# PR Review

*Read every changed line, prove what you find, hand back one review that lands in a single round.*

| | |
|---|---|
| **Produces** | A paste-ready review comment: a verdict, a scan table of findings, and a concrete fix for each — nothing posted to the PR. |
| **Use it when** | Someone opens a pull request and wants a careful read before it merges, or pushes fixes and wants a re-check of only what changed. |
| **Never** | Touch the PR. No posting, commenting, approving, resolving, or merging — read-only, always. |

## What it is

Reviewing a pull request well is not skimming a diff and typing "LGTM." It is reading every changed line, understanding what the change is supposed to do, then proving to yourself which of your worries are real before any of them reach the author. The output is a written review: a verdict, a short list of findings, and for each one a specific fix the author can apply. That review is a teaching document as much as a gate — the reason behind each point is what makes the fix land the first time instead of spawning a back-and-forth thread.

You reach for this whenever a diff is up for merge and deserves a careful human read: a feature, a bug fix, a security-sensitive change, a dependency bump. You also reach for it on a re-review, after the author pushed changes in response to your first pass and you need to look at only what moved.

The hard boundary is that you never mutate the PR. You read it, you form a view, you write the review, and you hand it to whoever asked. You do not click approve, you do not post a comment, you do not resolve a thread or merge the branch. Read access only. The person who owns the PR posts the review, or doesn't. Keeping your hands off the controls is what protects the author's ownership of their change. It also keeps the review honest — you are arguing your case in words, not enforcing it with a button.

What you produce, concretely: a single block of GitHub-flavored (or Azure DevOps-flavored) markdown. It is ordered so the reader gets the verdict and the shape of the review in under ten seconds, with the full depth of every finding one glance deeper.

## How you'd do it by hand

The tooling in the skill wraps two boring, deterministic steps — fetch the diff, render the review — around the one step that needs a human: the judgment in the middle. Strip the scripts away and the craft is the same. Here it is end to end.

### Fetch the change, read-only

Pull the diff and metadata without touching the PR. On GitHub that is the `gh` CLI; on Azure DevOps it is the REST API with an `az` bearer. Everything here is a GET.

```
# GitHub — metadata, unified diff, existing review threads
gh pr view  <url> --json title,body,files,additions,deletions,headRefOid
gh pr diff  <url>
gh api graphql -f query='... reviewThreads ...'   # resolved/outdated state

# Azure DevOps — no CLI diff endpoint; mint a bearer and hit REST
TOKEN=$(az account get-access-token \
  --resource 499b84ac-1321-427f-aa17-267ca6975798 --query accessToken -o tsv)
curl -s -H "Authorization: Bearer $TOKEN" \
```

```
  "https://dev.azure.com/<org>/<project>/_apis/git/pullrequests/<id>?api-
version=7.1"
```

Two things worth knowing. On GitHub the changed-file list caps at 100, so on a huge PR trust the diff's own paths over the file list. On Azure DevOps there is no unified-diff endpoint at all: you fetch each changed file's blob at the source and target commits and run `git diff --no-index` to reconstruct hunks. Line numbers then reflect real file content, so a `file:line` you cite is true. If the PR references a ticket (a ClickUp id or link in the title, body, or branch), pull the ticket's description and acceptance criteria too. You will need them for step zero.

Pin everything to the head commit you fetched. Every line you cite, every deep link you write, points at that exact SHA so it can't drift under you.

## Decide: first review or re-review

Look at the existing threads. If none of them are yours, this is a first review and you read the whole diff. If your earlier comments are already on the PR, this is a re-review. Look at only what changed since your last pass, acknowledge out loud what the author fixed, and do not reopen settled points or bolt on new unrelated asks. Late, unrelated churn is a review failure, not diligence. Keep your prior findings around so a resolved one becomes an acknowledgement rather than a repeat.

## Review in priority order

Read every changed line. Only lockfiles, generated code, and bulk data get skimmed; never skim a hand-written class or function and assume it's fine. Walk the change highest-altitude first, because that is where the comments that matter live.

1. **Intent (step zero).** Before you judge how the code is written, understand what it is meant to do. Read the PR body and, if there's a ticket, its acceptance criteria. A change can be flawless code that solves the wrong problem, and correctness means nothing without a target. Treat the description as a claim to verify, never a fact to repeat.
2. **Design / architecture.** Does this change belong here, integrate cleanly, and land at the right time? The most valuable comments you'll write are at this level.
3. **Correctness.** Edge cases, null and error paths, off-by-one, concurrency and races, resource leaks. Watch for read-modify-write with no concurrency token and check-then-act across boundaries (TOCTOU) — two failure shapes that pass every test and lose data in production.
4. **Security.** The checklist below.
5. **Tests.** Present, and meaningful — judged by intent, not coverage percentage.
6. **API / interface.** Names, contracts, backward compatibility.
7. **Readability / complexity.** Complex code invites future bugs; that is a reason to flag it, not a matter of taste.
8. **Performance.** Only where it is demonstrably load-bearing. No speculative asks.
9. **Docs / comments.** Do they match the behavior? A comment should explain why, not restate what.

### The security checklist

Run this on every diff, hard on any new endpoint or input:

- Secrets committed — keys, tokens, connection strings, default credentials.
- Injection across every input vector: params, headers, bodies, file uploads.
- Authentication and object-level authorization on new endpoints. Hunt insecure direct object references (IDOR) and privilege escalation — can one user reach another's object by changing an id?
- Server-side validation that rejects rather than sanitizes. Watch for empty strings that slip past an `if (value)` filter and silently widen a query.

- Supply chain: every new or bumped dependency gets an existence and provenance check (below); read lockfile and CI/build-script edits; on CI, `pull_request` versus `pull_request_target` is a real distinction.
- PII in logs or error messages.
- Failures fail closed, not open.

**The behavioural and currency check**
Two questions most reviews skip, and both catch real defects.

**What does it actually do?** Trace the real behavior and state it in your own words, then diff that against what the PR body, commit message, ticket, and code comments claim. A "fixes X" that only half-fixes X, a dimension the code leaves unset that the description says is handled, a metric or alert that never fires — each is a defect, flagged like any bug. The description is a hypothesis you test against the code, not a summary you echo. Where you can, exercise the real surface: read the test that covers it, resolve the id or URL, run the thing. A check that mocks the surface the change lives on proves nothing about that surface.

**Is it current?** Judge the change against best practice as of the month you're reviewing in, not habits from three years ago. Verify against the vendor's or library's live docs, not memory — APIs, idioms, security defaults, CI patterns, dependency versions, model ids. Call out anything deprecated or superseded, and when currency is the point, state the date and the source ("as of this month the vendor recommends X; this uses the older Y"). The rule cuts both ways: don't fault code for missing a "best practice" you haven't confirmed is actually current.

**The AI-authored lens**
When the diff is machine-written — an attribution trailer, the author says so, or it's obvious — the failure profile flips. AI code fails plausible, not sloppy: a clean surface over a hidden misalignment. Turn scrutiny up, not down.

- **Intent-alignment.** Does it do what was asked, or something coherent nearby? Plausible is not requested.
- **Dependency existence and provenance.** Registry-check every new package before you trust it. Hallucinated names recur across prompts and attackers pre-register them (slopsquatting). `npm view <pkg> time.created dist-tags.latest repository.url` — a package that appeared last month with no repo is a finding, not a dependency. Judge age and provenance together: a recent package under a known org can be fine. Name the check you ran in the review.
- **API reality.** Called methods and options must exist in the pinned version of the library, not in some blend of versions the model trained on. Spot-check the unfamiliar ones against that version's docs.
- **Over-engineering.** Speculative generality is the signature tell: abstractions for single-use code, config nobody asked for, handlers for states that can't occur. Flag it.
- **No polish credit.** A well-written-looking block earns more scrutiny, because polish is exactly what hides the misalignment.

## Verify every finding before you keep it

This is the discipline that separates a review people read from one they learn to skim. For each thing you want to flag:

- **Anchor it.** Cite the real `file:line` you read and the failure path. If you can't point at a line and explain how it breaks, it isn't a finding — it's a hunch, and it gets dropped.
- **Try to disprove it.** Assume every finding is a false positive until you fail to refute it. Re-read the surrounding code, check the null path, look for the guard you might have missed. The ones that survive an honest attempt to kill them are the ones worth writing.
- **Confirm the line against the real diff.** The number must point at the code you describe. Never carry a line from another file or a stale version.
- **State your evidence tier.** "Verified by tracing X" is a different claim from "this looks like it might." Never dress the second up as the first. If you're still unsure after trying to refute it, downgrade to a `question` and label it as one.

The mechanical version of anchoring, which the skill automates and you can imitate: keep a short verbatim snippet of the exact line each finding sits on, and before you finalize, grep the diff for that snippet on that file. If it isn't there, the line drifted or you hallucinated it — either way the finding doesn't ship.

### Write comments that land in one round

How a finding is phrased decides whether it's fixed once or argued about. Every comment:

- **Names one concern,** anchored to an exact `file:line`.
- **Carries a Conventional-Comment label** so severity is never ambiguous: `praise` / `nitpick` / `suggestion` / `issue` / `question` / `todo` / `note`, plus `(blocking)` or `(non-blocking)`. Default to non-blocking; blocking is for correctness, security, and broken interfaces.
- **States problem, why, and fix** — the consequence, not just the observation. "The handler swallows the error, so a failed write looks like success to the caller" beats "this feels off." A code-suggestion block the author can apply directly is best.
- **Comments on the code, never the author.** Name the code's property: not "why did you use threads here," but "the concurrency model adds complexity with no measured benefit; single-threaded is simpler." No "you did / you should."
- **Puts clarity in the code.** If something needs explaining, ask for clearer code or a repo comment — not an explanation that lives only in the review thread where no future reader will see it.
- **Asks real questions only.** Genuine uncertainty gets the `question` label and an actual question mark. A demand dressed as a question ("did you consider not doing this?") is neither honest nor useful.

Two severities, so there's never doubt about what holds the merge: blocking and non-blocking. The verdict falls out of them mechanically — any blocking finding means request-changes; otherwise non-blocking findings that matter mean comment; otherwise approve.

### Shape the output scan-first

A flat wall of equal-weight bullets is a failed review even when every bullet is correct, because the reader can't tell the blocker from the nit without reading all of them. Layer it instead:

- **Scan layer, read in seconds:** a one-line verdict plus a tally (how many blocking, how many not), a one-sentence TL;DR, a one-sentence trace of what the change actually does, the ticket line if there is one, and a one-row-per-finding table (severity, an eight-word headline, `file:line`). On a request-changes, a "mergeable after" line naming the blocking headlines so the path to green is one glance.
- **Depth layer, on demand:** each finding's full problem, why, and fix, collapsed behind a `<details>` on GitHub or as flat numbered sections on Azure DevOps, which doesn't render the disclosure.
- **Ordered by altitude:** design, correctness, and security before readability and nits; blocking before non-blocking. The first thing scanned is the thing that matters most.
- **A short praise footer** — at most two specific things done well. Skip it rather than pad it.

The depth is never cut to make the scan short. Same findings, same rigor, just layered.

Two honesty notes. Past roughly 400–500 changed lines, review rigor measurably drops, and a very large diff won't fit in your head at once. Work it file by file and say in a coverage line where you focused and what got a lighter pass — an honest "I read the core hard and skimmed the fixtures" beats a false clean bill. And only functional status markers belong in the text — a small closed set that signals severity and verdict. No decorative emoji; a review is not a chat message.

## Why it's built this way

Every rule here exists because a specific failure mode is common and expensive. The craft is the accumulated scar tissue.

**Verify before you comment, and refute before you keep.** A language model — and a tired human — will produce plausible findings that dissolve on a second read: a null check that's actually three lines up,

a race that can't happen because of a lock you skimmed past. If those ship, two things break. The author wastes a round disproving your ghost, and worse, they learn that your review is noise. Once an author starts skimming past your comments, the one real bug in the batch drowns with the nine imagined ones. Anchoring every finding to a line and trying to kill it first is the price of being believed.

**Signal over noise.** This is the same lesson from the other side. A review with five nitpicks and no design comment on a substantive diff hasn't been thorough — it's failed. It spent the author's attention on things a linter already catches and never looked up at the shape of the change. One well-anchored blocking finding is worth more than ten nits. Volume reads as diligence and isn't. As of 2026 this is the dominant shift in review practice: human attention goes to design, correctness, and security, and low-confidence machine noise is a known failure mode, not a sign of effort.

**Evidence tiers.** "Looks like" and "is" are different claims, and collapsing them is how a review loses credibility. When you mark your confidence honestly and downgrade uncertainty to a labeled question, the author knows exactly how much weight to give each point. Selling a hunch as a confirmed bug spends trust you don't get back.

**Read-only, never touch the PR.** The change belongs to its author. A review is an argument for a set of changes, and an argument you win with words is durable. One you enforce by clicking merge or resolving a thread is a power move that erodes the working relationship. Keeping strictly to read access also keeps you honest — you have to make the case well enough that someone chooses to act on it.

**Comment on the code, carry the why.** A finding without its reason is an order, and orders get resented and misapplied. The consequence is what teaches, and what makes the fix correct rather than merely compliant. Naming the code's property instead of the author's choice takes the ego out of the exchange, so the conversation stays about the change.

**Unblock, don't hold hostage.** If the codebase is better after the merge and only minor tweaks remain, the right move is approve-with-comments and trust the author. Review is there to protect code health, not to gate a PR on preference dressed as a defect. Correctness and security still block, and a deferral to a "follow-up ticket" still gets scrutinized before you let it stand. But minor means minor, and a review that holds every PR hostage to taste stops being read as help.

**Judge currency as of now.** Best practice moves. A pattern that was correct in 2022 can be deprecated today, and a "fix" can quietly reintroduce an idiom the vendor has since warned against. Checking the current docs rather than memory is what keeps the review from teaching yesterday's habits. The discipline cuts both ways: don't wave a pattern through because it used to be fine, and don't fault code for missing a best practice you haven't confirmed is current.

**The AI lens, specifically.** Machine-written code inverts the usual heuristic that clean code is trustworthy code. The defects hide under the polish: a hallucinated dependency, a method that exists in a different version of the library, an abstraction for a case that can't occur, an implementation that's coherent but solves a nearby problem instead of the asked one. The extra checks exist because the surface no longer signals the substance.

The through-line: a review is trustworthy exactly to the degree that its specific claims are provable and its noise is near zero. Everything above is in service of that.

## At a glance

```
# 1. Fetch read-only — GitHub
gh pr view <url> --json title,body,files,additions,deletions,headRefOid
gh pr diff <url>
gh api graphql -f query='... reviewThreads ...'      # resolved/outdated state

# 1. Fetch read-only — Azure DevOps (no diff endpoint; reconstruct hunks)
TOKEN=$(az account get-access-token \
  --resource 499b84ac-1321-427f-aa17-267ca6975798 --query accessToken -o tsv)
```

```
curl -s -H "Authorization: Bearer $TOKEN" \
  "https://dev.azure.com/<org>/<project>/_apis/git/pullrequests/<id>?api-
version=7.1"

# Verify a dependency the diff adds before trusting it
npm view <pkg> time.created dist-tags.latest repository.url

# Confirm a finding's line hasn't drifted (anchor check by hand)
gh pr diff <url> | grep -F '<exact snippet from the line you cited>'
```

Pre-review checklist:

- Read the PR body and the ticket's AC before judging a single line. Correctness needs a target.
- Read every changed line; skim only lockfiles, generated code, and bulk data.
- Walk it in order: intent, design, correctness, security, tests, API, readability, perf, docs.
- Anchor every finding to a real `file:line`; try to refute it; drop what won't survive.
- State your evidence tier. Unsure after refuting → downgrade to a labeled `question`.
- One concern per comment, Conventional-Comment label, problem + why + concrete fix.
- Comment on the code, not the author. Put clarity in the code, not the thread.
- Two severities. Verdict is mechanical: any blocking → request-changes; else matters → comment; else approve.
- Scan layer first (verdict, tally, table), depth on demand, ordered blocking-first.
- On a big diff (> 500 lines), state coverage honestly. On a re-review, only what changed.
- Verify behaviour against the PR's claims; verify currency against live docs.
- Never post, comment, approve, resolve, or merge. Hand back the review; the owner posts it.