Reproducible Blog Posts with ZZCOLLAB: A Quarto Workflow

Ronald G. Thomas

2026-02-10

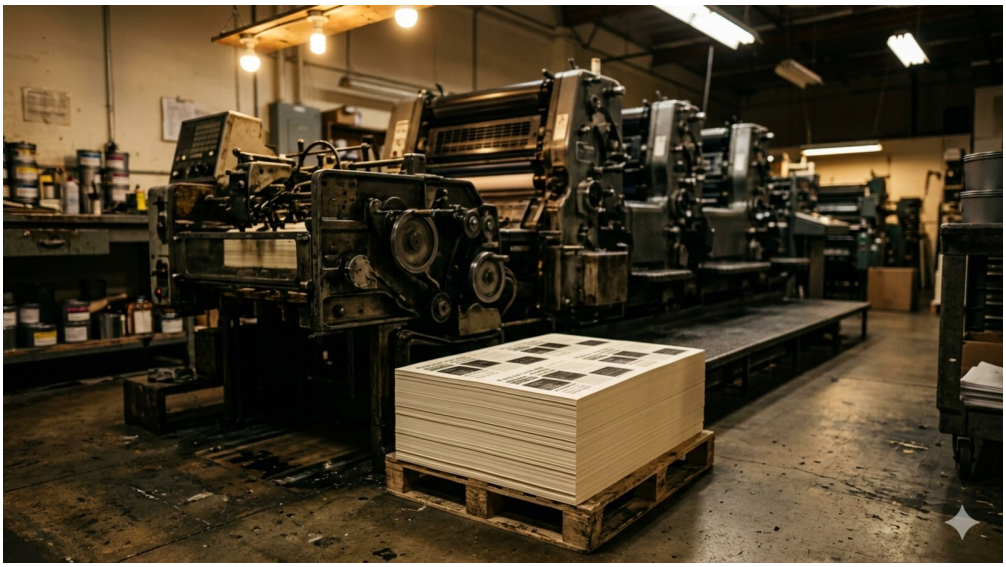

Figure 1: A Quarto-themed workspace representing reproducible document authoring

Quarto provides the rendering engine; ZZCOLLAB provides the reproducible scaffolding around it.

Introduction

I did not really know how much infrastructure a blog post needed until I returned to one of my own R-based posts six months after writing it and watched it fail to render. A package had updated, a system library had shifted, and the code that once produced clean diagnostic plots now threw cryptic errors. The rendered HTML was frozen in time, but the source code was dead.

Technical blog posts that embed R code face a well-documented fragility problem. Package updates introduce breaking changes, data sources move or disappear, and system-level dependencies shift beneath the analysis. A post that rendered correctly in 2024 may fail silently in 2025, producing different results or refusing to compile entirely.

The conventional response is to freeze rendered output and accept that the source code is a snapshot rather than a living document. We describe an alternative: treating each blog post as a standalone ZZCOLLAB reproducible research project, complete with its own Docker environment, pinned package versions, and continuous integration pipeline.

Motivations

- I was frustrated that returning to a six-month-old post required hours of debugging broken package dependencies before I could even render it again.
- I wanted each blog post to be independently clonable so that a collaborator could reproduce the results without asking me what versions of R and ggplot2 I originally used.
- I needed a workflow that scaled to 40+ posts without requiring a monorepo that coupled unrelated analyses together.
- I was already using ZZCOLLAB for research projects and wanted a consistent mental model across all my computational work, whether a journal article or a tutorial post.
- I wanted CI/CD to catch breakage immediately on push rather than discovering it months later when a reader reports a dead link or a wrong figure.

Objectives

1. Document the lead-in directory pattern that separates git repositories from archival material.
2. Explain the dual-symlink architecture that reconciles Quarto discovery with rrtools/ZZCOLLAB conventions.
3. Walk through the steps for creating, initializing, and publishing a new blog post project.
4. Describe the five-pillar reproducibility framework as applied to the blog post context.

I am documenting my learning process here. If you spot errors or have better approaches, please let me know.

Figure 2: A wooden puzzle piece resting exactly in the seam between two halves of a map: narrative and code fitting together into one document.

What is a Reproducible Blog Post?

A reproducible blog post is a technical document whose computational results can be regenerated from source by anyone, on any machine, at any point in the future. It goes beyond simply sharing code alongside prose. Concretely, a reader can clone a single repository, build a Docker container, and produce an identical HTML document without installing anything beyond Docker itself.

The distinction matters because most technical blog posts are reproducible only in theory. They share code snippets but omit the system libraries, the exact package versions, and the data preparation steps that produced the published figures. A truly reproducible blog post packages all of these elements together, much as a research compendium packages the materials needed to reproduce a journal article.

The Problem

Consider a blog post that fits a linear model to the Palmer Penguins dataset and generates diagnostic plots. The post depends on:

- A specific R version (4.5.1)
- The `palmerpenguins` package (for data)
- `ggplot2`, `broom`, `patchwork` (for analysis)
- System libraries for PNG rendering (`libpng`, `cairo`)
- Quarto itself (for HTML generation)

Any of these may change independently. Without explicit version management, reproducing the post requires reconstructing the original environment through trial and error.

The Solution

Each blog post becomes a self-contained project with five components:

1. **Dockerfile**: defines the computational environment
2. **renv.lock**: pins exact R package versions
3. **.Rprofile**: configures R session behavior
4. **Source code**: analysis scripts and narrative
5. **Data**: raw and derived datasets

A collaborator (or the author, six months later) can clone the repository, run `make docker-build && make r`, and reproduce the post in an identical environment.

The Lead-In Pattern

Blog post projects follow the same directory convention used across all research software projects in this lab:

```
~/Dropbox/prj/qblog/posts/
+-- 29-setupquarto/          # lead-in directory
|   +-- archive/             # old drafts, artifacts
|   +-- setupquarto/         # git repo (the project)
|       +-- .git/
|       +-- Dockerfile
|       +-- index.qmd -> analysis/report/index.qmd
|       +-- ...
```

The **lead-in directory** (`29-setupquarto/`) is not managed by git. It serves as a container for the git repository and an `archive/` directory for superseded drafts, exploratory notebooks, and other material that does not belong in version control.

The **project directory** (`setupquarto/`) is the git repository. Its name matches the GitHub repository name and omits the numeric prefix.

This pattern mirrors the layout used for research software under `~/prj/sfw/`:

```
~/prj/sfw/07-zzcollab/
|   +-- archive/
|   +-- zzcollab/            # git repo
```

The numeric prefix provides chronological ordering in file browsers without polluting repository names.

Creating a New Post

Step 1: Create the Lead-In

```
mkdir -p \
  ~/Dropbox/prj/qblog/posts/43-newpost/archive
cd ~/Dropbox/prj/qblog/posts/43-newpost
```

Step 2: Initialize the ZZCOLLAB Project

Copy the scaffold from an existing post or use the ZZCOLLAB CLI:

```
# Option A: Copy from template
cp -r \
  ~/Dropbox/prj/qblog/posts/39-templatepost/\
templatepost \
  ./newpost

# Option B: Use zzcollab CLI
zzc analysis newpost
```

Both approaches produce the same directory structure.

Step 3: Write the Post

The blog post content lives at `analysis/report/index.qmd`. A root-level symlink provides Quarto compatibility:

```
# Already created by scaffold:
# newpost/index.qmd -> analysis/report/index.qmd
```

Open `analysis/report/index.qmd` and write. Analysis code that produces figures or derived data belongs in `analysis/scripts/`. Reusable utility functions belong in `R/`.

Step 4: Initialize Git and Push

```
cd newpost
git init -b main
git add .
git commit -m \
  "Initial scaffold for blog post: newpost"
gh repo create rgt47/newpost --public \
```

```
    --source=. --remote=origin
git remote set-url origin \
    git@github.com:rgt47/newpost.git
git push -u origin main
```

Each blog post is its own GitHub repository. Collaborators clone a single post and receive the complete reproducible environment.

Connecting to Quarto

The Quarto site lives at `~/Dropbox/prj/qblog/`. It discovers blog posts through a glob pattern in `blog/index.qmd`:

```
# blog/index.qmd
listing:
  contents: "../posts/*/*/index.qmd"
  type: default
  sort: "date desc"
  categories: true
```

The double-wildcard `posts/*/*/index.qmd` matches the lead-in/project nesting. Quarto resolves the `index.qmd` symlink transparently and reads the YAML front matter to populate the blog listing.

How Quarto Discovers Posts

The resolution chain for a blog post:

```
blog/index.qmd
  +-- glob: ../posts/*/*/index.qmd
      +-- matches:
          posts/29-setupquarto/setupquarto/index.qmd
          +-- symlink: analysis/report/index.qmd
              +-- actual file with YAML front matter
```

Quarto reads the `title`, `date`, `categories`, `description`, `image`, and `draft` fields from the YAML header. Posts with `draft: false` are excluded from the listing.

Site Configuration

The site-level `_quarto.yml` defines navigation, theme, and rendering defaults. It does not enumerate posts; the listing glob handles discovery automatically. Adding a new post requires only creating the lead-in directory and project, with no modification to `_quarto.yml` or any other site-level file.

GitHub Integration

One Repository per Post

Each blog post is a standalone GitHub repository under the `rgt47` organization. This design provides:

- **Independent version history**: each post tracks its own commits without polluting a monorepo log
- **Isolated CI/CD**: a failing build in one post does not block others
- **Focused collaboration**: a contributor clones only the post they are reviewing
- **Independent dependency management**: each post pins its own package versions

CI/CD Pipeline

Each repository includes `.github/workflows/blog-render.yml`, which executes on push:

1. Build the Docker image from `Dockerfile`
2. Restore R packages from `renv.lock`
3. Run unit tests (`tests/testthat/`)
4. Execute analysis scripts (`analysis/scripts/`)
5. Render the Quarto report to HTML
6. Upload the rendered artifact

The pipeline confirms that the post renders successfully in a clean environment. If a package update breaks the build, the failure is detected immediately rather than months later.

Collaborator Workflow

A collaborator reproducing or reviewing a post:

```
git clone \
  git@github.com:rgt47/setupquarto.git
cd setupquarto
make docker-build    # build the Docker image
make r               # enter the container
# Inside the container:
Rscript analysis/scripts/01_prepare_data.R
quarto render analysis/report/index.qmd
```

The `Makefile` automates common operations. `make r` mounts the project directory into the container and drops the user into an R-ready shell.

The Five Pillars in Blog Context

ZZCOLLAB's reproducibility framework rests on five components. Each serves a specific role in the blog post context:

1. **Dockerfile** Defines the base R version (rocker/tidyverse:4.5.1), system dependencies, and the `analyst` user. Ensures the computational environment is identical across machines and CI/CD runners.
2. **renv.lock** Pins exact package versions. A minimal lockfile (172 bytes) specifies only the R version and CRAN repository; `renv::install()` populates it from the `DESCRIPTION` file on first use. A full lockfile captures the complete dependency tree.
3. **.Rprofile** Configures R session behavior. Activates `renv`, sets repository mirrors, enables auto-snapshot, and detects whether the session is running inside a ZZCOLLAB container.
4. **Source code** Analysis scripts in `analysis/scripts/` produce figures and derived data. The narrative in `analysis/report/index.qmd` loads pre-computed results rather than running analysis inline. This separation ensures that rendering the narrative is fast and that analysis code is independently testable.
5. **Data** Raw data in `analysis/data/raw_data/` is immutable. Derived data in `analysis/data/derived_data/` is produced by analysis scripts and may be regenerated. Each dataset includes a `README.md` documenting its provenance.

Directory Structure Reference

Complete annotated structure of a blog post project:

```
setupquarto/                    # git repo root
+-- .git/
+-- .github/
|   +-- workflows/
|       +-- blog-render.yml     # CI/CD pipeline
+-- .gitignore
+-- .Rbuildignore
+-- .Rprofile                   # renv activation
+-- .zzcollab/                  # zzcollab metadata
+-- analysis/
|   +-- data/
|   |   +-- raw_data/           # immutable inputs
|   |   +-- derived_data/       # script outputs
|   +-- figures/                # generated plots
|   +-- media/
|   |   +-- images/             # photos, diagrams
|   +-- report/
|   |   +-- index.qmd           # the blog post
|   |   +-- data -> ../data     # convenience
|   |   +-- figures -> ../figures
|   |   +-- media -> ../media
```

```
|   +-- scripts/
|       +-- 01_prepare_data.R
|       +-- 02_fit_models.R
|       +-- 03_generate_figures.R
+-- R/                          # reusable functions
+-- tests/
|   +-- testthat/               # unit tests
|   +-- integration/            # pipeline tests
+-- Dockerfile
+-- Makefile
+-- DESCRIPTION                 # R package metadata
+-- NAMESPACE
+-- renv.lock
+-- renv/
|   +-- activate.R
+-- index.qmd -> analysis/report/index.qmd
+-- data -> analysis/data
+-- figures -> analysis/figures
+-- media -> analysis/media
```

Figure 3: Three glass jars of different colored liquids beside an empty jar and funnel: combining R, prose, and output into one compendium

Reproducible workflows require careful architecture, much like building a well-organized library.

The Symlink Architecture

The project uses two layers of symlinks to reconcile three competing requirements. First, Quarto expects `index.qmd` at the directory root it discovers via glob. Second, the rrtools/ZZCOLLAB convention places the narrative at `analysis/report/index.qmd`. Third, relative paths in the `.qmd` file must resolve to `analysis/data/`, `analysis/figures/`, and `analysis/media/` regardless of which directory Quarto considers the "working directory" during rendering.

Layer 1: Root-Level Symlinks

```
setupquarto/                  # git repo root
+-- index.qmd -> analysis/report/index.qmd
+-- data -> analysis/data
+-- figures -> analysis/figures
+-- media -> analysis/media
```

Purpose: Quarto's blog listing glob (`../posts/*/*/index.qmd`) matches files at the project root. The `index.qmd` symlink makes the post discoverable without duplicating the file.

The `data/`, `figures/`, and `media/` symlinks allow Quarto to resolve relative image and data paths when it renders from the project root. If the `.qmd` references `figures/eda-overview.png`, Quarto resolves it through the root-level `figures/` symlink to `analysis/figures/eda-overview.png`.

Git behavior: Git stores symlinks as text files containing the relative target path. On clone, git recreates the symlinks. The `git add .` command captures symlinks automatically:

```
$ git ls-files -s index.qmd
120000 <hash> 0   index.qmd
```

The `120000` mode indicates a symbolic link.

Layer 2: Report-Level Symlinks

```
analysis/report/
+-- index.qmd            # the actual blog post
+-- data -> ../data      # resolves to analysis/data
+-- figures -> ../figures # resolves to
|                         # analysis/figures
+-- media -> ../media     # resolves to
                          # analysis/media
```

Purpose: When an author edits `index.qmd` inside `analysis/report/` and uses a relative path like `data/derived_data/results.csv`, the `data` symlink resolves to `../data` (i.e., `analysis/data`). This ensures paths work correctly whether Quarto renders from the project root (following the root-level symlink) or from `analysis/report/` directly.

Without these symlinks, an `![](figures/plot.png)` reference in `index.qmd` would resolve differently depending on the rendering context:

- From root: `setupquarto/figures/plot.png` (works via root symlink)
- From `analysis/report/`: `analysis/report/figures/plot.png` (fails: no such directory)

The report-level symlinks make both contexts resolve to `analysis/figures/plot.png`.

Layer 3: The Quarto Site Glob

At the site level, Quarto discovers posts through the listing in `blog/index.qmd`:

```yaml
listing:
  contents: "../posts/*/*/index.qmd"
```

The resolution chain:

```
blog/index.qmd (listing page)
  |
  +-- glob: ../posts/*/*/index.qmd
      |
      +-- posts/29-setupquarto/    <- lead-in
      |   +-- setupquarto/         <- project
      |       +-- index.qmd        <- symlink
      |           +-- target:
      |               analysis/report/index.qmd
      |               +-- YAML: title, date, ...
      |
      +-- posts/08-palmerpenguins.../
          +-- palmerpenguinspart1/
              +-- index.qmd ->
                  analysis/report/index.qmd
```

A critical constraint discovered during development: **Quarto's listing glob does not follow symlinked directories.** If `posts/29-setupquarto/` were itself a symlink to an external path, Quarto would skip it entirely. The lead-in directory must be a real directory; only the `index.qmd` file inside may be a symlink. This is why the project lives inside `qblog/posts/` rather than being symlinked from an external location.

Why Not a Single Symlink?

An earlier design placed project repositories outside the Quarto site (at `~/prj/blog/NN-name/name/`) and used directory-level symlinks:

```
# DOES NOT WORK:
# Quarto skips symlinked directories
posts/29-setupquarto ->
  ~/prj/blog/29-setupquarto/setupquarto/
```

Quarto's internal file discovery skips symlinked directories during glob matching. Individual posts could still be rendered by explicit path (`quarto render posts/29-setupquarto/index.qmd`), but the blog listing page would not discover them. This forced the current architecture where project directories reside physically inside the qblog tree.

Image Path Resolution

Images referenced in the `.qmd` file demonstrate the symlink chain in action. For project-local images stored in `analysis/media/images/`:

```
![](media/images/diagram.png)
```

This resolves through the report-level symlink: `media -> ../media -> analysis/media/images/diagram.png`.

Creating the Symlinks

The migration script creates both layers:

```
# Root-level symlinks
ln -s analysis/report/index.qmd index.qmd
ln -s analysis/data data
ln -s analysis/figures figures
ln -s analysis/media media

# Report-level symlinks
cd analysis/report
ln -s ../data data
ln -s ../figures figures
ln -s ../media media
```

These are relative symlinks, which is critical for portability. An absolute symlink (`/Users/zenn/Dropbox/...`) would break on any other machine. Relative symlinks survive cloning, moving the project, and CI/CD environments.

Verifying Symlink Integrity

To confirm symlinks are correctly configured:

```
# From the project root:
ls -la index.qmd data figures media
# Should show -> targets

# From analysis/report/:
ls -la data figures media
# Should show -> ../data, ../figures, ../media

# Verify resolution:
file index.qmd
# Should show: "symbolic link to ..."

# Verify the target exists:
readlink -f index.qmd
# Should show the absolute path to the actual file
```

Things to Watch Out For

1. **Symlinks on Windows.** Windows requires developer mode or administrative privileges to create symbolic links. If collaborators use Windows, document this requirement prominently or provide a setup script that checks permissions.
2. **Dropbox follows symlinks.** Dropbox does not sync symlinks as symlinks; it follows them and syncs the target content. The `index.qmd` symlink at the project root becomes a regular file on other Dropbox-connected machines. Treat git as the canonical source, not Dropbox.
3. **Quarto skips symlinked directories.** This was the most time-consuming discovery in the entire project. Directory-level symlinks in `posts/` are invisible to the blog listing glob. Only file-level symlinks work.
4. **`git config core.symlinks`.** Some git configurations on Windows disable symlink support. Collaborators may need to run `git config core.symlinks true` to restore correct behavior after cloning.
5. **Freeze cache and Docker.** Quarto's `freeze` feature caches rendered output outside the Docker container. If a post executes R code during rendering (rather than loading pre-computed results), the freeze cache may produce inconsistent results between local and CI environments.

Daily Workflow

Command	Action
`make r`	Enter the Docker container
`make docker-build`	Rebuild the Docker image
`make docker-post-render`	Render the post inside Docker
`make check-renv`	Validate renv.lock against code

Command	Action
`quarto render index.qmd --to html`	Render locally (outside Docker)
`quarto preview`	Live preview with auto-reload

Uninstall / Rollback

To remove the zzcollab scaffolding from a post:

```
# 1. Remove Docker artifacts
docker rmi $(docker images -q <post-image>) 2>/dev/null
rm Dockerfile Makefile

# 2. Remove renv
rm -ri renv/ renv.lock .Rprofile

# 3. Remove symlinks (keeps the target files intact)
rm index.qmd data figures media

# 4. Move index.qmd out of analysis/report/ to the root
cp analysis/report/index.qmd ./index.qmd
```

The post's `.qmd` content is unchanged; only the reproducibility scaffolding is removed.

Figure 4: UCSD Geisel Library, a space for focused research

Academic research and technical writing share a common requirement: disciplined organization of complex materials.

What Did We Learn?

Lessons Learned

Conceptual Understanding:

- Reproducibility for blog posts is not merely about sharing code. It requires packaging the entire computational environment: R version, system libraries, package versions, and data provenance.
- The lead-in pattern (numbered directory containing a git repository and an archive folder) provides a clean separation between version-controlled and exploratory material.
- Quarto's glob-based post discovery is powerful but has a critical limitation: it does not follow symlinked directories. Understanding this constraint drove the entire architecture.
- Each blog post as its own repository provides independence at the cost of administrative overhead. For a lab producing dozens of posts, this trade-off is worth explicit consideration.

Technical Skills:

- Creating and verifying dual-layer relative symlinks that work across rendering contexts (project root vs. `analysis/report/`).
- Configuring Quarto listing globs with the double wildcard pattern (`posts/*/*/index.qmd`) to match the lead-in/project nesting.
- Setting up CI/CD pipelines that build Docker images, restore `renv` environments, and render Quarto reports in a single workflow.
- Using `git ls-files -s` to verify that git stores symlinks with the `120000` mode and the correct relative target path.

Gotchas and Pitfalls:

- Absolute symlinks break portability. Always use relative paths when creating symlinks, even if the absolute path seems more readable during initial setup.
- A full site render across 42+ posts is prohibitively slow. In practice, render only the modified post locally and rely on CI/CD for validation.
- The `renv.lock` must be committed to each post repository individually. A shared lockfile across posts defeats the purpose of independent dependency management.
- Dropbox silently converts symlinks to regular files on sync. Never rely on Dropbox as the primary synchronization mechanism for projects that use symlinks.

Limitations

- **Symlink fragility across operating systems.** macOS and Linux handle symlinks transparently; Windows requires special configuration. This limits cross-platform collaboration without additional setup documentation.

- **Repository proliferation.** With 42 posts, this workflow creates 42 GitHub repositories. Actions minutes, Dependabot alerts, and repository settings must be managed individually.
- **Quarto freeze interaction.** The freeze cache lives outside the Docker container and may produce inconsistent results if posts execute R code during rendering rather than loading pre-computed outputs.
- **Dropbox and symlinks are incompatible.** Dropbox follows symlinks and syncs target content, making git the only reliable synchronization mechanism.
- **Rendering cost.** A full site render requires entering each post's Docker container, restoring packages, and executing Quarto. For 42 posts, this is prohibitively slow without selective rendering.
- **Initial setup overhead.** The scaffold, symlinks, CI/CD workflow, and renv initialization require non-trivial effort for each new post, though this is amortized over the post's lifetime.

Opportunities for Improvement

1. **Automate post scaffolding.** A single CLI command (`zzc blog 43-newpost`) could create the lead-in directory, initialize the ZZCOLLAB project, create both symlink layers, and push an initial commit to GitHub.
2. **Shared Dockerfile caching.** Posts that use the same base profile (e.g., `ubuntu_x11_analysis`) could share a pre-built Docker image from a container registry, eliminating redundant builds.
3. **Selective site rendering.** A script that detects which posts have changed since the last render and re-renders only those would make full-site builds feasible.
4. **Symlink validation hook.** A pre-commit git hook could verify that all expected symlinks exist and point to valid targets, catching broken links before they reach CI/CD.
5. **Centralized dependency dashboard.** A monitoring tool that scans all 42 renv.lock files and flags posts using outdated or vulnerable package versions would simplify maintenance.
6. **Template compliance checker.** An automated script that validates YAML front matter fields, symlink integrity, and directory structure against the ZZCOLLAB blog post specification.

Wrapping Up

This post documented a workflow for treating each blog post as a standalone reproducible research project using ZZCOLLAB, Docker, renv, and Quarto. The architecture emerged from a concrete frustration: returning to a post that no longer rendered and having no reliable way to reconstruct the environment that originally produced it.

The workflow is not without costs. Each post generates its own GitHub repository, requires its own Docker image, and demands its own CI/CD pipeline. The initial setup overhead is real. But the payoff is substantial: any post can be cloned and reproduced by anyone, at any time, without guessing at package versions or system dependencies.

In conclusion, five points merit emphasis. First, the lead-in pattern (`NN-name/name/`) cleanly separates git repositories from archival material. Second, dual-layer relative symlinks reconcile Quarto's discovery mechanism with the rrtools/ZZCOLLAB directory convention. Third, Quarto's listing glob does not follow symlinked directories: this constraint drove the entire architecture. Fourth, the five pillars (Dockerfile, renv.lock, .Rprofile, source code, data) provide a complete reproducibility

contract for each post. Fifth, CI/CD catches breakage on push rather than months later when a reader reports a problem.

Appendix: rrtools as the Predecessor Pattern

ZZCOLLAB did not appear in a vacuum. The compendium pattern documented above is a refinement of the **rrtools** package introduced by Marwick, Boettiger, and Mullen (2018), which in turn operationalized the 'research compendium' construct introduced by Gentleman and Temple Lang (2007). This appendix preserves the rrtools framing for readers who encounter the older pattern in the literature or in existing project repositories, and clarifies what zzcollab adds.

The problem rrtools (and zzcollab) tried to solve

When sharing R code with a collaborator, several predictable failure modes arise:

- Different R versions on each machine
- Mismatched package versions
- Missing system dependencies (pandoc, LaTeX, image libraries)
- Missing supplementary files referenced by the analysis (bibliography files, LaTeX preambles, datasets, images)
- Collaborator-specific R startup configurations (`.Rprofile`, `.Renviron`)

A real-world scenario unfolds like this:

1. The author emails an R Markdown file to a colleague, Joe.
2. Joe attempts to run it with `R -e "source('peng1.Rmd')"`.
3. R is not installed on Joe's system.
4. After installing R, Joe gets an error: 'could not find function `render`'.
5. Joe installs the rmarkdown package.
6. Now pandoc is missing.
7. After installing pandoc, a required package is missing.
8. After installing the package, supplementary files are missing (bibliography, images).
9. The cycle continues until both parties give up or one party invests an afternoon in environment debugging.

The rrtools framework addressed this by defining a fixed research-compendium directory layout (`R/`, `data/`, `vignettes/`, `analysis/`) and pairing it with a Dockerfile that pinned the R version, the system libraries, and the package versions. The compendium directory was itself an R package, which gave it a `DESCRIPTION` file as a canonical manifest of dependencies.

What zzcollab adds

The zzcollab framework retains rrtools's directory layout and Dockerfile-plus-DESCRIPTION pattern, and extends it on three axes:

1. **Profile-based base images.** rrtools assumed each project would author its own Dockerfile from `rocker/r-ver` or similar. zzcollab provides named profiles (`minimal`, `analysis`, `modeling`, `publishing`, `shiny`) with pre-built base images, reducing the per-project Docker work to selecting a profile.
2. **Renv integration as a first-class layer.** The original rrtools approach used `DESCRIPTION` for dependency declaration; zzcollab additionally pins exact package versions via `renv.lock`. This is a stricter reproducibility contract.
3. **Make-driven workflow.** zzcollab projects ship a `Makefile` that exposes `make r` (enter the container), `make check-renv` (validate package state), `make test`, and `make docker-render-qmd` as the canonical entry points, rather than requiring the analyst to remember per-project Docker commands.

For readers maintaining an existing rrtools project, the migration to zzcollab is mechanical (copy the project's `R/`, `analysis/`, `data/` into a new zzcollab scaffold) and reversible. The two patterns coexist: a project on rrtools still satisfies the compendium-tier requirements of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist; zzcollab is the construct's recommended implementation of that tier in 2026, but rrtools remains a valid alternative.

See Also

Related Posts

- Blog Post Template (the reference implementation for this workflow)
- Unix Command-Line Workspace Setup for Data Science Development (terminal and shell setup for the development environment)
- Multi-Laptop macOS Bootstrap (configuration management for development tools)

Key Resources

- Marwick, B., Boettiger, C., & Mullen, L. (2018). Packaging Data Analytical Work Reproducibly Using R (and Friends). *The American Statistician*, 72(1), 80–88.

- ZZCOLLAB documentation: `~/prj/sfw/07-zzcollab/zzcollab/docs/`
- Quarto Blog Documentation
- The rocker Project
- renv documentation

Reproducibility

Environment: R 4.5.1, Quarto 1.6+, Docker (rocker/tidyverse:4.5.1)

Project template: `posts/39-templatepost/`

Site configuration: `_quarto.yml` at repository root

Migration script: `migrate_posts.sh` at repository root (converts flat post directories to lead-in/project structure)

```
sessionInfo()
```

```
R version 4.6.1 (2026-06-24)
Platform: aarch64-apple-darwin25.4.0
Running under: macOS Tahoe 26.6.2

Matrix products: default
BLAS:   /opt/homebrew/Cellar/openblas/0.3.34/lib/libopenblasp-r0.3.34.dylib
LAPACK: /opt/homebrew/Cellar/r/4.6.1/lib/R/lib/libRlapack.dylib;  LAPACK version 3.12.1

locale:
[1] en_US.UTF-8/en_US.UTF-8/en_US.UTF-8/C/en_US.UTF-8/en_US.UTF-8

time zone: America/Los_Angeles
tzcode source: internal

attached base packages:
[1] stats     graphics  grDevices utils     datasets  methods   base

loaded via a namespace (and not attached):
 [1] compiler_4.6.1  fastmap_1.2.0   cli_3.6.6       tools_4.6.1
 [5] htmltools_0.5.9 parallel_4.6.1  otel_0.2.0      yaml_2.3.12
 [9] rmarkdown_2.31  knitr_1.51      jsonlite_2.0.0  xfun_0.58
[13] digest_0.6.39   rlang_1.3.0     evaluate_1.0.5
```

Let's Connect

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

I would enjoy hearing from you if:

- You spot an error or a better approach to any of the code in this post.
- You have suggestions for topics you would like to see covered.
- You want to discuss R programming, data science, or reproducible research.
- You have questions about anything in this tutorial.
- You just want to say hello and connect.

Related posts in this cluster

This post is part of the *ZZCOLLAB Reproducible Compendia* series. Recommended reading order:

1. **Post 01: Reproducible Blog Posts with ZZCOLLAB** (this post)
2. Post 02: Constructing a reproducible blog post using zzcollab tools

3. Post 03: From Markdown to Blog Post: A ZZCOLLAB workflow
4. Post 04: Sharing R Code via Docker: R Markdown Reports
5. Post 05: A 55-Item Initiation Checklist for zzcollab Data Analyses
6. Post 06: Seven Required Elements for a zzc Manuscript report.Rmd
7. Post 07: A tiered CI strategy for zzcollab research compendia
8. Post 08: GitHub Actions workflows for zzcollab research compendia