Git Cheat Sheet

GitVibes — Git for Vibe Coders · neovand.github.io/gitvibes

Some of these commands have a blank to fill in. Three notations mark one:

`<branch> <file> <url>` Your own word goes here — `git switch <branch>` means `git switch main`. Never type the angle brackets.

`<commit>` Any way of naming one commit: a hash like `a1b2c3d`, a branch or tag name, or a walk back from where you are — `HEAD~2` is "two commits before HEAD".

`<remote>` The nickname for a repository elsewhere. It is almost always `origin`, the one `git clone` sets up for you.

Setup & Config

```
git init
```

Initialize a new local repository

Creates the hidden .git folder that will hold all history. Safe to run in a folder that already has files — nothing is tracked until you add and commit.

```
git init <directory>
```

Create a new repo in the specified directory

Makes the directory (creating it if needed) and initializes it in one step — equivalent to mkdir + cd + git init.

```
git clone <url>
```

Clone a remote repository to your machine

Downloads the entire history, not just the latest files, and wires up 'origin' pointing back at the source so push and pull just work.

```
git clone --depth 1 <url>
```

Shallow clone with only the latest commit

Grabs only the newest snapshot — much faster for huge repos when you don't need history. Deepen later with git fetch --unshallow.

```
git config --global user.name "<name>"
```

Set your name for all repositories

Recorded into every commit you make on this machine. Use --local inside a repo to override it for that one project (e.g. a work identity).

```
git config --global user.email "<email>"
```

Set your email for all repositories

Match an email registered on your GitHub account or commits will not link to your profile. The same --local override applies per repo.

```
git config --global core.editor "<editor>"
```

Set default text editor for Git

The editor Git opens for commit messages, rebase plans and merges. 'code --wait' makes VS Code work — --wait keeps Git paused until you close the tab.

```
git config --list
```

List all current configuration settings

Shows the merged result of system, global and local config. Add --show-origin to see which file each value comes from when settings fight.

```
git config --global credential.helper <helper>
```

Store HTTPS credentials (osxkeychain, manager, libsecret)

Lets HTTPS remotes remember your token so you authenticate once. macOS: osxkeychain · Windows: manager · Linux: libsecret.

```
ssh-keygen -t ed25519 -C "<email>"
```

Generate an SSH key pair for authentication

Creates a modern key pair in ~/.ssh. Paste the .pub half into GitHub → Settings → SSH keys; the private half never leaves your machine.

```
ssh -T git@github.com
```

Test your SSH connection to GitHub

Prints a greeting with your username when the key is set up. 'Permission denied (publickey)' means GitHub never saw your key — check ssh-add and the uploaded .pub.

```
gh auth login
```

Sign in to GitHub from the terminal (browser flow, no tokens to manage)

The GitHub CLI stores credentials for both git-over-HTTPS and the API. The browser flow means no personal access token to create, scope, or rotate.

```
git config --global core.excludesFile
~/.gitignore_global
```

Global ignore file for OS and editor junk

One machine-wide list for .DS_Store, Thumbs.db and editor artifacts — so each project's .gitignore stays about the project.

```
git config core.hooksPath .githooks
```

Use a versioned, shareable hooks directory

Points Git at a hooks folder that lives in the repo itself, so the whole team — and any AI agent that clones it — gets the same checks automatically.

Basic Workflow

```
git status
```

Show the working tree status

Your home base: staged, unstaged and untracked files at a glance. It even names the exact command to undo whatever it is showing you.

```
git add <file>
```

Stage a specific file for commit

Staging marks the file's CURRENT content for the next commit — edit the file again afterwards and the new edits are not included until you re-add.

```
git add .
```

Stage all changes in the current directory

Convenient but indiscriminate: it grabs everything, including secrets and debug files. After an AI session, review git status first and stage selectively.

```
git add -p
```

Interactively stage hunks of changes

Shows every change hunk-by-hunk and asks y/n. The single best habit for reviewing AI-generated edits before they become history.

```
git commit -m "<message>"
```

Commit staged changes with a message

Write the subject in imperative mood ('fix login bug'), ideally under 50 characters. Conventional prefixes (feat:, fix:, chore:) make history scannable.

```
git commit -am "<message>"
```

Stage and commit tracked files in one step

The -a stages every TRACKED file's changes first. Brand-new files are not included — they still need an explicit git add.

```
git commit --amend
```

Modify the most recent commit

Folds whatever is staged (or just a new message) into the previous commit. It rewrites history: safe before pushing, never after.

```
git diff
```

Show unstaged changes in the working directory

The unstaged view — exactly what git restore would throw away. Narrow it to one area with a path: git diff src/.

```
git diff --staged
```

Show changes staged for the next commit

Exactly what the next commit will contain — the last checkpoint before git commit. Read this instead of trusting an AI's summary of its own changes.

```
git rm --cached <file>
```

Stop tracking a file without deleting it (pair with .gitignore)

The file stays on disk but leaves the index — the way out after committing something that belongs in .gitignore, like .env or node_modules.

```
git commit --no-verify
```

Skip pre-commit and commit-msg hooks (use sparingly)

Bypasses the checks your hooks enforce. Defensible for a WIP commit on a private branch; a red flag anywhere near main or a PR.

Branching

```
git branch
```

List all local branches

The current branch is starred. Add -v to see each branch's tip commit and how far ahead or behind its upstream it sits.

```
git branch -a
```

List all branches including remote

The origin/* entries are your last-fetched snapshots of the server, not live views — run git fetch first for a current picture.

```
git branch <name>
```

Create a new branch

Creates the branch at your current commit but stays where you are (use switch -c to also move). A branch is just a tiny pointer — make them freely.

```
git branch -d <name>
```

Delete a fully merged branch

Refuses if the branch holds unmerged work — a built-in safety net. Deleting a merged branch removes only the label, never the commits.

```
git branch -D <name>
```

Force delete a branch regardless of merge status

Skips the merge check; the commits become unreachable (still recoverable via reflog for a few weeks). For experiments you truly abandon.

```
git switch <branch>
```

Switch to an existing branch

The modern, purpose-built half of the old checkout. It refuses to overwrite uncommitted changes — stash or commit first.

```
git switch -c <branch>
```

Create and switch to a new branch

Branch and move in one step. Add a start point (git switch -c fix main) to branch from somewhere other than where you stand.

```
git merge <branch>
```

Merge the specified branch into the current branch

Brings <branch> INTO the branch you are on — so switch to the receiving branch first. Fast-forwards when possible, otherwise creates a merge commit.

Remote

```
git remote -v
```

List all remote connections with URLs

Two URLs per remote (fetch and push, usually identical). The fastest answer to 'where does this repo actually push to?'

```
git remote add <name> <url>
```

Add a new remote repository

'origin' is only a convention. On a fork, add the original as 'upstream' and you can pull their updates alongside your own remote.

```
git remote remove <name>
```

Remove a remote connection

Deletes only the local connection and its remote-tracking branches. The repository on the server is untouched.

```
git remote rename <old> <new>
```

Rename an existing remote

Renames the remote and rewrites every branch's upstream reference to match — nothing to reconfigure afterwards.

```
git fetch <remote>
```

Download objects and refs from a remote

Downloads new commits into origin/* WITHOUT touching your working files. Always safe: look first (git log origin/main), merge second.

```
git fetch --prune
```

Fetch and remove stale remote-tracking branches

Also deletes local origin/* entries whose server branches are gone — the cure for a branch list full of ghosts after PRs merge.

```
git pull <remote> <branch>
```

Fetch and merge changes from a remote branch

Literally fetch + merge. If both you and the remote have new commits it produces a merge commit — or set pull.rebase true for linear pulls.

```
git pull --rebase
```

Fetch and replay your local commits on top (linear history)

Replays your unpushed commits onto the incoming ones — no merge-commit noise. The tidiest way to catch up on a busy shared branch.

```
git push <remote> <branch>
```

Push local commits to a remote branch

Refuses if the remote has commits you have not seen — fetch or pull first. Nothing on your machine leaves it until you push.

```
git push -u <remote> <branch>
```

Push and set the upstream tracking branch

The -u links the local and remote branch once; from then on plain git push and git pull need no arguments. Use it on each branch's first push.

```
git push --force-with-lease
```

Force push, but abort if someone else pushed first (safe force)

Overwrites the remote only if it still matches your last fetch — a teammate's surprise push makes it abort. Never plain --force on shared branches.

Stashing

```
git stash
```

Stash tracked changes (staged and unstaged; add -u for untracked files)

Sweeps uncommitted changes into a side drawer and leaves a clean tree — the escape hatch when an urgent fix interrupts messy work.

```
git stash push -m "<message>"
```

Stash changes with a descriptive message

Unnamed stashes all look alike a week later. A message ('WIP: auth refactor') is the difference between a drawer and a junk pile.

```
git stash list
```

List all stashed changesets

Shows the stack — stash@{0} is the newest. Inspect any entry's contents with git stash show -p stash@{n} before applying it.

```
git stash pop
```

Apply the latest stash and remove it from the list

Apply + drop in one step. On a conflict the stash entry is kept — resolve the files, then git stash drop it yourself.

```
git stash apply
```

Apply the latest stash but keep it in the list

Applies but keeps the entry — the safer choice when you might want the same changes on another branch, or aren't sure they fit here.

```
git stash apply stash@{n}
```

Apply a specific stash by index

Stashes are not branch-bound: stash on one branch, switch, and apply somewhere else entirely. Get n from git stash list.

```
git stash drop stash@{n}
```

Remove a specific stash entry

Deletes one entry and renumbers the rest. Dropped stashes are genuinely hard to recover — peek with show -p first.

```
git stash clear
```

Remove all stash entries

Empties the whole stack permanently — one of the rare Git deletions with no reflog trail. Be sure before you run it.

Undoing

```
git restore <file>
```

Discard changes in the working directory

Overwrites the file with its last committed version — the discarded edits are gone for good. The one everyday command with no undo: git diff first.

```
git restore --staged <file>
```

Unstage a file while keeping changes

Pulls a file back out of the staging area; your edits stay in the file untouched. The move when git add . grabbed something it shouldn't have.

```
git reset <file>
```

Unstage a file (same as restore --staged)

Identical effect to restore --staged — the older spelling you will still meet in documentation, Stack Overflow answers and AI suggestions.

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

Undo last commit but keep changes staged

Uncommits but keeps everything staged, ready to recommit. For 'right changes, wrong packaging' — reshape one commit into several, or fix the message.

```
git reset --mixed HEAD~1
```

Undo last commit and unstage changes

The default reset: uncommit and unstage, with all edits intact in your files. Good for reconsidering what the last commit should contain.

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

Undo last commit and discard all changes

Erases the commit AND every uncommitted change. The commit itself is recoverable via reflog for weeks; uncommitted work lost with it is not.

```
git revert <commit>
```

Create a new commit that undoes a previous commit

The pushed-history undo: adds a NEW commit containing the inverse changes. Nothing is rewritten, so shared branches stay consistent for everyone.

```
git clean -fd
```

Remove untracked files and directories

Deletes untracked files — build junk, stray AI-generated files. Preview with git clean -nd first: untracked files have no history to recover from.

History & Inspection

```
git log
```

Show the full commit history

Authors, dates and full messages. Scope it to a path (git log src/auth/) to read just one area's story.

```
git log --oneline
```

Show commit history in compact format

One commit per line — the skimmable view, and usually enough to find the hash you are about to reset, revert or cherry-pick.

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

Visualize branch history as an ASCII graph

Branches and merges drawn right in the terminal, across every branch at once — the closest thing to a commit-graph UI without leaving the shell.

```
git log -p <file>
```

Show commit history with diffs for a file

Every commit that touched the file, each with its diff — the answer to 'when did this function change, and what did the change look like?'

```
git log --author="<name>"
```

Filter commits by author

Filter history to one committer — including isolating (or excluding) an AI agent's commits when it commits under its own identity.

```
git show <commit>
```

Display details and diff for a commit

The full story of one commit: message, author, and the complete diff. git show HEAD is a quick self-review of what you just committed.

```
git blame <file>
```

Show who last modified each line of a file

The last commit to touch every line, with author and date. Feed that hash to git show to learn the WHY behind a suspicious line.

```
git diff <branch1>..<branch2>
```

Show differences between two branches

What branch2 has that branch1 lacks. Run git diff main..feature before opening a PR to see the change exactly as reviewers will.

Rebase & Cherry-pick

```
git rebase <branch>
```

Reapply commits on top of another branch

Lifts your commits and replays them onto the other branch's tip — linear history, no merge commit. Golden rule: never rebase commits others already have.

```
git rebase -i HEAD~n
```

Interactively rebase the last n commits

The history editor: reorder, reword, squash or drop the last n commits. The standard cleanup between 'wip wip wip' and a reviewable PR.

```
git rebase --continue
```

Continue after resolving rebase conflicts

After fixing a conflict, git add the files and continue — no commit needed mid-rebase. Git repeats the pause for each conflicting commit.

```
git rebase --abort
```

Cancel the rebase and return to original state

A guaranteed full rewind to the pre-rebase state. No shame in it: abort, breathe, then retry — or merge instead.

```
git cherry-pick <commit>
```

Apply a specific commit to the current branch

Copies one commit's changes onto your branch as a new commit. Perfect for lifting the single good fix out of an abandoned experiment.

```
git cherry-pick <start>..<end>
```

Apply a range of commits (excludes <start> itself; use <start>^.. to include it)

The range excludes <start> — a classic off-by-one. Write <start>^..<end> when <start> itself should come along.

```
git cherry-pick --continue
```

Resume a cherry-pick after resolving conflicts

Same rhythm as a conflicted rebase: fix the files, git add them, continue. The remaining commits in the range keep applying.

```
git cherry-pick --abort
```

Cancel a conflicted cherry-pick and restore the branch

Cancels the pick and restores the branch to where it stood — cleanly unwinding even a half-applied multi-commit range.

Tags

```
git tag
```

List all tags

Alphabetical list. Filter with a pattern (git tag -l 'v2.*'), or add -n to print each tag's message beside it.

```
git tag <name>
```

Create a lightweight tag at the current commit

A lightweight tag is just a pointer — fine as a private bookmark, too thin for a release: no author, no date, no message.

```
git tag -a <name> -m "<message>"
```

Create an annotated tag with a message

Annotated tags are full objects with tagger, date and message — what release tooling expects. The default choice for version numbers.

```
git tag -a <name> <commit>
```

Tag a specific commit

Tags retroactively — realized v1.2.0 actually shipped three commits ago? Find the hash in git log and tag it now.

```
git push <remote> <tag>
```

Push a specific tag to the remote

Tags do NOT ride along with a normal git push — each must be pushed explicitly, or see --tags for all at once.

```
git push <remote> --tags
```

Push all tags to the remote

Ships every local tag in one go — including your private bookmark tags, so explicit single-tag pushes keep the remote cleaner.

```
git tag -d <name>
```

Delete a local tag

Local only. If the tag was pushed, also run git push origin --delete <name> — and expect anyone who already fetched it to still have it.

Advanced

```
git bisect start
```

Begin a binary search to find a buggy commit

Mark one good and one bad commit and Git halves the suspect range each round — a thousand commits take about ten tests to narrow to one.

```
git bisect good <commit>
```

Mark a commit as good during bisect

Also your answer at each checked-out midpoint when the tests pass — bare (no argument) it marks the commit you are standing on.

```
git bisect bad <commit>
```

Mark a commit as bad during bisect

Bare, it marks the current checkout — so a plain git bisect bad is the usual answer when the midpoint fails your test.

```
git bisect reset
```

End the bisect session and return to original HEAD

Run it even after Git names the culprit — until you do, you are still detached at the last tested commit, not back on your branch.

```
git reflog
```

Show a log of all reference updates (recovery tool)

Every place HEAD has been for ~90 days — it survives resets, rebases and deleted branches. The panic button when work 'disappeared'.

```
git reset --hard HEAD@{1}
```

Move the branch back to where HEAD was one move ago

The standard rescue after a wrong reset: HEAD@{1} means "wherever HEAD was one move ago". Read git reflog to pick the right index.

```
git submodule add <url> <path>
```

Add a repository as a submodule

Pins another repo at an exact commit inside yours. Powerful and famously fussy — for dependencies, a package manager is usually kinder.

```
git submodule update --init --recursive
```

Initialize and update all submodules

The incantation after cloning a repo that uses submodules — until you run it, the submodule folders sit empty and builds fail mysteriously.

```
git worktree add <path> <branch>
```

Create a linked working tree for a branch

A second working directory sharing the same .git — run tests on one branch while editing another, or give each AI agent its own checkout.

```
git worktree add -b <branch> <path>
```

Create a new branch in its own working tree

Creates the branch and its directory in one step — the fast lane for spinning up a parallel agent workspace on fresh work.

```
git worktree list
```

List all working trees for this repo

Every checkout of this repository and which branch each one holds — your fleet dashboard when several agents work in parallel.

```
git worktree remove <path>
```

Remove a working tree (branch survives)

Deletes the directory; the branch and its commits survive. Git refuses if uncommitted changes would be lost (--force overrides).

```
git worktree prune
```

Clean up records of deleted working trees

Housekeeping for worktrees removed with plain rm -rf instead of worktree remove — clears the stale bookkeeping entries.