Two Git Guardrails: Auto-Push on Commit and a Check on the Way Out

Ronald G. Thomas

2026-11-11



A good guardrail is invisible until the moment it is needed, and then it asks one short question.

Introduction

For several years the git repositories on my machines were kept in sync with GitHub by a scheduled backup sweep. Every so often a job walked each repository and ran the obvious three commands:

```
git add -A
git commit -m "Auto-backup: <timestamp>"
git push
```

The sweep did what it promised. Remotes rarely fell more than a day behind, and work done on one laptop appeared on the next. Its defect was the first line. `git add -A` stages everything in the working tree that is not ignored, so the sweep committed whatever happened to be on disk at the moment it ran, whether or not anyone intended it to be part of the project.

The failure that finally retired it was small. Claude Code writes a machine-local permissions file, `.claude/settings.local.json`, into any directory where it is granted a permission. In one research compendium I had untracked that file by hand on one machine, but had not added an ignore rule. Some days later the sweep on a second machine found three such files (at the repository root and in two subdirectories), committed them, and pushed. The next `git pull` on the first machine refused to run:

```
error: The following untracked working tree files would be
overwritten by merge:
    .claude/settings.local.json
    analysis/report/.claude/settings.local.json
    docs/.claude/settings.local.json
Please move or remove them before you merge.
Aborting
```

Nothing was lost, but an audit of the other repositories found the same files sitting unignored in twenty-six more, each one sweep away from the same commit. The sweep had to go. The question was what to put in its place, since its benefit (remotes that stay current without my remembering to push) was real.

This post describes the replacement: two small hooks that divide the sweep's job along the line it should have respected all along, the line between what I decided to commit and what merely exists on disk.

Motivations

- Remotes should stay current without a manual `git push` after every commit.
- Nothing should ever be staged or committed without a deliberate action by the user.
- Uncommitted work should not be silently stranded on one machine when attention moves elsewhere.
- Safety checks must never block scripts, `make`, editors, or agent tools that run a shell non-interactively.
- Each repository's own hooks must keep working.

Objectives

1. Install a global git `post-commit` hook that pushes the current branch after each commit, with explicit conditions under which it declines.
2. Preserve repository-local hooks under a global `core.hooksPath`.
3. Install a zsh hook that, on leaving a repository, lists uncommitted changes and offers to commit tracked files.
4. Test both hooks against throwaway repositories, including the cases in which they must do nothing.



What This Does Not Cover

- **Backups.** Neither hook is a backup system. Work that is never committed is never pushed. A file-level backup (Time Machine, Borg, or similar) remains necessary.
- **Bash.** The leave check uses zsh hook functions. Bash has no `chpwd` equivalent; it can be approximated through `PROMPT_COMMAND`, but that is not shown here.
- **Team repositories.** Automatic pushing suits a single author who works on several machines. On a shared branch with review requirements it is the wrong default; the hook's opt-out exists for that case.

Prerequisites

- git 2.9 or later (for `core.hooksPath`). Tested with git 2.56.0.
- zsh 5.x for the leave check. Tested with zsh 5.9.2.
- Repositories whose branches track an upstream (`git push -u` once per branch). The hook never creates remote branches.

Part 1: Pushing Only What Was Committed

The division of labor

The sweep combined three decisions into one unattended step: what to stage, whether to commit, and when to push. Only the last is safe to automate. Staging and committing express intent; pushing merely propagates intent already expressed. A `post-commit` hook sits exactly at that boundary: it runs only after the user has committed, and it has nothing to decide except whether to push.

Global hooks without losing local ones

A hook that should run in every repository belongs in a single directory named by `core.hooksPath`:

```
mkdir -p ~/.config/git/hooks
git config --global core.hooksPath ~/.config/git/hooks
```

This setting has a side effect that is easy to miss. When `core.hooksPath` is set, git stops looking in each repository's `.git/hooks/` directory altogether. Any repository that relied on its own `pre-commit` hook (a linter, a guard against committing to a deploy branch) silently loses it.

The remedy is a small dispatcher. Every hook name in the global directory is a symlink to one script, which forwards to the repository-local hook of the same name when one exists:

```
#!/usr/bin/env bash
# Global hook dispatcher. Setting core.hooksPath makes git ignore each
# repository's own .git/hooks directory. Symlink every hook name in
# the global hooks directory to this file so that a repository-local
# hook of the same name still runs. --git-common-dir is used because
# --git-path hooks would resolve back to core.hooksPath.

hook_name=$(basename "$0")
common_dir=$(git rev-parse --git-common-dir 2>/dev/null) || exit 0
local_hook="$common_dir/hooks/$hook_name"

if [[ -x "$local_hook" ]]; then
  exec "$local_hook" "$@"
fi
exit 0
```

`--git-common-dir` rather than `--git-path hooks` matters: the latter honors `core.hooksPath` and therefore points back at the global directory, which would make the dispatcher call itself.

The post-commit hook

`post-commit` is the one hook that does its own work, so it is a real file rather than a symlink. It first forwards to any local `post-commit`, then decides whether to push:

```
#!/usr/bin/env bash
# Push the current branch to its upstream after every commit.
#
# Only commits the user deliberately made are pushed; nothing is ever
# staged on the user's behalf.
#
# Skips, without pushing:
#   - GIT_AUTOPUSH=0 in the environment (one commit)
#   - repositories with hooks.autopush = false
```

```
#   - repositories with a root netlify.toml, unless hooks.autopush = true
#   - detached HEAD, and branches with no upstream (never creates
#     remote branches)
#   - commits made mid-rebase, merge, cherry-pick, revert, or bisect
#
# The push runs in the background; outcomes are appended to a log.
# Set GIT_AUTOPUSH_SYNC=1 to push in the foreground instead.

# Run the repository-local post-commit hook first, if there is one.
common_dir=$(git rev-parse --git-common-dir 2>/dev/null)
if [[ -n "$common_dir" && -x "$common_dir/hooks/post-commit" ]]; then
  "$common_dir/hooks/post-commit" "$@"
fi

[[ "${GIT_AUTOPUSH:-1}" == 0 ]] && exit 0
autopush=$(git config --bool hooks.autopush)
[[ "$autopush" == false ]] && exit 0

# A push to a Netlify-backed repository deploys a live site.
toplevel=$(git rev-parse --show-toplevel) || exit 0
[[ -f "$toplevel/netlify.toml" && "$autopush" != true ]] && exit 0

git_dir=$(git rev-parse --git-dir) || exit 0
for marker in rebase-merge rebase-apply MERGE_HEAD CHERRY_PICK_HEAD \
  REVERT_HEAD BISECT_LOG; do
  [[ -e "$git_dir/$marker" ]] && exit 0
done

branch=$(git symbolic-ref -q --short HEAD) || exit 0
remote=$(git config "branch.$branch.remote") || exit 0
merge_ref=$(git config "branch.$branch.merge") || exit 0
[[ "$remote" == . ]] && exit 0

log_dir=${XDG_STATE_HOME:-$HOME/.local/state}
log_file=${GIT_AUTOPUSH_LOG:-$log_dir/git-autopush.log}
mkdir -p "$(dirname "$log_file")"

do_push() {
  local stamp out
  stamp=$(date '+%Y-%m-%d %H:%M:%S')
  # A temporary index from `git commit <paths>` must not leak into
  # the push.
  unset GIT_INDEX_FILE
  if out=$(git push --quiet "$remote" "HEAD:$merge_ref" 2>&1); then
    echo "$stamp OK   $toplevel $branch -> $remote/${merge_ref#refs/heads/}"
  else
    echo "$stamp FAIL $toplevel $branch -> $remote/${merge_ref#refs/heads/}"
```

```
    echo "$out" | sed 's/^/    /'
  fi >> "$log_file"
}

if [[ "${GIT_AUTOPUSH_SYNC:-0}" == 1 ]]; then
  do_push
else
  do_push </dev/null >/dev/null 2>&1 &
  disown
fi
exit 0
```

When it declines to push

Each skip condition answers a specific failure that automatic pushing would otherwise cause.

Condition	Why the hook does not push
`GIT_AUTOPUSH=0 git commit ...`	One-off: a commit the user wants to hold locally
`git config hooks.autopush false`	Per-repository opt-out (shared or reviewed branches)
Root `netlify.toml`	A push deploys a live site; it must be deliberate
Rebase, merge, cherry-pick, revert, bisect in progress	Each replayed commit would otherwise trigger a push of a half-finished history
Detached `HEAD`	There is no branch to push
No upstream, or upstream is a local branch	The hook never creates remote branches

The background push keeps `git commit` fast; network latency never reaches the terminal. The cost is that failures are also invisible there. They go to the log instead, which should be checked whenever a remote looks stale:

```
tail -n 20 ~/.local/state/git-autopush.log
```

Installation

```
cd ~/.config/git/hooks
# copy _dispatch and post-commit here, then:
chmod +x _dispatch post-commit
for h in pre-commit prepare-commit-msg commit-msg post-checkout \
  post-merge pre-push pre-rebase post-rewrite; do
  ln -sf _dispatch "$h"
done
git config --global core.hooksPath ~/.config/git/hooks
```

Then retire the sweep itself (its launchd agent, cron entry, or systemd timer). Running both defeats the purpose.



Part 2: A Check on the Way Out

The gap the push hook leaves

The push hook covers only what was committed. The sweep, for all its faults, also caught the opposite case: the afternoon's edits that were never committed at all, left behind when attention moved to another project. Without the sweep, that work stays on one machine until I happen to return to it.

The natural moment to catch it is when I leave the repository. Git has no such event; its hooks fire only on git operations. The shell does have one. zsh calls every function registered in the `chpwd` hook array after any change of directory in the current shell, whatever caused it: `cd`, `pushd`, `popd`, or a jump helper such as autojump's `j` or zoxide's `z`, all of which end in an ordinary `cd`. A second array, `zshexit`, runs when an interactive shell exits.

The module

```
# Offer to commit uncommitted work when leaving a git repository.
#
# Fires when the repository root changes (cd, pushd, popd, autojump,
# zoxide; anything that changes directory in the current shell) and
# when an interactive shell exits inside a repository. Lists the
# changes and offers `git commit -a`: tracked files only. Untracked
# files are listed but never added.
#
```

```
# Silent in non-interactive shells and when stdin or stdout is not a
# terminal, so scripts, make, editors, and agent tools never block.
# Opt out for a shell with GIT_LEAVE_CHECK=0.
#
# Source from ~/.zshrc:  source ~/.config/zsh/git-leave-check.zsh

typeset -g _GIT_LEAVE_LAST_ROOT=""

_git_leave_check() {
  emulate -L zsh
  local root=$1
  [[ "${GIT_LEAVE_CHECK:-1}" == 0 ]] && return
  [[ -o interactive && -t 0 && -t 1 ]] || return
  local -a lines tracked
  lines=(${(f)"$(command git -C "$root" status --porcelain 2>/dev/null)"})
  (( ${#lines} )) || return
  tracked=(${lines:#\?\?*})
  local shown=${root/#$HOME/~}
  print -P "%F{yellow}Leaving ${shown//\%/%%} with uncommitted changes:%f"
  print -rl -- "${(@)lines[1,15]}"
  (( ${#lines} > 15 )) && print -r -- "  ... and $(( ${#lines} - 15 )) more"
  if (( ! ${#tracked} )); then
    print -r -- "Only untracked files; add them explicitly to commit them."
    return
  fi
  local reply msg ask='[c]ommit tracked changes, any other key to skip: '
  read -k 1 "reply?$ask" || return
  print
  [[ $reply == [cC] ]] || return
  read "msg?Commit message: " || return
  if [[ -z ${msg//[[:space:]]/} ]]; then
    print -r -- "Empty message; not committing."
    return
  fi
  command git -C "$root" commit -a -m "$msg"
}

_git_leave_chpwd() {
  emulate -L zsh
  local root prev
  root=$(command git rev-parse --show-toplevel 2>/dev/null)
  [[ "$root" == "$_GIT_LEAVE_LAST_ROOT" ]] && return
  prev=$_GIT_LEAVE_LAST_ROOT
  _GIT_LEAVE_LAST_ROOT=$root
  [[ -n "$prev" ]] && _git_leave_check "$prev"
}
```

```
_git_leave_zshexit() {
  emulate -L zsh
  local root
  root=$(command git rev-parse --show-toplevel 2>/dev/null) || return
  _git_leave_check "$root"
}

autoload -Uz add-zsh-hook
add-zsh-hook chpwd _git_leave_chpwd
add-zsh-hook zshexit _git_leave_zshexit
_GIT_LEAVE_LAST_ROOT=$(command git rev-parse --show-toplevel 2>/dev/null)
```

An illustrative session, jumping with autojump from a paper to a package:

```
$ j zztable1
/Users/me/prj/zztable1
Leaving ~/prj/paper with uncommitted changes:
 M analysis/report/report.Rmd
?? scratch.R
[c]ommit tracked changes, any other key to skip: c
Commit message: Revise discussion section
[main a1b2c3d] Revise discussion section
 1 file changed, 12 insertions(+), 4 deletions(-)
```

Design decisions

Tracked files only. The module offers `git commit -a`, which stages modifications and deletions of files git already tracks. It never runs `git add -A`. Untracked files are listed so that they are noticed, but adding one remains a deliberate act. This is the lesson of the sweep applied directly: the file that caused the incident was untracked.

Repository boundaries, not directories. The module compares repository roots, so moving between subdirectories of one repository is silent. Moving into a nested repository counts as leaving the outer one.

Interactive terminals only. The guard `[[ -o interactive && -t 0 && -t 1 ]]` matters more than it appears. Many tools start a shell that reads `~/.zshrc`: editor integrations, build systems, and coding agents among them. A prompt that waits for a keypress inside such a shell hangs the tool. With the guard, the module is inert anywhere a person is not at the keyboard.

Composes with other directory hooks. `add-zsh-hook` appends to the `chpwd_functions` array rather than replacing it, so the module coexists with autojump's own `chpwd` function (which records the directory) and with anything else registered there. In my own setup the check lives inside an existing `chpwd` function that warns when the repository being entered is behind its upstream; the leave check runs first, so a commit and its push complete before the warning about the next repository appears.

Installation

```
mkdir -p ~/.config/zsh
# copy git-leave-check.zsh there, then add to ~/.zshrc:
source ~/.config/zsh/git-leave-check.zsh
```

How the Two Hooks Work Together

The hooks were written separately but compose: a commit made at the leave prompt is an ordinary commit, so the `post-commit` hook pushes it. Leaving a repository therefore becomes a two-keystroke route from uncommitted edits to an up-to-date remote, with a human decision at the only point that requires one.

One consequence deserves explicit mention. The push hook pushes the branch, not the commit: `git push <remote> HEAD:<branch>` sends every local commit the remote lacks. A commit deliberately held back with `GIT_AUTOPUSH=0` will therefore be published by the next ordinary commit on the same branch. To hold work locally for longer, set `hooks.autopush false` on the repository, or keep the work on a branch without an upstream.

Testing

Both hooks were exercised against throwaway repositories rather than real ones.

The push hook was tested against a local bare repository standing in for the remote, with `GIT_AUTOPUSH_SYNC=1` so that each result could be checked immediately. Eight cases: a default commit is pushed; `GIT_AUTOPUSH=0`, `hooks.autopush false`, and a root `netlify.toml` each leave the remote unchanged; `hooks.autopush true` overrides `netlify.toml` and pushes the accumulated commits; a branch with no upstream is not created on the remote; repository-local `pre-commit` and `post-commit` hooks still run through the dispatcher; and a rejected non-fast-forward push is written to the log.

The leave check needs a real terminal, because it reads single keypresses. It was driven with `expect`, which spawns zsh under a pseudo-terminal and answers each prompt only after the prompt appears. (Piping keystrokes in through `script` does not work: the input arrives before zsh reads it, followed by an end-of-file.) Eight cases: committing on leave, with an untracked file correctly left out; moving within a repository and its subdirectories without a prompt; declining; untracked files only, which produces a note but no prompt; a whitespace-only commit message, which is refused; the exit hook; a non-interactive shell, which stays silent; and `GIT_LEAVE_CHECK=0`.

Things to Watch Out For

1. **Silent push failures.** A background push that is rejected (most often because another machine pushed first) leaves no trace in the terminal. Check the log when a remote looks stale, or set `GIT_AUTOPUSH_SYNC=1` while diagnosing.

2. **'Everything up-to-date' after a commit.** A manual `git push` right after committing usually reports nothing to do, because the background push has already finished. That message is correct, not a sign of failure.
3. **Held commits leak.** As above, a commit made with `GIT_AUTOPUSH=0` is pushed by the next unflagged commit on the same branch.
4. **Ignore rules with a slash are anchored.** The natural repair for the incident was a `.gitignore` line reading `.claude/settings.local.json`. It ignores only the file at the repository root. A pattern containing a slash anywhere but the end is relative to the `.gitignore` that contains it; to match at any depth, write `**/.claude/settings.local.json`. Confirm with `git check-ignore -v <path>`.
5. **Untracking does not ignore.** `git rm --cached` removes a file from the index for the current commit only. Without an ignore rule, the next `git add -A` on any machine re-adds it. That is precisely how the incident began.
6. **Not every exit runs `zshexit`.** Closing a terminal window sends the shell a hangup, and the prompt cannot be answered when no terminal remains. Editors and GUI git clients never trigger either hook. The leave check is a reminder, not a guarantee.
7. **Nested repositories prompt more often.** A blog whose posts are separate repositories treats every move between posts as leaving a repository.



Lessons Learned

The sweep's design looked economical because it reduced three decisions to one scheduled job. The cost of that economy was invisible until a file that no one had decided to share was shared across machines. Separating the automatable step (propagation) from the steps that express intent (staging and committing) removed the failure without giving up the convenience.

A second lesson came from the repair. The first ignore rule I wrote was wrong for two of the three files, and only a status check after the push revealed it. Verifying a fix with the tool's own diagnostic (`git check-ignore -v`) is cheaper than inferring that it worked.

Limitations

- The push hook assumes a single author per branch. On branches that others also push to, rejected pushes will be frequent and will accumulate in the log.
- The leave check covers only interactive zsh sessions. Work edited in an IDE and never touched from a terminal is not checked.
- Both hooks act per machine. A machine that still runs the old sweep reintroduces the original problem, and a global ignore rule must be installed on every machine (or kept in a synced dotfiles repository) to be effective.

Opportunities for Improvement

- A `repos-dirty` command that lists every repository under a project root with uncommitted or unpushed work, for an end-of-day check that does not depend on leaving each repository.
- A periodic notification when the push log contains new `FAIL` entries, so that rejected pushes are not discovered days later.
- A `PROMPT_COMMAND` port of the leave check for bash users.

Wrapping Up

An unattended `git add -A && commit && push` sweep conflates intent with existence. Two small hooks separate them again: a global `post-commit` hook that propagates every deliberate commit, and a zsh hook that offers a deliberate commit at the moment work would otherwise be stranded.

What Did We Learn?

- Automate propagation, not intent.
- `core.hooksPath` disables per-repository hooks unless a dispatcher forwards to them.
- The shell, not git, owns the 'left a repository' event.
- Prompts belong only in interactive terminals.
- Untracking a file without ignoring it defers the problem; an ignore pattern containing a slash is anchored to its directory.

See Also

Related posts:

- Setting Up Git for Data Science Workflows: the baseline git configuration these hooks extend.

- Multi-Laptop macOS Bootstrap: keeping hook scripts and shell modules in a versioned dotfiles repository so that every machine runs the same guardrails.
- Refactoring a Personal Toolbox: why the leave check is a sourced shell function while the push hook is a script.

Key resources:

- githooks(5): hook names, arguments, and the effect of `core.hooksPath`.
- gitignore(5): pattern anchoring rules.
- zsh manual: hook functions: `chpwd`, `zshexit`, and the `*_functions` arrays.

Reproducibility

Tested configuration:

Component	Version
Operating system	macOS 26.6.2
Shell	zsh 5.9.2
git	2.56.0
expect	system
Last verified	2026-11-11

Configuration files:

- `analysis/configs/hooks/_dispatch`: the hook dispatcher.
- `analysis/configs/hooks/post-commit`: the auto-push hook.
- `analysis/configs/git-leave-check.zsh`: the leave-check module.

The code listings in this post are copies of these files.

AI involvement. The hook code, the test harnesses, and the first draft of this post were produced with Claude Code (Anthropic), working from my existing dotfiles hooks and from an incident it helped diagnose. The tests described above were run in that session. The decision to retire the sweep, the design constraints (tracked files only, no prompts outside interactive terminals), and editorial judgment over the final text are mine.

Rendered on 2026-11-11 at 16:19 PST. Source: ~/prj/web/ryyblog/posts/sh-autopush-and-leave-check/analysis/report/index.qmd

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

Corrections, suggestions, and questions are welcome.

Related posts in this cluster

This post is part of the *Shell Scripting and Git Tooling* series. Recommended reading order:

1. Refactoring a Personal Toolbox: Scripts versus Shell Functions
2. A Mac Workflow for Tracking Daily Research Progress
3. **Two Git Guardrails: Auto-Push on Commit and a Check on the Way Out** (this post)