# GIT ENGINEERING REFERENCE

Author: ddonle.com

## Core Model

Git stores immutable objects identified by SHA hash.

- blob – file contents
- tree – directory snapshot
- commit – tree + parents + metadata
- tag – named pointer to commit

```
A---B---C  (main)
     \
      D---E  (feature)
```

Working Tree → Index (staging) → Commit → Ref.
HEAD points to current branch ref.

## Referring to Commits

Any command accepting `<commit>` understands all of these:

| | |
|---|---|
| `HEAD` | current commit |
| `HEAD~3` | 3 commits ago (also `HEAD^^^`) |
| `main` | tip of branch |
| `origin/main` | remote tracking branch |
| `v1.0` | a tag |
| `3e887ab` | short SHA |
| `HEAD@{2}` | reflog entry (2 moves ago) |

`git rev-parse HEAD~3` – resolve any expression to a full SHA.

## Repository Setup

`git init` – initialize repository in current directory.
`git clone URL` – copy remote repository and history.
`git clone --depth=1 URL` – shallow clone for CI speed.
`git config --list` – show effective configuration.
`git config user.name "X"` – set author identity.
`git config pull.rebase true` – rebase instead of merge on pull.

## .gitattributes

Placed at repo root; controls per-path Git behavior.
`* text=auto` – normalize line endings automatically.
`*.sh text eol=lf` – force LF for shell scripts.
`*.png binary` – treat as binary; disable diff/merge.
`*.cs merge=union` – use union merge strategy for C# files.
`*.pdf diff=pdf` – use named diff driver for PDFs.
Combine with `core.autocrlf` and `core.eol` in config for full cross-platform line ending control.

## Staging + Commit Engineering

`git status` – show working tree vs index state.
`git add file` – stage file content.
`git add -p` – interactively stage hunks.
`git commit -m "msg"` – create commit from index.
`git commit --amend --no-edit` – modify previous commit, keep message.
`git commit --fixup <hash>` – mark commit for autosquash.
`git commit -am ’msg’` — stage all tracked files and commit in one step
`git rebase -i --autosquash` – automatically squash fixups.

## Diff

`git diff` – unstaged changes vs index.
`git diff --staged` – staged changes vs last commit.
`git diff branch1..branch2` – changes between branch tips.
`git diff branch1...branch2` – changes since common ancestor.
`git diff HEAD 1 -- file` – single file change over last commit.

## History Inspection

`git log --oneline --graph --all` – visualize commit DAG.
`git log -p` – show patch for each commit.
`git log --stat` – show file-level diff summary.
`git log --follow file` – track history across renames.
`git blame file` – show line authorship history.
`git log -S"text"` – find commits adding/removing string.
`git show <hash>` – inspect specific commit.

## Branching

`git branch` – list local branches.
`git switch -c feature/x` – create and checkout branch.
`git branch -m old new` – rename branch.
`git branch -vv` – show tracking and upstream info.
`git branch --set-upstream-to=origin/main` – set tracking branch.
`git branch --sort=-committerdate` — list branches by most recently committed to. Useful on repos with many branches.
`git branch -d / git branch -D` — safe delete and force delete

## Merging

`git merge feature` – merge branch into current branch.
`git merge --no-ff feature` – force merge commit.
`git merge --squash feature` – combine changes without history.
`git merge --abort` – cancel in-progress merge.

## Cherry-Pick

`git cherry-pick <hash>` – apply a single commit onto HEAD.
`git cherry-pick A..B` – apply a range of commits (A exclusive).
`git cherry-pick A^..B` – apply range inclusive of A.
`git cherry-pick -n <hash>` – apply changes without committing.
`git cherry-pick --abort` – cancel in-progress cherry-pick.
`git cherry-pick --continue` – continue after resolving conflicts.

## Combining Branches

```
Before:   A--B--C        (main)
               \
                D--E     (feature)
```

**Rebase** – linear history, rewrites SHAs:

```
          A--B--C--D’--E’ (main)
```

`git switch feature`   `git rebase main`

**Merge** – preserves topology, adds merge commit:

```
          A--B--C----M   (main)
               \    /
                D--E
```

`git switch main`   `git merge feature`

**Squash** – one new commit, no branch history:

```
          A--B--C--S     (main, S=D+E)
```

`git switch main`   `git merge --squash feature`

**Rebase vs Merge:** Rebase for clean local/feature history. Merge to preserve true topology on shared branches. Squash to keep main pristine.

**Never rebase shared history.**

## Rebasing

`git rebase main` – replay commits onto new base.
`git rebase -i HEAD~N` – interactively edit last N commits.
`git rebase --onto A B C` – transplant C commits from B onto A.
`git rebase --abort` – cancel rebase.

Use for linear history, not shared public history.

## Conflict Resolution

Git finds the common ancestor, then combines changes. If only one side changed a section, it’s taken automatically. If **both** sides changed the same lines → conflict.

Conflict markers inserted into file:

```
<<<<<<< HEAD       (your version)
=======            (divider)
>>>>>>> feature    (incoming)
```

`git config merge.conflictstyle diff3` – also show ancestor version in conflict blocks. Highly recommended.
`git status` – identify conflicted files.
Resolve manually → `git add file` → `git merge --continue`
**Strategies:**
`git merge -X ours` – auto-resolve conflicts preferring current branch.
`git merge -X theirs` – auto-resolve conflicts preferring incoming.
`git merge -s ours` – discard all incoming changes entirely.

## Rerere (Reuse Recorded Resolution)

`git config rerere.enabled true` – enable rerere globally.
`git rerere` – reuse a previously recorded conflict resolution.
`git rerere diff` – show current vs recorded resolution.
`git rerere forget path` – discard a recorded resolution.
Once enabled, Git silently records how you resolve conflicts and replays the same resolution next time it sees the same conflict. Invaluable on long-lived or frequently rebased branches.

## Undo + Recovery

`git reset --soft HEAD 1` – undo commit, keep staged changes.
`git reset HEAD 1` – unstage changes.
[DANGER] `git reset --hard HEAD 1` – discard changes permanently.
`git restore file` – restore working copy from index.
`git reflog` – show ref history including rewrites. Use to recover from hard resets.
`git checkout HEAD@{2}` – recover previous state.
[DANGER] `git clean -fd` – delete all untracked files/dirs.

## Remote Operations

`git remote -v` – list remotes.
`git fetch --prune` – update remote refs and remove deleted ones.
`git pull` – fetch + merge (or rebase if configured).
`git push` – update remote branch.
`git push --force-with-lease` – safe force push.
[DANGER] `git push --force` – overwrites remote, may lose others' work.
`git push origin --delete branch` – delete remote branch.

## Refspecs

Format: `[+]<src>:<dst>`
`git fetch origin refs/heads/*:refs/remotes/origin/*` – explicit fetch mapping.
`git push origin HEAD:refs/heads/feature` – push to named remote ref.
`git fetch origin refs/pull/123/head:pr-123` – fetch a GitHub PR locally.
`+` prefix forces update even without fast-forward.
Omitting `<dst>` deletes the remote ref on push.

## Stash

`git stash push -m "msg"` – temporarily store working changes.
`git stash -u` – include untracked files.
`git stash list` – list stashes.
`git stash pop` – apply and remove latest stash.
`git stash branch name` – create branch from stash.

## Hooks

Stored in `.git/hooks/` – must be executable.
`pre-commit` – runs before commit; non-zero exit aborts commit.
`commit-msg` – validate/edit commit message. $1 = message file path.
`pre-push` – runs before push; use to block failing tests.
`post-merge` – runs after merge; useful for dependency installs.
`git commit --no-verify` – bypass hooks (use sparingly).
For team-shared hooks use a tool like `pre-commit` or `husky`.

## Bundle + Archive

**Bundle** – pack repo history into a single portable file.
`git bundle create repo.bundle --all` – bundle entire repo.
`git bundle create patch.bundle HEAD 5..HEAD` – bundle last 5 commits.
`git clone repo.bundle myrepo` – clone from bundle.
`git fetch repo.bundle main:main` – fetch from bundle into existing repo.
**Archive** – export a snapshot without Git metadata.
`git archive --format=zip HEAD > release.zip` – zip working tree at HEAD.
`git archive v1.0 | gzip > release.tar.gz` – archive a tag.

## Worktrees

`git worktree add ../path branch` – check out branch in new directory.
`git worktree add -b new ../path` – create and check out new branch.
`git worktree list` – list all linked worktrees.
`git worktree remove ../path` – remove a worktree.
Enables working on multiple branches simultaneously without stashing. Each worktree shares the same object store.

## Advanced Debugging

`git bisect start` – begin binary search for bad commit.
`git bisect bad` – mark current commit as bad.
`git bisect good` – mark known good commit.
`git bisect run script` – automate testing during bisect.
`git bisect reset` – exit bisect and return to original HEAD.

## Submodules + Sparse Checkout

`git clone --recurse-submodules URL` – clone and init submodules in one step.
`git submodule add URL` – include external repository.
`git submodule update --init --recursive` – initialize submodules.
`git submodule foreach git pull` – update all submodules to latest.
`git fetch --recurse-submodules` – fetch upstream changes in submodules.
`git push --recurse-submodules=check` – abort if submodule changes unpushed.
`git submodule deinit path` – unregister a submodule.
`git sparse-checkout init` – enable sparse mode.
`git sparse-checkout set dir/` – checkout subset of tree.
`git sparse-checkout list` – show current sparse patterns.
`git sparse-checkout disable` – return to full checkout.

## Plumbing Commands

`git cat-file -p <hash>` – inspect raw object.
`git ls-tree <commit>` – inspect tree object.
`git rev-parse HEAD` – resolve ref to hash.
`git update-ref ref hash` – manually update reference.
`git fsck` – verify object integrity.
`git gc` – optimize and garbage collect repository.

## Maintenance + Performance

`git maintenance start` – register repo for scheduled background maintenance.
`git maintenance run` – run all maintenance tasks immediately.
`git maintenance stop` – unregister from scheduled maintenance.
`git config core.commitGraph true` – enable commit-graph for faster traversal.
`git commit-graph write --reachable` – manually write commit-graph file.
`git config feature.manyFiles true` – optimize index for large repos.
Prefer `git maintenance` over ad-hoc `git gc` in long-lived repos.

## Release + Tagging

`git tag v1.0` – lightweight tag.
`git tag -a v1.0 -m "msg"` – annotated tag.
`git tag -s v1.0` – signed tag.
`git push --tags` – push all tags.

## Large-Scale Engineering Practices

• Prefer rebasing feature branches locally. • Protect main branches. • Use signed commits in regulated environments. • Use shallow clones in CI pipelines. • Enable commit graph for performance: `git config core.commitGraph true` • Prefer force-with-lease over force.

## .gitignore

`*.ext` – ignore by extension anywhere in tree.
`dir/` – ignore a directory.
`/root-only` – anchored to repo root (has leading /).
`!important.log` – negate a previous rule; later rules win.
`**/logs/` – recursive glob; matches at any depth.
`.git/info/exclude` – local-only ignores; never committed.
`git check-ignore -v file` – debug why a file is being ignored.
Order matters: rules are evaluated top-to-bottom, last match wins.

| | |
|---|---|
| 🟦 Core concepts | ⬛ Remote / collaboration |
| 🟧 History rewriting | 🟪 Config / tooling |
| 🟥 Destructive / recovery | 🟩 Workflow / release |