SCRIPT

Git Script to Automate Branch Cleanup and Prune Remotes

A practical Bash script for safely identifying stale local branches, pruning remote-tracking references, and deleting branches only after validation. The workflow defaults to dry-run behavior and includes safeguards, rollback guidance, and production checks.

Programming - July 04, 2026

Why branch cleanup becomes an operational problem

Long-lived Git repositories accumulate stale local branches and outdated remote-tracking references faster than most teams notice. That creates real operational friction: developers see noisy branch lists, automation can target refs that no longer exist, and storage or CI jobs can behave unpredictably when old references are left behind. In security-sensitive environments, stale branches also make it harder to reason about what code is still active, reviewed, or eligible for release.

This post gives you a practical Bash script that automates branch cleanup while defaulting to safe behavior. After reading it, you will be able to identify stale branches, prune remote-tracking refs, dry-run deletions first, and decide what to verify before running cleanup in production repositories.

What this script does and does not do

The script below is a conservative Bash workflow for repositories where you want to clean up stale local branches after they have been merged, and prune remote-tracking references after the upstream has removed them.

It does:

- Verify that Git is installed and that the current directory is inside a Git repository.
- Fetch and prune remote-tracking references from a chosen remote.
- List local branches that appear merged into a target branch.
- Exclude protected branches such as main, master, and the currently checked-out branch.
- Run in dry-run mode by default for deletions.
- Require explicit confirmation before actually deleting branches.
- Emit readable status messages and a compact summary.

It does not:

- Decide whether a branch is safe to delete based on business context or release policy.
- Inspect open pull requests, change requests, or deployment state.
- Force-delete unmerged branches unless you deliberately change the script.
- Replace branch governance, protections, or review policy.

If your workflow needs to evaluate structured input before acting, the same operational pattern used in a JavaScript script to parse and validate JSON payloads is a useful comparison: validate first, act second, and make failure modes explicit.

Requirements, assumptions, and permissions

Script language

Bash shell script using standard Git CLI commands.

Assumptions

- You run the script from inside a Git working tree.
- origin is the default remote unless you override it.

- The repository uses a recognizable target branch such as main or master.
- Branch deletion policy allows removing local branches already merged into the target branch.

Required tools

- Bash 4+ or a compatible POSIX-like shell with Bash available.
- Git CLI installed and on PATH.
- A user account with permission to read the repository and delete local branches.

Optional but recommended permissions

- Permission to fetch and prune from the selected remote.
- Permission to delete local branches in the working copy.
- If you extend the script to delete remote branches, explicit authorization for remote ref deletion.

Module and API access

This script does not require external APIs or third-party modules. It uses only local Git commands:

- git rev-parse
- git fetch --prune
- git branch --merged
- git branch -d
- git symbolic-ref

The script

How the script works

The script follows a deliberate sequence so you can validate each stage before making changes.

1. Validate the environment

It first checks that Git is available and that the current directory is inside a repository. This prevents accidental execution in a random working directory and gives a clear error message if the environment is not ready.

2. Resolve the target branch

The target branch is the branch used to decide whether a local branch is considered merged. If you do not pass --target, the script tries origin/main, then origin/master, then local main or master. If none exists, it exits and asks you to specify --target explicitly.

3. Prune remote-tracking refs

git fetch --prune origin removes stale remote-tracking references that no longer exist on the remote. This is a metadata cleanup step; it does not delete remote branches, and it does not modify working files.

4. Identify merged branches

The script collects branches merged into the target branch using git branch --merged. It then filters out protected branches and the currently checked-out branch. That reduces the risk of deleting active work or mainline references.

5. Dry-run by default

If you run the script without --apply, it lists the eligible branches and stops. This is the safe default and is appropriate for first-time runs, shared repositories, and regulated environments where you need review before action.

6. Confirm and delete

With --apply, the script prompts for confirmation before deleting anything. It uses git branch -d by default, which refuses to delete unmerged branches. If you add --force, it switches to git branch -D, which can delete even when Git thinks the branch is not merged. That is intentionally opt-in.

Parameters, input, and expected output

Command-line parameters

- --remote REMOTE: remote name to prune and inspect, usually origin
- --target BRANCH: branch used as the merge baseline, such as origin/main
- --apply: enable deletion instead of dry-run
- --force: use force deletion with git branch -D
- --verbose: show more diagnostic output

Expected input

The script expects only shell arguments and a Git repository state. It does not read a data file. The important implicit inputs are:

- Current branch state
- Local branch list
- Remote-tracking refs
- Merge history relative to the target branch

Example command

Example dry-run output

Example apply output

What to change before production use

The script is usable as-is for many internal repositories, but production use usually requires policy tuning.

- Review PROTECTED_BRANCHES and add any release or environment branches that should never be deleted locally.
- Set the correct default remote if your repo does not use origin.
- Change the inferred target branch logic if your mainline branch naming is different.
- Decide whether force deletion should be removed entirely in stricter environments.
- If your change-management process requires approval, keep --apply behind a controlled wrapper rather than allowing ad hoc execution.

If you want to follow a similar validate-before-act pattern in another automation workflow, the operational discipline is similar to how to get started with JavaScript: start with a minimal, observable workflow, verify output, and expand only after behavior is predictable.

Validation and testing steps

Before using the script widely, test it in a clone or a non-critical repository.

- Run git status and confirm you are in the expected repo.
- Execute the script without --apply first.
- Check that the eligible branches are exactly the ones you expect.
- Confirm the target branch is correct and current with your release baseline.
- Re-run with --verbose if you need to inspect skipped branches.
- Test --apply only after confirming the dry-run output.
- Verify that deleted branches no longer appear in git branch.
- Run git fetch --prune origin separately if you want to confirm the remote-tracking cleanup behavior before bundling it into automation.

A simple preflight check can also help catch mismatch errors:

If those commands return unexpected branches, do not run deletion until the target branch is corrected.

Cleanup, rollback, and recovery guidance

Branch cleanup is reversible only if the branch tip is still reachable.

If you deleted the wrong local branch

- Use git reflog to find the commit hash that pointed to the branch tip.
- Recreate the branch with git branch <branch-name> <commit-hash>.
- If the branch had unpushed work, verify whether any local commits were lost from the symbolic reference before you continue.

If the remote-tracking ref is stale but the remote branch still exists

- Run git fetch --prune <remote> again.
- Check whether the remote branch was recreated under the same name.
- Confirm the remote repository state before treating a missing ref as authoritative.

If the script skipped a branch you expected to delete

That usually means one of three things:

- The branch is not actually merged into the selected target branch.
- The branch is protected by the script.
- You are on the branch you intended to clean up, so the safety check prevented deletion.

Security and privacy notes

This script has a small attack surface because it uses local Git operations only, but operational security still matters.

- Do not run --force on repositories with uncertain merge history unless you have explicit approval.
- Avoid using the script on a working tree with uncommitted local changes if your cleanup process might distract from active work.
- Treat branch names and commit messages as potentially sensitive metadata in regulated environments.
- If your repository contains protected release branches, keep them in the protection list and validate them against actual repository policy.
- For shared systems, prefer a wrapper that logs who ran the script, when, and against which repository path.

Common mistakes

Mistake	Why it matters	Better approach
Running with --apply before reviewing dry-run output	You can delete the wrong branch if the merge baseline is incorrect	Start with d
Using git branch -D by default	Force deletion can remove branches that Git would normally protect	Keep safe deletion as the default
Pruning without checking the remote name	You may prune the wrong upstream if the repo uses a nonstandard remote	Pass --remote
Assuming merged means safe to delete in every case	A branch can be merged but still needed for audit, release, or rollback reasons	
Forgetting to exclude the current branch	Deleting the checked-out branch can interrupt the session and create confusion	Always filte

Final takeaway

A good Git branch cleanup script should be boring, predictable, and hard to misuse: prune stale remote-tracking refs, identify merged local branches, dry-run by default, and require explicit approval before deletion so cleanup improves repository hygiene without creating recovery problems.

Use this guidance together with Ubuntu security patch audit script and Python JSONDecodeError to connect the workflow with related operational context already available on the site.

> Part of the Programming: Git Insights content cluster.