TUTORIAL

How to Use Git Rebase to Clean Up Commit History

Git rebase is the standard way to rewrite a messy local commit history into a clean, reviewable sequence. This tutorial shows when to use it, how to prepare safely, how to run an interactive rebase, and how to validate the result before you push.

Programming - July 29, 2026

Why commit cleanup matters

Messy commit history makes code reviews harder, complicates bisecting, and creates unnecessary noise when multiple small fixes are mixed into one logical change. git rebase lets you rewrite a local branch so the final history reads as a deliberate sequence of changes instead of an edit log.

In this tutorial, you will learn how to use git rebase to clean up commit history safely, when it is appropriate, how to prepare for the rewrite, and how to validate the final branch before it is shared with others.

When rebase is the right tool

Use rebase when the branch history is still local or when your team explicitly allows history rewriting on feature branches. The goal is to take a series of commits such as "fix typo", "wip", and "address review comments" and turn them into a smaller, clearer set of commits that each represent one logical change.

Rebase is not the right tool for every branch. If the branch is already shared and others may have based work on it, rewriting history can cause avoidable merge problems. If you need to preserve the exact public history, use merge commits instead.

Stop here if the branch is already shared broadly

Do not rewrite history on a branch that other people are actively using unless your team has a clear process for coordinated force-pushes. If that is the case, confirm the branch policy first. For local-only work, or branches that you own and have not published yet, rebase is usually safe.

If you are also carrying uncommitted changes, review How to Rebase Git Commits Without Losing Local Changes before you start. A clean working tree makes the workflow simpler and lowers the chance of losing edits during the rewrite.

Prerequisites and preparation

The safest rebase workflow starts before the first command is run. Your goal is to make the operation reversible and to reduce surprises while Git pauses for conflict resolution.

Goal

Prepare a branch that can be rewritten without losing work or confusing collaborators.

Action

Before rebasing, make sure of the following:

- Your working tree is clean, or your uncommitted work is safely stashed or committed.
- You know which commits you want to keep, squash, edit, or drop.
- The branch is local or otherwise safe to rewrite.

- You have a way to compare the rewritten branch with the original state if needed.

A simple checkpoint is useful before any interactive rebase:

If you want an extra safety reference, create a temporary backup branch:

Expected output

You should know exactly which branch you are on, see the recent commit sequence, and have a restore point if the rewrite does not go as planned.

Validation

Confirm that git status reports a clean tree and that git log --oneline shows only the commits you intend to reorganize.

Common failure

A common mistake is starting a rebase while local changes are still unstaged or while you are not on the correct branch. Another common failure is forgetting that rebase changes commit IDs, which matters if any commit has already been shared.

Clean up history with interactive rebase

Interactive rebase is the practical tool for commit cleanup because it gives you explicit control over each commit in the range you choose.

Goal

Rewrite a sequence of commits so the final history is easier to review and maintain.

Action

Start an interactive rebase from the parent of the first commit you want to change. For the last five commits, for example:

Git opens an editor with a list of commits and actions such as pick, reword, edit, squash, fixup, and drop.

Typical cleanup patterns:

- Use reword to improve a commit message without changing content.
- Use squash to combine a commit into the previous one and keep both messages for editing.
- Use fixup to combine a cleanup commit into the previous one and discard the cleanup message.
- Use drop to remove an unwanted commit entirely.
- Use edit when you need to pause and modify the commit content before continuing.

A common cleanup sequence might look like this:

Expected output

After you save and close the editor, Git begins replaying the selected commits in the new order and with the chosen transformations.

Validation

Check the rewritten history after the rebase completes:

The output should show a shorter or cleaner sequence of commits with better messages and fewer noise commits.

Common failure

The most frequent issue is choosing the wrong rebase range. If you start too far back, you may rewrite commits you did not intend to touch. If you start too late, the noisy commit you wanted to remove remains in history.

Handle conflicts during rebase

Conflicts are normal when replaying commits onto a different base or when the cleaned-up sequence changes file context. If you want a deeper operational view of paused rebases and safe conflict handling, see Git Rebase Conflicts: Resolve and Preserve Commit History.

Goal

Resolve conflicts without losing the intent of the rewritten commits.

Action

When Git stops for a conflict, inspect the files it reports, resolve the conflicting sections, and then continue the rebase:

If you realize the chosen resolution is wrong, you can stop and return to the pre-rebase state:

Expected output

The rebase continues commit by commit until the rewritten history is complete.

Validation

After each conflict resolution, use git status to confirm that only the expected files remain staged and that the repository is not left in an uncertain state.

Common failure

A common mistake is resolving the immediate conflict but not checking the surrounding logic. Another is running git add . without reviewing what changed, which can accidentally stage unrelated edits.

Edit a commit in the middle of the series

Sometimes cleanup is not just about combining or renaming commits. You may need to adjust the contents of a specific commit, split it, or fix an earlier mistake that only becomes visible after the branch is replayed.

Goal

Modify one historical commit while preserving the rest of the branch structure.

Action

Mark the target commit as edit in the interactive rebase list. When Git pauses, make the necessary changes, stage them, and amend the commit:

If you need to split one commit into several logical commits, pause on that commit, reset it partially or fully as appropriate for your workflow, then create the new commits before continuing the rebase.

Expected output

The target commit is replaced by a corrected version, and the later commits are replayed on top of it.

Validation

Use git log --oneline --decorate to verify that the revised commit appears in the right place and that the resulting commit messages still tell a coherent story.

Common failure

The main risk here is accidentally changing more than the intended commit. Another risk is forgetting that any downstream commit may need to be adjusted after the earlier commit changes.

Validate the rewritten branch before you share it

A clean-looking history is not enough. You should verify that the content of the rewritten branch still matches what you intended to ship.

Goal

Confirm that the rewrite preserved functionality while improving history quality.

Action

Use a combination of history inspection and repository checks:

If you created a backup branch, compare the rewritten branch with the original to see exactly what changed. You can also run the relevant test suite, linters, or build checks for the project.

Expected output

The log should be easier to read, and the diff between the backup and the rewritten branch should reflect only the intended history cleanup, not accidental content loss.

Validation

Verify all three of these before you push or hand off the branch:

- The commit sequence matches the intended logical order.
- No unrelated files disappeared or changed unexpectedly.
- The project tests or checks still pass for the modified area.

Common failure

The most dangerous failure is assuming that a successful rebase means the branch is correct. Rebase only proves that Git could replay the commits; it does not prove that the rewritten history still contains everything you meant to keep.

Publish the cleaned history carefully

Once the branch is validated, you can share it. If the branch already exists on a remote and you are replacing its history, use the safest team-approved update procedure for that repository. In many workflows, that means a force push with lease-style protection rather than an unguarded overwrite.

Goal

Update the remote branch without clobbering someone else?s newer work.

Action

If your team allows rewriting the remote branch, push using the project?s approved method and then inform collaborators that the branch history changed. If the branch is strictly local or newly created, a normal push may be enough once the cleaned history is ready.

Expected output

The remote branch now reflects the cleaned commit sequence, and anyone reviewing it sees the improved history instead of the original noisy series.

Validation

After pushing, fetch the remote state and confirm that the remote tip matches the rewritten branch you reviewed locally.

Common failure

The common mistake is pushing a rewritten branch without coordination. That can leave teammates with diverged local histories and unnecessary recovery work.

A practical decision rule for everyday use

A good rule is simple: rebase local, in-progress, or explicitly rewriteable branches to make the history easier to read; avoid rewriting branches that other people are already consuming unless the team has a controlled process for it.

If you want to reduce the chance of pain during a paused rewrite, especially when conflicts are likely, Resolving Git Merge Conflicts with Interactive Rebase is a useful companion workflow for the cases where the rebase does not apply cleanly.

Final takeaway

git rebase is the right tool when you want a clean, reviewable commit history and you are working in a branch that is safe to rewrite. The reliable workflow is: prepare a backup, run interactive rebase, resolve conflicts carefully, validate the rewritten branch, and only then publish it. If you keep those checks in order, rebase becomes a controlled history-editing tool rather than a risky one.

Use this guidance together with learning machine learning to connect the workflow with related operational context already available on the site.