ITMANAGEMENT101.CO.UK

```
priya@laptop:~/project (main)$ git log --graph
```

The Ultimate Git Cheat Sheet

The commands you'll use most · Undo anything
Branching for one, a team or open source
Every term explained · VS Code · AI

Edition 1 · November 2026

By Linda Garrould · IT Management 101

Scan the code for the latest edition and the full online guide.

Welcome

Git is free version control software that records every change to a project's files, so you can see what changed, undo mistakes and work with other people without overwriting each other. The Git commands you will use most are `git status`, `git add`, `git commit`, `git switch`, `git pull` and `git push`. This booklet gives those and more than 40 others with working examples. It also covers how to undo any commit, which branching strategy suits a solo developer, a weekly-release team or an open-source project, every term you'll meet, the common mistakes, and Git in Visual Studio Code and with AI assistants such as Claude and ChatGPT.

Contents

Start 2
The 20 commands you'll use most

Git basics 4
What Git is, Git vs SVN, setting up, your first project

Branching 9
Strategies for one person, a weekly-release team and open source

Glossary 16
Every term, and why it's a 'pull' request

Commands 19
The full reference, checkout vs switch, commit messages

Undo and mistakes 26
Which undo to use; the mistakes everyone makes

Merging 30
Merge vs rebase, resolving conflicts

Tools and AI 31
VS Code, AI assistants, questions answered

Further reading 34

Index 35
I want to...: forty everyday jobs

Survival card (back cover) 36

How to read it

Commands are shown as you type them. `<hash>` means a commit ID such as a1b2c3d. Where a command can lose work, the note says so.

Printing. Print at A5, or choose two pages per sheet on A4; for a folded booklet, use your printer's booklet setting. Free to print and share for non-commercial use, including in schools, colleges and user groups, under the CC BY-NC-ND 4.0 licence: credit IT Management 101, don't sell it and don't share altered versions. The latest edition is always at itmanagement101.co.uk/git-cheat-sheet. © Garrould Ltd 2026.

The 20 commands you'll use most

These twenty commands cover almost everything you will do in a normal week. The full reference starts on page 19.

Command	What it does
`git init`	Start tracking the current folder
`git clone URL`	Copy a repository and its history
`git status`	What has changed and what is staged
`git add file`	Stage a change (git add . for everything)
`git commit -m "Message"`	Save the staged changes as a snapshot
`git log --oneline --graph`	Compact history
`git diff`	Changes you haven't staged yet
`git switch -c name`	Create a branch and move to it
`git switch main`	Move to another branch
`git branch -m new-name`	Rename the current branch
`git branch -d name`	Delete a merged branch
`git merge name`	Merge a branch into the current one
`git pull`	Fetch and merge others' changes
`git push`	Send your commits to the shared copy
`git stash / git stash pop`	Put work aside, then bring it back
`git restore file`	Throw away unstaged edits to a file
`git reset --soft HEAD~1`	Undo the last commit, keep the changes
`git revert <hash>`	Undo a commit that is already shared
`git cherry-pick <hash>`	Copy one commit onto this branch
`git tag -a v1.2.0 -m "Release"`	Mark a release

What is Git, and why use it?

Git is a version control system: software that keeps the complete history of a set of files, normally a software project's source code. Every time you commit, Git saves a snapshot of the project along with who made the change, when and why. You can compare any two versions, go back to any earlier one, and work on several changes at once on separate branches before combining them.

- **You can undo anything.** Every committed version is kept, so a bad change is a revert away, not a lost afternoon.
- **You can work in parallel.** Each change gets its own branch; half-finished work never disturbs the version customers use.
- **Teams don't overwrite each other.** Git merges everyone's changes and flags the rare clash for a person to decide.
- **You know why the code is the way it is.** Each line can be traced to the commit, and the message, that introduced it.
- **It's the industry standard.** Almost every software job, open-source project, deployment pipeline and AI coding tool assumes Git.
- **It isn't only for code.** Documentation, configuration, infrastructure scripts and website content all benefit from history and review.

Linus Torvalds wrote Git in April 2005, in a matter of weeks, after the Linux kernel project lost its free licence for the commercial tool BitKeeper. He designed it for speed, for very large projects and for thousands of contributors working in parallel. Junio Hamano took over as maintainer that July and still leads it. The name is British slang for an unpleasant person: Torvalds joked that he names all his projects after himself, first Linux, now git.

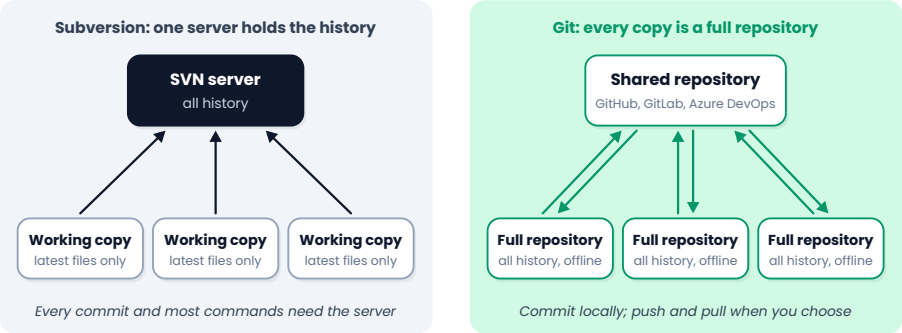
Git vs GitHub

Git is the tool; GitHub is a place to host it. Git runs on your computer and works without any internet connection. GitHub, GitLab, Bitbucket and Azure DevOps are online services that store a shared copy of your repository and add what teams need around it: pull requests, code review, issue tracking, permissions and automated builds. You can use Git with none of them, or with all of them.

Git vs SVN and older systems

The big difference: Subversion (SVN) is centralised, with one server holding the history, while Git is distributed, so every copy is a complete repository. With SVN you check out a working copy and every commit goes straight to the server. With Git you commit locally, as often as you like, and push to a shared repository when you are ready. That one design choice is why Git branches are cheap, why it is fast, and why you can work on a train.

Centralised (SVN) vs distributed (Git)



Year	System
1972	SCCS (Source Code Control System) at Bell Labs: one file at a time
1982	RCS (Revision Control System): still one file at a time, on one machine
1990	CVS (Concurrent Versions System): many files, a central server
2000	Subversion (SVN): a better CVS, with atomic commits
2005	Git and Mercurial: distributed, every copy is a full repository

Git vs SVN at a glance

	Git	Subversion (SVN)
Model	Distributed: every clone has the full history	Centralised: one server holds the history
Work offline	Commit, branch, diff and view history with no network	Commit, log, branch and update need the server; status and diff work offline
Committing	Commit locally, then push when ready	Every commit goes straight to the server
Branches	Cheap pointers; created and merged in seconds	Copies of directories; merging has historically been harder
Version IDs	A hash such as a1b2c3d	A rising number: r1, r2, r3
Partial checkout	Whole repository (sparse checkout is possible)	Any sub-folder on its own
Large binary files	Awkward; Git LFS (Large File Storage) helps, and adds file locking	Handled well, with file locking
Access control	Per repository, set by the hosting service	Can be set per folder
Speed	Most commands are local, so fast	Depends on the network and server

SVN still has its place. Teams with very large binary files that must be locked while someone edits them, with huge repositories where people only need one folder, or that need permissions set per folder, sometimes stay with SVN, or with Perforce in games studios. For almost everything else, Git has won.

SVN commands in Git

SVN	Git	Note
`svn checkout URL`	`git clone URL`	Git copies the whole history, not just the latest files
`svn update`	`git pull`	
`svn commit`	`git commit, then git push`	Two steps: save locally, then share
`svn add / svn delete`	`git add / git rm`	
`svn log`	`git log`	
`svn revert file`	`git restore file`	
`svn copy trunk branches/x`	`git switch -c x`	A Git branch is a label, not a copy
`svn merge`	`git merge`	

Setting up Git

Install Git, tell it who you are, then either start a new repository with `git init` or copy an existing one with `git clone`. On Linux, install it with `sudo apt install git` (Debian, Ubuntu) or `sudo dnf install git` (Fedora, Red Hat); new to the terminal? See the Linux commands cheat sheet. On a Mac, typing `git` offers to install Apple's command-line tools; on Windows, install Git for Windows, which includes Git Bash. Check with `git --version`; at the time of writing the latest release is 2.56, from September 2026.

Once per computer

Command	What it does
`git config --global user.name "Priya Shah"`	Your name on every commit
`git config --global user.email "priya@example.co.uk"`	Your email; match your hosting account
`git config --global init.defaultBranch main`	Call the first branch main
`git config --global core.editor "code --wait"`	Use VS Code for commit messages (or nano). On a Mac, first add the code command: see Git in VS Code
`git config --global pull.rebase false`	git pull merges rather than stopping when branches diverge

Connect to GitHub or GitLab with an SSH (Secure Shell) key rather than a password. Run `ssh-keygen -t ed25519 -C "priya@example.co.uk"` (Azure DevOps accepts only RSA keys: `ssh-keygen -t rsa -b 4096`), then paste the contents of the `.pub` file into your account's SSH keys page. Test it with `ssh -T git@github.com`.

Cloning with an https:// address instead? Your account password won't work: GitHub stopped accepting passwords for Git in August 2021. Git for Windows and GitHub Desktop include Git Credential Manager, which signs you in through the browser; otherwise create a personal access token on the hosting site and paste it when Git asks for a password.

.gitignore

A .gitignore file lists what Git should never track. Put it in place before your first commit: dependency folders such as `node_modules/`, build output, logs, editor settings and, above all, `.env` files and anything holding passwords or keys. GitHub publishes ready-made templates for most languages at github.com/github/gitignore. If a file is already tracked, adding it to .gitignore isn't enough: run `git rm --cached file` too and commit that. Colleagues who pull it will have the file deleted from their copy, so warn them, and back up a .env file first.

Your first project, step by step

1. Create the project and the repository

```
mkdir invoice-tool && cd invoice-tool
git init
```

2. Tell Git what to ignore, before anything else (in Git Bash, macOS or Linux; in PowerShell, create .gitignore in your editor)

printf 'node_modules/\n.env\n*.log\n' > .gitignore

3. Add your first files and commit

```
git add .
git commit -m "Initial commit"
```

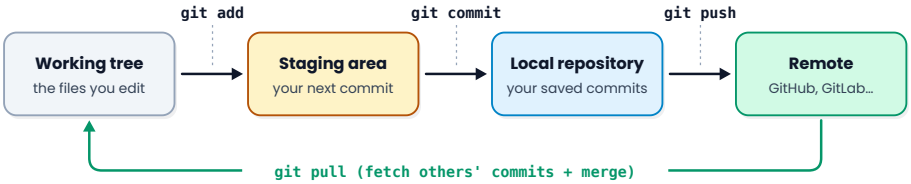
4. Create an empty repository on your hosting service (don't add a README or licence), then connect and push

```
git remote add origin git@github.com:priya/invoice-tool.git
git branch -M main
git push -u origin main
```

5. From now on, the daily loop

```
git switch -c feature/pdf-export
# edit, test...
git add .
git status
git commit -m "Add PDF export"
git push -u origin feature/pdf-export
```

Where your changes live in Git



Branching strategies

A branching strategy is the team's agreement on which branches exist, how long they live and how work reaches production. The right one depends mostly on how often you release. The shortest summary: release continuously and keep branches short-lived (trunk-based development or GitHub flow); release on a schedule and add release branches; support several versions at once and consider GitFlow; accept work from strangers and use forks.

Strategy	How it works	Best for	Watch out
Trunk-based development	Everyone commits to main, or to branches merged within a day or two at most	Teams with good automated tests that deploy often	Needs feature flags and discipline
GitHub flow	Short feature branch, pull request, merge to main, deploy main	Solo developers and small teams shipping continuously	No place for a release in testing
Release branches	Main plus a branch per release; fixes go to main first, then are cherry-picked to the release	Teams on a fixed release rhythm, such as weekly	Every fix needs a second pull request
GitFlow	develop, main, feature, release and hotfix branches	Products with versioned releases supported side by side	Heavy for teams that deploy often
Forking workflow	Contributors work in their own copy and send pull requests	Open-source projects and outside contributors	Contributors must keep forks up to date

GitFlow, described by Vincent Driessen in 2010, was the default for years. In 2020 he added a note to the original article recommending simpler workflows such as GitHub flow for software that is delivered continuously, and keeping GitFlow for explicitly versioned software. Atlassian now describes it as a legacy workflow, and the DORA (DevOps Research and Assessment) studies link short-lived branches merged at least daily with better delivery performance. GitFlow is still worth knowing, because many established codebases use it.

These recommendations follow published practice: Microsoft's Release Flow, trunkbaseddevelopment.com, Martin Fowler's branching patterns, the DORA research, and GitHub's own guidance on GitHub flow and contributing to projects.

Trunk-based development vs GitHub flow vs GitFlow

Trunk-based development means everyone integrates into one branch, main (the "trunk"), at least daily, using branches that live hours or a day or two rather than weeks, and feature flags to hide unfinished work. GitHub flow is the same idea with a pull request on every change. GitFlow adds long-lived develop, release and hotfix branches, and suits software that ships numbered versions supported side by side.

Four questions settle most choices:

If the answer is yes to...	Use
Do you deploy every time something merges?	Trunk-based development or GitHub flow
Do you release on a schedule, such as weekly?	Add a release branch per release (as in the team example below)
Do customers run several versions that you patch?	GitFlow, or long-lived release branches per version
Do strangers contribute?	Forking workflow with pull requests

A solo developer

Working alone, keep it simple: main is the version that works. Use a short branch for anything that takes more than one sitting or might not work out; a small, tested fix can go straight onto main. Either way, run your tests before main moves on. Push to a hosting service even if nobody else will see it: it is also your off-site backup.

1. Start anything bigger than a quick fix on a short branch

```
git switch -c fix/vat-rounding
```

2. Commit as you go, in small steps

```
git commit -am "Round VAT per line, not per invoice"
```

3. Merge back when it works, then delete the branch

```
git switch main
git merge fix/vat-rounding
git branch -d fix/vat-rounding
```

4. Tag what you release, so you can always get back to it

```
git tag -a v1.3.0 -m "Release 1.3.0"
git push origin main v1.3.0
```

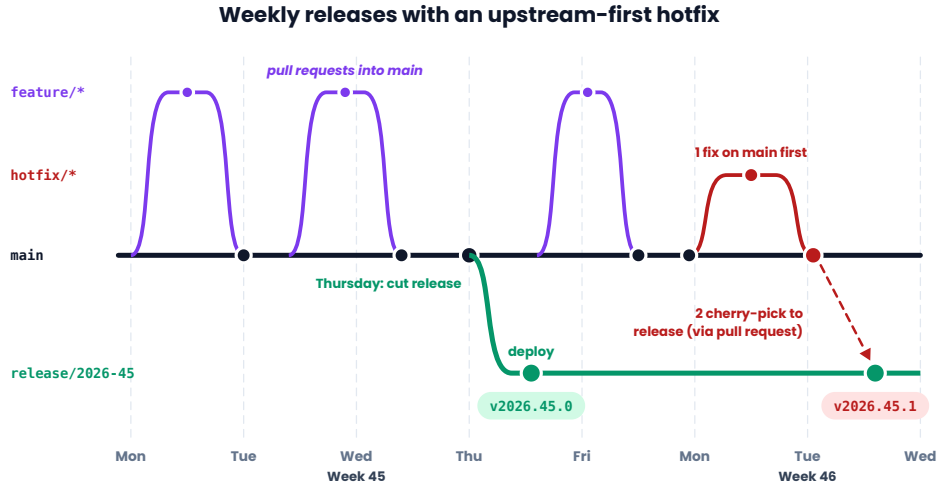
A team of ten, releasing weekly

For a team of around ten working in weekly sprints, releasing to production every week and occasionally needing an urgent fix, use short-lived feature branches into main plus a release branch per week. This is trunk-based development with release branches, the pattern Microsoft calls Release Flow. Release branches are never merged back into main. Every fix is made on main first and then cherry-picked into the release branch ("upstream first"), so a fix can't be forgotten in next week's release.

- **main is always releasable.** Nobody commits to it directly: protect it in GitHub, GitLab or Azure DevOps so every change arrives by pull request with at least one approval, passing status checks and force pushes blocked. A CODEOWNERS file can require review from the people who own the code being changed.
- **One branch per ticket, merged within a day or two.** Name it after the ticket: `feature/PAY-142-card-refunds`. Small branches mean small reviews and few conflicts.
- **Cut a release branch on a fixed day.** On Thursday morning, `release/2026-45` (the year and week number) is created from main, deployed to staging, tested and released to production that afternoon. main carries straight on with next week's work.
- **Tag every production release.** `v2026.45.0` marks exactly what went live, so anyone can rebuild it.
- **Tidy up after each release.** Only the release manager creates `release/*` branches. Once the next week's release is live, delete the old release branch; its tag keeps the code.
- **Hotfixes go to main first, then to the release.** Fix the bug on a branch from main, merge it, then cherry-pick that one commit into the live release branch through a second pull request. Only the fix travels, not next week's untested work. The exception: if the bug no longer exists on main, or main has changed so much that the fix doesn't apply, fix the release branch directly and record on the ticket why main needs no change, or make the matching fix on main the same day.

If a bug is found in production the following Monday, the fix is merged into main and cherry-picked into release/2026-45, which is released as v2026.45.1 on Tuesday. Thursday's release/2026-46 is cut from main, so it already has the fix.

The week, and an upstream-first hotfix



The hotfix, step by step

1. Branch from main and reproduce the bug with a failing test

```
git fetch origin
git switch -c hotfix/PAY-160-refund-crash origin/main
```

2. Fix, commit, push and open a pull request into main

```
git commit -am "Fix crash when refunding a zero-value line"
git push -u origin hotfix/PAY-160-refund-crash
```

3. After review and passing checks, merge it, then cherry-pick the fix onto a branch from the live release

```
git fetch origin
git log --oneline -5 origin/main
git switch -c hotfix/PAY-160-to-2026-45 origin/release/2026-45
git cherry-pick -x <fix-commit-hash>
git push -u origin hotfix/PAY-160-to-2026-45
```

4. Open a pull request into release/2026-45 (change the base branch: it defaults to main). After review, merge, tag and deploy

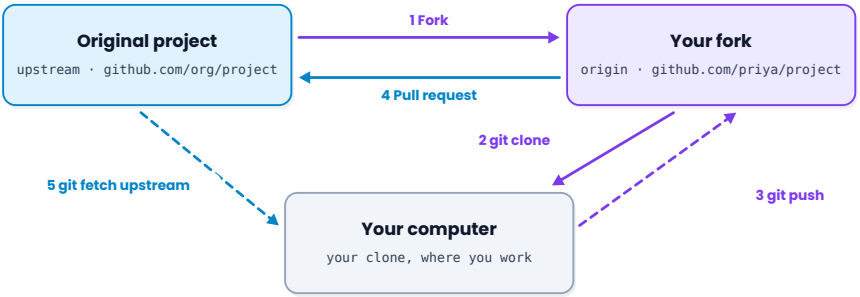
```
git switch release/2026-45
git pull
git tag -a v2026.45.1 -m "Hotfix PAY-160"
git push origin v2026.45.1
```

Would GitFlow work for this team? It would, but its long-lived develop branch and separate release and hotfix merges add ceremony that a team deploying weekly rarely needs. Choose it if you must keep, say, versions 3 and 4 of a product supported side by side for different customers.

Open source: forks and your first contribution

Open-source projects use a forking workflow: you can't push to their repository, so you work in your own copy and send them a pull request. It is how most people contribute to projects whose maintainers they have never met, and a good first contribution (a documentation fix, a test, a small bug fix) is the best way to learn it.

Contributing to an open-source project



1. Read the rules first
   Look for CONTRIBUTING.md, the code of conduct and issues labelled 'good first issue', and search open and closed issues in case it has been discussed. For anything bigger than a typo, comment on the issue before you start, so nobody duplicates your work.
2. Fork the project
   On GitHub, press Fork. You now have your own copy, at github.com/priya/project, that you can push to.
3. Clone your fork and add the original as 'upstream'

```
git clone git@github.com:priya/project.git
cd project
git remote add upstream https://github.com/original-org/project.git
```

Open source, continued

4. Branch from the latest upstream main (never work on your fork's main, which you keep as a clean copy of upstream)

```
git fetch upstream
git switch -c fix/typo-in-install-guide upstream/main
```

5. Make the change, following the project's style and tests

```
git add docs/install.md
git commit -s -m "docs: fix typo in install guide"
```

6. Push to your fork and open a pull request: say what it changes and why, add 'Closes #123' and tick Allow edits from maintainers

```
git push -u origin fix/typo-in-install-guide
```

7. Respond to review by pushing more commits to the same branch

```
git commit -am "docs: reword per review"
git push
```

8. Keep your fork current for next time (or press Sync fork on GitHub)

```
git switch main
git fetch upstream
git merge --ff-only upstream/main
git push
```

- **Many projects require a sign-off or an agreement.** `git commit -s` adds the Signed-off-by line, with your real name and email, used by the Linux kernel and others under the DCO (Developer Certificate of Origin); some projects ask you to sign a CLA (Contributor Licence Agreement) instead. `-s` is a statement that you have the right to submit the work, not a cryptographic signature (that is `-S`).
- **One change per pull request.** A focused pull request (PR) with a clear description gets reviewed; a sprawling one waits.
- **Be patient and polite.** Maintainers are often volunteers. If a week passes without a reply, a friendly comment in the same thread is fine; don't chase them privately.
- **Run the tests before you push.** CI (continuous integration) will run them anyway, and a red build is the quickest way to be ignored.

Git terms explained

Git has its own vocabulary, and most confusion comes from it. These are the terms you'll meet in commands, error messages and code reviews, in plain English.

Term	What it means
Repository (repo)	A project folder that Git tracks, with its entire history stored in a hidden .git folder.
Working tree	The files you can see and edit: your current version of the project.
Staging area (index)	The list of changes that will go into the next commit. git add puts changes here.
Commit	A saved snapshot of the staged changes, with a message, an author, a date and a link to the commit before it.
Hash (SHA, Secure Hash Algorithm)	The unique ID of a commit, such as a1b2c3d (the short form of a 40-character code). Git calculates it from the commit's contents, so any change gives a new ID. Git uses SHA-1 by default; SHA-256 is available and planned to become the default in Git 3.0.
HEAD	Git's pointer to where you are now: usually the latest commit on the branch you have checked out.
Detached HEAD	You are looking at a specific commit rather than a branch. New commits here are easy to lose; create a branch (git switch -c) to keep them.
Branch	A movable label pointing at a commit, used for a separate line of work. Creating one is instant.
main (formerly master)	The usual name of the primary branch. GitHub switched its default from master to main in October 2020.
Tag	A fixed label on a commit, usually marking a release. Annotated tags (-a) also store a message and author.
Remote	Another copy of the repository that you push to and pull from, usually on a hosting service.
origin	The default name Git gives the remote you cloned from.
upstream	Three meanings: the original project your fork came from; the remote branch your local branch tracks; and, in 'upstream first', main as opposed to a release branch.
Clone	A full copy of a repository, history included, made with git clone.

Git terms, continued

Term	What it means
Fork	Your own server-side copy of someone else's repository, so you can push changes without write access to theirs. A hosting feature, not a Git command.
Fetch	Download new commits from a remote without changing your files.
Pull	Fetch, then merge (or rebase) the new commits into your branch.
Push	Upload your commits to a remote branch.
Merge	Combine another branch's work into the current branch.
Fast-forward	A merge where your branch has no new commits of its own, so Git moves the label forward with no merge commit.
Merge commit	A commit with two parents, recording where two branches joined.
Rebase	Replay your commits on top of another branch, giving a straight-line history. It rewrites commit IDs, so don't rebase shared branches.
Squash	Combine several commits into one, often when a pull request is merged.
Cherry-pick	Copy a single commit from one branch to another.
Stash	A shelf for uncommitted changes, so you can switch branch and come back to them.
Conflict	Two branches changed the same lines differently. Git marks the clash and asks you to choose.
Diff	The line-by-line difference between two versions.
Reflog	Git's private record of everywhere HEAD has been: kept for 90 days by default, and 30 days for commits no branch points to. The way back after most mistakes.

Git terms, continued

Term	What it means
.gitignore	A file listing what Git should never track: build output, logs, .env files, node_modules.
Pull request (PR)	A request to merge one branch into another, with a page for reviewing, discussing and testing the change first. See below for why it is called a pull.
Merge request (MR)	GitLab's name for a pull request. It does exactly the same job.
Code review	Colleagues reading a pull request's changes and approving them or asking for changes.
CI (continuous integration)	Automated builds and tests that run on every push or pull request.
Protected branch	A branch, usually main, that can only change through approved pull requests.
Git LFS (Large File Storage)	An extension that keeps large files such as videos and design files outside the normal history.
Hook	A script Git runs automatically at a point such as before a commit, for example to run tests or a formatter.
Submodule	A repository nested inside another at a fixed commit. Powerful, and easy to get wrong.
Trunk	Another name for main: the branch everyone integrates into.
Feature flag	A setting that hides unfinished code in production, so it can be merged before it is ready.

Why is it called a pull request?

A pull request asks the maintainers of a repository to merge your branch, with a page for review, discussion and automated checks before it lands. It is called a pull request because in Git you cannot push into someone else's repository; they have to pull your changes into theirs. Since July 2005 Git has had a `git request-pull` command, which writes a summary email asking a maintainer to pull from your copy: that is how Linux kernel maintainers still send their work up to Linus Torvalds. In February 2008 GitHub added a Pull Request button that sent the request as a message; by 2010 it had become a web page with the changes and a discussion, and reviews and automated checks followed. GitHub kept the name. GitLab, launched in 2011, chose "merge request", describing what happens rather than who does it. They are the same thing.

Git commands: the full reference

git config: set up and configure

`git config --global user.name` Set the name recorded on your commits

```
git config --global user.name "Priya Shah"
```

`git config --global user.email` Set the email recorded on your commits

```
git config --global user.email "priya@example.co.uk"
```

Use the address your hosting service knows

`git config --global init.defaultBranch` Name the first branch of new repositories

```
git config --global init.defaultBranch main
```

Git 2.28 or later

`git config --global pull.rebase` Choose what git pull does when branches have diverged

```
git config --global pull.rebase false
```

false merges; true rebases

`git config --list --show-origin` Show every setting and the file it came from

```
git config --list --show-origin
```

`git help` Open the manual for any command

```
git help rebase
```

git rebase -h for a short summary

Start a project

`git init` Turn the current folder into a repository

```
git init
```

Creates a hidden .git folder

`git clone` Copy a repository, its branches and its history

```
git clone git@github.com:acme/website.git
```

Add a folder name to choose where it goes

`git remote -v` List the remote repositories this one talks to

```
git remote -v
```

`git remote add` Connect a remote

```
git remote add origin git@github.com:priya/website.git
```

Everyday changes

`git status` Show changed, staged and untracked files

```
git status -s
```

-s for the short format

`git add` Stage changes for the next commit

```
git add src/login.js
```

git add -p lets you pick parts of a file

`git commit` Save the staged changes with a message

```
git commit -m "Fix login timeout on slow connections"
```

Leave out -m to write a longer message in your editor

`git commit -a` Stage every tracked file and commit in one step

```
git commit -am "Update copyright year"
```

New files still need git add

`git diff` Show changes not yet staged

```
git diff
```

git diff --staged shows what will be committed

`git rm` Delete a file and stage the deletion

```
git rm old-notes.txt
```

--cached stops tracking but keeps the file; -r for a folder

`git mv` Rename or move a file and stage it

```
git mv Readme.md README.md
```

The safe way to change a name's case on Mac and Windows

Branches

`git branch` List local branches; -a includes remote ones

```
git branch -a
```

`git switch -c` Create a branch and move to it

```
git switch -c feature/invoice-pdf
```

Older equivalent: git checkout -b

`git switch` Move to an existing branch

```
git switch main
```

Commit or stash your changes first

`git branch -m` Rename a branch

```
git branch -m old-name new-name
```

On the current branch: git branch -m new-name

`git branch -d` Delete a branch that has been merged

```
git branch -d feature/invoice-pdf
```

-D forces it, even if unmerged

`git push origin --delete` Delete a branch on the remote

```
git push origin --delete feature/invoice-pdf
```

`git merge` Bring another branch's commits into this one

```
git merge feature/invoice-pdf
```

--no-ff always records a merge commit

`git rebase` Replay your commits on top of another branch

```
git rebase main
```

Only on branches nobody else uses

Sharing with others

`git fetch` Download others' commits without changing your work

```
git fetch origin
```

Safe at any time

`git pull` Fetch, then merge (or rebase) into your branch

```
git pull
```

git fetch followed by git merge

`git push` Send your commits to the remote

```
git push
```

First time on a branch: git push -u origin feature/x

`git push --force-with-lease` Overwrite a remote branch, but only if nobody else has pushed to it

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

Never plain --force on a shared branch. Editors that fetch in the background (VS Code does) can let the lease pass anyway; add --force-if-includes (Git 2.30+) for full protection.

History and investigation

`git log` Show commit history

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

-p shows each commit's changes

`git show` Show one commit in full

```
git show a1b2c3d
```

`git blame` Show who last changed each line, and when

```
git blame -L 40,60 src/login.js
```

For context, not for blame

`git log -S` Find the commits that added or removed some text

```
git log -S "TIMEOUT_MS" --oneline
```

`git bisect` Binary-search history for the commit that broke something

```
git bisect start; git bisect bad; git bisect good v1.4.0
```

Then mark each step good or bad; git bisect reset when you have finished

`git reflog` Every position HEAD has been in: the safety net

```
git reflog
```

Recover 'lost' commits from here

Undo and fix

`git restore` Discard unstaged edits to a file

```
git restore src/login.js
```

Cannot be undone

`git restore --staged` Unstage a file but keep the edits

```
git restore --staged src/login.js
```

`git commit --amend` Change the last commit's message or add a forgotten file

```
git commit --amend
```

Not yet pushed only

`git reset --soft` Undo commits but keep their changes staged

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

Not yet pushed only

`git reset --hard` Throw away commits and all uncommitted edits

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

Destructive

`git revert` Add a new commit that reverses an earlier one

```
git revert a1b2c3d
```

The safe undo for shared history

`git clean` Delete untracked files

```
git clean -n
```

-n previews; -f actually deletes

git stash, cherry-pick and tag

`git stash` Put uncommitted changes aside

```
git stash push -m "half-done invoice layout"
```

-u includes untracked files

`git stash list` See what is stashed

```
git stash list
```

`git stash pop` Bring the latest stash back and remove it from the list

```
git stash pop
```

git stash apply keeps it in the list

`git cherry-pick` Copy one commit onto the current branch

```
git cherry-pick a1b2c3d
```

Ideal for back-porting a hotfix

`git tag` Mark a commit, usually a release

```
git tag -a v2.4.0 -m "Release 2.4.0"
```

Push it with git push origin v2.4.0

Power moves

`git rebase -i` Squash, reword or reorder your recent commits

```
git rebase -i HEAD~3
```

Not yet pushed only; change pick to squash or fixup

`git commit --fixup` Record a fix to an earlier commit, to be folded in later

```
git commit --fixup a1b2c3d
```

Then git rebase -i --autosquash main

`git switch -` Jump back to the branch you were on before

```
git switch -
```

`git pull --rebase` Replay your local commits on top of others' instead of merging

```
git pull --rebase
```

Keeps history straight on your own branch

`git worktree add` Check out a second branch in another folder, at the same time

```
git worktree add -b hotfix/PAY-160 ../hotfix origin/main
```

Start a hotfix in another folder without stashing

`git config alias` Make your own short commands

```
git config --global alias.lg "log --oneline --graph --all"
```

Then git lg

`git remote set-url` Change where a remote points, for example from an https:// address to SSH

```
git remote set-url origin git@github.com:priya/website.git
```

`git log --author --since` Filter history by person and date

```
git log --author="Priya" --since="2 weeks ago" --oneline
```

checkout, switch and restore

`git checkout` used to do two unrelated jobs: moving between branches and throwing away changes to files. Git 2.23 split them into `git switch` and `git restore`, which are harder to misuse. Both versions work:

Older	Newer	What it does
`git checkout branch`	`git switch branch`	Move to a branch
`git checkout -b new`	`git switch -c new`	Create a branch and move to it
`git checkout -- file`	`git restore file`	Discard unstaged edits
`git reset HEAD file`	`git restore --staged file`	Unstage a file
`git checkout a1b2c3d`	`git switch --detach a1b2c3d`	Look at an old commit (detached HEAD)

Renaming and deleting branches

Renaming or deleting a branch that is already on the remote takes two steps: change it locally, then update the remote. To rename: `git branch -m old-name new-name`, `git push -u origin new-name`, then `git push origin --delete old-name`. To delete: `git branch -d name` and `git push origin --delete name`. Colleagues' copies keep the old name as origin/old-name until they run `git fetch --prune`.

Writing good commit messages

A good commit message has a short summary line in the imperative mood, such as "Fix login timeout on slow connections", then a blank line and a few lines on why, if it isn't obvious. Keep the summary under about 72 characters so it reads well in `git log --oneline` and on GitHub.

Conventional Commits is a widely used convention that adds a type to the start of the summary. It makes history scannable and lets tools generate release notes and version numbers automatically. A `!` after the type, or a `BREAKING CHANGE:` footer, marks a change that breaks compatibility.

Type	Use it for	Example
`feat`	A new feature	`feat(invoices): add PDF download`
`fix`	A bug fix	`fix(login): retry once on network timeout`
`docs`	Documentation only	`docs: explain the release process`
`refactor`	Code change that neither fixes nor adds	`refactor: split payment service`
`test`	Adding or fixing tests	`test: cover refunds of zero-value lines`
`chore`	Housekeeping: builds, dependencies	`chore: update dependencies`

How to undo things

To undo the last commit and keep the work: `git reset --soft HEAD~1`. `HEAD~1` means one commit before where you are now; `HEAD~3` would be three. If that commit is already pushed, run `git revert HEAD` instead.

How you undo a commit depends on one question: has anyone else got it yet? If you haven't pushed, you can rewrite your own history with `git reset` or `git commit --amend`. If you have, add a new commit that reverses it with `git revert`, so nobody else's copy is disturbed.

What happened	Command	Watch out
Edited a file and want the last committed version back	`git restore file`	Your edits are gone for good
Staged something by mistake	`git restore --staged file`	Your edits stay in the file
Last commit's message is wrong, or a file is missing from it	`git commit --amend`	Not yet pushed only
Undo the last commit, keep the work	`git reset --soft HEAD~1`	Not yet pushed only
Undo the last commit and unstage everything	`git reset HEAD~1`	Same as --mixed; edits stay in your files
Throw away the last commit and every uncommitted edit	`git reset --hard HEAD~1`	Destructive: check git status first
Make my branch match the remote exactly, discarding local commits and edits	`git fetch, then git reset --hard origin/main`	Destructive: copy anything you want to keep onto another branch first
Undo a commit that is already pushed	`git revert <hash>`	Safe: adds a new commit
Undo a merge that is already pushed	`git revert -m 1 <merge-hash>`	-m 1 keeps the branch you merged into. Merging the same branch again later won't bring the changes back; revert the revert first
Committed to main instead of a branch (not pushed)	`git switch -c new-branch, then git branch -f main origin/main`	Your commit stays on new-branch and main goes back to match the remote; uncommitted edits are untouched
Deleted a branch or lost commits	`git reflog, then git switch -c rescue <hash>`	Act within 30 days
Pushed a password or key	`Revoke or change it immediately`	Removing it from history is not enough

Common mistakes, and the fix

Nearly everyone makes these, usually in their first year. Each has a fix, and most are prevented by a habit or a setting.

Mistake	Why it hurts	The fix
Committing a password, API (application programming interface) key or .env file	Anyone with the repository has it, in every copy, forever.	Revoke the secret at once, then remove it from history (git filter-repo, a separate install) if needed. Add .env to .gitignore and turn on your host's secret scanning.
Committing straight to main	Unreviewed changes reach everyone, and main may stop working.	Protect main so changes arrive by pull request. If you have not pushed: git switch -c my-work, then git branch -f main origin/main.
Force-pushing a shared branch	It rewrites history under your colleagues' feet and can delete their work.	Use git push --force-with-lease, and only on your own branches. Never on main.
Rebasing commits others already have	Their copies no longer match; merging them creates duplicates.	Rebase only local or personal branches. Merge or revert on shared ones.
Starting work without pulling first	You build on an old version and meet conflicts later.	git pull (or git fetch and look) before you start, and keep branches short.
One giant commit at the end of the week	Impossible to review, hard to revert, useless in history.	Commit each logical step. git add -p stages part of a file.
Messages like 'fix' or 'updates'	Six months later nobody knows what changed or why.	A short summary in the imperative mood, plus why if it isn't obvious. Consider Conventional Commits.

Common mistakes, continued

Mistake	Why it hurts	The fix
git add . with no .gitignore	Build files, logs and node_modules end up in history.	Create a .gitignore first; github.com/github/gitignore has templates for most languages.
git reset --hard with uncommitted work	Uncommitted edits are deleted with no way back.	Run git status first. git stash is the safe way to clear your working tree.
Panicking in a merge conflict	Half-resolved files with <<<<<<< markers get committed.	Read the markers, choose, test, then git add and git commit. git merge --abort starts again.
Long-lived feature branches	The longer a branch lives, the worse the merge.	Merge to main every day or two; hide unfinished features behind a feature flag.
Large binary files in history	Clones become slow and huge, permanently.	Use Git LFS (Large File Storage) for videos, design files and datasets, or keep them out of Git.
Mixed line endings	Every line shows as changed when Windows and Mac users share files.	Commit a .gitattributes file containing * text=auto, then run git add --renormalize . once and commit. It applies to everyone, unlike core.autocrlf (which Git for Windows already sets).
Renaming a file by changing its case only	Mac and Windows ignore the change, so Git never sees it.	git mv readme.md README.md
Committing in detached HEAD	Those commits belong to no branch and are easily lost.	git switch -c new-branch before you leave, or find them later in git reflog.
Forgetting what you stashed	Stashes pile up and get dropped by accident.	Name them (git stash push -m) and clear them out with git stash list.

Git error messages decoded

Git's error messages are precise but terse. These are the ones you are most likely to meet, quoted as Git prints them, with what they mean and what to do.

Git says	It means	Do this
`Your branch is ahead of 'origin/main' by 1 commit`	You have commits that aren't on the remote yet.	git push
`Your branch is behind 'origin/main' by 2 commits`	Others have pushed work you don't have.	git pull
`Updates were rejected because the remote contains work that you do not have locally`	Someone pushed first.	git pull, fix any conflicts, then git push. Never --force to make it go away
`fatal: refusing to merge unrelated histories`	The remote was created with its own first commit, usually a README.	Once: git pull origin main --allow-unrelated-histories, then git push
`error: src refspec main does not match any`	Your branch is called master, or you haven't committed yet.	git branch -M main, after at least one commit
`You are in 'detached HEAD' state`	You checked out a commit, not a branch.	git switch -c new-branch to keep any work, or git switch main
`Please commit your changes or stash them before you switch branches`	Uncommitted edits would be overwritten.	git stash, switch, then git stash pop
`CONFLICT (content): Merge conflict in file`	Two branches changed the same lines.	See resolving a merge conflict
`fatal: Need to specify how to reconcile divergent branches`	Both you and the remote have new commits, and Git doesn't know whether to merge or rebase.	git config --global pull.rebase false once, then git pull
`Permission denied (publickey)`	The server doesn't recognise your SSH key.	Add your .pub key to your account; test with ssh -T git@github.com
`fatal: not a git repository (or any of the parent directories)`	You are not inside a repository folder.	cd into the project, or git init
`warning: in the working copy of 'file', LF will be replaced by CRLF the next time Git touches it`	Git is converting line endings for Windows.	Usually harmless; add a .gitattributes file to make it consistent
`fatal: The current branch feature/x has no upstream branch`	Git doesn't know where to push a branch it hasn't pushed before.	git push -u origin feature/x (once; plain git push works after that)
`error: Your local changes to the following files would be overwritten by merge`	git pull would overwrite edits you haven't committed.	Commit them, or git stash, git pull, then git stash pop

Merge, rebase and conflicts

Merge and rebase both bring two lines of work together; the difference is what the history looks like afterwards and whether commits are rewritten. The safe rule: merge into shared branches, rebase only your own.

	git merge	git rebase
What it does	Joins two branches with a merge commit	Replays your commits on top of the other branch
History	Shows exactly what happened, including the branching	A straight line, as if the work happened in order
Commit IDs	Unchanged	Rewritten (new hashes)
Safe on shared branches?	Yes	No: only on branches nobody else uses
Conflicts	Resolved once, in the merge	May need resolving once per replayed commit
Use it when	Bringing finished work into main	Tidying your own branch before a pull request

Squash merging turns all the commits on a pull request into one when it is merged, which keeps main's history tidy. The catch: Git no longer sees the original branch as merged, so delete it with `git branch -D`.

Resolving a merge conflict

A merge conflict means two branches changed the same lines in different ways, and Git needs a person to decide. It is routine, not an error. Git marks the clash in the file like this:

```
<<<<<<< HEAD
const TIMEOUT_MS = 30000;
=======
const TIMEOUT_MS = 45000;
>>>>>>> feature/slow-networks
```

1. Run `git status` to see which files conflict.
2. Open each file. Between `<<<<<<< HEAD` and `=======` is your branch's version; between `=======` and `>>>>>>>` is the incoming one. During a rebase the sides are reversed: HEAD is the branch you are rebasing onto.
3. Edit the file to the correct result, which may be one side, the other, or a mix, and delete all three marker lines.
4. Test it, then `git add` the file. When every file is resolved, `git commit` (or `git rebase --continue` during a rebase).
5. Lost? `git merge --abort` puts everything back as it was before the merge (`git rebase --abort` during a rebase).

Git in Visual Studio Code

Visual Studio Code has Git built in: open a folder that is a repository and the Source Control view shows every change, ready to stage, commit, push and pull without typing a command. It uses the Git installed on your computer, so install Git first. The terminal is still there (Ctrl+` opens it) when you need a command the buttons don't offer.

To	In VS Code
Open Source Control	Ctrl+Shift+G (Control+Shift+G on a Mac too) shows every changed file.
See a change	Click a file to open a side-by-side diff of what you changed.
Stage	Hover over a file and click +; or stage selected lines from the diff.
Commit	Type a message in the box at the top and press Commit (Ctrl+Enter, or Cmd+Enter on a Mac).
Push and pull	Sync Changes pushes and pulls in one go; the ... menu has each separately.
Switch or create a branch	Click the branch name in the bottom-left status bar.
Resolve a conflict	Conflicted files open with Accept Current, Accept Incoming and Accept Both above each clash, or in the three-way merge editor.
See history	The Source Control Graph shows branches and commits; the Timeline view shows one file's history.

To make VS Code the editor Git opens for commit messages and rebases, run `git config --global core.editor "code --wait"` (on a Mac, first run Shell Command: Install 'code' command in PATH from the Command Palette). GitHub Copilot, if you use it, adds a button in the commit message box that drafts a message from your changes; read it before you commit.

Three extensions worth knowing

Extension	What it adds
GitHub Pull Requests	Review, comment on and merge pull requests without leaving the editor (GitLab Workflow does the same for GitLab).
GitLens	Inline blame, rich history and comparisons; the free tier covers most needs.
Code for IBM i	Edit IBM i source in VS Code; source kept as stream files in the IFS (Integrated File System) can be managed with Git.

Git with Claude, ChatGPT and AI

AI assistants such as Claude, ChatGPT and GitHub Copilot are useful with Git: they explain commands, decode error messages, draft commit messages and help untangle conflicts. They are also confidently wrong sometimes, and Git has commands that destroy work. Use them as a knowledgeable colleague whose advice you check, not as an operator with your keys.

Do	How
Ask it to explain before you run	Paste a command you found and ask what it does, what it changes and how to undo it.
Paste the error and git status, not your secrets	The exact message plus `git status` output gives it enough to diagnose most problems. Remove passwords, keys and customer data first.
Ask for the safest option	Add 'my branch is shared with others' or 'nothing is pushed yet' to your question; the right answer often depends on it.
Draft commit messages from the diff	Give it `git diff --staged` and ask for a Conventional Commits message. Read it before you commit: it can only describe what it sees, not why.
Untangle a conflict	Paste the conflicted block and say what each side was meant to do. Check the result runs.
Review a pull request	Ask what could break, what is untested and what a reviewer would question. Treat it as a second reviewer, not the only one.
Generate a .gitignore or a hook	It is good at boilerplate for a given language or framework.

AI: what not to do, and good prompts

- Don't run a destructive command (`reset --hard`, `push --force`, `clean -f`, `filter-repo`) because an assistant suggested it. Check the reflog and your remote first.
- Don't paste secrets, customer data or code you are not allowed to share. Check your organisation's rules on which AI services may see its code.
- Don't trust invented flags. Assistants sometimes produce options that don't exist; `git help <command>` settles it.
- Don't let an AI agent push to main. Agents that can run Git themselves, such as Claude Code, GitHub Copilot's agent mode and OpenAI's Codex, should work on their own branch and open a pull request that a person reviews.

Explain safely

```
I'm on a feature branch that I've already pushed and a colleague has pulled. I want
to undo my last two commits. What's the safest way, and what will it do to my
colleague's copy?
```

Commit message

```
Here is my staged diff. Write a Conventional Commits message: a summary line under
72 characters and a short body explaining why. Don't invent reasons that aren't in
the diff.
```

Rescue

```
Here is my git reflog. I think I lost a commit called 'Add refund export' after a
reset. Which entry is it, and how do I get it back on a new branch?
```

Agents that run Git for you. Coding agents can now edit files, run tests and make commits on their own. Set them up the way you would a new starter: their own branch, no permission to push to main or force-push, and every change arriving as a pull request a person reviews. Start one on a new branch from a clean commit, so `git diff main` shows exactly what it changed and deleting the branch undoes it.

Questions answered

What is Git in simple terms?
Git is version control: it records every saved change to a set of files, who made it and why, so you can see history, undo mistakes and work alongside other people without overwriting each other. It runs on your own computer and is free and open source.

What is the difference between Git and GitHub?
Git is the version control tool that runs on your computer. GitHub is a website that hosts Git repositories and adds pull requests, issue tracking, automation and access control. GitLab, Bitbucket and Azure DevOps do the same job as GitHub.

How do I undo the last commit in Git?
If you haven't pushed it, git reset --soft HEAD~1 removes the commit and keeps your changes staged. If you have pushed it, use git revert followed by the commit's hash, which adds a new commit that reverses it without rewriting shared history.

What is the difference between git fetch and git pull?
git fetch downloads new commits from the remote but leaves your files and branches alone, so you can look first. git pull does a fetch and then merges (or rebases) those commits into your current branch.

Should I use merge or rebase?
Merge to bring finished work into a shared branch such as main; it keeps the real history and never rewrites commits. Rebase to tidy your own unpublished branch before opening a pull request. Never rebase commits other people already have.

What is trunk-based development?
A branching strategy in which everyone merges small changes into main at least once a day, instead of keeping long-lived feature branches. Unfinished work is hidden behind feature flags, and automated tests keep main releasable at all times.

Further reading

- Pro Git, the free official book: `git-scm.com/book/en/v2`
- Git reference manual: `git-scm.com/docs`
- GitHub's .gitignore templates: `github.com/github/gitignore`
- Conventional Commits: `www.conventionalcommits.org/`
- Trunk-based development: `trunkbaseddevelopment.com/`
- Version control in VS Code: `code.visualstudio.com/docs/sourcecontrol/overview`
- Open Source Guides: how to contribute: `opensource.guide/how-to-contribute/`
- Microsoft: how we develop with Release Flow: `learn.microsoft.com`
- Martin Fowler: patterns for managing source code branches: `martinfowler.com/articles/branching-patterns.html`
- The Linux commands cheat sheet: `itmanagement101.co.uk/linux-commands-notebook`

Index: I want to...

Forty everyday jobs, the command for each, and the page that explains it.

I want to...	Command	Page
Start tracking a folder	`git init`	19
Copy an existing project	`git clone URL`	19
See what has changed	`git status`	20
Stage part of a file	`git add -p`	20
Save my work	`git commit -m "..."`	20
Fix my last commit	`git commit --amend`	22
Undo my last commit (not pushed)	`git reset --soft HEAD~1`	22
Undo a pushed commit	`git revert <hash>`	22
Undo a pushed merge	`git revert -m 1 <hash>`	26
Throw away edits to a file	`git restore file`	22
Unstage a file	`git restore --staged file`	22
Create a branch	`git switch -c name`	21
Rename a branch	`git branch -m new`	21
Delete a branch	`git branch -d name`	21
Delete a branch on the remote	`git push origin --delete name`	21
Get my colleagues' changes	`git pull`	21
Look before I pull	`git fetch`	21
Share my branch the first time	`git push -u origin name`	21
Combine a branch into main	`git merge name`	21
Squash or tidy my commits	`git rebase -i HEAD~3`	24
Copy a fix to another branch	`git cherry-pick <hash>`	23
Put work aside	`git stash`	23
Mark a release	`git tag -a v1.0 -m ...`	23
See the history	`git log --oneline --graph`	22
Find who changed a line	`git blame file`	22
Find when a bug arrived	`git bisect`	22
Recover lost work	`git reflog`	22
Stop tracking a file, keep it	`git rm --cached file`	7
Ignore files	`.gitignore`	7
Change a file name's case	`git mv old New`	20
Abandon a merge	`git merge --abort`	30
Force-push safely	`git push --force-with-lease`	21
Contribute to open source	`git remote add upstream`	14
Keep my fork current	`git fetch upstream`	15
Write a clear commit message	`feat(scope): summary`	25
Use VS Code for commit messages	`core.editor "code --wait"`	7
Set my name and email	`git config --global user.name`	19
Start a hotfix	`git switch -c hotfix/x origin/main`	13
Move a commit off main (not pushed)	`git switch -c new; git branch -f main origin/main`	26
Make my branch match the remote	`git reset --hard origin/main`	26

PIN IT UP

Git survival card

24 commands to keep within reach

`git status`	what's changed?	`git merge name`	join a branch
`git add -p`	stage part of a file	`git stash / pop`	put work aside
`git commit -m "msg"`	save a snapshot	`git cherry-pick <hash>`	copy a commit
`git log --oneline`	short history	`git restore file`	discard edits
`git diff --staged`	what will be committed	`git restore --staged file`	unstage
`git switch -c name`	new branch	`git commit --amend`	fix last commit
`git switch main`	change branch	`git reset --soft HEAD~1`	undo commit
`git branch -m new`	rename branch	`git revert <hash>`	undo if shared
`git branch -d name`	delete branch	`git reflog`	find lost work
`git pull`	get others' work	`git tag -a v1.0 -m "v1.0"`	tag it
`git push -u origin name`	first push	`git blame file`	who changed this line
`git fetch`	look before you pull	`git push --force-with-lease`	safer

Undo: has anyone else got it?

No: `git reset --soft HEAD~1` or `git commit --amend`
Yes: `git revert <hash>`, never rewrite shared history
Lost something: `git reflog`

Five rules

Pull before you start · Small commits, clear messages · Never commit secrets · Never force-push a shared branch · Review every AI suggestion before you run it

Latest edition and the full guide: `itmanagement101.co.uk/git-cheat-sheet`. Spotted a mistake or want something added? Leave a comment on the page and I'll consider it for the next edition.

Edition 1 · November 2026 · © Garrould Ltd 2026 · Licensed under CC BY-NC-ND 4.0 · Published by Garrould Ltd, Wreningham, Norfolk. Information is provided in good faith; try commands on a copy before using them on work that matters.