Pull-Request Workflow with Agents

Last modified: 2026-07-15 23:18:00 (PDT)

The practices below apply whether you are driving an agent's work or doing the work yourself. They keep parallel sessions from colliding and keep a pull request moving toward a clean, mergeable state.

File an Issue Before Starting

When starting a **new** piece of work, go **issue-first**: before branching, editing, or opening a PR, make sure a tracking issue exists. Search the tracker first; if no open issue covers the task, **file one** (`gh issue create` / `glab issue create`), then proceed. Never jump straight into a PR without a tracking issue behind it.

The issue is the durable record of intent, scope, and "done" criteria — it gives reviewers context, lets the PR auto-close it via `Closes #N`, and keeps the work discoverable even if the PR stalls. Skip only when the task is already tracked by an open issue.

When the issue is a **bug report**, include a minimal reproducible example (a reprex — https://reprex.tidyverse.org/) whenever you can. A reprex is what a maintainer needs to confirm and fix the bug, and it's what they'll ask for anyway, so providing it up front saves a round trip. The `reprexes` skill helps reduce the problem to a minimal, self-contained example.

When filing an issue that contains a list of independent subissues, file each subissue as a child issue linked under the parent (GitHub sub-issues feature: `mcp__github__sub_issue_write` in remote sessions, or `gh api` with the sub-issues endpoint in local sessions).

Claim a PR or Issue Before Working on It

Before starting a work session on a GitHub PR or issue — i.e. before fetching the branch, making edits, or invoking an automated review cycle — post a brief comment on the PR/issue so other people and any automated review bots know not to start a conflicting parallel session.

Use:

```
gh pr comment <N> --body "Working on this --- paws off until I'm done."
gh issue comment <N> --body "Working on this --- paws off until I'm done."
```

Then proceed with the work. After the session ends (PR merged, issue closed, or work otherwise paused), follow up with a closing comment so the PR/issue is unclaimed for the next person.

Skip the claim step if the most recent comment already says you are working on it. This applies to any task that will push commits to a PR branch or run iterative review loops. It does **not** apply to read-only inspection (showing a PR, checking status, explaining a diff) — those don't risk a parallel session.

This includes a PR **you opened yourself**: in repos with an active `@claude` agent (`claude.yml`), the agent can push commits to your branch on PR activity — e.g. merging `main` in — and collide with your in-flight push, so claim early to flag the branch as actively worked. (See `memories/tools.md`, "(**claude?**) CI action", for the collision-recovery steps.)

When starting work from an issue, follow the claim comment with an immediate draft PR — see `pr-on-claim` for the mechanics. An open PR is a stronger "in-flight" signal than a comment alone.

Handing off mid-task to another agent, on user request ("finish what you're doing, then relinquish holds; I'll put another agent on them"): don't just stop — leave the next agent a clean starting point. On each claimed PR/issue: (1) post a status comment on the PR itself distinguishing what's **done** from what's genuinely **not done** (the actual point of the issue, not just the side-fixes found along the way) and any blocker still open, so the next agent doesn't have to re-derive it from the diff; (2) post the closing/unclaim comment on the issue per the pattern above; (3) `unsubscribe_pr_activity` (or stop babysitting locally) so you don't keep auto-fixing a PR you no longer own; (4) stop any background watch/poll task tied to that work (e.g. a `ScheduleWakeup` or a `Monitor`/background-Bash wait) so it doesn't fire into a session that's moved on. A merge-conflict-free `git status` and a pushed branch are not enough on their own — the status comment is what makes the handoff legible. (ucdavis/bcs `gia` session, 2026-07-06: handed off PRs #310 and #311 mid-implementation this way, each blocked on the same slow `renv::restore()`.)

Keep Your Branch Synced with Main

Whenever `main` has moved ahead of a PR branch you're working on, **merge `main` into the PR branch** before the next push or review trigger. Don't wait for a conflict to surface or for someone to ask.

This fragment covers the single-branch-vs-`main` case. When orchestrating a multi-agent `ultracode` session, merges can happen at more points than that — see `ultracode-merge-conflicts` for the broader check (worktree-isolated agent branches, concurrent `parallel()` results) and the note on GitHub's mergeable indicator not evaluating custom `.gitattributes` merge drivers.

Always check for merge conflicts with main before pushing results to remote. Run this before every push, not just before triggering a review:

```
git fetch origin main
git log --oneline ..origin/main | head    # any commits? main is ahead --- merge it in
git merge origin/main
```

If the push is rejected because `main` has moved (`! [rejected]` with `(fetch first)` or `(non-fast-forward)`), fetch and merge before retrying — don't force-push.

Always do this before triggering a fresh review too, so the reviewer evaluates the PR against current `main` rather than a stale snapshot.

Don't rebase or squash-rewrite a published PR branch unless explicitly asked — a merge commit is the right move because it matches GitHub's "Update branch" button and preserves the PR history.

If the merge has conflicts, resolve them, run the project's standard pre-commit checks (render / lint / spell / tests), commit, then push. Don't push a half-resolved merge.

After merging main, re-check version parity. In R packages with a `version-check` CI job, the branch's `DESCRIPTION Version:` must *exceed* main's. A conflict-free merge can silently put them at parity — main advanced (e.g. another PR merged between when you last bumped and now). After every merge of main, compare versions:

```
git show origin/main:DESCRIPTION | grep ^Version
grep ^Version DESCRIPTION
```

If they match, bump the branch's `Version:` by one patch level before pushing.

Re-check `main` again right before the final push, not just at the start of a merge. Resolving a conflict (rerunning generators, fixing prose, updating a CHANGELOG entry) can take long enough for `main` to advance a second time. A `git fetch origin main` immediately before `git push` — after conflict resolution is done, not only before it started — catches that case; an earlier CI failure on a commit you thought was current is a symptom of skipping this second check.

A conflict-free merge does not mean derived artifacts are in sync. If your branch regenerates a generated tree (e.g. `codex-skills/`, a lockfile, rendered docs) and `main` added a

new *source* input the generator consumes (a new skill, a new dependency), git merges both cleanly — but the generator never ran against the new input on your branch, so its output is missing or stale and the sync check fails on `main` after both land. After merging `main`, re-run the generator and commit the result whenever main touched the generator's inputs — don't trust the absence of conflicts. (Concretely: merge the PR that adds the new skill *first*, then sync the wrapper-regenerating branch and rerun `scripts/sync-codex-skill-wrappers.py` before merging it.)

A CI failure on a brand-new PR's very first commit (e.g. the empty claim-commit from `pr-on-claim`) is a signal to check `main`'s position before debugging the failure itself. A local checkout that sat around since before the session started can already be many commits behind `main` — the failure (a stale generated-tree check, a check `main` has since added or dropped) often isn't a real problem with your change at all, just `main` having moved. `git fetch origin main && git log --oneline ..origin/main` first; if `main` is ahead, merge it in and re-run the checks before treating the failure as something to fix in the diff. (`stack-prs` #359: an empty-commit draft PR failed `validate` on a stale `codex-skills/` generated tree, and the `require-changelog` job on a newly-added `CHANGELOG.md` requirement from PR #354 — both were `main` having advanced past a checkout that predated the session, not a defect in the new skill.)

A real conflict inside a file whose logic is also copied elsewhere (an extracted script, a doc example) needs the copy re-synced too, not just the conflicted file resolved. When a PR extracts inline logic (e.g. a workflow step's shell block) into a standalone script for testability, and `main` independently changes that same inline logic while the PR is open, resolving the merge conflict in the workflow file is not enough — the extracted script must be updated to match `main`'s new logic exactly, or the PR silently reverts `main`'s fix the moment it merges. Diff the extracted copy against `main`'s current inline version line-for-line (strip indentation, `diff`) to confirm an exact match, not just "looks about right." If the PR carries tests against the extracted copy (fixtures, unit tests), add regression coverage for whatever `main`'s change fixed — the merge is the natural moment to catch a gap the original PR's tests didn't anticipate, and to prove the new fixtures actually catch the regression (temporarily revert the fix, confirm the test fails, then restore). (gha#176: `main` landed #173's lenient verdict-matching fix to `claude-code-review.yml`'s inline fail-check logic while a PR extracting that same logic to `scripts/check-review-execution.sh` was still open; the conflict resolution updated the script to match verbatim and added two new fixtures for #173's specific fix, verified to fail against the pre-fix logic.)

A textual conflict in a skill file can be the symptom of a conceptual duplicate, not just competing edits to the same line. When merging `main` into a branch that's authoring a new skill, if the conflict lands in a `## Relationship to other skills` section (or `main` added an entirely new skill in the same territory), that's a signal to re-run `skill-builder`'s Step 0 judgment — not just resolve the diff mechanically. Compare the new skill against whatever landed on `main`: are they the same concern (fold into one, redirect), or genuinely

distinct (cross-link both directions so neither reads as an unexplained near-duplicate)? `skill-builder`'s in-flight-work scan only runs once, at the start; `main` can grow a colliding skill in the time a PR is open, so the check has to be repeated at merge time too. (PR #352's `check-info-quality` landed alongside `#344`'s independently-authored `fact-check-prose` this way — distinct enough to keep both, resolved by adding an explicit boundary in each skill's Relationship section rather than consolidating.)

Two PRs that each append a new terminal numbered subsection to the same file (e.g. `### 5. ...` in a `CLAUDE.md` review-guidelines list) will conflict on merge even when neither side's content actually disagrees. This isn't an editorial clash — it's two authors both writing to "the next number" at the same insertion point. Resolve by keeping **both** additions and renumbering sequentially from the collision point on, not by dropping either side; then grep the file for any other place that names the old numbering (a cross-reference, an index). This is also a reason `fully-clean`'s CI-green-and-review-clean verdict is a snapshot, not a mergeability guarantee — `main` can pick up its own append in the same spot after your last review round, so a PR can go from "reviewed clean" to "needs a merge conflict resolved" with no defect in its own diff. Before reporting a PR ready to merge, re-check with `git fetch origin main` plus the `git merge-tree` command from `resolve-conflicts`, not just a cached `mergeable` flag or an earlier green CI run. (gha#211: `main` merged #209's own new `### 5. Check for AI-generated prose tells` subsection between this PR's clean review and its actual merge — `git merge-tree` surfaced a real conflict that neither PR's own CI nor review status had flagged, since neither had rerun since `main` advanced.)

After merging a PR that extracts an inline block into a reusable unit (a composite action, a shared script/function), check other open PRs that still edit that same inline block — your merge just broke their textual diff, even though their intended change is usually trivial to re-apply to the new location. This is the mirror image of the case above: there, you're the one resyncing after `main` moved a copy of your logic; here, *you* are the one who moved the logic, so the burden of noticing and fixing the resulting conflict falls on you, not on the sibling PR's author waiting to hit it. Don't wait for that PR's own merge/CI to surface the conflict — check every open PR touching the same file right after your extraction merges: `git merge-tree "$(git merge-base origin/main origin/<sibling-branch>)" origin/main origin/<sibling-branch>` (or `gh pr diff <N>` against the new `main`) shows whether it still applies cleanly. Re-apply the sibling PR's actual semantic change (not a mechanical `--theirs`) to the new location, verify with a direct diff that the extracted unit now differs from `main` by exactly that PR's intended change and nothing else, then push to their branch and flag what you did in a PR comment. (gha#201 extracted `claude-code-review.yml`'s `claude_args` block into a new `run-claude-review-attempt` composite action to support a retry; gha#202, open in parallel, edited that same inline block to allowlist `WebFetch`/ `Bash(curl:*)`. Proactively rebasing #202 and re-applying its allowlist change to the new composite action — rather than leaving its author to discover a conflict — let it merge within the hour instead of stalling.)

An add/add conflict on a *shared config file* usually means two PRs independently

fixed the same root cause — reconcile the reasoning, don't just pick a side. This generalizes the skill-file case above beyond skills: a repo-wide CI/lint/build config fix (a new tool config file, a workflow tweak) is exactly the kind of change multiple sessions or bots are likely to attempt in parallel once a check starts failing on `main` for everyone. When the conflict is a whole-file add/add (not just competing edits to an existing file), read both sides' reasoning — code comments, commit messages, the PR discussion — before resolving; usually one side's explanation is more complete (covers a case the other missed, cites the tool's actual constraint) and should win outright rather than mechanically merging fragments of both. Re-diff the PR against `origin/main` after resolving to confirm the PR's remaining changes are its own original scope, not a reintroduction of what the other, now-merged PR already added. (`d-morrison/altdoc#7` vs `#18`: both independently added a `jarl.toml` excluding the same fixture directory for the same `jarl-check` failure; `#18` merged first, `#7`'s merge conflicted on the new file, resolved by keeping `#18`'s more detailed comment and re-confirming `#7`'s diff against `main` was back down to just its own four files. This same "append-collision" pattern struck a third time one insertion point over: this bullet and the two above it were each added by independent PRs landing in quick succession, all appending after the same "PR #352's `check-info-quality`…" paragraph — resolved, per the guidance above, by keeping all three rather than picking one.)

A `dirty` `mergeable_state` on a bot-opened PR can mean a sibling PR already closed the same issue, not just that `main` drifted. An issue-triggered `@claude` workflow can fire twice on the same issue in quick succession (a duplicate dispatch, or two people independently routing the same request), producing two independent PRs that both fully resolve it — including adding the identical new file. The second PR's merge conflict is an add/add on that new file, and it looks like ordinary main-drift, but treating it that way and mechanically resolving in favor of "ours" silently reintroduces a duplicate the other PR's merge already published. Before resolving, check the PR's linked issue for **other** cross-referenced PRs/closing events — if one already merged and closed it, diff the conflicting file against `main`: if it's the sibling PR's already-published version, keep `main`'s content and keep only this PR's genuinely distinct remainder (a piece the sibling PR never did), rather than re-adding a second copy. (`ai-config#501`: issue #500 was independently resolved twice — `#502` merged first, adding `shared/writing/math-derivation-steps.md` and closing #500, but never wiring it into `CLAUDE.md`; `#501` added a second copy of the same fragment plus the missing `CLAUDE.md` wiring. Resolved by keeping `main`'s published fragment and `#501`'s wiring, turning a `dirty` merge into a clean `+8/-0` diff.)

A merge into a growing numbered list (e.g. `gha`'s `CLAUDE.md` "Code review guidelines" section) can produce zero blank lines between two adjacent headings even with no textual conflict — lint catches it, git doesn't. When a section is a hotspot several PRs independently append items to (each PR adding its own `### N.` block at the end), a clean three-way merge can still splice one PR's closing line directly against the next PR's heading with no blank line between them — this doesn't produce a `<<<<<<<` conflict marker (git resolves it as a straightforward insertion), so it's easy to push without noticing. `markdownlint`'s MD022 (blanks-around-headings) is what actually catches it, as a CI failure with no proximate code

change to explain it. Re-run the repo's markdown lint (or at minimum re-read the diff around every `### N.` boundary you didn't personally write) after any merge that touches a shared growing list, not just after a merge with conflicts. (gha#208: an out-of-band merge from `main` — done by a different session, not the one that opened the PR — landed a new item 7 directly against the PR's own item 6 with no blank line; `lint-markdown`'s MD022 failed with no conflict marker anywhere in the diff to point at.)

Driving a Pull Request to Clean

Whenever you are working a PR/MR, run the full **ARDI** loop by default, without being asked: **A**ddress every flagged item, **R**ebut findings that are wrong, **D**efer out-of-scope items to tracked issues, then **I**terate with a fresh review — repeating until the latest review is **fully clean**. Don't stop at "review-clean, just needs approval" and hand triage back; keep the cycle going until it's genuinely clean.

The loop's terminal action is to **report the PR ready, not to merge it**. Merging is human-gated — it happens only on an explicit human "merge it" (the `merge-it` skill), never as a step ARDI takes on its own. So when you carry a PR across a `ScheduleWakeup` or `/loop` wait, **never** bake a self-merge directive like "if clean and CI green, merge it" into the wakeup/loop prompt: a scheduled prompt fires back as a user-role turn, so a self-authored "merge it" only *looks* like human approval (and Claude Code's auto-mode classifier will rightly deny it as a self-authored merge). Drive to fully clean, report ready, and leave the merge — and any other destructive one-off, e.g. a `gh workflow run` that force-pushes — for explicit human authorization.

The one exception: if the human has explicitly granted the `mwc` (merge-when-confident) session permission, that grant is a live human instruction, not a self-authored one, so baking a self-merge step into a wakeup/loop prompt is fine for the rest of that session. See `mwc` for the grant's scope and limits.

In the **clear-all family** (`ardia`, `gia`, `gii`, `gip`), "report ready, don't merge" gates only the merge — it does **not** pause the sweep. A clean-but-unmerged PR is not a stop; move to the next item, and stack it when it isn't naturally independent of that PR. See `stack-dont-pause`.

Self-review against the project's own stated conventions before every push, not just the first — and don't just re-read the criteria, actually run the applicable review skills against your own diff and iterate on what they find, the same ARD cycle you'd run against an external reviewer's findings. Don't treat the review bot as the mechanism that discovers a project's documented conventions — self-apply them first. When a project's own `CLAUDE.md` (or equivalent agent doc) already states specific criteria — a DRY/no-duplication rule, a doc-sync checklist for a new input, a changelog-category rule, a citation requirement, a "new logic needs test coverage" norm, a prose-quality check like `fact-check-prose`, `fix-forward-references`, or `detect-informal-definitions` — a

first-pass implementation checked only against feature correctness forces the review loop to spend a round re-deriving what the project's own docs already said. Before every push, re-read the project's own stated review criteria and actually invoke the review skills/checks it names against the diff (not just recall them from memory), the same way an external reviewer would apply them. Address every finding your own self-review surfaces — fix, rebut, or defer, exactly like the ARD step above — before the push goes out; a self-review that finds issues and pushes anyway has only moved the round to the external reviewer instead of skipping it. Repeat until your own self-review pass is clean, then push. (gha#219/#220: one review round surfaced five findings — a DRY duplication, an incomplete-coverage doc overclaim, a wrong changelog category, an uncited claim, and missing test coverage for new logic — all catchable this way, since each was a direct match against gha's own `CLAUDE.md` conventions, not new information the review surfaced.)

Proactively self-correct a technical claim you already told a reviewer, the moment further testing shows it was wrong — don't wait for the reviewer to catch it. If you stated a rationale (an approach is safe, a risk doesn't apply, a backstop exists) and then discover through your own follow-up verification that it's false, post the correction with the actual evidence immediately, rather than leaving the stale claim standing until a review round re-raises it. This keeps the review loop converging instead of churning on a claim you already know is wrong. (d-morrison/rme#989 / ucdavis/epi204#363: after telling both reviewers `references.bib` didn't share `CLAUDE.md`'s union-merge corruption risk, a follow-up merge simulation showed it does — posted the correction with repro steps on both PRs before either reviewer re-raised it.)

What "Fully Clean" Means

"Fully clean" is the terminal state the ARDI review loop drives toward. A PR/MR is **fully clean** when **both** of these hold:

1. **All CI workflows are green.** Every workflow passes — not just the required checks and not just the review job. This includes non-gating checks like the Coverage / codecov job: don't merge around a red Coverage run just because it isn't a required check, unless there's a specific, stated reason for that merge (the project wants to maintain decent coverage, so a red Coverage job is a real signal to fix, not to ignore). **`codecov/patch` is a separate check from the repo's own Coverage workflow job, and both must be green.** The Coverage job runs the coverage-instrumented test suite; `codecov/patch` is the Codecov service's own status check, gating the PR's DIFF against a minimum patch-coverage percentage — a repo can have a fully green Coverage job while `codecov/patch` still fails (uncovered new lines in the diff). When delegating implement-a-PR work to a subagent, name this check explicitly in the brief ("ensure `codecov/patch` passes, not just the test suite") — a subagent that only runs the local test suite and checks it's green has no way to know it also needs to check a service-side status check unless told.

2. **The latest review is totally clean:** no nits, and every item that wasn't directly **Addressed** is either **Deferred** to a tracked follow-up issue, or **Rebutted with a rebuttal that actually convinced the reviewer** — i.e. the reviewer did *not* re-raise it on the next round. A rebuttal the reviewer still disputes does **not** count as clean.

A clean CI run and a clean review verdict are a snapshot, not a standing guarantee of mergeability. `main` can advance after your last check — including gaining its own independent addition that collides with yours (see `sync-with-main.md`'s "two PRs append the same numbered subsection" case) — so re-verify the branch still merges cleanly against current `main` before reporting a PR ready, not just trust the last green run.

Threads: at fully-clean, every **inline** review thread is resolved, and the only conversation left open is the final all-clear exchange — the reviewer's all-clear comment and your reply to it. (The all-clear is usually a top-level PR comment, not an inline thread.)

Deadlock -> escalate to a human. If you and the reviewer(s) can't reach consensus on an item (a rebuttal was exchanged and neither side is budging), don't loop forever and don't unilaterally override the reviewer — request a **human reviewer**, `@`-mention them in a comment summarizing the impasse, and surface the open item.

An automated reviewer's verdict on a disputed factual/technical claim is not stable across independent runs, even with identical evidence available each time. Don't treat one round's "settled, no need to keep arguing" as durable: the very same review job, re-triggered later with no new code changes, can re-raise a claim it previously retracted — and then retract it again on a subsequent run — purely from re-deriving the question differently each time, not from anything changing in the PR. This means a rebuttal thread's outcome (however many rounds of citations and counter-citations) doesn't itself resolve a genuine deadlock the way a human's decision does; only escalating per the bullet above actually settles it. The one thing that DOES help going forward: fold the authoritative citation/evidence directly into the code or doc being reviewed (a comment, not just a PR conversation reply) — a fresh reviewer run re-deriving the claim from scratch is more likely to find the citation sitting right next to what it's evaluating than to dig through prior thread history for it, though even that is not a guarantee against a bot that ignores context already in front of it. (Sparta#852, 2026-07-14: the same `@claude` review job's independent runs on this PR gave three different verdicts on the identical `gitglossary(7)`-backed pathspec claim across three re-triggers with no intervening code change to the claim itself — "settled, accurate" -> "backwards, needs more work" -> "accurate after all, retracting my own prior finding" — resolved only once the human merged it directly rather than by winning the argument with the bot.)

A review job's pass/fail conclusion can diverge from whether a genuine clean verdict was actually posted — check both directions, not just the check's color. The familiar direction: a green review job that posted only a stub with no verdict (a stalled/crashed review run) is NOT a clean verdict — re-trigger and read the actual comment before trusting green. The inverse, easy to miss: a review job reporting FAILURE can still have posted a complete, genuine "Ready for merge" verdict with real findings-review content — some guard

scripts that gate the job's own pass/fail on detecting a verdict string can misfire and report failure even though a full review ran and passed. Read the posted comment body, not just the check conclusion, before concluding a PR is or isn't clean. If the check is a **required** check and you've independently confirmed the posted content is genuinely clean, that is still not authorization to merge past it yourself — a required check failing is exactly the "stop and ask" case even under a merge-when-confident grant (see `mwc`'s scope note); report the evidence and let the human decide whether to override, fix the guard script, or relax branch protection. (Learned on sparta#590/#594/#598, 2026-07-02: two independent PRs hit the inverse misfire in the same session, and an attempt to merge past the required check on verified-clean content was correctly blocked by the harness's own permission system.)

Address Every In-Scope Review Comment

When iterating on a PR with a reviewer, **address every in-scope flagged item**, regardless of severity label. The reviewer's "Not a blocker", "minor", "nit", "optional", "consider", or "if you want" labels are for prioritization, not a free pass for the implementer.

For each flagged item, do exactly one of:

1. **Fix it in this PR.** The default path — most nits are 1–3 line changes.
2. **Defer to a tracked issue.** Only when the fix expands the PR's scope (new feature, broader refactor, separate concern) or the requester has explicitly said this PR shouldn't grow. File a follow-up issue and reference it in a PR comment so the item isn't lost.

Then trigger another review and repeat until the PR is **fully clean** — zero flagged items under any heading, no "non-blocking", "harmless", "minor observation", or "could improve" sections. "Looks good" / "no findings" / "approved" with no follow-on bullets is the bar. Resolve every inline review thread along the way, leaving only the final all-clear exchange.

Do **not** report "ready to merge with one minor nit noted" / "harmless as-is" / "can address if you want" — that hedging just pushes triage back to the requester. If after 3–4 rounds the reviewer keeps generating new nits each cycle (asymptotic noise), surface that and ask whether to keep going or accept the current state.

Noise is per-item, not per-round — don't stop the whole loop over one recurring flag. A long-running PR can have both real findings (worth fixing every round) and one specific item the reviewer re-raises verbatim round after round even though it's already deferred/tracked (e.g. a file-length guideline already split into a follow-up issue). Keep fixing every *new* finding as it appears — don't let the recurring item make you stop processing genuinely new ones. But stop re-litigating *that one item* every round: reply once pointing at the tracked issue, and hold on it specifically rather than re-deferring it on each pass. Surface the pattern to the user (which item, how many rounds, where it's tracked) and let them decide whether to resolve it now (e.g. do the split) or leave it as accepted recurring noise — don't decide unilaterally to

either keep re-processing it or silently drop it. (rme#706 ran 100+ review rounds: each round's *new* findings — a missing derivation step, a missing i.i.d. hypothesis, an unverified citation locator — got fixed every time; the one recurring file-length flag got a single reply-and-hold each round until the user weighed in.)

When a finding is a pattern (a formatting/style rule broken in one spot), apply it everywhere it recurs in the same file, not just the flagged line. A reviewer that flags one inconsistent list-item format is telling you about the rule, not just that one item — fix every occurrence in the same file that breaks it in the same pass, rather than waiting for the reviewer to flag each occurrence in a separate round. Re-scan the whole changed file for the same pattern before pushing the fix.

When a prose fix changes wording that's also paraphrased elsewhere in the same PR (a CHANGELOG entry, a PR description, a cross-reference), sync that copy too. A CHANGELOG entry written before the review lands often quotes or paraphrases the exact phrase a reviewer later flags; fixing the source prose but leaving the paraphrase stale reintroduces the same wording issue one file over. Grep the diff for the flagged phrase before considering the finding closed. (ai-config#373: fixed "routing/dispatch site" in the skill per review, but the CHANGELOG entry still said it until a follow-up commit.)

This generalizes to a skill's own inline restatement of a fragment it links to. A `SKILL.md` that links a backing `shared/` fragment for the full detail often *also* restates the fragment's approach or word list inline (in its `description` field, or a short procedure-step summary) so a reader doesn't have to open the linked file. Fixing a bug in the fragment doesn't automatically fix these inline restatements — they're a second, independent copy of the same claim, and a review round after the fragment fix can catch them going stale exactly like a CHANGELOG paraphrase does. Grep the whole PR diff for the fixed phrase/word-list, not just the fragment file, before considering a fragment fix complete. (`ai-config#507`: fixing `forward-references.md`'s regex left `fix-forward-references/SKILL.md`'s own `description` field and Step 2 summary describing the old, already-fixed approach — caught in a second review round.)

References