Testing Data Analysis Workflows in R

Ronald G. Thomas

2025-07-25

Figure 1: A stack of graduated wire-mesh sieves, coarse mesh on top and finer mesh below, beside a pile of gravel: data passing through successive layers of validation, coarse checks first, finer checks last

Testing data analysis workflows transforms fragile scripts into reliable research infrastructure.

Introduction

I did not really know how to systematically test a data analysis pipeline until I applied software engineering practices from testthat and assertr to my own research workflow. Data analysis testing presents challenges that differ fundamentally from traditional software testing: one must validate data quality, ensure computational reproducibility, and verify that analytical results are both correct and meaningful.

For years I relied on informal validation – checking that results "looked reasonable" and trusting that my code was correct because it ran without errors. These approaches failed to scale. When a collaborator's data arrived with unexpected column types, my pipeline broke silently. When a package update changed default behavior, my results shifted without warning.

This post presents the testing taxonomy developed for data analysis workflows: unit tests for individual functions, data validation tests for quality assurance, integration tests for full pipelines, and reproducibility tests for deterministic results. All examples use the Palmer Penguins dataset and are immediately applicable to one's own research code.

More formally, this post documents the testing discipline at the Project Compendium tier of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist. Post 52 positions the compendium tier (zzcollab and its predecessors) as the project-level reproducibility unit. Tests are the discipline that closes that unit's loop, in the sense that an analysis is only as reproducible as its tests exercise. This post is the testing companion to Reproducible Blog Posts with ZZCOLLAB (the compendium-tier keystone).

Motivations

- My research pipeline broke after a data pull introduced unexpected missing values in a column I assumed was complete.
- A collaborator could not reproduce my results because a package update changed the default random seed handling.
- I was writing the same validation checks repeatedly across projects and wanted a systematic, reusable approach.
- The gap between software engineering testing practices and data science workflows seemed unnecessarily wide.
- As Wilson et al. (2017) note, "An important aspect of this is to validate the code using software development practices that prevent errors and software testing methods that can help detect them when they occur."

Objectives

1. Distinguish between computational reproducibility (the pipeline works) and result correctness (the results are accurate) as separate testing goals.
2. Implement unit tests, data validation tests, integration tests, and reproducibility tests using testthat and assertr.
3. Organize test files following R package conventions and demonstrate how to run them from the terminal.
4. Set up continuous integration with GitHub Actions to automate testing on every push.

This learning process is documented here. Errors spotted or better approaches are always welcome.

Systematic testing adapts software engineering rigor to the specific needs of data analysis.

Prerequisites and Setup

Install the required packages:

```
install.packages(
  c("testthat", "assertr",
    "palmerpenguins", "dplyr",
    "ggplot2", "validate")
)
```

Load libraries:

```
library(testthat)
library(assertr)
library(palmerpenguins)
library(dplyr)
library(ggplot2)
```

Load the sample data:

```
data(penguins)
glimpse(penguins)
```

Background assumed: Familiarity with R and the tidyverse. No prior experience with testing frameworks is required.

What is Testing for Data Analysis?

Testing for data analysis is the practice of writing automated checks that verify an analysis pipeline produces correct, reproducible results. Unlike traditional software testing, which focuses on function inputs and outputs, data analysis testing must also address data quality, statistical properties, and computational determinism.

Think of it as quality control for research code. Just as a manufacturing process has inspection points that verify each component meets specifications, a data analysis pipeline has tests that verify each stage (data loading, cleaning, modeling, and reporting) produces expected results.

Testing serves two primary goals. Computational reproducibility means another researcher can use the same code and data to obtain identical results. Result correctness means the results generated by the code are accurate and meaningful. Both goals require different types of tests, and both are necessary for trustworthy research.

Getting Started

The Testing Taxonomy

Test Type	What It Checks	When to Use
Unit	Individual functions	Helper functions
Data validation	Data quality	After loading
Integration	Full pipeline	Before finalizing
Reproducibility	Same results	Random processes

Unit Tests

Unit tests verify that individual functions behave correctly with known inputs and expected outputs:

```
test_that("outlier detection works", {
  test_data <- c(1, 2, 3, 100, 4, 5)
  outliers <- detect_outliers(
    test_data, method = "iqr"
  )

  expect_equal(outliers, 100)
  expect_length(outliers, 1)
})
```

Data Validation Tests

Data validation tests ensure data meets quality requirements before analysis proceeds:

```
test_that("data meets quality standards", {
  data <- read.csv(
    "analysis/data/raw_data/penguins.csv"
  )

  expect_equal(ncol(data), 8)
  expect_true(
    all(
      c("species", "body_mass_g") %in%
        names(data)
    )
  )
  expect_true(
    all(data$body_mass_g > 0, na.rm = TRUE)
  )
  expect_true(
    all(data$body_mass_g < 10000, na.rm = TRUE)
  )
})
```

Integration Tests

Integration tests verify that pipeline components work together:

```
test_that("pipeline runs successfully", {
  expect_no_error({
    raw_data <- load_raw_data()
    clean_data <- clean_data(raw_data)
    model <- fit_model(clean_data)
    results <- generate_results(model)
  })

  expect_s3_class(results, "data.frame")
  expect_true(nrow(results) > 0)
})
```

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** (this post)
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

Reproducibility Tests

Reproducibility tests confirm that results are deterministic when using the same random seed:

```
test_that("bootstrap is reproducible", {
  set.seed(42)
  results1 <- bootstrap_analysis(
    data, n_boots = 1000
  )

  set.seed(42)
  results2 <- bootstrap_analysis(
    data, n_boots = 1000
  )

  expect_equal(
    results1$estimate, results2$estimate
  )
  expect_equal(
    results1$ci_lower, results2$ci_lower
  )
})
```

Figure 2: A stack of three graduated ceramic bowls nested by size with a wooden spoon resting across the largest: layered levels of testing, broad and shallow at the base, narrow and deep at the top

The testing pyramid for data analysis: unit tests form the broad base, integration tests provide pipeline coverage, and reproducibility tests ensure determinism.

The testthat Framework

Basics

The testthat package provides the foundation for testing in R. Tests follow the Arrange-Act-Assert pattern:

```
library(testthat)

test_that("description of test", {
  x <- c(1, 2, 3, 4, 5)
  result <- mean(x)
  expect_equal(result, 3)
})
```

Common Expectations

```
expect_equal(result, expected)
expect_identical(result, expected)
expect_true(condition)
expect_false(condition)
expect_type(x, "double")
expect_s3_class(model, "lm")
expect_error(bad_function())
expect_warning(risky_function())
expect_no_error(safe_function())
```

Test File Organization

```
tests/
  testthat.R
  testthat/
    helper-test-data.R
    test-data-loading.R
    test-data-cleaning.R
    test-models.R
    test-visualization.R
```

Test files must start with `test-` to be discovered by the test runner.

Helper Functions

Create reusable test utilities in `helper-*.R` files:

```
create_test_penguins <- function(n = 50) {
  data.frame(
    species = sample(
      c("Adelie", "Chinstrap", "Gentoo"),
      n, replace = TRUE
    ),
    bill_length_mm = rnorm(
      n, mean = 44, sd = 5
    ),
    body_mass_g = rnorm(
      n, mean = 4200, sd = 800
    )
  )
}

expect_valid_model <- function(model) {
  expect_s3_class(model, "lm")
  expect_true(length(coef(model)) > 0)
```

```
  expect_true(!any(is.na(coef(model))))
}
```

Running Tests

```
devtools::test()

testthat::test_file(
  "tests/testthat/test-data-cleaning.R"
)

testthat::test_dir(
  "tests/testthat", filter = "model"
)
```

Data Validation with assertr

Pipeline-Friendly Assertions

The assertr package integrates data validation into tidyverse pipelines:

```
library(assertr)

penguins |>
  verify(nrow(.) > 300) |>
  verify(ncol(.) == 8) |>
  assert(
    within_bounds(0, 10000), body_mass_g
  ) |>
  assert(
    in_set("Adelie", "Chinstrap", "Gentoo"),
    species
  ) |>
  insist(within_n_sds(3), bill_length_mm)
```

assertr Functions

- `verify()`: Check that a logical condition is TRUE.
- `assert()`: Check that a predicate holds for a column.
- `insist()`: Check that a predicate holds using row-wise computation.

Domain-Specific Validation

Create validation functions tailored to the data:

```
validate_penguin_data <- function(data) {
  test_that("penguin data validation", {
    required_cols <- c(
      "species", "island",
      "bill_length_mm", "bill_depth_mm",
      "flipper_length_mm", "body_mass_g",
      "sex", "year"
    )
    expect_true(
      all(required_cols %in% names(data))
    )

    valid_species <- c(
      "Adelie", "Chinstrap", "Gentoo"
    )
    expect_true(
      all(data$species %in% valid_species)
    )

    expect_true(
      all(data$bill_length_mm > 0,
          na.rm = TRUE)
    )
    expect_true(
      all(data$year >= 2007 &
          data$year <= 2009)
    )
  })
}
```

Reproducibility Testing

Seed Management

Always document and test seed usage:

```
test_that("cross-validation is reproducible", {
  data <- create_test_penguins(100)

  set.seed(123)
  cv1 <- perform_cv(data, folds = 5)
```

```
  set.seed(123)
  cv2 <- perform_cv(data, folds = 5)

  expect_equal(
    cv1$fold_assignments,
    cv2$fold_assignments
  )
  expect_equal(cv1$rmse, cv2$rmse)
})
```

Testing Against Known Results

Store expected values from verified runs:

```
test_that("regression coefficients match", {
  data(penguins, package = "palmerpenguins")
  clean_data <- na.omit(penguins)

  model <- lm(
    body_mass_g ~ flipper_length_mm,
    data = clean_data
  )
  coefs <- coef(model)

  expect_equal(
    coefs["(Intercept)"],
    -5780.83, tolerance = 0.1
  )
  expect_equal(
    coefs["flipper_length_mm"],
    49.69, tolerance = 0.01
  )
})
```

Package Version Testing

Track package versions to identify when results might change:

```
test_that("expected package versions", {
  expect_true(
    packageVersion("dplyr") >= "1.0.0"
  )
  expect_true(
    packageVersion("ggplot2") >= "3.4.0"
  )
```

```
  if (requireNamespace("renv", quietly = TRUE)) {
    status <- renv::status()
    expect_true(status$synchronized)
  }
})
```

Testing Analysis Scripts

Script Execution Tests

Verify that analysis scripts run without error:

```
test_that("analysis scripts run", {
  scripts <- c(
    "scripts/01_load_data.R",
    "scripts/02_clean_data.R",
    "scripts/03_fit_models.R",
    "scripts/04_create_figures.R"
  )

  for (script in scripts) {
    expect_true(file.exists(script))
    expect_no_error(
      source(script, local = new.env()),
      info = paste("Failed:", script)
    )
  }
})
```

Output Validation

Test that scripts produce expected output files:

```
test_that("scripts produce outputs", {
  source(
    "scripts/02_clean_data.R",
    local = new.env()
  )

  expect_true(file.exists(
    "analysis/data/derived_data/clean.rds"
  ))

  clean_data <- readRDS(
```

```
    "analysis/data/derived_data/clean.rds"
  )
  expect_true(nrow(clean_data) > 300)
  expect_false(any(is.na(clean_data)))
})
```

Continuous Integration

GitHub Actions for R

Create `.github/workflows/test-analysis.yml`:

```yaml
name: Test Analysis
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - uses: r-lib/actions/setup-r@v2
    - uses: r-lib/actions/setup-r-dependencies@v2

    - name: Run tests
      run: |
        testthat::test_dir('tests/testthat')
      shell: Rscript {0}

    - name: Data validation
      run: |
        source('tests/validate_data.R')
      shell: Rscript {0}
```

Tests run automatically on every push, catching problems before they reach collaborators.

Complete Example

Penguins Regression Testing

A complete test file for a regression analysis:

```r
test_that("body mass model properties", {
  data(penguins, package = "palmerpenguins")
  clean_data <- na.omit(penguins)

  model <- lm(
    body_mass_g ~ flipper_length_mm + species,
    data = clean_data
  )

  expect_s3_class(model, "lm")
  expect_equal(length(coef(model)), 4)

  r_squared <- summary(model)$r.squared
  expect_true(r_squared > 0.8)

  expect_true(
    abs(mean(resid(model))) < 1e-10
  )
})

test_that("model is reproducible", {
  data(penguins, package = "palmerpenguins")
  clean_data <- na.omit(penguins)

  model1 <- lm(
    body_mass_g ~ flipper_length_mm + species,
    data = clean_data
  )
  model2 <- lm(
    body_mass_g ~ flipper_length_mm + species,
    data = clean_data
  )

  expect_equal(coef(model1), coef(model2))
  expect_equal(
    summary(model1)$r.squared,
    summary(model2)$r.squared
  )
})
```

Things to Watch Out For

1. **Floating-point comparison requires tolerance.** Never use `expect_identical()` for numeric results. Use `expect_equal()` with an appropriate `tolerance` argument.
2. **Tests that depend on external data are fragile.** Tests that read from a database or API will fail when the source is unavailable. Use local test fixtures instead.

3. **Random seed scope is global.** Setting `set.seed()` in one test affects subsequent tests in the same session. Reset seeds explicitly in each test that uses randomness.
4. **Package updates change defaults silently.** A dplyr update that changes `summarise()` behavior will not cause a test failure unless the specific output values are tested, not just the output structure.
5. **Over-testing implementation details creates brittleness.** Test what the function should produce, not how it produces it. Testing internal variable names means refactoring breaks tests without changing functionality.

Figure 3: A testing dashboard showing green checkmarks for passing tests and a pipeline diagram

Automated testing provides continuous assurance that an analysis pipeline produces correct results.

Lessons Learned

Conceptual Understanding

- Testing data analysis requires addressing both computational reproducibility (the pipeline works) and result correctness (the results are accurate); these are distinct goals requiring different test types.
- Data validation tests are the most immediately valuable addition to an analysis workflow because data problems are the most common source of silent failures.
- Integration tests that run the full pipeline end-to-end catch interaction effects that unit tests miss.
- Reproducibility tests with fixed seeds provide a baseline against which future changes can be measured.

Technical Skills

- The testthat Arrange-Act-Assert pattern structures tests clearly and makes failures easy to diagnose.
- assertr's pipeline integration (`verify`, `assert`, `insist`) fits naturally into tidyverse workflows without requiring separate test files.

- Helper functions in `helper-*.R` files reduce test code duplication and centralize test data generation.
- GitHub Actions with `r-lib/actions/setup-r@v2` provides automated testing with minimal configuration.

Gotchas and Pitfalls

- `expect_equal()` with default tolerance may be too loose for some statistical comparisons; specify tolerance explicitly.
- Tests that modify global state (working directory, options, environment variables) can cause cascading failures in other tests.
- The assertr `insist()` function uses row-wise computation, which is slower than `assert()` for large datasets.
- Test discovery requires files to start with `test-`; a misnamed file will be silently ignored.

Limitations

- This post focuses on R-specific tools (testthat, assertr) and does not address testing in Python, Julia, or other data science languages.
- The examples use the Palmer Penguins dataset, which is small and well-behaved. Testing strategies for large, messy, real-world datasets require additional considerations.
- Continuous integration with GitHub Actions requires a public or paid private repository; self-hosted runners are an alternative.
- Data validation tests assume known data structure; they do not detect novel failure modes in previously unseen data.
- The testing taxonomy presented here is not exhaustive; performance testing, security testing, and accessibility testing are outside scope.
- assertr is not actively maintained at the same pace as testthat; consider the validate package as a more actively developed alternative.

Opportunities for Improvement

1. Implement property-based testing using the `quickcheck` or `hedgehog` R packages to generate random test inputs automatically.
2. Add snapshot testing with `testthat::expect_ snapshot()` to detect unexpected changes in complex output (tables, plots, reports).
3. Create a test template that can be copied into new analysis projects with pre-built data validation and reproducibility tests.
4. Integrate code coverage reporting using the `covr` package to identify untested code paths.
5. Develop a testing checklist specific to clinical research workflows, addressing regulatory requirements for validated analysis code.
6. Explore the `pointblank` package as a more modern alternative to assertr for data validation.

Wrapping Up

Testing data analysis workflows requires adapting traditional software testing practices to the specific challenges of data science. The combination of unit tests, data validation, integration tests, and reproducibility tests provides comprehensive coverage for research code.

What developing this testing approach demonstrated is that the investment pays off immediately. The first time a data validation test catches an unexpected column type before it propagates through the pipeline, the time spent writing tests is repaid. The first time a reproducibility test confirms that a collaborator's results match one's own, the value of systematic testing becomes undeniable.

Starting with data validation tests is recommended: they require the least effort and catch the most common failures. Unit tests for helper functions come next, then integration tests for the full pipeline. Reproducibility tests can follow once the foundation is solid.

In conclusion, four points merit emphasis. First, systematic testing ensures both computational reproducibility (the pipeline runs) and result correctness (the results are accurate), which are distinct goals requiring different test types. Second, testthat provides the structural foundation while assertr adds pipeline-friendly validation directly within tidyverse workflows. Third, data validation tests are the highest-value addition to any analysis workflow because data problems are the most common source of silent failures. Fourth, continuous integration via GitHub Actions automates testing on every code change, catching regressions before they reach collaborators.

See Also

Related posts:

- Blog Post Template: The ZZCOLLAB template includes a full test suite

Key resources:

- testthat Package: Official documentation
- assertr Package: Pipeline-friendly data validation
- R Packages (Wickham): Testing chapter from the R Packages book
- Good Enough Practices (Wilson et al., 2017): Testing in computational research

Reproducibility

All code in this post uses `eval: false` to prevent execution during rendering. To run the examples:

```
Rscript -e "
  library(testthat)
  library(assertr)
  library(palmerpenguins)
  testthat::test_local()
"
```

Project files:

```
testingfordataanalysisworkflow/
  analysis/report/index.qmd  (this post)
  tests/testthat/             (test files)
  analysis/media/images/      (hero, ambiance)
```

References:

- Wilson, G., Bryan, J., Cranston, K., Kitzes, J., Nederbragt, L., & Teal, T.K. (2017). Good enough practices in scientific computing. *PLOS Computational Biology*, 13(6), e1005510.
- Wickham, H. (2011). testthat: Get started with testing. *The R Journal*, 3(1), 5-10.

Let's Connect

- **GitHub:** rgt47
- **Twitter/X:** @rgt47
- **LinkedIn:** Ronald Glenn Thomas
- **Email:** rgtlab.org/contact

I would enjoy hearing from readers who:

- Spot an error or a better approach to any of the code in this post.
- Have suggestions for topics to cover.
- Want to discuss R programming, data science, or reproducible research.
- Have questions about anything in this tutorial.
- Simply want to say hello and connect.