Exemplar Code Review: The zzobj2fig Package

Ronald G. Thomas

2026-08-18

A companion piece to Code Review for AI-Assisted R Package Development: the same five phases, applied to a real package with a real editor open.

Introduction

The companion post on this site lays out a five-phase framework for reviewing R packages built through human-AI collaboration. A framework read in the abstract is easy to nod along with and hard to actually run. This post applies it, phase by phase, to `zzobj2fig`, a real package that converts data frames and statistical objects to publication-quality LaTeX tables with automatic PDF generation and cropping. The package was designed by a human author and implemented by an AI assistant (Claude). It exports 52 functions, includes 29 S3 methods, and targets academic researchers who need journal-ready tables.

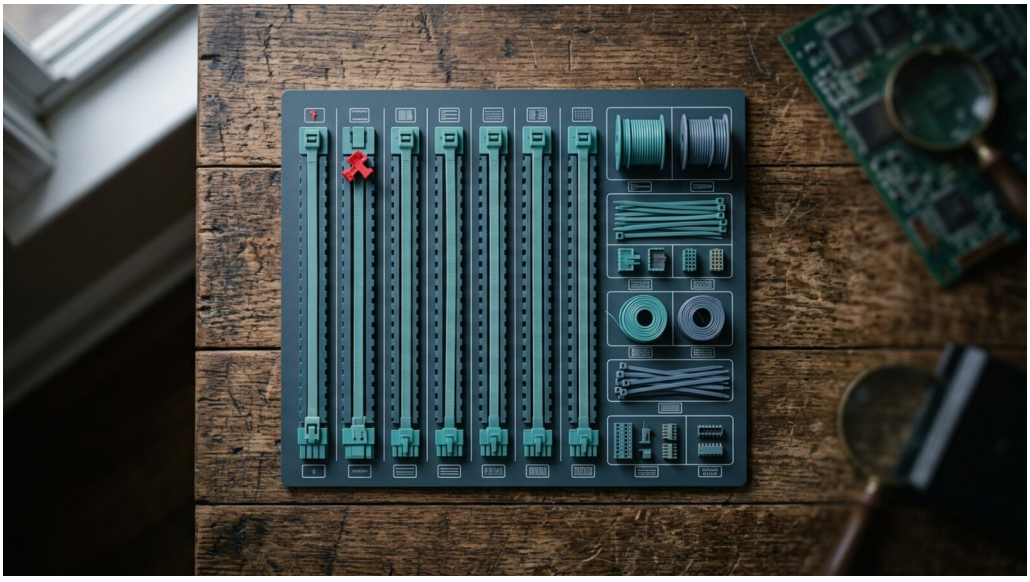
The review environment throughout is `zzvim-R`, a Neovim/Vim plugin for R development that provides side-by-side source and terminal panes, smart code submission, object inspection shortcuts, and chunk navigation for R Markdown and Quarto files. None of the review methodology depends on this specific tool, but the concrete commands below assume it, since a review walkthrough is more useful with actual keystrokes than with "open your editor of choice."

Motivations

- **A framework needs a worked example.** Abstract review criteria ("check design alignment") do not specify what checking looks like in a terminal.
- **Debugging tools belong in a review, not just a bug fix.** `browser()`, `trace()`, and `debug()` are review instruments as much as debugging instruments; this post treats them that way.
- **CRAN submission is downstream of review, not separate from it.** A review that stops at "the code looks fine" and does not verify a clean-environment install has not actually finished.

Objectives

1. Walk through structural review (`DESCRIPTION`, `NAMESPACE`, dependency audit) using a real package's actual metadata.
2. Demonstrate function-level review with interactive debugging (`trace()`, `debug()`, `browser()`) on `zzobj2fig`'s core internal function.
3. Apply the AI-generated-code-pattern checklist concretely, with `vimgrep` and `grep` commands that surface each pattern.
4. Carry the review through remediation, final verification, and a CRAN pre-submission checklist.



Prerequisites

Before starting, establish a review workspace separate from the package's own working tree, so review notes survive a `git clean` or a branch switch:

```
mkdir -p ~/code-reviews/zzobj2fig
cd ~/code-reviews/zzobj2fig
mkdir notes test-outputs findings debug-sessions
touch notes/REVIEW_LOG.md
```

`REVIEW_LOG.md` tracks phase status and a running findings-severity count:

```
# zzobj2fig Code Review Log

**Reviewer:** [Your name]
**Start Date:** [Date]
**Package Version:** 0.2.0
```

Phase	Status
Pre-Review Setup	Not Started
Structural Review	Not Started
Function Review	Not Started
Test Suite Review	Not Started
Integration Review	Not Started
Security Review	Not Started
Documentation Review	Not Started
Remediation	Not Started
Final Verification	Not Started
CRAN Preparation	Not Started

Then set up the editor and load the package under review:

```
cd ~/prj/d01/zzobj2fig
git checkout -b code-review-$(date +%Y%m%d)
vim .
```

```
:e R/obj2fig.R
" Launch an R terminal: <LocalLeader>w (vertical) or
" <LocalLeader>W (horizontal), or <LocalLeader>rr for host R with renv
```

```
library(devtools)
library(testthat)
devtools::load_all()
# zzobj2fig 0.2.0 - LaTeX table generation for R
```

Run initial diagnostics and record everything in the review log:

```
devtools::document()
check_results <- devtools::check()
test_results <- devtools::test()
if (requireNamespace("covr", quietly = TRUE)) {
  coverage <- covr::package_coverage()
  print(coverage)
}
```

For `zzobj2fig`, the reported metrics were approximately 13 R source files totalling 5,300 lines of code, 52 exported functions plus 29 S3 methods, and 7 test files totalling 1,200 lines of tests.

Structural Review

DESCRIPTION

```
Package: zzobj2fig
Title: Generate LaTeX Tables and PDF Outputs
Version: 0.2.0
Description: A package to create LaTeX tables from dataframes and
    statistical objects with optional styling and generate cropped PDF
    figures. Supports multiple output formats, themes for journal styling
    (APA, Nature, NEJM), S3 methods for lm/glm objects, and R Markdown
    integration via custom knitr engine.
License: GPL (>= 3) + file LICENSE
```

The review questions here are mechanical but easy to skip under time pressure: is the title in title case, is the description a complete sentence ending with a period, are all authors specified with roles, does the `LICENSE` file actually exist, and are the R version requirements reasonable.

NAMESPACE and dependency audit

`zzobj2fig` exports a main API (`o2f`, `o2f_batch`, `o2f_inline`), theme functions (`o2f_theme`, `o2f_theme_set`, `o2f_theme_get`), advanced features (`o2f_footnote`, `o2f_header_above`, `o2f_collapse_rows`), and 29 S3 methods covering `lm`, `glm`, `anova`, `htest`, survival models (`coxph`, `survfit`, `survdiff`), and mixed models (`lmerMod`, `glmerMod`, `lme`).

For each `Imports` entry, check actual usage before accepting the dependency:

```
imports <- c("kableExtra", "stats", "utils")
for (pkg in imports) {
  pattern <- paste0(pkg, "::")
  matches <- system(paste0("grep -r '", pattern, "' R/"), intern = TRUE)
  cat("\n", pkg, "usage:\n")
  cat(matches, sep = "\n")
}
```

All three imports checked out as essential: `kableExtra` for core table generation, `stats` for `coef`/`confint`/`nobs` in the model methods, `utils` for `methods()`. `Suggests` entries were audited the same way, checking for conditional use via `requireNamespace()`.

Function-by-Function Review

Build a function inventory before reviewing anything, and review in dependency order rather than file order, since reviewing a caller before its callee means re-reading the same context twice:

```
r_files <- list.files("R", pattern = "\\.R$", full.names = TRUE)
functions <- list()
for (file in r_files) {
  content <- readLines(file)
  fn_lines <- grep("^[a-zA-Z_\\.][a-zA-Z0-9_\\.]*\\s*<-\\s*function", content)
  for (line_num in fn_lines) {
    fn_name <- sub("\\s*<-.*", "", content[line_num])
    functions[[fn_name]] <- list(file = file, line = line_num)
  }
}
cat("Functions to review:", length(functions), "\n")
```

For `zzobj2fig`, the review order followed the dependency graph: `zzz.R` (package initialization) first, then `obj2fig.R` (core internal implementation), `s3-methods.R`, `themes.R`, `advanced-features.R`, `broom-methods.R`, `formatting.R`, `inline.R`, `latex-include.R`, `batch.R`, `caching.R`, `output-formats.R`, and `knitr-engine.R` last.

Debugging the core function

`o2f_internal()` (`obj2fig.R:52-239`) is the function everything else calls into, so it gets the most scrutiny. Rather than reading the implementation statically, trace it interactively:

```
# Insert a breakpoint without modifying the source
trace("o2f_internal", tracer = browser, where = asNamespace("zzobj2fig"))

test_df <- data.frame(x = 1:3, y = c("a", "b", "c"))
o2f(test_df, "debug_test", sub_dir = tempdir())

# In the browser: n (next line), s (step into), c (continue), Q (quit)

untrace("o2f_internal", where = asNamespace("zzobj2fig"))
```

Stepping through in this way, four regions of the function got separate attention: input validation (lines 69-127, checked for `call. = FALSE` on every `stop()`, which was used consistently), theme resolution (lines 129-143), directory creation (lines 159-174), and the LaTeX compilation step (lines

223-225), where stepping into `compile_latex()` with `s` surfaced the actual system call being constructed.

`debug()` and `debugonce()` cover the cases `trace()` does not fit as well: `debug(zzobj2fig:::compile_latex)` for a function you expect to step through repeatedly across several test calls, and `debugonce(zzobj2fig:::sanitize_column_names)` when you want exactly one interactive pass and no more.

S3 dispatch review

For the `o2f` generic, confirm the dispatch table before debugging any single method:

```
o2f <- function(x, ...) {
  UseMethod("o2f")
}
methods(o2f)

debug(zzobj2fig:::o2f.lm)
model <- lm(mpg ~ cyl + hp, data = mtcars)
o2f(model, "test_lm", sub_dir = tempdir())
# In browser: print(x), str(s), print(coef_df)
undebug(zzobj2fig:::o2f.lm)
```

Manual verification without the debugger is worth doing too, since it checks the assumption the code depends on rather than just the code itself:

```
model <- lm(mpg ~ cyl + hp, data = mtcars)
s <- summary(model)
coef_df <- as.data.frame(s$coefficients)
colnames(coef_df)
# [1] "Estimate"   "Std. Error" "t value"    "Pr(>|t|)"
```

That last line matters: `o2f.lm` assumes `summary.lm()`'s coefficient matrix has exactly those column names, and that assumption is only safe because it was checked, not because it seemed likely.

AI-generated code patterns, operationalized

The companion post's pattern checklist becomes concrete searches:

```
" Over-engineering: nested function chains
:vimgrep /function.*function.*function/j R/*.R
```

```
# Defensive redundancy: repeated validation
system("grep -n 'is.data.frame' R/*.R", intern = TRUE)
```

```
" Inconsistent style: mixed assignment or pipe operators
:vimgrep /[^<!=]=[^=]/j R/*.R
:vimgrep /%>%/j R/*.R
:vimgrep /|>/j R/*.R
```

```
# Hallucinated functions: verify every referenced function exists
exists("kable", where = "package:kableExtra")
exists("row_spec", where = "package:kableExtra")
tryCatch(
  zzobj2fig:::nonexistent_function(),
  error = function(e) cat("Function not found:", e$message, "\n")
)
```

Test Suite Review

Run coverage first, then read a sample of test bodies rather than trusting the percentage alone:

```
library(covr)
coverage <- package_coverage()
print(coverage)
report(coverage)
```

A representative test from `zzobj2fig`'s suite:

```
test_that("o2f handles basic dataframe conversion correctly", {
  skip_if_no_latex()
  skip_if_not(system("pdfcrop -version") == 0, "pdfcrop not available")
  dir.create("test_output", showWarnings = FALSE)
```

```
  on.exit(unlink("test_output", recursive = TRUE))

  test_df <- data.frame(col1 = 1:3, col2 = letters[1:3])
  output_file <- o2f(test_df, "test_table", sub_dir = "test_output")

  expect_true(file.exists("test_output/test_table.tex"))
  expect_true(file.exists("test_output/test_table.pdf"))
})
```

Test quality assessment for this file: it skips gracefully when `xelatex`/`pdfcrop` are unavailable, creates an isolated output directory and cleans it up with `on.exit()`, and checks actual file creation rather than merely that the call did not error. What it does not do, and what a follow-up pass flagged as an opportunity, is verify file *contents* or confirm the generated PDF is valid, not just present.

To find gaps systematically rather than by inspection alone, cross the export list against the test file contents:

```
exports <- getNamespaceExports("zzobj2fig")
test_files <- list.files("tests/testthat", pattern = "^test-.*\\.R$",
                         full.names = TRUE)
test_content <- unlist(lapply(test_files, readLines))
untested <- character(0)
for (fn in exports) {
  if (!any(grepl(fn, test_content, fixed = TRUE))) {
    untested <- c(untested, fn)
  }
}
cat("Potentially untested functions:\n")
cat(untested, sep = "\n")
```

Integration and Security Review

Tracing data flow

```
trace("o2f", tracer = quote(cat(">>> Entering o2f\n")),
      where = asNamespace("zzobj2fig"))
trace("o2f_internal", tracer = quote(cat(">>> Entering o2f_internal\n")),
      where = asNamespace("zzobj2fig"))

test_df <- data.frame(x = 1:3, y = c("a", "b", "c"))
o2f(test_df, "trace_test", sub_dir = tempdir())

untrace("o2f", where = asNamespace("zzobj2fig"))
untrace("o2f_internal", where = asNamespace("zzobj2fig"))
```

The resulting call chain: `o2f()` (S3 generic) dispatches to `o2f.data.frame()` or another method, which calls `o2f_internal()`, which in turn calls `sanitize_column_names()`, `sanitize_table_cells()`, and `create_latex_table()` (writing the `.tex` file via `kableExtra::kable()`), then `compile_latex()` (`system("xelatex ...")`) and `crop_pdf()` (`system("pdfcrop ...")`).

System command review

Every `system()` call is a potential injection point, so each one gets an individual verdict rather than a blanket pass:

```
:vimgrep /system\s*(/j R/*.R
:copen
```

For `zzobj2fig`'s two `system()` call sites: `compile_latex()` uses `shQuote()` on every path argument, which prevents shell injection, and `basename()` on the target, which removes directory traversal; the command itself is fixed (`xelatex`, not user-controlled). Same pattern in `crop_pdf()`: quoted paths, a numeric (pre-validated) margin, and a fixed command structure. Verdict: system command usage is safe.

A final sweep for credential or secret handling turned up nothing to flag: the package works with statistical output, not with credentials, so this check was fast, but skipping it is exactly how a package accumulates an unreviewed dependency on an environment variable no one documented.

Documentation Review

```
library(zzobj2fig)
?o2f
help(package = "zzobj2fig")
args(o2f.default)
```

Per-function checklist: title concise, description explains purpose, every parameter has `@param`, return value has `@return`, examples run (wrapped in `\dontrun{}` if they require a working LaTeX installation), and related functions are cross-linked with `@seealso`. Vignettes get a build check (`devtools::build_vignettes()`) in addition to a read-through, since a vignette that reads well but does not build is worse than no vignette at all.

Remediation

Findings sort into the same four tiers as the companion post's framework: critical (security, crashes, incorrect calculations), major (significant behavioral deviation, missing validation, confusing errors), minor (style, documentation gaps, small inefficiencies), and enhancement (feature suggestions, performance, API polish).

A representative fix cycle for `zzobj2fig`, working entirely inside the editor and terminal split:

```
:e R/s3-methods.R
:173
```

```
devtools::load_all()
debugonce(zzobj2fig:::o2f.lm)
o2f(model, digits = 3)
```

```
test_that("o2f.lm validates digits parameter", {
  model <- lm(mpg ~ cyl, data = mtcars)
  expect_error(o2f(model, digits = -1), "non-negative")
  expect_error(o2f(model, digits = "three"), "numeric")
})
```

```
devtools::test()
```

The pattern generalizes: reproduce interactively, fix the source, add a test that would have caught the original defect, then confirm the whole suite still passes, not just the new test.

Final Verification and CRAN Submission

```
test_results <- devtools::test()
if (any(test_results$failed > 0)) {
  stop("Tests failed! Fix before proceeding.")
}
check_results <- devtools::check()
print(check_results)
```

CRAN submission needs zero errors and zero warnings; notes should be either eliminated or individually justified. A clean-environment install test catches what an in-session `devtools::load_all()` never will:

```
devtools::build()
install.packages("../zzobj2fig_0.2.0.tar.gz", repos = NULL, type = "source")
library(zzobj2fig)
check_latex_deps()
test_df <- data.frame(x = 1:3, y = letters[1:3])
o2f(test_df, "test", sub_dir = tempdir())
```

The pre-submission checklist for `zzobj2fig` covered package metadata (version, description as a single paragraph, valid license and maintainer email), code quality (`R CMD check` clean, no undocumented functions, no missing imports or exports, examples running without long-running calls left un-wrapped), system dependencies (`SystemRequirements` documented, graceful failure when LaTeX is missing), documentation (`README`, `NEWS.md`, vignettes building), and testing (platform coverage, graceful skips when dependencies are absent).

Establishing Ownership

Before claiming ownership of `zzobj2fig`, the review closed with a set of questions answerable without referring back to the source. The architecture questions: what is the main entry point, how does S3 dispatch work, where is global state stored, what system dependencies are required. The implementation questions: how are special LaTeX characters handled, what happens when compilation fails, how does the theme system resolve names to objects, how are footnotes implemented. The debugging questions: where would you look for wrong table formatting, how would you diagnose a LaTeX compilation error, what would cause `o2f` to return silently without creating files.

Demonstrating debugging proficiency without AI assistance is the concrete test:

```
test_df <- data.frame(
  name = c("Test & Co.", "100% Complete", "Price: $50"),
  value = 1:3
)
trace("sanitize_table_cells", tracer = browser,
      where = asNamespace("zzobj2fig"))
o2f(test_df, "special_chars", sub_dir = tempdir())
# In browser: what input does the function receive, what
# transformations are applied, what output is produced?
untrace("sanitize_table_cells", where = asNamespace("zzobj2fig"))
```

And writing original code that uses the package, rather than only reading it, is what actually proves the understanding claimed above:

```r
my_journal_theme <- o2f_theme(
  name = "my_journal",
  scolor = "gray!5",
  header_bold = TRUE,
  font_size = "small",
  striped = TRUE
)
o2f_theme_register(my_journal_theme)
o2f_theme_set("my_journal")

model <- lm(mpg ~ wt + hp + cyl, data = mtcars)
o2f(model, "my_regression", digits = 2,
    caption = "Regression Results", label = "tab:regression")
```

Things to Watch Out For

- **Reviewing by file order instead of dependency order** doubles the re-reading; start at `zzz.R` and work outward along the call graph.
- **Trusting `trace()` output without `untrace()`-ing afterward.** A forgotten trace silently slows every subsequent call and produces console noise that looks like a new bug.
- **Treating "the test passes" as "the behavior is verified."** The `o2f` file-existence test above passes even if the generated LaTeX is malformed; content verification is a separate, unaddressed gap.
- **Skipping the security sweep on packages that "obviously" have no sensitive data.** The check took minutes; skipping it is how an undocumented credential dependency goes unnoticed.
- **CRAN's "0 warnings" bar creeping into "0 warnings, some notes I never actually justified."** An unjustified note is a deferred warning.
- **Confusing debugging (finding a fix) with review (verifying the whole function).** `debugonce()` on one input tells you about that input; the systematic function-review pass is still required.

Lessons Learned

Conceptual:

- Applying an abstract review framework to a real package surfaces edge cases (S3 dispatch verification, LaTeX-specific system-call review) that a generic checklist would not anticipate.
- A five-phase structure holds up under a 52-function, 29-method package without needing modification, which is some evidence the phase boundaries are drawn in the right places.

Technical:

- `trace(fun, tracer = browser, where = asNamespace(pkg))` is a non-invasive way to step through an internal function without editing the source.

- `getNamespaceExports()` plus a grep across test file contents is a fast, if approximate, way to find untested exports.
- `shQuote()` plus `basename()` on every path argument is the concrete pattern that makes a `system()` call defensible in a security review, not just an assertion that it is "probably fine."

Gotchas:

- File-existence tests can pass while content is wrong; this gap should be closed before the next release, not carried forward indefinitely.
- `debug()` persists across calls until explicitly removed with `undebug()`; forgetting to remove it mid-review is a common self-inflicted confusion.
- A package with zero sensitive-data handling still needs the security pass; "obviously safe" is a conclusion the pass should produce, not a reason to skip it.

Limitations

- This walkthrough covers one package's review; a package with compiled code, non-standard evaluation, or heavier external I/O would surface phases this post does not exercise.
- The `zzvim-R` commands shown are specific to that plugin; the underlying R debugging calls (`trace`, `debug`, `browser`) are editor-agnostic, but the keystroke workflow is not.
- Coverage numbers, pass/fail results, and the "system command usage is safe" verdict are carried over from the source document as reported there; this post does not independently re-run `zzobj2fig`'s test suite or re-verify those figures.

Opportunities for Improvement

1. A follow-up pass that verifies LaTeX table *content*, not just file existence, closing the gap the test-review section identified.
2. A parallel walkthrough using a non-Vim editor's debugging integration, to separate the review methodology from the `zzvim-R`-specific keystrokes.
3. A worked example of the "companion post's" ownership test actually failing for a function, to show what an honest "not yet ready to ship" verdict looks like in practice.
4. Extending the security-review pass with a worked compiled-code example, since `zzobj2fig` has none to exercise that branch of the checklist.

Wrapping Up

What Did We Learn?

The companion methodology post argues that review requires genuine understanding, not a read-through. This walkthrough is the evidence for that claim in a specific case: understanding `o2f_internal()` required stepping through it with `browser()`, not reading it top to bottom; understanding the test suite's actual coverage required reading test bodies, not the coverage percentage;

and understanding the security posture of two `system()` calls required checking the quoting discipline at each call site individually, not trusting that "the package looks careful" as a whole. None of that substitutes for the framework. It is what the framework looks like when it is actually run against a package with 52 exported functions and a LaTeX compilation step that could execute arbitrary shell commands if the quoting were wrong.

Main takeaways:

- Interactive debugging (`trace`, `debug`, `browser`) is a review tool, not just a bug-fix tool.
- A single `system()` call needs its own individual security verdict; "the package is fine" is not a verdict.
- File-existence tests and content-correctness tests check different things, and a review that only checks the former has an unclosed gap worth naming explicitly, not silently.

If you are trying this yourself: start with the function inventory script before opening any single file, and budget real time for the CRAN pre-submission checklist. It is the part most often compressed under deadline pressure, and it is also the part a reviewer other than the author will actually see.

See Also

This post applies the framework from Code Review for AI-Assisted R Package Development, which covers the five-phase structure, the AI-pattern checklist, and the ownership test in the abstract. Read that post first if the phase-by-phase reasoning here is unfamiliar.

Key resources:

- zzvim-R — the Neovim/Vim plugin used as the review environment throughout
- Advanced R, 2nd ed. — debugging tools chapter
- CRAN Repository Policy
- testthat documentation

Reproducibility

Source document: `docs/EXEMPLAR_CODE_REVIEW_ZZTAB2FIG.md`, from the `zzobj2fig` package (`~/prj/sfw/10-zzobj2fig/zzobj2fig/`). This post condenses that guide's 10-part walkthrough; code excerpts and the `zzvim-R` command reference are reproduced from the source. Package metrics, coverage figures, and review verdicts reported above are carried over from the source document and have not been independently re-verified for this post.

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-exemplar/analysis/report/index.

Let's Connect

Questions, corrections, or a report of where this walkthrough's specifics no longer match a current `zzobj2fig` release 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
7. **Exemplar Code Review: The zzobj2fig Package** (this post)