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

R.G. Thomas

2025-06-30

This white paper presents an approach to achieving reproducibility in R workflows by combining three 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 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 an approach to solving this problem by combining three 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 a colleague
2. They attempt to run your analysis with the commands you provided
3. But R isn't installed on their system
4. After installing R, they get an error: "could not find function 'render' " since they don't have the `rmarkdown` package installed
5. They install the rmarkdown package and run 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 they 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 systematic 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/username/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 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, 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 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 a Docker-first framework that creates reproducible research compendia with containerized development workflows. The framework extends the research compendium concept introduced by Ben Marwick's rrtools, adding container-based development and automated dependency validation. Team members install zzrrtools once on their system, then can create or join any zzrrtools-based project using the same framework. A research compendium organizes digital research materials to enable others to inspect, reproduce, and extend the research.

2.2 Key Features of zzrrtools

zzrrtools creates containerized research compendia with these key features:

- **Docker-first development**: All workflows operate within containers, eliminating "works on my machine" issues
- **Centralized framework**: One-time zzrrtools installation enables consistent project creation and team collaboration
- **Multi-service architecture**: Provides specialized Docker environments for interactive R sessions, shell development, and paper rendering
- **Flexible base images**: Choice of minimal (rocker/r-ver) or pre-packaged (rgt47/r-pluspackages) Docker templates with common R packages
- **Advanced dependency validation**: Automated renv consistency checking with CRAN verification and pre-commit validation
- **Shell-based workflows**: Optimized for command-line development with rich automation via Make targets
- **Team collaboration focus**: Designed for multi-developer teams working on shared research projects

2.3 zzrrtools Workflow: Self-Replicating Collaboration

The zzrrtools workflow is designed for team collaboration:

```
# Project creator initializes the research compendium
~/prj/zzrrtools/zzrrtools.sh [OPTIONS]

# Available options:
#   --dotfiles DIR       Copy personal dotfiles (with leading dots)
#   --dotfiles-nodot DIR Copy dotfiles (without leading dots)
#   --base-image NAME    Use custom Docker base image (default: rocker/r-ver)
#                        Popular options: rgt47/r-pluspackages (includes tidyverse)
#   --no-docker          Skip Docker image build
#   --next-steps         Show workflow guidance
```

This creates a Docker-first research compendium with:

- **Framework-based setup** using centrally installed zzrrtools for consistent project creation
- **Multi-service Docker architecture** for interactive R, shell, and rendering
- **Advanced renv validation** with CRAN verification and pre-commit checking
- **Dual-track automation** supporting both native R and Docker workflows
- **Navigation shortcuts** via symbolic links (a→data, p→paper, s→scripts)
- **Developer environment integration** with personal dotfiles support
- **Team synchronization** via automated dependency validation

Team members can collaborate after one-time zzrrtools installation:

```
# One-time zzrrtools installation
git clone zzrrtools-repo ~/prj/zzrrtools
cd ~/prj/zzrrtools && ./install.sh  # Creates zzrrtools command in PATH

# Per-project workflow
git clone project-repo && cd project-repo
zzrrtools  # Set up project environment (command available anywhere)
make docker-r   # Start interactive R session
```

2.3.1 Integrated renv Consistency Checking

The workflow includes advanced renv management through the `check_renv_for_commit.R` script, which provides automated dependency validation and team conflict prevention. This script:

- **Scans multiple directories** (R/, scripts/, analysis/) for package dependencies
- **Validates against CRAN** to ensure packages exist and are properly named

- **Synchronizes dependencies** across code files, DESCRIPTION, and renv.lock
- **Provides automated fixes** to maintain team environment consistency
- **Integrates with CI/CD** for fail-fast validation workflows

Usage examples:

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

# Auto-fix dependency issues
Rscript check_renv_for_commit.R --fix

# CI/CD validation with fail-fast
Rscript check_renv_for_commit.R --fix --fail-on-issues --quiet

# Via Make targets (recommended)
make check-renv          # Check dependencies
make check-renv-fix      # Fix dependency issues
make docker-check-renv-fix  # Fix in container
```

This 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 directory structure that follows research compendium best practices. The structure includes organized data folders, analysis directories, testing frameworks, and 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**: R package organization with documentation and testing
- **Integration support**: Works 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, team members 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    # Additional analysis
   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")
}

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:

```
# 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**: Project maintainer runs `zzrrtools.sh` to create project structure
2. **Immediate containerization**: Build Docker container and switch to container-based development from day one
3. **Initial development**: Create exploratory scripts in `scripts/` directory **inside the container**
4. **Collaborative iteration**: Team members clone repo, build identical container, add 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:

- **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
- **Simple 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 directory structure and R package framework created by zzrrtools becomes most useful 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 complete 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 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"]
```

This configuration provides a minimal R installation with LaTeX support for PDF rendering and secure non-root user execution.

Docker Compose Integration:

The setup also includes a 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 complete 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 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 workflow integrates zzrrtools, renv, and Docker with a clear governance structure suitable for multi-developer research teams:

Project Maintainer Role: - Creates and maintains the research compendium structure - Manages renv environment and package dependencies using 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 data organization
   - Initialize renv environment with `renv::init()`
   - Create Dockerfile with container configuration
2. **Establish Development Environment (Maintainer)**:
   - Install required packages and develop initial analysis
   - Create 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 Docker Hub:

   Docker Hub provides public image hosting that enables reproducible research by sharing computational environments while protecting private research code in GitHub.

   Docker Hub (Recommended for Reproducible Research)

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

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

# Push to GitHub Container Registry (automatically private)
docker push ghcr.io/username/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 username \
  --password-stdin
docker build -t ghcr.io/username/penguins_analysis:v1.0 .
docker push ghcr.io/username/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**: Direct authentication in CI/CD
- **Simple team sharing**: Repository collaborators automatically have access
- **Package management**: Integrated with GitHub Packages ecosystem

Docker Workflow:

The zzrrtools setup provides multiple approaches for working with containers, from simple Make commands to direct Docker execution. Make commands offer 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 Docker workflow options including Docker Compose and direct commands, see Appendix D: 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

A team of data scientists collaborates on Palmer Penguins analysis using zzrrtools' Docker-first workflow that eliminates environment setup friction:

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

- **Additional team members**: Can join without any local R installation

The collaboration model emphasizes **zero-setup team onboarding** through self-replicating project distribution and containerized development environments.

> **Collaboration Philosophy**: - **Self-contained projects**: Each repository includes its own setup script - **Container-first development**: All work happens in identical Docker environments - **Automated dependency validation**: Pre-commit checks prevent conflicts - **Shell-based workflows**: Command-line tools for maximum flexibility

Key Workflow Principles: - Joe initializes with zzrrtools and commits the setup script to the repository - Team members run the included script - no separate installation required - All development occurs in containers for perfect environment consistency - Dependency changes are validated before commits to prevent team conflicts - Each developer works in identical environments regardless of host system

6.2 Implementation Example

6.2.1 Project Maintainer Setup (Joe)

Joe initializes a private GitHub repository and creates the containerized research environment:

```
# Create repository and initialize zzrrtools
git clone https://github.com/joe/penguins_analysis.git
cd penguins_analysis
~/prj/zzrrtools/zzrrtools.sh --dotfiles ~/.config/shell \
                             --base-image rgt47/r-pluspackages
```

This creates: - Research compendium directory structure - Docker container configuration - renv dependency management - Automated validation scripts - GitHub Actions workflows

For a detailed view of the complete directory structure created by zzrrtools, see Appendix C: Directory Structure.

Joe completes the setup and begins development:

```
# Validate dependencies and build container
make check-renv
make docker-build

# Start container-based development
make docker-r
```

For detailed information on renv dependency validation, troubleshooting, and team collaboration workflows, see Appendix G: renv Management and Validation.

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.

For a complete test suite with data validation, statistical tests, and integration tests, see Appendix B: Test Suite.

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

zzrrtools generates a Dockerfile with multiple template options. The standard template uses rocker/r-ver, while the pluspackages template includes common R packages like tidyverse. Both provide:

- **R version consistency**: Matches exact R version specified in renv.lock
- **Development environment**: Includes zsh, vim, tmux, Node.js for plugin support
- **Security**: Non-root user execution with proper file permissions
- **TinyTeX integration**: LaTeX support for PDF rendering (pluspackages template)
- **Pre-installed packages**: Common packages like tidyverse, DT, testthat (pluspackages template)

The generated Dockerfile includes development tools and optimizations:

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

# Install comprehensive development environment
RUN apt-get update && apt-get install -y \
    git ssh curl wget vim tmux zsh build-essential \
    libcurl4-openssl-dev libssl-dev libxml2-dev \
    libfontconfig1-dev libharfbuzz-dev libfribidi-dev \
    libfreetype6-dev libpng-dev libtiff5-dev libjpeg-dev \
    man-db pandoc \
    && rm -rf /var/lib/apt/lists/*

# Install Node.js for vim plugins
RUN curl -fsSL https://deb.nodesource.com/setup_lts.x | bash - && \
    apt-get install -y nodejs

# Create non-root user with zsh shell
```

```
ARG USERNAME=analyst
RUN useradd --create-home --shell /bin/zsh ${USERNAME}

# Set up R environment with renv
WORKDIR /home/${USERNAME}/project
COPY --chown=${USERNAME}:${USERNAME} renv.lock ./
USER ${USERNAME}
RUN R -e "install.packages('renv'); renv::restore()"

# Copy project files and install
COPY --chown=${USERNAME}:${USERNAME} . .
RUN R -e "devtools::install('.')"

CMD ["/bin/zsh"]
```

Alternative: Pre-packaged Template

For projects using common packages, the pluspackages template includes TinyTeX and popular R packages:

```
# Install TinyTeX for PDF rendering
RUN R -e "install.packages('tinytex')" && \
    R -e "tinytex::install_tinytex()" && \
    /root/.TinyTeX/bin/*/tlmgr path add

# Install common R packages (cached layer)
RUN R -e "install.packages(c('renv', 'remotes', 'devtools', \
    'testthat', 'naniar', 'DT', 'conflicted', 'ggthemes', \
    'datapasta', 'janitor', 'kableExtra', 'tidytuesdayR', \
    'tidyverse'), repos = c(CRAN = 'https://cloud.r-project.org'))"

# Give user write permissions to R library
RUN chown -R ${USERNAME}:${USERNAME} /usr/local/lib/R/site-library
```

For production Dockerfiles with development environment configuration (zsh, vim plugins, dotfiles integration), see Appendix F: 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 from the renv lockfile:

```
# Extract R version from renv.lock
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 use `rocker/r-ver:4.4.0`. This maintains 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, GitHub Actions automatically rebuilds the Docker image with updated renv.lock specifications and pushes the updated environment to Docker Hub for team access. This ensures collaborators have access to the identical development environment.

Step 10: Prepare for Team Handoff

Before handing off to team members, Joe must complete several critical steps:

```
# 1. Validate all dependencies are properly captured
make check-renv-fix

# 2. Run complete test suite to ensure everything works
make docker-test

# 3. Render paper to verify end-to-end workflow
make docker-render

# 4. Build and tag the Docker image
make docker-build
docker tag penguins_analysis ghcr.io/joe/penguins_analysis:v1.0

# 5. Push Docker image to Docker Hub for team access
echo $GITHUB_TOKEN | docker login ghcr.io -u joe --password-stdin
docker push ghcr.io/joe/penguins_analysis:v1.0
```

```
# 6. Commit all setup files and push to repository
git add .
git commit -m "Initial research compendium setup with Docker environment"
git push origin main
```

Step 11: Enable Team Access

Joe configures repository permissions for team collaboration:

1. **Repository Settings** → **Collaborators** → Add team members
2. **Grant "Write" access** to enable forking and pull requests
3. **Share repository URL** and Docker Hub image name with team
4. **Document onboarding process** in README or team communication

At this point, Joe has established a reproducible research framework ready for collaborative development. The containerized environment is available via the registry, and team members can join with zero local setup requirements.

6.2.2 Team Member Onboarding (Sam)

What Sam Receives from Joe: - Repository URL: `https://github.com/joe/penguins_analysis` - Container registry access (GitHub Personal Access Token) - Brief project overview and development guidelines

Sam's Onboarding Process:

```
# 1. Clone the zzrrtools framework
git clone https://github.com/username/zzrrtools.git ~/prj/zzrrtools
cd ~/prj/zzrrtools
./install.sh  # Creates zzrrtools command in PATH

# 2. Fork and clone the project repository
git clone https://github.com/sam/penguins_analysis.git  # Sam's fork
cd penguins_analysis

# 3. Run zzrrtools setup with personal preferences
zzrrtools --dotfiles ~/.config/shell  # Customize with personal dotfiles
# Or use other options:
# zzrrtools --base-image rgt47/r-pluspackages  # Use pre-packaged template
# zzrrtools --no-docker                       # Skip Docker build
# zzrrtools --next-steps                      # Show workflow guidance
```

```
# 4. Pull the pre-built Docker environment
docker pull ghcr.io/joe/penguins_analysis:v1.0

# 5. Start development in identical environment
make docker-r
```

Key Advantage: After one-time zzrrtools installation, Sam has access to the framework for all future projects. The development environment setup becomes a simple script execution.

Sam develops new analysis components within the same containerized environment, ensuring identical results across team members.

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

When Sam completes the analysis iteration, the submission process follows these steps:

1. **Validate dependencies**: Run `make docker-check-renv-fix` to ensure package consistency
2. **Run complete test suite**: Execute `make docker-test` to verify all tests pass
3. **Verify paper rendering**: Run `make docker-render` to confirm analysis integrates properly
4. **Commit changes**:

   ```
   git add .
   git commit -m "Add body mass analysis and associated tests"
   git push origin feature/body-mass-analysis
   ```

5. **Create pull request**: Submit pull request from Sam's fork to Joe's original repository
6. **Notify Joe**: Alert project maintainer about new packages or Docker changes needed

Note: Only Joe (project maintainer) can accept pull requests. The official Docker image is automatically rebuilt and pushed to Docker Hub by GitHub Actions when package dependencies change.

CI Feedback Loop: If the CI workflow fails (e.g., renv validation issues), Sam receives automatic GitHub notifications and can view detailed failure logs to fix the issues before Joe reviews the PR.

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 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.

6.3 Complete Handoff Workflow Summary

Initiating Developer (Joe) Responsibilities: 1. Run `zzrrtools.sh` to create research compendium 2. Develop initial analysis and create tests 3. Validate dependencies with `make check-renv-fix`
4. Build initial Docker image and configure automated rebuilds via GitHub Actions 5. Push team Docker image to Docker Hub (public registry for reproducibility) 6. Commit all files and push to repository 7. Grant team member repository access
8. Share repository URL and Docker Hub image name

Joining Developer (Sam) Process: 1. Receive repository URL and Docker Hub image name from Joe 2. Clone zzrrtools framework: `git clone zzrrtools ~/prj/zzrrtools` 3. Install zzrrtools: `cd ~/prj/zzrrtools && ./install.sh` (creates `zzrrtools` command in PATH) 4. Fork and clone project repository 5. Pull pre-built team Docker image: `docker pull [TEAM]/project:latest` (from Docker Hub) 6. Start development immediately: `make docker-r` (no local setup needed) 7. Submit contributions via pull requests to private repository

Key Success Factor: The containerized environment and centralized zzrrtools framework eliminate project-specific configuration requirements for team members after one-time framework installation.

6.4 Collaboration Results

This workflow achieves: - Identical development environments across team members - Dependency validation preventing conflicts - Standardized project structure - Automated testing and CI/CD integration

For GitHub Actions setup instructions, workflow examples, and CI/CD configuration, see Appendix E: 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 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. **Test regularly**: Use CI/CD pipelines to 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

7.5 Troubleshooting Common Issues

Docker Build Failures: - Try: `export DOCKER_BUILDKIT=0` (disable BuildKit) - Check Docker has sufficient memory/disk space - Ensure Docker is running and up to date

Platform Warnings on ARM64/Apple Silicon: - Use updated Makefile with `--platform linux/amd64` flags - Or set: `export DOCKER_DEFAULT_PLATFORM=linux/amd64`

Permission Errors in Container: - Rebuild image after copying dotfiles - Check file ownership in project directory

Package Name Errors: - Ensure directory name contains only letters/numbers/periods - Avoid underscores and special characters

Missing Dotfiles in Container: - Use `--dotfiles` or `--dotfiles-nodot` flag during setup - Rebuild Docker image after adding dotfiles

Despite these challenges, the benefits of reproducible research outweigh the implementation costs, particularly for collaborative and long-term projects. The approach described in this white paper provides a foundation for achieving reproducibility that meets the standards expected in 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 testing for code validation, data scientists and researchers can create truly portable, reproducible, and reliable workflows.

The 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 method for long-term reproducibility in R, meeting the standards required for data science and research documentation. The combination of standardized research compendium structure, dependency management, and containerized environments creates a foundation for reproducible research. By adopting this approach, the R community can make progress toward the goal of 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 (Extended 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 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

Note: GitHub now offers two token types: - **Fine-grained personal access tokens** (recommended for new projects) - **Personal access tokens (classic)** (for broader compatibility)

2. Create New Token: - Click "Generate new token" and select the appropriate type - 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 project maintainer* - **read:packages** (Download Docker images from GitHub Container Registry) - *Required for all team members* - **delete:packages** (Delete packages from GitHub Package Registry) - *Optional but recommended*

Note: Team members only need `read:packages` and `repo`, but the project maintainer needs all container permissions to push Docker images.

Token Type Recommendation: Use fine-grained personal access tokens for new projects as they provide better security and more precise permissions.

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

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

# Login to container registry (works 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 B: Comprehensive Test Suite

This appendix provides a 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
# 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
```

```r
  # 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")
```

```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")
```

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

Appendix C: Directory Structure

The zzrrtools setup creates the following directory structure:

```
project/
   DESCRIPTION              # Package metadata and dependencies
   LICENSE                  # Project license
   README.md               # Project documentation
   project.Rproj           # R 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
   ZZRRTOOLS_USER_GUIDE.md # Comprehensive user documentation
   .zshrc_docker           # zsh configuration for Docker container
   .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:

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

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

Workflows: - **.github/workflows/**: GitHub Actions for testing, checking, and rendering - **setup_renv.R**: Package environment setup - **RRTOOLS_USER_GUIDE.md**: 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 research projects while maintaining clear organization and following established research compendium principles.

Appendix D: Docker Workflow Options

The zzrrtools setup provides a multi-service Docker architecture optimized for collaborative shell-based development workflows.

9.1 Multi-Service Docker Architecture

zzrrtools creates specialized Docker environments for different development tasks:

```
# Build the containerized research environment
make docker-build

# Interactive R console (primary development environment)
make docker-r

# Development shell with personal dotfiles
make docker-zsh

# Interactive bash session
make docker-bash

# RStudio Server (web-based IDE)
make docker-rstudio    # Access at http://localhost:8787

# Render research paper
make docker-render

# Run tests
make docker-test

# Package checking
make docker-check

# renv dependency validation
```

```
make docker-check-renv-fix

# See all available commands
make help
```

Collaborative Benefits: - **Zero-setup onboarding**: Team members run identical commands - **Consistent environments**: Same container across all developer machines - **ARM64/Apple Silicon support**: Platform-specific flags ensure compatibility - **Shell-optimized workflows**: Command-line development with rich tooling - **Personal customization**: Dotfiles integration for familiar environments - **Web-based development**: Optional RStudio Server for GUI-based workflows

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 setup**: 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/username/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/username/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/username/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/username/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/username/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: - **Full flexibility**: Complete 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 E: GitHub Actions CI/CD Setup

GitHub Actions provides testing and deployment for research compendia. This appendix covers 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) tests your research code whenever changes are made. For research compendia, this means:

- **Testing**: Every push triggers your test suite
- **Environment consistency**: Tests run in identical Docker environments
- **Early error detection**: Problems caught 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
```

```yaml
    - 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`:

```yaml
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
```

```yaml
      - 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`:

```yaml
name: Render Research Paper

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

```
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 }}

      - 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
- **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 if dependency issues are detected

9.3.2 R Package Check Features:

- **Multi-platform Testing**: Tests on Ubuntu, macOS, and Windows
- **R CMD Check**: Package validation
- **renv Integration**: 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:

- **Building**: Builds on pushes and tags
- **Multi-platform**: Supports AMD64 and ARM64 platforms
- **Semantic Versioning**: 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 a team member 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 CI/CD setup ensures that research compendia remain reproducible, tested, and deployment-ready throughout the development lifecycle.

Appendix F: Docker Configuration Examples

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

9.1 Production Dockerfile

The following Dockerfile provides a 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 Production 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
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/username/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 use `rocker/r-ver:4.4.0`. This maintains consistency between the package environment and system environment.

Appendix G: renv Management and Validation

This appendix provides 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 advanced team collaboration features through dependency validation:

- **Team conflict prevention**: Pre-commit validation stops dependency inconsistencies before they reach the repository
- **Automated dependency discovery**: Scans R/, scripts/, and analysis/ directories for `library()`, `require()`, and `pkg::` calls
- **Multi-source synchronization**: Ensures packages are consistent across code files, DESCRIPTION, and renv.lock
- **CRAN validation**: Verifies packages exist and are properly named before team integration
- **Automatic fixing**: Updates DESCRIPTION and regenerates renv.lock to maintain team synchronization
- **CI/CD fail-fast**: Provides proper exit codes for automated workflows
- **Interactive collaboration mode**: Guides developers through dependency resolution during development

9.2 Team Collaboration Commands

```
# Team development workflow (via Make)
make check-renv          # Interactive dependency checking
make check-renv-fix      # Auto-fix dependency issues
make check-renv-ci       # CI/CD validation with fail-fast

# Docker-based validation (no local R required)
make docker-check-renv-fix  # Fix dependencies in container

# Direct script usage
Rscript check_renv_for_commit.R --fix --fail-on-issues  # CI mode
Rscript check_renv_for_commit.R --quiet                 # Minimal output
Rscript check_renv_for_commit.R --help                  # Usage info
```

9.3 Multi-Developer Workflow

1. **Install packages in container**: Use `install.packages()` or `renv::install()` within Docker environment
2. **Validate team dependencies**: Run `make check-renv` to check for conflicts before committing

3. **Review team impacts**: Script identifies packages that would affect other team members
4. **Synchronize team environment**: Use `make check-renv-fix` to update shared dependency files
5. **Commit with team confidence**: Other developers can reproduce your exact environment

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.