Engineer

Repository

a store that keeps saved versions of a set of files

What it is

A repository, nearly always shortened to repo in conversation, is where a version control system keeps the full record of a project: every recorded state of every file, who made each change and why, and the names that lead into that history. The working files you edit are one view of it; the repository is everything behind that view.

Git defines the word in its own glossary with some precision: “A collection of refs together with an object database containing all objects which are reachable from the refs”. Both halves are worth unpacking. The object database holds the content. Each version of a file, each directory listing and each commit is stored as an object and named by a hash of its contents, which the Git book describes as a content-addressable filesystem: a key-value store where the key is computed from the value. Two identical files are one object, and any change, however small, produces a new name. The refs are the human side: branches and tags, short names that each point at one commit and give people a way in.

Most repositories people work in come with a working tree attached. The files you open in an editor sit in the project directory, and the repository itself lives beside them in a hidden `.git` directory. A bare repository has no working tree at all. Its directory name conventionally ends in `.git`, it holds only the history, and it is the form a server keeps, because nobody edits files there; people push to it and fetch from it. Git’s glossary says as much: publishers of public repositories usually make bare ones available. The Git book shows how to make one from an existing project with `git clone --bare my_project my_project.git`.

Cloning is where Git’s design shows. A clone receives nearly all the data the server has: by default, every version of every file for the whole history of the project comes down with it. Every clone is therefore a complete repository,

able to stand in for the server if the server is lost. The exceptions are small but matter, and the Git book names them: server-side hooks and similar settings stay behind.

Two variations come up often. A shallow repository, created by cloning with `--depth`, keeps only recent commits; the glossary says the older parents are “cauterized away”, and the history can be deepened later with a fetch. A submodule is a repository held inside another one, with its own history, pinned by the outer repository to one exact commit.

Use the concept in a decision

Put “What is a repository?” to work in a real decision. We can help relate the concept to the system you are building or buying. Tested, released and reversible, so changes reach customers on a Tuesday afternoon rather than late on a Friday night.

Explore our services Get in touch →

Test the assumptions behind the term

Use the explanation and linked work to frame questions for your project. We can examine the assumptions with you before a tool is chosen. Automated GIS SaaS deployment with GitHub Actions — deployment, data and reporting, all off the hands

Automated GIS SaaS deployment with GitHub Actions Get in touch →

Why it matters

A repository is a project’s memory. When a line of code looks wrong, its history says when it arrived, in which commit, and with what explanation. That is only as good as the commits people write, which is why teams that care about their history put rules on commit messages and enforce them with hooks rather than goodwill.

It is also the unit that most tooling works on. Access is granted per repository, hooks run per repository, and continuous integration usually starts from one. Draw the boundary of a repository and you have drawn the boundary of what one review, one check and one permission covers.

Repositories keep what they are given. A commit that deletes a file adds a new state; it leaves every earlier state where it was, in every clone that already holds it. That permanence is the point of version control, and it cuts both ways: an old version is always recoverable, and so is a password that someone committed by mistake.

Finally, a repository can hold far more than application code. Infrastructure definitions, documentation and configuration all gain the same things from it: a history, a review step and a way to rebuild a known state. A server whose setup lives in a repository can be rebuilt by someone who has never seen it.

What it looks like in practice

Four repositories built over eighteen months by one person share one commit standard: a conventional prefix naming the kind of change, a scope, a subject under a fixed length, no emojis and no attribution trailers. A commit-message hook in every repository that carries work rejects anything that breaks it, which makes the standard a property of the repository rather than of whoever is committing. The history then shows what the work was. At the documented July 2026 implementation stage, the largest repository's history read 156 feature commits, 133 fixes, 128 documentation, 81 chores, 26 refactors, 20 style, 5 performance and 3 test. The hook stops short of quality: the format is checked and the content is not.

For the company's own infrastructure, deployed public services and the configuration of other components, the brief was to have it described in a repository rather than in a head. The recorded implementation held 12,065 lines of YAML, split into 19 playbooks and 34 roles; those counts describe that stage, not the current repository. Backup and mail automation are configured there, with operational activation a separate step. The estate can be reproduced from the repository: a host sits in a tier group whose variables declare the roles it runs, and a converged host reports zero changes. Each change takes longer than an edit made directly on the server, and a preflight play was later added in front of the converge.

A repository also keeps what it was given while attention was elsewhere. Work for a different company rode along through a repository restructure, including a scratch log holding live plaintext credentials that sat in the tree past every guard for most of the repository's history. None of it ran, and that is exactly why it survived. The answer was a scope rule at the top of the repository's instructions, one company's infrastructure and only that, plus a quarantine instead of a deletion, so the history stays legible while the secret scanner, the vault guard and the emoji check still read the quarantined files.

Where it goes wrong

- **A secret committed once outlives its deletion.** Removing the file in the next commit leaves it in the previous one and in every clone made since. Treat the credential as public: rotate it first, then decide whether rewriting history is worth the disruption to everyone who has a copy.
- **A push meets a checked-out branch.** Pushing into a non-bare repository whose current branch is the one being updated would leave its working tree out of step with its history, so Git refuses such a push by default through `receive.denyCurrentBranch`. A repository that people push to should be bare.
- **A shallow clone answers history questions from part of the history.** A clone made with `--depth` carries only recent commits, so `git log`, `git blame` and any script that derives a version from the last tag see only that slice. Build and release jobs that need the past should fetch the full history.
- **A local hook stops at the next clone.** Client-side hooks live in `.git/hooks`, and the Git book is explicit that a clone does not copy them. A policy enforced only there ends the day someone clones afresh. Keep the hook scripts tracked in the repository and point `core.hooksPath` at them, or enforce the rule on the server side with `pre-receive`.
- **Large binaries swell every clone forever.** Because a clone carries every version of every file, each replaced copy of a large binary stays in the history and every new clone downloads all of them.
- **Files that nothing runs go unread.** A disabled task, a stale inventory or a scratch log still sits in every clone, yet no test, build or deploy touches it, so it escapes review. The repository's contents should match what the project is responsible for, and anything else should be removed or set apart where the guards can still see it.

Further reading

- gitglossary: repository — Git's own definition, next to bare repository, worktree, shallow repository and submodule.
- Getting a Git Repository — the two ways to get one, starting fresh in a directory or cloning, and what a clone brings down.
- Git Objects — how the object database stores content under a hash of that content.
- gitrepository-layout — what each file and directory inside a repository is for.
- Getting Git on a Server — turning an existing repository into a bare one and sharing it.
- Git Hooks — client-side and server-side hooks, and why a clone leaves the client-side ones behind.

Glossary

Choose for your actual constraints

Bring the deployment path, the operational risks and the people on call. We can scope a change with checks, a recovery route and clear ownership.

See what it costs Get in touch →

Make the terminology practical

Ask how “What is a repository?” applies to your organisation. A concrete example from your current system is enough to start the discussion.

Explore our services

welcome@engineer.company +45 50 10 61 84 Copenhagen, Denmark

Take this page with you

Contact card

Carefully Crafted With Love by D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – present · Engineer ApS · All rights reserved.

Printed from https://engineer.company/glossary/repository/