EXEMPLAR_CODE_REVIEW_ZZTAB2FIG

January 03, 2026 at 10:05 AM

Exemplar Code Review: zztab2fig Package

A Step-by-Step Guide for New Developers Using zzvim-R

Introduction

This document provides a detailed, hands-on walkthrough of conducting a code review for the `zztab2fig` R package using **zzvim-R** as your integrated development environment. The package was designed by a human author and implemented by an AI coding assistant (Claude). This guide assumes no prior experience with formal code review and walks through every step from initial setup to CRAN submission.

The `zztab2fig` package converts data frames and statistical objects to publication-quality LaTeX tables with automatic PDF generation and cropping. It exports 52 functions, includes 29 S3 methods, and targets academic researchers who need journal-ready tables.

Why zzvim-R for Code Review?

zzvim-R provides an efficient environment for R package code review:

- **Side-by-side editing**: View source code and R terminal simultaneously
- **Smart code execution**: Send functions, blocks, or selections to R with `<CR>`
- **Object inspection**: Quickly examine objects with `<LocalLeader>h/s/d/p`
- **Workspace monitoring**: Track memory usage and loaded objects with HUD
- **Chunk navigation**: Move through R Markdown files efficiently
- **Debugging integration**: Set breakpoints and step through code interactively

zzvim-R Key Mappings Quick Reference

Before starting, familiarize yourself with these essential mappings:

Key	Action
`<CR>`	Smart code submission (context-aware)
`<LocalLeader>r`	Launch R in Docker container
`<LocalLeader>rr`	Launch host R with renv

Key	Action
`<LocalLeader>rh`	Launch host R vanilla
`<LocalLeader>w`	Open R terminal (vertical split)
`<LocalLeader>W`	Open R terminal (horizontal split)
`<LocalLeader>h`	`head()` on word under cursor
`<LocalLeader>s`	`str()` on word under cursor
`<LocalLeader>d`	`dim()` on word under cursor
`<LocalLeader>p`	`print()` on word under cursor
`<LocalLeader>n`	`names()` on word under cursor
`<LocalLeader>g`	`glimpse()` on word under cursor
`<LocalLeader>0`	Open workspace HUD (5 tabs)
`<LocalLeader>m`	Memory usage HUD
`<LocalLeader>e`	Data frames HUD
`<LocalLeader>j`	Next R Markdown chunk
`<LocalLeader>k`	Previous R Markdown chunk
`<LocalLeader>l`	Execute current chunk
`<LocalLeader>t`	Execute all previous chunks
`<LocalLeader>q`	Quit R session
`<LocalLeader>c`	Interrupt R (Ctrl-C)

Part 1: Pre-Review Preparation

Step 1.1: Create Your Review Workspace

Before touching any code, establish an organized workspace for your review.

```
# Create a directory for review notes
mkdir -p ~/code-reviews/zztab2fig
cd ~/code-reviews/zztab2fig

# Create subdirectories for organization
mkdir notes
mkdir test-outputs
mkdir findings
mkdir debug-sessions
```

Create a master tracking document:

```
touch notes/REVIEW_LOG.md
```

Add the following template to `REVIEW_LOG.md`:

REVIEW_LOG.md Template

```
# zztab2fig Code Review Log
```

```
**Reviewer:** [Your name]
**Start Date:** [Date]
**Package Version:** 0.2.0
**IDE:** zzvim-R
```

Review Progress Table:

Phase	Status	Date Started	Date Completed
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		

Findings Summary Table:

Severity	Count	Resolved
Critical	0	0
Major	0	0
Minor	0	0
Enhancement	0	0

Step 1.2: Clone and Set Up the Package

```
# Navigate to the package directory
cd ~/prj/d01/zztab2fig

# Verify you have the latest version
git status
git log --oneline -5

# Create a review branch (optional but recommended)
git checkout -b code-review-$(date +%Y%m%d)
```

Step 1.3: Set Up zzvim-R Environment

Open Vim and navigate to the package directory:

```
cd ~/prj/d01/zztab2fig
vim .
```

Initial zzvim-R setup:

1. Open a source file to establish the R file type:

   ```
   :e R/tab2fig.R
   ```

2. Launch an R terminal in a vertical split:

   ```
   " Press <LocalLeader>w for vertical split
   " Or <LocalLeader>W for horizontal split
   " Or <LocalLeader>rr for host R with renv
   ```

3. Your screen should now show:

   ```
   ┌──────────────────────────┬──────────────────────────┐
   │                          │                          │
   │   R/tab2fig.R            │   R Terminal             │
   │   (source code)          │   > _                    │
   │                          │                          │
   │                          │                          │
   └──────────────────────────┴──────────────────────────┘
   ```

4. Load devtools in the R terminal by typing in the source pane and pressing <CR>:

   ```
   library(devtools)
   library(testthat)
   ```

5. Load the package in development mode:

   ```
   devtools::load_all()
   ```

   You should see: `zztab2fig 0.2.0 - LaTeX table generation for R`

Step 1.4: Run Initial Diagnostics with zzvim-R

With your R terminal open, execute these diagnostic commands. Position your cursor on each line and press <CR> to send it to R:

```
# Generate fresh documentation
devtools::document()

# Run R CMD check (this takes a few minutes)
check_results <- devtools::check()

# Run the test suite
test_results <- devtools::test()

# Check test coverage (requires covr package)
if (requireNamespace("covr", quietly = TRUE)) {
  coverage <- covr::package_coverage()
```

```
  print(coverage)
}
```

zzvim-R workflow tip: After running `devtools::check()`, use `<LocalLeader>0` to open the workspace HUD and monitor memory usage during the check process.

Record all output in your review log. Note any warnings or errors.

Step 1.5: Gather Package Metrics

Create a metrics script and execute it interactively:

```
# Count lines of code
r_files <- list.files("R", pattern = "\\.R$", full.names = TRUE)
total_lines <- sum(sapply(r_files, function(f) length(readLines(f))))
cat("Total R code lines:", total_lines, "\n")

# Count exported functions
ns <- readLines("NAMESPACE")
exports <- grep("^export\\(", ns, value = TRUE)
cat("Exported functions:", length(exports), "\n")

# Count test files
test_files <- list.files("tests/testthat", pattern = "^test-.*\\.R$")
cat("Test files:", length(test_files), "\n")
```

zzvim-R workflow: Select all lines visually (`V` then move down) and press `<CR>` to send the entire block to R.

For `zztab2fig`, you should find approximately:

- 13 R source files with approximately 5,300 lines of code
- 52 exported functions plus 29 S3 methods
- 7 test files with approximately 1,200 lines of tests

Part 2: Structural Review

Step 2.1: Verify Package Structure

In Vim, use netrw to explore the directory structure:

```
:Explore
```

Or use the terminal to list files:

```
ls -la
```

Expected structure:

```
zztab2fig/
├── DESCRIPTION         # Package metadata
```

```
├── NAMESPACE            # Export/import declarations
├── LICENSE              # License file
├── README.md            # User-facing documentation
├── R/                   # Source code (13 files)
├── man/                 # Documentation (auto-generated)
├── tests/
│   └── testthat/        # Test files (7 files)
├── vignettes/           # Long-form documentation
└── .Rbuildignore        # Files excluded from package
```

Verification checklist:

- ☐ DESCRIPTION exists and is properly formatted
- ☐ NAMESPACE exists (should be auto-generated by roxygen2)
- ☐ R/ directory contains .R files
- ☐ man/ directory exists
- ☐ tests/testthat/ directory exists
- ☐ testthat.R exists in tests/

Step 2.2: Review the DESCRIPTION File

Open DESCRIPTION in Vim:

```
:e DESCRIPTION
```

Read it carefully, then verify in R (press <CR> on each line):

```r
desc <- read.dcf("DESCRIPTION")
print(desc)
```

For zztab2fig, verify:

```
Package: zztab2fig
Title: Generate LaTeX Tables and PDF Outputs
Version: 0.2.0
Authors@R: person("RG", "Thomas", email = "rgthomas@ucsd.edu",
                  role = c("aut", "cre"),
                  comment = c(ORCID = "0000-0003-1686-4965"))
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
```

Review questions to answer:

1. Is the Title in title case (not sentence case)?
2. Is the Description a complete sentence ending with a period?
3. Are all authors properly specified with roles?
4. Is the license appropriate and does LICENSE file exist?

5. Are R version requirements reasonable? (R >= 4.0 is specified)

Step 2.3: Audit the NAMESPACE

Open and examine NAMESPACE:

```
:e NAMESPACE
```

Or read it in R:

```
ns <- readLines("NAMESPACE")
cat(ns, sep = "\n")
```

For zztab2fig, examine:

1. **Exported functions (52 total):**
   - Main API: `t2f`, `t2f_batch`, `t2f_inline`
   - Theme functions: `t2f_theme`, `t2f_theme_set`, `t2f_theme_get`, etc.
   - Advanced features: `t2f_footnote`, `t2f_header_above`, `t2f_collapse_rows`
   - Helpers: `geometry`, `babel`, `fontspec`
2. **S3 methods (29 total):**
   - `t2f.data.frame`, `t2f.lm`, `t2f.glm`, `t2f.anova`, `t2f.htest`
   - Survival: `t2f.coxph`, `t2f.survfit`, `t2f.survdiff`
   - Mixed models: `t2f.lmerMod`, `t2f.glmerMod`, `t2f.lme`
   - Print methods for custom classes
3. **Imports:**
   - `kableExtra`: kable, row_spec, kable_styling, column_spec
   - `stats`: coef, confint, nobs
   - `utils`: methods

Review questions:

- Are there any internal functions accidentally exported?
- Is the API surface appropriately minimal?
- Are all S3 methods properly registered?

Step 2.4: Review Dependencies

Check each dependency for necessity. In your R terminal:

```
# List Imports
imports <- c("kableExtra", "stats", "utils")

# For each import, check where it's used
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")
}
```

Dependency assessment for zztab2fig:

Package	Usage	Necessity
kableExtra	Core table generation	Essential
stats	coef, confint, nobs	Essential for model methods
utils	methods() function	Essential

Suggests analysis:

```
suggests <- c("broom", "broom.mixed", "survival", "lme4", "nlme",
              "MASS", "nnet", "testthat", "knitr", "rmarkdown",
              "tinytex", "dplyr", "stringr", "digest", "future.apply",
              "palmerpenguins")

for (pkg in suggests) {
  cmd <- paste0("grep -r 'requireNamespace.*", pkg, "' R/")
  matches <- system(cmd, intern = TRUE)
  if (length(matches) > 0) {
    cat(pkg, ": Used conditionally\n")
  }
}
```

Part 3: Function-by-Function Review with zzvim-R

This is the core of your review. You will examine every function in detail using zzvim-R's navigation and inspection capabilities.

Step 3.1: Create a Function Inventory

Generate a list of all functions to review:

```
# Get all R files
r_files <- list.files("R", pattern = "\\.R$", full.names = TRUE)

# Extract function definitions
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)
```

```
  }
}

# Print inventory
cat("Functions to review:", length(functions), "\n\n")
for (fn in names(functions)) {
  cat(sprintf("  %s (%s:%d)\n",
              fn, basename(functions[[fn]]$file), functions[[fn]]$line))
}
```

Step 3.2: Establish Review Order

Review files in a logical order based on dependencies:

1. **zzz.R** - Package initialization (start here)
2. **tab2fig.R** - Core internal implementation
3. **s3-methods.R** - S3 generic and methods
4. **themes.R** - Theme system
5. **advanced-features.R** - Footnotes, headers, collapse
6. **broom-methods.R** - Statistical object methods
7. **formatting.R** - Cell formatting utilities
8. **inline.R** - R Markdown integration
9. **latex-include.R** - LaTeX inclusion helpers
10. **batch.R** - Batch processing
11. **caching.R** - PDF caching
12. **output-formats.R** - PNG/SVG conversion
13. **knitr-engine.R** - Custom knitr engine

Step 3.3: zzvim-R Navigation for Code Review

Opening files efficiently:

```
" Open a specific file
:e R/tab2fig.R

" Jump to a specific line
:123

" Or combine them
:e R/tab2fig.R | 123

" Use Vim's gf (go to file) on require/source statements
" Position cursor on filename and press gf
```

Searching for function definitions:

```
" Search for function definition in current file
/function_name.*<-.*function
```

```
" Search across all R files (using vimgrep)
:vimgrep /t2f_internal/j R/*.R
:copen  " Open quickfix list

" Jump between matches
:cn  " Next match
:cp  " Previous match
```

Using tags for navigation (if available):

```
" Generate tags file
:!Rscript -e "rtags::rtags('R', ofile='tags')"

" Jump to function definition
:tag t2f_internal

" Return from tag jump
<C-t>
```

Step 3.4: Review Template for Each Function

For each function, create a review note using this template:

Function Review Template

```
## Function: [function_name]

**File:** R/[filename].R:[line_number]
**Type:** [Exported | Internal]
**Status:** [Pending | Reviewed | Needs Changes | Approved]

### Purpose
[One sentence describing what this function does]
```

Parameters Table:

Parameter	Type	Validated	Notes
param1	character	Yes	
param2	numeric	No	Missing validation

```
### Return Value
[Description of return value and its type]

### Understanding Summary
[2-3 sentences explaining the implementation in your own words]
```

```
### Code Quality
- [ ] Input validation present
- [ ] Error messages informative
- [ ] Edge cases handled
- [ ] No obvious bugs
- [ ] Code is readable
- [ ] Naming is consistent

### Test Coverage
- [ ] Basic functionality tested
- [ ] Edge cases tested
- [ ] Error conditions tested

### Concerns
1. [Issue description]
2. [Issue description]

### Questions
1. [Question requiring investigation]
```

Step 3.5: Detailed Review of Core Functions with Debugging

Let us walk through reviewing the most critical functions using zzvim-R's debugging capabilities.

3.5.1: Review t2f_internal (tab2fig.R:52-239) Open `R/tab2fig.R` in Vim:

```
:e R/tab2fig.R
:52
```

Read the function signature (visible in your editor):

```r
t2f_internal <- function(df, filename = NULL,
                sub_dir = "figures",
                scolor = NULL, verbose = FALSE,
                extra_packages = NULL,
                document_class = NULL,
                caption = NULL,
                caption_short = NULL,
                label = NULL,
                align = NULL,
                longtable = FALSE,
                crop = TRUE,
                crop_margin = 10,
                theme = NULL,
```

```
                footnote = NULL,
                header_above = NULL,
                collapse_rows = NULL)
```

Interactive debugging with browser():

To understand how the function works, insert a `browser()` call and step through execution. In your R terminal:

```
# Method 1: Use trace() to insert browser without modifying code
trace("t2f_internal", tracer = browser, where = asNamespace("zztab2fig"))

# Create test data and call the function
test_df <- data.frame(x = 1:3, y = c("a", "b", "c"))
t2f(test_df, "debug_test", sub_dir = tempdir())

# When browser() activates, use these commands:
# n - next line (step over)
# s - step into function call
# c - continue to next breakpoint or end
# Q - quit debugging
# ls() - list objects in current environment
# print(variable) - inspect variable

# Remove the trace when done
untrace("t2f_internal", where = asNamespace("zztab2fig"))
```

zzvim-R debugging workflow:

1. Position cursor on the `trace()` command and press `<CR>`
2. Position cursor on the test code and press `<CR>`
3. The R terminal enters debug mode
4. Type debug commands directly in the terminal
5. Use `<LocalLeader>s` on variable names to inspect with `str()`

Step through the implementation:

1. **Input Validation (lines 69-127):**

   In debug mode, after the function starts, examine the validation:

   ```
   # In browser, type:
   n  # Step to next line
   print(df)
   print(filename)
   class(df)
   ```

   Assessment questions:

   - Does every parameter get validated?
   - Are error messages informative?

- Is `call. = FALSE` used consistently? (Yes, good practice)

2. **Theme Resolution (lines 129-143):**

```
# Continue stepping
n
n
print(resolved_theme)
str(theme_settings)
```

3. **Directory Creation (lines 159-174):**

```
# Check directory handling
print(sub_dir)
dir.exists(sub_dir)
```

4. **LaTeX Compilation (lines 223-225):**

```
# Step into compile_latex to see how it works
s  # Step into
print(tex_file)
print(getwd())
```

Using debug() for deeper inspection:

```
# Debug a specific internal function
debug(zztab2fig:::compile_latex)

# Run test
t2f(test_df, "debug_test2", sub_dir = tempdir())

# Step through compile_latex
# Undebug when done
undebug(zztab2fig:::compile_latex)
```

Using debugonce() for single-shot debugging:

```
# Debug only the next call
debugonce(zztab2fig:::sanitize_column_names)

# This will enter debug mode only once
t2f(test_df, "test", sub_dir = tempdir())
```

3.5.2: Review t2f Generic and Methods (s3-methods.R) Open the file:

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

Review the generic:

```
t2f <- function(x, ...) {
  UseMethod("t2f")
}
```

Debug S3 method dispatch:

```
# See which method will be called
methods(t2f)

# Debug a specific method
debug(zztab2fig:::t2f.lm)

# Create test object and call
model <- lm(mpg ~ cyl + hp, data = mtcars)
t2f(model, "test_lm", sub_dir = tempdir())

# In browser, examine:
# print(x)  # The model object
# str(s)    # The summary
# print(coef_df)  # The coefficient data frame

undebug(zztab2fig:::t2f.lm)
```

Verify the implementation interactively:

Position cursor on each line and press <CR>:

```
# Test the function manually
model <- lm(mpg ~ cyl + hp, data = mtcars)

# Use zzvim-R inspection: position cursor on 'model' and press:
# <LocalLeader>s for str(model)
# <LocalLeader>p for print(model)

# Trace what the function does step by step
s <- summary(model)
coef_df <- as.data.frame(s$coefficients)

# Inspect with zzvim-R: cursor on coef_df, press <LocalLeader>h
print(coef_df)

# Verify column names match what the code expects
colnames(coef_df)
# [1] "Estimate"   "Std. Error" "t value"    "Pr(>|t|)"
```

3.5.3: Review the Theme System (themes.R) Open the file:

```
:e R/themes.R
```

Debug theme resolution:

```
# Trace theme lookup
trace("resolve_theme", tracer = browser, where = asNamespace("zztab2fig"))
```

```r
# Test with different theme inputs
t2f(mtcars[1:5,], "test", theme = "apa", sub_dir = tempdir())

# In browser:
# print(theme)
# Check how it resolves theme names to objects

untrace("resolve_theme", where = asNamespace("zztab2fig"))
```

Inspect the environment state:

```r
# Access the internal environment
env <- zztab2fig:::.t2f_env

# Examine its contents
ls(env)
env$current_theme
env$custom_themes
```

Step 3.6: AI-Specific Code Patterns to Watch For

During your review, specifically check for these AI-generated code patterns:

1. Over-Engineering

Look for unnecessary abstraction:

```
" Search for complex function chains
:vimgrep /function.*function.*function/j R/*.R
```

2. Defensive Redundancy

Check for duplicate validation:

```r
# Use grep to find repeated validation patterns
system("grep -n 'is.data.frame' R/*.R", intern = TRUE)
```

3. Inconsistent Style

Search for style variations in Vim:

```
" Check for assignment operator consistency
:vimgrep /[^<!=]=[^=]/j R/*.R

" Check for pipe operator usage
:vimgrep /%>%/j R/*.R
:vimgrep /|>/j R/*.R
```

4. Hallucinated Functions

Verify all function calls exist:

```r
# Debug by checking if functions exist
exists("kable", where = "package:kableExtra")
exists("row_spec", where = "package:kableExtra")

# Or use tryCatch to find undefined functions
tryCatch(
  zztab2fig:::nonexistent_function(),
  error = function(e) cat("Function not found:", e$message, "\n")
)
```

Part 4: Test Suite Review with zzvim-R

Step 4.1: Test Coverage Assessment

Run coverage analysis in your R terminal:

```r
library(covr)

# Run coverage analysis
coverage <- package_coverage()

# Print summary
print(coverage)

# Generate detailed HTML report
report(coverage)

# Use zzvim-R HUD to monitor memory during coverage analysis
# Press <LocalLeader>m
```

For zztab2fig, examine the coverage report:

Questions to answer:

- What is the overall line coverage percentage?
- Which functions have low or zero coverage?
- Are critical code paths covered?

Step 4.2: Review Test Infrastructure

Open the test helper file:

```
:e tests/testthat/helper.R
```

For zztab2fig:

```r
skip_if_no_latex <- function() {
  # Check if xelatex command exists
  has_xelatex <- suppressWarnings(
```

```
    system("xelatex --version", ignore.stdout = TRUE, ignore.stderr = TRUE)
  ) == 0

  if (!has_xelatex) {
    testthat::skip("xelatex not available")
  }

  # Try compiling a minimal fontspec document
  # ...
}
```

Debug test helper functions:

```
# Test the helper function directly
source("tests/testthat/helper.R")

# Debug it
debug(skip_if_no_latex)
skip_if_no_latex()
undebug(skip_if_no_latex)
```

Step 4.3: Review Each Test File with zzvim-R

Navigate to test files:

```
:e tests/testthat/test-zztab2fig.R
```

Use chunk navigation for test files:

In test files, each `test_that()` block functions like a chunk. Navigate with:

```
" Search for test blocks
/test_that

" Jump between test blocks
n  " Next match
N  " Previous match
```

Run individual tests interactively:

Position cursor inside a test block and send it to R:

```
test_that("t2f 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],
    stringsAsFactors = FALSE
  )

  output_file <- t2f(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"))
})
```

Debug failing tests:

```
# If a test fails, debug it
test_that("failing test", {
  browser()  # Add this temporarily
  # ... test code ...
})

# Or use testthat's built-in debugging
testthat::test_file("tests/testthat/test-zztab2fig.R", reporter = "debug")
```

Test quality checklist:

- ☒ Uses `skip_if_*` for external dependencies
- ☒ Creates isolated test directory
- ☒ Uses `on.exit` for cleanup (prevents test pollution)
- ☒ Tests actual behavior (file creation)
- ☐ Could verify file contents, not just existence
- ☐ Could verify PDF is valid

Step 4.4: Check for Missing Tests

Compare tested functionality against documented functionality:

```
# Get all exported functions
exports <- getNamespaceExports("zztab2fig")

# Check which have tests
test_files <- list.files("tests/testthat", pattern = "^test-.*\\.R$",
                         full.names = TRUE)
test_content <- unlist(lapply(test_files, readLines))

# Find untested exports
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")
```

Step 4.5: Evaluate Test Independence

Check for tests that depend on each other:

```
" Search for global state modifications in test files
:vimgrep /options\s*(/j tests/testthat/*.R
:vimgrep /t2f_theme_set/j tests/testthat/*.R
```

Part 5: Integration and Security Review

Step 5.1: Trace Data Flow with Debugging

Use `trace()` to observe data flow through the package:

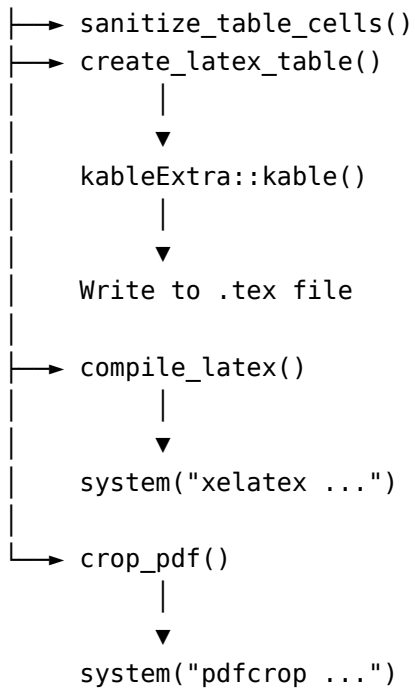
```
# Trace multiple functions to see the call chain
trace("t2f", tracer = quote(cat(">>> Entering t2f\n")),
      where = asNamespace("zztab2fig"))
trace("t2f_internal", tracer = quote(cat(">>> Entering t2f_internal\n")),
      where = asNamespace("zztab2fig"))

# Run a simple example
test_df <- data.frame(x = 1:3, y = c("a", "b", "c"))
t2f(test_df, "trace_test", sub_dir = tempdir())

# Clean up traces
untrace("t2f", where = asNamespace("zztab2fig"))
untrace("t2f_internal", where = asNamespace("zztab2fig"))
```

Data flow diagram:

```
User Input (data.frame)
    │
    ▼
t2f() [S3 generic]
    │
    ▼
t2f.data.frame() [or other method]
    │
    ▼
t2f_internal()
    │
    ├──► sanitize_column_names()
```



Step 5.2: Review System Command Execution

Search for all `system()` calls in Vim:

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

Debug system calls:

```
# Trace system calls to see what commands are executed
trace("system", tracer = quote(cat("System call:", commandArgs(), "\n")))

# Run a test
t2f(mtcars[1:3,], "system_test", sub_dir = tempdir())

untrace("system")
```

Security assessment for each system call:

1. **compile_latex:**
   - `shQuote()` used: Prevents shell injection
   - `basename()` used: Removes directory traversal
   - Fixed command: Only xelatex, not user-controlled
2. **crop_pdf:**
   - All paths quoted with `shQuote()`
   - Margin string is numeric (validated earlier)
   - Fixed command structure

Verdict: System command usage is safe.

Step 5.3: Review File Operations

Search for file operations in Vim:

```
:vimgrep /file\.\|writeLines\|readLines/j R/*.R
:copen
```

Debug file operations:

```
# Trace file writing
trace("writeLines", tracer = quote(cat("Writing to:", con, "\n")))

t2f(mtcars[1:3,], "file_test", sub_dir = tempdir())

untrace("writeLines")
```

Step 5.4: Check for Sensitive Data Handling

Search in Vim:

```
:vimgrep /password\|secret\|token\|key/j R/*.R
```

For zztab2fig: No sensitive data handling. The package works with statistical output, not credentials.

Part 6: Documentation Review

Step 6.1: Review Function Documentation

Use zzvim-R to access R help:

```
# Load the package
library(zztab2fig)

# View help for main function
?t2f

# Check all available help topics
help(package = "zztab2fig")

# Verify all parameters documented
args(t2f.default)
```

zzvim-R shortcut: Position cursor on a function name and press `<LocalLeader>y` to call `help()` on it.

Documentation checklist for t2f:

- ☐ Title is concise and descriptive
- ☐ Description explains purpose

- ☐ All parameters have `@param` tags
- ☐ Return value documented with `@return`
- ☐ Examples provided (use `\dontrun{}` if requires LaTeX)
- ☐ Related functions linked with `@seealso`

Step 6.2: Build and Check Documentation

```
# Generate documentation
devtools::document()

# Check for documentation warnings
devtools::check_man()

# Build package manual
devtools::build_manual()
```

Step 6.3: Review Vignettes with zzvim-R

Open vignettes in Vim:

```
:e vignettes/quickstart.Rmd
```

Use zzvim-R chunk navigation:

- `<LocalLeader>j` - Jump to next chunk
- `<LocalLeader>k` - Jump to previous chunk
- `<LocalLeader>l` - Execute current chunk
- `<LocalLeader>t` - Execute all previous chunks

Verify vignettes build:

```
devtools::build_vignettes()
```

Step 6.4: Review README

```
:e README.md
```

Checklist:

- ☐ Installation instructions present
- ☐ Basic usage example included
- ☐ Examples work when copied
- ☐ Links are valid
- ☐ Badges (if any) are current

Part 7: Remediation

Step 7.1: Categorize Findings

Sort all findings by severity:

Critical (must fix before release):

- Security vulnerabilities
- Crashes or data corruption
- Incorrect calculations

Major (should fix before release):

- Significant deviations from expected behavior
- Missing input validation
- Confusing error messages

Minor (fix when convenient):

- Style inconsistencies
- Documentation gaps
- Minor inefficiencies

Enhancements (consider for future):

- Feature suggestions
- Performance improvements
- API polish

Step 7.2: Implement Fixes with zzvim-R

Workflow for fixing issues:

1. Open the file containing the issue:

   ```
   :e R/s3-methods.R
   :173  " Jump to line
   ```

2. Make the edit in Vim

3. Test the fix interactively:

   ```r
   # Reload the package
   devtools::load_all()

   # Test the specific fix
   model <- lm(mpg ~ cyl, data = mtcars)

   # Debug to verify
   debugonce(zztab2fig:::t2f.lm)
   t2f(model, digits = 3)
   ```

4. Add a test for the fix:

```
:e tests/testthat/test-zztab2fig.R
G  " Go to end of file
```

Add test:

```
test_that("t2f.lm validates digits parameter", {
  model <- lm(mpg ~ cyl, data = mtcars)
  expect_error(t2f(model, digits = -1), "non-negative")
  expect_error(t2f(model, digits = "three"), "numeric")
})
```

5. Run tests to verify:

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

Part 8: Final Verification

Step 8.1: Complete Test Suite

Run all tests and verify they pass:

```
# Run tests
test_results <- devtools::test()

# Check for failures
if (any(test_results$failed > 0)) {
  stop("Tests failed! Fix before proceeding.")
}
```

Step 8.2: R CMD check

```
# Full check (as CRAN would run)
check_results <- devtools::check()

# Check for errors, warnings, notes
print(check_results)
```

For CRAN submission, you need:

- 0 errors
- 0 warnings
- 0 notes (or notes you can justify)

Step 8.3: Fresh Installation Test

```
# Build the package
devtools::build()

# Install from the built tarball
```

```r
install.packages("../zztab2fig_0.2.0.tar.gz", repos = NULL, type = "source")

# Verify it works
library(zztab2fig)
check_latex_deps()

# Quick functional test
test_df <- data.frame(x = 1:3, y = letters[1:3])
t2f(test_df, "test", sub_dir = tempdir())
```

Part 9: Establishing Ownership

Step 9.1: Knowledge Verification

Before claiming ownership, verify you can answer these questions without referring to the code:

Architecture questions:

1. What is the main entry point for users?
2. How does the S3 dispatch work?
3. Where is global state stored?
4. What system dependencies are required?

Implementation questions:

1. How are special LaTeX characters handled?
2. What happens when LaTeX compilation fails?
3. How does the theme system work?
4. How are footnotes implemented?

Debugging questions:

1. Where would you look if a table has wrong formatting?
2. How would you diagnose a LaTeX compilation error?
3. What would cause t2f to return silently without creating files?

Step 9.2: Demonstrate Debugging Proficiency

Show you can debug issues without AI assistance:

```r
# Scenario: A user reports that special characters break tables
# Debug approach:

# 1. Create a test case that reproduces the issue
test_df <- data.frame(
  name = c("Test & Co.", "100% Complete", "Price: $50"),
  value = 1:3
)
```

```r
# 2. Set up debugging
trace("sanitize_table_cells", tracer = browser,
      where = asNamespace("zztab2fig"))

# 3. Run and step through
t2f(test_df, "special_chars", sub_dir = tempdir())

# 4. In browser, examine:
# - What input does the function receive?
# - What transformations are applied?
# - What output is produced?

# 5. Clean up
untrace("sanitize_table_cells", where = asNamespace("zztab2fig"))
```

Step 9.3: Create Your Own Examples

Write original code using the package to prove understanding:

```r
# Example: Create a custom theme for my journal
my_journal_theme <- t2f_theme(
  name = "my_journal",
  scolor = "gray!5",
  header_bold = TRUE,
  font_size = "small",
  extra_packages = list(
    geometry(margin = "1in"),
    "\\usepackage{mathpazo}"
  ),
  striped = TRUE
)

# Register and use it
t2f_theme_register(my_journal_theme)
t2f_theme_set("my_journal")

# Create a regression table
model <- lm(mpg ~ wt + hp + cyl, data = mtcars)
t2f(model, "my_regression", digits = 2,
    caption = "Regression Results",
    label = "tab:regression")
```

Part 10: CRAN Submission

Step 10.1: CRAN Policy Review

Read the current CRAN policies: https://cran.r-project.org/web/packages/policies.html

Key requirements:

- Package must pass `R CMD check` with no errors, warnings
- Package must work on Linux, macOS, and Windows
- Examples must run in reasonable time
- Vignettes must build successfully
- License must be appropriate
- No external downloads during installation

Step 10.2: Pre-Submission Checklist

Package Metadata:

☐ Version number is appropriate (0.2.0)
☐ Description is a single paragraph
☐ License is valid and file exists
☐ Maintainer email is valid
☐ URL and BugReports are valid links

Code Quality:

☐ R CMD check passes with 0 errors, 0 warnings
☐ No undocumented functions
☐ No missing imports/exports
☐ Examples run without errors
☐ No long-running examples (use `\donttest{}` if needed)

System Dependencies:

☐ SystemRequirements field documents external deps
☐ Package handles missing system deps gracefully
☐ Error messages guide users to install dependencies

Documentation:

☐ README.md is informative
☐ NEWS.md exists with changelog
☐ All vignettes build
☐ Examples are useful

Testing:

☐ Tests pass on all platforms
☐ Tests skip gracefully when deps missing
☐ Test coverage is adequate

Step 10.3: Build and Final Check

```r
# Build source package
devtools::build()

# Run CRAN-like check
devtools::check(cran = TRUE)

# Check reverse dependencies (if updating)
# revdepcheck::revdep_check()
```

Step 10.4: Submit to CRAN

```r
# Submit to CRAN
devtools::release()
```

Appendix A: zzvim-R Debugging Reference

R Debugging Functions

Function	Purpose
`browser()`	Insert breakpoint, enter interactive debug mode
`debug(fun)`	Debug function on every call
`debugonce(fun)`	Debug function on next call only
`undebug(fun)`	Remove debug flag
`trace(fun, tracer)`	Insert code at function entry
`untrace(fun)`	Remove trace
`traceback()`	Show call stack after error
`recover()`	Enter browser at any point in call stack
`options(error = recover)`	Automatically recover on error

Browser Commands

Command	Action
`n`	Execute next line (step over)
`s`	Step into function call
`f`	Finish current function
`c`	Continue to next breakpoint
`Q`	Quit debugging
`where`	Show call stack
`ls()`	List objects in current frame

zzvim-R Inspection Shortcuts

Key	Function	Use Case
`<LocalLeader>h`	`head()`	Preview data
`<LocalLeader>s`	`str()`	See structure
`<LocalLeader>d`	`dim()`	Check dimensions
`<LocalLeader>p`	`print()`	Full output
`<LocalLeader>n`	`names()`	Column/element names
`<LocalLeader>g`	`glimpse()`	Tidyverse preview
`<LocalLeader>0`	HUD	Full workspace view

Common Debugging Patterns

Pattern 1: Trace function entry

```
trace("function_name", tracer = browser, where = asNamespace("pkg"))
# ... run code ...
untrace("function_name", where = asNamespace("pkg"))
```

Pattern 2: Conditional breakpoint

```
trace("function_name",
      tracer = quote(if (nrow(df) > 100) browser()),
      where = asNamespace("pkg"))
```

Pattern 3: Log function calls

```
trace("function_name",
      tracer = quote(cat("Called with:", deparse(substitute(x)), "\n")),
      where = asNamespace("pkg"))
```

Pattern 4: Debug on error

```
options(error = recover)
# ... run code that errors ...
options(error = NULL)  # Reset
```

Appendix B: Common Findings Reference

Input Validation Issues

```
# Bad: No validation
f <- function(x) {
  sum(x)
}

# Good: With validation
f <- function(x) {
```

```r
  if (!is.numeric(x)) {
    stop("`x` must be numeric.", call. = FALSE)
  }
  sum(x)
}
```

Error Message Issues

```r
# Bad: Unhelpful error
stop("Invalid input")

# Good: Informative error
stop("`x` must be a positive integer, got: ", class(x)[1], call. = FALSE)
```

Resource Cleanup Issues

```r
# Bad: No cleanup
f <- function() {
  old_wd <- setwd("/tmp")
  # ... code that might error ...
  setwd(old_wd)
}

# Good: With cleanup
f <- function() {
  old_wd <- setwd("/tmp")
  on.exit(setwd(old_wd))
  # ... code that might error ...
}
```

Appendix C: Checklist Summary

Pre-Review

- ☐ Created review workspace
- ☐ Set up zzvim-R environment
- ☐ Launched R terminal with `<LocalLeader>w`
- ☐ Loaded package with `devtools::load_all()`
- ☐ Ran initial diagnostics
- ☐ Documented package metrics

Structural Review

- ☐ Verified package structure
- ☐ Reviewed DESCRIPTION

- ☐ Audited NAMESPACE
- ☐ Assessed dependencies

Function Review

- ☐ Created function inventory
- ☐ Reviewed each function systematically
- ☐ Used `browser()` and `debug()` for understanding
- ☐ Checked for AI-specific patterns
- ☐ Documented findings

Test Review

- ☐ Assessed test coverage
- ☐ Reviewed test quality
- ☐ Ran tests interactively with zzvim-R
- ☐ Checked test independence
- ☐ Identified missing tests

Integration/Security Review

- ☐ Traced data flow with `trace()`
- ☐ Reviewed system commands
- ☐ Checked file operations
- ☐ Verified no sensitive data exposure

Documentation Review

- ☐ Verified function documentation
- ☐ Used `<LocalLeader>y` for help lookup
- ☐ Checked vignettes build
- ☐ Reviewed README
- ☐ Verified NEWS.md exists

Final Steps

- ☐ Fixed all critical issues
- ☐ R CMD check passes
- ☐ Fresh installation works
- ☐ Debugging proficiency demonstrated
- ☐ Knowledge verification complete
- ☐ CRAN checklist complete

This exemplar review guide provides a concrete template for conducting thorough code reviews of AI-assisted R packages using zzvim-R as your IDE. Adapt the specifics to your package while maintaining the systematic approach and debugging practices.