A Review Checklist for AI-Assisted R Packages

Ronald G. Thomas

2026-09-06



A checklist is a claim about what goes wrong. This one was tested against fifteen packages, and it was wrong about several things.

Introduction

A companion post argues that reviewing an AI-assisted R package requires a different kind of attention than reviewing a hand-written one, and it ends with an ownership test rather than a checklist. This post is the operational half: the specific checks worth running, and a report of what happened when they were run against fifteen packages being prepared for CRAN.

The report is the more useful part. Published checklists are almost never tested, so it is rarely possible to tell which items earn their place. This one predicted several failure modes that turned out to be absent, missed a defect class it should have anticipated, and produced one false positive that was acted on before it was verified. All three are documented below, because a checklist whose track record is unknown is a checklist nobody should trust.

Motivations

- **The R-specific gap is real.** Generic checklists for reviewing AI-generated code exist and are language-agnostic. The R package standards that do exist, from CRAN and from rOpenSci, contain no guidance on AI-assisted development at all.
- **Conformance is solved; judgment is not.** `R CMD check`, `pkgcheck`, `lintr`, and `goodpractice` already cover most of what a machine can decide. What remains is exactly where generated code fails.
- **A checklist that has never been run is a hypothesis.** Treating it as one, and reporting the result, is cheaper than discovering the gaps during a CRAN review.

Objectives

1. Give the checks specific to AI-generated R code, each with a detection method rather than an exhortation.
2. Say explicitly what to delegate to existing CRAN resources rather than duplicating them.
3. Report what the checklist caught, and what it did not, across fifteen real packages.
4. Record the checklist's own errors, so that the next revision has something to correct.



What This Does Not Cover

CRAN conformance is out of scope, deliberately. It is well served already, by resources that are maintained and will stay current in a way a blog post will not:

- The CRAN submission checklist, which is short, official, and normative.
- The CRAN *Repository Policy*, which is binding and revised without announcement.

- ThinkR's *Prepare for CRAN*, the most complete community checklist, with a runnable summary script at the top.
- The releasing chapter of *R Packages*, and `usethis::use_release_issue()`, which generates a live checklist as a GitHub issue.

Run those. The checklist below assumes they have passed, and addresses only what they do not look for.

What Is an AI Failure Mode?

A failure mode is a characteristic way a thing breaks. An *AI failure mode*, as the term is used here, is a defect that appears in machine-generated code at a noticeably different rate, or in a different shape, than in code a person wrote by hand.

The distinction matters because review is a search, and a search needs priors about where to look. A person writing R makes off-by-one errors, forgets edge cases, and gets careless near the end of a long function. A language model does not get careless and rarely makes off-by-one errors, but it will produce a call to a function that does not exist, and do so with the same fluency as the surrounding code that does. Reviewing generated code with a checklist built for human error searches in the wrong places.

One mechanism, several symptoms

Generated code is produced by predicting plausible continuations of text. That single fact accounts for most of what follows.

- **Plausibility is not existence.** A function name that *ought* to exist is emitted as though it does. This is hallucinated functionality, and in its supply-chain form, fabricated dependencies.
- **Plausibility reflects the corpus.** Idioms that dominated the training data outrank current ones, which in R means `RCurl`, `rjson`, `XML`, and `sp` appearing in new code. This is the outdated-patterns mode, and it is the one case where the failure has a published list attached: rOpenSci names those packages, and `pkgcheck` flags them by name.
- **Plausibility rewards resemblance.** Code can match the shape of a correct solution while missing the requirement it was meant to satisfy. This is pattern mimicry, and it is the hardest of the group to detect, because the code reads correctly.
- **There is no persistent intent.** A convention established in the third function does not bind the twelfth, which produces scope drift across a long generation session.
- **Producing more is as cheap as producing less.** Unrequested generality and guards that cannot fire accumulate at no cost to the generator and considerable cost to the reader. This is over-engineering and defensive redundancy.
- **Prose and code come from the same process.** A comment or a citation is another plausible continuation, not a claim checked against the code beneath it. Hence comments that describe intent the code does not implement, and citations to papers that do not say what the text claims.

Why they are hard to catch

Three properties, none of which is typical of human error.

They fail silently, or not at all. A hallucinated call sitting in an error branch never executes, so the tests pass and the defect ships. A person's typo usually breaks something visible immediately.

They read well. Fluency is what the generator optimizes, and reviewers skim fluent code. That instinct is exactly backwards here: in generated code, reading well is uninformative about being correct.

The tests share the flaw. When one model writes both the implementation and the tests, both encode the same reading of the requirement. A misreading is present on both sides, so the test confirms it rather than catching it. This is why a green suite from a single generation session is evidence of internal consistency and nothing more, and why the checklist below asks you to write the tests that matter yourself.

The status of this category

These groupings are observational. They come from watching generated code, not from measurement, and no published base rate exists for how often each occurs per thousand generated lines. That should temper how much weight the list carries.

The exercise reported later in this post is a small piece of evidence against the strong version of the claim. Across fifteen packages, the loud modes were the ones that did not appear: no hallucinated functions, no fabricated dependencies, no fabricated citations, no build-time traps. What was actually wrong was quieter and, in one case, not on the list at all. Two readings are available and this exercise cannot separate them: either those packages had already been reviewed and the loud failures caught, or the discourse overstates their frequency.

Either way, the honest framing is that an AI failure mode is a place worth looking, not a defect you should expect to find.

The Checklist

Six groups. Each item states how to detect the problem, because an instruction to "check for X" without a method is not a check.

Each item also carries one of two tags.

VERIFY means an assistant or a tool can do the work and you confirm the result. Twenty-two of the twenty-nine items are of this kind, and you should use a tool for them: a scan over parsed source is faster and more consistent than reading for `system.file()` calls by hand.

YOURS means the item cannot be delegated, and doing it another way is not doing it. There are seven.

The dividing line is not difficulty. It is whether the item asks a question the **code** can answer, or one only the **requirement** can answer. An assistant can read code. It cannot read your intent, and it cannot serve as the independent party when the thing under review is its own output. Verifying

a citation is a case of the second problem: fabricated citations are an AI failure mode, so asking the same class of tool to audit them is asking the suspect to investigate itself.

The tag is a claim about delegability, not about confidence. **VERIFY** results deserve the same suspicion as any other generated artifact. Two of the findings reported later in this post were **VERIFY**-class, were stated as settled, and were wrong.

1. Code that runs at build time

Code in `R/` executes when the package is **built**, often on a machine that is not the user's, and the cached result ships. *R Packages* states the rule directly: any R code outside a function is suspicious and should be carefully reviewed. This belongs at the top of an AI-specific list because top-level code is the signature of a script idiom, and a script idiom is what a model trained on scripts produces.

- ☐ **VERIFY** No `system.file()` call at top level. The path is resolved at build time and will not exist on the user's machine.
- ☐ **VERIFY** No capability probe at top level. A color table or feature check computed on a headless build server reports that result forever, to everyone.
- ☐ **VERIFY** No `foo <- pkgB::blah` aliasing. This freezes the definition as it was on the build machine, so a later bug fix in `blah()` never reaches your users. Write `foo <- function(...) pkgB::blah(...)`.
- ☐ **VERIFY** Remaining top-level code creates objects only, and none of those objects depend on the build environment.

Detection: parse each file in `R/` and list every top-level expression that is not a function definition. Constants, `R6Class()` calls, `new.env()`, and `utils::globalVariables()` are benign; anything that reads the filesystem, probes capabilities, or captures another package's binding is not.

2. Provenance and licensing

This is the group with legal consequences, and the one where AI-assisted development is genuinely exposed. CRAN requires the maintainer to warrant that "the ownership of copyright and intellectual property rights of all components of the package must be clear and unambiguous," and that they act on behalf of all credited authors. That is a claim about the origin of code whose origin the author cannot inspect.

Two facts sharpen it. Stack Overflow answers are licensed CC BY-SA, which is compatible only with GPLv3, and models are trained heavily on Stack Overflow. And a derivative work does not require copied text: *R Packages* notes that re-implementing an algorithm from a source can qualify even when no line is shared.

- ☐ **YOURS** Any generated code resembling an identifiable existing implementation traced and attributed, or rewritten.
- ☐ **VERIFY** License compatible with every dependency, remembering that compatibility is not symmetric: MIT code can be bundled into a GPL package, never the reverse.
- ☐ **VERIFY** Bundled or adapted code attributed, with authors given `ctb` and copyright holders `cph` in `Authors@R`, and `LICENSE.note` present if components carry different licenses.

- ☐ **YOURS** You can honestly sign the maintainer warranty. If you cannot say where a function came from, you cannot warrant that its copyright is unambiguous.

3. Fabricated dependencies

Published measurements put the rate of non-existent package suggestions from large language models at roughly one in five, and attackers register the names that models hallucinate most often, so that a later install of the "missing" dependency retrieves attacker-controlled code. This is a supply-chain vulnerability, not a typo.

- ☐ **VERIFY** Every entry in `Imports` and `Suggests` verified to exist in the registry it claims to come from.
- ☐ **VERIFY** Publisher and registration date plausible for a package of its stated age. A package registered last month that exactly matches the name an assistant suggested is a red flag, not a convenience.
- ☐ **VERIFY** Versions pinned and hashes verified.
- ☐ **VERIFY** Every referenced function confirmed to exist, including in error branches and escaped examples, where a bad reference fails silently because nothing executes it.

4. The test suite is not independent evidence

If the same assistant wrote both the implementation and the tests, they were derived from the same reading of the requirement. Any misreading is present in both, and the test confirms it rather than catching it. A green suite in this situation demonstrates internal consistency, which is worth something, and correctness, which it does not demonstrate at all.

- ☐ **YOURS** At least one test per critical behavior written by you, from the requirement, without reading the implementation. This is the only test in the suite that is actually evidence.
- ☐ **VERIFY** No test asserts on the exact text of an R message. *Writing R Extensions* is explicit that these "change, and they can be translated."
- ☐ **VERIFY** No test asserts the **absence** of a warning, which the same manual calls out as a habit of testthat users specifically.
- ☐ **VERIFY** Skip count read and every skip confirmed as intended. Skips are close to invisible in continuous integration, so a suite can appear to pass while much of it never ran.
- ☐ **VERIFY** Suite passes in randomized order. Order dependence is common in generated suites, because each test is written in the context of the ones before it.
- ☐ **YOURS** Every snapshot read before acceptance. Accepting one because the framework offered to is the coverage fallacy in different clothes: it records what the code does, not what it should do.

5. Numerical correctness

Relevant to any package doing real computation, and the group where a wrong answer is least likely to announce itself.

- ☐ **VERIFY** No floating-point equality comparison; tolerances used and justified.

- ☐ **VERIFY** Numerical tests carry a tolerance. *Writing R Extensions* is blunt: test accuracy only "if you have done a formal error analysis," and checking that probabilities numerically sum to one is "silly." Extended precision differs by CPU, compiler and flags, and arm64 macOS is the platform most often seen to differ.
- ☐ **YOURS** Every numeric literal justified, cited, or parameterized. An unexplained `1e-8` tolerance or iteration cap is a design decision nobody made.
- ☐ **YOURS** Every implemented algorithm traced to a citable source, and the citation verified to exist **and** to describe the method actually implemented.
- ☐ **YOURS** Results compared against a reference implementation where one exists.

6. R semantics generated code gets wrong

- ☐ **VERIFY** Missing-value policy explicit per function; `NaN` and `Inf` handled deliberately rather than incidentally.
- ☐ **VERIFY** Reference-semantics objects (`data.table`, `R6`, environments) not mutated through arguments. Generated code assumes copy-on-modify throughout, which is false for these.
- ☐ **VERIFY** S3 methods for foreign generics registered with `@exportS3Method pkg::generic`. Plain `@export` silently fails, and the result works under `load_all()` and breaks once installed, which is the one configuration interactive development never exercises.
- ☐ **VERIFY** Every use of a `Suggests` package below `R/` guarded with `requireNamespace()` or `rlang::check_installed()`.
- ☐ **VERIFY** Base-package functions declared. `stats` and `utils` are attached in your session and will resolve silently while remaining undeclared.
- ☐ **VERIFY** User state restored: `options()`, `par()`, `setwd()`, RNG. Prefer `withr::local_*()`, which captures and restores for you.



What It Caught

The checklist above was run against fifteen R packages under active development, all built with the same scaffolding and all headed for CRAN. Fourteen reached a clean `R CMD check --as-cran`. The fifteenth was a scaffold rather than a package: one file below `R/`, no exports, an empty `man/`.

Two caveats on that number before anything is read into it. Every check ran on one platform, macOS arm64 under R 4.6.1, so it says nothing about Windows, Linux, or R-devel. And it was reached only after fixing the defects described below; the first pass was considerably messier.

The findings are more interesting than the pass rate.

The predicted failure mode was absent

Section 1 above is the item I was most confident about, and it found almost nothing. Across fifteen packages, top-level code in `R/` existed but was uniformly benign: `R6Class()` definitions, `utils::globalVariables()` calls, constant vectors, and `new.env(parent = emptyenv())`. There was no `system.file()` at top level, no capability probe, and no `pkg::fun` aliasing anywhere. Eight packages had no top-level code at all.

The same held for the failure modes that get the most attention in writing about AI-generated code. No fabricated dependency. No hallucinated function. No fabricated citation. The checklist predicted these; the packages did not have them.

That is a real result and it deserves stating plainly rather than buried: **the failure modes most discussed in the literature were the ones least present in practice.** One reasonable explanation is that these packages had already been through the review process the companion post describes, so the loud failures were caught earlier. Another is that the base rates are simply lower than the discourse suggests. This exercise cannot distinguish the two.

What was actually wrong

The defects that surfaced were quieter and mostly mechanical.

An unescaped `%` cascading into three unrelated-looking symptoms. One package had `\code{"80%"}` and `dx: 100%` inside roxygen comments. In Rd, `%` begins a comment and swallows the rest of the line, which was consuming closing braces. This produced three separate reports that looked unconnected: roxygen warning about mismatched braces, an undocumented `width` argument in one Rd file, and a "Dropping empty section \details" note in another. Escaping them as `\%` fixed all three at once. Nothing in the AI-failure-mode literature predicts this, and no item on my checklist covered it.

Documentation drift with a timestamp. Another package had a codoc mismatch: the documentation described `data = NULL` while the function took `strata_var` and `strata_val`. The file times settle the question. The Rd was generated at 16:06 and the source edited at 16:29, and `roxygenise()` was never re-run. `R CMD check` caught it only because the *signature* changed. Had the semantics changed while the arguments stayed put, nothing would have flagged it, which is the case the companion post's function-by-function review exists to catch.

A 42MB vendored library shipping in the tarball. One package carried a `.cache/R/renv/...` tree for a Linux platform, gitignored but not listed in `.Rbuildignore`, producing a 21MB source tarball against CRAN's 10MB limit. One line fixed it; the tarball fell to 1.0MB.

Undeclared base-package functions. Three packages used `stats` or `utils` functions with neither an `Imports` entry nor a namespace import. These resolve silently in an interactive session because those packages are attached, which is exactly why they survive.

Documentation that was never generated. One package had complete roxygen comments and an empty `man/` directory. `roxygenise()` had simply never been run, so sixteen exported functions were undocumented.

The rest were metadata: development version numbers still at `0.0.0.9000`, a changelog sentence sitting in the `Description` field, a stale `Date`, a `VignetteBuilder` declared with no vignettes to build, and a 71KB development roadmap shipping in a tarball.

A false positive, and acting on it too early

One item on the checklist was wrong, and worth recording because of how it failed.

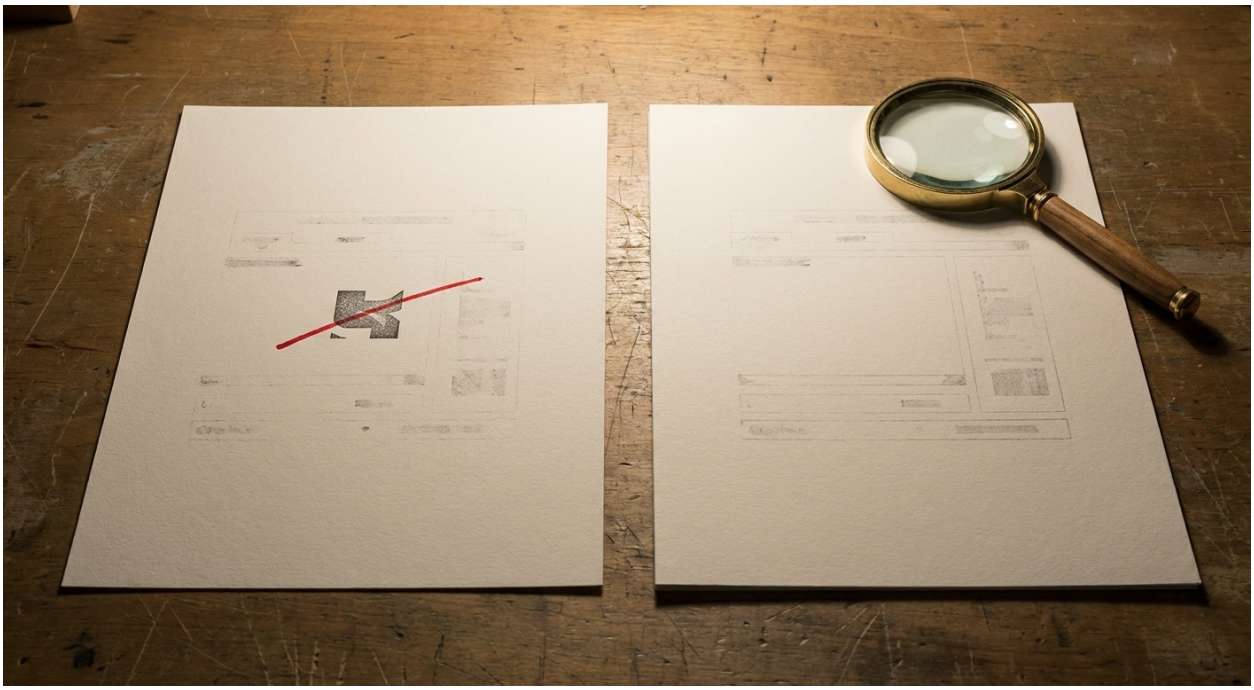
To find exported functions lacking a `@return` tag, I compared the count of `@export` tags to the count of `@return` tags per package. That heuristic reported one package missing roughly fifteen return values. It is wrong: functions sharing a help topic through `@rdname`, or inheriting through `@inherit`, are documented once for the group, so the counts do not correspond.

The failure that mattered was not the bad heuristic but the order of operations. I acted on it, adding `@return` tags to ten `print()` methods, and only then checked the generated `Rd` files, which is where the answer actually lives. Every package had complete `\value` sections already. The ten edits were reverted.

The correct check is one line and operates on the output rather than the input:

```
# Rd topics lacking a \value section
rds <- list.files("man", pattern = "[.]Rd$", full.names = TRUE)
Filter(function(p) {
  txt <- paste(readLines(p, warn = FALSE), collapse = "\n")
  !grepl("\\\\docType\\{(package|data)\\}", txt) &&
    !grepl("\\\\value\\{", txt)
}, rds)
```

The general lesson generalizes past this instance: **verify against the artifact the requirement is about.** CRAN reads `Rd` files, so the check belongs on `Rd` files. A proxy measured on the source is a hypothesis about the artifact, and hypotheses should be confirmed before they are acted on.



Lessons Learned

Conceptual:

- The loud failure modes are the ones that get written about, and in this sample they were the ones that did not occur. Quiet, mechanical defects dominated.
- Several findings had one cause and several symptoms. Reading the check output as a list of independent problems would have produced three fixes where one was needed.
- Documentation drift is the failure mode this exercise supports most strongly, and the only one where the evidence was unambiguous.

Technical:

- Check the built tarball, not the source directory. Several findings concerned what `.Rbuildignore` did and did not exclude, which is invisible until the tarball exists.
- `R CMD build --no-build-vignettes` produces vignette warnings that are artifacts of the flag. I generated ten of them and briefly believed they were real.
- `_R_CHECK_CRAN_INCOMING_REMOTE_=false` avoids a network dependency in the incoming-feasibility check, which otherwise fails when CRAN's server is slow.

Gotchas:

- A clean `R CMD check` says nothing about whether the implementation matches the design. Fourteen packages reached `Status: OK` without a single line of their code being read.

Limitations

- Fifteen packages by one author using one scaffold is not a sample anyone should generalize from. The absence of fabricated dependencies here is evidence about these packages, not about AI-generated R code.
- All checks ran on macOS arm64 under one R version. No Windows, no Linux, no R-devel. For the numerical items in particular this is the wrong platform to draw conclusions from.
- Nothing in this exercise tested correctness. Every finding above is mechanical. The judgment items on the checklist, which are the ones that matter most, were not exercised at all.
- The checklist has now been run once. One run tells you which items fire, not which items are worth keeping.

Wrapping Up

What Did We Learn?

A checklist is a claim about what goes wrong, and claims can be checked. This one was right that AI-assisted packages need attention beyond `R CMD check`, and wrong about where that attention was most needed. The failure modes it emphasized were absent; the defects it found were quieter and, in one case, entirely unanticipated.

Main takeaways:

- Delegate CRAN conformance to resources that are maintained. Spend your own effort on what they do not cover.
- Verify against the artifact the requirement concerns, not a proxy for it, and confirm before acting.
- A clean check is a floor. Fourteen packages passed without anyone reading their code.
- Seven of twenty-nine items cannot be delegated. Tag them, so that the small irreducible set is visible rather than scattered.
- Report what your checklist missed. An untested checklist is a hypothesis wearing the costume of a standard.

If you are trying this yourself: run the automated tools first and clear them completely, because their output is noisy enough to hide the judgment items underneath. Then work the six groups above. Then read the code, which none of this substitutes for.

See Also

The argument behind this checklist, including the five review phases and the ownership test, is in Code Review for AI-Assisted R Package Development. An applied walkthrough on a single package is in Exemplar Code Review: The zzobj2fig Package.

Key resources:

- CRAN submission checklist

- CRAN Repository Policy
- Writing R Extensions
- R Packages, 2nd ed. — Wickham and Bryan
- Prepare for CRAN — ThinkR
- rOpenSci Packages: Development, Maintenance, and Peer Review
- pkgcheck

Reproducibility

Sources. Checklist items are drawn from *Writing R Extensions* (R 4.6.x), the CRAN *Repository Policy*, *R Packages* 2nd ed., and the rOpenSci development guide, each read from source rather than from summaries, and cited inline where a claim is theirs.

The empirical section reports a run against fifteen packages on 2026-09-05, on macOS arm64 under R 4.6.1, using `R CMD build` followed by `R CMD check --as-cran` on the resulting tarball. Counts and findings are from those check logs.

AI involvement in this post: drafting was AI-assisted, and the package checks were run and interpreted with assistance. The false positive described above was mine in that process, and is reported because a post arguing for verification before action should say when that order was not followed. The framework, the argument, and the editorial judgment are the author's.

Session information:

```
R version 4.6.2 (2026-10-29)
Platform: aarch64-apple-darwin23
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRblas.0.dylib
LAPACK: /Library/Frameworks/R.framework/Versions/4.6/Resources/lib/libRlapack.dylib;  LAPACK version 3.12.

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_4.6.2  fastmap_1.2.0   cli_3.6.6       tools_4.6.2
 [5] htmltools_0.5.9 parallel_4.6.2  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.62
[13] digest_0.6.39   rlang_1.2.0     evaluate_1.0.5
```

Rendered on 2026-11-23 at 17:52 PST. Source: ~/prj/ryyblog/posts/rp-ai-package-review-checklist/analysis/report/index.qmd

---

Let's Connect

Corrections are particularly welcome here, since the value of this post depends on the findings being accurate.

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