TUTORIAL

How to Rebase Git Commits Without Losing Local Changes

Rebasing with local changes is safe when you separate your work, choose the right storage strategy, and validate the result before you continue. This tutorial shows a practical workflow for preserving uncommitted edits while rewriting commit history.

Programming - July 29, 2026

Why this matters

A rebase rewrites commit history. If you start one while you still have local edits in your working tree, Git may stop with conflicts, silently move your changes in a way you did not expect, or make it harder to recover what you were working on. In a shared repository, that can mean lost context, broken builds, and time spent reconstructing a working tree instead of shipping the change.

This tutorial shows how to rebase Git commits without losing local changes. You will learn how to decide whether the approach applies, how to protect uncommitted work, how to perform the rebase safely, and how to verify that your final branch contains both the rewritten commits and your local edits.

What you will end up with

By the end of the process, you should have:

- a branch rebased onto a new base commit or branch
- your local work preserved, either as a commit, stash entry, or separate branch

- a clean validation path that confirms no edits were dropped
- a rollback option if the rebase or conflict resolution goes wrong

If your workflow depends on preserving a linear history, also review Git Rebase vs Merge: When to Use Each Strategy. When your branch already has conflicts, Git Rebase vs Merge: Resolving Conflicts in Feature Branches helps explain what changes during conflict resolution.

Prerequisites and stop-here checks

Before you rebase, confirm the following.

Your working tree state

Goal: know whether you have uncommitted changes that must be preserved.

Action: run:

Expected output: either an empty result or a short list of modified, deleted, or untracked files.

Validation: if the output is empty, you can rebase normally. If not, decide how to preserve those changes before continuing.

Common failure: starting the rebase with unrelated edits still in the working tree and then confusing those edits with rebase-generated conflict markers.

Stop-here-if warning

Stop here if the repository is in a partially resolved conflict state. If git status shows unresolved paths, finish or abort that operation first. Rebasing on top of an unfinished merge or rebase is a common way to corrupt your mental model of the branch state.

Your recovery path

Goal: make sure you can undo the operation if needed.

Action: confirm that you know how to inspect the branch history and return to the original commit with git reflog if required.

Expected output: a clear fallback path, not a guess.

Validation: run:

Common failure: assuming git reset --hard is the only recovery option and acting too late.

Choose the right protection method for local changes

You have three practical ways to protect local work before a rebase:

- commit the changes on a temporary branch
- stash the changes, rebase, then reapply the stash
- use an interactive workflow that pauses before the rebase and restores the edits afterward

For technical work, the safest default is usually a temporary commit on a throwaway branch if the changes are meaningful, or a stash entry if the edits are short-lived and not ready for review. The right choice depends on whether the changes belong in version control yet.

Temporary branch and commit

Goal: preserve work as a real commit that can be cherry-picked, inspected, or amended later.

Action:

Expected output: one commit capturing your current edits.

Validation: git log --oneline --decorate -n 3 should show the WIP commit on the temporary branch.

Common failure: using this method when the changes are not supposed to be committed at all, such as local test data or secret material.

Stash the changes

Goal: temporarily remove local edits from the working tree without committing them.

Action:

Expected output: the working tree becomes clean, and the stash entry stores tracked and untracked changes when -u is used.

Validation:

Common failure: forgetting -u and then assuming untracked files were saved.

Stop-here-if warning

Stop here if your local edits include generated files, build artifacts, secrets, or other content you would not want restored automatically. Put those files in a deliberate location first, or exclude them from the stash set. Reapplying them blindly can recreate the very problem you were trying to avoid.

Rebase with a clean working tree

Goal: move your branch commits onto the new upstream base without mixing in local edits.

Action: after preserving your local work, run the rebase.

Typical examples include rebasing your feature branch onto main or onto a newer integration branch.

Expected output: Git rewrites the branch commits and either completes successfully or stops at a conflict.

Validation: git status should show either a clean branch after completion or a specific conflict state that names the affected paths.

Common failure: using the wrong upstream reference and rebasing onto a branch that does not represent the intended base. Always confirm the target branch before you start.

Restore local changes safely

Goal: bring your saved edits back after the rebase.

If you used stash

Action:

Expected output: your stashed edits reappear in the working tree, and Git either applies them cleanly or reports conflicts.

Validation: run git status --short and inspect the files that changed. If there are conflicts, resolve them before you continue.

Common failure: using git stash pop and assuming success without checking whether the stash was actually applied cleanly. If the apply fails, the stash may still remain, but you must verify its state.

If you used a temporary branch commit

Action: cherry-pick or merge the WIP commit onto the rebased branch, depending on your workflow.

Expected output: your preserved changes become new commit(s) on top of the rebased branch.

Validation: confirm the commit appears in git log --oneline --decorate and that the file diffs match the intended edits.

Common failure: cherry-picking the preservation commit before the rebase is complete, which can create unnecessary conflicts or duplicate work.

Validate that nothing was lost

Goal: prove the rebased branch contains both the intended history rewrite and the local edits you protected.

Action: inspect the diff and compare it to your saved state.

Expected output: the branch history shows the rewritten commits in the new order, and the diff reflects the files you expected to change.

Validation rules:

- git status --short should show only the files you still intend to modify
- git diff should contain your expected edits, not surprise deletions
- if you used a stash, verify the stash list is empty only when you are certain the restore succeeded

If the repository uses tests or build checks, run them now. A rebase can preserve the file contents and still leave you with a branch that no longer compiles because of changed commit ordering.

Common failure: validating only the commit graph and not the working tree. A clean history is not the same as a correct code state.

Common conflict scenarios and safe responses

A rebase with local changes usually fails in one of three ways.

The rebase stops with content conflicts

Goal: resolve the conflicts in the rebased version of each commit.

Action: open the files Git marks as conflicted, resolve them, then continue:

Expected output: Git advances to the next commit or finishes the rebase.

Validation: git status should no longer list unresolved paths.

Common failure: resolving the file once and assuming the entire rebase is done. A multi-commit rebase may require the same file to be revisited more than once.

The stash reapply conflicts with rewritten commits

Goal: separate rebase conflicts from local-edit conflicts.

Action: resolve the rebase first, then apply the stash and handle only the remaining local differences.

Expected output: the source of each conflict is clearer because the history rewrite and the local edit restore happen in separate steps.

Validation: compare the final diff against your pre-rebase notes or saved patch.

Common failure: applying the stash before the rebase has finished, which makes it much harder to tell which change caused each conflict.

You realize the rebase target was wrong

Goal: abort cleanly and return to the pre-rebase state.

Action:

Expected output: Git restores the branch to the state it had before the rebase started.

Validation: run git status --short and inspect the branch pointer with git log --oneline -n 3.

Common failure: continuing after noticing the target is wrong and then trying to fix the mistake with extra commits.

Operational follow-up after the rebase

Goal: make the rebased branch safe to hand off, review, or push.

Action: review the final branch state before sharing it.

Expected output: a clean or intentionally modified working tree and a history that matches your rebase goal.

Validation checklist:

- confirm the base branch is the one you intended
- confirm no unexpected files were dropped during stash pop or cherry-pick

- confirm tests or linters pass if the repository expects them
- if the branch was already pushed, coordinate before force-pushing because rebased histories change commit IDs

If your team routinely chooses between rebasing and merging feature branches, it is worth standardizing the rule set so every engineer knows when rewritten history is acceptable and when merge commits are the safer operational choice.

Practical rule of thumb

If the local edits are disposable, stash them. If they matter, commit them on a temporary branch. If the branch is already unstable, stop and clean up the conflict state before rebasing. The safest rebase is the one you start with a clean working tree, a verified upstream target, and a known recovery path.

When you follow that sequence, you can rebase Git commits without losing local changes and still end with a branch that is easy to inspect, test, and share.

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