

Git & GitHub interview questions

27 questions · 15 easy · 10 medium · 2 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is the difference between Git and GitHub?

Easy



#basics

Git is a **distributed version control system**: a command-line tool that tracks changes to files. Every clone has the full history, so you can commit, branch and view history offline.

GitHub is a hosting service for Git repositories, plus collaboration features on top: pull requests and code review, issues, Actions (CI/CD), permissions, forks and releases.

GitLab, Bitbucket and Azure Repos are alternatives to GitHub; Git itself works the same with all of them.

2

Explain the working tree, staging area and repository.

Easy



#basics #staging

A file moves through three places:

- **Working tree**: the files you edit on disk.
- **Staging area (index)**: a snapshot of what will go into the next commit. `git add` copies changes here.
- **Repository** (`.git`): the committed history. `git commit` saves the staged snapshot as a new commit.

The staging area lets you **commit only part of your changes**, for example two separate commits from one editing session, or even single hunks with `git add -p`.

`git status` shows what is in each area; `git diff` shows unstaged changes and `git diff --staged` shows staged ones.

Example

```
git status
git add src/app.ts           # stage one file
git add -p                   # stage selected hunks interactively
git diff --staged            # what will be committed
git commit -m "feat: add login form"
```

3

What is the difference between `git commit` and `git push`?

Easy



#basics #remote

`git commit` records a snapshot in your **local** repository. Nobody else sees it, and it works offline.

`git push` uploads your local commits to a **remote** (such as `origin` on GitHub) and moves the remote branch to point at them.

So the usual flow is: edit, `add`, `commit` (possibly many times), then `push` when you want to share. If the remote has commits you do not have, the push is rejected and you must pull (merge or rebase) first.

4

`git fetch` vs `git pull`?

Easy



#remote

`git fetch` downloads new commits and branches from the remote and updates your **remote-tracking branches** (`origin/main`). It **never touches your working files** or your local branches, so it is always safe.

`git pull` is `fetch` followed by integrating the changes into your current branch: a **merge** by default, or a **rebase** with `git pull --rebase` (or `pull.rebase = true` in config).

Many developers prefer `fetch` first, look at what changed (`git log HEAD..origin/main`), then merge or rebase deliberately.

Example

```
git fetch origin
git log --oneline HEAD..origin/main    # commits I don't have yet
git rebase origin/main                 # or: git merge origin/main
git pull --rebase                       # fetch + rebase in one step
```

5

What is a branch in Git? Why are branches cheap?

Easy



#branches

A branch is just a **movable pointer to a commit**: a tiny file containing a commit hash. Each new commit on the branch moves the pointer forward.

Because creating a branch only writes that pointer (no files are copied), branches are instant and cheap, which is why Git workflows use a branch for every feature or fix.

- `git switch -c feature/login` creates and switches (older: `git checkout -b`)
- `git branch` lists, `git branch -d name` deletes a merged branch, `-D` forces
- `git push -u origin feature/login` publishes it and sets upstream tracking

6

What is the difference between merge and rebase? When do you use each?

Medium



#merge #rebase

Both integrate changes from one branch into another.

Merge creates a **merge commit** with two parents. History stays exactly as it happened and nothing is rewritten, but the log can get noisy.

Rebase replays your commits one by one on top of the target branch, creating **new commits with new hashes**. History becomes linear and easy to read, but it is rewritten.

Golden rule: **never rebase commits others have already pulled** (shared branches like `main`). Typical team practice:

- rebase your own feature branch onto `main` to stay up to date
- merge (or squash merge) the finished feature into `main` through a PR

Example

```
# on feature branch
git fetch origin
git rebase origin/main      # replay my commits on the latest main
git push --force-with-lease # history changed, so a normal push is
                             rejected

# alternative: merge main into my branch (no rewrite)
git merge origin/main
```

? Likely follow-up: Why is rebasing a shared branch dangerous?

7

What is a fast-forward merge?

Easy



#merge

If the target branch has **no new commits** since your branch was created, Git does not need a merge commit: it simply **moves the branch pointer forward** to your latest commit. History stays linear.

If both branches have new commits, Git creates a **three-way merge** commit instead (using the common ancestor).

Options:

- `git merge --no-ff` forces a merge commit even when a fast-forward is possible, so the feature stays visible as a group
- `git merge --ff-only` refuses to merge unless it can fast-forward (a safe setting for `git pull`)

8

What is a merge conflict and how do you resolve it?

Easy



#merge #conflicts

A conflict happens when two branches change **the same lines of the same file** (or one edits a file the other deletes), so Git cannot decide which version wins.

Steps:

- Git stops and marks the files; `git status` lists them

- open each file, find the markers `<<<<<<`, `=====`, `>>>>>>`, and edit to the correct final code (often a combination of both)
- remove the markers, run the tests, then `git add` the file
- finish with `git commit` (merge) or `git rebase --continue` (rebase)

To give up, use `git merge --abort` or `git rebase --abort`. Reduce conflicts with small PRs, frequent syncing with `main`, and talking to whoever touches the same code.

Example

```
git merge feature/login
# CONFLICT (content): Merge conflict in src/auth.ts
git status                # shows "both modified"
# edit src/auth.ts, remove <<<<<< ===== >>>>>> markers
git add src/auth.ts
git commit                # or: git rebase --continue
```

9

Explain `git reset` (soft, mixed, hard), `git revert` and `git restore`.

Medium 🔥🔥 #undo #reset

`git reset <commit>` moves the current branch pointer back:

- `--soft`: keeps the changes **staged**
- `--mixed` (default): keeps the changes in the working tree, **unstaged**
- `--hard`: **throws the changes away** (recoverable only through reflog, and uncommitted work is lost)

Reset rewrites history, so use it only on commits not yet pushed.

`git revert <commit>` creates a **new commit** that undoes an earlier one. History is kept, so it is the safe way to undo something already pushed to a shared branch.

`git restore` works on files: `git restore file` discards working-tree changes, `git restore --staged file` unstages.

Example

```
git reset --soft HEAD~1      # undo last commit, keep changes staged
git reset HEAD~1             # undo last commit, keep changes unstaged
git reset --hard HEAD~1      # undo last commit and discard changes
git revert a1b2c3d           # new commit that reverses a1b2c3d (safe
when pushed)
git restore --staged app.ts   # unstage
git restore app.ts           # discard local edits
```

? **Likely follow-up:** You already pushed a bad commit to `main`. What do you do?

10

How do you undo the last commit?

Easy 🔥🔥 #undo

It depends on whether it has been pushed and what you want to keep.

Not pushed yet:

- fix the message or add a forgotten file: `git commit --amend`
- undo the commit but keep the work: `git reset --soft HEAD~1` (staged) or `git reset HEAD~1` (unstaged)
- throw it away completely: `git reset --hard HEAD~1`

Already pushed to a shared branch: `git revert HEAD`, which adds a new commit reversing it without rewriting history.

On your own pushed feature branch, you may reset and then `git push --force-with-lease`, as long as nobody else builds on it.

11

What does `git commit --amend` do?

Easy 🔥🔥 #undo #commits

It **replaces the last commit** with a new one that combines the previous commit and whatever is currently staged. Use it to fix a typo in the message or add a forgotten file.

- `git commit --amend -m "new message"` changes the message
- `git add forgotten.ts` then `git commit --amend --no-edit` adds the file and keeps the message

Because the commit gets a new hash, amending a commit that is already pushed means the remote branch has diverged, and you would need a force push. Only do that on your own branch.

12

What is `git stash` and when would you use it?

Easy

🔥🔥 #stash

`git stash` saves your uncommitted changes (staged and unstaged) on a stack and returns you to a clean working tree, without making a commit.

Typical case: you are halfway through a feature and need to switch branches to fix an urgent bug.

- `git stash push -m "wip login"` saves with a name; `-u` also includes untracked files
- `git stash list` shows saved entries
- `git stash pop` re-applies the latest and removes it; `git stash apply` keeps it on the stack
- `git stash drop` deletes one; `git stash branch new-branch` creates a branch from it

Pop can cause conflicts, just like a merge.

Example

```
git stash push -u -m "wip: login form"
git switch main && git switch -c hotfix/crash
# ...fix, commit, push...
git switch feature/login
git stash pop
```

13

What is `git cherry-pick`?

Medium

🔥🔥 #cherry-pick

`git cherry-pick <commit>` takes the changes from one specific commit and applies them as a new commit on your current branch (with a new hash).

Common uses:

- backporting a bug fix from `main` to a release branch

- rescuing a commit made on the wrong branch
- taking one useful commit from an abandoned branch

You can pick a range (`A^..B`), use `-x` to record the original hash in the message, and resolve conflicts like a merge (`--continue`, `--abort`). Use it sparingly: the same change existing as two different commits can confuse later merges.

Example

```
git switch release/2.3
git cherry-pick -x 9f3e2a1      # bring the fix from main
git push origin release/2.3
```

14

What is `HEAD`? What is a detached HEAD?

Medium 🔥🔥 #head #basics

`HEAD` is a pointer to **where you are now**. Normally it points to a branch (`refs/heads/main`), and that branch points to a commit. When you commit, the branch moves and `HEAD` moves with it.

Detached HEAD means `HEAD` points directly at a commit instead of a branch. It happens when you check out a commit hash, a tag or a remote branch directly (`git checkout v1.2.0`).

You can look around and even commit, but those commits belong to no branch, and once you switch away they are only reachable through the reflog. To keep them, create a branch: `git switch -c fix-from-here`.

`HEAD~1` is the parent of `HEAD`; `HEAD~3` goes three commits back.

15

What is the difference between `origin` and `upstream`?

Easy 🔥🔥 #remote #forks

Both are just **remote names**, conventions rather than special keywords.

- `origin` is the default name for the repository you cloned from. In a fork workflow, that is your fork.
- `upstream` usually names the original repository you forked from, added with `git remote add upstream <url>`.

Flow for open source: fetch from `upstream` to stay current, push your branches to `origin`, then open a PR from your fork to upstream.

Separately, a branch's **upstream (tracking) branch** is the remote branch it syncs with, set with `git push -u`, so plain `git pull` and `git push` know where to go.

Example

```
git remote add upstream https://github.com/org/project.git
git fetch upstream
git rebase upstream/main
git push origin feature/docs-fix
git remote -v
```

16

Walk me through a pull request workflow and good code review practice.

Easy



#github #workflow

A typical flow:

- update `main`, create a branch: `feature/123-login`
- small, focused commits with clear messages
- push and open a **pull request** with a description: what, why, how to test, screenshots
- CI runs tests, lint and builds; reviewers comment, you push fixes
- once approved and green, merge (often **squash merge**), delete the branch, deploy

Good review practice: keep PRs small (a few hundred lines), review for correctness, design, readability, tests and security rather than style that a linter can catch, and **be kind and specific**. **Branch protection** rules on `main` can require reviews and passing checks before merging.

? Likely follow-up: How do you keep a long-running PR up to date with `main`?

17

Compare Git Flow, GitHub Flow and trunk-based development.

Medium



#workflow #branches

- **Git Flow**: long-lived `main` and `develop`, plus `feature/*`, `release/*` and `hotfix/*` branches. Good for versioned software with planned releases; heavy for web apps that deploy constantly.

- **GitHub Flow:** one long-lived `main` that is always deployable. Short feature branches, a PR, review, merge, deploy. Simple and popular for web apps.
- **Trunk-based development:** everyone integrates into `main` (trunk) at least daily with very short-lived branches or direct commits. Unfinished work is hidden behind **feature flags**. It **minimises merge pain** and enables continuous delivery, but needs strong automated tests.

The trend is towards GitHub Flow or trunk-based with CI/CD.

18

What is `.gitignore`? Why is a file still tracked after I added it there?

Easy 🔥🔥 #gitignore #basics

`.gitignore` lists patterns of files Git should not track: `node_modules/`, build output, `.env` secrets, logs, editor files. Patterns support `*`, `**`, a trailing `/` for directories and `!` to re-include.

It only affects **untracked** files. If a file was already committed, **ignoring it does nothing**; you must remove it from the index while keeping it on disk:

`git rm --cached .env` (or `-r` for a folder), then commit.

If a secret was committed, removing it now is not enough: it is still in history. **Rotate the secret**, and rewrite history (`git filter-repo`) if needed. Use a global ignore file for personal editor files.

Example

```
echo ".env" >> .gitignore
git rm --cached .env
git commit -m "chore: stop tracking .env"
git check-ignore -v dist/app.js    # which rule ignores this file?
```

19

What does squashing commits mean?

Easy 🔥🔥 #rebase #history

Squashing combines several commits into one. Ten "wip" and "fix typo" commits become a single meaningful commit, which keeps `main`'s history readable and makes reverting a feature one step.

Two common ways:

- **Squash and merge** on GitHub: the whole PR lands on `main` as one commit
- **Interactive rebase:** `git rebase -i HEAD~5`, then mark commits as `squash` or `fixup`

The downside is losing the fine-grained history of how the feature was built, and squashed branches look unmerged to Git, so delete them after merging.

20

What can you do with interactive rebase?

Medium 🔥🔥 #rebase #history

`git rebase -i <base>` opens a list of your commits, oldest first, and lets you rewrite local history before sharing it:

- `pick`: keep the commit
- `reword`: change its message
- `edit`: stop to amend it (split it, add a file)
- `squash`: merge into the previous commit, combining messages
- `fixup`: merge into the previous commit, discarding this message
- `drop`: delete it
- reorder by moving lines

Tip: `git commit --fixup <hash>` then `git rebase -i --autosquash` places fix commits automatically. Only rewrite commits that are not shared, then push with `--force-with-lease`.

Example

```
git rebase -i HEAD~4
# pick    a1b2c3d feat: add login form
# fixup   d4e5f6a fix typo
# reword  7a8b9c0 add tests
# drop    1d2e3f4 debug logging
```

21

Why use `--force-with-lease` instead of `--force`?

Medium 🔥🔥 #remote #safety

After a rebase or amend, your branch has diverged from the remote, so a normal push is rejected and you need to force it.

`git push --force` overwrites the remote branch no matter what is there. If a teammate pushed commits in the meantime, they are silently destroyed.

`git push --force-with-lease` only overwrites if the remote branch is still at the commit you last fetched. If someone pushed since, it refuses, and you can fetch and integrate their work first.

Rules: never force push to `main` or other shared branches (protect them on GitHub), and only force push your own feature branches.

22

How do you recover a commit or branch you lost?

Medium 🔥🔥 #reflog #undo

Use the **reflog**. Git records every position `HEAD` (and each branch) has pointed to locally, including before resets, rebases, amends and deleted branches. Commits stay in the repository until garbage collection (by default 30 to 90 days).

- `git reflog` lists entries like `HEAD@{3}: reset: moving to HEAD~2`
- find the hash from before the mistake
- recover it: `git reset --hard <hash>`, or more safely `git branch rescue <hash>`

This is why almost nothing committed is ever truly lost in Git. Uncommitted changes discarded by `reset --hard` or `restore`, however, are gone. The reflog is local only; it is not pushed.

Example

```
git reset --hard HEAD~3      # oops
git reflog
# 4f2a9c1 HEAD@{0}: reset: moving to HEAD~3
# 8be71d0 HEAD@{1}: commit: add payment webhook
git branch rescue 8be71d0    # or: git reset --hard HEAD@{1}
```

23

What are Git tags and how do they relate to releases and semver?

Easy 🔥👉 #tags #releases

A **tag** is a name for a specific commit that, unlike a branch, does not move. Tags mark releases like `v2.4.1`.

- **Lightweight** tag: just a pointer.

- **Annotated** tag (`git tag -a`): a full object with author, date and message (and optionally a signature). Use these for releases.
- Tags are not pushed by default: `git push origin v2.4.1` or `--tags`.

Semantic versioning is MAJOR.MINOR.PATCH: **MAJOR** for breaking changes, **MINOR** for backwards-compatible features, **PATCH** for bug fixes. GitHub **Releases** are built on tags and add notes and build artifacts; CI often deploys or publishes when a `v*` tag is pushed.

Example

```
git tag -a v2.4.1 -m "Fix checkout rounding bug"
git push origin v2.4.1
git tag --list "v2.*"
git checkout v2.4.0          # detached HEAD at that release
```

24

What is `git bisect`?

Medium



#debugging #history

`git bisect` finds the commit that introduced a bug with a **binary search** through history.

You mark one commit as **bad** (has the bug) and an older one as **good**. Git checks out the commit halfway between; you test and mark it good or bad, and it halves the range again. Across 1000 commits it needs only about 10 steps.

You can automate it with `git bisect run <test-command>`: Git uses the script's exit code (0 = good) to decide. Finish with `git bisect reset` to return to where you started.

Example

```
git bisect start
git bisect bad          # current commit is broken
git bisect good v2.3.0  # this release was fine
git bisect run npm test # let the tests decide each step
git bisect reset
```

25

What are Git hooks? Give examples.

Medium



#hooks #automation

Hooks are scripts Git runs automatically at certain events. Client-side hooks live in `.git/hooks`:

- `pre-commit`: lint and format staged files, block commits with secrets
- `commit-msg`: enforce a message format such as Conventional Commits
- `pre-push`: run tests before pushing

Server-side hooks (`pre-receive`, `post-receive`) run on the remote; on GitHub you use branch protection, Actions and webhooks instead.

`.git/hooks` is **not committed**, so teams share hooks with tools like **Husky** plus **lint-staged** (JavaScript), or the `pre-commit` framework. Hooks can be skipped with `--no-verify`, so CI must still enforce the same checks.

26

What are Git submodules, and what are the alternatives?

Hard



#submodules

A **submodule** embeds another Git repository inside yours at a **pinned commit**. The parent repo stores only the URL (in `.gitmodules`) and the exact commit hash, not the files.

Commands: `git submodule add <url> libs/ui`, and after cloning `git submodule update --init --recursive` (or `git clone --recurse-submodules`).

They are known for being awkward: people forget to init or update them, detached HEADs inside, and two-step commits when changing both repos.

Alternatives:

- a **package manager** (publish the shared code as an npm package)
- a **monorepo** with workspaces (pnpm, Turborepo, Nx)
- `git subtree`, which copies the code into your repo

How does Git store data internally?

Hard

#internals

Git is a **content-addressable** store. Everything lives in `.git/objects` as objects named by the hash (SHA-1, or SHA-256 in new repos) of their content:

- **blob**: a file's contents (no name)
- **tree**: a directory listing, mapping names to blobs and sub-trees
- **commit**: points to one root tree plus parent commit(s), author, date and message
- **tag**: an annotated tag object

So each commit is a **full snapshot, not a diff**; unchanged files simply reuse the same blob, and pack files compress similar objects. Branches and tags are just refs (files containing a hash).

Because a commit's hash covers its parents, changing any old commit changes every hash after it. That is why rewriting history requires a force push.