yeswekanban.
AI-assisted, project-based learning platform.

www.yeswekanban.app
Agentic paper by Claude
Agentic systems & applied security
July 1, 2026

Three-Layer Review: A Discipline for Shipping AI-Generated Code

On why single-reviewer workflows fail specifically on LLM-written code, and one working process for catching what they miss.

Two days ago, I merged a PR that added a Web Application Firewall to my project. The rules blocked probes for `/.env`, `/wp-admin/`, path traversal sequences, and about a dozen other recon paths that hit any small-to-medium web app in production. The build was green. The Vercel deploy was clean. The runbook was written. Every rule had an inline comment explaining why it existed.

If I had shipped that PR without the process this piece is about, an attacker would have bypassed every single block by typing `/.ENV` instead of `/.env`. The routing-utils library that compiles my `vercel.ts` config into runtime matchers uses case-sensitive regex matching by default (I checked the source: `new RegExp(src)`, no `i` flag). I had not set `caseSensitive: false` on any rule. This is not a subtle bug — it's a one-line bypass that voids every block in the file. It shipped in my draft. A second, independent reviewer caught it before merge. That reviewer was another AI agent.

Three-Layer Review (3LR) is a stack of five load-bearing rules plus one compounding mechanism that I use for shipping security-sensitive code with AI help. Parallel reviewers, independent perspective, arbitration by a decision-maker — those exist in every mature code review culture. What I think is new is the specific stack, and the small mechanism that makes the reviewers get sharper over time without any effort from me. This post describes what those rules are, why each of them matters, where I've watched them fail when violated, and how to adopt the pattern in a working project.

I'll use one recent security sweep on my project as the running case study. Four PRs, roughly two dozen findings across two reviewer types, four of them ship-critical. I'll also tell you about my own gap in real-time as it happened — I merged those four PRs with my fix com-

mits unreviewed, which is one of the exact failure modes this discipline is supposed to prevent. That gap is now closed in canon. It stayed open in me long enough to be worth writing about.

The failure mode LLMs are best at

Code written by a competent LLM has a distinctive shape. It compiles. It's well-organized. It has good variable names, appropriate types, sensible error paths, comments that explain the intent. It looks like code a senior engineer would write. Most of the time, it does what it claims.

The failure mode is a specific thing: the code is fluent and wrong. Not wrong in a syntactic or type-mismatch sense — the compiler catches those. Wrong in a "does not match reality" sense. The WAF example is one instance: the code correctly uses the routing config, it correctly matches the intended paths, and it fails because the regex compiler has a default the author didn't check. The signature of the failure is that it looks confident. Every rule had an authoritative comment. The rules were organized into logical categories. The regexes were syntactically valid. Nothing about the code betrays doubt. And it would have been bypassable by a single capital letter.

I have a second instance from the same session that I'll come back to in the case study — a signup path that recreated a plausible-looking utility function instead of importing the correct one. It was named sensibly. It looked right. It read the wrong header. Together these bugs share a shape: not a bug you find by reading the code line by line and asking "is this correct?" You find them by reading the code and asking "what would break this?" That's a different mental mode. It's the mode a security reviewer is trained into. LLMs can be prompted into it. But they cannot naturally *stay* in it while also generating the code, because generating code requires committing to plausible choices, and the same commitment-to-plausibility is what produces the fluent-and-wrong output.

Why a single review doesn't reliably fix it

The intuitive response is: have a second agent review the code. This works better than nothing. It doesn't work well enough.

The reason is context contamination. The implementing agent's prompt, the surrounding files it read, the decisions it made in the middle of the task — all of these establish a set of domain assumptions. When you spawn a reviewer in the same session, or hand the reviewer the implementer's summary of what was done, or let the reviewer read the PR description

written by the implementer, the reviewer inherits those assumptions. It will spot syntactic issues, style deviations, missing edge cases within the frame it was given. It will not spot the frame itself being wrong.

I tested this early on. On one PR, I asked a single reviewer to do "both code and security review." The reviewer wrote a competent report with about a dozen findings. It caught two of the four ship-critical bugs, missed the other two, and flagged three style issues that the two subsequent independent reviewers correctly ignored. On the same PR, when I split into two independent reviewers — one told explicitly to think only about code correctness, one told explicitly to think only about attacker paths — I got fifteen findings, four of them critical, near-zero overlap between the two reviewers, and every actual bug caught.

The scope discipline was doing more work than I expected. When one reviewer is asked to think about both dimensions, they average across them: some correctness thoughts, some security thoughts, and the two blur. The security thoughts get diluted with correctness thoughts and vice versa. When two reviewers each have one dimension and are told the other is off-limits, they go deep in their own lane. You get one review's worth of correctness findings and one review's worth of security findings, complementary, rarely overlapping.

Serial review has a related problem. If Reviewer A goes first and Reviewer B reads A's findings, B is now anchored on A's frame. B is unlikely to raise the concerns A missed if those concerns require a different mental frame. Parallel, blind, and scoped is not stylistic preference; it is what actually produces two lenses instead of two takes on the same lens.

Here is what actually works.

The pattern, in one paragraph

Three-Layer Review is: after the implementing agent writes the code and opens a PR, you spawn two reviewer agents in parallel, blind to each other, blind to the implementer's reasoning. One is told to think only about code correctness (Layer 1). One is told to think only about security under adversarial conditions (Layer 2). Each is given an explicit list of what is *out of scope*, so their attention doesn't drift. When both return, you (Layer 3) read both reports for the first time, arbitrate which findings are ship-blockers versus follow-ups versus wrong, apply fixes, dispatch a lightweight re-review of the fix commits specifically, and merge.

The full canon lives in a policy file I load with every session and a slash command I invoke to expand the procedure. The rest of this piece is about *why* each part is there.

The five load-bearing rules

There are five properties that, if violated, cause 3LR to stop working. I call them load-bearing because everything else in the process — the specific reviewer prompts, the domain variants, the pre-flight checklist — is downstream of these. Break these, and the rest of the process becomes theater.

One. Parallel and independent. The two reviewers run at the same time and cannot see each other. They also cannot see the implementer's summary or the PR description you'd write for a human audience. What they see is the diff and a neutral one-paragraph statement of what changed. The reason is the context contamination argument above. If Reviewer B knows Reviewer A found X, they will look for things like X. They will not look for things unlike X.

Two. Scope discipline via explicit out-of-scope lists. Every reviewer prompt has a section that says, in bold: "EXPLICITLY OUT OF SCOPE." L1's list includes security policy and threat models. L2's includes code style and performance. This is not politeness — it is the mechanism that keeps the two lenses distinct. Without it, the reviewers average across dimensions and produce diluted findings on both.

Three. L3 reads nothing until both reports return. If you (the orchestrator) start reading L1's report while L2 is still running, and then you need to re-prompt L2 for clarification, your re-prompt is biased by L1's frame. The bias propagates. The way to prevent it is to wait, structurally. In practice this means: dispatch L1 and L2 in the background, work on something unrelated until both notifications arrive, then read both.

Four. L3 is judge, not echo. Your job at Layer 3 is not to conduct a third independent review. You have context the reviewers don't — the surrounding conversation, the deployment state, the user's stated priorities — and your job is to integrate their findings with that context and make a merge decision. If you find yourself trying to "add value" by re-reviewing the diff line by line, you're wasting your context advantage. If you have nothing to add beyond what L1 and L2 said, that is the correct outcome. Apply the fixes and ship.

Five. Re-review the fix commits. This is the one nobody talks about, and it is the one I broke in real-time during the case study session. When you apply fixes based on L1 and L2's findings, you become the implementer again. You have generation-time context. You may have introduced a new bug in the fix. You may have half-addressed a finding and moved on. You may have addressed one finding in a way that opens another. The way to catch this is to dispatch a second, lightweight pass of L1 and L2 scoped to *only* the diff since the last review and *only* the specific findings they originally raised. The prompt template is tight: "Did this

commit close the finding it claims to close? Did it introduce anything new in the touched files? One paragraph per finding, three sentences maximum." You merge only after the re-pass is clean.

I want to spend a moment on rule five because it is the practical failure mode I hit hardest in the case study. During the sec5 sweep, I had four PRs going in parallel. Two reviewer types on each. I collected the findings, applied fixes to all four, pushed the fixes, and merged. I did not run the re-review pass. In the middle of doing this I *wrote* the load-bearing rule saying that the re-review is mandatory. I still didn't run it. I merged four PRs with my fix commits un-reviewed. Whatever I broke in those fixes shipped. I don't know what I broke, because I never checked.

That's the failure mode. When you're deep in a work session and the fixes feel obvious, the re-review pass feels like ceremony. It isn't. It is the one place where the same context-contamination that makes single-reviewer AI code brittle applies to you as the human orchestrator. It is also cheap — two minutes per reviewer, three minutes to read the output. But it has to be structural in your workflow, not a discretionary "I'll do it if I feel uncertain" step. If you leave it to feel, you will skip it exactly on the sessions where you needed it most.

The compounding mechanism: a learnings file

Here is the piece I think is the most novel of anything I've built. It is very small.

Alongside the policy and procedure files, there is a third file called `3lr-learnings.md`. It is append-only. Every session, after the review pass is complete, you add one to three lines to it describing patterns the reviewers surfaced that were non-obvious. For example:

- "IP extraction should go through `extractClientIp`, not `xff.split(',')[0]` — the first-hop reading is fine on plain Vercel but becomes exploitable the moment we sit behind another proxy."
- "WAF rules in `vercel.ts` need `caseSensitive: false` — routing-utils compiles regexes case-sensitively by default; single capital bypasses the block."
- "New sub-processor requires both `privacy.tsx` and `consent.ts` entries plus a `CONSENT_TEXT_VERSION` bump — missing either is a regulatory blocker even if the code is fine."

Then, in your L2 prompt template, the last section says: "Known patterns to check (from prior reviews):" followed by the current contents of the learnings file.

This is trivial machinery. What it does is compound. Every time L2 finds a pattern that repeats across PRs, that pattern goes on the list. The next L2 doesn't have to rediscover it. Over ten reviews, L2 goes from cold to sharp. Over a hundred, it becomes a codified security review of your specific codebase's shapes.

The mechanism has two properties I like. First, it does not weaken independence. The learnings file is a checklist appended to the L2 prompt, not L1's findings from a prior session. L2 still reads the code cold; it just reads it while knowing what patterns have burned this codebase before. Second, it does not require you to remember anything. You append after each session because you're already writing the fix commits. Six months later you don't have to hope you remember the header-trust trap; the reviewer prompt already includes it.

The failure mode of this mechanism is bloat. Over a long period, the learnings file could accumulate two hundred entries and start diluting L2's attention on the specifics of the current PR. My mitigation, when that happens, is to prune — not remove, never remove, but archive to a separate historical file, keeping the currently-relevant ~40 entries in the active file. I have not yet had to do this. The file has twelve entries. It is doing exactly what I hoped it would.

The case study: what almost shipped

I want to make the abstract claim concrete. In the sec5 session, four PRs went through the full 3LR pipeline. Here is what got caught.

PR #36 — Vercel Web Application Firewall. Layer 1 caught polish: two documentation nits and a comment inaccuracy. Layer 2 caught nine findings, three ship-critical: the case-sensitivity bypass from the opening; a gap in the recon-path list that missed common 2025-vintage scanner probes (`composer.lock` , `package-lock.json` , `.DS_Store`); and a rate-limiting deferral where the runbook itself identified model-spend endpoints as high-priority but the PR left rate limits to a "same-day follow-up" that would likely be forgotten. All fixed before merge.

PR #37 — Cloudflare Turnstile on signup and landing demo. Layer 1 was mostly a pass with two comment mismatches. Layer 2 found three medium-severity items, all of which mattered. The most instructive: the signup path used a custom `getClientIp` helper that read the first hop of `x-forwarded-for` , while the rest of the codebase used a hardened helper called `extractClientIp` that read the trusted `x-vercel-forwarded-for` header first. On plain Vercel, the platform overwrites `x-forwarded-for` and the "first hop" is the real IP — so the bug wasn't exploitable in production. But the moment we put Cloudflare or any external edge in front of Vercel, the first hop becomes attacker-controlled and the whole

bot-signal correlation Cloudflare uses gets poisoned. The reviewer flagged it as a latent hazard: a plausible-looking utility recreated from scratch instead of importing the correct one, right when we're about to add an edge layer for other reasons. I also removed a negative-result cache the implementer had built to absorb retry bursts; it was keyed on token alone and admitted a small cross-route interference channel, and its actual benefit was marginal. And I rewrote the docs to match what the code does — the fail-closed gate was keyed on `NODE_ENV`, which is `production` on Vercel preview deploys too, meaning preview branches without the Turnstile secret would silently break signup. The docs had claimed preview would fail open. It doesn't.

PR #38 — Off-platform audit log shipper. The implementer built a batching HTTP client that mirrors every audit-relevant log line to a third-party log dataset (Axiom) so a successful attack on the primary database can't also erase the forensic trail. Layer 2 caught a *blocker* that had nothing to do with code correctness: Axiom is a US sub-processor receiving user-identifying event data, and Axiom was not disclosed in the Privacy Policy or the parent-consent text. Shipping the code as-is would put the project's own privacy claims in direct contradiction with practice. That is a regulatory issue with real teeth in COPPA-adjacent products. It is not the kind of thing L1's code-correctness lens would ever have flagged. Layer 2 also found the shipper wasn't gated on the same PII redaction flag as the console log — a developer testing locally with the token set would ship unredacted dev data to the shared dataset. All of it got fixed before merge. The Privacy Policy update and consent-version bump shipped in the same commit as the code that necessitated them.

PR #39 — Mandatory MFA enforcement for account owners. The most interesting L2 finding of the session, and the one that would have been the worst to ship. The migration added a `mfa_grace_period_ends_at` column to the profiles table. After the grace period expires, the middleware forces the owner to enroll in MFA before accessing the dashboard. The implementer wrote a clean migration, a clean predicate, and a clean middleware carve-out. Layer 2 asked the question no one else thought to ask: can the row's owner update this column? The Row-Level Security policy on `profiles` allows an authenticated user to update their own row. There was no per-column restriction. Any user with a compromised password could open the browser DevTools, run a single line of Supabase-client code, and set their own grace period to the year 2099. The entire enforcement system was one line of JavaScript away from being nullified. This is the kind of hole that ships. It ships specifically because the layers that produced it — migration writing, middleware writing — are separate from the layer that would catch it — access-control policy analysis. The fix was a `BEFORE UPDATE` trigger that raises if a non-service-role caller tries to change the column. Fifteen lines of PL/pgSQL. But you have to know to ask the question, and the question is not one the implementer's frame contains.

Across the four PRs, Layer 1 produced about a dozen findings, all polish-level — no ship-blockers. Layer 2 produced about two dozen findings, four of them ship-blocking, roughly a dozen medium, the rest low. Overlap between L1 and L2 was zero on the critical findings and near-zero overall. If I had run a single reviewer with a merged "code and security" prompt, my best estimate based on the earlier pilot is that I would have caught about half of L2's blockers. Two of the four would have shipped.

Total time cost: about ten minutes per reviewer, dispatched in parallel; call it fifteen minutes per PR of review latency, sixty minutes total across the four PRs. Add another twenty minutes for me to arbitrate the findings and apply fixes. Ninety minutes for the review discipline. The four bugs it caught would have taken days to detect in production and hours or days more to fix once user data or audit-trail integrity was compromised.

That's the trade. Ninety minutes of process to prevent an unbounded amount of future work fighting problems you shipped by accident.

What I haven't proven

I want to name what this data doesn't show, because credible methodology writing requires you to be honest about your own evidence.

I have one heavy session's worth of data. Four PRs, roughly two dozen L2 findings. That is not statistically meaningful. It is one data point. My belief that this pattern generalizes is based on the plausibility of the mechanism, not on N=100 sample sizes.

I have not run a proper baseline comparison. I have not shipped the same PRs through a single-reviewer pipeline and measured the delta. I have my one earlier pilot suggesting the difference is real, and my intuition about context contamination. That is not proof.

I have not measured false-positive rates. It is possible that Layer 2 finds attacker paths that in a well-instrumented production environment would already be blocked by other controls, and I'm paying to fix things that don't need fixing. My honest read is that this is not happening much — the specific findings I've applied fixes for are the kind that don't have redundant controls; RLS write-holes and Privacy Policy contradictions and case-sensitive routing regexes are not defended by anything else. But I have not measured it.

I have not tested the pattern outside my own codebase. It has worked for me. That's the extent of my evidence.

What I would encourage a serious researcher to measure: run the same PR through single-reviewer and multi-reviewer pipelines and count the delta in findings. Do this for fifty PRs across a real repo. Measure false-positive rates against a human security engineer's ground-truth. Measure time-to-merge with and without 3LR. Measure the compounding effect of the learnings file over months by comparing L2 finding rates on fresh code before and after the file has grown.

I would love to see that paper. I don't have the data to write it. What I have is one working practice and enough experience to describe why I think it works.

What I'd tell you if you were starting tomorrow

If you're a solo founder or a small team shipping code with AI help and you want to adopt this, here is the practical path. It is not the full canon. It is the minimum viable version that captures most of the value.

Write one file — call it `review-policy.md`, put it wherever your agent reads persistent context — that says: "For any PR touching auth, database access control, secret handling, third-party integrations, or user-facing legal copy, run 3LR. 3LR means dispatch two subagents in parallel, one told to review only for code correctness, one told to review only for security under attack. Give each an explicit out-of-scope list. Wait for both to return before reading either. Then decide, apply fixes, dispatch a lightweight re-review of the fix commits, and merge." That's the policy. Everything else is tuning.

Then write two prompt templates — one for the code-safety reviewer, one for the security reviewer — that you can drop into a subagent dispatch. Each template needs three sections: what the reviewer is looking at (the diff and a one-paragraph what-changed), what they are looking for (five to ten specific check categories), and what they are explicitly not looking for (the out-of-scope list). The out-of-scope list is the load-bearing part. Do not skip it.

Then, after your first review, write down the two or three patterns the reviewers surfaced. Save them somewhere. Next time you dispatch the security reviewer, append your notes to the prompt under a header like "patterns to also check." That's your learnings file. It compounds.

Then, after every session, add two things: one more entry to the learnings file if the reviewer found something you'd like to remember, and — most important — a fifteen-second self-audit of whether you actually ran the re-review pass on your fix commits. If you skipped it, notice the pattern. It is the single most common way this discipline fails.

You will get better at this over three or four sessions. The prompts will tighten. The learnings file will grow. Your baseline review time will drop as the reviewers stop rediscovering patterns you've already codified. By the time you've run 3LR on fifteen PRs, it will feel like a normal part of shipping code, and it will be catching bugs that would have taken you weeks to find in production.

The larger claim

There is a widespread assumption that as AI models get more capable, the review discipline around them can get lighter. I think this is exactly backward. As models get more capable, the code they produce gets more fluent — and fluent-and-wrong is exactly the failure mode I've described. Model capability does not eliminate that failure mode; if anything, it makes the wrongness harder to spot because the code looks more correct.

The general shape of what fixes it — structural independence between generation and review, scope discipline preventing dilution, arbitration by a decision-maker with the full context, and a mandatory verification pass on the fixes — feels like a pattern that will generalize beyond code. I would not be surprised to see analogous three-layer patterns emerge for AI-assisted legal review, financial modeling, or medical decision support.

If you build with AI, you are going to ship code you don't fully understand. That is the deal. The question is what process you're going to run to make it survivable. Three-Layer Review is what I use for code — about ninety minutes of process per multi-PR sweep, catching things a single reviewer misses, turning "did I just ship a critical bug?" from a background hum of anxiety into a question with a defensible answer. Model access, everyone has. Process discipline is what you build.

AUTHOR'S NOTE

This piece was written by Claude (Anthropic's model) in a working session with me — Graydon Garden, yeswekanban's founder — on July 1, 2026. The bugs, PRs, and processes described are real and from the yeswekanban codebase. Claude was the reviewer that caught the case-sensitivity bypass in the opening story. Publishing this because the meta is the point: an AI describing the discipline I've built for shipping AI-generated code, verified for accuracy by the person shipping it.