OXFORD

BioScience, 2025, **0**, 1–5
https://doi.org/10.1093/biosci/biaf148
Advance access publication date: 0 2025
Professional Biologist

Dependencies, archivals, and package development in the R ecosystem

Tad A. Dallas and Michael Krabbe Borregaard

Tad A. Dallas (tad.a.dallas@gmail.com) is affiliated with the Department of Biological Sciences at the University of South Carolina, in Columbia, South Carolina, in the United States. Michael Krabbe Borregaard is affiliated with the Center for Macroecology, Evolution, and Climate at the University of Copenhagen, in Copenhagen, Denmark.

Abstract

The accelerating rate of statistical package development has allowed researchers to easily perform increasingly advanced statistical analyses and has enhanced engagement of researchers in both writing analytical code and developing packages. However, this increase in the number of packages forms a correspondingly complex web of package interdependencies. This web aligns the many packages into a coherent ecosystem and greatly facilitates the ongoing development of new functionality, but package dependencies are not necessarily coordinated and, as , such also represent potential points for code to break. In the present article, we explore temporal trends in R package development, highlighting the rise in packages, dependencies, and archivals (as a way to quantify breaking packages). Following R Core Team developer Kurt Hornik's question from over a decade ago, we wonder "Are there too many R packages?"

Keywords: reproducibility, package maintenance, dependencies, CRAN, R

R Core Team developer

Kurt Hornik asked this question in 2012 (Hornik 2012). In light of further computational development, the increase in the use of R, and the continued generation of numerous R packages, it seems like a good time to revisit this question. At the time Kurt Hornik published his perspective (Hornik 2012), there were 3425 packages on the Comprehensive R Archive Network (CRAN). This caused concern over the superabundance of packages leading to burden on the CRAN team and on the end user to find the *right* package among an ocean of possibilities. This number has now grown to 22,638 (as of September 2025) on CRAN, which excludes R packages hosted only on GitHub, BioConductor, or elsewhere. This means that some of the concerns expressed in Hornik (2012) and elsewhere (Theußl et al. 2011, Claes et al. 2014, Decan et al. 2016) may now be even more important.

However, the question of whether there are too many packages is vaguely defined, and we suggest rephrasing the question to one of package ecosystem stability. Individual R packages do not exist in isolation. Often, packages rely on other existing packages for much of their functionality, packages that are imported as dependencies. Over time, a type of dependency network forms, where specialized end user packages rely on other more general packages, which again rely on even more basal packages. The resulting network is informally referred to as an *ecosystem*, where the ecological metaphor signifies that emergent outcomes of interactions between packages follow complex but tractable rules. We argue in the present article that the question of whether there are too many R packages needs a definition of what *too many* means, and we explore this question in the context of the stability of the R ecosystem.

Ecological stability is in itself a tricky concept (Mikkelson 1997), which could both mean the ability to not be affected (resistance) by a disturbance or the ability to recover quickly (resilience) from it (Harrison 1979, Donohue et al. 2016). In the study of networks, a common measure of resistance would be the bond percolation threshold, or the threshold of link removal that causes the graph to fracture (Newman 1986, Radicchi 2015). In the present article, we focus on the resistance of the R ecosystem to disturbances that appear naturally as a function of package development and deprecation. The second part of Kurt Hornik's question, whether the right package becomes too hard to find, is orthogonal and possibly ameliorated by dedicated package search mechanisms, such as CRAN task views, which curates recommended packages around an analytical theme (Zeileis 2005).

A package dependency represents code that is often not under control of the package author, representing a point of potential failure (Frakes and Kang 2005, Cox 2019). Package maintainers make the implicit assumption that dependencies will be available long term and that any further development of those dependencies will not break the functionality of their package. However, different package versions may behave differently, and packages become deprecated or archived over time, such as if they fail to meet CRAN checks, potentially breaking depending package functionality and previously reproducible workflows.

The issue of potential deprecation is particularly pronounced for packages that are basal in the ecosystem (i.e., which have many packages that depend on them, *reverse dependencies*). Packages that rely on them may again be relied on by other packages, creating second or third order transitive dependencies. Errors in such packages may have widespread consequences for analytical pipelines across entire scientific fields. For instance, the rise of ggplot2 as a way to visualize data has led to the production of over 1300 packages that rely on ggplot2 and over 40 packages that specifically have the word *ggplot* in the package title, suggesting that they use ggplot2 to produce a specific type of visualization. An error in one of the 34 dependencies of ggplot2 therefore

Received: September 17, 2024. **Revised:** June 10, 2025. **Accepted:** August 20, 2025

potentially has far-reaching consequences. An even more concerning possibility is the potential for silent bugs within a widely used dependency that may not lead directly to package failure but silently give a wrong result, leading to intractable errors in analytical pipelines. Software bugs have had devastating consequences for academic careers in the past (Miller 2006), and there is no reason to assume this will not continue happening.

On the other hand, dependencies may also further ecosystem stability. Errors in widely used dependencies are often discovered and resolved quickly because of the wide user base (i.e., they confer ecosystem resilience), and reliance on the same functionality from multiple packages should increase the consistency and replicability of analyses. Furthermore, the development of R packages that form a cohesive dependency structure might be more resistant to package failures (e.g., tidyverse; Wickham et al. 2019), because these currently have a dedicated development team built around common design principles. Packages with a large number of reverse dependencies may also be less prone to noisy failure; they are too big to fail quietly, because the failure of such a package would have far-reaching consequences. Finally, deliberately modular package designs, where small packages each handle a single well-defined task and are then imported by multiple other packages, leads to a high number of dependencies but, from the perspective of the ecosystem, reduce the number of points of failure. Small modular packages also tend to be simpler, leading to more stable packages that are easier to maintain.

In the present article, we address the potential consequences of numbers of packages and patterns in dependency structure, with a focus on packages accessible via CRAN. We acknowledge that R packages are also distributed through other means such as GitHub, a version control platform, and other platforms such as BioConductor. We restricted our exploration to CRAN, because it is maintained by developers close to the R Core Team development group and is supported by the R Foundation. Most importantly, CRAN has clear standards of practice for submitted packages to ensure that they adhere to R package stylistic standards, as well as build checks before a package is hosted. We explore the current landscape of the CRAN ecosystem, highlighting a number of issues that may hinder long-term reproducibility of scientific analyses. Specifically, we focus on the rise in package dependencies and package deprecation rates, with implications for reproducibility. Our goal is to call attention to how the landscape of package development affects the structure of package interdependencies and the reproducibility of the analytical pipelines that depend on these packages.

Placing R in the context of other languages

R is not unique in the increased number of packages or libraries available (table 1). Importantly, the issues of the stability of package code is not particular to the R programming language but is a property of the ongoing development of any massively decentralized software system. The Linux package ecosystem has spent decades developing ways of navigating dependency hell (a situation where two installed packages depend on mutually incompatible versions of the same dependency, leading attempts to install the two packages on the same system to fail). Open source package ecosystems for modern programming languages, such as Python, Julia, Lua, and JavaScript, all have different solutions to ensuring software stability.

Other language ecosystems face even higher numbers of potentially deprecated packages or fail points for dependencies. For instance, Rust has over 165,000 packages (called *crates*), whereas javascript has over 3 million packages hosted on npm, which has adopted a modular structure of small and highly specialized packages. This includes, for example, a package whose sole purpose is to tell whether a number is odd or even: is-odd. At the time of writing this, is-odd had 129 reverse dependencies (packages that depend on it). On the other hand, the more recent (and much smaller) Julia package ecosystem has adopted a system where each project creates its own separate package environment (the set of installed packages and their versions). This reduces the potential for dependency incompatibilities, because it allows different projects to rely on different versions of a given package. Although this represents one promising avenue for ensuring long-term stability, the best way of ensuring software ecosystem stability remains a matter of ongoing development.

Table 1. A comparison of different programming languages in terms of the name of the package management system (if any), the number of packages or libraries available, and the year of first release of the language.

Language	Package manager	Number packages	First release
Python	pip or conda	570,000+	1991
R	install.packages()	22,638	1993
javascript	npm	3.1 million	1995
Julia	Pkg	11,728	2009
Rust	cargo	165,468	2012

In the present article, we focus on the R ecosystem, because R is a widely used language by the biological research community (Lai et al. 2019). This decision comes with no implication that the R ecosystem is particularly problematic, and, in fact, CRAN policies, such as the requirement for tests to pass (if they are defined), are put in place as ways to safeguard ecosystem package stability.

The rise in package dependencies

As a programming language matures and is adopted more widely, it will generally tend to accumulate packages, and packages will tend to accumulate dependencies. The proliferation of tools making it straightforward to write an R package strengthens this tendency (Wickham 2015). New dependencies may implement more functionality or diversify existing functionality. This pattern is apparent in the R ecosystem, with an increasing tendency of updates and releases of existing R packages to have more imports over time (imports are dependencies that are installed and typically loaded; figure 1).

Each additional dependency comes with a probability that it will be deprecated, be archived, or undergo breaking changes. Many of the developers maintaining R packages may not consider long-term maintenance costs, increasing this risk with time. In professional software development, there are established practices for handling this situation. Package authors can write unit tests, which test the functionality of a package, and a failure of these unit tests will alert the authors of any changes to the usability and integrity of the code. A simple form of unit tests is performed by CRAN, because the package is routinely recompiled, which requires running the examples of any functions and compile the vignette. Packages that fail these automated tests cause the CRAN team to alert the package maintainer and to archive the package if the issue persists. This makes sure that all packages can at least still be built, but it does require that the appropriate unit tests are defined. Equally important is that the unit

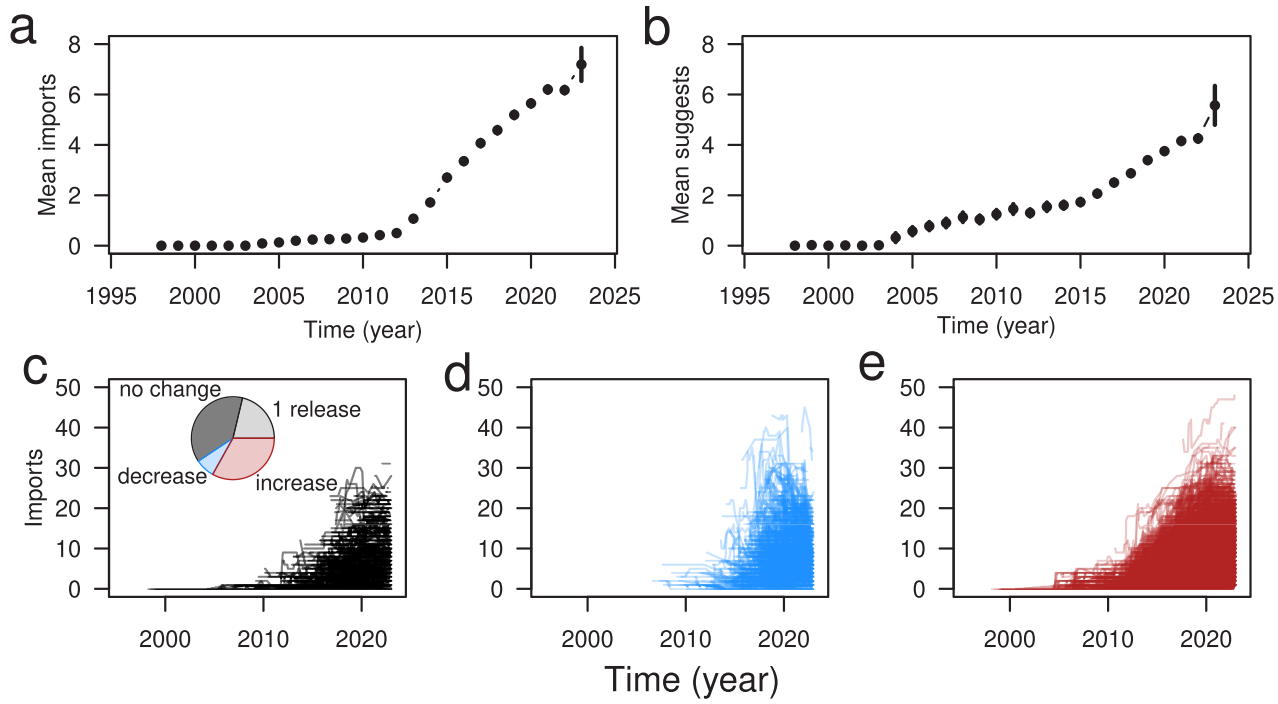

Figure 1. The average number of imports and suggests are increasing through time (a–b). The error bars represent twice the standard error. For a given package, where each package is represented as a line in panels (c–e), the number of imported packages through time may not change (c), may decrease (d), or may increase (e). The inset pie chart highlight the tendency for packages to increase imports, or remain unchanged (note that a sizable portion of this corresponds to single release packages with no active R maintenance). On the basis of these time series trajectories, we estimate that the average package is increasing the number of imports and suggests by 1 new package every 730 or 672 days, respectively.

tests realize a broad coverage of the package code. This means that they should test all possible function calls, with all combinations of input arguments, which is a fairly demanding task on the package author. With the creation of the testthat package in R (Wickham 2011), unit testing in R has become more common (see the number of reverse dependencies and *suggests* on the testthat CRAN webpage, doi.org/10.32614/CRAN.package.testthat). Adopting a culture of expecting a good unit test code coverage seems key to ensure the stability of the biology R ecosystem and should be emphasized by, for example, reviewers and journal editors. However, this is still likely a relatively small subset of R packages, and the adoption of these methods is going to vary by subfield. This creates a potentially unstable situation for subfield package ecosystems with implications for the reproducibility and integrity of research pipelines.

Some of these issues stem from the structure of R's package manager itself, which does not allow for careful version control or enforce downstream testing. It is important to note that package managers such as renv (Ushey 2022) go a long way in ensuring specific package versions are used, including the ability to install archived CRAN packages or packages from GitHub or BioConductor. However, the widespread adoption of these managers and of other tools such as Docker (Boettiger 2015), is not yet realized by R users, although package managers are commonly used in other languages such as python (pip and conda) and julia (Pkg). Furthermore, there are ongoing conversations about how to improve dependency versioning and package management in R (Ooms 2013). This puts the onus on the package author for being aware of recursive dependencies and programming defensively.

Package deprecation and archival

The rate of package deprecation on CRAN is a reasonable metric for the frequency of changes that potentially break workflows and can be at least partially observed through the rate of package archivals. Packages are archived for a number of reasons, the most common being that there are uncorrected issues during package checks, that the package depends on a package not on CRAN, or that there are broken dependencies. With the accelerating rate of package creation and with developers potentially not considering the long-term cost of package maintenance, we also see an increasing number of archivals, although the past 2 years have seen a decrease in archivals despite a steady increase in the number of average dependencies (figure 2).

The variety of R package purposes

Packages in R have a variety of purposes, including data analysis, visualization, data access, and interfacing with web resources or other programming languages. There are likely temporal trends in the types of R packages that are being developed and maintained. R package creation (devtools), testing (testthat), and documentation (roxygen2) have been straightforward with the continued maintenance of central tools. Newer tools to construct web documentation from the R package structure (pkgdown) provide a way to introduce new ways for users to view package documentation. No R packages should be seen as a standalone tool, just as most analytical workflows rarely stand alone (e.g., the analyses in the present article resulted in 5 attached packages and 37 loaded via namespace).

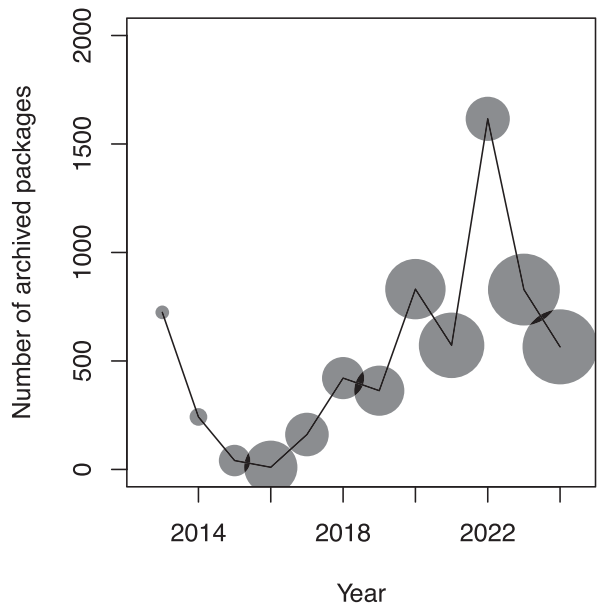

Figure 2. The total number of archived packages broken down by year, with point size proportional to the mean number of imports the package had. The highest number of archived packages was in 2022, with 1853 archived packages.

Moving forward

Much like Kurt Hornik, we would not be willing to predict the number of R packages on CRAN in 10 years' time (Hornik 2012), nor are we willing to say that there are "too many" R packages currently. As we have tried to highlight, it is not necessarily the raw number of packages that should be concerning but the dependencies of packages on others. Most notably, the formation of clusters within R package dependencies serves to increase package reliance on nonbase R packages that may change depending on developer decisions. Some of these effects may be too early to see, such as the creation of the tidyverse R package (Wickham et al. 2019), which depends on 29 other packages (116 if recursive dependencies are considered). Currently, less than 50 packages depend on the tidyverse package, and the developers of tidyverse have cautioned users against using tidyverse as a package dependency. Despite this, it is likely to receive a wide uptake for analytical pipelines for particular scientific analyses—in particular, seeing how often the tidyverse is used as a teaching tool (Cetinkaya-Rundel et al. 2022). The extremely popular ggplot2 already followed a dramatically increasing trajectory, with 489 reverse dependencies, 4014 reverse imports, and 1599 reverse suggests as of August 2024, on the basis of the description file of the ggplot2 package. These extremely popular packages form clusters in R ecosystem space. Although their popularity means that any breakages can have potentially far-reaching consequences, they also mean that failures are very quickly noticed and acted on. The shift toward R package development and maintenance on platforms other than CRAN (e.g., GitHub) and aggregators of R packages such as R-Universe offer promising ways to find relevant R packages that are supported by the R foundation and ROpenSci. R-Universe also has an API (application programming interface), allowing for analyses such as the ones we have performed to be done on packages outside of CRAN.

The increase in the number of R packages and the availability of common tools to manipulate, analyze, and visualize data (e.g., the *tidyverse*) has allowed newcomers and experienced R coders alike to quickly gain insights from data. And it is important to note that these issues are hardly unique to the R programming environment, because other commonly used programming languages have similar interdependency issues (Decan et al. 2016, Schueller et al. 2022, Jia et al. 2024). As statistical tools, data types, and analytical pipelines become more and more complex, relying on existing tools is not inherently a negative thing. However, reliance on too many tools or relying on tools that rely on numerous other tools may start to create a situation where the negative effects of dependencies can influence the long-term availability of packages. Scientific journals employing data editors (e.g., *American Naturalist*, *Ecology Letters*) helps ensure reproducibility at the time of publication, but long-term reproducibility is truly in the hands of package developers, code authors, and possibly code editors or reviewers. These roles are largely held academic researchers, along with a mix of other roles such as reviewer, editor, manager, and fundraiser. That is, package maintenance and stability are largely in our collective hands.

Acknowledgments

This material is based on work supported by the National Science Foundation (NSF DBI award no. 2329961).

Author contributions

Tad A. Dallas (Conceptualization, Formal analysis, Investigation, Writing - original draft), and Michael Krabbe Borregaard (Conceptualization, Formal analysis, Investigation, Writing - original draft, Writing - review & editing)

R code is available on figshare at https://doi.org/10.6084/m9.figshare.25537282.v1.

References cited

Boettiger C. 2015. An introduction to Docker for reproducible research. *ACM SIGOPS Operating Systems Review* 49: 71–79.

Cetinkaya-Rundel M, Hardin J, Baumer BS, McNamara A, Horton NJ, Rundel C. 2022. An educator's perspective of the tidyverse. *Technological Innovations in Statistics Education* 14: T514154352. https://doi.org/10.5070/T514154352

Claes M, Mens T, Grosjean P. 2014. On the maintainability of CRAN packages. Pages 308–312 in *2014 Software Evolution Week: IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*. Institute of Electrical and Electronics Engineers.

Cox R. 2019. Surviving software dependencies. *Communications of the ACM* 62: 36–43.

Decan A, Mens T, Claes M, Grosjean P. 2016. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. Pages 493–504 in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. Institute of Electrical and Electronics Engineers.

Donohue I, et al. 2016. Navigating the complexity of ecological stability. *Ecology Letters* 19: 1172–1185.

Frakes WB, Kang K. 2005. Software reuse research: Status and future. *IEEE Transactions on Software Engineering* 31: 529–536.

Harrison GW. 1979. Stability under environmental stress: Resistance, resilience, persistence, and variability. *American Naturalist* 113: 659–669.

Hornik K. 2012. Are there too many R packages? *Austrian Journal of Statistics* 41: 59–66.

Jia X, Zhou Y, Hussain Y, Yang W. 2024. An empirical study on Python library dependency and conflict issues. Pages 504–515 in *2024 IEEE*

24th International Conference on Software Quality, Reliability and Security (QRS). Institute of Electrical and Electronics Engineers.

Lai J, Lortie CJ, Muenchen RA, Yang J, Ma K. 2019. Evaluating the popularity of R in ecology. *Ecosphere* 10: e02567.

Mikkelson GM. 1997. The problem of defining stability. *Perspectives on Science* 5: 481–498.

Miller G. 2006. Scientific publishing. A scientist's nightmare: Software problem leads to five retractions. *Science* 314: 1856–1857.

Newman CM. 1986. Percolation theory: A selective survey of rigorous results. Pages 147–167 in Papanicolauo G, eds. *Advances in Multiphase Flow and Related Problems*. SIAM.

Ooms J. 2013. Possible directions for improving dependency versioning in R. *R Journal* 5: 197–206. https://journal.r-project.org/archive/2013-1

Radicchi F. 2015. Predicting percolation thresholds in networks. *Physical Review E* 91: 010801.

Schueller W, Wachs J, Servedio VD, Thurner S, Loreto V. 2022. Evolving collaboration, dependencies, and use in the rust open source software ecosystem. *Scientific Data* 9: 703.

Theußl S, Ligges U, Hornik K. 2011. Prospects and challenges in R package development. *Computational Statistics* 26: 395–404.

Ushey K. 2022. *renv: Project Environments*, vers. 0.16.0. The Comprehensive R Archive Network. https://CRAN.R-project.org/package=renv

Wickham H. 2011. testthat: Get started with testing. *R Journal* 3: 5–10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf

Wickham H. 2015. *R Packages: Organize, Test, Document, and Share Your Code*. O'Reilly.

Wickham H, et al. 2019. Welcome to the tidyverse. *Journal of Open Source Software* 4: 1686.

Zeileis A. 2005. CRAN task views. *R News* 5: 39–40.

Received: September 17, 2024. **Revised:** June 10, 2025. **Accepted:** August 20, 2025