Paper PP11

# Optimizing Package Development: A Detailed Guide to Workflow and Best Practices

Tejaskumar Mehta, Merck & Co., Inc., Rahway, NJ, USA
Rikkin Parekh, Merck & Co., Inc., Rahway, NJ, USA

## ABSTRACT

Developing your own R package is a powerful way to systematically organize code, enhance reproducibility, and share analytical tools with a broader audience. This paper will present a detailed, step-by-step guide tailored specifically for beginners aiming to create robust and professional-quality R packages.

Readers will learn how to establish a well-structured package framework; design clean and reusable functions and produce thorough documentation that ensures ease of use and clarity. The guide will emphasize the importance of rigorous testing to verify code reliability and will introduce effective strategies for managing package dependencies. Additionally, it will provide practical instructions on how to create your package and hosting it on GitHub, facilitating wider accessibility and collaboration.

By following this comprehensive roadmap, beginners will acquire the essential skills and confidence needed to transform their R scripts into polished, maintainable packages that significantly improve project workflows and foster collaborative development.

## INTRODUCTION

In R, packages are the main way to organize and share code. An R package groups together functions, data, documentation, and tests in a clear and consistent structure, making it easy for others to install and use. The number of packages available on the Comprehensive R Archive Network (CRAN) shows how important packages are to the R community. Because of this, learning how to develop R packages is an essential skill for anyone working with R.

Creating your own R package has multiple benefits:

- It enables easy distribution of code to others.
- It improves reproducibility and organization of work.
- It encourages good programming practices, by imposing a consistent structure on code.
- It makes it simpler to maintain, test, and document functions over time.

Modern R package development emphasizes automation, consistency, and simplicity, allowing developers to focus on analytical objectives rather than low-level technical details. This approach encourages streamlined workflows and the use of standardized conventions to improve efficiency and code quality. Supporting tools such as devtools, usethis, and roxygen2 automate many common tasks, including package creation, documentation, and testing. By reducing manual effort and complexity, these tools make the package development process more efficient, maintainable, and accessible, particularly for users transitioning from script-based programming to structured software development.

## R PACKAGE PROCESS

For illustrative purposes, this paper demonstrates the step-by-step process of R package development using a single package named ***dataverse***. The package includes the implementation of the ***fn_fetch()*** function, which serves as an example of core functionality within the package.

### 1. Prepare Session and Folder Structure.

### 1.1 Load devtools and usethis.

```
install.packages(c( "devtools", "roxygen2", "testthat", "covr", "knitr"))
library(devtools)
library(usethis)
```

Before starting any R package work, it is best to install and load a small set of development helper packages that standardize workflow and prevent common mistakes. Install them from CRAN with *install.packages(c("devtools", "roxygen2", "testthat", "covr", "knitr", "usethis"))*, then load them using *library(devtools)* and *library(usethis)* (User can also load *roxygen2, testthat, covr,* and *knitr* as needed). These packages provide a layer of convenience on top of base R by offering safe, repeatable commands for common development tasks. Rather than manually creating folders or editing configuration files, these tools enforce consistent structure and reduce beginner mistakes.

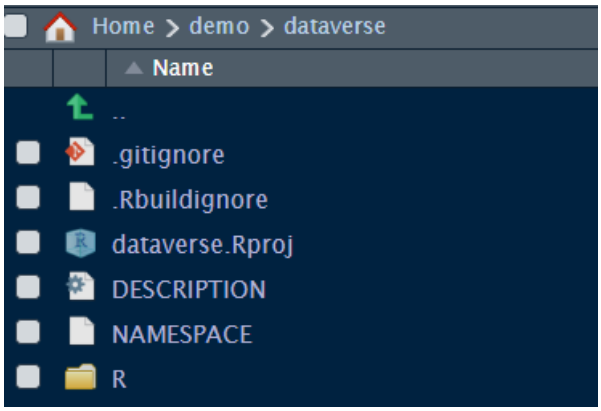
### 1.2 Create the package structure.

Development begins by creating a dedicated package directory using *usethis* package. This step establishes a standardized structure, including folders for R code, documentation, tests, and metadata. Early standardization reduces ambiguity later and supports collaboration and reproducibility.

```
dir.create('~/demo/dataverse', recursive = TRUE, showWarnings = FALSE)
usethis::create_package('~/demo/dataverse')
```

These commands first ensure that the target directory (and any missing parents) exists without emitting warnings if it already does, then create a new R package skeleton in that location. The usethis step generates the standard package layout (e.g., *DESCRIPTION, NAMESPACE, R/),* adds essential metadata, and readies the project for tools like R CMD check and devtools. This approach is recommended for beginners because manually assembling package folders is error-prone and small structural mistakes can lead to confusing build, check, or install failures later.

**Output**




### 2. Enable version control.

Version control is activated early so that every change is tracked. This allows developers to experiment freely, knowing they can always return to a previous state. For collaborative or regulated environments, version history is essential.

**Prerequisites:**

- Git is already installed and configured on system (e.g., user.name and user.email is set).
- Authentication for chosen hosting service (e.g., GitHub) is set up (e.g., SSH keys or personal access tokens).
- A remote repository exists, and user has permissions to push it.

### 2.1 Initialize local version control.

```
usethis::use_git()
```

Above command creates the *.git* repository, adds a *.gitignore* (and possibly an *.Rbuildignore*), makes an initial commit, and leaves the project under Git without a remote configured.

### 2.2 Connect the repository to a remote named origin using either HTTPS or SSH. Use terminal for git commands.

```
git remote add origin https://github.com/<username>/dataverse.git
```

```
git remote add origin git@github.com:<username>/dataverse.git
```

### 2.3 Rename the default branch to main.

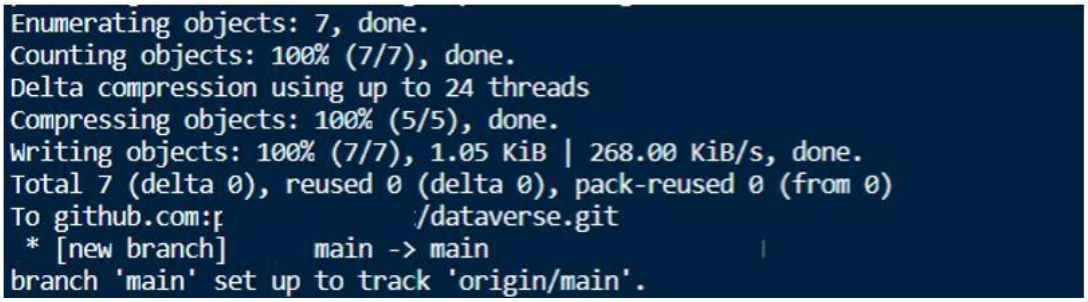
```
git branch -M main
```

### 2.4 Push and establish upstream tracking.

```
git push -u origin main
```

### Output

Git counts the local repository objects (commits, trees, blobs), compresses them with delta compression, and writes them to the remote, showing progress. It creates a new branch and reports total such as objects, which is typical on an initial push. The -u flag sets local main to track origin/main, so future push/pull commands can omit the remote/branch and status will show ahead/behind information.




### 3. Develop R Function and create documentation of the function.

We will implement the *fn_fetch()* function, designed to retrieve a dataset by name and optionally apply refinements. If lib is NULL, the function will extract an R data frame from the global environment; otherwise, it will read a SAS dataset from the specified *path[[lib]]*. After loading the data, the function can:

- Filter rows using the sub_set parameter.
- Select specific columns via the keep argument.
- Arrange the data based on columns specified in by argument.

The function returns the resulting refined data frame. For the purposes of this paper, the workflow is designed for interactive RStudio sessions, and “active file” means the file currently open and focused in the RStudio editor.

### 3.1 Create an R script for the function.

```
usethis::use_r("fn_fetch")
```

Above command step creates a properly named R script in the correct location. Using helper commands ensures naming consistency and keeps related code discoverable. Ensure the following R packages are installed before proceeding further steps: *dplyr, haven, rlang, magrittr, tidyselect*. If any stated package is missing, install with *install.packages(c("dplyr", "haven", "rlang", "magrittr", "tidyselect")).*

```
fn_fetch <- function(domain, lib = NULL, sub_set = NULL, by = NULL, keep = NULL, path
= NULL) {
  if (is.null(lib)) {
    if (exists(domain, envir = .GlobalEnv)) {
      df <- get(domain, envir = .GlobalEnv)
    } else {
      stop(paste("Data frame", domain, "does not exist in the global environment."))
    }
  } else {
    if (is.null(path)) {
      stop("Argument 'path' must be provided when 'lib' is not NULL.")
    }
    if (!(lib %in% names(path))) {
      stop(paste0("Library '", lib, "' not found in 'path'."))
    }
    sas_file <- file.path(path[[lib]], paste0(domain, ".sas7bdat"))
    if (!file.exists(sas_file)) {
      stop(paste0("SAS dataset file '", sas_file, "' does not exist."))
    }
    df <- haven::read_sas(sas_file)
  }

  if (!is.null(sub_set) && sub_set != "") {

    df <- dplyr::filter(df, !!rlang::parse_quo(sub_set, env = rlang::env_parent()))
  }
  if (!is.null(keep) && length(keep) > 0) {
    df <- df %>% dplyr::select(tidyselect::all_of(keep))
  }
  if (!is.null(by) && length(by) > 0) {
    df <- df %>% dplyr::arrange(dplyr::across(tidyselect::all_of(by)))
  }
  return(df)
}
```

**Output**
`R/fn_fetch.R`

### 3.2 Add roxygen2 comments to your functions so documentation lives alongside the code.

These structured comments are parsed to automatically produce the formal help pages and update the *NAMESPACE*, keeping docs and exports in sync.

```
#' Fetch and optionally filter, select, and arrange a data frame
#'
#' @param domain Character. Name of the data frame or SAS dataset (without extension)
to fetch.
#' @param lib Character or NULL. If NULL, fetches data frame from global environment.
#'   If character, used as a key to access a path in `path` list to read SAS dataset.
#' @param sub_set Character or NULL. Filter expression as string (e.g., "age > 30").
#' @param by Character vector or NULL. Columns to arrange by.
#' @param keep Character vector or NULL. Columns to select.
#' @param path Named list or NULL. Named list of library paths where SAS datasets are
stored.
#'   Required if `lib` is not NULL.
#'
#' @return Filtered, selected, and arranged data frame.
#'
#' @examples
#'
#' # Define path to SAS datasets inside installed package
#' path <- list(test = system.file("extdata", package = "dataverse"))
#'
#' # Fetch SAS dataset "testdata" from library "test" filtering age > 30
#' df <- fn_fetch("testdata", lib = "test", sub_set = "age > 30", path = path)
#' head(df)
```

```
#'
#' # Fetch from global environment
#' sample_df <- data.frame(id = 1:3, age = c(20, 30, 40))
#' assign("mydata", sample_df, envir = .GlobalEnv)
#' df2 <- fn_fetch("mydata", lib = NULL)
#' print(df2)
#' rm(mydata, envir = .GlobalEnv)
#' @importFrom magrittr %>%
#' @importFrom dplyr filter select arrange across
#' @importFrom tidyselect all_of
#' @importFrom rlang parse_expr parse_quo env_parent
#' @importFrom haven read_sas
#' @export
```

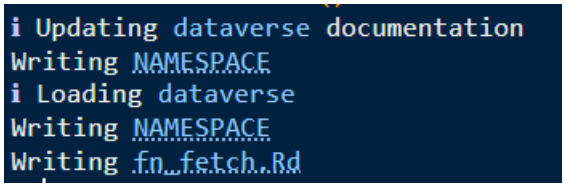
### 3.3 Generate documentation of the function.

```
devtools::document()
```

Above step processes the *Roxygen2* comments to automatically generate or update the documentation files located in *man/*.Rd* and to update the *NAMESPACE* file. These files are essential for ensuring that the package functions are properly documented and exported.

### Output

```
man/fn_fetch.Rd
```

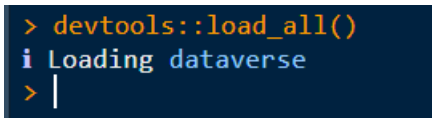



### 3.4 Load the package from source.

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

Instead of installing the package repeatedly, it is loaded directly from source during development. This shortens the feedback loop and allows rapid iteration after each edit.

### Output




## 4. Unit Testing Strategy.

Testing is integrated early using the testthat framework to ensure *fn_fetch()* behaves reliably. Each test encodes a specific expectation, such as correctly sourcing data from the global environment or a SAS library, applying *sub_set* filters to rows, selecting only the columns in keep, arranging by the specified by variables, and returning a data frame with consistent types and error handling when inputs (e.g., missing lib/path, nonexistent columns) are invalid.

### 4.1 Create dummy data for testing (Optional).

Create a small dummy dataset and save it as a SAS file for testing. It builds a 5-row data frame (id, age, gender), ensures *inst/extdata* exists, and writes *inst/extdata/testdata.sas7bdat* using haven. The file is installed with the package. Purpose of this dataset is to provide a stable fixture to test *fn_fetch()*.

Sample code to create dummy dataset:

```
library(haven)
dir.create("inst/extdata", recursive = TRUE, showWarnings = FALSE)
dummy_df <- data.frame(
id = 1:5,
age = c(25, 30, 22, 40, 35),
gender = c("M", "F", "F", "M", "F"),
stringsAsFactors = FALSE)
haven::write_sas(dummy_df, "inst/extdata/testdata.sas7bdat")
```

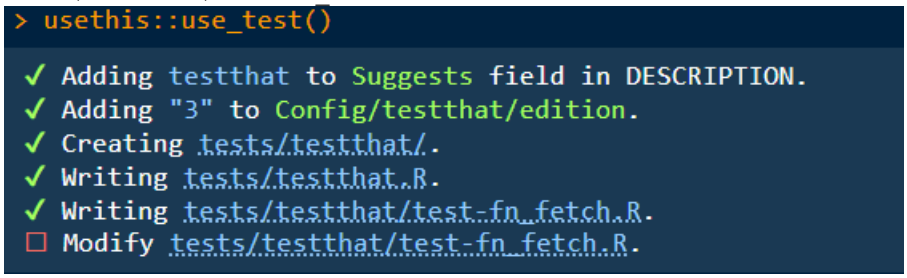
### 4.2 Write test cases.

```
usethis::use_test()
```

Above command sets up a new test file for the selected function using the *testthat* framework by ensuring the *tests/testthat/* directory exists, configuring testthat if needed, and creating a file like *tests/testthat/test-fn_fetch.R* where user can immediately start writing tests. Ensure function fn_fetch.R is active while using *usethis::use_test()*

**Output**

```
tests/testthat.R
tests/testthat/test-fn_fetch.R
```




Add following testcases to *test-fn_fetch.R*

```
test_that("fn_fetch reads SAS dataset correctly", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  df <- fn_fetch("testdata", lib = "test", path = path)
  expect_true(nrow(df) > 0)
  expect_true(all(c("id", "age", "gender") %in% colnames(df)))
})

test_that("fn_fetch filters rows with sub_set", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  df <- fn_fetch("testdata", lib = "test", sub_set = "age > 30", path = path)
  expect_true(all(df$age > 30))
})

test_that("fn_fetch selects columns with keep", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  df <- fn_fetch("testdata", lib = "test", keep = c("id", "gender"), path = path)
  expect_equal(colnames(df), c("id", "gender"))
})

test_that("fn_fetch arranges rows with by", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  df <- fn_fetch("testdata", lib = "test", by = "age", path = path)
  expect_equal(df$age, sort(df$age))
})

test_that("fn_fetch errors if path missing when lib provided", {
  expect_error(
    fn_fetch("testdata", lib = "test"),
    "Argument 'path' must be provided")
})
```

```
test_that("fn_fetch errors if lib not in path", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  expect_error(
    fn_fetch("testdata", lib = "wronglib", path = path),
    "Library 'wronglib' not found")
})

test_that("fn_fetch errors if SAS file missing", {
  path <- list(test = system.file("extdata", package = "dataverse"))
  expect_error(
    fn_fetch("nonexistent", lib = "test", path = path),
    "does not exist")
})

test_that("fn_fetch fetches from global environment", {
  sample_df <- data.frame(id = 1:3, age = c(20, 30, 40))
  assign("mydata", sample_df, envir = .GlobalEnv)
  df <- fn_fetch("mydata", lib = NULL)
  expect_equal(df, sample_df)
  rm(mydata, envir = .GlobalEnv)
})

test_that("fn_fetch errors if domain missing in global env", {
  expect_error(
    fn_fetch("nonexistent_df", lib = NULL),
    "does not exist")
})
```

### 4.3 Perform Test for function.

```
devtools::test_active_file()
```

Above command runs tests related to the currently active source file (*test-fn_fetch.R*) and the output generally consist of a summary indicating the number of tests run, the number of passing tests, the number of failing tests, and the number of warnings or skipped tests.

**Output**

```
> devtools::test_active_file()
[ FAIL 0 | WARN 0 | SKIP 0 | PASS 10 ]
```

```
devtools::test_coverage_active_file()
```

Above command runs tests coverage related to the currently active source file *(test-fn_fetch.R)* and reports code coverage for that file.

### Output

- A percentage of lines covered for that file (e.g., “Coverage: 85%”).
- A per-line breakdown indicating which lines were executed by tests and which were missed.
- A table or listing of uncovered lines/functions so you can target gaps.

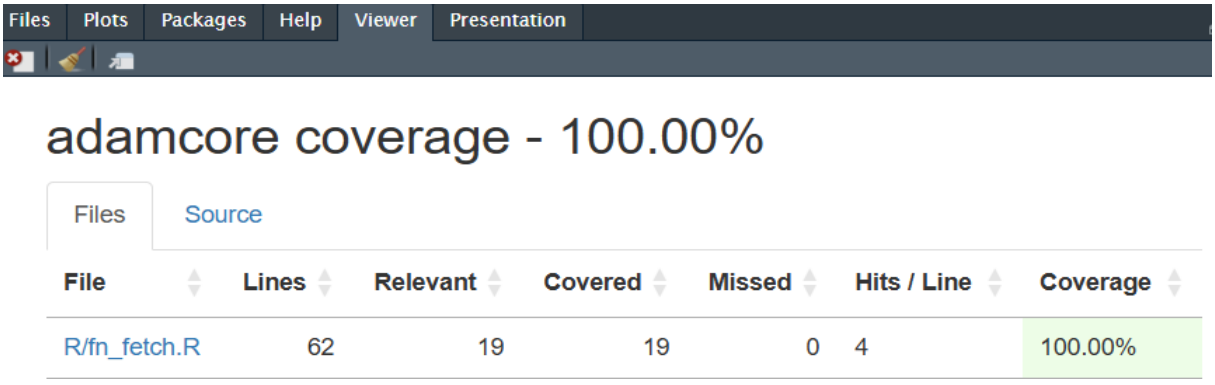
### 4.4 Perform Test for package.

```
devtools::test_coverage()
```

Above command runs package’s test suite and measures code coverage across the whole package using *covr package*.

**Output**

- Executes tests (typically those in *tests/testthat*)
- Computes which lines/functions were executed.
- Prints a summary to the console (overall coverage percentage, plus per-file percentages)
- Returns a covr coverage object you can further inspect.




### 5. Finalize Documentation and local installation.

#### 5.1 Update DESCRIPTION & Documents.

```
devtools::document()
```

Documentation is generated and maintained using structured comments within the source code. This approach ensures that documentation remains consistent and synchronized with the implementation details.

Sample *DESCRIPTION* file.

```
Package: dataverse
Title: Core Functions for ADaM/SDTM Development
Version: 0.0.0.9000
Authors@R:
    person("First", "Last", , "first.last@company.com", role = c("aut", "cre"))
Description: Core functions to support ADaM and SDTM workflows.
License: MIT + file LICENSE
Encoding: UTF-8
Roxygen: list(markdown = TRUE)
RoxygenNote: 7.3.3
Config/testthat/edition: 3
Imports: dplyr, haven, rlang, magrittr, tidyselect
Suggests: testthat (>= 3.0.0)
Depends: R (>= 4.1.0)

```

#### 5.2 Use appropriate license.

```
usethis::use_mit_license()
```

```
> usethis::use_mit_license()
✓ Writing LICENSE.
✓ Writing LICENSE.md.
✓ Adding "^LICENSE\\.md$" to .Rbuildignore.
```

### 5.3 Perform Check.

```
devtools::check()
```

Running checks early helps identify structural or metadata issues before they accumulate. At this stage, warnings and informational notes are expected and serve as guidance for subsequent steps

**Output**

```
✓  checking examples (1.7s)
✓  checking for unstated dependencies in 'tests' ...
─  checking tests ...
✓  Running 'testthat.R' (1.6s)
✓  checking for non-standard things in the check directory
✓  checking for detritus in the temp directory


── R CMD check results ──────────────────────────────────── dataverse 0.0.0.9000 ────
Duration: 37.2s

0 errors ✓ | 0 warnings ✓ | 0 notes ✓
```

### 5.4 Local Installation.

```
devtools::install()
```

Installing the package replicates the process end users will follow to load it, ensuring that installation succeeds outside the development environment.

**Output**

```
*** installing help indices
  converting help for package 'dataverse'
    finding HTML links ... done
    fn_fetch                                html
** building package indices
** testing if installed package can be loaded from temporary location
** testing if installed package can be loaded from final location
** testing if installed package keeps a record of temporary installation path
* DONE (dataverse)
>
```

### 5.5 Create README.Rmd & README.md files.

```
usethis::use_readme_rmd()
```

The *README* provides a concise overview along with example usage. Authoring it in *R Markdown* enables examples to be executed and validated. To generate the final *README.md* file, knit the *README.Rmd* document.

**Output**

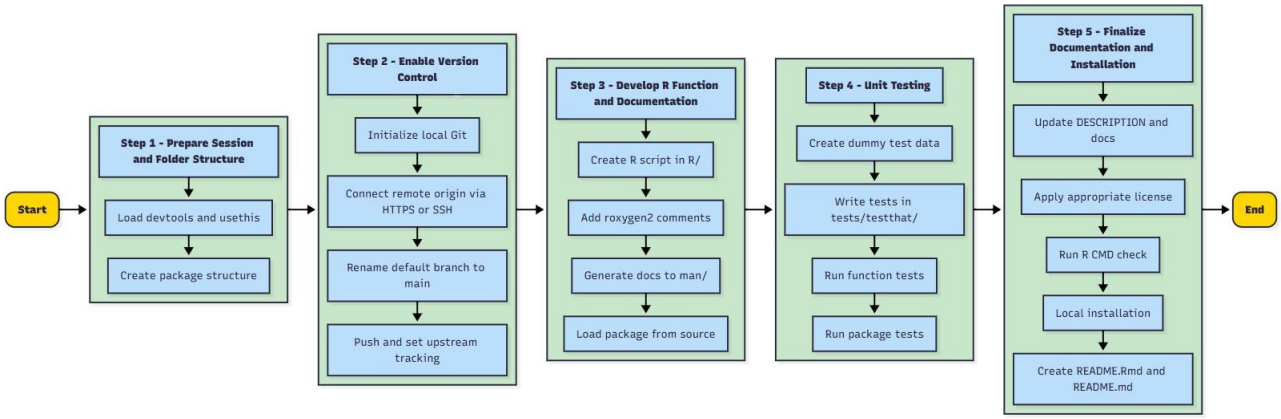
```
> usethis::use_readme_rmd()

 ✓ Writing README.Rmd.
 ✓ Adding "^README\\.Rmd$" to .Rbuildignore.
 ☐ Modify README.Rmd.
 ✓ Writing .git/hooks/pre-commit.
```

## VISUAL WORKFLOW SUMMARY (WORKFLOW DIAGRAMS)

The development process follows a structured sequence starting from package creation, function design, documentation, unit testing, and final validation. Each step contributes to producing a reliable and maintainable R package.

The following figures visually summarize the R package development workflow described in the steps above. These static diagrams are suitable for reinforce such as how individual steps connect across the lifecycle.

**Overall R Package Development Workflow**




The figure shows the complete progression from preparing the R session to producing a package that is ready for routine use. It corresponds to Steps 1–5 and emphasizes that package development is a structured process rather than ad hoc scripting.

## CONCLUSION

The paper demonstrated a clear and beginner-friendly approach to developing an R package, showing how simple R scripts can be transformed into a structured, reusable, and professional tool. Using the *dataverse* package as an example, the workflow highlighted essential practices such as defining scope, organizing code, documenting functions, implementing unit tests, and validating packages using modern development tools.

By following this step-by-step process, beginners can build confidence in package development while adopting best practices that improve reproducibility, maintainability, and collaboration. Overall, this guide provides a practical foundation for creating reliable R packages that support efficient and scalable analytical workflows.

## REFERENCES

[1] Wickham, H., & Bryan, J. (2023). R Packages: Organize, Test, Document, and Share Your Code (2nd ed.). O'Reilly Media.
[2] OpenAI. (2026). ChatGPT-5 (version 5.2) [Large language model]. OpenAI. https://chatgpt.com/

## ACKNOWLEDGMENTS

The authors would like to thank Amy Gillespie, Mukesh Patel, Naveen Kommuru and Shuo Geng for reviewing the paper and providing valuable suggestions.

## CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Tejaskumar Mehta
Company: Merck & Co., Inc., Rahway, NJ, USA
Email: tejaskumar.mehta@merck.com

Co-Author Name: Rikkin Parekh
Company: Merck & Co., Inc., Rahway, NJ, USA
Email: rikkin.parekh@merck.com