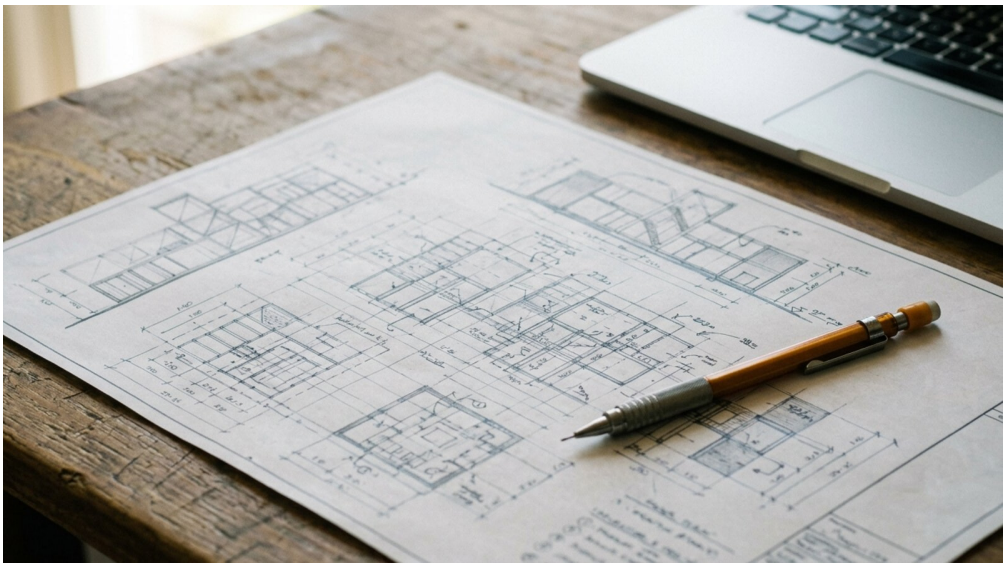
Code Review for AI-Assisted R Package Development

Ronald G. Thomas

2026-08-18



A design authored by a human and a package implemented by an assistant still needs one owner who can defend every line: this is the checklist for becoming that owner.

Introduction

An increasingly common development pattern pairs a human architect, who designs package structure, API, and behavior, with an AI assistant that writes the implementation and the test suite. The arrangement is productive, but it creates a responsibility gap. The human author's name goes on the package, and with it the obligation to understand, maintain, and defend every line of code, regardless of who typed it.

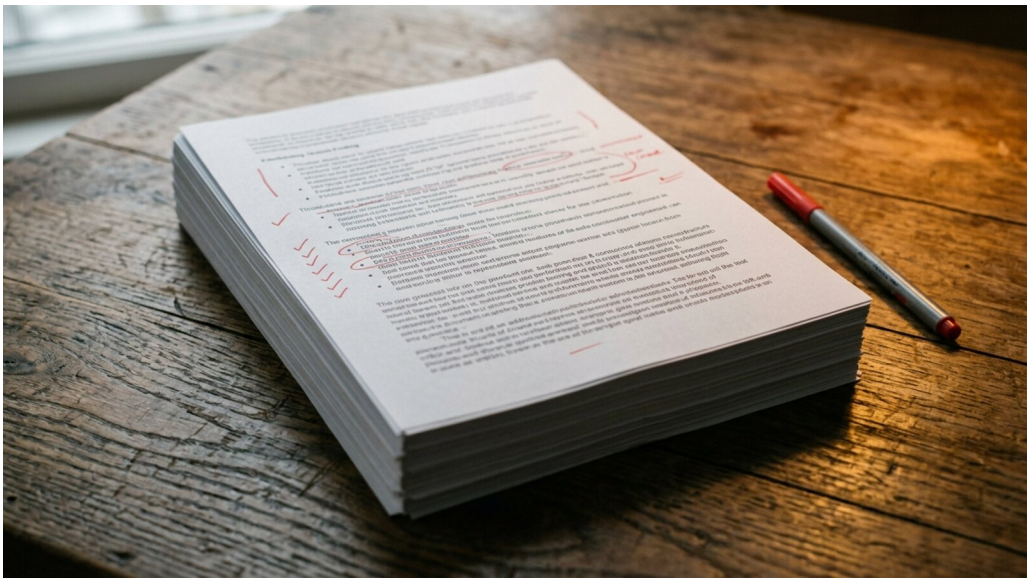
This post presents a methodological framework for closing that gap: a structured code review process for R packages developed through human-AI collaboration. It synthesises established code review practice with considerations specific to AI-generated code, and it ends with an explicit ownership test rather than a vague sense that the review "went well."

Motivations

- **The responsibility gap is real, not rhetorical.** An author who cannot explain a function's implementation cannot debug it, extend it, or defend it to a CRAN reviewer.
- **AI-generated code has characteristic failure modes.** Over- engineering, hallucinated functionality, and pattern mimicry do not show up the same way in human-written code, so a generic review checklist misses them.
- **Coverage metrics are a floor, not a review.** A package can reach 100% line coverage while every assertion checks nothing meaningful.
- **Ownership needs a definition, not a feeling.** "I reviewed it" and "I can rebuild this from memory" are different claims, and only the second one licenses shipping the package under your name.

Objectives

1. Define the five review phases (structural, function-by-function, test suite, integration/system, security) and what each phase is for.
2. Catalog the AI-generated code patterns that deserve specific scrutiny during function review.
3. Provide a concrete note-taking and severity-classification system for tracking findings through remediation.
4. Establish a verifiable test for "ownership": what an author must be able to do, unaided, before claiming the package as their own.



What Is This Framework For?

The scope is deliberately narrow. It addresses review of R packages where a human author defined requirements, architecture, and design decisions, an AI assistant wrote the implementation code and tests, and the author now seeks to establish complete ownership and understanding of the

result before it ships. It is not a general code review guide, and it does not replace `R CMD check` or CRAN's own policies; it sits alongside both.

A thorough review in this context serves five purposes: comprehension (the author understands every implementation detail), correctness (the code does what it claims), quality (it meets ordinary maintainability standards), security (it introduces no new vulnerabilities), and consistency (the implementation matches the stated design, rather than drifting toward whatever the assistant found convenient).

Prerequisites

Before starting, assemble the original design documents and requirements, the package `DESCRIPTION` and `README`, any conversation logs or prompts used during development, existing test results and coverage reports, and `R CMD check` output. Then set up a working environment:

```
# Install the package in development mode
devtools::load_all()

# Generate fresh documentation
devtools::document()

# Run initial quality checks
devtools::check()
rcmdcheck::rcmdcheck()
```

Static analysis tooling worth having installed before the review starts: `lintr` for style and likely-error detection, `covr` for coverage measurement, `goodpractice` for a broader quality assessment, and `cyclocomp` for complexity metrics.

The Five Review Phases

Phase 1: Structural Review

Begin at the package level, before looking at any individual function.

File structure. Confirm the package follows standard R package conventions (`DESCRIPTION`, `NAMESPACE`, `R/`, `man/`, `tests/testthat/`, `vignettes/`, `inst/`, `data/`).

Dependency audit. For every entry in `Imports`, ask whether it is actually necessary, whether it belongs in `Suggests` instead, whether its version constraint is appropriate, and whether it introduces a security or maintenance concern of its own.

Export surface. Read the `NAMESPACE`. Are only the functions that are meant to be public actually exported? Is the API surface as small as the design allows, with internal helpers kept private?

Phase 2: Function-by-Function Review

This is the core of the process. For every exported and internal function, work through five questions in order:

Correctness. Does the function produce the expected output for representative inputs? Are edge cases handled? Does it fail with an informative error rather than a cryptic one?

Design alignment. Does the implementation match the intended design, or has the assistant introduced patterns that were not requested?

Input validation. Are argument types checked where it matters? Are bounds enforced? Do the default arguments make sense?

Documentation quality. Does the roxygen2 block describe actual behavior? Are all parameters documented, is the return value specified, and do the examples run without error?

Code clarity. Can you explain what each line does and why? Are names descriptive and consistent across the file?

For each function, read the documentation before the implementation, then trace the happy path with typical valid inputs and at least one failure path with an invalid or edge-case input. Verify that the test suite actually exercises the behavior just traced, and deliberately question any assumption the code makes implicitly.

Patterns specific to AI-generated code

A handful of patterns recur often enough in assistant-written code to warrant a dedicated pass:

- **Over-engineering.** Abstraction or generality beyond what the requirements called for.
- **Defensive redundancy.** Checks or fallbacks that duplicate a guarantee already made elsewhere.
- **Pattern mimicry.** Code that looks correct because it resembles a familiar idiom, while missing a requirement specific to this package.
- **Inconsistent style.** Naming or structural variation across functions that a single author would not have introduced.
- **Hallucinated functionality.** References to functions, arguments, or packages that do not exist.
- **Outdated patterns.** Deprecated functions or superseded approaches that were correct at some point in the assistant's training data but are not the current recommendation.

Phase 3: Test Suite Review

Coverage numbers are a starting point, not a conclusion:

```
coverage <- covr::package_coverage()
covr::report(coverage)
```

High coverage does not guarantee test quality. For each test file, separately assess whether the assertions check substantive behavior or merely that the code runs without erroring, whether boundary conditions are tested, whether expected failures are verified rather than only successes, whether tests are independent of one another's side effects, and whether a reader can tell what each test is meant to validate.

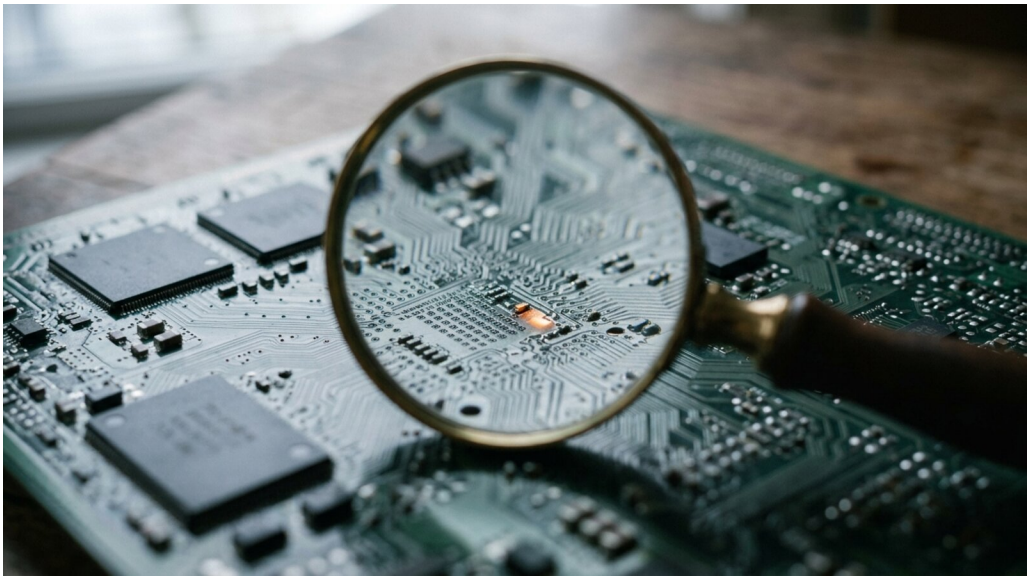
Then check correspondence in both directions: every documented behavior should have a corresponding test, and every test should correspond to a documented or intended behavior. Tests that check conditions no longer relevant to the design are as much a maintenance liability as missing tests.

Phase 4: Integration and System Review

Trace data flow across function boundaries rather than within a single function. Do the functions compose correctly when chained? Are intermediate data structures consistent from one stage to the next? Identify and evaluate any state management: global variables or options, environment modifications, file system operations, external connections. Verify error propagation: are errors caught at the right level, do the messages carry enough context to act on, and is cleanup performed correctly when something fails partway through?

Phase 5: Security and Safety Review

Examine every user-facing input for the standard vulnerability classes: unsanitized input used in file paths, input passed to system commands, query construction, and serialization or deserialization of untrusted data. For network and file system operations, confirm connections are closed, temporary files are cleaned up, credentials are handled appropriately, and URLs are validated before use. Finally, consider the security posture of the dependency chain itself: are the dependencies actively maintained, do any have known vulnerabilities, and is the chain as small as the design allows?



Documentation Review

User-facing documentation and developer documentation get separate passes. The `README` should state the package's purpose clearly, give installation instructions, include a basic usage example, and point to further documentation. Vignettes should build without errors, present coherent workflows with realistic examples, and complement rather than duplicate the function-level documentation. Internal documentation should explain any non-obvious algorithm, record the reasoning behind design decisions, and document the contribution process if the package expects one.

Annotation and Severity

Maintain structured notes during the review rather than relying on memory. A minimal per-function template:

```
## Function: function_name

### Status: [Reviewed | Needs Changes | Approved]

### Understanding
[Summary of what the function does, in your own words]

### Concerns
- [Issue 1]
- [Issue 2]

### Questions
- [Question for further investigation]

### Changes Required
- [ ] Change 1
- [ ] Change 2
```

Classify every finding by severity: **Critical** (incorrect behavior, a security vulnerability, or a data-corruption risk), **Major** (a significant deviation from requirements or poor error handling), **Minor** (style inconsistency, a documentation gap, or a small inefficiency), or **Enhancement** (a suggestion beyond the current requirements). The severity tier determines the order of remediation, not whether an issue gets fixed at all.

Remediation and Final Verification

Work through issues in severity order: critical issues affecting correctness or security first, then major issues affecting usability or maintainability, then minor issues and enhancements as time allows. For each fix, understand the root cause before touching the code, and implement the fix

yourself rather than delegating it back to the assistant without understanding it. Add or modify a test to prevent regression, and update documentation if the observable behavior changed.

After remediation, re-run the full test suite and `R CMD check`, confirm coverage has not decreased, and review the diff in version control before considering the issue closed. Before calling the review complete, run the full verification pass:

```
# Full check
devtools::check()

# Test coverage
covr::package_coverage()

# Additional static analysis
lintr::lint_package()
goodpractice::gp()
```

Then test installation in a clean environment (`remove.packages()` followed by `devtools::install()`) and run every example in the documentation with `devtools::run_examples()`.

Per-Package Review Checklist

The five phases above are the reasoning behind this checklist; the checklist itself is what to actually run through for a given package. Copy it per package rather than trying to hold it in memory across multiple reviews in progress at once.

Phase 1: Structural

- ☐ File structure follows standard R package conventions
- ☐ Every `Imports` entry is necessary, correctly tiered (`Imports` vs `Suggests`), and version-constrained appropriately

- ☐ `NAMESPACE` exports only what is meant to be public

Phase 2: Function-by-function

- ☐ Every exported and internal function reviewed for correctness, design alignment, input validation, documentation quality, and code clarity
- ☐ Happy path and at least one failure path traced for each function, with the test suite confirmed to exercise both
- ☐ Every AI-generated-code pattern in this post's list (over-engineering, defensive redundancy, pattern mimicry, inconsistent style, hallucinated functionality, outdated patterns) checked for, not just read past

Phase 3: Test suite

- ☐ Coverage report reviewed file by file, not just as a percentage
- ☐ A sample of test bodies read to confirm assertions check substantive behavior, not just that the code runs
- ☐ Every documented behavior has a corresponding test, and every test corresponds to a documented or intended behavior

Phase 4: Integration and system

- ☐ Data flow traced across function boundaries, not just within individual functions
- ☐ State management (globals, options, environment modifications, file system, external connections) identified and evaluated
- ☐ Error propagation verified: caught at the right level, with actionable messages and correct cleanup on partial failure

Phase 5: Security and safety

- ☐ Every user-facing input checked against the standard vulnerability classes (unsanitized paths, shell commands, query construction, deserialization)
- ☐ Network and file system operations confirmed to close connections, clean up temporary files, and validate URLs
- ☐ Dependency chain reviewed for maintenance status and known vulnerabilities

Before shipping: ownership sign-off

- ☐ Can explain any function's purpose and implementation without referring to the code
- ☐ Can predict the function's behavior for a novel input
- ☐ Can identify where a new requirement would require a change
- ☐ Can debug an issue without AI assistance
- ☐ Can respond to a user's issue, review an external contribution, and extend the package's functionality unaided

A "no" anywhere in this checklist is a blocker, not a note for later; see Establishing Ownership below for what each unchecked box actually costs if it ships unresolved.

Things to Watch Out For

- **Coverage as a proxy for quality.** A test that asserts nothing meaningful still counts toward the coverage percentage.
- **Delegating the fix back to the assistant.** Understanding the root cause and then asking the assistant to patch it is not the same as understanding the fix; do the remediation yourself.
- **Treating hallucinated functionality as a rare edge case.** It surfaces often enough in assistant-written code that a dedicated existence check for every referenced function and package is worth the time.
- **Reviewing functions in file order instead of dependency order.** Reviewing a caller before its callee means re-reading context twice; work from the bottom of the dependency graph up.
- **Skipping the "why," not just the "what."** A function can be correct and still misaligned with the stated design; design alignment is a separate question from correctness.
- **Conflating "reviewed" with "understood."** The annotation template's "Understanding" field is not decorative; if you cannot fill it in from memory, the function has not actually been reviewed.

Lessons Learned

Conceptual understanding:

- Ownership is a testable claim, not a feeling of familiarity.
- Coverage and correctness are different axes; a review needs both.
- AI-generated code has a distinct failure signature (over-engineering, mimicry, hallucination) that a generic checklist will not catch.

Technical:

- `covr::package_coverage()` plus `covr::report()` gives a navigable, file-by-file coverage view, not just a percentage.
- `goodpractice::gp()` and `lintr::lint_package()` catch a different class of issue than `R CMD check` and are worth running separately.
- A clean-environment install/reinstall test surfaces dependency issues that a `devtools::load_all()` session will never expose.

Gotchas:

- High coverage can mask assertion-free tests; read a sample of test bodies, not just the coverage report.
- `R CMD check` passing does not imply the design was followed; that requires the separate design-alignment pass.
- A function's documentation can describe intended behavior accurately while the implementation has drifted from it; verify both directions.

Limitations

- The framework is scoped to R packages specifically; the phase structure generalizes, but the tooling references (`covr`, `lintr`, `goodpractice`) do not transfer to other languages without substitution.
- It assumes the human author already has enough R fluency to evaluate correctness and design alignment; it is not a tutorial on reading R.
- It does not specify how much reviewer time is proportionate to package size or risk; that judgment is left to the reviewer.
- The ownership test in the final section is self-administered; it provides no external verification that the answers given are accurate.

Opportunities for Improvement

1. A companion checklist scaled down for small, low-risk packages, where the full five-phase process is disproportionate.
2. Worked severity-classification examples, since "Major" versus "Critical" boundary cases are where reviewers disagree most.
3. Guidance on reviewing an AI-assisted *contribution* to an already-owned package, as distinct from reviewing an entire AI-assisted package from scratch.
4. A shared vocabulary for documenting AI involvement in `CITATION.cff` or a `NEWS.md` entry, referenced in this framework's §8.3 but not specified in detail.
5. An adaptation of the framework for packages with compiled code (C/C++/Rust via `cpp11` or extendr), where the security review phase needs additional scrutiny.

Establishing Ownership

Before claiming ownership of the package, verify that you can, without referring to the code: explain any function's purpose and implementation, predict its behavior for a novel input, identify where a new requirement would require a change, and debug an issue without AI assistance. Separately, confirm maintenance readiness: can you respond to a user's issue, review an external contribution, adapt to a breaking change in a dependency, and extend the package's functionality yourself?

Consider also documenting the development process itself: that AI assistance was used, what review process was undertaken, and a clear version-control history showing the remediation work. This is not required by CRAN, but it is honest, and it gives a future maintainer (possibly a future version of yourself) the context this review process assumed from the start.

Review does not end at first release. Any AI-generated change should go through the same review before merging, static analysis should run in continuous integration, and older code sections benefit from periodic re-review as the author's own understanding of the package deepens. Knowledge decays without use: working with the codebase regularly, documenting decisions as they are made, and keeping review notes accessible are what keep the ownership claim true over time, not just true at the moment of first publication.

Wrapping Up

What Did We Learn?

Thorough review of an AI-assisted package requires the same rigor a careful reviewer would bring to any package, plus a specific sensitivity to the failure modes assistant-written code tends to produce. The process demands more than a read-through; it requires achieving genuine understanding of every component before accepting responsibility for it. The five-phase structure (structural, function-by-function, test suite, integration, security) gives that process a checklist rather than leaving it to instinct. The ownership test at the end gives it a stopping condition: review is complete when the answers to the ownership questions are honestly yes, not when the reviewer is tired of reading code.

Main takeaways:

- Coverage is a floor, not a target; read test bodies, not just coverage percentages.
- AI-generated code has a specific set of failure patterns worth a dedicated review pass.
- Ownership is demonstrated by prediction and unaided debugging, not claimed by completing a checklist.

If you are trying this yourself: start with the structural review before touching any function body, keep the annotation template open in a second window rather than trying to hold findings in memory, and do not flip a package to "reviewed" until you can honestly answer every question in the Establishing Ownership section.

See Also

A companion post walks this framework through an applied review of a real package: Exemplar Code Review: The zzobj2fig Package, which uses `zzvim-R` as the review environment and works through structural review, function debugging with `browser()` and `trace()`, and a CRAN-submission checklist that this post does not cover.

Three further companions transpose this same five-phase structure and ownership test onto prose rather than code: Review Methodology for AI-Assisted Blog Post Drafting for blog-length prose, Review Methodology for AI-Assisted Textbook Drafting for book-length, pedagogically structured material, and Review Methodology for AI-Assisted Research Paper Drafting for academic manuscripts, grounded in published journal and publisher AI-disclosure policy.

Key resources:

- R Packages, 2nd ed. — Wickham and Bryan
- Advanced R, 2nd ed. — Wickham
- Writing R Extensions — R Core Team
- CRAN Repository Policy
- rOpenSci Packages: Development, Maintenance, and Peer Review

---

Reproducibility

Source document: `docs/CODE_REVIEW_METHODOLOGY.md`, from the `zzobj2fig` package (`~/prj/sfw/10-zzobj2fig/zzobj2fig/`). This post is a direct translation of that white paper into blog format; no methodological claims were added beyond the source.

Session information:

```
R version 4.6.1 (2026-06-24)
Platform: aarch64-apple-darwin25.4.0
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblasp-r0.3.34.dylib
LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.12.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.6.1  fastmap_1.2.0   cli_3.6.6       tools_4.6.1
 [5] htmltools_0.5.9 parallel_4.6.1  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.58
[13] digest_0.6.39   rlang_1.3.0     evaluate_1.0.5
```

Rendered on 2026-08-18 at 11:00 PDT. Source: ~/prj/rgtlab/posts/rp-code-review-methodology/analysis/report/ind

---

Let's Connect

Questions, corrections, or a different view on where this framework is too strict or too loose are welcome in the comment thread below.

- **GitHub**: rgt47
- **Email**: Contact form

Related posts in this cluster

This post is part of the *R Package Development and Testing* series. Recommended reading order:

1. Updating an R Package: A Complete Development Workflow
2. Writing a Simple Vim Plugin for REPL Interaction
3. Introducing zzvim-R
4. Testing Data Analysis Workflows in R
5. From testthat to tinytest: Converting an R Package Test Suite
6. **Code Review for AI-Assisted R Package Development** (this post)
7. Exemplar Code Review: The zzobj2fig Package