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

R.G. Thomas

2025-06-27

This white paper presents a comprehensive approach to achieving reproducibility in R workflows by combining three powerful tools: zzrrtools for creating structured research compendia, renv for R package management, and Docker for containerizing the computing environment. The zzrrtools framework 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: **zzrrtools** 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 "$(pwd):/home/analyst/project" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  ghcr.io/joe/penguins_analysis:v1.0
```

(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.

The Docker command shown above represents the end goal of this white paper's approach, but achieving this level of reproducibility requires understanding and implementing three complementary technologies. The following sections provide a comprehensive framework for building such reproducible environments from the ground up.

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 **zzrrtools** 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.

With this three-level framework established, we can now examine how each tool addresses its specific layer of reproducibility. We begin with zzrrtools, which tackles the foundational challenge of project-level organization and provides the structural framework upon which package and system-level reproducibility can be built.

2 zzrrtools: Project-Level Reproducibility

2.1 What is zzrrtools?

zzrrtools is an enhanced framework built upon Ben Marwick's rrtools 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 zzrrtools

zzrrtools 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 zzrrtools Workflow with renv Consistency Checking

The zzrrtools workflow using the custom setup script involves:

```
# Run the rrtools setup script in your project directory
~/prj/zzrrtools/zzrrtools.sh
```

This automated script creates a comprehensive research compendium that includes:

- **Enhanced R package structure** with proper DESCRIPTION, NAMESPACE, and documentation
- **Comprehensive directory organization** with data, analysis, scripts, and documentation folders
- **Automated renv setup** with a curated list of commonly-used R packages

- **renv consistency checking** with automated dependency validation before commits
- **Docker integration** using rocker/r-ver with TinyTeX support
- **GitHub Actions workflows** for automated testing, checking, and paper rendering
- **Make-based build system** supporting both native R and Docker workflows
- **Symbolic links** for easy navigation between directories

The script automatically organizes existing files and creates a professional research compendium structure that follows best practices for reproducible research.

2.3.1 Integrated renv Consistency Checking

The workflow includes automated renv management through the `check_renv_for_commit.R` script, which ensures package dependency consistency before committing to version control. This script validates dependencies across code files, DESCRIPTION, and renv.lock, providing automatic fixes and CI/CD integration.

Basic usage:

```
# Check dependencies before commit
Rscript check_renv_for_commit.R

# Auto-fix issues for CI/CD
Rscript check_renv_for_commit.R --fix --fail-on-issues
```

This automated approach ensures collaborators can reliably reproduce package environments and CI/CD pipelines have all necessary dependency information.

2.4 Research Compendium Structure

The zzrrtools setup creates a comprehensive directory structure that follows research compendium best practices. The structure includes organized data folders, analysis directories, testing frameworks, and automated workflows.

Key organizational principles:

- **Data management**: Separate folders for raw, derived, and external data with proper documentation
- **Analysis workflow**: Dedicated spaces for papers, figures, tables, and working scripts
- **Package structure**: Professional R package organization with documentation and testing
- **Automation support**: Integration with Docker, GitHub Actions, and build systems

This organizational framework provides the foundation for reproducible research while supporting team collaboration and automated workflows.

2.5 Iterative Development Workflow

For collaborative analysis development, the research compendium structure supports a phased approach that balances rapid iteration with publication-quality outputs:

2.5.1 Phase 1: Exploration & Development (`scripts/`)

During active analysis development, Joe and Sam should work primarily in the `scripts/` directory:

```
scripts/
  01_data_exploration.R
  02_penguin_correlations.R
  03_species_analysis.R
  04_body_mass_analysis.R    # Sam's contribution
  05_visualization_experiments.R
```

Benefits of script-based development: - **Fast iteration**: No need to knit/render documents during development - **Interactive debugging**: Can run code line-by-line in R console - **Version control friendly**: Pure R files produce clean diffs in Git - **Easy collaboration**: Contributors can add numbered script files - **Flexible experimentation**: Quick to test ideas and approaches

2.5.2 Phase 2: Function Extraction (`R/`)

As analysis patterns emerge, extract reusable functions to the `R/` directory:

```r
# R/penguin_utils.R
calculate_species_correlation <- function(data, x_var, y_var,
                                          species_filter = NULL) {
  # Reusable function extracted from scripts
  if (!is.null(species_filter)) {
    data <- data[data$species == species_filter, ]
  }
  cor(data[[x_var]], data[[y_var]], use = "complete.obs")
}
```

```r
create_species_plot <- function(data, x_var, y_var) {
  # Standardized plotting function
  ggplot(data, aes_string(x = x_var, y = y_var,
                          color = "species")) +
    geom_point() +
    theme_minimal()
}
```

2.5.3 Phase 3: Publication Integration (`analysis/paper/`)

Once analysis approaches stabilize, integrate polished results into the manuscript:

```r
# In analysis/paper/paper.Rmd
# Option 1: Source complete scripts
source("../../scripts/02_penguin_correlations.R")
source("../../scripts/04_body_mass_analysis.R")

# Option 2: Use extracted functions
library(here)
source(here("R", "penguin_utils.R"))

correlation_result <- calculate_species_correlation(
  penguins, "flipper_length_mm", "bill_length_mm"
)
```

2.5.4 Recommended Collaborative Workflow:

1. **Project initialization**: Joe runs `zzrrtools.sh` to create project structure
2. **Immediate containerization**: Joe builds Docker container and switches to container-based development from day one
3. **Initial development**: Joe creates exploratory scripts in `scripts/` directory **inside the container**
4. **Collaborative iteration**: Sam clones repo, builds identical container, adds additional script files through pull requests **from within the container**
5. **Code review in scripts**: Both developers refine analysis logic in script files while working in identical Docker environments
6. **Function extraction**: Move stable, reusable code to `R/` directory
7. **Paper integration**: Source scripts or use functions in `analysis/paper/paper.Rmd`
8. **Continuous validation**: All development and testing occurs within the containerized environment

Why this container-first approach works:

- **True reproducibility**: Eliminates "works on my machine" problems from day one
- **Identical environments**: All collaborators work in exactly the same computational environment
- **No environment drift**: Cannot occur when everyone develops within containers
- **Speed**: Script development is faster than R Markdown knitting
- **Modularity**: Each script can focus on a specific analysis aspect
- **Testability**: Functions in `R/` can be easily unit tested in the same environment they'll run in production
- **Seamless collaboration**: Environment setup becomes a one-time `docker build` command for all contributors
- **Development-production parity**: The development environment IS the production environment

This container-first, phased approach gives collaborators the speed of script-based development during exploration while maintaining the reproducibility and narrative flow of literate programming for final outputs. Most importantly, it ensures that all development occurs within the exact computational environment that will be used for final analysis and publication.

While zzrrtools establishes the organizational foundation for reproducible research, it relies on consistent R package environments to function effectively across different systems. The standardized directory structure and R package framework created by zzrrtools becomes most powerful when combined with precise dependency management. This is where renv becomes essential, providing the package-level consistency that complements zzrrtools' structural approach.

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 successfully addresses package-level reproducibility by ensuring identical R package versions across environments, even perfect package consistency cannot prevent analyses from failing or producing different results due to variations in R versions, operating systems, or system-level dependencies. A comprehensive reproducibility solution requires addressing these system-level differences, which is where Docker containerization 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

The zzrrtools setup uses a streamlined Dockerfile based on rocker/r-ver with TinyTeX for LaTeX support. The R version is dynamically matched to the renv.lock file:

```
# Use R version from renv.lock for perfect consistency
ARG R_VERSION=4.3.0
FROM rocker/r-ver:${R_VERSION}

# Prevent interactive prompts
ENV DEBIAN_FRONTEND=noninteractive
```

```
# Install minimal 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/*

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

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

# Create user R library directory and assign permissions
RUN mkdir -p /home/${USERNAME}/R/library && \
    chown -R ${USERNAME}:${USERNAME} /home/${USERNAME}/R

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

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

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

# Install base R packages to user library
RUN Rscript -e '.libPaths(Sys.getenv("R_LIBS_USER")); \
    install.packages(c("tinytex", "rmarkdown", "renv"), \
    repos = "https://cloud.r-project.org")'

# Install TinyTeX in user directory
RUN Rscript -e 'tinytex::install_tinytex()'

# Add TinyTeX binaries to PATH
ENV PATH=/home/${USERNAME}/.TinyTeX/bin/x86_64-linux:$PATH

# Restore R packages via renv
```

```
RUN Rscript -e '.libPaths(Sys.getenv("R_LIBS_USER")); \
    renv::restore()'

# Default to interactive shell
CMD ["/bin/bash"]
```

Key advantages of using rocker/r-ver with TinyTeX:

- **Lightweight base**: Minimal R installation without unnecessary packages
- **TinyTeX integration**: Efficient LaTeX distribution for PDF rendering
- **Security focused**: Non-root user execution for better security
- **User-specific libraries**: Isolated package management in user directory
- **Flexible deployment**: Can be extended with additional tools as needed

Docker Compose Integration:

The setup also includes a comprehensive docker-compose.yml that provides multiple development environments:

```
# Multiple services for different workflows
services:
  r-session:   # Interactive R session
  bash:        # Bash shell access
  research:    # Automated paper rendering
  test:        # Package testing
  check:       # Package checking
```

This allows developers to choose their preferred development environment while maintaining identical package dependencies and system configuration.

5 Combining zzrrtools, 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:

- **zzrrtools** 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 zzrrtools, renv, and Docker with a clear governance structure suitable for multi-developer research teams:

Project Maintainer Role (Joe): - Creates and maintains the research compendium structure - Manages renv environment and package dependencies using automated consistency checking - Updates and maintains Docker images - Reviews and approves contributor changes - Runs renv validation before accepting pull requests

Contributor Role (Other Developers): - Access the private research compendium as invited collaborators - Add analysis content, papers, and documentation using feature branches - Propose new package dependencies through contributions - Submit changes via pull requests from feature branches

Workflow Steps:

1. **Initialize Research Compendium (Maintainer)**:
    - Create standardized project structure using zzrrtools framework
    - Set up analysis directories with comprehensive data organization
    - Initialize renv environment with `renv::init()`
    - Create Dockerfile with enhanced container configuration
2. **Establish Development Environment (Maintainer)**:
    - Install required packages and develop initial analysis
    - Create comprehensive tests for analytical functions
    - Use the renv consistency checker to validate and create initial lockfile:

      ```
      # Validate dependencies and create snapshot
      Rscript check_renv_for_commit.R --fix
      ```

    - Build and test Docker image locally
3. **Maintain Infrastructure (Maintainer)**:

- Review contributor pull requests for package additions
- Use renv consistency checker to validate and update dependencies:

```
# Validate contributor's package requirements
Rscript check_renv_for_commit.R --fail-on-issues

# If validation passes, update environment
Rscript check_renv_for_commit.R --fix
```

- 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` (zzrrtools-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 using GitHub Container Registry:

   GitHub provides GitHub Container Registry (ghcr.io) that's free for private repositories and automatically manages access permissions.

   GitHub Container Registry (Recommended for Private Repos)

```
# Build the image with GitHub Container Registry URL
docker build -t ghcr.io/joe/penguins_analysis:v1.0 .

# Login to GitHub Container Registry (using GitHub Personal Access Token)
echo $GITHUB_TOKEN | docker login ghcr.io -u joe \
  --password-stdin

# Push to GitHub Container Registry (automatically private)
docker push ghcr.io/joe/penguins_analysis:v1.0
```

Setting up GitHub Personal Access Token:

Create a Personal Access Token with the required permissions for container registry operations. The token must include `write:packages` and `read:packages` scopes, plus `repo` access for private repositories.

For detailed step-by-step instructions, see Appendix A: GitHub Personal Access Token Setup.

```
# Export token as environment variable (replace with your actual token)
export GITHUB_TOKEN=ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

# Login and push
echo $GITHUB_TOKEN | docker login ghcr.io -u joe \
  --password-stdin
docker build -t ghcr.io/joe/penguins_analysis:v1.0 .
docker push ghcr.io/joe/penguins_analysis:v1.0
```

Note: If you get a "permission_denied" error when pushing, ensure your token includes the correct scopes (see Appendix A for details).

GitHub Container Registry Benefits:

- **Free tier**: 0.5GB storage included, no billing currently active
- **Automatic access control**: Inherits repository permissions
- **Integrated with GitHub Actions**: Seamless authentication in CI/CD
- **Simple team sharing**: Repository collaborators automatically have access
- **Package management**: Integrated with GitHub Packages ecosystem

Enhanced Docker Workflow:

The zzrrtools setup provides multiple approaches for working with containers, from simple Make commands to direct Docker execution. The recommended approach uses Make commands for simplicity:

```
# Build and run with Make (recommended)
make docker-build    # Build the container
make docker-r        # Interactive R session
make docker-render   # Render research paper
```

For complete Docker workflow options including Docker Compose and direct commands, see Appendix C: Docker Workflow Options.

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 zzrrtools, 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:

- **Joe** (`joe`): Project maintainer who owns the repository
- **Sam** (`sam`): Contributor who extends the analysis

Joe will set up the initial research compendium structure using zzrrtools and create a basic analysis. Sam will extend the analysis with additional visualizations and propose new package dependencies through pull requests. They'll use GitHub for version control and GitHub Container Registry to share the containerized environment.

> **Role Summary**: - **Joe** (`joe`): Repository owner, infrastructure manager, renv maintainer, Docker image builder - **Sam** (`sam`): Contributor, content developer, fork-based collaborator

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

6.2 Step-by-Step Implementation

6.3 Joe: Project Setup and Initial Analysis

Step 1: Create and Initialize the Private GitHub Repository

Joe creates a new **private** GitHub repository called "penguins_analysis" and clones it locally:

```
# Joe creates: https://github.com/joe/penguins_analysis
git clone https://github.com/joe/penguins_analysis.git
```

Repository Setup:

Joe creates a private GitHub repository ("penguins_analysis") to demonstrate collaborative workflows for proprietary research, sensitive data, or early development phases. GitHub's integrated ecosystem provides free private repositories, container registry, and CI/CD automation in a single platform.

Key setup decisions: - **Private repository** for controlled access and data protection - **Container registry** integration for reproducible environments
- **Collaborative permissions** allowing Sam write access with fork-based workflow for governance

Step 2: Create Research Compendium with Enhanced rrtools Setup

Joe uses the rrtools setup script to create a comprehensive research compendium:

```
# Run the rrtools setup script
~/prj/zzrrtools/zzrrtools.sh
```

This automated script creates a complete research compendium structure that includes:

- **R package framework** with proper DESCRIPTION, NAMESPACE, and utility functions
- **Comprehensive directory organization** with structured data/ (raw_data, derived_data, metadata, validation, external_data), analysis/ (paper, figures, templates, tables), scripts/, tests/, docs/, and archive/ subdirectories
- **Intelligent file organization** that automatically moves existing files to appropriate locations based on file type (.R → scripts/, .Rmd → analysis/, .csv → data/raw_data/, .pdf → docs/)
- **Automated renv setup** with curated package collection and consistency checking via `check_renv_for_commit.R` script
- **Docker integration** using rocker/verse with development environment including zsh, vim, and development tools
- **Dotfiles integration** for container development (.vimrc, .zshrc_docker) to maintain familiar development environment
- **GitHub Actions workflows** for automated CI/CD with renv validation steps
- **Comprehensive Makefile** with both native R and Docker targets, including renv validation commands
- **Symbolic links** for easy navigation (a→data, n→analysis, f→figures, s→scripts, etc.)
- **User documentation** including comprehensive user guide (RRTOOLS_USER_GUIDE.md)

The script automatically organizes any existing files into the appropriate directories, preserves important files in the project root, and creates a professional research compendium structure following best practices.

Step 3: Complete the renv Environment Setup

Since the zzrrtools script already created the basic structure and renv configuration, Joe completes the package environment setup:

```
# The setup script created setup_renv.R - run it to install packages
source("setup_renv.R")
```

This installs a curated collection of R packages covering data manipulation, statistical analysis, visualization, and document generation. The automated setup ensures consistent package environments and proper dependency management through renv snapshots.

Step 3b: Validate renv Environment Consistency

The zzrrtools setup includes an automated renv consistency checking script that Joe can use to ensure dependency management remains clean:

```
# Check renv consistency (Joe runs this before commits)
Rscript check_renv_for_commit.R

# Or using the Makefile shortcut
make check-renv

# Automatically fix common issues
make check-renv-fix

# For CI/CD environments
make check-renv-ci
```

The validation script ensures package environment consistency by verifying dependencies across code files, DESCRIPTION, and renv.lock, preventing common collaboration issues where team members have mismatched environments.

Step 4: Create Initial Analysis Paper

Joe creates an initial analysis examining flipper length vs. bill length relationships in the Palmer Penguins dataset, implementing basic visualization and statistical exploration within the research compendium structure.

Step 5: Create Tests for Analysis Functions

Joe implements testing to ensure reproducible research through data validation, error detection, and environment verification. Testing provides collaboration confidence and supports publication standards by validating data integrity, statistical relationships, and pipeline functionality.

Joe sets up the testing framework and creates basic data validation tests to verify dataset availability, dimensions, and required columns. These tests ensure the analysis environment is correctly configured and catch data-related issues early in the development process.

Step 6: Create a .gitignore file

Joe configures version control to track source code and dependencies while excluding generated outputs and temporary files. The principle: track the "recipe" (code + dependencies), not the "meal" (outputs).

Joe creates a `.gitignore` file excluding renv libraries, generated outputs, temporary files, and system artifacts. This keeps the repository lightweight while ensuring collaborators can recreate the complete environment from tracked dependencies.

Step 7: Create a Dockerfile

Joe creates a Dockerfile using the rrtools setup, based on rocker/r-ver with dynamic R version matching. The core structure provides:

- **R version consistency**: Matches exact R version specified in renv.lock
- **Development environment**: Includes zsh, vim, and essential development tools
- **Security**: Non-root user execution with proper file permissions
- **renv integration**: Automatic package restoration from lockfile

A minimal example Dockerfile structure:

```
ARG R_VERSION=4.3.0
FROM rocker/r-ver:${R_VERSION}

# Install system dependencies
RUN apt-get update && apt-get install -y \
    libxml2-dev libcurl4-openssl-dev libssl-dev \
    pandoc git curl && rm -rf /var/lib/apt/lists/*

# Create non-root user and set working directory
ARG USERNAME=analyst
RUN useradd --create-home --shell /bin/bash ${USERNAME}
WORKDIR /home/${USERNAME}/project

# Copy and restore R environment
COPY --chown=${USERNAME}:${USERNAME} renv.lock ./
```

```
USER ${USERNAME}
RUN R -e "install.packages('renv'); renv::restore()"

# Copy project and install as package
COPY --chown=${USERNAME}:${USERNAME} . .
RUN R -e "devtools::install('.')"
CMD ["/bin/bash"]
```

Note: A comprehensive production Dockerfile with development environment configuration (zsh, vim plugins, dotfiles integration) is provided in **Appendix E: Docker Configuration Examples**.

R Version Synchronization:

The Dockerfile uses a build argument to ensure the R version exactly matches what's specified in `renv.lock`. This eliminates potential issues from R version mismatches between the package environment and the underlying R interpreter. The build command extracts the R version directly from the renv lockfile:

```
# Extract R version from renv.lock using JSON parsing (works without
# renv installed)
R_VERSION=$(python3 -c "
import json
with open('renv.lock', 'r') as f:
    lockfile = json.load(f)
    print(lockfile['R']['Version'])
")

# Alternative using jq (if available)
# R_VERSION=$(jq -r '.R.Version' renv.lock)

# Build Docker image with extracted R version
docker build --build-arg R_VERSION=${R_VERSION} \
  -t ghcr.io/joe/penguins_analysis:v1.0 .
```

If the `renv.lock` file specifies R 4.3.1, the Docker image will use `rocker/r-ver:4.3.1`. If renv is updated to R 4.4.0, the Docker build will automatically use `rocker/r-ver:4.4.0`. This maintains perfect consistency between the package environment and system environment.

Step 8: Container-Based Development

Joe performs all development work inside the Docker container, ensuring consistent environments and immediate visibility of changes to the host system through volume mounting. The

container provides a complete development environment with package management, editing tools, and validation utilities.

Step 9: Update and Share Environment

When package dependencies change, Joe rebuilds the Docker image with updated renv.lock specifications and pushes the updated environment to the container registry for team access. This ensures collaborators have access to the identical development environment.

Step 10: Finalize and Share

Joe validates the complete workflow by running tests and rendering the analysis paper, then commits the project and shares the updated Docker environment with the team for collaborative development.

At this point, Joe has established a complete reproducible research framework ready for collaborative development. Sam can now access the private repository, pull the Docker environment, and contribute through the established fork-based workflow.

6.4 Sam: Extending the Analysis

Step 1: Repository Access and Fork Setup

Sam gains access to Joe's private repository, creates a fork for isolated development, and establishes the upstream connection. This fork-based workflow provides clear governance with controlled integration while maintaining professional collaboration standards.

Step 2: Environment Access Setup

Sam synchronizes with the upstream repository and configures authentication for the private container registry. This includes creating a GitHub Personal Access Token with appropriate permissions for repository and package access.

Step 3: Container-Based Development

Sam authenticates with the container registry, pulls Joe's environment, creates a feature branch, and enters the containerized development environment with the repository mounted for seamless file access.

Step 4: Analysis Development

Sam develops new analysis components in the scripts directory, working within the identical computational environment as Joe. The volume-mounted approach ensures seamless file synchronization and persistent access to local development tools.

Sam creates a new analysis examining body mass vs. bill length relationships, developing visualizations with species-specific correlations and integrating ggplot2 for enhanced plotting.

After iterative development and testing, Sam integrates the refined analysis into the research paper.

Step 5: Paper Integration and Testing

Sam integrates the new analysis into the research paper, combining Joe's original visualizations with the new body mass analysis. Sam also creates tests to validate the new functionality and ensure package dependencies are properly documented.

Step 6: Validation and Quality Assurance

Sam creates tests for the new body mass analysis, validates data integrity and statistical relationships, then runs the complete test suite and verifies paper rendering to ensure no regressions before submission.

Step 7: Contribution Submission

Sam commits the completed analysis, tests, and documentation to their feature branch and creates a cross-repository pull request to the original repository. This ensures proper code review and governance while maintaining clear attribution of contributions.

At this point, Sam has successfully contributed new analysis through the established collaborative workflow. Joe reviews the pull request, tests the changes in the containerized environment, and merges the contribution while maintaining project governance and quality standards.

6.5 Key Benefits Demonstrated in This Example

This collaborative workflow demonstrates several advantages of the zzrrtools + renv + Docker approach:

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**: Sam 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.

For complete GitHub Actions setup instructions, workflow examples, and CI/CD configuration, see Appendix D: GitHub Actions CI/CD Setup.

The collaborative workflow demonstrated above illustrates the power of combining zzrrtools, renv, and Docker for reproducible research. However, successful implementation of this approach requires understanding both when it's most beneficial and how to apply it effectively. The following best practices and considerations provide guidance for teams considering this strategy.

7 Best Practices and Considerations

7.1 When to Use This Approach

The zzrrtools + 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

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.

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.

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

Despite these challenges, the benefits of reproducible research far outweigh the implementation costs, particularly for collaborative and long-term projects. The approach described in this white paper provides a robust foundation for achieving reproducibility that meets the standards expected in professional data science and academic research.

8 Conclusion

Achieving full reproducibility in R requires addressing project organization, package dependencies, and system-level consistency, while ensuring code quality through testing. By combining zzrrtools 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.

9 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 (Enhanced in zzrrtools framework)
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/

Appendix A: GitHub Personal Access Token Setup

This appendix provides detailed step-by-step instructions for creating a GitHub Personal Access Token with the required permissions for container registry operations.

Why Container Access is Required: Both collaborators need container-related permissions because Docker images are stored in GitHub Container Registry as private packages that require authentication to access.

9.1 Step-by-Step Token Creation

1. Navigate to GitHub Settings: - Go to GitHub.com and sign in - Click your profile picture (top right) → Settings - In the left sidebar: Developer settings → Personal access tokens → Tokens (classic)

Note: GitHub now offers two token types: - **Fine-grained personal access tokens** (recommended for enhanced security) - **Personal access tokens (classic)** (documented here for broader compatibility)

2. Create New Token: - Click "Generate new token" → "Generate new token (classic)" - Add a descriptive note (e.g., "Docker Container Registry Access") - Set expiration (recommended: 90 days for security)

3. Select Required Scopes (check these boxes): - **repo** (Full control of private repositories) - *Required for private repos* - **write:packages** (Upload Docker images to GitHub Container Registry) - *Required for Joe* - **read:packages** (Download Docker images from GitHub Container Registry) - *Required for both Joe and Sam* - **delete:packages** (Delete packages from GitHub Package Registry) - *Optional but recommended*

Note: Sam only needs `read:packages` and `repo`, but Joe needs all container permissions to push Docker images.

Token Type Recommendation: While this guide uses classic tokens for broader compatibility, GitHub recommends fine-grained personal access tokens for enhanced security when working within your own organizations.

4. Generate and Copy Token: - Click "Generate token" at the bottom - **Important**: Copy the token immediately - you won't see it again - Store it securely (see security practices below)

9.2 Token Security Best Practices

- **Never commit tokens to repositories** - Use `.gitignore` to exclude files containing tokens
- **Use environment variables** - Store tokens in shell environment variables
- **Set reasonable expiration dates** - Use 30-90 day expiration for security
- **Revoke unused tokens** - Clean up tokens when no longer needed
- **Consider GitHub CLI** - Use `gh auth login` for easier management
- **Monitor token usage** - Check GitHub Settings → Developer settings → Personal access tokens for activity

9.3 Alternative: Using GitHub CLI

For easier token management, consider using GitHub CLI instead of manual tokens:

```
# Install and authenticate (handles tokens automatically)
gh auth login --scopes write:packages,read:packages,repo

# Login to container registry (automatic with gh auth)
echo $(gh auth token) | docker login ghcr.io \
  -u $(gh api user --jq .login) --password-stdin
```

9.4 Troubleshooting Common Issues

"permission_denied: The token provided does not match expected scopes" - Verify your token includes `write:packages` and `read:packages` scopes - For private repositories, ensure `repo` scope is also selected - Create a new token with correct permissions if needed

Token not recognized: - Ensure token is properly exported: `export GITHUB_TOKEN=your_token_here` - Verify token hasn't expired - Check that you're using the full token (starts with `ghp_`) 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.

Appendix F: Comprehensive Test Suite

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.

9.1 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
```

```r
# 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))
})

# 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
# 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
# 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.
})
```

9.2 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")
```

9.3 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.

Appendix B: Enhanced Directory Structure

The zzrrtools setup creates the following comprehensive directory structure:

```
project/
   DESCRIPTION               # Package metadata and dependencies
   LICENSE                   # Project license
   README.md                 # Project documentation
   project.Rproj             # RStudio project file
   renv.lock                 # Package dependency lockfile
   setup_renv.R              # Automated renv setup script
   Dockerfile                # Container specification
   docker-compose.yml        # Multi-service Docker setup
   Makefile                  # Build automation (native R + Docker)
   .Rprofile                 # R startup configuration
   .dockerignore             # Docker build exclusions
   RRTOOLS_USER_GUIDE.md     # Comprehensive user documentation
   .github/workflows/        # Multiple GitHub Actions workflows
      docker-ci.yml          # Docker-based CI/CD
      r-package.yml          # R package checking
      render-paper.yml       # Automated paper rendering
   check_renv_for_commit.R   # renv consistency validation script
   R/                        # R functions and utilities
      utils.R                # Pre-built utility functions
   man/                      # Generated function documentation
   data/                     # Comprehensive data organization
      raw_data/              # Original, unmodified data
      derived_data/          # Processed/cleaned data
      metadata/              # Data documentation
      validation/            # Data validation scripts
      external_data/         # Third-party datasets
   analysis/                 # Research analysis
      paper/                 # Manuscript with PDF output
      figures/               # Generated plots and charts
      tables/                # Generated tables
      templates/             # Document templates and CSL styles
   scripts/                  # Working R scripts and code snippets
   tests/testthat/           # Unit tests and validation
   vignettes/                # Package vignettes and tutorials
   inst/doc/                 # Package documentation
   docs/                     # Additional documentation
   archive/                  # Archived files and old versions
```

```
[a,n,f,t,s,m,e,o,c]    # Symbolic links for easy navigation
```

9.1 Key Features Explained:

Comprehensive Data Organization: - **raw_data/**: Original, unmodified datasets as received - **derived_data/**: Processed, cleaned, or transformed data - **metadata/**: Documentation about data sources, collection methods, variables - **validation/**: Scripts that verify data integrity and quality - **external_data/**: Third-party datasets or reference data

Multiple Output Formats: - **figures/**: Generated plots, charts, and visualizations - **tables/**: Generated summary tables and statistical results - **paper/**: Main manuscript and analysis documents - **templates/**: Document templates and citation style files

Professional R Package Structure: - **R/**: Custom functions and utilities - **man/**: Generated documentation for R functions - **tests/testthat/**: Unit tests and validation scripts - **vignettes/**: Long-form documentation and tutorials - **DESCRIPTION**: Package metadata and dependency specifications

Docker Orchestration: - **Dockerfile**: Main container specification - **docker-compose.yml**: Multi-service development environments - **Makefile**: Build automation supporting both native R and Docker workflows

Automated Workflows: - **.github/workflows/**: GitHub Actions for testing, checking, and rendering - **setup_renv.R**: Automated package environment setup - **RRTOOLS_USER_GUIDE.md**: Comprehensive usage documentation

Navigation Shortcuts: - **Symbolic links**: Single-letter shortcuts for easy navigation - `a` → analysis/, `n` → analysis/, `f` → figures/ - `t` → tests/, `s` → scripts/, `m` → man/ - `e` → external_data/, `o` → output/, `c` → cache/

This structure supports complex research projects while maintaining clear organization and following established research compendium principles.

Appendix C: Docker Workflow Options

The zzrrtools setup provides multiple approaches for working with Docker containers, each suited to different development preferences and use cases.

9.1 Option 1: Make Commands (Recommended)

The Makefile provides simplified commands that abstract complex Docker syntax:

```
# Build Docker image
make docker-build

# Interactive R session (command line users)
make docker-r

# Interactive bash session
make docker-bash

# Render research paper
make docker-render

# Run tests
make docker-test

# Package checking
make docker-check

# See all available commands
make help
```

Benefits: - **Simple syntax**: Easy-to-remember commands - **Consistent interface**: Same commands work across different projects - **Hidden complexity**: Complex Docker flags are abstracted away - **Documentation**: `make help` shows all available options

9.2 Option 2: Docker Compose Services

Docker Compose orchestrates multiple container configurations:

```
# Interactive R session
docker-compose run --rm r-session

# Bash shell access
docker-compose run --rm bash

# Automated paper rendering
docker-compose run --rm research
```

```
# Package testing
docker-compose run --rm test

# Package checking
docker-compose run --rm check
```

Docker Compose Configuration Example:

```
services:
  r-session:
    build: .
    volumes:
      - .:/home/analyst/project
      - ./cache:/home/analyst/cache
    working_dir: /home/analyst/project

  bash:
    build: .
    volumes:
      - .:/home/analyst/project
    working_dir: /home/analyst/project
    entrypoint: ["/bin/bash"]

  research:
    build: .
    volumes:
      - .:/home/analyst/project
      - ./analysis/figures:/home/analyst/output
    working_dir: /home/analyst/project
    command: ["R", "-e", "rmarkdown::render('analysis/paper/paper.Rmd')"]
```

Benefits: - **Service orchestration**: Multiple predefined container configurations - **Volume management**: Consistent volume mounting across services - **Environment isolation**: Different services for different purposes - **Parallel execution**: Can run multiple services simultaneously

9.3 Option 3: Direct Docker Commands

For maximum control, use Docker commands directly:

```
# Basic interactive session
docker run --rm -it -v "$(pwd):/home/analyst/project" \
  ghcr.io/joe/penguins_analysis:v1.0

# Interactive session with mounted cache
docker run --rm -it \
  -v "$(pwd):/home/analyst/project" \
  -v "$(pwd)/cache:/home/analyst/cache" \
  -w /home/analyst/project \
  ghcr.io/joe/penguins_analysis:v1.0

# Render research paper
docker run --rm \
  -v "$(pwd):/home/analyst/project" \
  -v "$(pwd)/analysis/figures:/home/analyst/output" \
  -w /home/analyst/project \
  ghcr.io/joe/penguins_analysis:v1.0 \
  R -e "rmarkdown::render('analysis/paper/paper.Rmd')"

# Run specific tests
docker run --rm \
  -v "$(pwd):/home/analyst/project" \
  -w /home/analyst/project \
  ghcr.io/joe/penguins_analysis:v1.0 \
  R -e "testthat::test_file('tests/testthat/test-data-integrity.R')"

# Interactive bash session
docker run --rm -it \
  -v "$(pwd):/home/analyst/project" \
  -w /home/analyst/project \
  ghcr.io/joe/penguins_analysis:v1.0 \
  /bin/bash
```

Common Docker Flags Explained: - `--rm`: Remove container when it exits - `-it`: Interactive terminal session - `-v`: Mount volume (host:container) - `-w`: Set working directory inside container - `--entrypoint`: Override default command

Benefits: - **Maximum flexibility**: Full control over container configuration - **Educational**: Shows exactly what's happening under the hood - **Troubleshooting**: Easier to debug when you see all options - **Portability**: Commands work on any Docker installation

9.4 Volume Mounting Strategies

Project Files:

```
# Mount entire project directory
-v "$(pwd):/home/analyst/project"
```

Output Separation:

```
# Separate outputs from source
-v "$(pwd)/analysis/figures:/home/analyst/output"
```

Cache Persistence:

```
# Persistent package cache across sessions
-v "$(pwd)/cache:/home/analyst/cache"
```

Read-only Source:

```
# Protect source files from modification
-v "$(pwd):/home/analyst/project:ro"
```

9.5 Choosing the Right Approach

Use Make Commands When: - You want simplicity and consistency - You're new to Docker - You're focusing on analysis rather than infrastructure

Use Docker Compose When: - You need multiple service configurations - You're working with a team using standardized environments - You want to define complex volume and networking setups

Use Direct Commands When: - You need maximum flexibility - You're troubleshooting container issues - You're creating custom workflows not covered by Make targets

All three approaches can be used together in the same project, depending on the specific task and user preferences.

Appendix D: GitHub Actions CI/CD Setup

GitHub Actions provides automated testing and deployment for research compendia. This appendix covers comprehensive CI/CD setup for reproducible research workflows.

9.1 Understanding GitHub Actions for Research

What is CI/CD for Research?

Continuous Integration/Continuous Deployment (CI/CD) automatically tests your research code whenever changes are made. For research compendia, this means:

- **Automated testing**: Every push triggers your test suite automatically
- **Environment consistency**: Tests run in identical Docker environments
- **Early error detection**: Problems caught immediately during development
- **Collaboration confidence**: Team members see if changes break functionality
- **Reproducibility validation**: Ensures analysis works across different systems

9.2 Step-by-Step Setup

9.2.1 Step 1: Create Workflow Directory

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

9.2.2 Step 2: Docker-based CI Workflow with renv Validation

Create `.github/workflows/docker-ci.yml`:

```
name: Docker CI with renv Validation

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

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout repository
      uses: actions/checkout@v4

    - name: Set up R for renv validation
```

```
      uses: r-lib/actions/setup-r@v2
      with:
        r-version: 'release'

    - name: Install renv for validation
      run: |
        install.packages("renv")
      shell: Rscript {0}

    - name: Validate renv consistency before Docker build
      run: |
        # Validate renv environment before building Docker image
        Rscript check_renv_for_commit.R --fail-on-issues --quiet

    - name: Set up Docker Buildx
      uses: docker/setup-buildx-action@v3

    - name: Extract R version from renv.lock
      id: r-version
      run: |
        R_VERSION=$(Rscript -e "cat(renv::lockfile_read()\$R\$Version)")
        echo "r-version=${R_VERSION}" >> $GITHUB_OUTPUT

    - name: Build Docker image
      uses: docker/build-push-action@v5
      with:
        context: .
        push: false
        tags: ${{ github.repository }}:latest
        build-args: |
          R_VERSION=${{ steps.r-version.outputs.r-version }}
        cache-from: type=gha
        cache-to: type=gha,mode=max

    - name: Run tests in container
      run: |
        docker run --rm -v $PWD:/home/analyst/project \
          ${{ github.repository }}:latest \
          R -e "testthat::test_dir('tests/testthat')"

    - name: Render research paper
      run: |
```

```
        docker run --rm -v $PWD:/home/analyst/project \
          -v $PWD/analysis/figures:/home/analyst/output \
          ${{ github.repository }}:latest \
          R -e "rmarkdown::render('analysis/paper/paper.Rmd')"

    - name: Upload rendered paper
      uses: actions/upload-artifact@v4
      if: success()
      with:
        name: research-paper
        path: analysis/paper/paper.pdf
```

9.2.3 Step 3: R Package Check Workflow

Create `.github/workflows/r-package.yml`:

```
name: R Package Check

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

jobs:
  R-CMD-check:
    runs-on: ${{ matrix.config.os }}

    name: ${{ matrix.config.os }} (${{ matrix.config.r }})

    strategy:
      fail-fast: false
      matrix:
        config:
          - {os: ubuntu-latest,   r: 'release'}
          - {os: macOS-latest,    r: 'release'}
          - {os: windows-latest,  r: 'release'}

    env:
      GITHUB_PAT: ${{ secrets.GITHUB_TOKEN }}
      R_KEEP_PKG_SOURCE: yes
```

```
    steps:
      - uses: actions/checkout@v4

      - uses: r-lib/actions/setup-pandoc@v2

      - uses: r-lib/actions/setup-r@v2
        with:
          r-version: ${{ matrix.config.r }}
          http-user-agent: ${{ matrix.config.http-user-agent }}
          use-public-rspm: true

      - uses: r-lib/actions/setup-renv@v2

      - name: Install system dependencies
        if: runner.os == 'Linux'
        run: |
          sudo apt-get update
          sudo apt-get install -y \
            libcurl4-openssl-dev \
            libssl-dev \
            libxml2-dev

      - name: Validate renv consistency
        run: |
          # Use the renv validation script included in the repository
          Rscript check_renv_for_commit.R --fail-on-issues --quiet

      - uses: r-lib/actions/check-r-package@v2
        with:
          upload-snapshots: true
```

9.2.4 Step 4: Automated Paper Rendering

Create `.github/workflows/render-paper.yml`:

```
name: Render Research Paper

on:
  workflow_dispatch:  # Manual trigger
  push:
    branches: [ main, master ]
```

```yaml
    paths:
      - 'analysis/paper/**'
      - 'analysis/data/**'
      - 'R/**'
      - 'data/**'

jobs:
  render:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Build Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          push: false
          tags: paper-render:latest
          cache-from: type=gha
          cache-to: type=gha,mode=max

      - name: Render paper in container
        run: |
          docker run --rm \
            -v $PWD:/home/analyst/project \
            -v $PWD/analysis/figures:/home/analyst/output \
            paper-render:latest \
            R -e "rmarkdown::render('analysis/paper/paper.Rmd')"

      - name: Upload rendered paper
        uses: actions/upload-artifact@v4
        with:
          name: research-paper-${{ github.sha }}
          path: |
            analysis/paper/paper.pdf
            analysis/figures/*.png
            analysis/figures/*.jpg
```

```
          retention-days: 30
```

9.2.5 Step 5: Container Registry Integration

Create `.github/workflows/container-publish.yml`:

```yaml
name: Build and Push Container

on:
  push:
    branches: [ main ]
    tags: [ 'v*' ]
  pull_request:
    branches: [ main ]

env:
  REGISTRY: ghcr.io
  IMAGE_NAME: ${{ github.repository }}

jobs:
  build-and-push:
    runs-on: ubuntu-latest
    permissions:
      contents: read
      packages: write

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up Docker Buildx
        uses: docker/setup-buildx-action@v3

      - name: Log in to Container Registry
        if: github.event_name != 'pull_request'
        uses: docker/login-action@v3
        with:
          registry: ${{ env.REGISTRY }}
          username: ${{ github.actor }}
          password: ${{ secrets.GITHUB_TOKEN }}
```

```yaml
      - name: Extract metadata
        id: meta
        uses: docker/metadata-action@v5
        with:
          images: ${{ env.REGISTRY }}/${{ env.IMAGE_NAME }}
          tags: |
            type=ref,event=branch
            type=ref,event=pr
            type=semver,pattern={{version}}
            type=semver,pattern={{major}}.{{minor}}

      - name: Build and push Docker image
        uses: docker/build-push-action@v5
        with:
          context: .
          platforms: linux/amd64,linux/arm64
          push: ${{ github.event_name != 'pull_request' }}
          tags: ${{ steps.meta.outputs.tags }}
          labels: ${{ steps.meta.outputs.labels }}
          cache-from: type=gha
          cache-to: type=gha,mode=max
```

9.3 Workflow Explanations

9.3.1 Docker CI Workflow Features:

- **Pre-build renv Validation**: Validates package dependency consistency before Docker build (prevents build failures)
- **Dynamic R Version**: Extracts R version from renv.lock and passes it to Docker build
- **Build Testing**: Ensures Docker image builds with latest changes using correct R version
- **Comprehensive Testing**: Runs R package tests and renders paper in container
- **Artifact Generation**: Saves rendered papers as downloadable artifacts
- **Caching**: Uses GitHub Actions cache for faster builds
- **Early Failure**: Stops pipeline immediately if dependency issues are detected

9.3.2 R Package Check Features:

- **Multi-platform Testing**: Tests on Ubuntu, macOS, and Windows
- **R CMD Check**: Comprehensive package validation
- **renv Integration**: Automatically restores package environment
- **renv Consistency Validation**: Verifies dependency synchronization across platforms

- **System Dependencies**: Installs required system libraries

9.3.3 Paper Rendering Features:

- **Selective Triggering**: Only runs when relevant files change
- **Manual Execution**: Can be triggered manually via GitHub interface
- **Artifact Storage**: Saves PDFs and figures with retention policy
- **Path-based Triggers**: Responds to changes in analysis files

9.3.4 Container Publishing Features:

- **Automated Building**: Builds on pushes and tags
- **Multi-architecture**: Supports AMD64 and ARM64 platforms
- **Semantic Versioning**: Automatic tagging based on git tags
- **Security**: Uses built-in GitHub token for authentication

9.4 Authentication and Permissions

9.4.1 Built-in GITHUB_TOKEN:

The built-in `GITHUB_TOKEN` automatically provides: - Read access to repository contents - Write access to GitHub Packages (when permissions are set) - No manual setup required

9.4.2 Setting Repository Permissions:

1. **Repository Settings → Actions → General**
2. **Workflow permissions**: Choose "Read and write permissions"
3. **Allow GitHub Actions to create and approve pull requests**: Enable if needed

9.4.3 Using Personal Access Tokens (Advanced):

For broader permissions, create repository secrets:

1. **Repository Settings → Secrets and variables → Actions**
2. **New repository secret**: Add `GHCR_TOKEN` with Personal Access Token
3. **Reference in workflow**: `password: ${{ secrets.GHCR_TOKEN }}`

9.5 Integration with Collaborative Workflow

9.5.1 Pull Request Integration:

When Sam submits a pull request: 1. GitHub automatically triggers CI workflows 2. Tests run in clean environment identical to production 3. Results displayed directly in pull request interface 4. Merge can be blocked if tests fail

9.5.2 Branch Protection Rules:

Enable in **Repository Settings** → **Branches**: - **Require status checks**: Force CI to pass before merging - **Require branches to be up to date**: Ensure latest code is tested - **Include administrators**: Apply rules to all users

9.6 Monitoring and Troubleshooting

9.6.1 Viewing Workflow Results:

1. **Repository** → **Actions** tab
2. Click specific workflow run to see details
3. Expand steps to see detailed logs
4. Download artifacts (rendered papers, test results)

9.6.2 Common Issues and Solutions:

Docker Build Failures: - Check Dockerfile syntax - Verify all COPY paths exist - Ensure base image is accessible

renv Restore Failures: - Verify renv.lock is committed - Check for platform-specific packages - Consider using RSPM for faster installs

Permission Errors: - Verify GITHUB_TOKEN permissions - Check repository secrets configuration - Ensure workflows have necessary permissions

9.6.3 Performance Optimization:

Caching Strategies: - Docker layer caching with `cache-from/cache-to` - renv package caching with `r-lib/actions/setup-renv` - Artifact caching for large datasets

Parallel Execution: - Run tests and documentation in parallel jobs - Use matrix strategies for multi-platform testing - Conditional execution based on changed files

This comprehensive CI/CD setup ensures that research compendia remain reproducible, tested, and deployment-ready throughout the development lifecycle.

Appendix E: Docker Configuration Examples

This appendix provides comprehensive Dockerfile examples ranging from minimal configurations to full development environments.

9.1 Comprehensive Production Dockerfile

The following Dockerfile provides a complete development environment with zsh, vim plugins, dotfiles integration, and development tools:

```
# Use R version from renv.lock for perfect consistency
ARG R_VERSION=4.3.0
FROM rocker/r-ver:${R_VERSION}

# Install system dependencies including zsh and development tools
RUN apt-get update && apt-get install -y \
    libxml2-dev \
    libcurl4-openssl-dev \
    libssl-dev \
    libgit2-dev \
    libfontconfig1-dev \
    libcairo2-dev \
    libxt-dev \
    pandoc \
    zsh \
    curl \
    git \
    fonts-dejavu \
    && rm -rf /var/lib/apt/lists/*
```

```
# Create non-root user with zsh as default shell
ARG USERNAME=analyst
RUN useradd --create-home --shell /bin/zsh ${USERNAME}

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

# Copy project files first (for better Docker layer caching)
COPY --chown=${USERNAME}:${USERNAME} DESCRIPTION .
COPY --chown=${USERNAME}:${USERNAME} renv.lock* ./
COPY --chown=${USERNAME}:${USERNAME} .Rprofile* ./
COPY --chown=${USERNAME}:${USERNAME} renv/activate.R* renv/activate.R

# Configure renv library path
ENV RENV_PATHS_LIBRARY renv/library

# Switch to non-root user for R package installation
USER ${USERNAME}

# Install renv and essential R packages
RUN R -e "install.packages(c('renv', 'remotes', 'devtools', 'knitr', \
    'rmarkdown'), repos = c(CRAN = 'https://cloud.r-project.org'))"

# Restore R packages from lockfile (if exists)
RUN R -e "if (file.exists('renv.lock')) renv::restore() else \
    cat('No renv.lock found, skipping restore\\n')"

# Copy dotfiles for development environment
# Note: Ensure .vimrc and .zshrc_docker exist in build context or create
# defaults
COPY --chown=${USERNAME}:${USERNAME} .vimrc /home/${USERNAME}/.vimrc
COPY --chown=${USERNAME}:${USERNAME} .zshrc_docker /home/${USERNAME}/.zshrc

# Install zsh plugins for shell experience
RUN mkdir -p /home/${USERNAME}/.zsh && \
    git clone https://github.com/zsh-users/zsh-autosuggestions \
        /home/${USERNAME}/.zsh/zsh-autosuggestions && \
    chown -R ${USERNAME}:${USERNAME} /home/${USERNAME}/.zsh

# Install vim-plug and configure vim environment
RUN mkdir -p /home/${USERNAME}/.vim/autoload && \
    curl -fLo /home/${USERNAME}/.vim/autoload/plug.vim \
```

```
    https://raw.githubusercontent.com/junegunn/vim-plug/master/plug.vim && \
    chown -R ${USERNAME}:${USERNAME} /home/${USERNAME}/.vim

# Install vim plugins (suppress interactive mode)
RUN vim +PlugInstall +qall || true

# Copy rest of project
COPY --chown=${USERNAME}:${USERNAME} . .

# Install the research compendium as a package
RUN R -e "devtools::install('.', dependencies = TRUE)"

# Set default shell to zsh for development experience
WORKDIR /home/${USERNAME}/project
CMD ["/bin/zsh"]
```

9.2 Features of the Comprehensive Dockerfile

This production-ready Dockerfile provides:

- **R version consistency**: Matches exact R version specified in renv.lock for perfect environment alignment
- **Minimal base**: rocker/r-ver provides clean R installation without unnecessary packages
- **Shell environment**: zsh with autosuggestions and professional prompt for improved productivity
- **Editor environment**: vim with plugins configured automatically during build
- **Dotfiles integration**: Personal development preferences (.vimrc, .zshrc) copied from host system
- **Development tools**: git, curl, pandoc, and essential development libraries pre-installed
- **Security**: Non-root user execution with proper file permissions
- **renv integration**: Automatic package restoration with proper library path configuration
- **Container-optimized workflow**: Optimized layer caching and build process for efficient rebuilds

9.3 R Version Extraction

The Dockerfile uses a build argument to ensure the R version exactly matches what's specified in `renv.lock`. The build command extracts the R version directly from the renv lockfile:

```
# Extract R version from renv.lock using JSON parsing (works without
# renv installed)
R_VERSION=$(python3 -c "
import json
with open('renv.lock', 'r') as f:
    lockfile = json.load(f)
    print(lockfile['R']['Version'])
")

# Alternative using jq (if available)
# R_VERSION=$(jq -r '.R.Version' renv.lock)

# Build Docker image with extracted R version
docker build --build-arg R_VERSION=${R_VERSION} \
  -t ghcr.io/joe/penguins_analysis:v1.0 .
```

If the `renv.lock` file specifies R 4.3.1, the Docker image will use `rocker/r-ver:4.3.1`. If renv is updated to R 4.4.0, the Docker build will automatically use `rocker/r-ver:4.4.0`. This maintains perfect consistency between the package environment and system environment.

Appendix G: renv Management and Validation

This appendix provides comprehensive details for the renv consistency checking script and dependency management workflows.

9.1 renv Consistency Checker Features

The `check_renv_for_commit.R` script provides comprehensive package dependency validation:

- **Comprehensive analysis**: Scans R/, scripts/, and analysis/ directories for package dependencies
- **Multi-source validation**: Compares packages across code files, DESCRIPTION, and renv.lock
- **CRAN validation**: Verifies packages are available and properly named
- **Automatic fixing**: Can update DESCRIPTION and regenerate renv.lock automatically
- **CI/CD integration**: Supports automated workflows with proper exit codes
- **Interactive mode**: Provides detailed feedback and user prompts during development

9.2 Command Options

```
# Interactive dependency checking (recommended for development)
Rscript check_renv_for_commit.R

# Automated fixing for CI/CD pipelines
Rscript check_renv_for_commit.R --fix --fail-on-issues

# Quick snapshot update only
Rscript check_renv_for_commit.R --snapshot

# Quiet mode with minimal output
Rscript check_renv_for_commit.R --quiet

# Help and usage information
Rscript check_renv_for_commit.R --help
```

9.3 Typical Development Workflow

1. **Add new packages to your analysis**: Install packages normally with `install.packages()` or `renv::install()`
2. **Before committing**: Run `Rscript check_renv_for_commit.R` to validate consistency
3. **Review findings**: The script identifies missing packages, invalid entries, and synchronization issues
4. **Auto-fix if desired**: Use `--fix` flag to automatically update DESCRIPTION and renv.lock
5. **Commit with confidence**: Proceed with git commit knowing dependencies are properly managed

9.4 Integration with Development Workflows

9.4.1 Pre-commit Hooks

```
# Add to .git/hooks/pre-commit
Rscript check_renv_for_commit.R --fail-on-issues --quiet
```

9.4.2 Makefile Integration

```
check-renv:
    Rscript check_renv_for_commit.R

check-renv-fix:
    Rscript check_renv_for_commit.R --fix

check-renv-ci:
    Rscript check_renv_for_commit.R --quiet --fail-on-issues
```

9.4.3 CI/CD Integration

```
- name: Validate renv consistency
  run: Rscript check_renv_for_commit.R --fail-on-issues --quiet
```

This approach ensures that collaborators can reliably reproduce your package environment and that CI/CD pipelines have all necessary dependency information.