Pull-Request Workflow with Agents

Last modified: 2026-08-03 15:41:00 (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/github-actions.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.

Verify a mid-task "already done" claim against real PR state before trusting or redoing it. A PR you claimed and are actively driving can still gain commits from a **second, independently-running session** under the same account — a `<github-webhook-activity>` review-comment-reply event can describe work ("Addressed… Pushed in `<sha>`") that this session never did. Don't assume it's fabricated or injected, and don't reflexively redo the same fix: cross-check the PR's actual commit list (`gh pr view --json commits` / `pull_request_read get_commits`) and review threads before either (a) trusting the claim, or (b) starting the same fix yourself. If a commit with that SHA genuinely exists, authored close to when the event arrived, treat it as confirmation a live parallel session owns this PR right now — stop pushing further speculative fixes yourself, and, if genuinely in doubt, ask whether to keep driving or step back, rather than racing the other session's pushes. This gap is distinct from the initial claim check above: it's not about claiming a PR before starting, but about **re-verifying you're still the sole active driver** once work has been under way for a while — especially when you picked up the PR mid-session (e.g. by answering a diagnostic question about it) rather than through the normal claim-then-branch flow, so no fresh "paws off" check ever ran right before you started pushing. (`d-morrison/gha#286`, 2026-07-24: a webhook event delivered a review-comment reply attributed to `d-morrison` reading exactly like a Claude-authored reply, claiming a fix "Addressed… Pushed in 3fb8c5b" that this session hadn't made; verified real via `get_commits` before proceeding — a second live session, not injection.)

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 fetch origin main
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.)

The same staleness trap has a silent variant with no CI failure to flag it: a worktree/branch named after a PR's followup can still be based on a `main` from before that PR actually merged. A worktree directory or branch name suggesting "after PR #N" (e.g. `pr-N-followup-...`) is not proof the branch's actual base commit postdates

#N's merge — it can have been created earlier and simply named for its intended purpose. Trusting that naming, then reasoning from `git show <hash>` for a commit found via `git log --all` (which lists every reachable commit across all refs, not just your branch's ancestry) can make content look present when it isn't actually in your branch yet. Verify with `git log --oneline HEAD..origin/main` or by reading the actual blob your branch would produce (`git show HEAD:<path>`, or the working tree itself before assuming what it contains), not a commit hash pulled from `--all`. If `main` has moved, merge it in before building further edits on the assumption the missing content exists. (`ai-config#637`: a worktree named `pr-636-followup-...` was cut from a `main` snapshot that predated #636's own merge; an edit referencing "the bullet above" – added by #636 – was written and committed before the bullet actually existed on the branch, caught only when `git push` reported `main has moved` and the subsequent merge produced a real conflict.)

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.)

When `main` DELETES a file your branch references, resolving the marked conflict is not enough — grep the whole tree for the deleted path. Git only conflicts where both sides edited the same lines, so a merge that brings in a deletion flags the file that *used* the thing, and nothing else. Any other reference to the deleted path — a docstring citing it as precedent, a comment, a doc cross-reference — merges cleanly and silently becomes a dangling reference, because those files were never in the conflict's scope. After resolving any merge that removed a file, run `grep -rn "<deleted-path>"` across the repo and re-point or reword each hit; then distinguish live references (must be fixed) from historical citations of the removal itself (correct as-is, leave them). This is the deletion counterpart to the extracted-copy case above: there the logic moved and a copy went stale, here it vanished and the pointers went dead. (ai-config#696: `main` retired `scripts/check-new-line-breaks.py` via #703 while the PR was open. The `validate.yml` conflict was visible and resolved, but `scripts/check-memory-file-size.py`'s docstring cited the deleted script as its advisory-exit-code precedent — a file the conflict never touched, caught only by grepping for the path afterward.)

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.)

The same collision can land before you write a line, and then it produces no conflict at all — just duplicated work nobody flags. The bullet above catches a duplicate at merge time, via a conflict. When `main` gains the colliding content while your change is still *planned* rather than written, there is nothing to conflict with: you write the duplicate, push it, and the review has to argue you out of content that was already redundant on arrival. So re-run the dupe check after any fetch that brings in new commits, not only at merge — a plan researched an hour ago was researched against a different `main`.

The cheap version is to read what actually arrived rather than only the count: `git log --oneline <old>..origin/main` plus `git diff --stat` over the same range, then ask whether any of it covers something still on your list. In a session that loads skills or plugins from the repo, a new one appearing in the session's own skill listing is the same signal arriving for free.

This is a *timing* gap, and it composes with the *scope* gap rather than replacing it. `check-open-prs-before-duplicating` covers work that is still in flight, unmerged, and therefore invisible to any check against `main`; run that one too, since a duplicate is just as wasted whether the collision has landed yet or not. Both checks share the same weakness — each runs once, at the start, and answers for the moment it ran.

Dropping the planned work is the cheap outcome, so record why in the issue and the PR body rather than deleting it silently — otherwise the next person re-proposes it. (ai-config#774, 2026-07-28: a planned `profile-before-optimising` fragment was mooted by `skills/measure-performance`, which merged via #762 during the session and covered the same two chapters — including the specific gap the fragment was meant to fill. It surfaced only because the new skill appeared in the session's skill list after a routine fast-forward; the plan had been written before it existed. Dropped before implementation, with the reasoning recorded in both the issue and the PR body, and the neighbouring fragments cross-linked to the skill instead.)

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.)

The same splice happens to LIST ITEMS, and there `markdownlint` most likely does NOT catch it — so nothing turns red at all. The case above is a heading spliced against preceding text, which MD022 decides. The changelog case is a *bullet* spliced onto the previous item's continuation line:

```
  `data-raw/precompute-true-effects-chunk.R` (#429).
* The `docs` workflow's "Build site" step no longer times out intermittently.
```

That is a valid **tight** list item, so a list in which every other entry is blank-line separated silently starts mixing tight and loose items and renders inconsistently. `markdownlint`'s blanks-around-lists rule governs the boundaries *of* a list, not the gaps *between* its items, and no default rule enforces consistent looseness within one list — so unlike the heading case, CI stays green and only a human reading the merged section notices.

Check it mechanically instead of by eye; one line decides it:

```
awk 'prev !~ /^[[:space:]]*$/ && /^[*+\-] / {print FILENAME":"NR": "$0} {prev=$0}' NEWS.md
```

Use `[[:space:]]*` rather than a bare `/^$/` — a whitespace-only preceding line is not a violation and produces false positives. The pattern `[*+\-]` covers all three common Markdown unordered-list markers; `^\*` alone would miss `-` and `+` bullets.

Two consequences. Run this after any merge into a changelog or other growing bulleted list, alongside the heading check above. And note that a `merge=union` driver on such a file (see `configure-gitattributes`) *increases* the rate of this defect, since union resolves an append collision by keeping both sides with no conflict to review — so confirm a detector is wired into CI before enabling one, not after. (ucdavis/bcs#422/#430, 2026-07-26: a clean three-way merge spliced one PR's `## Bug fixes` bullet against another's; the check above then found **four** pre-existing instances in the same `NEWS.md`, in a repo that had no Markdown linting at all.)

A commit claiming "I've pulled main and resolved the merge conflicts" can be lying — verify it actually merged before trusting the claim. A genuine conflict-resolution commit is a merge commit (two parents); a commit that just hand-edits files to *look* resolved, without running a real `git merge`, is an ordinary single-parent commit — and it never actually incorporates whatever new state of `main` prompted the "resolve conflicts" request in the first place. This is easy to miss because GitHub's own `mergeable`/`mergeStateStatus` fields don't distinguish the two: both look identical from the PR page until you check the commit graph. Verify with `git show -s --format="%P" <commit>` — one hash means no real merge happened, regardless of what the commit message says. If a branch needed conflict resolution more than once and each attempt claimed success but the PR still shows `CONFLICTING`, check every "resolved conflicts" commit in its history this way before trying yet another resolution attempt on top of a foundation that was never actually re-merged. (`Lacaedemon/sparta#1070`, 2026-07-27: an automated PR-authoring agent pushed two consecutive commits both titled "Resolve merge conflicts", each claiming to have pulled `main`; both were single-parent commits that never touched `main`'s actual current state, so the PR kept showing `CONFLICTING` no matter how many times the agent "fixed" it. A real `git merge origin/main` — the first one actually run against this branch in three attempts — surfaced the genuine conflicts and resolved them for good.)

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.

Because the loop ends there, **the clean verdict is also where `ums` runs** — don't hold the pass for the merge, which is on the human's clock rather than this session's and may land after a `/clear` or not at all. See `CLAUDE.md`'s "Run UMS proactively, as learnings accumulate"; the merge-time pass in `post-merge` then only has to cover what the merge itself taught.

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.)

A fix is not "pushed" until it is on the PR's head commit — verify with a SHA comparison before telling a reviewer you pushed it. From inside a session, an edited working tree and a pushed commit feel identical, so a round that edits the files, writes the reply, and never runs `git push` produces a reply asserting a fix that does not exist on the branch. Nothing contradicts it: CI reports green, because it correctly validated the older head; the next review round reviews code without the fix; and the session's own recollection of having made the change agrees with the reply. That makes it worse than an ordinary wrong claim — it is a false statement about *state*, which a reviewer has no reason to doubt and no cheap way to check. Before posting any reply that asserts a push, compare `git rev-parse HEAD` against the PR's own `head.sha` (`pull_request_read get`); if they differ, push first, then reply naming the real SHA. Run the same comparison in every periodic check-in on a PR you are babysitting, since the failure is silent and survives each round until something explicitly looks for it. This is the `algorithmatize-checks` rule applied to your own claims: two SHAs decide it exactly, so never substitute recollection. (d-morrison/altdoc#54, 2026-07-25: two review fixes were edited locally and a PR comment said they were "addressed in the latest push"; the head sat at the pre-fix commit for over an hour, with 14 green checks validating a branch carrying neither fix, until a scheduled check-in compared the SHAs.)

When the change affects downstream consumers, validate it against a real consumer repo before reporting the PR ready — a package's own test fixtures are built to exercise its code, not to resemble the packages that will actually use it. Fixtures are minimal by construction and tend to share one shape, so whole branches of new code can be structurally unreachable from them. A real consumer brings the input variety fixtures lack, and it is usually one clone plus one command to check.

Three classes of gap this catches, none of them findable in a fixture:

- **Input shapes no fixture happens to contain.** A real package carries metadata the fixtures never needed — an entry of a different kind, an extra tag, an unusual name — so a branch written for it has never actually run on real input.
- **Message formatting under real counts.** Fixtures usually trip the plural path; a real repo hitting the same code with exactly one item exercises the singular wording, which no test asserted.
- **The migration/upgrade path, as opposed to the fresh-install path.** This is the one fixtures can never reach: a fixture is created new by the test, so it always gets the current templates. An existing consumer has the *old* config, and whether the feature reaches it at all is a different question from whether it works. Verify the claim in the changelog by running the documented migration step, rather than describing it.

Do it against a throwaway copy and push nothing to the consumer; the deliverable is evidence in the PR, not a change there. Record what the run covered in a PR comment, so a reviewer can see which paths real input reached. (d-morrison/altdoc#34: running the new reference-index generator against `d-morrison/rpt` covered a `\docType{package}` topic, the singular form of a missing-topic warning, and the documented "existing settings files do not pick this up automatically" caveat — confirmed by the page generating while `grep -c reference.html docs/index.html` returned `0`. None of the three were reachable from the repo's own fixture packages.)

Verify a blocker you assert in a PR body or a reply, with the same rigor you apply to a reviewer's claims — a stated blocker becomes a premise other people build on. The reviewer-facing checks above all point outward: verify the suggestion, verify the literal, verify the push landed. The inward case is easier to miss, because a limit you hit yourself feels like an observation rather than a claim. It is still a claim, and writing it into a PR body publishes it as settled fact: a reviewer reading "the pinned tool is unavailable in this sandbox" will reason from it, recommend a follow-up around it, and never re-test it, so one unverified sentence quietly redirects the review. Before asserting that something is unavailable, blocked, or impossible here, actually attempt it once — an install, a fetch, a single command — and say what you tried. A negative result from one incidental symptom (a failed version query, a single 403) is evidence the thing is not *already set up*, not evidence it cannot be. When a blocker you published turns out to be false, correct it where it was published, not only in the thread that surfaced it. (d-morrison/altdoc#76, 2026-07-27: the PR body said roxygen2 8.0.0 — the version `DESCRIPTION` pins — was unavailable, inferred from one failed `packageVersion()` call with no install attempted. The review built a "this may need a follow-up" recommendation on top of it. A single `install.packages()` disproved it, and the regeneration landed in the same round the finding did.)

A blocker that was true when you published it can stop being true while the PR is open, and withdrawing it is your job, not the reviewer's. The bullet above covers a blocker that was never true. This is the harder case, because the caveat was correct and

diligent when written, so nothing about it reads as a defect later — and a sentence saying "this could not be checked" is one nobody re-checks, least of all the reviewer, who has no way to know the environment moved. It keeps steering the review regardless: a verdict can repeat the caveat back as an accepted limitation, which makes the stale claim look corroborated. So when the cause of a blocker changes — a host unblocked, a tool installed, a quota reset, a dependency published — re-run the check and withdraw the caveat where it was published, saying explicitly that it is withdrawn rather than quietly deleting the sentence. A reader who saw the original needs to know it was retested, not be left wondering whether it was ever true. (ai-config#774, 2026-07-28: the PR body said four `adv-r.hadley.nz` anchors could not be verified because the host was egress-blocked, which was accurate when written. The host was unblocked mid-session, and all 16 URLs then verified 200 with every anchor resolving. The review had already absorbed the caveat — it listed those anchors as "unverified per the PR body's own caveat … not a new finding" — so leaving it would have shipped a limitation that no longer existed, blessed by a reviewer who could not have known.)

An instruction's own suggested code is not exempt from the project-conventions self-review above. The self-review rule assumes you wrote the diff; a snippet handed to you in an issue, a task description, or a design doc slips past it, because adopting someone else's suggestion does not feel like authoring. It is authoring — once pushed, it is your diff, and the project's conventions bind it exactly as they bind anything you wrote yourself. Run the same convention check over borrowed code before pushing it, especially when the suggestion is a plausible-looking one-liner and the convention it breaks is documented rather than linted. (d-morrison/altdoc#73: the issue proposed ending a function with a bare trailing `hashes`, which reads as a fix for the fragility it names but is still an implicit return, so a statement added after it silently becomes the return value. The lab manual asks for an explicit `return()` regardless. Review caught it; the project's own stated convention would have, one step earlier.)

When the code path under test has a staging or transform step between input and output, a passing unit suite is not evidence it works — exercise the real path once. Fixtures instantiate the shape the test author had in mind, so a wrong assumption about *where* the code runs is invisible to every one of them: the tests and the bug share the assumption. This is the same gap the downstream-consumer rule above covers, one level in — there the missing variety is the consumer's input, here it is the pipeline's own directory layout, timing, or intermediate representation. One real invocation is usually cheap, and it tests the assumption the fixtures encode rather than re-confirming it. (d-morrison/altdoc#76: a guard checked for the copied logo under `docs/`, but the `quarto_website` path stages into `_quarto/` first, so the logo line was dropped on every render of the one generator the feature wired up. Seventeen unit assertions passed throughout; one throwaway render found it immediately.)

When new code branches on a third-party tool's behavior, read that tool's own config or docs for the specific behavior — don't infer it from what the tool broadly does. The bullet above covers your own pipeline's layout; this one covers the tools that pipeline drives. An inference of the form "it builds HTML, so link to `.html`" is exactly the shape that feels too obvious to check, and a tool's defaults routinely contradict it. Two properties make this

worse than an ordinary wrong guess. The inference usually lands in a branch your own fixtures cannot reach — you have no fixture for someone else's renderer — so the test suite agrees with you. And it produces output that is well-formed and plausible (a link, a path, a flag), so a reviewer skimming the diff has nothing to catch, and the failure surfaces only in a consumer's published site. Name the setting you are relying on, and check its actual default before writing the branch. (d-morrison/altdoc#78, 2026-07-27: a generator-to-extension map gave mkdocs `.html`, reasoning that mkdocs compiles Markdown to HTML. Its `use_directory_urls` default is `TRUE`, so it serves `/man/foo/` and never `/man/foo.html` — every reference link the feature emitted for that generator would have 404'd. Caught in review, not by the 39 tests.)

A regression test written alongside a fix can lock the bug in rather than catch it — assert the two paths that diverge, not the one you just touched. A test authored in the same pass as the code tends to record what the code *does*, because you run it, see it pass, and move on. That is usually harmless. It becomes a lock when the fixture is thin enough that the buggy and the correct path produce the *same* output: the assertion then encodes the degraded result as intent, and every later reviewer reads a green suite as evidence the behavior was chosen. The next round's finding lands on your test, not just your code.

The tell is the same each time: **a fixture missing the input variety that makes the two paths differ.** So when a bug is an asymmetry — nested versus top-level, second render versus first, one generator versus another — build the fixture so both sides are present and assert them together. Either side alone is unfalsifiable, since the case that reveals the bug is the *comparison*. Then prove it: revert the fix and confirm the new test actually fails. A regression test never seen to fail is a guess about what it covers. (d-morrison/altdoc#78, 2026-07-27: twice. A `.pdf` vignette test asserted the entry's extension but never its label, so an extension leaking into the label passed; and a nested-article test built no source tree, so top-level and nested resolved identically and a nested-only title bug was pinned as expected output. Both were found by review reading the test, not the code.)

A systematic audit done by skimming is worse than the one-at-a-time version it replaces. Batching a check — "rather than wait for the next round to find divergence number four, compare all four at once" — is the right instinct, and it inverts if each lookup gets less care than it would have alone. Two things make the batched form more dangerous, not less. Its output is usually a claim recorded somewhere durable (a comment, a doc, a table), so an error is published rather than merely held; and it arrives labelled *audited*, which is precisely the word that stops the next reader from checking. A wrong comment in a block written to prevent a specific future change invites that change while appearing to forbid it. Concretely: when the thing being audited is a function, grep for the function, not for a pattern in its file — a file with several functions will hand you the first match, which is often not the one you mean. Name the function in whatever you write down, so the claim stays checkable. (d-morrison/altdoc#78, 2026-07-27: a commit written to get ahead of a one-finding-per-round loop claimed mkdocs' sidebar matched only `\.md`. It matches `\.md$|\.pdf$`; the grep had returned a different function 120 lines above the sidebar builder in the same file. Caught by the very next review round.)

Adding an explanation supersedes whatever the file already said about the same thing, so re-read the older passage — your own diff is the likeliest source of a contradiction nobody flags. The sync rules in `address-every-comment` all fire on an external trigger: a reviewer quotes a phrase, or behavior changes and a changelog goes stale. This one has no trigger at all. You add a paragraph explaining that something was misunderstood, and the note recording the original misunderstanding sits a few lines below, still stating it as fact. Nothing conflicts, no check fires, and both passages read plausibly on their own — but a reader who reaches the older one first comes away with exactly the belief the new text was written to remove.

The tell is a diff that adds an explanation, a correction, or a "what this actually means" paragraph near existing prose. Re-read the surrounding passage as a whole rather than diffing your addition in isolation, and treat a historical record ("we observed N of X") as a claim your explanation may have just falsified. When the older passage recorded a *different* session's observation, correct it with reasoning that stands on its own rather than restating it as though you had seen it — an inference presented as an observation is the same defect one level up. (ai-config#770, 2026-07-28: an added explanation established that seven reported orphans were misclassifications, while the note two lines below went on calling them "already deleted from the repo." Caught by review, in the same hunk as the text that contradicted it.)

And when the explanation you add is a *mechanism* claim, test the class it distinguishes, not just the sample in front of you. The bullet above is about contradicting old text; this is about the new text being unfalsifiable on the evidence you gathered. A classifier validated on a population containing no positive instance of the class it is supposed to catch will report a clean result either way, so "it returned zero" is not evidence it works — it is the same missing-input-variety tell the regression-test bullet above describes, moved from a fixture to a diagnostic. Ask what a true positive would look like, confirm one exists in what you tested, and if none does, say so instead of claiming the mechanism separates the cases. (ai-config#770, same day: a `git log -- skills/<name>` probe was said to separate "deleted from the repo" from "never ours" *exactly*, on the evidence that it reported zero false orphans. The repo contained no deleted-but-still-installed skill at all, so there was nothing for it to get wrong; and `git rev-parse --is-shallow-repository` returned `true`, meaning anything deleted before the shallow boundary would have been silently misread as harness-provided. The claim went into a PR reply before either check was run, and ai-config#765 had independently reached the correct conclusion.)

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 and check runs are green AND completed.** Every workflow and check run passes — not just the required checks and not just the review job. "Green"

means finished with a passing outcome (success or skipped), not merely "currently reporting green while still running" — never treat a workflow or check run that's still queued or in progress as clean, even if nothing has failed yet. **A reviewer's posted verdict does not mean the review check has finished, so don't let a clean verdict stand in for criterion 1 on its own job.** The bot posts its comment and then its run keeps going (bookkeeping steps, a cost tally, the gate job that consumes its result), so a full `Ready for merge` comment can sit on the PR for minutes while `claude-review` still reads `in_progress` and the `require-review` gate is still `queued`. Reading the verdict and moving straight to "clean" skips the very state this criterion exists to catch. The gap runs the other way from the stub-review case below: there the check is green and the verdict is missing, here the verdict is real and the check is unfinished. Re-read the check runs after the verdict lands, not just before. (ai-config#712, 2026-07-24: the round-2 verdict posted at `04:06`, about two minutes before its own `claude-review` job completed at `04:06:56` and `require-review` at `04:07:03`.) (The exact field names and casing for these states differ by API surface — REST's check-runs endpoint returns lowercase `status`/`conclusion` strings like `completed`/`success`, while `gh pr checks`/GraphQL's rollup returns uppercase `state` values like `SUCCESS`; don't hard-code one casing when scripting a check.) **A raw Actions workflow run and a check run are not the same thing, and the usual lookups (`gh pr checks`, `get_check_runs`) only cover check runs (plus legacy commit statuses) — not every workflow run necessarily produces one.** A workflow run that's blocked on `action_required` (e.g. pending manual approval) before any job starts can complete with zero jobs and consequently zero check runs, making it invisible to a check-runs-only poll. This normally doesn't affect mergeability (GitHub's branch-protection required-checks gate operates on checks, not raw workflow runs, so a check-run-less run can't be wired as required), but if something about a PR's CI state looks off despite `gh pr checks` reporting all-clear, cross-check the raw workflow runs before trusting the checks-only view. **`gh run list --commit <head-sha>` is not a reliable substitute for this cross-check on its own**: it returns every attempt for that SHA (including superseded/cancelled re-runs, so an old failed attempt can look like an outstanding blocker), and a run triggered by `issue_comment` or a `workflow_dispatch` invoked without an explicit `ref` can be recorded against the default branch's SHA rather than the PR's head SHA and be missed by a `--commit` filter entirely. Neither `--commit` nor `--branch` is fully reliable for this, because GitHub itself does not record a reliable PR linkage for these trigger types: an `issue_comment`-triggered run on this very PR (#635, run 29967418653) recorded `head_branch: main` and an empty `pull_requests` array via the raw REST API (`GET /repos/{owner}/{repo}/actions/runs/{id}`) — verified directly, not assumed — so no single filter (commit, branch, or the API's own PR-linkage field) reliably narrows these runs to the ones for this PR. Treat this cross-check as best-effort: `gh run list -R <repo> --workflow <name>` (unfiltered, or windowed by approximate timestamp) and eyeball for anomalies near when the PR activity happened, rather than trusting any one filtered command to be exhaustive. 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. That review must be a genuine posted verdict at the current head commit, from an external reviewer if one is reachable — self-review is a fallback for when no working external reviewer is available, never a substitute once one is (see the `ardi` skill's step 2 for the availability-recheck procedure). Re-check availability right before declaring clean, not just at whichever round self-review first started; an inferred "probably clean" from green CI and resolved threads does not satisfy this.

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.

Re-check version parity in that same sweep, not only conflict-freedom. `sync-with-main` already covers comparing `DESCRIPTION` versions *after merging `main` in*. The case that rule misses is the one with no merge at all: `main` advances on its own after your last review round and lands on the branch's exact version, so an R package's `version-check` job (which requires the branch to *exceed* `main`) goes from green to red with nothing to point at. There is no conflict, no failing check yet, and no warning — the last run passed because `main` was still a version behind when it ran. So the declare-ready sweep needs both `git merge-tree` for conflicts and a direct version comparison; either one alone reports a PR ready that isn't. (`UCD-SERG/serocalculator#392`, 2026-07-25: the final pre-declaration check found `main` had reached `1.4.1.9016`, exactly the branch's version, minutes after a clean `Ready for merge` verdict on an otherwise all-green head.)

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.) Check this mechanically rather than from a memory of which threads you replied to. Which field name to look for depends on

the surface: the GitHub MCP tool `pull_request_read get_review_comments` returns thread objects under a `review_threads` key with snake_case `is_resolved/is_outdated`, while a raw `gh api graphql reviewThreads` query — what `resolve-pr-threads`, `pr-status`, and `ard` use — returns camelCase `isResolved/isOutdated`. Both are correct on their own surface; this is the same REST-vs-GraphQL casing split the check-state paragraph above already warns about, so read the response you actually get rather than assuming one spelling. Either way, sweeping for the unresolved ones is the entire check. An **outdated** thread (`is_outdated: true` — the code it anchored to has since changed) still counts as unresolved: addressing a finding and resolving its thread are separate actions, and only the second clears this criterion. An addressed-but-unresolved thread reads as outstanding work to every later reviewer, which is exactly what this criterion exists to prevent.

One finding can own two threads, so sweep by thread id rather than by finding. When a reviewer re-raises an item you already answered, the re-raise often opens a **new** thread instead of continuing the original — same file, same line, same finding, different `threadId`. Resolving the one you remember replying to therefore leaves a second thread behind, and it is easy to miss twice over: it is usually marked `is_outdated: true` (the line it anchored to has since changed), and your own memory of the exchange says the item was settled. Neither of those clears it. Re-read the thread list before declaring clean and resolve every entry whose `is_resolved` is false, whatever you recall about the finding it carries; reply on the second thread too, pointing at the first, so a reader landing on either one sees the resolution. (`d-morrison/altdoc#61`, 2026-07-25: the round-4 re-raise of an unused fixture parameter opened `PRRT_...TyfeQ` alongside the original `PRRT_...TyeRc`; resolving the original left the re-raise outstanding, caught only by a mechanical sweep of all seven threads.)

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.)

A third case, distinct from either misfire above: some checks are designed to NEVER fail regardless of their own posted content, so their green color carries zero signal at all. A CI-runner-relative benchmark check that gates a soft threshold (e.g. "regressed beyond 20% vs. baseline") may deliberately report success/pass at the GitHub-check level even when it posts a `:warning:` regression comment, precisely because the project has decided that threshold is "a human call, not an auto-block" rather than a hard gate. `gh pr checks` (or the equivalent status API) showing this check as PASS is consequently not evidence there is nothing to look at — it only means the check ran, not that its content was clean. Read the check's own posted comment body every time, the same discipline the review-job case above already demands, but don't expect the check's pass/fail conclusion to ever flip for this class of check even on a real, large regression. (Sparta#995/#998/#999, 2026-07-19: `gh pr checks` reported `benchmark` as PASS across three separate PRs while the actual posted comment showed regressions of 45%, 38.8%, and 36.9% respectively against the CI-runner baseline — two were real, fixable redundant-computation bugs; the third traced to a stale baseline that predated an earlier PR's own accepted cost increase and hadn't been refreshed yet, since the refresh workflow only runs on a weekly schedule, not on every main push.)

A fourth case: a review job can post a syntactically valid, confidently stated verdict that is nonetheless invalid because it rests on a hallucinated premise about the PR's own state — not a stub (no verdict) and not a misfire (guard-script/check-conclusion mismatch), but a fabricated fact baked into an otherwise well-formed review. A reviewer that infers PR state from a commit message rather than querying the

PR's actual `state`/`merged` API fields can mistake a routine `Merge remote-tracking branch 'origin/main' into <PR-branch>` commit — pushed to resolve a sync conflict on the still-open PR branch itself — for evidence the *PR* was merged into `main`, and confidently report "PR is closed, no action taken" while never actually reviewing the diff. This reads exactly like a legitimate all-clear (a `### Verdict` section is present, the job reports success), so the stub-detection guards described in CLAUDE.md's "Do the review yourself when the (**claude?**) workflow doesn't produce a verdict" section don't catch it. Sanity-check any surprising verdict — especially "nothing to review" or "already merged/closed" — against the PR's real API state before trusting it, and re-trigger for a genuine review rather than accepting a verdict-shaped comment built on a false premise. (gha#293/gha#295, 2026-07-24: after a merge-conflict-resolution push, the re-triggered `claude-code-review` run reported "The PR is closed — it was merged as commit `db11634`" even though the PR was still open and `db11634` was only the PR branch's own merge-with-main commit; re-triggering once more produced a genuine review of the actual diff.)

A fifth case, and the one that decides what "reachable" means in criterion 2 above: an external reviewer can decline to review at all, posting a refusal in the shape of a review. Unlike the four cases above — all of which are a review that ran and produced something misleading — this is a reviewer that never ran, and says so in a `COMMENTED` review whose whole body is the refusal (e.g. Copilot's *"unable to review this pull request because the user who requested the review has reached their quota limit"*). Three consequences for driving a PR to fully clean:

- **A refusing reviewer is not "reachable,"** so criterion 2's external-verdict requirement falls to whichever external reviewer *is* working. Don't stall a PR waiting for a reviewer that is refusing — but don't quietly downgrade to self-review either while another external reviewer is answering normally.
- **Reviewers fail independently.** One can be quota-dead while another reviews the same head normally, so check each one rather than generalizing from the first refusal.
- **Keep re-requesting each round anyway.** A quota resets on its own schedule, so a reviewer that refused a few pushes ago can come back mid-session — which is exactly what criterion 2's "re-check availability right before declaring clean" is for. Say so explicitly when reporting a PR ready: name which reviewer's verdict the clean call rests on, and which one never weighed in at this head.

The mechanics of detecting a refusal (it arrives as a posted review, not an API error, so the request call's success proves nothing) are in `memories/github.md`'s GitHub MCP tools section. (`ucdavis/rampp#111`, 2026-07-24/25: Copilot refused three times across two heads for quota while `claude-review` posted genuine verdicts at both; the PR was reported clean — and merged — on `claude-review`'s verdict, with Copilot's absence stated in the ready-for-merge comment rather than papered over.)

A sixth case runs the other way from all five above: the review is genuine and complete, but the workflow posts the reviewer's own tool invocation instead of the re-

view body. The comment opens with a literal `gh pr comment <N> --repo <owner>/<repo> --body "$(cat <<'EOF'` and closes with `EOF\n)"`, wrapping a real, correct verdict as unrendered text — the model emitted a shell command as its final response and the workflow posted that string verbatim. Nothing is lost, and the same body usually also lands as a properly-rendered sibling comment, so the PR carries the review twice. Two reasons not to shrug at it: a comment opening with a raw `gh` invocation reads as a broken run, so a human is likely to discount a review that actually passed; and a verdict-detecting guard script (`check-review-execution.sh`) is now matching against a shell command rather than prose, which can misfire into a needless stub-retry and a second full review's cost. Read the body and extract the verdict from inside the heredoc rather than re-triggering. (`UCD-SERG/serocalculator#392`, 2026-07-25; filed as d-morrison/gha#312, which proposes unwrapping the pattern before posting.)

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.** Only when the fix expands the PR's scope (new feature, broader refactor, separate concern), the requester has explicitly said this PR shouldn't grow, or the flagged content isn't actually yours to fix here (see the `main`-sync case below). File a follow-up issue and reference it in a PR comment so the item isn't lost — except in the `main`-sync case, where the "follow-up" is fixing it on `main` directly, not a new issue.

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.

Always resolve an inline thread the moment its comment is successfully addressed — the fix pushed and a reply posted naming it — in the same pass, whatever workflow you're in: a formal `ard`/`ardi` round, a CI-monitor nudge, or a one-off fix outside any loop. Addressing without resolving leaves a thread that reads as outstanding work to every later reviewer, blocks `fully-clean`'s every-inline-thread-resolved criterion, and drags stale noise into the next review round. The per-disposition settlement rules in `ard` step 4b still govern the exceptions: a **Rebut** stays open until the reviewer drops it, and an **Address** you're not confident fully settles the concern gets a reply asking for confirmation instead of a resolve. The `resolve-pr-threads` skill sweeps any stragglers, but it's a backstop — resolve-on-address is the default, not a cleanup step.

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.)

The same sync is needed when the review fix is to CODE BEHAVIOR rather than to wording — and that case is easier to miss, because nothing about fixing a bug points at the changelog. The rule above fires on a recognizable trigger: a reviewer quotes a phrase, so you go looking for that phrase. A behavior finding gives you no phrase to grep. You change the code, update the PR body's description of what it now does, and the `NEWS.md`/`CHANGELOG.md` entry — written before the review, in prose that described the *old* behavior correctly — goes on asserting it. Every later round then reviews a diff whose changelog contradicts its own code, and no reviewer flags it, since each file reads plausibly on its own. The shipped result is worse than a stale paraphrase: a user reading the release notes is told the opposite of what the release does. So after any Address that changes behavior, re-read the PR's changelog entry against the new behavior — not just the code and the PR body. Fold

it into the same pre-push self-review pass `ardi` already requires; a changelog entry is a claim about the diff, so `fact-check-prose` applies to it exactly as it applies to any other prose in the PR. (d-morrison/altdoc#78, 2026-07-27: review round 2 established that mkdocs serves `/man/foo/`, not `/man/foo.html`; the code and the PR body were corrected that round, while `NEWS.md` kept saying links point at `.html` under "`mkdocs` and `quarto_website`" through two further clean review rounds. Caught by a `main`-sync merge conflict that happened to land in that entry — not by any review, and not by any check.)

A flagged item that came in via a `main`-sync merge, not your own diff, is still a Defer — just one where the follow-up is fixing it on `main` directly, not filing a per-PR issue. This is not the ARD skill's "Acknowledge" disposition: `skills/ard/SKILL.md` reserves Acknowledge for praise or a no-ask observation, and explicitly warns against stretching it to dodge a real finding — a redundant config line a reviewer flags is a real finding with an implied fix request, so it needs a real disposition, not a label that means "no change requested." When a reviewer flags something (a redundant config line, a stale pattern) inside a file your branch only touches because you merged `main` in to resolve a conflict, check provenance before fixing it: `git log`/`git blame` the flagged line, or just compare against `origin/main`'s current content. If it's identical to `main`, "fixing" it on your branch alone doesn't fix anything — it just makes your branch disagree with `main` on unrelated content the next person to touch that file will have to reconcile again. Reply agreeing the finding is correct but out of scope for this PR, and leave it for whoever owns that file's actual content to fix on `main` directly — no follow-up issue needed, since the fix target is `main` itself, not this PR's own change. (`UCD-SERG/serocalculator#503`: a review flagged `.Rbuildignore`'s `^\.posit/assistant$` as redundant with the existing `^\.posit$` pattern above it — both lines had landed together in an already-merged `main` commit (#579), picked up via a routine `main`-sync merge, not introduced by #503's own diff. Deferred to `main` instead of fixed on the branch.)

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.)

A bot that re-raises an item as "not addressed" may simply not have seen your reply — check the timestamps before treating it as an impasse. An automated reviewer gathers the PR's comments once, when its run starts. A rebuttal posted after that snapshot is invisible to it, so the next round reports the item as still open and unaddressed even though a substantive reply is sitting in the thread. The tell is a re-raise that repeats

the original finding verbatim and speaks only to whether the *code* changed, without engaging any argument you made. Before escalating, compare your reply's timestamp against the review run's `started_at` (`gh run view <id> --json startedAt`, or the `started_at` field each run carries in `get_check_runs` when `gh` is absent): if the reply landed after the run began, it is a stale re-raise, not a genuine disagreement. Reply once pointing at the earlier rebuttal (link it directly — the next run will see it), and don't count that round toward the rebuttal-didn't-convince-them test in `fully-clean.md`.

The ordering fix is cheap: when a round is Rebut-only, post the rebuttal **before** anything that triggers the next review (a push, an `@claude` mention), so it is in the snapshot the next run reads. When a round mixes Address and Rebut, post the rebuttals first and push the code second, for the same reason. (d-morrison/altdoc#34: a `\pkg{}` rendering rebuttal carrying a `pandoc` run that disproved the finding's implied hazard was posted about a minute before the follow-up review job started; that review reported the item "wasn't addressed in `9398d5d`" and re-posted the identical suggestion.)

A finding can be right while its `suggestion` block is wrong — verify the suggested literal before applying it. A GitHub ```` ```suggestion ```` block is one-click-appliable, which is exactly what makes an unverified one dangerous: the surrounding prose argues for a change you agree with, so the concrete replacement rides in on that agreement without being checked itself. Treat any file path, version, flag, or command inside a suggestion as a claim to verify, not as text to accept — the same standard `fact-check-prose` applies to the diff. Accepting a bad literal is worse than ignoring the finding, because it publishes a specific wrong value under the reviewer's apparent authority. When the suggestion is wrong but its point stands, fix the underlying issue your own way and say in the reply why the suggested form was set aside — silently deviating reads as having missed it. (ai-config#726: a review correctly flagged that a `<path>` placeholder didn't say where a script came from, but suggested `<path-to-gha-checkout>/check-new-line-breaks.py` — one directory level too high, since the composite action's directory and the script inside it share a name. `git ls-files` in the gha checkout settled it in one command. Applying the suggestion verbatim would have documented a nonexistent path in the entry whose whole purpose is getting someone to run that script.)

The same check applies to a fix a reviewer describes in prose rather than in a `suggestion` block, and the sharpest test is the reviewer's own example. A finding that ships a concrete repro case has handed you a test fixture: run the proposed fix against that very case before adopting it. A reviewer reasoning about a fix in the abstract can propose one that is directionally right and still insufficient – it closes the failure mode they named while leaving the case they cited broken – and adopting it verbatim converts their partial diagnosis into your shipped bug, with the review thread reading as though the item were settled. When the proposed fix falls short, prefer eliminating the failure mode outright over layering another patch onto it, and post the evidence (the fix applied to their example, and what it still produces) rather than just asserting it was insufficient. (gha#318, 2026-07-26: a review correctly found that a heredoc-terminator regex lacked an end-of-line anchor, and suggested adding one. Tested against the reviewer's own indented-`EOF` example, the suggested anchor

still truncated the body, because the terminator's leading `[ \t]*` accepted a space-indented closing line real bash rejects. Matching whole lines against the tag – how bash itself ends a heredoc – removed the whole lazy-quantifier/anchor failure mode instead of narrowing it; the reply carried the failing output of the suggested form.)

And the mirror case: a finding can be wrong on its stated grounds while still pointing at something real. The two bullets above check the reviewer's *fix*; this one checks their *premise*. A confidently reasoned factual claim – this pattern is valid, that value is in range, this call is safe – invites one of two lazy responses: accept it because it sounds authoritative, or dismiss the whole item once you notice the claim is false. Both lose information, because a reviewer usually arrives at a wrong premise while looking at something that genuinely bothered them.

So reproduce the claim before answering it, and answer the concern separately from the premise. When the premise turns out to be false, say so with the command and its output rather than by assertion, and then address what prompted it anyway – a reader who tested your example and got a different result has a real problem even if their explanation of it was wrong. Expect the corrected mechanism to be more useful than the original text: a premise worth disputing usually sits on something you had not fully explained. (ai-config#756, 2026-07-28: a review held that `[\x{2014}]` is valid PCRE and so could not produce the "code point value too large" error the fragment described, and proposed an out-of-range `[\x{110000}]` instead. Running it showed the original failing exactly as written – the cause is the locale, since PCRE in non-UTF mode rejects any `\x{}` above `0xFF`, and the same command succeeds under `LC_ALL=C.UTF-8`. The proposed replacement would have been worse, failing unconditionally and hiding that environment-dependence, which is the whole reason the swallowed error is dangerous. The reviewer's actual worry – that a reader might not reproduce it – was right, and sharper than stated.)

When a finding cites a source, read the cited source before reproducing anything – it is the cheaper instrument, and it is the one that can show the finding backwards rather than merely unsupported. The bullet above says to reproduce the claim. That is right, and it is the second thing to do when a citation is on the table, because reproduction tests the *behavior* while the citation tests the *reasoning*, and only the second can catch a finding whose own evidence contradicts it. A citation is also the most persuasive part of a review and the least likely to be checked: a linked changelog entry reads as settled fact, so the finding inherits authority it never earned, and a one-click `suggestion` block turns that borrowed authority into an applied edit.

Grep the cited document for the mechanism the finding names. One command usually decides it, which makes this an `algorithmatize-checks` case rather than a judgment call, and a fabricated mechanism produces a clean zero-hit result that is hard to argue with. Then quote the entry in the reply rather than paraphrasing the disagreement, and reproduce the behavior as the independent second leg.

Do not stop at winning the point. A finding that misread a source usually did so because the claim it questioned had nothing checkable next to it, so fold the citation into the file itself, per `fully-clean`'s note that a fresh review run re-derives from scratch and will not read the thread. (ai-config#762, 2026-07-28: a review held that `htmlwidgets::saveWidget(selfcontained = TRUE)` no longer needs pandoc, citing htmlwidgets 1.6.0 as having "switched to `base64enc::dataURI()`", and supplied a suggestion block deleting the `rmarkdown::pandoc_available()` gate. `grep -inE 'pandoc|base64'` over that NEWS file returned six pandoc hits and zero base64 hits, and the 1.6.0 entry says the path "now uses the `{rmarkdown}` package to discover and call pandoc" – so the citation established the opposite of the finding, and incidentally made the gate the *same lookup* htmlwidgets performs rather than a proxy for it. Applying the suggestion would have removed the only warning before a hard error, in the one step that exists for running headless. The reviewer accepted the rebuttal on the next round and called its own prior claim a hallucination.)

When a reviewer hedges a finding because it depends on code it cannot see, check whether *you* can see it — the hedge is an invitation, not a verdict. Automated reviewers work from the diff, so a finding that turns on a reusable workflow, a dependency's internals, or another repo's behavior arrives with language like "moderate rather than high confidence", "depends on behavior not visible in this diff", or "worth the author confirming intent". That hedge is a fact about the *reviewer's* visibility, not about how likely the finding is. You frequently have access it lacks: the repo cloned locally, a pinned dependency vendored in, or permission to fetch the source.

Reading it converts a maybe into a settled yes or no, and that changes the disposition. Confirmed, it earns a fix or a precisely-scoped follow-up issue with the mechanism recorded; disproved, it earns a Rebut with evidence instead of a vague "I think this is fine". Either way the next reader is spared re-deriving it. Quote the specific lines you checked, since a follow-up issue that merely repeats the reviewer's hedge is barely more useful than the review comment it came from.

(`UCD-SERG/serodynamics#274`, 2026-07-28: a review flagged possible duplicate review dispatch at moderate confidence, explicitly because the reusable workflow in `d-morrison/gha` was not visible to it. That repo was cloned locally. Reading both matchers showed the reusable fires on `@claude[[:space:]]+review` and the local job on a punctuation-tolerant superset, so the plainest phrasing — `@claude review` — matches both and dispatches twice. The follow-up issue could then record the exact overlap table and note that the upstream gap motivating the local job had since been closed, making "broaden upstream, delete the local job" a real option.)

References