Best Practices for Reproducible Package Management in R

Paper AB09

Cole Arendt, RStudio, Inc., Raleigh, North Carolina
Sean Lopp, RStudio, Inc., Denver, Colorado

2019-02-02

ABSTRACT

R packages are the staple of an R programmer's effectiveness. Since R is an open source language with a robust package (extension) framework, "Base R" (R with no packages) can be relatively featureless. However, add the CRAN package network of 12,000 packages (and growing), and the R programmer instantly becomes enormously efficient and powerful. It raises an immediate question, though: How do we use those open source packages in a reliable and reproducible way? We will discuss best practices for managing packages as an individual developer and as an enterprise. If we can do this successfully, we will establish a firm foundation on which to build reliable results.

INTRODUCTION

The R package ecosystem extends the R programming language in just about any direction imaginable. It enables and empowers the open source community surrounding R to collaborate and contribute to collective improvements that all users can share. These packages simplify (among other things) the process of fitting models, analyzing data, logging, visualizing, adding emojis, creating user interfaces, building REST APIs, and munging spreadsheets.

In fact, the R package ecosystem might be *the* most valuable part of the R programming language. Packages are shared most often via the "Comprehensive R Archive Network" (CRAN), but also via GitHub, BioConductor and RForge.

With over 12,000 such packages, the R programmer's toolbelt is enormously varied and she can be incredibly effective as a result. However, this vibrant ecosystem does not come without its risks, so the *way* in which these packages are managed by a researcher is important for the reproducibility of her work.

WHY R PACKAGES

The first question a security minded or IT staff person might ask, though, is *why*? Why expose yourself to the risk of using an open source package that might break unexpectedly, have a security vulnerability, or do something subtly incorrect? Would not R *by itself* be much more reliable, reproducible, and secure?

These are valid and immensely valuable questions. We would answer on several fronts:

- The R package ecosystem contains decades or centuries of collective software engineering, research, and algorithm optimization. To omit this work from your toolbelt without inquiry is to epitomize re-inventing the wheel
- Not all packages are created equal. Some are fringe experiments or pet projects and are not fitting for reproducible research. Likewise, others are serious software engineering funded by reputable organizations or maintained by renown programmers.

- R packages have *standards.* `R CMD CHECK`, as well as many of `CRAN`'s checks help validate that an R package does not have certain types of illicit, insecure, or dangerous behavior.
- R packages thrive in a community of open source contributors who are vetting, testing, validating, and improving algorithms and processes. Such a community is far more effective and thorough than any CI pipeline.

Suffice it to say that R packages are desirable to the R programmer, and we will assume this for the rest of our work. However, the concerns above must still be addressed, and we will propose best practices for addressing these concerns.

AN ANALOGY

Let us imagine a chef who is working for a five-star restaurant. This chef creates remarkable meals in his kitchen and becomes renown for his work. We might guess that this chef cares *very much* about where he gets his ingredients from, as the ingredients are integral to his dishes. Further, this chef will care very much *more* about his ingredients and spend much more time examining and understanding them than the average person.

In our analogy, the chef is a researcher and his dishes are his research. His ingredients are the R packages that support and facilitate his work. He has a distributor that he gets his ingredients from, and then he has "best practices" for his management of the ingredients when they arrive in his kitchen.

We will make use of this analogy throughout our paper.

THE SOURCE

The first task that our chef must explore is how to *get* ingredients. Similarly, our discussion of R packages begins with *acquiring them*. There are a handful of canonical *sources* for R packages, with varying levels of reproducibility:

- CRAN
- Bioconductor
- GitHub / GitLab
- In-house / self-owned
- RForge

Certain decisions might be made to throw out entire sources. For instance, GitHub is mostly under control of individual developers and so a package could *completely disappear* on any given day. However, in practice, even GitHub packages have standards.

Further, other sources have more formal standards, like CRAN or BioConductor. These sources run tests and sanity checks, and they also preserve older source versions of packages in an "archive" so that any given package version can be rebuilt at any time.

As a result of these and other factors, we recommend that practitioners have a thoughtful look at what sources are acceptable for retrieving R packages and what standards should be implemented for their usage. One recommendation that accompanies this one is to **set up your own package repository**.

Setting up your own package repository from the selected sources above allows you to set up an "ingredient inspector" or "middle-man" who can audit, preserve, and validate your chosen packages while removing some level of dependence on the package sources themselves. Further, it is immensely helpful in enterprise environments that may be offline or have limited access to the internet.

If you are interested in setting up your own repository, popular options are Artifactory, miniCRAN, and RStudio Package Manager. Each varies in its approach at some level, so a survey of the capabilities of each is helpful to the decision making process. A private repository can have the added benefit of facilitating private package development for one's own enterprise or organization.

A final recommendation on package sources is to consider a source or inspector that can track or freeze the state of the repository. This approach is the approach taken by MRAN (Microsoft's copy of CRAN). Microsoft takes a copy of CRAN every day, so that users can point to a given date with the `checkpoint` package (i.e. "2018-01-01") and see CRAN as it existed on that day. This type of "frozen" repository is helpful for researchers who need to reproduce work done at a previous time or ensure that package installation always returns the same packages. RStudio Package Manager adopts a similar approach, except *every* event to the repository is tracked as an immutable transaction, allowing full reproducibility of the repository to a moment in time.

THE INGREDIENTS

Once the sources have been decided and an architecture for delivering such ingredients has been designed, the question moves to *which* ingredients. Though a chef may have access to many sources, she must still decide which ingredients will make the most delicious meal.

For R packages, these decisions traditionally fall along a handful of axes:

- The license the package source is under
- The popularity of the package
- Is the package under active development
- Does the package meet source requirements (i.e. CRAN)
- Has it passed necessary security checks

Every researcher and every organization will differ in how they decide along these points. However, it is worth noting some common tools for this type of decision making.

For deciding on licenses, there are websites like rdrr.io and cran.rstudio.com that allow the exploration of package sources. Further, tools like RStudio Package Manager add helpful search and filtering features along licenses.

For popularity, there is much soft / informal exploration of social sentiment and GitHub traffic, as well as more explicit data captured by cranlogs on worldwide package use, and private usage information captured by Package Manager.

The other decision points are a bit more sensitive to each organization, but they are greatly simplified by having an internal repository where these types of package exploration, validation, and auditing can take place.

THE RECIPE

Once sources have been vetted and ingredients have been decided, our chef is free to work her magic. Most chefs that care about reproducibility will keep a list of the ingredients that went into their dish, and the same should be true for researchers.

Lest we arrive at a tasty meal that we cannot reproduce, we ought to take proactive efforts during the development process to ensure reproducibility. As was the case above, there are many factors which might be included into the researcher's approach. While above we spoke about package sources and the "middle-man" who handles the packages before they arrive in the kitchen, now we will discuss the chef's actions. You might imagine she writes down all the ingredients up front, or incrementally as she goes, or perhaps only when her dish is complete. The same is true for a researcher.

As was discussed in a blog post on this topic, the researcher's perspective of their project may very much shape her choices at the outset. For instance, she may opt to use `sessionInfo()` or `devtools::session_info()` as a lightweight way to record package versions intermittently as she goes. To this approach, I would also suggest `packrat::.snapshotImpl("./", snapshot.sources = FALSE)` be included as a lightweight way to record exact package versions.

In the case of a frozen repository like MRAN or Package Manager, the researcher may opt at the outset to install packages from a snapshot in time by using the `checkpoint` package or by specifying a Package Manager transaction:

```
options("repos" = "https://mypackagemanager.com/cran/773")
```

Further, the researcher might decide to opt to use `packrat` or a similar package that takes a proactive approach to tracking and ensuring that packages are managed in a reproducible fashion. `packrat`, in particular, is a very powerful tool that RStudio uses within its enterprise offerings. When using `packrat`, the most important thing to track and maintain is the `packrat/packrat.lock` file, which houses everything needed to rebuild a project.

Though `packrat` is a powerful tool, it also has some user experience problems that can make working with it tricky. RStudio is in the process of building a reimaginination of packrat that keeps its strengths while discarding some of the difficulties. If using `packrat` to manage reproducibility, discussion is welcome at community.rstudio.com for how to manage projects.

Specifically, a few highlights:

- do not version control package sources
- use the packrat global cache if possible
- version control the `packrat/packrat.lock` file
- consider version controlling `packrat/packrat.opts`

THE WHOLE PROCESS

The chef has now done a full audit of her ingredient source, the method by which she acquires the ingredients, and the ingredients actually used within a given recipe. If she has any further desires to think about reproducibility, she is leaning towards bundling up this entire process.

The way that is done in the digital realm is with virtual machine images or a containerization tool like Docker or Singularity. In this paradigm, the researcher desires to "snapshot" the entire kitchen (R packages, system dependencies, operating system) and all ingredients, utensils, or other components of the process.

There is much thinking about this paradigm, but it is mostly orthogonal to the discussion of R packages. If R packages are not being maintained in a reproducible fashion as discussed above, the reproducibility that these whole-environment snapshots will provide still remains limited.

As a result, we recommend to start the journey of reproducibility by managing R packages as discussed above. When moving into an ephemeral paradigm like Docker, Singularity, Kubernetes, or Infrastructure as Code, the researcher will benefit from having thought deeply about package reproducibility and many of the lessons learned will be transferrable to this alternative philosophy.

CONCLUSION

When a chef embarks on a quest to make delicious food, she most often begins in more humble employment than at the most famous restaurants around. In such employment, she may not need the same standard of reproducibility in her work that she will once established and making world-renown dishes.

Similarly, not every researcher or R programmer needs to make the same decisions around reproducibility for R packages. Armed with the appropriate best practices and axes upon which decisions can be made, the researcher is free to explore her package sources, package selection, and list of installed packages. By adequately managing these fronts, she is freed from the concern that open source efficiency and high quality reproducibility are diametrically opposed. Rather, the open source ecosystem serves to optimize, improve, and extend even the reproducibility of packages over time, paving the way to a more reproducible (and tasty) future.