Research Compendia for Full Reproducibility in R: An rrtools, renv, and Docker Strategy

R.G. Thomas

2025-06-18

This white paper presents a comprehensive approach to achieving reproducibility in R workflows by combining three powerful tools: rrtools for creating structured research compendia, renv for R package management, and Docker for containerizing the computing environment. The rrtools package provides a standardized research compendium structure, renv manages package dependencies, and Docker ensures consistent execution environments. Together, these tools create self-contained research compendia that run identically across different systems. The paper includes a practical case study demonstrating multi-developer collaborative workflows with clear governance roles, where a project maintainer manages the technical infrastructure while multiple contributors extend the research analysis.

Table of contents

Executive Summary

Reproducibility is key to conducting professional data analysis, yet in practice, achieving it consistently with R workflows can be quite challenging. R projects frequently break when transferred between computers due to mismatched R versions, package dependencies, or inconsistent project organization. This white paper describes a comprehensive approach to solving this problem by combining three powerful tools: **rrtools** for creating structured research compendia, **renv** for R package management, and **Docker** for containerizing the computing environment. Together, these tools ensure that an R workflow runs identically across different computers by providing standardized project structure, identical R packages and versions, consistent R versions, and the same operating system libraries as the original setup.

Motivation

Imagine you've written code that you want to share with a colleague. At first glance, this may seem like a straightforward task—simply share the files electronically. However, ensuring that your colleague can run the code without errors, and obtain the same results is often much more challenging than anticipated.

When sharing R code, several potential problems can arise that can lead to code that won't run or won't match your results:

- Different versions of R installed on each machine
- Mismatched R package versions
- Missing or mismatched system dependencies (like pandoc or LaTeX)
- Missing supplemental files referenced by the program (bibliography files, LaTeX preambles, datasets, images)
- Different R startup configurations (.Rprofile or .Renviron)
- Different Operating Systems (macOS, Windows, Linux, etc.)

A real-world scenario often unfolds like this:

1. You email your analysis files to your colleague, Joe
2. Joe attempts to run your analysis with the commands you provided
3. But R isn't installed on Joe's system
4. After installing R, Joe gets an error: "could not find function 'render' " since he doesn't have the `rmarkdown` package installed
5. Joe installs the rmarkdown package and runs the R command again
6. Now pandoc is missing
7. After installing pandoc, a required package, say `ggplot`, is missing
8. After installing `ggplot`, several external files are missing (e.g. bibliography, images)
9. And so on…

This cycle of troubleshooting can be time-consuming and frustrating. Even when the code eventually runs, there's no guarantee that Joe will get the same results that you did.

To ensure true reproducibility, your colleague should have a computing environment as similar to yours as possible. Given the dynamic nature of open source software, not to mention hardware and operating system differences, this can be difficult to achieve through manual installation and configuration.

The approach outlined in this white paper offers a more robust solution. Rather than sending standalone text files, with modest additional effort, you can provide a complete, containerized, hardware and OS independent environment that includes everything needed to run your analysis. With this approach, your colleague can run a simple command like:

```
docker run \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  rgt47/penguins_analysis
```

(The details of this docker command are explained below.)

This creates an identical R environment on their desktop, ready for them to run or modify your code with confidence that it will work as intended.

1 Introduction

1.1 The Challenge of Reproducibility in R

R has become a standard tool for data science and statistical analysis across numerous scientific disciplines. However, as R projects grow in complexity, they often develop complex webs of dependencies that can make sharing and reproducing analyses difficult. Some common challenges include:

- Different R versions across machines
- Incompatible package versions
- Missing system-level dependencies
- Operating system differences (macOS vs. Windows vs. Linux)
- Conflicts with other installed packages
- R startup files (.Rprofile, .Renviron, .RData) that can affect code behavior

These challenges often manifest as the frustrating "it works on my machine" problem, where analysis code runs perfectly for the original author but fails when others attempt to use it. This undermines the scientific and collaborative potential of R-based analyses.

1.2 A Three-Level Solution

To address these challenges comprehensively, we need to tackle reproducibility at three distinct levels:

1. **Project-level reproducibility**: Ensuring consistent project structure and organization using research compendium standards
2. **Package-level reproducibility**: Ensuring exact package versions and dependencies are maintained
3. **System-level reproducibility**: Guaranteeing consistent R versions, operating system, and system libraries

The strategy presented in this white paper leverages **rrtools** for project-level structure, **renv** for package-level consistency, and **Docker** for system-level consistency. When combined, they provide a robust framework for end-to-end reproducible R workflows with proper research compendium organization.

2 rrtools: Project-Level Reproducibility

2.1 What is rrtools?

rrtools is an R package developed by Ben Marwick that provides instructions, templates, and functions for creating research compendia suitable for reproducible research. A research compendium is a standard and easily recognizable way of organizing the digital materials of a research project to enable others to inspect, reproduce, and extend the research.

2.2 Key Features of rrtools

rrtools creates a structured research compendium that follows established conventions:

- **Standardized directory structure**: Creates organized folders for data, analysis, papers, and figures following research compendium best practices
- **R package framework**: Uses R package structure to leverage existing tools for dependency management, documentation, and testing
- **Integrated documentation**: Automatically generates README files, citation information, and licensing documentation
- **Docker integration**: Provides functions to create Dockerfiles specifically designed for research compendia

- **Publication-ready structure**: Creates templates for academic papers and reports using R Markdown/Quarto

2.3 Basic rrtools Workflow

The typical workflow with rrtools involves:

```
# Install rrtools (if not already installed)
if (!require("rrtools", quietly = TRUE)) {
  devtools::install_github("benmarwick/rrtools")
}

# Create a new research compendium
rrtools::use_compendium("myproject")

# Add license
usethis::use_mit_license(copyright_holder = "Your Name")

# Create README and project documentation
rrtools::use_readme_qmd()

# Set up analysis directory structure
rrtools::use_analysis()

# Create Docker configuration
rrtools::use_dockerfile()

# Initialize package dependency management
renv::init()
```

This creates a complete research compendium with standardized structure that other researchers can easily understand and reproduce.

2.4 Research Compendium Structure

rrtools creates the following standardized directory structure:

```
myproject/
   DESCRIPTION          # Package metadata and dependencies
   LICENSE             # Project license
   README.qmd          # Project documentation
```

```
myproject.Rproj     # RStudio project file
renv.lock          # Package dependency lockfile
Dockerfile         # Container specification
.github/           # GitHub Actions for CI/CD
R/                 # R functions and utilities
data/              # Raw and processed data
analysis/          # Analysis scripts and notebooks
    paper/         # Manuscript and figures
    figures/       # Generated plots and charts
    data/          # Analysis-specific data
    templates/     # Document templates
tests/             # Unit tests and validation
```

This structure separates data, methods, and outputs while making relationships between them clear, following established research compendium principles.

3 renv: Package-Level Reproducibility

3.1 What is renv?

renv (Reproducible Environment) is an R package designed to create isolated, project-specific library environments. Instead of relying on a shared system-wide R library that might change over time, renv gives each project its own separate collection of packages with specific versions.

3.2 Key Features of renv

- **Isolated project library**: renv creates a project-specific library (typically in `renv/library`) containing only the packages used by that project. This isolation ensures that updates or changes to packages in one project won't affect others.
- **Lockfile for dependencies**: When you finish installing or updating packages, `renv::snapshot()` produces a `renv.lock` file - a JSON document listing each package and its exact version and source. This lockfile is designed to be committed to version control and shared.
- **Environment restoration**: On a new machine (or when reproducing past results), `renv::restore()` installs the exact versions of packages specified in the lockfile. This creates an R package environment identical to the one that created the lockfile, provided the same R version is available. The R version is important since critical components of the R system, such as random number generation, and default factor handling policy vary between versions.

3.3 Basic renv Workflow

The typical workflow with renv involves:

```
# One-time installation of renv
install.packages("renv")

# Initialize renv for the project
renv::init()  # Creates renv infrastructure

# Install project-specific packages
# ...

# Save the package state to renv.lock
renv::snapshot()

# Later or on another system...
renv::restore()  # Restore packages from renv.lock
```

While renv effectively handles package dependencies, it does not address differences in R versions or system libraries. This limitation is where Docker becomes essential.

4 Docker: System-Level Reproducibility

4.1 What is Docker?

Docker is a platform that allows you to package software into standardized units called containers. A Docker container is like a lightweight virtual machine that includes everything needed to run an application: the code, runtime, system tools, libraries, and settings.

4.2 Docker's Role in Reproducibility

While renv handles R packages, Docker ensures consistency for:

- **Operating system**: The specific Linux distribution or OS version
- **R interpreter**: The exact R version
- **System libraries**: Required C/C++ libraries and other dependencies
- **Computational environment**: Memory limits, CPU configuration, etc.
- **External tools**: pandoc, LaTeX, and other utilities needed for R Markdown

By running an R Markdown project in Docker, you eliminate differences in OS or R installation as potential sources of irreproducibility. Any machine running Docker will execute the container in an identical environment.

4.3 Docker Components for R Workflows

For R-based projects, a typical Docker approach involves:

1. **Base image**: Starting from a pre-configured R image (e.g., from the Rocker project)
2. **Dependencies**: Adding system and R package dependencies
3. **Configuration**: Setting working directories and environment variables
4. **Content**: Adding project files
5. **Execution**: Defining how the project should run

A simple Dockerfile for an R Markdown project might look like:

```
# Use R 4.3.0 on Linux as base image
FROM rocker/r-ver:4.3.0

# Set the working directory inside the container
WORKDIR /workspace

# Install renv and restore dependencies
RUN R -e "install.packages('renv', \
    repos='https://cloud.r-project.org')"

# Copy renv lockfile and infrastructure
COPY renv.lock renv/activate.R /workspace/

# Restore the R package environment
RUN R -e "renv::restore()"

# Default command when container runs
CMD ["/bin/bash"]
```

A more comprehensive Dockerfile that includes additional tools and user setup, following Docker best practices, might look like:

```
FROM rocker/r-ver:4.5.0

# Prevent interactive prompts during package installation
ENV DEBIAN_FRONTEND=noninteractive
```

```
# Install system dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    pandoc \
    vim \
    git \
    curl \
    fonts-dejavu \
    && apt-get clean && rm -rf /var/lib/apt/lists/*

# Install base R packages
RUN Rscript -e 'install.packages(c("tinytex", "rmarkdown", "renv"), repos = "https://cloud.r

# Install TinyTeX (minimal TeX)
RUN Rscript -e 'tinytex::install_tinytex()'

# Create non-root user
ARG USERNAME=analyst
RUN useradd --create-home --shell /bin/bash ${USERNAME}

# Set environment variable for user R library path
ENV R_LIBS_USER=/home/${USERNAME}/R/library

# Create user library directory with correct ownership
RUN mkdir -p /home/${USERNAME}/R/library && chown -R ${USERNAME}:${USERNAME} /home/${USERNAM

# Set working directory
WORKDIR /home/${USERNAME}

# Copy renv lockfile and infrastructure
COPY --chown=${USERNAME}:${USERNAME} renv.lock ./
COPY --chown=${USERNAME}:${USERNAME} renv/activate.R ./renv/

# Switch to non-root user
USER ${USERNAME}

# Run renv::restore with explicit library path to avoid permission errors
RUN Rscript -e '.libPaths(Sys.getenv("R_LIBS_USER")); renv::restore()'

# Default command
CMD ["/bin/bash"]
```

This Dockerfile follows Docker best practices by: - Using a specific R version for reproducibility

- Combining related commands to minimize layers - Creating a non-root user without sudo privileges for security - Using efficient package installation with cleanup - Making the username configurable via build arguments

5 Combining rrtools, renv, and Docker: A Comprehensive Approach

5.1 Why Use All Three?

Using any single tool improves reproducibility, but combining all three provides the most comprehensive solution:

- **rrtools** provides standardized project structure and research compendium organization
- **renv** guarantees the R packages and their versions
- **Docker** guarantees the OS and R version
- **Together** they achieve end-to-end reproducibility from project organization through package dependencies to operating system consistency

This comprehensive approach creates a fully portable, well-organized research compendium that can be shared and will produce identical results across different computers while following established research best practices.

5.2 Integration Strategy with Governance Model

The recommended workflow integrates rrtools, renv, and Docker with a clear governance structure suitable for multi-developer research teams:

Project Maintainer Role (Developer 1): - Creates and maintains the research compendium structure - Manages renv environment and package dependencies
- Updates and maintains Docker images - Reviews and approves contributor changes

Contributor Role (Other Developers): - Fork the research compendium for their contributions - Add analysis content, papers, and documentation - Propose new package dependencies through contributions - Submit changes via pull requests

Workflow Steps:

1. **Initialize Research Compendium (Maintainer)**:
    - Create standardized project structure with `rrtools::use_compendium()`
    - Set up analysis directories with `rrtools::use_analysis()`
    - Initialize renv environment with `renv::init()`
    - Create Dockerfile with `rrtools::use_dockerfile()`

2. **Establish Development Environment (Maintainer)**:

   - Install required packages and develop initial analysis
   - Create comprehensive tests for analytical functions
   - Use `renv::snapshot()` to create initial lockfile
   - Build and test Docker image locally

3. **Maintain Infrastructure (Maintainer)**:

   - Review contributor pull requests for package additions
   - Update `renv.lock` by selectively incorporating new dependencies
   - Rebuild Docker images when system dependencies change
   - Push updated images to container registry (Docker Hub, GitHub Container Registry)

4. **Collaborative Development (All Developers)**:

   Research Compendium Files in GitHub Repository:

   - **Project Structure**: `DESCRIPTION`, `LICENSE`, `README.qmd` (rrtools-generated)
   - **Analysis Content**: Files in `analysis/paper/` directory (R Markdown manuscripts)
   - **Dependencies**: `renv.lock` (managed by maintainer), `renv/activate.R`
   - **Infrastructure**: `Dockerfile` (maintained by project maintainer)
   - **Code**: `R/` directory (utility functions), `tests/` directory
   - **Documentation**: Generated README files and project documentation
   - **Configuration**: `.gitignore`, `.github/` (CI/CD workflows)

   Sharing the Docker image:

```
# Build the image
docker build -t rgt47/penguins_analysis:v1 .

# Push to Docker Hub (after docker login)
docker push rgt47/penguins_analysis:v1

# Alternative: Push to GitHub Container Registry
docker tag rgt47/penguins_analysis:v1 ghcr.io/rgt47/penguins_analysis:v1
docker push ghcr.io/rgt47/penguins_analysis:v1
```

   Docker run commands for collaborators:

   Basic usage (read-only analysis):

```
docker run --rm -it -v "$(pwd):/home/analyst/workspace" \
  rgt47/penguins_analysis:v1
```

Interactive development with output directory:

```
docker run --rm -it \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  rgt47/penguins_analysis:v1
```

Running a specific R Markdown analysis:

```
docker run --rm \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  rgt47/penguins_analysis:v1 \
  R -e "rmarkdown::render('analysis/paper/paper.Rmd')"
```

Command explanation:

- `--rm`: Automatically remove container when it exits
- `-it`: Interactive terminal (allows user input)
- `-v "$(pwd):/home/analyst/workspace"`: Mount research compendium to container's workspace
- `-v "$(pwd)/analysis/figures:/home/analyst/output"`: Mount figures directory for outputs
- `username/project-name:v1.0`: The Docker image to run

5. **Execute consistently**:

   - Run analyses in the Docker container for guaranteed reproducibility
   - Use volume mounts to access local files while maintaining environment consistency
   - Run tests within the container to verify functionality

This strategy ensures that your R Markdown documents and analyses will run identically for anyone who has access to your Docker container, regardless of their local setup.

6 Practical Example: Collaborative Research Compendium Development with Testing

The following case study demonstrates how two developers can collaborate on a research compendium using rrtools, renv, and Docker to ensure reproducibility, with integrated testing procedures to maintain code quality.

6.1 Project Scenario

Two data scientists are collaborating on an analysis of the Palmer Penguins dataset using the governance model established earlier. Developer 1 (project maintainer) will set up the initial research compendium structure using rrtools and create a basic analysis. Developer 2 (contributor) will extend the analysis with additional visualizations and propose new package dependencies. They'll use GitHub for version control and DockerHub to share the containerized environment.

Key Governance Points: - Developer 1 manages the renv environment and Docker images - Developer 2 contributes through pull requests from their fork - Package dependency changes require Developer 1's approval and integration - Both developers use the standardized rrtools research compendium structure

6.2 Step-by-Step Implementation

6.3 Developer 1: Project Setup and Initial Analysis

Step 1: Create and Initialize the GitHub Repository

Developer 1 creates a new GitHub repository called "penguins_analysis" and clones it locally.

Repository Visibility Decision:

For this example, we use a **public repository** because: - The Palmer Penguins dataset is publicly available with no sensitive information - This serves as a reproducible research demonstration that others can learn from - Public repos integrate seamlessly with Docker Hub for automated builds - It aligns with open science principles for educational content

When to use private repositories: - **Proprietary data**: Working with company data, customer information, or licensed datasets - **Sensitive research**: Medical data, personally identifiable information, or classified research - **Commercial projects**: Business analyses, competitive intelligence, or trade secrets - **Early development**: Preliminary research before public release or peer review - **Institutional requirements**: When organization policies mandate private repositories - **Collaborative restrictions**: When only specific team members should have access

Best practice: Start with a private repository during development, then make it public when ready to share, ensuring no sensitive information is accidentally exposed in the git history.

Step 2: Create Research Compendium with rrtools

Developer 1 opens R in a terminal and creates a structured research compendium:

```r
# Install rrtools if not already installed
if (!require("rrtools", quietly = TRUE)) {
  if (!require("devtools")) install.packages("devtools")
  devtools::install_github("benmarwick/rrtools")
}

# Create the research compendium structure
rrtools::use_compendium("penguins_analysis",
                        path = ".",
                        open = FALSE)

# Add MIT license
usethis::use_mit_license(copyright_holder = "Developer 1")

# Create README and documentation
rrtools::use_readme_qmd()

# Set up analysis directory structure
rrtools::use_analysis(location = "analysis",
                      data_in_git = FALSE)

# Initialize renv for package management
renv::init()
```

This creates the complete research compendium structure with standardized directories for data, analysis, papers, and documentation.

Step 3: Install Required Packages and Initialize Environment

Developer 1 installs the packages needed for the Palmer Penguins analysis:

```r
# Install required packages
options(repos = c(CRAN = "https://cloud.r-project.org"))

# Core analysis packages
install.packages("ggplot2")
install.packages("palmerpenguins")

# R Markdown rendering packages
install.packages("rmarkdown")
install.packages("knitr")

# Development and testing packages
```

```r
install.packages("testthat")
install.packages("devtools")

# Save package versions to renv.lock
renv::snapshot()

# Update DESCRIPTION file with dependencies
rrtools::add_dependencies_to_description()
```

Step 4: Create Initial Analysis Paper

Developer 1 creates the analysis in the research compendium structure by editing `analysis/paper/paper.Rmd`:

````
---
title: "Palmer Penguins Analysis"
author: "Developer 1"
date: "`r Sys.Date()`"
output: pdf_document
---

```{r setup1, include=FALSE}
library(ggplot2)
library(palmerpenguins)
```

# Flipper Length vs. Bill Length

```{r flipper-bill-plot1}
ggplot(palmerpenguins::penguins,
aes(x = flipper_length_mm, y = bill_length_mm)) +
  geom_point() +
  theme_minimal() +
  ggtitle("Flipper Length vs. Bill Length")
```
````

Step 5: Create Tests for Analysis Functions

While testing is uncommon in many data analysis projects, it provides significant value for reproducible research:

Why Test Data Analysis Code? - Data integrity validation: Ensure datasets have expected structure, ranges, and completeness - **Catch silent errors**: Detect when data changes

break assumptions (e.g., missing columns, unexpected NA patterns) - **Collaboration confidence**: New team members can verify their environment setup works correctly - **Refactoring safety**: Safely improve code knowing core functionality still works - **Publication standards**: Many journals increasingly expect computational reproducibility verification - **Debugging efficiency**: Isolate whether issues stem from environment, data, or analysis logic

Types of Tests for Data Analysis: - **Data validation**: Verify data structure and content meet expectations - **Statistical sanity checks**: Ensure results fall within reasonable ranges - **Regression tests**: Confirm outputs remain consistent across environment changes - **Integration tests**: Verify the full analysis pipeline executes successfully

The Iterative Testing Process:

Testing data analysis code follows an iterative development cycle that builds confidence progressively:

1. **Start Simple**: Begin with basic data availability and structure tests that verify your dataset loads correctly and has expected dimensions. These catch fundamental setup issues early.
2. **Build Systematically**: Add tests for data types, column existence, and value ranges. Each test validates one assumption your analysis depends on.
3. **Test Incrementally**: As you develop new analysis functions, write corresponding tests before moving to the next feature. This "test-first" mindset catches issues immediately rather than during final verification.
4. **Validate Continuously**: Run tests frequently during development—after each major change, before commits, and when switching between environments. The Docker+renv setup makes this consistent across machines.
5. **Expand Coverage**: Once basic functionality works, add edge case tests, statistical validation tests, and integration tests that verify the complete analysis pipeline.

Beyond Basic Testing: The comprehensive test suite provided in the Appendix demonstrates advanced testing strategies that can be adapted for any data analysis project. These tests cover data validation, statistical relationships, visualization functions, and complete pipeline integration. Consider implementing similar comprehensive testing as your project matures, particularly for: - Long-term research projects requiring ongoing validation - Collaborative analyses where multiple team members contribute code - Production analytical pipelines that process data regularly - Academic publications where methodological rigor is essential

Developer 1 creates a test directory structure and initial tests:

```
mkdir -p tests/testthat
```

Then creates a file `tests/testthat.R`:

```
library(testthat)
library(palmerpenguins)

# Run all tests in the testthat directory
test_dir("tests/testthat")
```

And a test file `tests/testthat/test-data-integrity.R`:

```
library(testthat)
library(palmerpenguins)

test_that("penguins data is available and has expected dimensions", {
  expect_true(exists("penguins", where = "package:palmerpenguins"))
  expect_equal(ncol(palmerpenguins::penguins), 8)
  expect_gt(nrow(palmerpenguins::penguins), 300)
})

test_that("penguins data has required columns", {
  expect_true("species" %in% names(palmerpenguins::penguins))
  expect_true("bill_length_mm" %in% names(palmerpenguins::penguins))
  expect_true("flipper_length_mm" %in% names(palmerpenguins::penguins))
  expect_true("body_mass_g" %in% names(palmerpenguins::penguins))
})
```

Step 6: Create a .gitignore file

A critical aspect of reproducible projects is understanding **what should and shouldn't be tracked in version control**. Not all files created during development need to be shared—in fact, including too many files can create confusion and bloat the repository.

Files that SHOULD be tracked (committed to Git): - **Source code**: `*.R`, `*.Rmd` files containing your analysis - **Dependency specifications**: `renv.lock` (exact package versions), `renv/activate.R` (renv setup) - **Infrastructure**: `Dockerfile`, `README.md`, `.gitignore` - **Tests**: All files in `tests/` directory that validate your analysis - **Configuration**: Any custom configuration files your analysis depends on - **Documentation**: Project documentation, methodology notes

Files that should NOT be tracked (excluded via .gitignore): - **Generated outputs**: PDFs, HTML files, plots—these are products of your code, not source materials - **Large package libraries**: `renv/library/` contains downloaded packages that can be recreated from `renv.lock` - **Temporary files**: R session data, cache files, intermediate processing files

- **Personal settings**: User-specific R configurations, local environment variables - **System artifacts**: OS-specific files, editor backup files

The principle: Track the "recipe" (code + dependencies), not the "meal" (outputs). Collaborators should run your code to generate outputs, not download pre-generated results.

Developer 1 creates a `.gitignore` file to exclude unnecessary files:

```
# renv - exclude downloaded packages but keep configuration
renv/library/          # Downloaded packages (recreated from renv.lock)
renv/local/           # Local package cache
renv/cellar/          # Package storage
renv/lock/            # Lock file backups
renv/python/          # Python environments
renv/staging/         # Temporary package staging

# R session files - personal and temporary
.Rhistory             # Command history (user-specific)
.RData                # Saved workspace (should start fresh)
.Ruserdata            # User session data

# Generated output files - recreated by running code
*.html                # Rendered R Markdown HTML
*.pdf                 # Rendered R Markdown PDF
*.docx                # Rendered R Markdown Word docs
output/               # Directory for analysis outputs
figures/              # Generated plots and charts
cache/                # Computation cache files

# System and editor files
.DS_Store             # macOS system files
Thumbs.db             # Windows thumbnail cache
*.tmp                 # Temporary files
*~                    # Editor backup files
```

Repository size consideration: This approach keeps the Git repository lightweight and focused. The `renv/library/` directory alone can contain hundreds of megabytes of downloaded packages, but collaborators can recreate this exactly using `renv::restore()` from the small `renv.lock` file.

Step 7: Create a Dockerfile

Developer 1 creates a Dockerfile following Docker best practices that excludes the R Markdown file to ensure that collaborators' local files are used:

```
FROM rocker/r-ver:4.3.0

# Install system dependencies in a single layer
RUN apt-get update && \
    apt-get install -y --no-install-recommends \
        pandoc \
        vim \
        git && \
    apt-get clean && \
    rm -rf /var/lib/apt/lists/*

# Install renv
RUN Rscript -e 'install.packages("renv", repos="https://cloud.r-project.org")'

# Create non-root user
ARG USERNAME=analyst
RUN useradd --create-home --shell /bin/bash ${USERNAME}

# Set working directory
WORKDIR /home/${USERNAME}

# Copy renv files and change ownership
COPY --chown=${USERNAME}:${USERNAME} renv.lock ./
COPY --chown=${USERNAME}:${USERNAME} renv/activate.R ./renv/

# Switch to non-root user
USER ${USERNAME}

# Restore R packages
RUN Rscript -e 'renv::restore()'

# Create output and test directories
RUN mkdir -p output tests/testthat

CMD ["/bin/bash"]
```

Step 8: Build and Push the Docker Image

```
# Build with platform specification for compatibility
docker build -t rgt47/penguins_analysis:v1 . --platform=linux/amd64
docker login
docker push rgt47/penguins_analysis:v1
```

Step 9: Run tests before committing

Developer 1 runs the tests to make sure everything is working correctly:

```
# Run all tests in the testthat directory
R -e "testthat::test_dir('tests/testthat')"
```

Step 10: Commit and Push to GitHub

After confirming the tests pass, Developer 1 commits the project files:

```
git add .
git commit -m "Initial renv setup, Docker environment, and tests"
git push origin main
```

Step 11: Communicate with Developer 2

Developer 1 provides these instructions to Developer 2:

1. Fork the research compendium repository on GitHub
2. Pull the prebuilt Docker image from DockerHub
3. Run the container interactively, mounting the local repository
4. Create a new branch for feature development
5. Extend the analysis in `analysis/paper/paper.Rmd`
6. Document any new package needs (Developer 1 will manage renv updates)
7. Write tests for new functionality
8. Run tests to verify changes
9. Push changes to their fork and create a pull request

Important: Developer 2 cannot directly modify `renv.lock` or update Docker images. Package dependency changes must be proposed through pull requests and will be managed by Developer 1.

6.4 Developer 2: Extending the Analysis

Step 1: Fork the Repository

Since Developer 2 has their own GitHub account (let's assume it's `dev2_github`) and the repository is public, they can fork it without needing an invitation from Developer 1:

1. Navigate to `https://github.com/rgt47/penguins_analysis` in a web browser
2. Click the "Fork" button in the top-right corner
3. Select `dev2_github` as the destination account
4. This creates `https://github.com/dev2_github/penguins_analysis`

Note: This fork-based approach doesn't require Developer 1 to invite Developer 2 as a collaborator. Developer 2 can contribute through pull requests from their fork. Alternatively, Developer 1 could invite Developer 2 as a direct collaborator (Settings → Manage access → Invite collaborator), which would allow Developer 2 to push branches directly to the original repository, but this is not necessary for the workflow described here.

Step 2: Clone the Forked Repository and Pull the Docker Image

Developer 2 clones their own fork (not the original repository):

```
# Clone the forked repository
git clone https://github.com/dev2_github/penguins_analysis.git
cd penguins_analysis

# Add the original repository as an upstream remote for future updates
git remote add upstream https://github.com/rgt47/penguins_analysis.git

# Pull the Docker image
docker pull rgt47/penguins_analysis:v1
```

Step 3: Create a Feature Branch

```
git branch body-mass-analysis
git checkout body-mass-analysis
```

Step 4: Run Docker Interactively

Developer 2 runs the container with the local repository mounted:

```
docker run --rm -it \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/output:/home/analyst/output" \
  -w /home/analyst/workspace \
  rgt47/penguins_analysis:v1 /bin/bash
```

This approach: - Uses the renv-restored environment from the container - Mounts the local directory to `/home/analyst/workspace` in the container - Creates a shared output directory for generated files - Allows Developer 2 to access and modify files directly from their local machine

Step 5: Extend the Analysis

Developer 2 modifies `analysis/paper/paper.Rmd` to add a second plot for body mass vs. bill length:

````
---
title: "Palmer Penguins Analysis"
author: "Collaborative Research Team"
date: "`r Sys.Date()`"
output: pdf_document
---

```{r setup, include=FALSE}
library(ggplot2)
library(palmerpenguins)
# Note: If additional packages needed (e.g., plotly, DT),
# document them in pull request for Developer 1 to add
```

# Flipper Length vs. Bill Length

```{r flipper-bill-plot}
ggplot(palmerpenguins::penguins,
       aes(x = flipper_length_mm, y = bill_length_mm)) +
  geom_point() +
  theme_minimal() +
  ggtitle("Flipper Length vs. Bill Length")
```

# Body Mass vs. Bill Length

```{r mass-bill-plot}
# Developer 2's contribution: Additional analysis
ggplot(palmerpenguins::penguins,
       aes(x = body_mass_g, y = bill_length_mm, color = species)) +
  geom_point() +
  theme_minimal() +
  ggtitle("Body Mass vs. Bill Length by Species")
```
````

Step 6: Create Tests for New Analysis

Developer 2 adds a new test file `tests/testthat/test-body-mass-analysis.R`:

```
test_that("body mass data is valid", {
  expect_true(all(palmerpenguins::penguins$body_mass_g > 0, na.rm = TRUE))
  expect_true(is.numeric(palmerpenguins::penguins$body_mass_g))
})
```

```r
test_that("body mass correlates with bill length", {
  # Calculate correlation coefficient
  correlation <- cor(
    palmerpenguins::penguins$body_mass_g,
    palmerpenguins::penguins$bill_length_mm,
    use = "complete.obs"
  )

  # Verify correlation is a numeric value (not NA)
  expect_true(!is.na(correlation))

  # Test that the correlation is positive
  expect_true(correlation > 0)
})
```

Step 7: Run Tests to Verify Changes

Before committing, Developer 2 runs the tests to ensure that the new code doesn't break existing functionality and that the new analyses are working correctly:

```
R -e "testthat::test_dir('tests/testthat')"
```

Step 8: Commit and Push Changes to Their Fork

After confirming all tests pass, Developer 2 commits and pushes the changes to their forked repository:

```
git add analysis/paper/paper.Rmd tests/testthat/test-body-mass-analysis.R
git commit -m "Added body mass vs. bill length analysis with tests"

# Push to their own fork (origin), not the original repository
git push origin body-mass-analysis
```

Step 9: Create a Cross-Repository Pull Request

Developer 2 now creates a pull request from their fork to the original repository:

1. Navigate to their fork: `https://github.com/dev2_github/penguins_analysis`
2. GitHub will typically show a banner suggesting to create a pull request after pushing a new branch
3. Click "Compare & pull request" or navigate to the Pull Requests tab and click "New pull request"

4. Ensure the pull request is configured as:
    - **Base repository**: `rgt47/penguins_analysis` (the original)
    - **Base branch**: `main`
    - **Head repository**: `dev2_github/penguins_analysis` (their fork)
    - **Compare branch**: `body-mass-analysis`
5. Add a descriptive title: "Add body mass vs. bill length analysis"
6. In the description, include:

```
# Changes Made
- Added body mass vs. bill length scatter plot analysis
- Created comprehensive tests for body mass data validation
- Verified all existing tests continue to pass

# Testing
- All tests pass in the containerized environment
- New correlation analysis validates expected positive relationship

# Docker Environment
- Tested using `rgt47/penguins_analysis:v1` image
- No additional dependencies required
```

Important Notes for Cross-Repository Pull Requests: - Developer 2 cannot push directly to Developer 1's repository (unless given collaborator access) - The pull request allows Developer 1 to review changes before merging - This workflow maintains clear ownership and permission boundaries - It follows standard open-source contribution patterns

Step 10: Code Review and Merge by Developer 1

Developer 1 receives the pull request notification and conducts a thorough review:

Review Process: 1. **Examine the pull request on GitHub**: Review the code changes, commit messages, and description 2. **Test the changes locally**: Developer 1 can test the changes without affecting their main branch:

```
# Fetch the pull request to a local branch for testing
git fetch origin
git checkout -b review-body-mass-analysis
git pull https://github.com/dev2_github/penguins_analysis.git body-mass-analysis
```

3. **Verify in the Docker environment**: Test that all analyses work correctly using an interactive session:

```
# Start interactive container for comprehensive testing
docker run --rm -it \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  -w /home/analyst/workspace \
  rgt47/penguins_analysis:v1 bash

# Inside the container, run all verification steps:
# Run tests to ensure nothing breaks
R -e "testthat::test_dir('tests/testthat')"

# Test R Markdown rendering
R -e "rmarkdown::render('analysis/paper/paper.Rmd',
       output_dir='analysis/figures')"

# Verify renv environment status
R -e "renv::status()"

# Exit container when done
exit
```

4. **Manage package dependencies (if needed)**: If Developer 2 has documented new package requirements in their pull request, use an interactive session for package management:

```
# Interactive session for package management
docker run --rm -it \
  -v "$(pwd):/home/analyst/workspace" \
  -w /home/analyst/workspace \
  rgt47/penguins_analysis:v1 bash

# Inside container - install packages and update lockfile
R -e "
  # Install any new packages requested by contributors
  install.packages('new_package')  # Replace with actual package

  # Update the lockfile
  renv::snapshot()

  # Test that everything still works
  testthat::test_dir('tests/testthat')
"
```

```
# Exit when done
exit
```

5. **Review and merge**: If everything passes, Developer 1 merges the pull request through the GitHub interface

Post-Merge Steps: Developer 1 updates their local repository and cleans up:

```
# Switch to main branch and pull the merged changes
git checkout main
git pull origin main

# Delete the temporary review branch
git branch -d review-body-mass-analysis

# Run final verification using interactive session
docker run --rm -it \
  -v "$(pwd):/home/analyst/workspace" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  -w /home/analyst/workspace \
  rgt47/penguins_analysis:v1 bash

# Inside container - comprehensive final verification
R -e "
  testthat::test_dir('tests/testthat')
  rmarkdown::render('analysis/paper/paper.Rmd',
                    output_dir='analysis/figures')
  renv::status()
"

# Exit container
exit
```

Optional: Update Docker Image If the collaboration continues with more contributors, Developer 1 might consider updating the Docker image version to include any new system dependencies or optimizations:

```
# Build and push updated image (if needed)
docker build -t rgt47/penguins_analysis:v1.1 .
docker push rgt47/penguins_analysis:v1.1
```

Benefits of This Fork-Based Workflow: - **Security**: Developer 1 maintains control over the main repository - **Quality**: All changes go through review process before merging - **Traceability**: Clear history of who made what changes and when - **Scalability**: Multiple developers can work simultaneously on different features - **Standard practice**: Follows conventional open-source collaboration patterns

7 Continuous Integration Extension

To further enhance the workflow, the team should set up GitHub Actions for continuous integration. This automatically runs tests in the Docker environment whenever changes are pushed, ensuring code quality and catching issues early in the development process.

7.1 Understanding Continuous Integration for Research Compendia

What is CI and why use it?

Continuous Integration (CI) is a practice where code changes are automatically tested every time they're submitted to the repository. For research compendia, this means:

- **Automated testing**: Every push or pull request triggers your test suite automatically
- **Environment consistency**: Tests run in the same Docker environment across all machines
- **Early error detection**: Problems are caught immediately, not weeks later when reproducing results
- **Collaboration confidence**: Team members can see if their changes break existing functionality

How GitHub Actions works:

GitHub Actions is a CI/CD service built into GitHub. You create workflow files (written in YAML) that define what should happen when certain events occur (like pushes or pull requests). These workflows run on GitHub's servers, not your local machine.

7.2 Step-by-Step CI Setup

Step 1: Create the Workflow Directory

Every GitHub repository can have automated workflows. These are defined in YAML files stored in a special `.github/workflows/` directory. Developer 1 creates this structure:

```
# Create the directory structure for GitHub Actions
mkdir -p .github/workflows
```

This directory tells GitHub "look here for automation instructions."

Step 2: Create a Basic CI Workflow

Create a file called `.github/workflows/ci.yml` in your repository. This file contains instructions for GitHub's automation system:

```
# .github/workflows/ci.yml
# This file tells GitHub Actions what to do when code changes
name: Test Research Compendium

# When should this workflow run?
on:
  push:
    branches: [ main ]        # Run when pushing to main branch
  pull_request:
    branches: [ main ]        # Run when someone creates a pull request

# What jobs should be executed?
jobs:
  test:
    name: Run Tests in Docker
    runs-on: ubuntu-latest    # Use GitHub's Ubuntu servers

    steps:
    # Step 1: Download the repository code
    - name: Checkout code
      uses: actions/checkout@v4

    # Step 2: Pull our pre-built Docker image
    - name: Pull Docker image
      run: docker pull rgt47/penguins_analysis:v1

    # Step 3: Run tests inside the Docker container
    - name: Run tests
      run: |
        docker run --rm \
          -v "${{ github.workspace }}:/home/analyst/workspace" \
          -w /home/analyst/workspace \
          rgt47/penguins_analysis:v1 \
          R -e "testthat::test_dir('tests/testthat')"

    # Step 4: Test that R Markdown can render
```

```
    - name: Test R Markdown rendering
      run: |
        docker run --rm \
          -v "${{ github.workspace }}:/home/analyst/workspace" \
          -w /home/analyst/workspace \
          rgt47/penguins_analysis:v1 \
          R -e "rmarkdown::render('analysis/paper/paper.Rmd')"
```

What this workflow does:

- **Triggers**: Runs automatically when code is pushed to the main branch or when pull requests are created
- **Environment**: Uses the same Docker image that developers use locally, ensuring consistency
- **Tests**: Runs the test suite to verify data integrity and analysis functions
- **Validation**: Confirms that the R Markdown document can render successfully

Step 3: Add the CI File to Your Repository

```
# Add the CI configuration to version control
git add .github/workflows/ci.yml
git commit -m "Add basic CI workflow for automated testing"
git push origin main
```

After pushing, visit your GitHub repository and click the "Actions" tab to see the workflow running.

7.3 How CI Improves the Collaborative Workflow

Before CI: When Developer 2 submits a pull request, Developer 1 manually tests everything locally before merging.

With CI: When Developer 2 submits a pull request, GitHub automatically: 1. Downloads the proposed changes 2. Runs all tests in the Docker environment 3. Reports pass/fail status directly in the pull request 4. Prevents merging if tests fail (optional but recommended)

This means Developer 1 can see immediately whether proposed changes break anything, without manual testing.

7.4 Benefits for Research Reproducibility

1. **Consistent testing environment**: Every test runs in the identical Docker container
2. **Comprehensive validation**: Tests run on every change, catching regressions early
3. **Documentation of working state**: The CI history shows when the analysis last worked correctly
4. **Collaboration confidence**: Team members can contribute knowing their changes are automatically validated

7.5 Key Benefits Demonstrated in This Example

This collaborative workflow demonstrates several advantages of the rrtools + renv + Docker approach with integrated testing:

1. **Dependency consistency**: Both developers work with identical R package versions thanks to renv.
2. **Environment consistency**: The Docker container ensures the same R version and system libraries.
3. **Code quality**: Automated tests verify that the code works as expected and catches regressions.
4. **Research compendium structure**: rrtools provides standardized organization that other researchers can easily understand.
5. **Separation of concerns**: Analysis documents remain outside the Docker image, allowing for easier collaboration.
6. **Workflow flexibility**: Developer 2 can work in the container while editing files locally.
7. **Full reproducibility**: The entire research compendium environment is captured and shareable.
8. **Continuous integration**: Automated testing ensures ongoing code quality.

8 Best Practices and Considerations

8.1 When to Use This Approach

The rrtools + renv + Docker approach with testing is particularly valuable for:

- **Long-term research projects** where reproducibility over time is crucial
- **Collaborative analyses** with multiple contributors on different systems
- **Production analytical pipelines** that need to run consistently
- **Academic publications** where methods must be reproducible
- **Teaching and education** to ensure consistent student experiences
- **Complex analyses** that require rigorous testing to validate results

8.2 Tips for Efficient Implementation

1. **Keep Docker images minimal**: Include only what's necessary for reproducibility.
2. **Use specific version tags**: For both R packages and Docker base images, specify exact versions.
3. **Document system requirements**: Include notes on RAM and storage requirements.
4. **Leverage bind mounts**: Mount local directories to containers for easier development.
5. **Write meaningful tests**: Focus on validating both data integrity and analytical results.
6. **Automate testing**: Use CI/CD pipelines to automatically run tests on every change.
7. **Consider computational requirements**: Particularly for resource-intensive analyses.

8.3 Testing Strategies for R Analyses

Testing data analysis code differs from traditional software testing but provides crucial value for reproducible research:

1. **Data Validation Tests**: Ensure data has the expected structure, types, and values.
2. **Function Tests**: Verify that custom functions work as expected with known inputs and outputs.
3. **Edge Case Tests**: Check how code handles missing values, outliers, or unexpected inputs.
4. **Integration Tests**: Confirm that different parts of the analysis work correctly together.
5. **Regression Tests**: Make sure new changes don't break existing functionality.
6. **Output Validation**: Verify that final results match expected patterns or benchmarks.

While uncommon in traditional data analysis, these tests catch silent errors, validate assumptions, and provide confidence that analyses remain correct as code and data evolve.

8.4 Potential Challenges

Some challenges to be aware of:

- **Docker image size**: Images with many packages can become large
- **Learning curve**: Docker, renv, and testing frameworks require some initial learning
- **System-specific features**: Some analyses may rely on hardware features
- **Performance considerations**: Containers may have different performance characteristics
- **Test maintenance**: Tests need to be updated as the analysis evolves

9 Conclusion

Achieving full reproducibility in R requires addressing project organization, package dependencies, and system-level consistency, while ensuring code quality through testing. By combining rrtools for research compendium structure, renv for R package management, Docker for environment containerization, and automated testing for code validation, data scientists and researchers can create truly portable, reproducible, and reliable workflows.

The comprehensive approach presented in this white paper ensures that the common frustration of "it works on my machine" becomes a thing of the past. Instead, research compendia become easy to share and fully reproducible. A collaborator or reviewer can launch the Docker container and get identical results, without worrying about package versions, system setup, or project organization.

The case study demonstrates how two developers can effectively collaborate on an analysis while maintaining reproducibility and code quality throughout the project lifecycle. By integrating testing into the workflow, the team can be confident that their analysis is not only reproducible but also correct.

This strategy represents a best practice for long-term reproducibility in R, meeting the high standards required for professional data science and research documentation. The combination of standardized research compendium structure, rigorous dependency management, and containerized environments creates a robust foundation for reproducible research. By adopting this comprehensive approach, the R community can make significant strides toward the goal of fully reproducible and reliable research and analysis.

10 References

1. Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging data analytical work reproducibly using R (and friends). *The American Statistician*, 72(1), 80-88.
2. Marwick, B. (2017). rrtools: Creates a Reproducible Research Compendium. R package version 0.1.6. https://github.com/benmarwick/rrtools
3. Ushey, K., Wickham, H., & RStudio. (2023). renv: Project Environments. R package. https://rstudio.github.io/renv/
4. The Rocker Project. (2023). Docker containers for the R environment. https://www.rocker-project.org/
5. Wickham, H. (2023). testthat: Unit Testing for R. https://testthat.r-lib.org/
6. Horst, A.M., Hill, A.P., & Gorman, K.B. (2020). palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R package version 0.1.0. https://allisonhorst.github.io/palmerpenguins/
7. Marwick, B. (2016). Computational reproducibility in archaeological research: Basic principles and a case study of their implementation. *Journal of Archaeological Method and Theory*, 24(2), 424-473.

11 Appendix: Comprehensive Test Suite for Palmer Penguins Analysis

This appendix provides a complete set of tests that can be used to validate the Palmer Penguins analysis. These tests demonstrate best practices for data analysis testing and can be adapted for other projects.

11.1 Test File: `tests/testthat/test-comprehensive-analysis.R`

```
library(testthat)
library(palmerpenguins)
library(ggplot2)

# Test 1: Data Availability and Basic Structure
# Generic application: Verify your primary dataset loads correctly and has
# expected dimensions
# Catches: Package loading issues, file path problems, corrupted data files
test_that("Palmer Penguins dataset is available and has correct structure", {
  expect_true(exists("penguins", where = "package:palmerpenguins"))
  expect_s3_class(palmerpenguins::penguins, "data.frame")
  expect_equal(ncol(palmerpenguins::penguins), 8)  # Adapt: Set expected column count
  expect_gt(nrow(palmerpenguins::penguins), 300)   # Adapt: Set minimum row threshold
  expect_equal(nrow(palmerpenguins::penguins), 344)  # Adapt: Set exact expected
                                                      # count if known
})

# Test 2: Required Columns Exist with Correct Types
# Generic application: Ensure your analysis depends on columns that actually
# exist with correct types
# Catches: Column name changes, type coercion issues, CSV import problems
test_that("Dataset contains required columns with expected data types", {
  df <- palmerpenguins::penguins

  # Check column existence - Adapt: List columns your analysis requires
  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(df)))

  # Check data types - Adapt: Verify types match your analysis expectations
  expect_type(df$species, "integer")  # Factor stored as integer
```

```
  expect_type(df$bill_length_mm, "double")  # Continuous measurements
  expect_type(df$flipper_length_mm, "integer")  # Discrete measurements
  expect_type(df$body_mass_g, "integer")  # Integer measurements
})

# Test 3: Categorical Variables Have Expected Levels
# Generic application: Verify factor levels for categorical variables used in
# analysis
# Catches: Missing categories, typos in factor levels, data encoding issues
test_that("Species factor has expected levels", {
  species_levels <- levels(palmerpenguins::penguins$species)
  expected_species <- c("Adelie", "Chinstrap", "Gentoo")  # Adapt: Your expected
                                                          # categories
  expect_equal(sort(species_levels), sort(expected_species))
  expect_equal(length(species_levels), 3)  # Adapt: Expected number of categories
  # For other datasets: Test treatment groups, regions, product types, etc.
})

# Test 4: Data Value Ranges are Domain-Reasonable
# Generic application: Verify numeric values fall within realistic ranges for
# your domain
# Catches: Data entry errors, unit conversion mistakes, outliers from
# measurement errors
test_that("Measurement values fall within reasonable biological ranges", {
  df <- palmerpenguins::penguins

  # Bill length - Adapt: Set realistic bounds for your numeric variables
  bill_lengths <- df$bill_length_mm[!is.na(df$bill_length_mm)]
  expect_true(all(bill_lengths >= 30 & bill_lengths <= 70))  # Penguin-specific
                                                             # range

  # Flipper length - Examples for other domains:
  flipper_lengths <- df$flipper_length_mm[!is.na(df$flipper_length_mm)]
  expect_true(all(flipper_lengths >= 150 & flipper_lengths <= 250))
  # Finance: stock prices > 0, percentages 0-100
  # Health: age 0-120, BMI 10-80, blood pressure 50-300
  # Engineering: temperatures -273+°C, pressures > 0

  # Body mass
  body_masses <- df$body_mass_g[!is.na(df$body_mass_g)]
  expect_true(all(body_masses >= 2000 & body_masses <= 7000))
})
```

```r
# Test 5: Missing Data Patterns are as Expected
# Generic application: Verify missingness patterns match your data collection
# expectations
# Catches: Unexpected data loss, systematic missingness, data pipeline failures
test_that("Missing data follows expected patterns", {
  df <- palmerpenguins::penguins

  # Total missing values should be manageable
  total_na <- sum(is.na(df))
  expect_lt(total_na, nrow(df))  # Adapt: Set acceptable threshold for missing
                                 # data

  # Some variables may have expected missingness
  expect_gt(sum(is.na(df$sex)), 0)  # Sex determination sometimes difficult
  # Adapt examples: Optional survey questions, historical data gaps, sensor
  # failures

  # Critical variables should be complete
  expect_equal(sum(is.na(df$species)), 0)  # Primary identifier must be complete
  # Adapt: ID columns, primary keys, required fields should have no NAs
})

# Test 6: Expected Statistical Relationships Hold
# Generic application: Test known relationships between variables in your domain
# Catches: Data corruption, encoding errors, units mix-ups that break known
# patterns
test_that("Expected correlations between measurements exist", {
  df <- palmerpenguins::penguins

  # Test strong expected relationships
  correlation <- cor(df$flipper_length_mm, df$body_mass_g,
                     use = "complete.obs")
  expect_gt(correlation, 0.8)  # Strong positive correlation expected
  # Adapt examples: height vs weight, price vs quality, experience vs salary

  # Test weaker but expected relationships
  bill_cor <- cor(df$bill_length_mm, df$bill_depth_mm, use = "complete.obs")
  expect_gt(abs(bill_cor), 0.1)  # Some relationship should exist
  # Adapt: Education vs income, advertising vs sales, temperature vs energy use
})

# Test 7: Visualization Functions Work Correctly
```

```r
# Generic application: Ensure your key plots and visualizations can be
# generated
# Catches: Missing aesthetic mappings, incompatible data types, package conflicts
test_that("Basic plots can be generated without errors", {
  df <- palmerpenguins::penguins

  # Test basic plot creation without errors
  expect_no_error({
    p1 <- ggplot(df, aes(x = flipper_length_mm, y = bill_length_mm)) +
      geom_point() +
      theme_minimal()
  })
  # Adapt: Test your key plot types - histograms, boxplots, time series, etc.

  # Test that plot object is properly created
  p1 <- ggplot(df, aes(x = flipper_length_mm, y = bill_length_mm)) +
    geom_point()
  expect_s3_class(p1, "ggplot")  # Adapt: Check for your plotting framework objects
})

# Test 8: Data Filtering and Subsetting Work Correctly
# Generic application: Verify data manipulation operations produce expected results
# Catches: Logic errors in filtering, unexpected factor behaviors, indexing mistakes
test_that("Data can be properly filtered and subsetted", {
  df <- palmerpenguins::penguins

  # Test categorical filtering
  adelie_penguins <- df[df$species == "Adelie" & !is.na(df$species), ]
  expect_gt(nrow(adelie_penguins), 100)  # Adapt: Expected subset size
  expect_true(all(adelie_penguins$species == "Adelie", na.rm = TRUE))
  # Adapt: Filter by treatment groups, regions, time periods, etc.

  # Test missing data handling
  complete_cases <- df[complete.cases(df), ]
  expect_lt(nrow(complete_cases), nrow(df))  # Some rows should be removed
  expect_equal(sum(is.na(complete_cases)), 0)  # No NAs remaining
  # Adapt: Test your specific data cleaning operations
})

# Test 9: Summary Statistics are Reasonable
# Generic application: Verify computed statistics match domain knowledge
# expectations
```

```r
# Catches: Calculation errors, unit mistakes, algorithm bugs, extreme outliers
test_that("Summary statistics fall within expected ranges", {
  df <- palmerpenguins::penguins

  # Test means fall within expected ranges
  mean_flipper <- mean(df$flipper_length_mm, na.rm = TRUE)
  expect_gt(mean_flipper, 190)  # Adapt: Set realistic bounds for your variables
  expect_lt(mean_flipper, 210)
  # Examples: Average customer age 20-80, mean salary $30k-200k, etc.

  # Test other central tendencies
  mean_mass <- mean(df$body_mass_g, na.rm = TRUE)
  expect_gt(mean_mass, 4000)
  expect_lt(mean_mass, 5000)

  # Test variability measures are reasonable
  sd_flipper <- sd(df$flipper_length_mm, na.rm = TRUE)
  expect_gt(sd_flipper, 5)   # Not zero variance
  expect_lt(sd_flipper, 30)  # Not excessive variance
  # Adapt: CV should be <50%, SD should be meaningful relative to mean
})

# Test 10: Complete Analysis Pipeline Integration Test
# Generic application: Test your entire analysis workflow runs without errors
# Catches: Pipeline breaks, dependency issues, function interaction problems
test_that("Complete analysis pipeline executes successfully", {
  df <- palmerpenguins::penguins

  # Test that full workflow executes without errors
  expect_no_error({
    # Data preparation step
    clean_df <- df[complete.cases(df[c("flipper_length_mm", "bill_length_mm")]), ]

    # Statistical analysis step - Adapt: Your key analyses
    correlation_result <- cor.test(clean_df$flipper_length_mm,
                                   clean_df$bill_length_mm)

    # Visualization step - Adapt: Your key plots
    plot_result <- ggplot(clean_df,
                          aes(x = flipper_length_mm, y = bill_length_mm)) +
      geom_point() +
      geom_smooth(method = "lm") +
```

```
      theme_minimal() +
      labs(title = "Flipper Length vs. Bill Length",
           x = "Flipper Length (mm)",
           y = "Bill Length (mm)")
  })
  # Adapt: Add model fitting, prediction, reporting steps as needed

  # Verify analysis produces meaningful results
  clean_df <- df[complete.cases(df[c("flipper_length_mm", "bill_length_mm")]), ]
  correlation_result <- cor.test(clean_df$flipper_length_mm,
                                 clean_df$bill_length_mm)
  expect_lt(correlation_result$p.value, 0.05)  # Significant result expected
  # Adapt: Check model R², prediction accuracy, convergence, etc.
})
```

11.2 Running the Tests

To run all tests in your project:

```
# Run all tests
testthat::test_dir("tests/testthat")

# Run specific test file
testthat::test_file("tests/testthat/test-comprehensive-analysis.R")

# Run tests with detailed output
testthat::test_dir("tests/testthat", reporter = "detailed")
```

11.3 Test Categories Explained

Data Validation Tests (1-5): Verify data structure, types, ranges, and missing patterns **Statistical Tests (6)**: Confirm expected relationships in the data **Functional Tests (7-8)**: Ensure analysis functions work correctly **Sanity Tests (9)**: Check that summary statistics are reasonable **Integration Tests (10)**: Verify the complete analysis pipeline works end-to-end

These tests provide comprehensive coverage for a data analysis project and can catch issues ranging from data corruption to environment setup problems.