Sharing R Code via Docker: R Markdown Reports and Shiny Applications

Ronald G. Thomas

2026-05-17

2026-05-17 16:55 PDT

Figure 1: A sealed glass terrarium containing a small self-contained ecosystem: a Docker container as a self-contained, portable environment that runs the same anywhere.

Just as shipping containers standardized global freight, Docker containers standardize software delivery: for static analyses and interactive applications alike.

Introduction

I did not really know how fragile sharing R code could be until I sent a single `.Rmd` file to my colleague (let us call him Joe) by email. Joe then spent an entire afternoon debugging missing packages, incompatible R versions, and absent files. The code ran perfectly on my machine. Joe's Linux Mint workstation disagreed on every count.

The Shiny case is worse. A static R Markdown report requires R, packages, and possibly LaTeX. A Shiny application requires all of that plus a running web server, reactive dependencies, and often

JavaScript libraries or database connections. Each additional dependency is another opportunity for something to break on a collaborator's machine, and the failure mode is a blank browser tab with no useful error message.

We cover both cases. The core Docker machinery (Dockerfile authoring, base image selection, dependency installation) is explained once and applies to both. The post then presents two parallel case studies:

- **Case A**: Dockerizing an R Markdown report for static PDF output.
- **Case B**: Dockerizing a Shiny application for interactive browser use.

More formally, we document the Container layer of the Workflow Construct described in A Workflow Construct for the Modern Data Scientist. The Container layer is replaceable across runtimes (Docker on the laptop, Apptainer on HPC, Podman as a daemonless alternative); its core machinery is the same regardless of what the container ultimately serves.

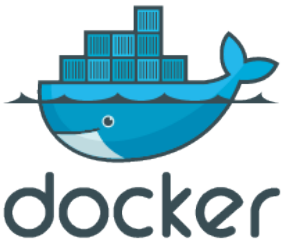
Motivations

- I had sent an R Markdown analysis to a colleague and watched him spend hours debugging environment issues that had nothing to do with the analysis itself.
- I had also sent a Shiny application as a zip file and observed the same pattern: hours resolving package version conflicts and missing system libraries before the app would launch.
- Every error my colleagues encountered was entirely preventable with proper packaging.
- I wanted a reproducible method for sharing R analyses that would work regardless of the recipient's operating system, R version, or installed packages.
- The experience convinced me that 'works on my machine' is not a valid standard for collaborative data science, and that Docker addresses both the static and interactive output cases.

Objectives

1. Explain the core Docker concepts (image, container, Dockerfile, layer caching, volume mounts, port mapping) that apply to all R containerization workflows.
2. Demonstrate the failure modes of naive sharing for both R Markdown and Shiny by walking through the sequence of errors a collaborator encounters.
3. Write complete Dockerfiles for each case: one for an R Markdown report, one for a Shiny application.
4. Document the complete workflow from authoring to collaboration, including image sharing options and Shiny-specific concerns (process supervision, port binding, host parameter).

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



The analysis begins on one machine, but it needs to run on many.

Prerequisites and Setup

Before proceeding, the following are required:

- **Docker Desktop** installed on the workstation (macOS, Windows, or Linux)
- **R 4.0+** with `rmarkdown` and/or `shiny` installed locally (for authoring)
- **A Docker Hub account** (free) for pushing and sharing images
- Basic familiarity with the terminal and R Markdown or Shiny application structure

What is Docker for R Users?

Docker is a tool that packages an application and its entire computing environment (operating system libraries, programming language, installed packages, configuration files, data) into a single portable unit called an image. When someone runs that image, they get a container that behaves identically to the environment in which the analysis was developed.

Think of it like shipping a complete laboratory instead of just a lab notebook. The naive approach to sharing code is like sending someone a lab notebook and hoping they have the same equipment, reagents, and room temperature. Docker is like shipping the entire lab bench, pre-loaded with everything needed to replicate the experiment.

For R users specifically, Docker solves the problem of environment divergence. The R version, installed packages, LaTeX distribution, auxiliary files, and data are all captured in the image. The recipient does not need to install anything except Docker itself.

Core Concepts

Image: A snapshot of a complete computing environment. Images are built from a Dockerfile and stored as a series of read-only layers.

Container: A running instance of an image. Containers are ephemeral by default; data written inside a container is lost when it stops unless a volume mount is used.

Dockerfile: A text file that defines how to build an image. Each instruction (`FROM`, `RUN`, `COPY`, `CMD`) creates a new layer. Docker caches layers, so unchanged instructions do not re-execute on subsequent builds.

Volume mount: A link between a directory on the host machine and a path inside the container. Volume mounts allow the container to read host files and write output that persists after the container stops.

Port mapping: For network services (Shiny applications), the container's internal port must be mapped to a host port with the `-p` flag, or the host machine cannot connect to the running service.

The Rocker Project

The Rocker Project maintains a family of Docker images for R. The relevant images for this post are:

Image	Contents	Use case
`rocker/r-ver`	Base R only	Lightweight scripts
`rocker/verse`	R + tidyverse + LaTeX	R Markdown PDF
`rocker/shiny`	R + Shiny Server	Shiny applications

All images accept a version tag (e.g., `rocker/verse:4`) to pin the R version.

Case A: Dockerizing an R Markdown Report

The Analysis

Assume you have a simple R Markdown file called `peng.Rmd` that analyses the Palmer Penguins dataset. The file uses `pacman` for package management, `tidyverse` for data manipulation and plotting, and `knitr` for output formatting. It also references a custom LaTeX preamble file (`preamble.tex`) and a logo image (`sudoku.png`) for the PDF header.

```
---
title: "Penguins analysis"
author: "R.G. Thomas"
date: "`r format(Sys.time(), '%B %d, %Y')`"
fontsize: 11pt
geometry: "left=3cm,right=5cm,top=2cm,bottom=2cm"
output:
  pdf_document:
    keep_tex: true
    includes:
      in_header: "preamble.tex"
---
```

```{r include=F, echo=F}
library(pacman)
p_load(palmerpenguins, tidyverse, knitr)

opts_chunk$set(
  warning = FALSE,
  message = FALSE,
  echo = FALSE,
  results = "asis",
  dev = "pdf"
)
```

Introduction

We can work with the dataset `penguins`
included in the package `palmerpenguins`.

```{r }
library(palmerpenguins)
```

One naive approach is to split the dataset
and do three separate analyses:

```{r }
df1 <- split(penguins, penguins$species)

foo <- function(df, z) {
  df |>
    ggplot(
      aes(
        x = bill_length_mm,
        y = flipper_length_mm
      )
    ) +
    geom_point(
      aes(color = island), alpha = .5
    ) +
    geom_smooth() +
    scale_color_manual(
      values = c("purple", "green", "red")
    ) +
    theme_bw() +
    labs(
      title = paste(
        z, " Penguin Anatomy Comparison"
```

```
      ),
      x = "Flipper length",
      y = "Bill length",
      color = "Island"
    )
  plotfile_name <- paste0(z, ".pdf")
  ggsave(plotfile_name)
  cat(
    paste0(
      "\\includegraphics[height=3cm]{",
      plotfile_name, "}"
    ),
    "\n"
  )
  cat("\\vspace{1cm}", "\n")
}

bar <- df1 |> map2(names(df1), foo)
```

The file runs cleanly on the author's machine and produces a PDF report with three species-specific scatter plots comparing bill length and flipper length.

The Naive Approach and Its Errors

The simplest approach is to email `peng.Rmd` to Joe and ask him to render it. The following sequence of errors occurred in the order presented.

Error 1: R Not Found. Linux cannot find `R`. Joe installs `r-base-core` via `apt`.

Error 2: Function render Not Found. R loads but `rmarkdown` is missing. The package installation fails due to missing system libraries (`libssl-dev`, `libcurl4-openssl-dev`, `libxml2-dev`, and others).

Error 3: Pandoc Version Too Old. The system pandoc is below 1.12.3. Joe installs a newer version.

Error 4: Package pacman Not Found. After resolving pandoc, `pacman` is missing.

Error 5: Missing preamble.tex. The author forgot to send the LaTeX preamble file. The path convention also differs between macOS and Linux.

Error 6: pdflatex Not Found. LaTeX is absent. Joe installs `tinytex`.

Error 7: Missing Logo File. `pandoc error: file sudoku.png not found`. The author forgot to send the logo image referenced in the preamble.

Error 8: Missing Bibliography File. Another file the author forgot to include.

After eight errors and several hours, Joe renders the report. Every error was environmental.

The Dockerfile

The following Dockerfile starts from `rocker/verse:4` (which includes R, tidyverse, and LaTeX), installs additional R packages, updates the LaTeX distribution, creates a non-root user, and copies all necessary files into the image.

```
FROM rocker/verse:4
RUN apt update
RUN apt install vim -y
RUN R -e "install.packages('pacman')"
RUN R -e "install.packages('palmerpenguins')"
RUN R -e "install.packages('tidyverse')"
RUN R -e "install.packages('knitr')"
RUN R -e "install.packages('rmarkdown')"
RUN tlmgr init-usertree
RUN tlmgr update --self --all
RUN tlmgr install fancyhdr adjustbox \
  geometry titling

RUN addgroup --system joe && \
  adduser --system --ingroup joe joe
RUN chmod -R 0777 \
  '/usr/local/lib/R/site-library'
RUN chown joe:joe -R /home/joe
USER joe
WORKDIR /home/joe
RUN mkdir -p /home/joe/shr
RUN mkdir -p /home/joe/output
COPY /preamble.tex /home/joe/shr
COPY sudoku.png /home/joe/shr
COPY peng.Rmd /home/joe/shr
CMD ["/bin/bash"]
```

Key decisions:

- **`FROM rocker/verse:4`** provides R 4.x with tidyverse and LaTeX pre-installed.
- **Each `RUN` installs a specific package**, making layer caching effective. If only the Rmd file changes, only the `COPY` layer rebuilds.
- **`tlmgr install`** adds the exact LaTeX packages referenced by `preamble.tex`.
- **A non-root user (`joe`)** runs the analysis as a security best practice.
- **`COPY` commands** place the preamble, logo, and Rmd file inside the image, so nothing is missing.

Building and Sharing the Image

Build with a tag and platform flag for cross-architecture compatibility:

```
docker build -t rgt47/penguin_review \
  --platform=linux/amd64 .
```

Option 1: Docker Hub (recommended for remote collaboration)

```
docker push rgt47/penguin_review
```

Joe pulls the image on his machine:

```
docker pull rgt47/penguin_review
```

Option 2: File transfer (useful for air-gapped networks)

```
docker save rgt47/penguin_review \
  | gzip > penguin_review.tgz
```

Joe loads the image from the archive:

```
docker load -i penguin_review.tgz
```

Running the Container

Joe runs the container with a volume mount that connects a local output directory to the container's output directory. This is how rendered files are extracted from the container.

```
droot="$PWD/output"
docker run -it --rm \
  --platform linux/x86_64 \
  -v "$droot":/home/joe/output \
  rgt47/penguin_review
```

Inside the container, Joe renders the report:

```
cd output
R -e "library(rmarkdown); \
  render('../shr/peng.Rmd')"
```

The rendered PDF appears in the local `output/` directory. No errors. No missing packages. No missing files.

The LaTeX Preamble

For reference, the `preamble.tex` file that caused Error 5 in the naive approach:

```
\usepackage[export]{adjustbox}
\usepackage{fancyhdr}
\usepackage{titling}

\pagestyle{fancy}

\pretitle{
\begin{flushright}
\includegraphics[width=3cm,valign=c]{
  sudoku.png}\\
\end{flushright}
\begin{flushleft} \LARGE }
\posttitle{
  \par\end{flushleft}\vskip 0.5em}
\predate{
  \begin{flushleft}\large}
\postdate{
  \par\end{flushleft}}
\preauthor{
  \begin{flushleft}\large}
\postauthor{
  \par\end{flushleft}}
\fancyfoot[L]{\currfilename}
\fancyfoot[R]{
  \includegraphics[width=.8cm]{sudoku.png}}
\fancyhead[L]{\today}
```

In the Docker image, this file is already in place at `/home/joe/shr/preamble.tex`. Joe never needs to know it exists.

Figure 2: A stack of identical sealed containers, each with a handwritten label, ready to be handed to a different person.

Packaging everything together so no piece is left behind.

Case B: Dockerizing a Shiny Application

The Shiny case introduces concerns absent from the R Markdown case: the container must run a persistent web server, bind to a network port accessible from the host, and handle multiple concurrent connections. The Dockerfile structure is similar, but the base image, the `CMD`, and the `docker run` flags differ.

The Application

Assume you have a single-file Shiny app (`app.R`) that displays an interactive scatter plot of the Palmer Penguins dataset.

```
library(shiny)
library(ggplot2)
library(DT)
library(palmerpenguins)

ui <- fluidPage(
  titlePanel("Palmer Penguins Explorer"),
  sidebarLayout(
```

```
    sidebarPanel(
      selectInput(
        "species", "Species:",
        choices = c(
          "All",
          unique(
            as.character(penguins$species)
          )
        )
      ),
      selectInput(
        "x_var", "X Variable:",
        choices = c(
          "bill_length_mm",
          "bill_depth_mm",
          "flipper_length_mm",
          "body_mass_g"
        )
      ),
      selectInput(
        "y_var", "Y Variable:",
        choices = c(
          "flipper_length_mm",
          "bill_length_mm",
          "bill_depth_mm",
          "body_mass_g"
        )
      )
    ),
    mainPanel(
      plotOutput("scatter"),
      DTOutput("table")
    )
  )
)

server <- function(input, output, session) {
  filtered_data <- reactive({
    if (input$species == "All") {
      penguins
    } else {
      penguins[
        penguins$species == input$species,
      ]
    }
  })
```

```
  output$scatter <- renderPlot({
    ggplot(
      filtered_data(),
      aes(
        x = .data[[input$x_var]],
        y = .data[[input$y_var]],
        colour = species
      )
    ) +
      geom_point(alpha = 0.7, size = 3) +
      theme_minimal() +
      labs(
        x = input$x_var,
        y = input$y_var,
        colour = "Species"
      )
  })

  output$table <- renderDT({
    filtered_data()
  })
}

shinyApp(ui, server)
```

The app runs flawlessly on the development machine.

The Naive Approach and Its Errors

You zip the directory and email it to Joe. The following sequence of errors occurred.

Error 1: Shiny Not Installed. `there is no package called 'shiny'`. Installing it triggers compilation of `httpuv` and its system dependencies.

Error 2: System Libraries Missing. The `httpuv` package fails to compile because `libssl-dev` and `libuv1-dev` are absent.

Error 3: Package DT Not Found. After installing Shiny, the app fails because `DT` is missing.

Error 4: palmerpenguins Not Found. The data package is missing.

Error 5: Port Already in Use. Another service occupies port 3838. Joe must identify and stop the conflicting process.

Error 6: Version Incompatibility. Joe's R version is 4.1; the app uses `.data[[]]` syntax from `rlang` 1.0+, which requires a newer `ggplot2` than what installed against R 4.1.

After six errors and several hours, Joe has a running application. Every error was environmental.

The Dockerfile

The following Dockerfile starts from `rocker/shiny:4`, which includes R and Shiny Server pre-installed.

```
FROM rocker/shiny:4

RUN apt-get update && apt-get install -y \
  libssl-dev \
  libcurl4-openssl-dev \
  libuv1-dev \
  && rm -rf /var/lib/apt/lists/*

RUN R -e "install.packages(c( \
  'ggplot2', \
  'DT', \
  'palmerpenguins', \
  'rlang' \
  ), repos = 'https://cran.r-project.org')"

RUN rm -rf /srv/shiny-server/*

COPY app.R /srv/shiny-server/app.R

EXPOSE 3838

CMD ["/usr/bin/shiny-server"]
```

Key decisions:

- **`FROM rocker/shiny:4`** provides R 4.x with Shiny Server pre-installed, eliminating manual web server configuration.
- **System libraries** are installed before R packages to ensure compilation of binary dependencies (`httpuv`, `openssl`).
- **`rm -rf /srv/shiny-server/*`** removes default sample apps, leaving only the target application.
- **`EXPOSE 3838`** documents the port that Shiny Server listens on (it does not publish the port; that requires `-p` at runtime).
- **`CMD`** starts Shiny Server when the container launches, so Joe does not need to type any commands inside the container.

Alternative: Single-File Approach

For simpler applications, it is possible to bypass Shiny Server and run the app directly with `Rscript`. This uses the smaller `rocker/r-ver` base image.

```
FROM rocker/r-ver:4

RUN R -e "install.packages(c( \
  'shiny', \
  'ggplot2', \
  'DT', \
  'palmerpenguins' \
  ), repos = 'https://cran.r-project.org')"

COPY app.R /app/app.R

EXPOSE 3838

CMD ["Rscript", "-e", \
  "shiny::runApp('/app/app.R', \
  host='0.0.0.0', port=3838)"]
```

The trade-off is the loss of Shiny Server features (logging, process management, multi-app serving), but for single-app containers it is often sufficient and produces a smaller image.

Building and Sharing the Image

Build with a tag:

```
docker build -t rgt47/penguins-shiny \
  --platform=linux/amd64 .
```

Option 1: Docker Hub

```
docker push rgt47/penguins-shiny
```

Joe pulls the image:

```
docker pull rgt47/penguins-shiny
```

Option 2: File transfer

```
docker save rgt47/penguins-shiny \
  | gzip > penguins-shiny.tgz
```

Joe loads the image:

```
docker load -i penguins-shiny.tgz
```

Running the Container

Joe runs the container with port mapping:

```
docker run -d --rm \
  -p 3838:3838 \
  --name penguins-app \
  rgt47/penguins-shiny
```

He opens a browser to `http://localhost:3838` and the application is running. No errors. No missing packages. No port conflicts.

To stop the application:

```
docker stop penguins-app
```

Shiny-Specific: Process Supervision and

Multi-User Deployment

The `rocker/shiny` image runs the open-source edition of Shiny Server, which does not efficiently handle many concurrent users. For production deployments with authentication and per-user container isolation, consider ShinyProxy. ShinyProxy acts as a reverse proxy and launches a fresh container for each authenticated user, providing isolation without requiring Shiny Server Pro.

For applications requiring persistent user uploads or database writes, mount a host directory to the container:

```
docker run -d --rm \
  -p 3838:3838 \
  -v "$PWD/uploads":/srv/shiny-server/uploads \
  --name penguins-app \
  rgt47/penguins-shiny
```

Figure 3: A quiet desk with a closed laptop and a cup of coffee, representing the conclusion of a completed workflow.

When the analysis runs on the first try, on any machine, the work is done.

Things to Watch Out For

1. **Volume mount paths must be absolute.** The `-v` flag requires an absolute path on the host side. Use `"$PWD/output"` or provide the full path explicitly.
2. **Platform flag matters for Apple Silicon.** When building on an M1/M2/M3 Mac, the default architecture is `arm64`. Use `--platform=linux/amd64` to ensure the image runs on Intel-based Linux machines.
3. **The `host` parameter must be `'0.0.0.0'` for Shiny.** By default, `shiny::runApp()` binds to `127.0.0.1`, which is only accessible inside the container. This is the most common reason a Dockerized Shiny app appears to start but shows a blank page.
4. **`EXPOSE` does not publish the port.** The `EXPOSE` instruction in a Dockerfile is documentation only. The `-p` flag at runtime is what makes the port reachable from the host.
5. **Image size can be large.** The `rocker/verse` base image is approximately 2 GB; `rocker/shiny` is approximately 1.5 GB. Adding packages increases this further. Use `rocker/r-ver` for a smaller base when the full tidyverse or LaTeX is not required.
6. **R package versions are not pinned.** The Dockerfiles above install the latest version of each package at build time. For strict reproducibility, use `renv` to pin exact package versions and restore from `renv.lock` inside the Dockerfile.
7. **Non-root user permissions.** If the container writes output files as a non-root user, the host directory may need appropriate permissions. The `chmod -R 0777` in the Case A Dockerfile is permissive; in production, use more restrictive permissions.

Daily Workflow

Task	Command
Build image (R Markdown)	`docker build -t rgt47/penguin_review --platform=linux/amd64 .`
Build image (Shiny)	`docker build -t rgt47/penguins-shiny --platform=linux/amd64 .`
Push to Docker Hub	`docker push rgt47/<image-name>`
Save image to file	`docker save rgt47/<image-name> | gzip > image.tgz`
Load image from file	`docker load -i image.tgz`
Run R Markdown container	`docker run -it --rm -v "$PWD/output":/home/joe/output rgt47/penguin_review`
Run Shiny container	`docker run -d --rm -p 3838:3838 --name penguins-app rgt47/penguins-shiny`
View Shiny logs	`docker logs penguins-app`
Stop Shiny container	`docker stop penguins-app`
List running containers	`docker ps`

Task	Command
Remove stopped containers	`docker container prune`

Uninstall / Rollback

To remove a specific image:

```
docker rmi rgt47/penguin_review
docker rmi rgt47/penguins-shiny
```

To remove all stopped containers and dangling images (reclaim disk space):

```
docker system prune
```

To remove all unused images (not just dangling):

```
docker system prune -a
```

Docker Desktop can be uninstalled via the application's settings menu on macOS and Windows. On Linux, remove the `docker-ce` package via the system package manager. Uninstalling Docker does not affect files outside the Docker storage directory (typically `/var/lib/docker`).

Lessons Learned

Conceptual Understanding

- The naive approach to sharing R code fails because it assumes the recipient's environment matches the author's environment. In practice, it never does.
- Docker eliminates environment divergence by packaging the complete computing context alongside the analysis code, whether that code produces a static file or a live web application.
- The `rocker` project provides pre-built Docker images for R that include common dependencies, significantly reducing Dockerfile complexity.
- Volume mounts are the mechanism for extracting output from containers; port mapping is the mechanism for connecting a container's network service to the host machine's browser.

Technical Skills

- Building a Dockerfile for R requires installing both R packages (`install.packages()`) and system libraries (`apt install`) for packages with compiled dependencies.
- The `rocker/verse` base image covers most R Markdown rendering requirements; `rocker/shiny` covers Shiny Server deployments; `rocker/r-ver` is the smallest base for scripts that do not need the full tidyverse or LaTeX.

- `docker save` and `docker load` provide an alternative to Docker Hub for sharing images on restricted networks.
- `docker run -d` starts a container in detached mode, which is appropriate for long-running server applications like Shiny.

Gotchas and Pitfalls

- Forgetting to include auxiliary files (preamble, images, data) in the Dockerfile `COPY` commands is the most common source of R Markdown build failures.
- Forgetting to set `host='0.0.0.0'` is the most common reason a Dockerized Shiny app appears to start but shows a blank page.
- The `--platform` flag is essential for cross-architecture compatibility between Apple Silicon and Intel machines.
- Running `docker build` without `--no-cache` may use stale cached layers if upstream packages have been updated; add `--no-cache` when a fully fresh build is required.

Limitations

- Docker Desktop is required on the recipient's machine, which may not be permitted in all organizational environments.
- Base images are large (1.5 to 2 GB before adding packages), making image sharing impractical on low-bandwidth connections.
- R package versions are not pinned in these Dockerfiles; future builds may install different versions, breaking reproducibility over time. Integrating `renv` addresses this but adds complexity.
- The Case B Dockerfile does not address authentication or access control; the Shiny app is accessible to anyone who can reach the mapped port.
- Shiny Server (open-source edition) does not efficiently handle many concurrent users. For production multi-user deployments, ShinyProxy or Shiny Server Pro is required.
- Neither Dockerfile includes a health check; orchestration tools (Kubernetes, Docker Compose) cannot verify the container is actually responding without one.
- Docker ensures environment consistency, not analytical validity. Code correctness is a separate concern.

Opportunities for Improvement

1. Integrate `renv` into both Dockerfiles to pin exact package versions and ensure long-term reproducibility.
2. Add a `Makefile` to the R Markdown image that automates the rendering step, allowing Joe to run `make render` instead of typing the full R command.
3. Use multi-stage Docker builds to separate the build environment (with compilation tools) from the runtime environment, reducing the final image size.
4. Add a `docker-compose.yml` for analyses that require additional services (e.g., a PostgreSQL database alongside R or Shiny).

5. Implement ShinyProxy for multi-user Shiny deployments with authentication and per-user container isolation.
6. Add a `HEALTHCHECK` instruction to the Shiny Dockerfile (`HEALTHCHECK CMD curl -f http://localhost:3838 || exit 1`) for container orchestration.
7. Explore GitHub Actions to automatically build and push Docker images whenever the analysis or application code is updated.

Wrapping Up

The contrast between the naive approach and the Docker approach illustrates a fundamental principle of reproducible research: the computing environment is part of the analysis. Sending an Rmd file or a Shiny application without its environment is like publishing a paper without its methodology section.

The naive approach produced eight separate errors for the R Markdown case and six for the Shiny case, each requiring Joe to diagnose and resolve a dependency issue. The Docker approach produced zero errors in either case. Joe pulled the image, ran a single command, and either opened a rendered PDF in his `output/` directory or a working application in his browser.

The Shiny case makes the stakes clearer: a missing dependency in a static analysis produces a clear error message. A missing dependency in a Shiny app produces a blank browser tab. Docker addresses both failure modes by fixing the entire stack at build time.

In conclusion, five points merit emphasis. First, the naive approach produced eight distinct environment errors for R Markdown and six for Shiny before the output could be obtained. Second, Docker eliminates all such errors by packaging the complete environment into a single image. Third, Case A uses `rocker/verse` (R + tidyverse + LaTeX) with volume mounts for PDF output. Fourth, Case B uses `rocker/shiny` (R + Shiny Server) with port mapping for browser access. Fifth, the `host='0.0.0.0'` parameter and the `-p` flag are the two settings most commonly missed when Dockerizing Shiny applications.

See Also

Related posts:

- "Configure the Command Line for Data Science Development" (terminal and Docker setup)
- "Setting Up R, Vimtex, and UltiSnips in Vim" (the Vim editing environment used to author the original analysis)

Key resources:

- Running your R script in Docker (Statworx)
- Rocker Project: Docker containers for R
- rocker/shiny Docker image
- Shiny Server documentation
- ShinyProxy: open-source Shiny deployment
- Docker documentation
- renv: Project Environments for R

Reproducibility

This workflow was developed on macOS with Docker Desktop 4.x, R 4.4, and Shiny 1.8.

Case A files:

File	Purpose
`peng.Rmd`	R Markdown analysis file
`Dockerfile`	Environment definition
`preamble.tex`	LaTeX header customization
`sudoku.png`	Logo image for PDF header

Case B files:

File	Purpose
`app.R`	Shiny application code
`Dockerfile`	Environment definition

To reproduce Case A:

```
docker build -t rgt47/penguin_review \
  --platform=linux/amd64 .

mkdir -p output
docker run -it --rm \
  --platform linux/x86_64 \
  -v "$PWD/output":/home/joe/output \
  rgt47/penguin_review

# Inside the container:
# cd output
# R -e "library(rmarkdown); render('../shr/peng.Rmd')"
```

To reproduce Case B:

```
docker build -t rgt47/penguins-shiny \
  --platform=linux/amd64 .

docker run -d --rm \
  -p 3838:3838 \
  --name penguins-app \
  rgt47/penguins-shiny

# Open browser to http://localhost:3838

docker stop penguins-app
```

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.

Rendered on 2026-05-17 at 17:08 PDT. Source: ~/prj/rgtlab/posts/zc-share-rmd-via-docker/index.qmd

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
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** (this post)
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