Simple Process for Achieving Full Reproducibility in R: A Docker and renv Strategy

R.G. Thomas

2025-06-17

This white paper presents a comprehensive approach to achieving reproducibility in R workflows by combining two powerful tools: renv for R package management and Docker for containerizing the computing environment. Together, these tools ensure that an R workflow runs identically across different systems with the same packages, R version, and system libraries as the original setup. The paper includes a practical case study demonstrating collaborative development using this approach, with an emphasis on testing procedures to maintain code quality.

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 or package dependencies. This white paper describes a comprehensive approach to solving this problem by combining two powerful tools: **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 assuring the code is run with the same R packages, the same package versions, same 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: Any of the following can lead to code that won't run or won't match the results you've gotten: - 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 Rmarkdown file, say `peng1.Rmd`, to your colleague, Joe
2. Joe attempts to run it with the command you give him: `R -e "source('peng1.Rmd')"`
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 "$(pwd):/home/analyst" -v "$(pwd)/output:/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.

0.1 Introduction

0.1.1 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.

0.1.2 A Two-Level Solution

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

1. **Package-level reproducibility**: Ensuring exact package versions and dependencies are maintained
2. **System-level reproducibility**: Guaranteeing consistent R versions, operating system, and system libraries

The strategy presented in this white paper leverages **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.

0.2 renv: Package-Level Reproducibility

0.2.1 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.

0.2.2 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.

0.3 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.

0.4 Docker: System-Level Reproducibility

0.4.1 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.

0.4.2 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.

0.4.3 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 4.1.0 on Linux as base image
FROM rocker/r-ver:4.1.0

# 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:4.3.0

# 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"]
```

This improved 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

0.5 Combining renv and Docker: A Comprehensive Approach

0.5.1 Why Use Both?

Using renv or Docker alone improves reproducibility, but combining them provides the most comprehensive solution:

- **Docker** guarantees the OS and R version
- **renv** guarantees the R packages and their versions
- **Together** they achieve end-to-end reproducibility from operating system to package dependencies

This combined approach creates a fully portable analytical environment that can be shared and will produce identical results across different computers.

0.5.2 Integration Strategy

The recommended workflow integrates renv and Docker in the following manner:

1. **Develop locally with renv**:
    - Initialize your R project with `renv::init()`
    - Install required packages and develop your analysis
    - Create tests for your analytical functions
2. **Snapshot dependencies**:
    - Use `renv::snapshot()` to create a lockfile
    - Commit both your code and `renv.lock` to version control

3. **Containerize with Docker**:

   - Create a Dockerfile that specifies the R version and incorporates the renv lockfile
   - Build and test the Docker image locally
   - Push the image to a container registry (Docker Hub, GitHub Container Registry)

4. **Share and collaborate**:

   Files to push to GitHub:

   - `*.R` and `*.Rmd` files (your analysis code)
   - `renv.lock` (package dependency lockfile)
   - `renv/activate.R` (renv activation script)
   - `Dockerfile` (container specification)
   - `README.md` (setup and usage instructions)
   - `tests/` directory (if using automated testing)
   - `.gitignore` (exclude `renv/library/` and other temporary files)

   Sharing the Docker image:

```
# Build the image
docker build -t username/project-name:v1.0 .

# Push to Docker Hub (after docker login)
docker push username/project-name:v1.0

# Alternative: Push to GitHub Container Registry
docker tag username/project-name:v1.0 ghcr.io/username/project-name:v1.0
docker push ghcr.io/username/project-name:v1.0
```

   Docker run commands for collaborators:

   Basic usage (read-only analysis):

```
docker run --rm -it -v "$(pwd):/workspace" \
  username/project-name:v1.0
```

   Interactive development with output directory:

```
docker run --rm -it \
  -v "$(pwd):/workspace" \
  -v "$(pwd)/output:/home/analyst/output" \
  username/project-name:v1.0
```

   Running a specific R script:

```
docker run --rm \
  -v "$(pwd):/workspace" \
  -v "$(pwd)/output:/home/analyst/output" \
  username/project-name:v1.0 \
  Rscript analysis.R
```

Command explanation:

- `--rm`: Automatically remove container when it exits
- `-it`: Interactive terminal (allows user input)
- `-v "$(pwd):/workspace"`: Mount current directory to container's /workspace
- `-v "$(pwd)/output:/home/analyst/output"`: Mount local output folder
- `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.

0.6 Practical Example: Collaborative R Markdown Development with Testing

The following case study demonstrates how two developers can collaborate on an R Markdown project using renv and Docker to ensure reproducibility, with integrated testing procedures to maintain code quality.

0.6.1 Project Scenario

Two data scientists are collaborating on an analysis of the Palmer Penguins dataset. Developer 1 will set up the initial project structure and create a basic analysis. Developer 2 will extend the analysis with additional visualizations. They'll use GitHub for version control and DockerHub to share the containerized environment. Both will implement testing to ensure code quality before merging changes.

0.6.2 Step-by-Step Implementation

0.6.2.1 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.

```
git clone https://github.com/rgt47/penguins_analysis.git
cd penguins_analysis
```

Step 2: Initialize renv and Install Required Packages

Developer 1 opens R in a terminal (by typing `R` at the command prompt) and runs these commands interactively:

```
# Initialize renv for the project
install.packages("renv")  # If not already installed
renv::init()  # Initialize renv for the project

# Install required packages
options(
  install.packages.check.source = "no",  # Don't ask about source vs binary
  repos = c(CRAN = "https://cloud.r-project.org")
)
```

```r
# Core analysis packages
install.packages("ggplot2", quiet = TRUE)
install.packages("palmerpenguins", quiet = TRUE)

# R Markdown rendering packages
install.packages("rmarkdown", quiet = TRUE)
install.packages("knitr", quiet = TRUE)

# Development and testing packages
install.packages("testthat", quiet = TRUE)
install.packages("devtools", quiet = TRUE)

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

This creates the necessary renv infrastructure, installs all required packages, and creates the `renv.lock` file with exact package versions. Exit R with `quit()` or `q()` when finished.

Step 3: Create Initial R Markdown Analysis

Developer 1 creates a file named `peng1.Rmd` with the following content:

````
---
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 4: 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`:

```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 5: 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 6: 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 7: Build and Push the Docker Image

```
docker build -t rgt47/penguins_analysis:v1 .
docker login
docker push rgt47/penguins_analysis:v1
```

Step 8: 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 9: 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 10: Communicate with Developer 2

Developer 1 provides these instructions to Developer 2:

1. Clone the GitHub repository
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 the peng1.Rmd file
6. Write tests for new functionality
7. Run tests to verify changes
8. Push changes back to GitHub
9. Create a pull request

0.6.2.2 Developer 2: Extending the Analysis

Step 1: Clone the Repository and Pull the Docker Image

```
git clone https://github.com/rgt47/penguins_analysis.git
cd penguins_analysis
docker pull rgt47/penguins_analysis:v1
```

Step 2: Create a Feature Branch

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

Step 3: 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 4: Extend the Analysis

Developer 2 modifies `peng1.Rmd` to add a second plot for body mass vs. bill length:

````
---
title: "Palmer Penguins Analysis"
author: "Developer 2"
date: "`r Sys.Date()`"
output: pdf_document
---

```{r setup2, include=FALSE}
library(ggplot2)
library(palmerpenguins)
```

## Flipper Length vs. Bill Length

```{r flipper-bill-plot2}
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}
ggplot(palmerpenguins::penguins,
       aes(x = body_mass_g, y = bill_length_mm)) +
  geom_point() +
  theme_minimal() +
  ggtitle("Body Mass vs. Bill Length")
```
````

Step 5: Create Tests for New Analysis

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

```r
context("Body Mass Analysis")

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))
})

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 6: 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
R -e "testthat::test_dir('tests/testthat')"
```

Step 7: Commit and Push Changes Back to GitHub

After confirming all tests pass, Developer 2 commits and pushes the changes:

```r
git add peng1.Rmd tests/testthat/test-body-mass-analysis.R
git commit -m "Added body mass vs. bill length analysis with tests"
git push origin body-mass-analysis
```

Step 8: Create a Pull Request

Developer 2 creates a pull request on GitHub from the `body-mass-analysis` branch to `main`, describing the changes made and noting that all tests pass.

Step 9: Code Review and Merge

Developer 1 reviews the changes, checks that the tests pass in the container environment, and merges the pull request if everything looks good:

```
git checkout main
git pull
docker run --rm \
  -v "$(pwd):/home/analyst/workspace" \
  -w /home/analyst/workspace \
  rgt47/penguins_analysis:v1 R -e "testthat::test_dir('tests/testthat')"
```

0.6.3 Continuous Integration Extension

To further enhance the workflow, the team could set up GitHub Actions for continuous integration, which would automatically run tests in the Docker environment whenever changes are pushed:

```
# .github/workflows/r-test.yml
name: R Tests

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest
    container:
      image: rgt47/penguins_analysis:v1

    steps:
    - uses: actions/checkout@v2

    - name: Run tests
      run: |
        R -e "testthat::test_dir('tests/testthat')"
```

0.6.4 Key Benefits Demonstrated in This Example

This collaborative workflow demonstrates several advantages of the 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. **Separation of concerns**: The R Markdown document remains outside the Docker image, allowing for easier collaboration.
5. **Workflow flexibility**: Developer 2 can work in the container while editing files locally.
6. **Full reproducibility**: The entire analysis environment is captured and shareable.
7. **Continuous integration**: Automated testing ensures ongoing code quality.

0.7 Best Practices and Considerations

0.7.1 When to Use This Approach

The 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

0.7.2 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.

0.7.3 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.

0.7.4 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

0.8 Conclusion

Achieving full reproducibility in R requires addressing both package dependencies and system-level consistency, while ensuring code quality through testing. By combining 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, R Markdown projects 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 or system setup.

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. By adopting this comprehensive approach, the R community can make significant strides toward the goal of fully reproducible and reliable research and analysis.

0.9 References

1. Thomas, R.G. "Docker and renv strategy."
2. "Palmer Penguins Analysis."
3. The Rocker Project. https://www.rocker-project.org/
4. renv documentation. https://rstudio.github.io/renv/
5. testthat documentation. https://testthat.r-lib.org/
6. Horst, A.M., Hill, A.P., & Gorman, K.B. (2022). palmerpenguins: Palmer Archipelago (Antarctica) penguin data. R Journal.

0.10 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.

0.10.1 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
```

```r
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))
})

# 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
})
```

```r
# 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
# 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
```

```r
# 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
# 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.
})
```

0.10.2 Running the Tests

To run all tests in your project:

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

```r
# 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")
```

0.10.3 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.

0.11 Prerequisites

In development

0.12 Step-by-Step Implementation

In development

0.13 Key Takeaways

In development

0.14 Further Reading

In development