Git CLI Cheat Sheet

Git CLI Cheat Sheet

A professional reference for Git commands, ranging from basic version control to advanced repository management and workflow strategies.

Configuration

Configure your identity, global settings, aliases, and signing keys.

User Identity & Global Config

```
# Set global user name and email
git config --global user.name "John Doe"
git config --global user.email "john.doe@example.com"

# Set default branch name for new repos
git config --global init.defaultBranch main

# Set default editor (VS Code, Vim, Nano)
git config --global core.editor "code --wait"
git config --global core.editor "vim"
git config --global core.editor "nano"

# Set default pull strategy (rebase instead of merge)
git config --global pull.rebase true

# Enable color output for all Git commands
git config --global color.ui auto

# List all configuration settings
git config --list
git config --global --list
git config --local --list

# Get a specific config value
git config --get user.name
git config --get core.editor

# Set a local config (overrides global for this repo only)
git config user.email "local-dev@project.com"
```

Aliases

Git aliases let you create custom shortcuts for frequently used commands.

```
# Single-command aliases
git config --global alias.st status
git config --global alias.co checkout
git config --global alias.br branch
git config --global alias.ci commit
git config --global alias.unstage 'reset HEAD --'

# Multi-command aliases (must use ! prefix)
git config --global alias.amend 'commit --amend --no-edit'
git config --global alias.last 'log -1 HEAD --stat'

# Advanced log aliases
git config --global alias.lg "log --oneline --graph --decorate --all"
git config --global alias.ls "log --pretty=format:'%C(yellow)%h%Cred%d\\ %Creset%s%Cblue\\ [%
git config --global alias.file-log "log --follow --patch -- "

# Alias for a diff with word-level changes
git config --global alias.wdiff "diff --word-diff"
```

.gitconfig File Structure

The global `~/.gitconfig` file stores all your configuration in INI-style sections:

```
# View the raw gitconfig file
cat ~/.gitconfig

# Example structure:
# [user]
#     name = John Doe
#     email = john.doe@example.com
# [core]
#     editor = code --wait
#     autocrlf = input
# [pull]
#     rebase = true
# [push]
#     default = current
#     autoSetupRemote = true
# [alias]
#     st = status
#     co = checkout
# [color]
#     ui = auto
# [init]
#     defaultBranch = main
```

SSH Key Setup

```
# Generate a new Ed25519 SSH key (recommended)
ssh-keygen -t ed25519 -C "user@example.com"
```

```
# Generate with a specific file name
ssh-keygen -t ed25519 -C "user@example.com" -f ~/.ssh/id_ed25519_github

# Generate RSA key (legacy, 4096-bit minimum)
ssh-keygen -t rsa -b 4096 -C "user@example.com"

# Start the SSH agent and add your key
eval "$(ssh-agent -s)"
ssh-add ~/.ssh/id_ed25519

# Test SSH connection to GitHub
ssh -T git@github.com

# Copy public key to clipboard (macOS)
pbcopy < ~/.ssh/id_ed25519.pub

# Copy public key to clipboard (Linux)
xclip -sel clip < ~/.ssh/id_ed25519.pub
```

GPG Commit Signing

GPG signing proves that a commit was actually made by you and has not been tampered with.

```
# List available GPG keys
gpg --list-secret-keys --keyid-format=long

# Configure GPG program
git config --global gpg.program gpg

# Enable GPG signing for all commits
git config --global commit.gpgsign true

# Set your signing key (use the key ID from gpg --list-secret-keys)
git config --global user.signingkey 3AA5C34371567BD2

# Create a signed tag
git tag -s v1.0.0 -m "Signed release v1.0.0"

# Verify a signed tag
git tag -v v1.0.0

# Verify a commit's signature
git log --show-signature

# Sign a single commit without global signing enabled
git commit -S -m "feat: signed commit"

# Skip signing for one commit when global signing is on
git commit --no-gpg-sign -m "quick fix"
```

Repository Setup

Initialize or clone repositories with specific constraints.

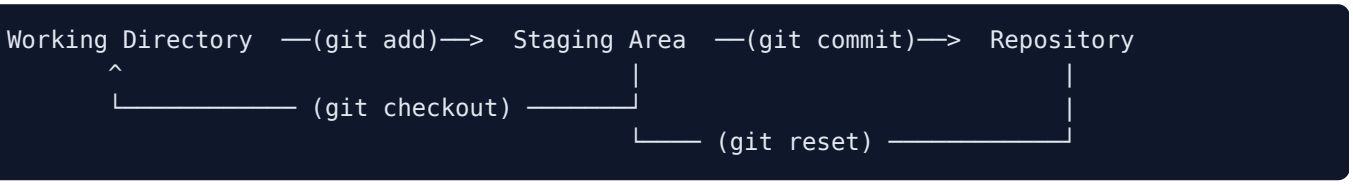
Initialization & Cloning

```
# Initialize a new local repository
git init
git init my-project

# Clone a repository (standard HTTPS)
git clone https://github.com/user/repo.git

# Clone via SSH
git clone git@github.com:user/repo.git

# Clone into a specific directory name
git clone https://github.com/user/repo.git my-folder

# Clone a specific branch only
git clone --branch feature/login --single-branch https://github.com/user/repo.git

# Shallow clone (last N commits — faster, smaller download)
git clone --depth 1 https://github.com/user/repo.git
git clone --depth 5 https://github.com/user/repo.git

# Unshallow a shallow clone later (fetch full history)
git fetch --unshallow

# Clone including submodules
git clone --recurse-submodules https://github.com/user/repo.git

# Create a bare repository (no working directory — for server hosting)
git init --bare project.git

# Mirror a repository (all refs, including remote-tracking)
git clone --mirror https://github.com/user/repo.git
```

The Three States

Git tracks files through three primary stages. Understanding this model is essential to using Git effectively.

1. **Working Directory**: The actual files on your disk. Edits here are "unstaged."
2. **Staging Area (Index)**: A snapshot of what will go into the next commit. Files are "staged" here.
3. **Repository (.git)**: The permanent, versioned record stored as commit objects.




```
# Check the state of your working directory and staging area
git status

# Short format (one line per file)
git status -s

# See changes in the working directory that are NOT yet staged
git diff

# See changes that ARE staged and ready to be committed
```

```
git diff --cached
git diff --staged

# See a word-level diff for readability
git diff --word-diff

# Compare two branches
git diff main..feature/login

# Compare a specific file between two commits
git diff abc123 def456 -- path/to/file.js
```

Basic Commands

Staging Files

```
# Stage a single file
git add file.txt
git add path/to/file.ts

# Stage multiple files
git add file1.txt file2.txt file3.txt

# Stage all changes in the current directory
git add .

# Stage all changes in the entire project
git add -A

# Stage changes interactively (hunk by hunk — highly recommended)
git add -p

# Stage only deletions
git add -u

# Stage a file that you previously committed but want to modify
git add --edit file.txt
```

Committing

```
# Commit staged changes with a message
git commit -m "feat: add login functionality"

# Commit with a detailed multi-line message
git commit -m "feat: add user authentication

- Add login form component
- Implement JWT token validation
- Add redirect on successful auth"

# Stage all tracked files and commit in one step
git commit -am "fix: correct typo in README"

# Amend the most recent commit (fix message or add forgotten files)
git add forgotten-file.ts
```

```
git commit --amend --no-edit

# Amend with a new message
git commit --amend -m "feat: add login functionality and fix validation"

# Commit on behalf of another author
git commit --author="Jane Doe <jane@example.com>" -m "update config"

# Create an empty commit (useful for triggering CI pipelines)
git commit --allow-empty -m "chore: trigger CI rebuild"

# Skip pre-commit hooks
git commit --no-verify -m "hotfix: urgent patch"
```

Inspecting Files & Commits

```
# Show the content of a file at a specific commit
git show HEAD:path/to/file.js
git show abc123:path/to/file.js

# Show the last commit
git show

# Show a specific commit (hash, message, author, diff)
git show <commit_hash>

# Show only the diff of a specific commit
git show <commit_hash> --stat

# Show the changes introduced by the last commit (patch)
git show HEAD -p
```

Removing & Moving Files

```
# Remove a file from Git and the working directory
git rm file.txt

# Remove a file from Git only (keep it on disk as untracked)
git rm --cached file.txt

# Remove an entire directory
git rm -r directory_name/

# Move or rename a file (stages the change automatically)
git mv old_name.js new_name.js

# Move a file to a different directory
git mv file.js src/components/file.js
```

Viewing History

```
# Basic one-line log
git log --oneline
```

```
# Show the last N commits
git log --oneline -5
git log -n 10

# Detailed log with full patch (differences)
git log -p

# Show log with a visual graph of all branches
git log --oneline --graph --all --decorate

# Show statistics of changes per commit
git log --stat

# Show log with a custom format
git log --pretty=format:"%h - %s (%cr) <%an>"

# Show only merge commits
git log --merges

# Show commits from a specific author
git log --author="John Doe"

# Show commits since a date
git log --since="2026-01-01"

# Show commits that modified a specific file
git log -- path/to/file.js

# Follow file history even through renames
git log --follow -- path/to/file.js

# Show the commit graph for a specific branch
git log --graph --oneline feature/login
```

Branching

Creating & Switching Branches

```
# List all local branches
git branch

# List all local AND remote branches
git branch -a

# List branches with the last commit on each
git branch -v

# Create a new branch (stay on current branch)
git branch feature/new-ui

# Create and immediately switch to the new branch
git checkout -b feature/new-ui

# Modern alternative: git switch (Git 2.23+)
git switch -c feature/new-ui
```

```
# Switch to an existing branch
git checkout main
git switch main

# Switch to the previous branch
git switch -

# Create a branch from a specific commit
git branch feature/fix-from-commit abc1234

# Create a branch from a remote tracking branch
git checkout -b feature/remote origin/feature/remote
git switch -c feature/remote origin/feature/remote
```

Comparing Branches

```
# Show commits on feature that are NOT on main
git log main..feature/login --oneline

# Show commits on main that are NOT on feature
git log feature/login..main --oneline

# Show commits on either branch but not both (divergence)
git log main...feature/login --oneline --left-right

# Diff the working trees of two branches
git diff main..feature/login

# List files that differ between two branches
git diff --name-only main..feature/login
```

Renaming & Deleting Branches

```
# Rename the current branch
git branch -m new-name

# Rename a specific branch
git branch -m old-name new-name

# Delete a local branch (only if fully merged)
git branch -d feature/old-ui

# Force delete a local branch (even if unmerged)
git branch -D feature/experimental

# Delete a remote branch
git push origin --delete feature/old-ui

# Prune remote-tracking branches that no longer exist on the remote
git fetch --prune
git remote prune origin
```

Tracking Remote Branches

```
# Set the upstream branch for the current branch
git branch --set-upstream-to=origin/main
git branch -u origin/main

# Push a new branch and set upstream tracking in one step
git push -u origin feature/new-ui

# Show which remote branch the current branch tracks
git branch -vv
```

switch vs checkout

Git 2.23+ introduced `switch` and `restore` as safer alternatives to the multi-purpose `checkout`:

```
# switch — only changes branches (no file restoration side effects)
git switch main
git switch -c new-branch

# restore — only restores files (no branch switching side effects)
git restore file.txt           # Restore from staging area
git restore --source HEAD~2 file.txt  # Restore from a specific commit
git restore --staged file.txt   # Unstage a file

# checkout — still works but does both jobs
git checkout main              # Switch branch
git checkout -- file.txt       # Restore file (confusing syntax)
```

Branching Strategies

Strategy	Best For	Key Pattern	Pros	Cons
Git Flow	Complex release cycles with scheduled releases	`main`, `develop`, `feature/*`, `release/*`, `hotfix/*`	Clear structure, supports parallel releases	Complex for small teams, many merge points
GitHub Flow	Continuous delivery, SaaS products	`main` + short-lived `feature/*` branches	Simple, fast, PR-driven	No explicit release management
Trunk-Based	High-velocity DevOps with feature flags	Commits directly to `main` (or very short-lived branches)	Fastest integration, minimal merge conflicts	Requires feature flags and robust CI

Merging & Rebasing

Merging

```
# Merge a branch into the current branch
git merge feature/login

# Fast-forward merge (only possible if no divergent commits)
git merge --ff-only feature/login

# Force a merge commit even if fast-forward is possible
# (preserves branch history visually)
```

```
git merge --no-ff feature/login

# Squash merge: combine all commits from a branch into one
# (does NOT create a merge commit; changes are staged)
git merge --squash feature/new-ui
git commit -m "feat: add new UI components"

# Abort a merge that has conflicts
git merge --abort

# Merge with a custom merge message
git merge feature/login -m "Merge feature/login into main"
```

Interactive Rebasing

Interactive rebase lets you rewrite, reorder, combine, or remove commits in your branch history.

```
# Start interactive rebase for the last 5 commits
git rebase -i HEAD~5

# Start interactive rebase up to a specific commit
git rebase -i abc1234

# The interactive editor shows these actions:
# pick   — keep the commit as-is
# reword — keep the commit but edit the message
# squash — combine with the previous commit (keep both messages)
# fixup  — combine with the previous commit (discard this message)
# drop   — remove the commit entirely
# edit   — pause to amend the commit content
```

Rebasing Examples

```
# Rebase the current branch onto main (replay your commits on top of main)
git rebase main

# Rebase and autosquash fixup commits (requires commits with fixup! prefix)
git rebase -i --autosquash HEAD~10

# Continue after resolving a rebase conflict
git add <resolved-file>
git rebase --continue

# Skip a commit that's causing issues
git rebase --skip

# Abort the rebase entirely (return to original state)
git rebase --abort
```

Merge vs Rebase — When to Use Which

Aspect	Merge	Rebase
History	Preserves exact commit history	Rewrites commit history (new hashes)

Aspect	Merge	Rebase
Safety	Non-destructive	Destructive (never rebase shared/public branches)
Cleanliness	Creates merge commits (noisy graph)	Linear, clean history
Use case	Integrating shared branches into main	Cleaning up local feature branch before merging

```
# Good: rebase your local feature branch onto latest main
git checkout feature/login
git rebase main

# Bad: rebase main or a shared branch (rewrites shared history)
# DO NOT: git checkout main && git rebase feature/login
```

Conflict Resolution Workflow

```
# When a merge or rebase encounters conflicts:

# 1. See which files have conflicts
git status

# 2. Open the conflicting file and look for conflict markers:
# <<<<<<< HEAD
# (your changes)
# =======
# (incoming changes)
# >>>>>>> feature/login

# 3. Edit the file to resolve conflicts, removing the markers

# 4. Stage the resolved file(s)
git add resolved-file.js

# 5. Continue the merge or rebase
git merge --continue    # for merge
git rebase --continue   # for rebase

# 6. Or abort if you want to give up
git merge --abort       # for merge
git rebase --abort      # for rebase

# Use a visual merge tool for conflict resolution
git mergetool

# Configure a merge tool
git config --global merge.tool vscode
git config --global mergetool.vscode.cmd 'code --wait $MERGED'
```

Stashing

Temporarily save uncommitted changes to clean your working directory without committing.

```
# Stash all tracked changes (default)
git stash
```

```
# Stash with a descriptive message
git stash save "work in progress: header redesign"

# Stash including untracked files (-u / --include-untracked)
git stash save "include new files" -u

# Stash everything (tracked, untracked, AND ignored files)
git stash save "everything" --all

# List all stashes with their index
git stash list

# Apply the most recent stash (keeps it in the list)
git stash apply

# Apply a specific stash by index
git stash apply stash@{2}

# Apply stash and remove it from the list (pop)
git stash pop

# Apply stash to a different branch than where it was created
git stash branch new-feature-branch stash@{1}

# Show the diff of the most recent stash
git stash show

# Show the full diff (patch) of a specific stash
git stash show -p stash@{0}

# Drop (delete) a specific stash
git stash drop stash@{2}

# Drop the most recent stash
git stash drop

# Clear ALL stashes at once
git stash clear
```

Remote Operations

Fetching & Pulling

```
# Fetch all remotes and branches (does NOT merge)
git fetch --all

# Fetch from a specific remote
git fetch origin

# Fetch and prune deleted remote branches
git fetch --prune

# Pull (fetch + merge) from the tracked upstream branch
git pull

# Pull from a specific remote and branch
```

```
git pull origin main

# Pull with rebase instead of merge (cleaner history)
git pull --rebase origin main

# Pull only fast-forward merges (refuse if rebase/merge needed)
git pull --ff-only
```

Pushing

```
# Push current branch to its tracked remote
git push

# Push a specific branch to a remote
git push origin feature/login

# Push a new branch and set upstream tracking (-u / --set-upstream)
git push -u origin feature/login

# Push all local branches to a remote
git push --all origin

# Delete a remote branch
git push origin --delete feature/old-ui

# Force push (overwrites remote history — DANGEROUS)
git push --force origin main

# Force-with-lease: safer alternative (fails if someone else pushed)
git push --force-with-lease origin main

# Push tags along with commits
git push --follow-tags

# Dry run: see what would be pushed without actually pushing
git push --dry-run
```

Remote Management

```
# List all remotes with their URLs
git remote -v

# Add a new remote (common: add upstream for forks)
git remote add upstream https://github.com/original/repo.git

# Show details of a remote (branches, HEAD, push/pull URLs)
git remote show origin

# Rename a remote
git remote rename origin old-origin

# Remove a remote entirely
git remote remove origin

# Change a remote's URL
git remote set-url origin git@github.com:user/new-repo.git
```

```
# Fetch from all remotes at once
git remote update
```

Fork Syncing Workflow

```
# After forking a repo, sync your fork with the upstream:
git remote add upstream https://github.com/original/repo.git
git fetch upstream
git checkout main
git merge upstream/main
git push origin main
```

Tags

Lightweight vs Annotated Tags

Type	Storage	Use Case
Lightweight	Just a pointer (commit hash)	Personal bookmarks, temporary markers
Annotated	Full object (tagger, date, message)	Releases, version milestones, public tags

Creating & Managing Tags

```
# Lightweight tag (just a pointer)
git tag v1.0.0

# Annotated tag (includes message, tagger, date)
git tag -a v1.1.0 -m "Release version 1.1.0 with new auth module"

# Tag a specific commit
git tag -a v1.0.0 abc1234 -m "Backport fix for critical bug"

# Sign a tag with GPG
git tag -s v2.0.0 -m "Signed release v2.0.0"

# List all tags
git tag

# List tags matching a glob pattern
git tag -l "v1.*"
git tag -l "v2.*"

# Show tag details (annotated tags show message + commit)
git show v1.1.0

# Verify a signed tag
git tag -v v2.0.0

# Delete a local tag
git tag -d v1.1.0

# Delete a remote tag
```

```
git push origin --delete v1.1.0
# Alternative syntax
git push origin :refs/tags/v1.1.0
```

Pushing Tags

```
# Push a single tag to remote
git push origin v1.0.0

# Push all local tags to remote
git push origin --tags

# Push tags automatically when pushing commits
git push --follow-tags
```

Cherry-pick

Apply the changes from specific commits to your current branch.

```
# Cherry-pick a single commit
git cherry-pick abc1234

# Cherry-pick multiple specific commits
git cherry-pick abc1234 def5678

# Cherry-pick a range of commits (exclusive of the first)
git cherry-pick abc1234..def5678

# Cherry-pick without committing (changes stay in staging area)
git cherry-pick --no-commit abc1234

# Cherry-pick and edit the commit message
git cherry-pick -e abc1234

# Continue after resolving cherry-pick conflicts
git add <resolved-file>
git cherry-pick --continue

# Abort the cherry-pick process entirely
git cherry-pick --abort

# Quit cherry-pick but keep partially applied changes
git cherry-pick --quit
```

Reset & Revert

Understanding the Three Reset Modes

Mode	Working Directory	Staging Area	Commit History	Use Case
`--soft`	Unchanged	Unchanged	Moved	Combine commits, change last message
`--mixed`	Unchanged	Reset	Moved	Unstage files, "undo add"

Mode	Working Directory	Staging Area	Commit History	Use Case
`--hard`	Reset	Reset	Moved	Throw everything away, start fresh

Reset Examples

```
# Soft reset: undo last commit, keep changes staged
# Useful for combining or amending commits
git reset --soft HEAD~1

# Mixed reset (default): undo last commit, unstage changes
# Changes remain in working directory
git reset --mixed HEAD~1
git reset HEAD~1

# Hard reset: completely discard the last commit and all changes
# WARNING: cannot be undone unless you use reflog
git reset --hard HEAD~1

# Reset to a specific commit
git reset --hard abc1234

# Unstage a single file (move from staging back to working directory)
git reset HEAD file.txt
# Modern alternative (Git 2.23+)
git restore --staged file.txt

# Reset the index but keep all working directory changes
git reset
```

Revert Examples

Revert creates a **new commit** that undoes the changes of a previous commit. Safe for shared branches.

```
# Revert a specific commit (creates a new undo commit)
git revert abc1234

# Revert the most recent commit
git revert HEAD

# Revert multiple commits in one go
git revert abc1234 def5678

# Revert a range of commits (creates individual revert commits)
git revert HEAD~3..HEAD

# Revert a merge commit (must specify the parent to revert to)
git revert -m 1 <merge_commit_hash>

# Revert without auto-commit (edit the revert commit yourself)
git revert --no-commit abc1234
```

Reflog

The `reflog` (reference log) records every move of `HEAD` and branch tips. It is the ultimate safety net for recovering "lost" commits — even after resets, amends, or failed rebases.

```
# View the reflog for the current branch
git reflog

# View the reflog for a specific branch
git reflog show main

# View a specific number of entries
git reflog -10

# View the reflog with a custom format
git reflog --pretty=format:'%h %gd %gs %cr'

# Scenario: recover a "lost" commit after a hard reset

# 1. You accidentally ran: git reset --hard HEAD~3
# 2. Check the reflog to find the commit you lost
git reflog
# Output:
# abc1234 HEAD@{0}: reset: moving to HEAD~3
# def5678 HEAD@{1}: commit: add important feature
# ...

# 3. Reset to the lost commit hash
git reset --hard def5678

# Scenario: recover after a bad rebase
git reflog
# Find the state before the rebase, then reset to it
git reset --hard HEAD@{5}

# Expire reflog entries older than 30 days
git reflog expire --expire=30.days.ago --all

# Expire all unreachable reflog entries immediately
git reflog expire --expire=now --all

# Prune unreachable objects after expiring reflog
git gc --prune=now
```

Worktrees

Worktrees let you check out multiple branches into separate directories simultaneously, enabling parallel work without `git stash` or `git checkout`.

```
# Add a worktree for an existing branch
git worktree add ../hotfix-dir hotfix/critical-bug

# Add a worktree and create a new branch at the same time
git worktree add -b feature/sidebar ../sidebar-dir

# Add a worktree at a specific commit (detached HEAD)
git worktree add ../temp-dir abc1234
```

```
# List all active worktrees
git worktree list

# Remove a worktree (must be clean — no uncommitted changes)
git worktree remove ../hotfix-dir

# Force remove a worktree (discard uncommitted changes)
git worktree remove --force ../hotfix-dir

# Clean up stale worktree administrative files
git worktree prune

# Lock a worktree (prevent pruning or removal)
git worktree lock ../hotfix-dir

# Unlock a worktree
git worktree unlock ../hotfix-dir
```

Bisect

Use binary search to find the exact commit that introduced a bug.

```
# Start a bisect session
git bisect start

# Mark the current (known bad) commit
git bisect bad

# Mark a known good commit
git bisect good v1.0.0

# Git checks out a commit halfway between good and bad.
# Test it, then mark as good or bad:
git bisect good    # bug is not here
git bisect bad     # bug is here

# Automated bisect: run a script that exits 0 for good, non-zero for bad
git bisect start HEAD v1.0.0
git bisect run npm test

# Visualize the bisect progress in gitk
git bisect visualize

# View the bisect log
git bisect log

# Replay a bisect session from a log file
git bisect replay bisect-log.txt

# End the bisect session and return to the original branch
git bisect reset
```

Blame & History Search

git blame

```
# Show who last modified each line of a file
git blame src/components/Login.tsx

# Show blame with the commit hash and author
git blame -e src/components/Login.tsx

# Limit blame to specific line numbers
git blame -L 10,30 src/components/Login.tsx

# Ignore whitespace changes in blame
git blame -w src/components/Login.tsx

# Show the original commit before a move/rename
git blame -M src/components/Login.tsx

# Show the original commit across file copies
git blame -C -C src/components/Login.tsx
```

History Search (Pickaxe)

```
# Pickaxe search: find commits that added or removed a specific string
git log -S "function handleLogin"

# Pickaxe with regex support
git log -S "handleLogin" --pickaxe-regex

# Grep search: find commits whose diff matches a regex
git log -G "TODO|FIXME|HACK"

# Find which commit added a specific string
git log -S "API_KEY" --oneline

# Find commits that modified a specific file path
git log --all -- path/to/file.js

# Show the full diff for pickaxe matches
git log -p -S "search_term"
```

shortlog

```
# Summarize commits by author
git shortlog

# Summarize with email addresses
git shortlog -e

# Show number of commits per author (sorted by count)
git shortlog -sn

# Summarize a specific branch
git shortlog feature/login
```

Submodules

Manage external repositories embedded within your project.

```
# Add a submodule at a specific path
git submodule add https://github.com/user/library.git vendor/library

# Add a submodule at a specific branch/tag
git submodule add -b v2.0 https://github.com/user/library.git vendor/library

# Initialize submodules recorded in .gitmodules
git submodule init

# Clone a repo and initialize all submodules in one step
git clone --recurse-submodules https://github.com/user/repo.git

# Update submodules to the commits recorded in the parent repo
git submodule update

# Initialize AND update in one step
git submodule update --init

# Recursively initialize and update all nested submodules
git submodule update --init --recursive

# Update all submodules to their latest remote versions
git submodule update --remote

# Update a specific submodule to the latest remote version
git submodule update --remote vendor/library

# Run a command inside every submodule
git submodule foreach 'git pull origin main'
git submodule foreach 'echo $path'
git submodule foreach 'git log --oneline -1'

# Show the status of all submodules
git submodule status

# Remove a submodule completely
git submodule deinit vendor/library
git rm vendor/library
rm -rf .git/modules/vendor/library
```

Hooks

Hooks are scripts that Git runs automatically before or after events like commit, push, and receive.

Hook Locations

```
# List available hooks in the current repo
ls .git/hooks/

# Set a custom hooks directory (shared with the team via version control)
git config --global core.hooksPath .githooks
```

```
# Skip all hooks for a single command
git commit --no-verify -m "emergency fix"
```

pre-commit Hook Example

Runs before the commit is created. Common uses: linting, formatting, running tests.

```
# .githooks/pre-commit
#!/bin/bash
# Run linter
npm run lint
if [ $? -ne 0 ]; then
  echo "Linting failed. Commit aborted."
  exit 1
fi

# Run tests
npm test
if [ $? -ne 0 ]; then
  echo "Tests failed. Commit aborted."
  exit 1
fi

# Check for large files (>5MB)
MAX_FILE_SIZE=5242880
find . -type f -not -path './.git/*' -size +${MAX_FILE_SIZE}c | while read file; do
  echo "File too large for commit: $file"
  exit 1
done

echo "Pre-commit checks passed."
```

commit-msg Hook Example

Validates the commit message format.

```
# .githooks/commit-msg
#!/bin/bash
# Enforce Conventional Commits format: type(scope): message
MSG_FILE=$1
MSG=$(cat "$MSG_FILE")

if ! echo "$MSG" | grep -qE "^(feat|fix|docs|style|refactor|perf|test|chore|build|ci)(\(.+\))
  echo "Invalid commit message format."
  echo "Expected: type(scope): description"
  echo "Types: feat, fix, docs, style, refactor, perf, test, chore, build, ci"
  echo "Example: feat(auth): add login validation"
  exit 1
fi
```

pre-push Hook Example

Runs before pushing to a remote. Common uses: running the full test suite.

```
# .githooks/pre-push
#!/bin/bash
echo "Running full test suite before push..."
npm run test:all
if [ $? -ne 0 ]; then
  echo "Tests failed. Push aborted."
  exit 1
fi
echo "All tests passed. Pushing."
```

Using Husky (Popular Hook Manager)

```
# Install Husky
npm install husky --save-dev

# Initialize Husky (creates .husky/ directory)
npx husky init

# Add a pre-commit hook
echo "npm test" > .husky/pre-commit

# Add a commit-msg hook
echo 'npx --no -- commitlint --edit "$1"' > .husky/commit-msg

# Add a pre-push hook
echo "npm run test:ci" > .husky/pre-push
```

Cleaning

Remove untracked files and directories from the working directory.

```
# Dry run: preview what would be deleted (SAFE — no files are removed)
git clean -n

# Dry run with directories included
git clean -nd

# Remove untracked files
git clean -f

# Remove untracked files AND directories
git clean -fd

# Remove untracked files AND ignored files (respecting .gitignore)
git clean -fX

# Remove everything: untracked files, directories, AND ignored files
git clean -fdx

# Remove files only in a specific directory
git clean -fd src/temp/

# Interactive cleaning (choose which files to delete)
git clean -i
```

```
# Discard all local changes (staged + unstaged) and untracked files
git reset --hard && git clean -fd
```

Discarding Changes

```
# Discard changes in a single file (restore to last committed version)
git checkout -- file.txt
# Modern alternative
git restore file.txt

# Discard ALL changes in the working directory
git checkout -- .
git restore .

# Unstage a file (keep changes, but remove from staging)
git reset HEAD file.txt
git restore --staged file.txt

# Discard everything and start fresh from last commit
git reset --hard HEAD
git clean -fd
```

.gitignore

Tell Git which files and directories to ignore.

```
# Check if a file is ignored and which rule ignores it
git check-ignore -v node_modules/

# Create a global gitignore (applies to all repos)
git config --global core.excludesfile ~/.gitignore_global
```

```
# Common .gitignore patterns
# Dependencies
node_modules/
vendor/

# Build output
dist/
build/
*.min.js
*.min.css

# Environment files
.env
.env.local
.env.*.local

# IDE and editor files
.vscode/
.idea/
*.swp
*.swo
```

```
# OS files
.DS_Store
Thumbs.db

# Logs
*.log
logs/

# Large files
*.zip
*.tar.gz
*.psd
```

Troubleshooting

Detached HEAD

You get a detached HEAD when you check out a specific commit instead of a branch. Any commits you make will be "orphaned" when you switch away.

```
# Check current state
git status
# "HEAD detached at abc1234"

# Option 1: Create a branch to save your work
git checkout -b fix-from-detached

# Option 2: Reattach to an existing branch
git checkout main

# Option 3: Move a branch to the detached commit
git branch -f main HEAD
git checkout main
```

Merge Conflicts

```
# See which files have conflicts
git status
git diff --name-only --diff-filter=U

# Open conflicting files and resolve markers:
# <<<<<<< HEAD
# (your changes)
# =======
# (incoming changes)
# >>>>>>> feature/login

# Use a visual merge tool
git mergetool

# After resolving all conflicts
git add .
git commit -m "merge: resolve conflicts in feature/login"
```

```
# Abort the merge entirely
git merge --abort
```

Force Push Recovery

If someone force-pushed and you lost commits, you can recover them.

```
# Find the reflog entry before the force push
git reflog

# Reset to the commit before the force push
git reset --hard HEAD@{2}

# Push again (careful: this may cause further conflicts)
git push --force-with-lease
```

Corrupted Repository

```
# Check repository integrity
git fsck

# Check with full output
git fsck --full

# Check for dangling objects
git fsck --dangling

# Recover dangling commits
git fsck --lost-found

# Remove corrupted objects and repack
git gc --prune=now

# If the .git directory is truly broken, last resort:
# Re-clone the repository
git clone https://github.com/user/repo.git repo-recovered
# Copy over any uncommitted work from the broken clone
```

Large Files & Git LFS

```
# Find large files in the current tree
find . -type f -size +1M | head -20

# Find the largest files in Git history
git rev-list --objects --all | \
  git cat-file --batch-check='%(objecttype) %(objectname) %(objectsize) %(rest)' | \
  awk '/^blob/ {print substr($0,6)}' | \
  sort -n -k2 | tail -10

# Install Git LFS
git lfs install

# Track specific file types with LFS
git lfs track "*.psd"
git lfs track "*.zip"
```

```
git lfs track "*.mp4"
git lfs track "datasets/*.csv"

# Track files above a certain size
git lfs track "*.bin" --attr

# View what patterns are tracked
cat .gitattributes

# Push LFS files to remote
git push origin main --all

# Pull LFS files
git lfs pull

# Migrate existing large files to LFS
git lfs migrate import --include="*.psd,*.zip" --everything
```

Shallow Clone Issues

```
# You have a shallow clone but need full history

# Unshallow: fetch the complete history
git fetch --unshallow

# If unshallow fails (server doesn't support it), re-clone
git clone https://github.com/user/repo.git repo-full

# Shallow clones limit what you can do:
# git log --full-history      # won't work
# git blame                   # limited
# git rebase -i HEAD~10       # may fail if depth < 10

# Fetch additional depth
git fetch --depth=50
```

Committed to Wrong Branch

```
# Undo the last commit but keep changes staged
git reset --soft HEAD~1

# Switch to the correct branch
git checkout correct-branch

# Commit the changes there
git commit -m "feat: add the feature"

# Alternatively: cherry-pick the commit to the correct branch
git checkout correct-branch
git cherry-pick wrong-branch
git checkout wrong-branch
git reset --hard HEAD~1
```

Undo an Accidental Commit (Sensitive Data)

```
# If you committed secrets, passwords, or tokens:

# 1. Remove the file from Git tracking
git rm --cached config/secrets.json

# 2. Add to .gitignore
echo "config/secrets.json" >> .gitignore

# 3. Commit the removal
git commit -m "chore: remove secrets from tracking"

# 4. If already pushed, you MUST rotate the credentials immediately
# 5. Then remove from history using git-filter-repo or BFG:
pip install git-filter-repo
git filter-repo --invert-paths --path config/secrets.json
```

Tips & Aliases

Essential Aliases

```
# Navigation
git config --global alias.st status
git config --global alias.br branch
git config --global alias.co checkout
git config --global alias.sw switch

# Committing
git config --global alias.ci commit
git config --global alias.amend 'commit --amend --no-edit'
git config --global alias.unstage 'reset HEAD --'

# Diffing
git config --global alias.df diff
git config --global alias.wdiff 'diff --word-diff'

# Logging
git config --global alias.lg "log --oneline --graph --decorate --all"
git config --global alias.last 'log -1 HEAD --stat'
git config --global alias.aliases 'config --get-regexp alias'

# Stash
git config --global alias.sl 'stash list'
git config --global alias.sp 'stash pop'
```

Productivity Tips

```
# Auto-correct mistyped commands
git config --global help.autocorrect 1

# Automatically set upstream when pushing a new branch
git config --global push.autoSetupRemote true

# Use git status -s for compact output
git status -s
```

```
# M  modified.txt       (staged modification)
# M  unstaged.txt       (unstaged modification)
# A  newfile.txt        (added)
# D  deleted.txt        (deleted)
# ?? untracked.txt      (untracked)

# Use git add -p for surgical staging (review each hunk)
git add -p

# Quickly view the diff of staged changes
git diff --cached

# See which branch you're on and what's changed (one-liner)
git status -sb

# Find which branch contains a specific commit
git branch --contains abc1234

# Search commit messages
git log --all --grep="fix crash" --oneline

# Show commits that changed a specific line of a file
git log -L 42,42:src/app.ts
```

Conventional Commits Quick Reference

Prefix	Purpose	Example
`feat:`	New feature	`feat(auth): add OAuth2 login flow`
`fix:`	Bug fix	`fix(api): handle null response from server`
`docs:`	Documentation	`docs: update README installation steps`
`style:`	Formatting (no logic change)	`style: format with prettier`
`refactor:`	Code restructuring	`refactor(db): extract query builder`
`perf:`	Performance improvement	`perf(api): cache frequent queries`
`test:`	Adding/updating tests	`test(auth): add login unit tests`
`chore:`	Maintenance	`chore: update dependencies`
`ci:`	CI/CD changes	`ci: add GitHub Actions workflow`
`build:`	Build system	`build: update webpack config`

Next Steps

- Docker CLI Cheat Sheet — Container commands for modern development
- Kubernetes kubectl Cheat Sheet — Essential kubectl commands for orchestration
- Helm CLI Cheat Sheet — Kubernetes package manager reference
- Bash CLI Tools Cheat Sheet — Unix command-line utilities