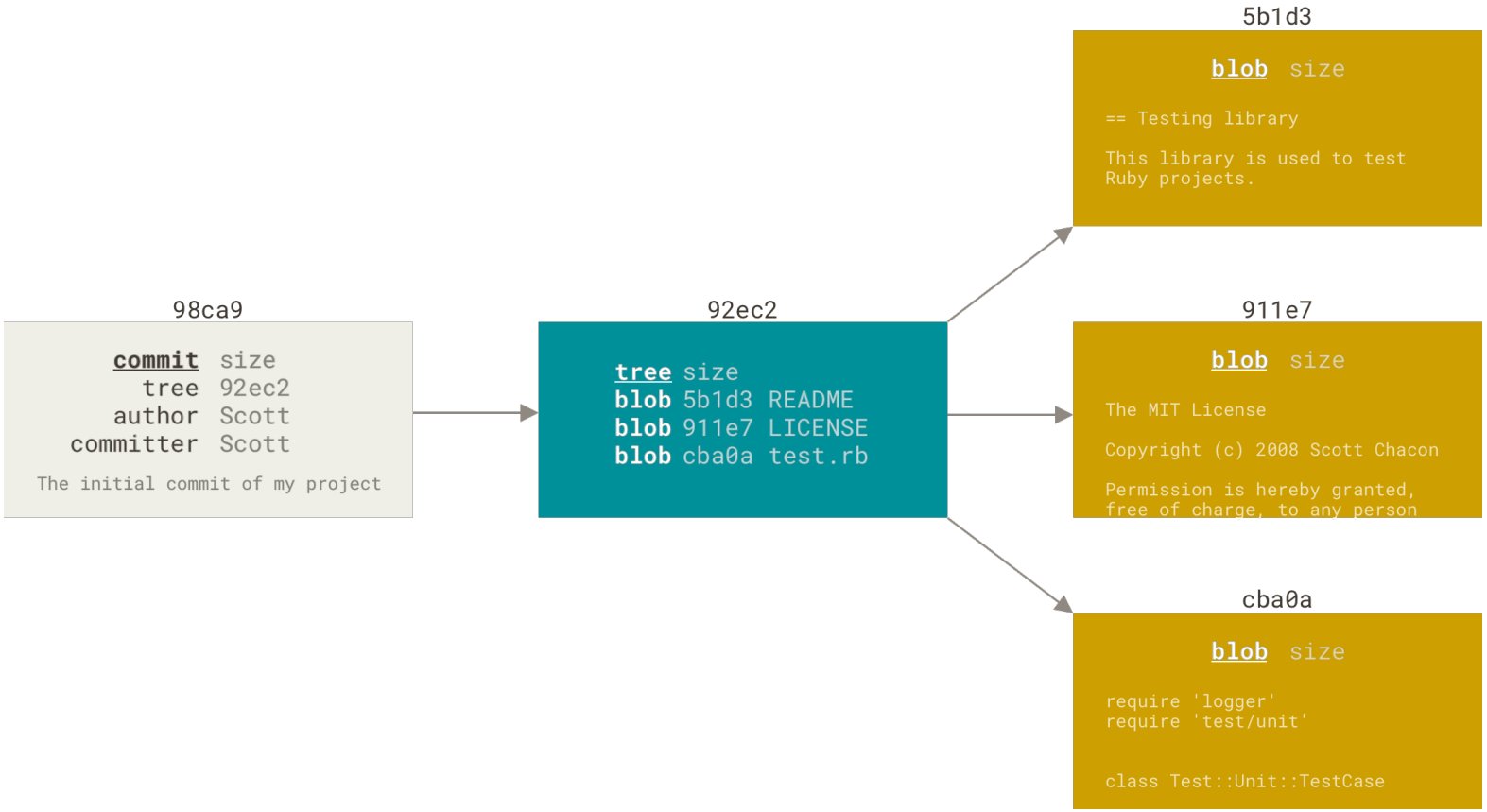
Modified, stages, committed



Source: Chacon i Straub 2014

Git commit

A commit is a pointer to a root tree plus metadata plus a pointer to the parent commit. A tree is a pointer to the contents of a directory. Each file is a blob.



Source: Chacon i Straub 2014

Git tagging

Tag specific points in history. E.g. release points.

- Lightweight `git tag <tag>`: pointer to a specific commit (like a branch that doesn't change)
- Annotated `git tag -a <tag> -m <message>`: stored as full objects (checksum, tagger name email and date, can be signed and verified)

You are always working on your local repository: so recall to

`git push <remote> <tag>`

Git branching

Diverging from the main line (master is the default name). In Git, branches are just pointers to commits.

- `HEAD` is a pointer to your current branch
  i.e. controls which actual files are in your working directory
- `git branch <branch>`: Create branch
- `git branch -d <branch>`: Remove a branch
- `git checkout <branch>`: Switch branch (change `HEAD`)
  if there are no conflicts in doing so
- `git checkout -b <branch>`: Create and switch
- There is also a `checkout` equivalent: `git switch <branch>` and `git switch -c <branch>`

Git branching and remotes

Recall you are always working on your local repository. If you need to share remotely:

- Push a created branch: `git push <remote> <branch>`
  `git push <remote> <localbranch>:<remotebranch>`
- Fetch (only download) remotes branch: `git fetch <remote>`
- Work on the new branch `git checkout -b <branch> <remote>/<branch>`
- Delete a remote branch: `git push <remote> --delete <branch>`

Please, could git branching be easier?

Let us call local branches as tracking branches and remote branches as upstream branches.

- `git clone` automatically tracks master to upstream origin/master
- Track upstream: `git checkout --track <remote>/<branch>`
- Quick alias: `git checkout <branch>`
  (if doesn't exist, exactly matches only one remote and was fetched)
- Change tracking upstream: `git branch -u <remote>/<branch>`
- Use `git pull` to fetch and merge in your current branch (HEAD)
- Use `git push` to push your current branch (HEAD)

Branching vocabulary

- Diverging
- Merging `git merge <branch>` (into current HEAD)
  - Merging fast-forward
  - Three-way merging
  - Merging conflicts
- Rebasing `git rebase <branch>` (in front of current HEAD)
- Stashing (a temporal one parallel to the three states)

Git pull: merge vs rebase?

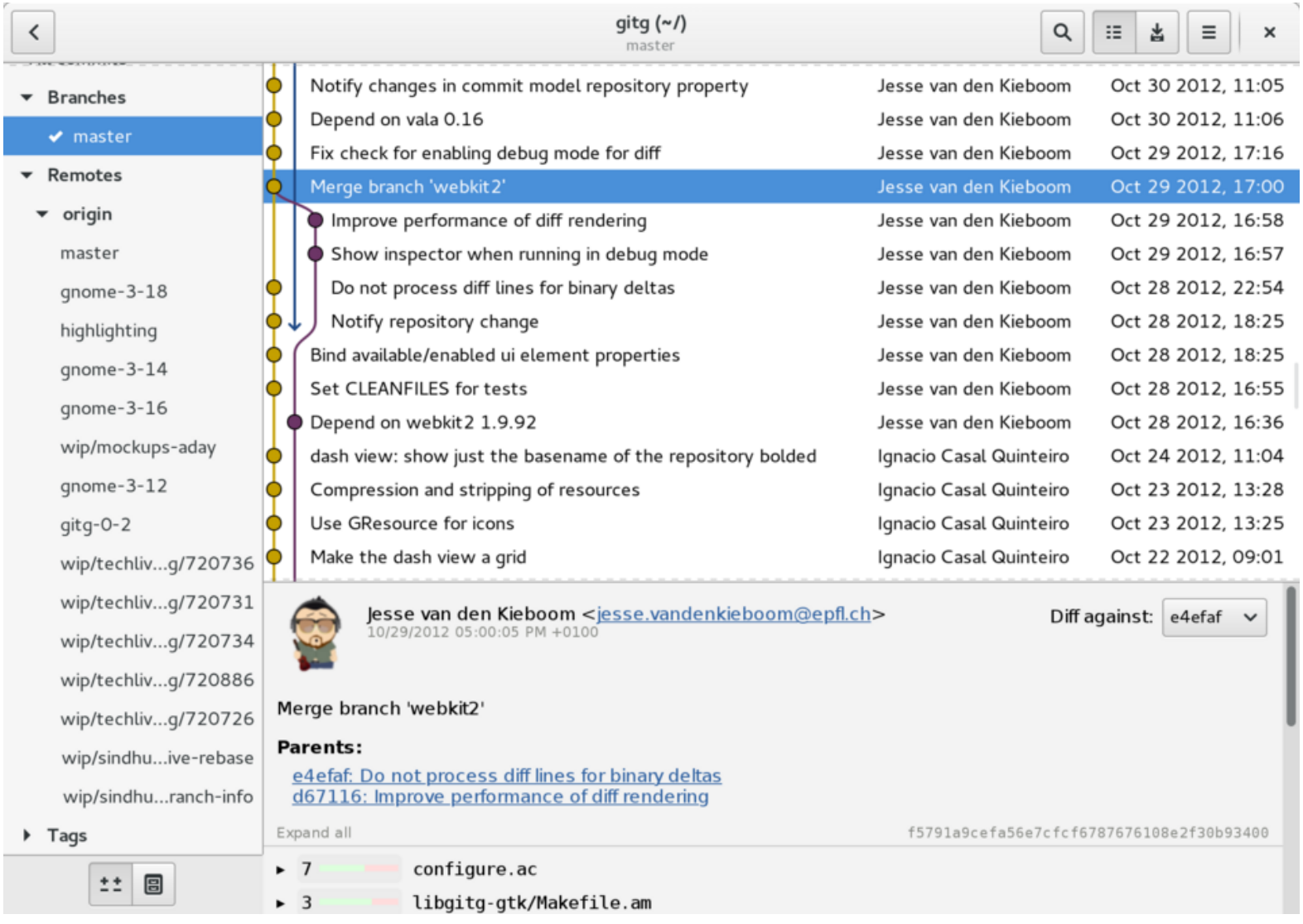
Each team decides: what actually happened or how the project was on production

The default behavior of `git pull` is fast-forward if possible, else create a merge commit. However:

- Rebasing: cleaner history, linear history
- BUT! Do not rebase commits that exist outside your repository and that people may have based work on
- That is, never do `git push --force`
- And always use `git pull -r`
  or set as default `git config --global pull.rebase "true"`

Too many branches for my taste

How to know: do you need a graphical tool to plot your project's history?



© The GNOME Project. Wikimedia Commons.

Merging workflow

Different options:

- Permanent (long-running) branches: master, develop, next, testing, hotfix, proposals... Strong code ownership (Shore i Warden 2008)
- Merge request/pull request: protected master, merge branches. Short-lived branches (topics): create, work, merge, and delete. Also for non writable repository, fork and merge request. Weak code ownership (Shore i Warden 2008)
- Agile trunk-based development: fail fast, minimize work in progress, pull requests are too slow for CI, we are all responsible for all our code so fix problems no matter where you find them, collective code ownership (Shore i Warden 2008)

Recall we aim at ours team project's lifecycle. Maybe for free software open collaboration projects the merging workflow has other necessities, see Chacon i Straub (2014) e.g. benevolent dictator workflow.

Who can I blame when pull request?

Git has author and committer.

- On merge pull requests: author=committer. There are two commits:
  1. Topic: author=committer=developer
  2. Merge: author=committer=maintainer (who is accepting pull request)
- On fast-forward pull requests: author!=committer. Only one commit:
  1. Topic: author=developer committer=maintainer

My Git cheatsheet

- `git clone`
- Snapshotting: `git status add rm commit log diff...`
- `git pull -r`
- `git push`
- `git checkout <branch>` and `git checkout -b <branch>`
- `git stash` and `git stash pop`
- Gitignores: `https://github.com/github/gitignore`

And other things? I need to look at the manual...

Exercise: Pull requests/Merge requests

Commit on a branch, push the branch and ask a merge request.

- Generate the merge request on Gitlab, don't go the hard way with `git request-pull origin/main origin`.
- Long-lived discussion: Try to comment on the merge request, make code suggestions, accept suggestions...
- A merge request is from branch (or fork/branch) to branch. So you can add new commits to the branch to work on the merge request.
- Go to settings -> Merge Requests -> Fast-forward merge.

`https://docs.gitlab.com/ee/user/project/merge_requests/`

References


Chacon, Scott i Ben Straub (2014). ProGit. 2a ed. CC BY-NC-SA 3.0. Apress. ISBN: 1484200772. URL: https://git-scm.com/book/en/v2.


Shore, James i Shane Warden (2008). The Art of Agile Development. O'Reilly. ISBN: 9780596527679. URL: https://discovery.upc.edu/permalink/34CSUC_UPC/8e3cvp/alma991005248477306711.