title in pt

author in pt

Índice

rOpenSci - Guia para desenvolvedores


```
@software{ropensci_2024_10797633,
  author       = {rOpenSci and
                  Anderson, Brooke and
                  Chamberlain, Scott and
                  DeCicco, Laura and
                  Gustavsen, Julia and
                  Krystalli, Anna and
                  Lepore, Mauro and
                  Mullen, Lincoln and
                  Ram, Karthik and
                  Ross, Noam and
                  Salmon, Maëlle and
                  Vidoni, Melina and
                  Riederer, Emily and
                  Sparks, Adam and
                  Hollister, Jeff},
  title        = {{rOpenSci Packages: Development, Maintenance, and
                   Peer Review}},
  month        = mar,
  year         = 2024,
  publisher    = {Zenodo},
  version      = {0.9.0},
  doi          = {10.5281/zenodo.10797633},
  url          = {https://doi.org/10.5281/zenodo.10797633}
}
```

Você também pode ler a versão em PDF deste livro.

Prefácio

Boas vindas! Este livro é um guia para autores, mantenedores, revisores e editores da rOpenSci.

A primeira seção do livro contém as nossas diretrizes para criar e testar pacotes do R.

A segunda seção é dedicada ao processo de revisão por pares de software da rOpenSci: o que é esse processo, quais são as nossas políticas e guias específicos para autores, editores e revisores durante todo o processo. Para *revisão de software estatístico*, consulte a página da Web e os recursos do projeto.

A terceira e última seção apresenta as nossas práticas recomendadas para você cuidar do seu pacote depois que ele tiver sido integrado: como colaborar com outros desenvolvedores, como documentar lançamentos, como promover o seu pacote e como aproveitar o GitHub como uma plataforma de desenvolvimento. A terceira seção também apresenta um capítulo para quem deseja começar a contribuir com os pacotes do rOpenSci.

Esperamos que você ache o guia útil e claro, e agradecemos suas sugestões no *issue tracker* do livro.
Feliz embalagem R!

A equipe editorial da rOpenSci.

Este livro é um documento vivo. Você pode ver as atualizações das nossas práticas recomendadas e políticas nas notas de versão.
Você pode citar este livro usando os metadados Zenodo e DOI.

Em Markowitz (NOAA) · Sam Albers · Toph Allen · Kaique dos S. Alves · Alison Appling · Denisse Fierro Arcos · Zebulun Arendsee · Taylor Arnold · Al-Ahmadgaid B. Asaad · Dean Attali · Mara Averick · Suzan Baert · James Balamuta · Vikram Baliga · David Bapst · Joëlle Barido-Sottani · Allison Barner · Cale Basaraba · John Baumgartner · Marcus Beck · Gabriel Becker · Jason Becker · Salvador Jesus Fernandez Bejarano · Dom Bennett · Ken Benoit · Aaron Berdanier · Fred Boehm · Carl Boettiger · Will Bolton · Ben Bond-Lamberty · Anne-Sophie Bonnet-Lebrun · Alison Boyer · Abby Bratt · François Briatte · Eric Brown · Julien Brun · Jenny Bryan · Lukas Burk · Lorenzo Busetto · Maria Paula Caldas · Mario Gavidia Calderón · Carlos Cámara-Menoyo · Brad Cannell · Joaquin Cavieres · Kevin Cazelles · Cathy Chamberlin · Jennifer Chang · Pierre Chausse · Jorge Cimentada · Nicholas Clark · Chase Clark · Jon Clayden · Dena Jane Clink · Will Cornwell · Nic Crane · Enrico Crema · Verónica Cruz-Alonso · Ildiko Czeller · Tad Dallas · Kauê de Sousa · Christophe Dervieux · Amanda Dobbyn · Jasmine Dumas · Dewey Dunnington · Remko Duursma · Mark Edmondson · Paul Egeler · Evan Eskew · Harry Eslick · Alexander Fischer · Kim Fitter · Robert M Flight · Sydney Foks · Stephen Formel · Zachary Stephen Longiaru Foster · Auriel Fournier · Kaija Gahm · Zach Gajewski · Carl Ganz · Duncan Garmonsway ·

Jan Laurens Geffert · Sharla Gelfand · Monica Gerber · Duncan Gillespie · David Gohel · A. Cagri gokcek · Guadalupe Gonzalez · Rohit Goswami · Laura Graham · Charles Gray · Matthias Grenié · Corinna Gries · Hugo Gruson · Ernest Guevarra · W Kyle Hamilton · Ivan Hanigan · Jeffrey Hanson · Liz Hare · Jon Harmon · Rayna Harris · Ted Hart · Nujcharee Haswell · Verena Haunschmid · Stephanie Hazlitt · Andrew Heiss · Max Held · Anna Hepworth · Bea Hernandez · Jim Hester · Peter Hickey · Roel Hogervorst · Kelly Hondula · Allison Horst · Sean Hughes · James Hunter · Brandon Hurr · Ger Inberg · Christopher Jackson · Najko Jahn · Tamora D James · Veronica Jimenez-Jacinto · Mike Johnson · Will Jones · Max Joseph · Megha Joshi · Krunoslav Juraic · Soumya Kalra · Zhian N. Kamvar · Michael Kane · Andee Kaplan · Tinula Kariyawasam · Hazel Kavılı · Jonathan Keane · Christopher T. Kenny · Os Keyes · Eunseop Kim · Aaron A. King · Michael Koontz · Bianca Kramer · Will Landau · Sam Lapp · Erin LeDell · Thomas Leeper · Sam Levin · Lisa Levinson · Stephanie Locke · Marion Louveaux · Robin Lovelace · Julia Stewart Lowndes · Tim Lucas · Muralidhar, M.A. · Andrew MacDonald · Jesse Maegan · Mike Mahoney · Tristan Mahr · Paula Andrea Martinez · Joao Martins · Ben Marwick · Claire Mason · Miles McBain · Lucy D'Agostino McGowan · Amelia McNamara · Elaine McVey · Bryce Mecum · Nolwenn Le Meur · François Michonneau · Mario Miguel · Helen Miller · Jessica Minnier · Priscilla Minotti · Nichole Monhait · Kelsey Montgomery · Ronny A. Hernández Mora · Natalia Morandeira · Ross Mounce · Athanasia Monika Mowinckel · Lincoln Mullen · Matt Mulvahill · Maria Victoria Munafó · David Neuzerling · Dillon Niederhut · Joel Nitta · Rory Nolan · Kari Norman · Jakub Nowosad · Matt Nunes · Daniel Nüst · Lauren O'Brien · Joseph O'Brien · Paul Oldham · Samantha Oliver · Dan Olner · Jeroen Ooms · Victor Ordu · Luis Osorio · Philipp Ottolinger · Mark Padgham · Marina Papadopoulou · Edzer Pebesma · Thomas Lin Pedersen · Antonio J. Pérez-Luque · Marcelo S. Perlin · Rafael Pilliard-Hellwig · Rodrigo Neto Pires · Lindsay Platt · Nicholas Potter · Joanne Potts · Josep Pueyo-Ros · Etienne Racine · Manuel Ramon · Nistara Randhawa · David Ranzolin · Quentin Read · Neal Richardson · tyler rinker · Emily Robinson · David Robinson · Alec Robitaille · Sam Rogers · Julia Romanowska · Xavier Rotllan-Puig · Bob Rudis · Edgar Ruiz · Kent Russel · Michael Sachs · Sheila Saia · Chitra M Saraswati · Alicia Schep · Klaus Schliep · Clemens Schmid · Patrick Schratz · Collin Schwantes · Marco Sciaini · Eric Scott · Heidi Seibold · Julia Silge · Peter Slaughter · Mike Smith · Tuija Sonkkila · Øystein Sørensen · Jemma Stachelek · Christine Stawitz · Irene Steves · Kelly Street · Matt Strimas-Mackey · Alex Stringer · Michael Sumner · Chung-Kai Sun · Sarah Supp · Jason Taylor · Filipe Teixeira · Andy Teucher · Jennifer Thompson · Joe Thorley · Nicholas Tierney · Tiffany Timbers · Tan Tran · Tim Trice · Utku Turk · Kyle Ueyama · Ted Underwood · Adithi R. Upadhya · Kevin Ushey · Josef Uyeda · Frans van Dunné · Mauricio Vargas · Remi Vergnon · Jake Wagner · Ben Ward · Elin Waring · Rachel Warnock · Leah Wasser · David Watkins · Lukas Weber · Marc Weber · Karissa Whiting · Stefan Widgren · Anna Willoughby · Saras Windecker · Luke Winslow · David Winter · Witold Wolski · Kara Woo · Marvin N. Wright · Jacob Wujciak-Jens · Bruna Wundervald · Lauren Yamane · Emily Zabor · Taras Zakharko · Hao Zhu · Chava Zibman · Naupaka Zimmerman · Jake Zwart · Felipe · santikka · kasselhingee · Bri · Flury · Vincent · eholmes · Pachá · Rich · Claudia · Jasmine · Zack · Lluís · becarioprecario · gaurav

Se você quiser contribuir com este livro (sugestões, correções), consulte o repositório do GitHub em particular as diretrizes de contribuição. Obrigado!

Agradecemos a todos os autores, revisores e editores convidados por nos ajudarem a aprimorar o sistema e este guia ao longo dos anos. Agradecemos também às seguintes pessoas que fizeram contribuições para este guia e suas versões anteriores: Katrin Leinweber, John Baumgartner, François Mi-

chonneau, Christophe Dervieux, Lorenzo Busetto, Ben Marwick, Nicholas Horton, Chris Kennedy, Mark Padgham, Jeroen Ooms, Sean Hughes, Jan Gorecki, Joseph Stachelek, Dean Attali, Julia Gustavsen, Nicholas Tierney, Rich FitzJohn, Tiffany Timbers, Hilmar Lapp, Miles McBain, Bryce Mecum, Jonathan Carroll, Carl Boettiger, Florian Privé, Stefanie Butland, Daniel Possenriede, Hadley Wickham, Mauro Lepore, Matthew Fidler, Luke McGuinness, Aaron Wolen, Indrajeet Patil, Kevin Wright, Will Landau, Hugo Gruson„ Hao Ye„ Sébastien Rochette„ Edward Wallace„ Alexander Fischer„ Maxime Jaunatre„ Thomas Zwagerman. Informe-nos se esquecemos de reconhecer a sua contribuição!

Parte I

Building Your Package

1 Packaging Guide

rOpenSci accepts packages that meet our guidelines via a streamlined Software Peer Review process. To ensure a consistent style across all of our tools we have written this chapter highlighting our guidelines for package development. Please also read and apply our chapter about continuous integration (CI). Further guidance for after the review process is provided in the third section of this book starting with a chapter about collaboration.

We recommend that package developers read Hadley Wickham and Jenny Bryan's thorough book on package development which is available for free online. Our guide is partially redundant with other resources but highlights rOpenSci's guidelines.

To read why submitting a package to rOpenSci is worth the effort to meet guidelines, have a look at reasons to submit.

1.1 Package name and metadata

1.1.1 Naming your package

- We strongly recommend short, descriptive names in lower case. If your package deals with one or more commercial services, please make sure the name does not violate branding guidelines. You can check if your package name is available, informative and not offensive by using the `pak::pkg_name_check()` function; also use a search engine as you'd thus see if it's offensive in a language other than English. In particular, do *not* choose a package name that's already used on CRAN or Bioconductor.
- There is a trade-off between the advantages of a unique package name and a less original package name.
  - A more unique package name might be easier to track (for you and us to assess package use for instance, less false positives when typing its name in GitHub code search) and search (for users to ask "how to use package blah" in a search engine).
  - On the other hand a *too* unique package name might make the package less discoverable (that is to say, to find it by searching "how to do this-thing in R"). It might be an argument for naming your package something very close to its topic such as geojson).

- Find other interesting aspects of naming your package in this blog post by Nick Tierney, and in case you change your mind, find out how to rename your package in this other blog post of Nick's.

1.1.2 Creating metadata for your package

We recommend you to use the `codemetar` package for creating and updating a JSON CodeMeta metadata file for your package via `codemetar::write_codemeta()`. It will automatically include all useful information, including GitHub topics. CodeMeta uses Schema.org terms so as it gains popularity the JSON metadata of your package might be used by third-party services, maybe even search engines.

1.2 Platforms

- Packages should run on all major platforms (Windows, macOS, Linux). Exceptions may be granted packages that interact with system-specific functions, or wrappers for utilities that only operate on limited platforms, but authors should make every effort for cross-platform compatibility, including system-specific compilation, or containerization of external utilities.

1.3 Package API

1.3.1 Function and argument naming

- Functions and arguments naming should be chosen to work together to form a common, logical programming API that is easy to read, and auto-complete.
  - Consider an `object_verb()` naming scheme for functions in your package that take a common data type or interact with a common API. `object` refers to the data/API and `verb` the primary action. This scheme helps avoid namespace conflicts with packages that may have similar verbs, and makes code readable and easy to auto-complete. For instance, in **stringi**, functions starting with `stri_` manipulate strings (`stri_join()`, `stri_sort()`, and in **googlesheets** functions starting with `gs_` are calls to the Google Sheets API (`gs_auth()`, `gs_user()`, `gs_download()`).
- For functions that manipulate an object/data and return an object/data of the same type, make the object/data the first argument of the function so as to enhance compatibility with the pipe operators (base R's `|>`, magrittr's `%>%`).
- We strongly recommend `snake_case` over all other styles unless you are porting over a package that is already in wide use.

- Avoid function name conflicts with base packages or other popular ones (e.g. `ggplot2`, `dplyr`, `magrittr`, `data.table`)
- Argument naming and order should be consistent across functions that use similar inputs.
- Package functions importing data should not import data to the global environment, but instead must return objects. Assignments to the global environment are to be avoided in general.

1.3.2 Console messages

- Use either the cli package, or base R's tools (`message()` and `warning()`) to communicate with the user in your functions.
- Highlights of the cli package include: automatic wrapping, respect of the NO_COLOR convention, many semantic elements, and extensive documentation. Read more in a blog post.
- Please do not use `print()` or `cat()` unless it's for a `print.*()` or `str.*()` methods, as these methods of printing messages are harder for users to suppress.
- Provide a way for users to opt out of verbosity, preferably at the package level: make message creation dependent on an environment variable or option (like "usethis.quiet" in the usethis package), rather than on a function parameter. The control of messages could be on several levels ("none,"inform", "debug") rather than logical (no messages at all / all messages). Control of verbosity is useful for end users but also in tests. More interesting comments can be found in an issue of the tidyverse design guide.

1.3.3 Interactive/Graphical Interfaces

If providing graphical user interface (GUI) (such as a Shiny app), to facilitate workflow, include a mechanism to automatically reproduce steps taken in the GUI. This could include auto-generation of code to reproduce the same outcomes, output of intermediate values produced in the interactive tool, or simply clear and well-documented mapping between GUI actions and scripted functions. (See also "Testing" below.)

The `tabulizer` package e.g. has an interactive workflow to extract tables, but can also only extract coordinates so one can re-run things as a script. Besides, two examples of shiny apps that do code generation are https://gdancik.shinyapps.io/shinyGEO/, and https://github.com/wallaceEcoMod/wallace/.

1.3.4 Input checking

We recommend your package use a consistent method of your choice for checking inputs – either base R, an R package, or custom helpers.

1.3.5 Packages wrapping web resources (API clients)

If your package accesses a web API or another web resource,

- Make sure requests send an user agent, that is, a way to identify what (your package) or who sent the request. The users should be able to override the package's default user agent. Ideally the user agent should be different on continuous integration services, and in development (based on, for instance, the GitHub usernames of the developers).
- You might choose different (better) defaults than the API, in which case you should document them.
- Your package should help with pagination, by allowing the users to not worry about it at all since your package does all necessary requests.
- Your package should help with rate limiting according to the API rules.
- Your package should reproduce API errors, and possibly explain them in informative error messages.
- Your package could export high-level functions and low-level functions, the latter allowing users to call API endpoints directly with more control (like `gh::gh()`).

For more information refer to the blog post Why You Should (or Shouldn't) Build an API Client.

1.4 Code Style

- For more information on how to style your code, name functions, and R scripts inside the `R/` folder, we recommend reading the code chapter in The R Packages book. We recommend the `styler` package for automating part of the code styling. We suggest reading the Tidyverse style guide.
- You can choose to use = over <- as long you are consistent with one choice within your package. We recommend avoiding the use of -> for assignment within a package. If you do use <- throughout your package, and you also use `R6` in that package, you'll be forced to use = for assignment within your `R6Class` construction - this is not considered an inconsistency because you can't use <- in this case.

1.5 CITATION file

- If your package does not yet have a CITATION file, you can create one with `usethis::use_citation()`, and populate it with values generated by the `citation()` function.
- CRAN requires CITATION files to be declared as `bibentry` items, and not in the previously-accepted form of `citEntry()`.

- If you archive each release of your GitHub repo on Zenodo, add the Zenodo top-level DOI to the CITATION file.
- If one day **after** review at rOpenSci you publish a software publication about your package, add it to the CITATION file.
- Less related to your package itself but to what supports it: if your package wraps a particular resource such as data source or, say, statistical algorithm, remind users of how to cite that resource via e.g. `citHeader()`. Maybe even add the reference for the resource.

As an example see the dynamite CITATION file which refers to the R manual as well as other associated publications.

```
citHeader("To cite dynamite in publications use:")

bibentry(
  key = "dynamitepaper",
  bibtype  = "Misc",
  doi = "10.48550/ARXIV.2302.01607",
  url = "https://arxiv.org/abs/2302.01607",
  author = c(person("Santtu", "Tikka"), person("Jouni", "Helske")),
  title = "dynamite: An R Package for Dynamic Multivariate Panel Models",
  publisher = "arXiv",
  year = "2023"
)

bibentry(
  key = "dmpmpaper",
  bibtype  = "Misc",
  title    = "Estimating Causal Effects from Panel Data with Dynamic
    Multivariate Panel Models",
  author = c(person("Santtu", "Tikka"), person("Jouni", "Helske")),
  publisher = "SocArxiv",
  year     = "2022",
  url      = "https://osf.io/preprints/socarxiv/mdwu5/"
)

bibentry(
  key = "dynamite",
  bibtype  = "Manual",
  title    = "Bayesian Modeling and Causal Inference for Multivariate
    Longitudinal Data",
  author = c(person("Santtu", "Tikka"), person("Jouni", "Helske")),
  note  = "R package version 1.0.0",
```

```
  year      = "2022",
  url       = "https://github.com/ropensci/dynamite"
)
```

- You could also create and store a `CITATION.cff` thanks to the cffr package. It also provides a GitHub Action workflow to keep the `CITATION.cff` file up-to-date.

1.6 README

- All packages should have a README file, named `README.md`, in the root of the repository. The README should include, from top to bottom:
  - The package name.
  - Badges for continuous integration and test coverage, the badge for rOpenSci peer-review once it has started (see below), a repostatus.org badge, and any other badges (e.g. R-universe).
  - Short description of goals of package (what does it do? why should a potential user care?), with descriptive links to all vignettes unless the package is small and there's only one vignette repeating the README. Please also ensure the vignettes are rendered and readable, see the "documentation website" section).
  - Installation instructions using e.g. the remotes package, pak package, or R-universe.
  - Any additional setup required (authentication tokens, etc).
  - Brief demonstration usage.
  - If applicable, how the package compares to other similar packages and/or how it relates to other packages.
  - Citation information i.e. Direct users to the preferred citation in the README by adding boilerplate text "here's how to cite my package". See e.g. ecmwfr README.

If you use another repo status badge such as a lifecycle badge, please also add a repostatus.org badge. Example of a repo README with two repo status badges.

- Once you have submitted a package and it has passed editor checks, add a peer-review badge via

```
[![](https://badges.ropensci.org/<issue_id>_status.svg)](https://github.com/ropensci/software
```

where issue_id is the number of the issue in the software-review repository. For instance, the badge for `rtimicropem` review uses the number 126 since it's the review issue number. The badge will first indicated "under review" and then "peer-reviewed" once your package has been onboarded (issue labelled "approved" and closed), and will link to the review issue.

- If your README has many badges consider ordering them in an html table to make it easier for newcomers to gather information at a glance. See examples in `drake` repo and in `qualtRics` repo. Possible sections are
  - Development (CI statuses cf CI chapter, Slack channel for discussion, repostatus)
  - Release/Published (CRAN version and release date badges from METACRAN, CRAN checks API badge, Zenodo badge)
  - Stats/Usage (downloads e.g. download badges from r-hub/cranlogs) The table should be more wide than it is long in order to mask the rest of the README.
- If your package connects to a data source or online service, or wraps other software, consider that your package README may be the first point of entry for users. It should provide enough information for users to understand the nature of the data, service, or software, and provide links to other relevant data and documentation. For instance, a README should not merely read, "Provides access to GooberDB," but also include, "…, an online repository of Goober sightings in South America. More information about GooberDB, and documentation of database structure and metadata can be found at *link*".
- We recommend not creating `README.md` directly, but from a `README.Rmd` file (an R Markdown file) if you have any demonstration code. The advantage of the `.Rmd` file is you can combine text with code that can be easily updated whenever your package is updated.
- Consider using `usethis::use_readme_rmd()` to get a template for a `README.Rmd` file and to automatically set up a pre-commit hook to ensure that `README.md` is always newer than `README.Rmd`.
- Extensive examples should be kept for a vignette. If you want to make the vignettes more accessible before installing the package, we suggest creating a website for your package.
- Add a code of conduct and contribution guidelines.
- See the `gistr` README for a good example README to follow for a small package, and `bowerbird` README for a good example README for a larger package.

1.7 Documentation

1.7.1 General

- All exported package functions should be fully documented with examples.
- If there is potential overlap or confusion with other packages providing similar functionality or having a similar name, add a note in the README, main vignette and potentially the Description field of DESCRIPTION. Examples in rtweet README, rebird README, and the non-rOpensci package slurmR.

- The package should contain top-level documentation for `?foobar`, (or `` ?`foobar-package` `` if there is a naming conflict). Optionally, you can use both `?foobar` and `` ?`foobar-package` `` for the package level manual file, using `@aliases` roxygen tag. `usethis::use_package_doc()` adds the template for the top-level documentation.
- The package should contain at least one **HTML** vignette providing a substantial coverage of package functions, illustrating realistic use cases and how functions are intended to interact. If the package is small, the vignette and the README may have very similar content.
- As is the case for a README, top-level documentation or vignettes may be the first point of entry for users. If your package connects to a data source or online service, or wraps other software, it should provide enough information for users to understand the nature of the data, service, or software, and provide links to other relevant data and documentation. For instance, a vignette intro or documentation should not merely read, "Provides access to GooberDB," but also include, "…, an online repository of Goober sightings in South America. More information about GooberDB, and documentation of database structure and metadata can be found at *link*". Any vignette should outline prerequisite knowledge to be able to understand the vignette upfront.

The general vignette should present a series of examples progressing in complexity from basic to advanced usage.

- Functionality likely to be used by only more advanced users or developers might be better put in a separate vignette (e.g. programming/NSE with dplyr).
- The README, the top-level package docs, vignettes, websites, etc., should all have enough information at the beginning to get a high-level overview of the package and the services/data it connects to, and provide navigation to other relevant pieces of documentation. This is to follow the principle of *multiple points of entry* i.e. to take into account the fact that any piece of documentation may be the first encounter the user has with the package and/or the tool/data it wraps.
- The vignette(s) should include citations to software and papers where appropriate.
- If your package provides access to a data source, we require that DESCRIPTION contains both (1) A brief identification and/or description of the organisation responsible for issuing data; and (2) The URL linking to public-facing page providing, describing, or enabling data access (which may often differ from URL leading directly to data source).
- Only use package startup messages when necessary (function masking for instance). Avoid package startup messages like "This is foobar 2.4-0" or citation guidance because they can be annoying to the user. Rely on documentation for such guidance.
- You can choose to have a README section about use cases of your package (other packages, blog posts, etc.), example.

1.7.2 roxygen2 use

- We request all submissions to use roxygen2 for documentation. roxygen2 is an R package that compiles `.Rd` files to your `man` folder in your package from tags written above each function. roxygen2 has support for Markdown syntax. One key advantage of using roxygen2 is that your `NAMESPACE` will always be automatically generated and up to date.
- More information on using roxygen2 documentation is available in the R packages book and in roxygen2 website itself.
- If you were writing Rd directly without roxygen2, the Rd2roxygen package contains functions to convert Rd to roxygen documentation.
- All functions should document the type of object returned under the `@return` heading.
- The default value for each parameter should be clearly documented. For example, instead of writing `A logical value determining if ...`, you should write `A logical value (default `TRUE`) determining if ...`. It is also good practice to indicate the default values directly in your function definition:

```
f <- function(a = TRUE) {
  # function code
}
```

- Documentation should support user navigation by including useful cross-links between related functions and documenting related functions together in groups or in common help pages. In particular, the `@family` tags, that automatically creates "See also" links and can help group functions together on pkgdown sites, is recommended for this purpose. See the "manual" section of The R Packages book and the "function grouping" section of the present chapter for more details.
- You can re-use documentation pieces (e.g. details about authentication, related packages) across the vignettes/README/man pages. Refer to roxygen2 vignette on documentation reuse.
- For including examples, you can use the classic `@examples` tag (plural "examples") but also the `@example <path>` tag (singular "example") for storing the example code in a separate R script (ideally under `man/`), and the `@exampleIf` tag for running examples conditionally and avoiding R CMD check failures. Refer to roxygen2 documentation about examples.
- Add `#' @noRd` to internal functions. You might be interested in the devtag experimental package for getting local manual pages when using `#' @noRd`.
- Starting from roxygen2 version 7.0.0, `R6` classes are officially supported. See the roxygen2 docs for details on how to document `R6` classes.

1.7.3 URLs in documentation

This subsection is particularly relevant to authors wishing to submit their package to CRAN. CRAN will check URLs in your documentation and does not allow redirect status codes such as 301. You can use the urlchecker package to reproduce these checks and, in particular, replace URLs with the URLs they redirect to. Others have used the option to escape some URLs (change `<https://ropensci.org/>` to `https://ropensci.org/`, or `\url{https://ropensci.org/}` to `https://ropensci.org/`.), but if you do so, you will need to implement some sort of URL checking yourself to prevent them from getting broken without your noticing. Furthermore, links would not be clickable from local docs.

1.8 Documentation website

We recommend creating a documentation website for your package using `pkgdown`. The R packages book features a chapter on pkgdown, and of course `pkgdown` has its own documentation website.

There are a few elements we'd like to underline here.

1.8.1 Automatic deployment of the documentation website

You only need to worry about automatic deployment of your website until approval and transfer of your package repo to the ropensci organization; indeed, after that a pkgdown website will be built for your package after each push to the GitHub repo. You can find the status of these builds at `https://dev.ropensci.org/job/package_name`, e.g. for `magick`; and the website at `https://docs.ropensci.org/package_name`, e.g. for `magick`. The website build will use your pkgdown config file if you have one, except for the styling that will use the `rotemplate` package. The resulting website will have a local search bar. Please report bugs, questions and feature requests about the central builds at https://github.com/ropensci/docs/ and about the template at https://github.com/ropensci/rotemplate/.

If your package vignettes need credentials (API keys, tokens, etc.) to knit, you might want to precompute them since credentials cannot be used on the docs server.

Before submission and before transfer, you could use the approach documented by `pkgdown` or the `tic` package for automatic deployment of the package's website. This would save you the hassle of running (and remembering to run) `pkgdown::build_site()` yourself every time the site needs to be updated. First refer to our chapter on continuous integration if you're not familiar with continuous integration. In any case, do not forget to update all occurrences of the website URL after transfer to the ropensci organization.

1.8.2 Grouping functions in the reference

When your package has many functions, use grouping in the reference, which you can do more or less automatically.

If you use roxygen2 above version 6.1.1, you should use the `@family` tag in your functions documentation to indicate grouping. This will give you links between functions in the local documentation of the installed package ("See also" section) *and* allow you to use the `pkgdown` `has_concept` function in the config file of your website. Non-rOpenSci example courtesy of `optiRum`: family tag, `pkgdown` config file and resulting reference section. To customize the text of the cross-reference title created by roxygen2 (`Other {family}:`), refer to roxygen2 docs regarding how to provide a `rd_family_title` list in `man/roxygen/meta.R`.

Less automatically, see the example of `drake` website and associated config file.

1.8.3 Branding of authors

You can make the names of (some) authors clickable by adding their URL, and you can even replace their names with a logo (think rOpenSci… or your organisation/company!). See `pkgdown` documentation.

1.8.4 Tweaking the navbar

You can make your website content easier to browse by tweaking the navbar, refer to `pkgdown` documentation. In particular, note that if you name the main vignette of your package "pkg-name.Rmd", it'll be accessible from the navbar as a `Get started` link instead of via `Articles > Vignette Title`.

1.8.5 Math rendering

Please refer to pkgdown documentation. Our template is compatible with this configuration.

1.8.6 Package logo

To use your package logo in the pkgdown homepage, refer to `usethis::use_logo()`. If your package doesn't have any logo, the rOpenSci docs builder will use rOpenSci logo instead.

1.9 Authorship

The `DESCRIPTION` file of a package should list package authors and contributors to a package, using the `Authors@R` syntax to indicate their roles (author/creator/contributor etc.) if there is more than one author, and using the comment field to indicate the ORCID ID of each author, if they have one (cf this post). See this section of "Writing R Extensions" for details. If you feel that your reviewers have made a substantial contribution to the development of your package, you may list them in the `Authors@R` field with a Reviewer contributor type (`"rev"`), like so:

```
person("Bea", "Hernández", role = "rev",
comment = "Bea reviewed the package (v. X.X.XX) for rOpenSci, see <https://github.com/ro
```

Only include reviewers after asking for their consent. Read more in this blog post "Thanking Your Reviewers: Gratitude through Semantic Metadata". Please do not list editors as contributors. Your participation in and contribution to rOpenSci is thanks enough!

1.9.1 Authorship of included code

Many packages include code from other software. Whether entire files or single functions are included from other packages, rOpenSci packages should follow the CRAN *Repository Policy*:

> The ownership of copyright and intellectual property rights of all components of the package must be clear and unambiguous (including from the authors specification in the DESCRIPTION file). Where code is copied (or derived) from the work of others (including from R itself), care must be taken that any copyright/license statements are preserved and authorship is not misrepresented.
>
> Preferably, an 'Authors@R' field would be used with 'ctb' roles for the authors of such code. Alternatively, the 'Author' field should list these authors as contributors.
>
> Where copyrights are held by an entity other than the package authors, this should preferably be indicated via 'cph' roles in the 'Authors@R' field, or using a 'Copyright' field (if necessary referring to an inst/COPYRIGHTS file).
>
> Trademarks must be respected.

1.10 Licence

The package needs to have a CRAN or OSI accepted license. For more explanations around licensing, refer to the R packages book.

1.11 Testing

- All packages should pass `R CMD check`/`devtools::check()` on all major platforms.
- All packages should have a test suite that covers major functionality of the package. The tests should also cover the behavior of the package in case of errors.
- It is good practice to write unit tests for all functions, and all package code in general, ensuring key functionality is covered. Test coverage below 75% will likely require additional tests or explanation before being sent for review.
- We recommend using testthat for writing tests. Strive to write tests as you write each new function. This serves the obvious need to have proper testing for the package, but allows you to think about various ways in which a function can fail, and to *defensively* code against those. More information.
- Tests should be easy to understand. We suggest reading the blog post *"Why Good Developers Write Bad Unit Tests"* by Michael Lynch.
- Packages with Shiny apps should use a unit-testing framework such as `shinytest2` or `shinytest` to test that interactive interfaces behave as expected.
- For testing your functions creating plots, we suggest using vdiffr, an extension of the testthat package that relies on testthat snapshot tests.
- If your package interacts with web resources (web APIs and other sources of data on the web) you might find the HTTP testing in R book by Scott Chamberlain and Maëlle Salmon relevant. Packages helping with HTTP testing (corresponding HTTP clients):
  - httptest2 (httr2);
  - httptest (httr);
  - vcr (httr, crul);
  - webfakes (httr, httr2, crul, curl).
- testthat has a function `skip_on_cran()` that you can use to not run tests on CRAN. We recommend using this on all functions that are API calls since they are quite likely to fail on CRAN. These tests should still run on continuous integration. Note that from testthat 3.1.2 `skip_if_offline()` automatically calls `skip_on_cran()`. More info on CRAN preparedness for API wrappers.
- If your package interacts with a database you might find dittodb useful.
- Once you've set up continuous integration (CI), use your package's code coverage report (cf this section of our book) to identify untested lines, and to add further tests.
- Even if you use continuous integration, we recommend that you run tests locally prior to submitting your package (you might need to set `Sys.setenv(NOT_CRAN="true")`).

1.12 Examples

- Include extensive examples in the documentation. In addition to demonstrating how to use the package, these can act as an easy way to test package functionality before there are proper tests. However, keep in mind we require tests in contributed packages.
- You can run examples with `devtools::run_examples()`. Note that when you run R CMD CHECK or equivalent (e.g., `devtools::check()`) your examples that are not wrapped in `\dontrun{}` or `\donttest{}` are run. Refer to the summary table in roxygen2 docs.
- To safe-guard examples (e.g. requiring authentication) to be run on CRAN you need to use `\dontrun{}`. However, for a first submission CRAN won't let you have all examples escaped so. In this case you might add some small toy examples, or wrap example code in `try()`. Also refer to the `@exampleIf` tag present, at the time of writing, in roxygen2 development version.
- In addition to running examples locally on your own computer, we strongly advise that you run examples on one of the continuous integration systems. Again, examples that are not wrapped in `\dontrun{}` or `\donttest{}` will be run, but for those that are you can configure your continuous integration builds to run them via R CMD check arguments `--run-dontrun` and/or `--run-donttest`.

1.13 Package dependencies

- Consider the trade-offs involved in relying on a package as a dependency. On one hand, using dependencies reduces coding effort, and can build on useful functionality developed by others, especially if the dependency performs complex tasks, is high-performance, and/or is well vetted and tested. On the other hand, having many dependencies places a burden on the maintainer to keep up with changes in those packages, at risk to your package's long-term sustainability. It also increases installation time and size, primarily a consideration on your and others' development cycle, and in automated build systems. "Heavy" packages - those with many dependencies themselves, and those with large amounts of compiled code - increase this cost. Here are some approaches to reducing dependencies:
  - Small, simple functions from a dependency package may be better copied into your own package if the dependency if you are using only a few functions in an otherwise large or heavy dependency. (See *Authorship* section above for how to acknowledge original authors of copied code.) On the other hand, complex functions with many edge cases (e.g. parsers) require considerable testing and vetting.
    * An common example of this is in returning tidyverse-style "tibbles" from package functions that provide data. One can avoid the modestly heavy **tibble** package dependency by returning a tibble created by modifying a data frame like so:

```
class(df) <- c("tbl_df", "tbl", "data.frame")
```

(Note that this approach is not universally endorsed.)

- Ensure that you are using the package where the function is defined, rather than one where it is re-exported. For instance many functions in **devtools** can be found in smaller specialty packages such as **sessioninfo**. The %>% function should be imported from **magrittr**, where it is defined, rather than the heavier **dplyr**, which re-exports it.
- Some dependencies are preferred because they provide easier to interpret function names and syntax than base R solutions. If this is the primary reason for using a function in a heavy dependency, consider wrapping the base R approach in a nicely-named internal function in your package. See e.g. the rlang R script providing functions with a syntax similar to purrr functions.
- If dependencies have overlapping functionality, see if you can rely on only one.
- More dependency-management tips can be found in the chapter "Dependencies: Mindset and Background" of the R packages book and in a post by Scott Chamberlain.

- Use `Imports` instead of `Depends` for packages providing functions from other packages. Make sure to list packages used for testing (`testthat`), and documentation (`knitr`, roxygen2) in your `Suggests` section of package dependencies (if you use `usethis` for adding testing infrastructure via `usethis::use_testthat()` or a vignette via usethis::use_vignette(), the necessary packages will be added to `DESCRIPTION`). If you use any package in the examples or tests of your package, make sure to list it in `Suggests`, if not already listed in `Imports`.
- If your (not Bioconductor) package depends on Bioconductor packages, make sure the installation instructions in the README and vignette are clear enough even for an user who is not familiar with the Bioconductor release cycle.
  - Should the user use `BiocManager` (recommended)? Document this.
  - Is the automatic installation of Bioconductor packages by `install.packages()` enough? In that case, mention that the user needs to run `setRepositories()` if they haven't set the necessary Bioconductor repositories yet.
  - If your package depends on Bioconductor after a certain version, mention it in DESCRIPTION and in the installation instructions.
- Specifying minimum dependencies (e.g. `glue (>= 1.3.0)` instead of just `glue`) should be a conscious choice. If you know for a fact that your package will break below a certain dependency version, specify it explicitly. But if you don't, then no need to specify a minimum dependency. In that case when a user reports a bug which is explicitly related to an older version of a dependency then address it then. An example of bad practice would be for a developer to consider the versions of their current state of dependencies to be the minimal version. That

would needlessly force everyone to upgrade (causing issues with other packages) when there is no good reason behind that version choice.

- For most cases where you must expose functions from dependencies to the user, you should import and re-export those individual functions rather than listing them in the `Depends` fields. For instance, if functions in your package produce `raster` objects, you might re-export only printing and plotting functions from the **raster** package.
- If your package uses a *system* dependency, you should
  - Indicate it in DESCRIPTION;
  - Check that it is listed by `sysreqsdb` to allow automatic tools to install it, and submit a contribution if not;
  - Check for it in a `configure` script (example) and give a helpful error message if it cannot be found (example). `configure` scripts can be challenging as they often require hacky solutions to make diverse system dependencies work across systems. Use examples (more here) as a starting point but note that it is common to encounter bugs and edge cases and often violate CRAN policies. Do not hesitate to ask for help on our forum.

1.14 Recommended scaffolding

- For HTTP requests we recommend using httr2, httr, curl, or crul over RCurl. If you like low level clients for HTTP, curl is best, whereas httr2, httr and crul are better for higher level access.
- For parsing JSON, use jsonlite instead of rjson or RJSONIO.
- For parsing, creating, and manipulating XML, we strongly recommend xml2 for most cases. You can refer to Daniel Nüst's notes about migration from XML to xml2.
- For spatial data, the sp package should be considered deprecated in favor of sf, and the packages rgdal, rgdal, and rgdal will be retired by the end of 2023. We recommend use of the spatial suites developed by the r-spatial and rspatial communities. See this GitHub issue for relevant discussions.

1.15 Version Control

- Your package source files have to be under version control, more specifically tracked with Git. You might find the gert package relevant, as well as some of usethis Git/GitHub related functionality; you can however use git as you want.
- Make sure to list "scrap" such as `.DS_Store` files in .gitignore. You might find the `usethis::git_vaccinate()` function, and the gitignore package relevant.

- A later section of this book contains some git workflow tips.

1.16 Miscellaneous CRAN gotchas

This is a collection of CRAN gotchas that are worth avoiding at the outset.

- Make sure your package title is in Title Case.
- Do not put a period on the end of your title.
- Do not put 'in R' or 'with R' in your title as this is obvious from packages hosted on CRAN. If you would like this information to be displayed on your website nonetheless, check the `pkgdown` documentation to learn how to override this.
- Avoid starting the description with the package name or "This package …".
- Make sure you include links to websites if you wrap a web API, scrape data from a site, etc. in the `Description` field of your `DESCRIPTION` file. URLs should be enclosed in angle brackets, e.g. `<https://www.r-project.org>`.
- In both the `Title` and `Description` fields, the names of packages or other external software must be quoted using single quotes (e.g., *'Rcpp' Integration for the 'Armadillo' Templated Linear Algebra Library*).
- Avoid long running tests and examples. Consider `testthat::skip_on_cran` in tests to skip things that take a long time but still test them locally and on continuous integration.
- Include top-level files such as `paper.md`, continuous integration configuration files, in your `.Rbuildignore` file.

For further gotchas, refer to the collaborative list maintained by ThinkR, "Prepare for CRAN".

1.16.1 CRAN checks

Once your package is on CRAN, it will be regularly checked on different platforms. Failures of such checks, when not false positives, can lead to the CRAN team's reaching out. You can monitor the state of the CRAN checks via

- the `foghorn` package.
- the CRAN checks badges.

1.17 Bioconductor gotchas

If you intend your package to be submitted to, or if your package is on, Bioconductor, refer to Bioconductor packaging guidelines and the updated developer book.

1.18 Further guidance

- If you are submitting a package to rOpenSci via the software-review repo, you can direct further questions to the rOpenSci team in the issue tracker, or in our discussion forum.
- Read the authors guide.
- Read, incorporate, and act on advice from the *Collaboration Guide* chapter.

1.18.1 Learning about package development

1.18.1.1 Books

- Hadley Wickham and Jenny Bryan's *R packages* book is an excellent, readable resource on package development which is available for free online (and can be bought in print).
- Writing R Extensions is the canonical, usually most up-to-date, reference for creating R packages.
- *Mastering Software Development in R* by Roger D. Peng, Sean Kross, and Brooke Anderson.
- *Advanced R* by Hadley Wickham
- *Tidyverse style guide*
- *Tidyverse design guide* (WIP) and the accompanying newsletter.

1.18.1.2 Tutorials

- Your first R package in 1 hour by Shannon Pileggi.
- this workflow description by Emil Hvitfeldt.
- This pictorial by Matthew J Denny.

1.18.1.3 Blogs

- R-hub blog.
- Some posts of the rOpenSci blog e.g. "How to precompute package vignettes or pkgdown articles".
- Package Development Corner section of rOpenSci newsletter.
- Some posts of the tidyverse blog e.g. "Upgrading to testthat edition 3".

1.18.1.4 MOOCs

There is a Coursera specialization corresponding to the book by Roger Peng, Sean Kross and Brooke Anderson, with a course specifically about R packages.

2 Continuous Integration Best Practices

This chapter summarizes our guidelines about continuous integration after explaining what continuous integration is.

Along with the previous chapter, it forms our guidelines for Software Peer Review.

2.1 What is continuous integration (CI)?

Continuous integration automatically runs tests on software. In the case of rOpenSci, CI practically means that a set of tests will be automatically run via GitHub, every time that you make a commit or pull request to GitHub.

CI automates the running of general package checks such as `R CMD check`, see testing. It is possible to set up CI before your tests are written, then CI will run the tests as you commit them to the repository.

2.2 Why use continuous integration (CI)?

All rOpenSci packages must use one form of continuous integration. This ensures that all commits, pull requests and new branches are run through `R CMD check`. The results of all tests are displayed on the pull request page on GitHub, providing another layer of information about problems and protection against breaking your package before merging changes. rOpenSci packages' continuous integration must also be linked to a code coverage service, indicating how many lines are covered by unit tests.

Both test status and code coverage should be reported via badges in your package README.

R packages should have CI for all operating systems (Linux, Mac OSX, Windows) when they contain:

- Compiled code
- Java dependencies
- Dependencies on other languages

- Packages with system calls
- Text munging such as getting people's names (in order to find encoding issues).
- Anything with file system / path calls

In case of any doubt regarding the applicability of these criteria to your package, it's better to add CI for all operating systems. Most CI services standards setups for R packages allow this with not much hassle.

2.3 Which continuous integration service(s)?

There are a number of continuous integration services, including standalone services (CircleCI, AppVeyor), and others integrated into code hosting or related services (GitHub Actions, GitLab, AWS Code Pipeline). Different services support different operating system configurations.

GitHub Actions is a convenient option for many R developers who already use GitHub as it is integrated into the platform and supports all needed operating Systems. There are actions supported for the R ecosystem, as well and first-class support in the {usethis} package. All packages submitted to rOpenSci for peer review are checked by our own `pkgcheck` system, described further in the Guide for Authors. These checks are also provided as a GitHub Action in the `ropensci-review-tools/pkgcheck-action` repository. Packages authors are encouraged to use that action to confirm prior to submission that a package passes all of our checks. See our blog post for more information.

usethis supports CI setup for other systems, though these functions are soft-deprecated. rOpenSci also supports the circle package, which aids in setting up CircleCI pipelines, and the tic package for building more complicated CI pipelines.

2.3.0.1 Testing using different versions of R

We require that rOpenSci packages are tested against the latest, previous and development versions of R to ensure both backwards and forwards compatibility with base R.

Details of how to run tests/checks using different versions of R locally can be found in the R-hub vignette on running Local Linux checks with Docker.

You can fine tune the deployment of tests with each versions by using a testing matrix.

If you develop a package depending on or intended for Bioconductor, you might find biocthis relevant.

2.3.0.2 Minimizing build times on CI

You can use these tips to minimize build time on CI:

- Cache installation of packages. The default r-lib/actions workflows do this.

2.3.0.3 System dependencies

You might find Hugo Gruson's post System Dependencies in R Packages & Automatic Testing useful.

2.3.1 Travis CI (Linux and Mac OSX)

We recommend moving away from Travis.

2.3.2 AppVeyor CI (Windows)

For continuous integration on Windows, see R + AppVeyor. Set it up using `usethis::use_appveyor()`.

Here are tips to minimize AppVeyor build time:

- Cache installation of packages. Example in a config file. It'll already be in the config file if you set AppVeyor CI up using `usethis::use_appveyor()`.
- Enable rolling builds.

We no longer transfer AppVeyor projects to ropensci AppVeyor account so after transfer of your repo to rOpenSci's "ropensci" GitHub organization the badge will be `[![AppVeyor Build Status](https://ci.appveyor.com/api/projects/status/github/ropensci/pkgname?branch=master&sv`

2.3.3 Circle CI (Linux and Mac OSX)

Circle CI is used, for example, by rOpenSci package `bomrang` as continuous integration service.

2.4 Test coverage

Continuous integration should also include reporting of test coverage via a testing service such as Codecov or Coveralls.

We recommend using Codecov. To activate Codecov for your repo, run `usethis::use_github_action("test-cover` to create a file `.github/workflows/test-coverage.yaml`. You also need to give Codecov access to your github repository, see Codecov quick start guide for how to set up access. Then add a Codecov status badge to the top of your README.md, see Codecov status badges.

Currently, Codecov has access to all ropensci github repositories by default. When your repository is accepted and transferred to ropensci, Codecov access should transfer automatically. You will need to update the URL of the badge to point to the rOpenSci-hosted repository.

For more details, see the README for the **covr** package for instructions, as well as `usethis::use_coverage()` and `usethis::use_github_action()`.

If you run coverage on several CI services the results will be merged.

2.5 Even more CI: OpenCPU

After transfer to rOpenSci's "ropensci" GitHub organization, each push to the repo will be built on OpenCPU and the person committing will receive a notification email. This is an additional CI service for package authors that allows for R functions in packages to be called remotely via https://ropensci.ocpu.io/ using the opencpu API. For more details about this service, consult the OpenCPU help page that also indicates where to ask questions.

2.6 Even more CI: rOpenSci docs

After transfer to rOpenSci's "ropensci" GitHub organization, a pkgdown website will be built for your package after each push to the GitHub repo. You can find the status of these builds at `https://ropensci.r-universe.dev/ui#packages` and in the commit status. The website build will use your pkgdown config file if you have one, except for the styling that will use the `rotemplate` package.

Please report bugs, questions and feature requests about the central builds and about the template at https://github.com/ropensci-org/rotemplate/.

3 Package Development Security Best Practices

This work-in-progress chapter includes guidance about managing secrets in packages and links for further reading.

3.1 Miscellaneous

We recommend the article Ten quick tips for staying safe online by Danielle Smalls and Greg Wilson.

3.2 GitHub access security

- We recommend you secure your GitHub account with two-factor (authentication) 2FA. It is *compulsory* for all ropensci GitHub organization members and outside collaborators so make sure to enable it before your package is approved.
- We also recommend you regularly check who has access to your package repository, and that you prune any unused access (such as from former collaborators).

3.3 https

- If the web service your package wraps has either https or http, opt for https.

3.4 Secrets in packages

This section contains guidance for when you develop a package interacting with a web resource requiring credentials (API keys, tokens, etc.). Also refer to the `httr` vignette about sharing secrets.

3.4.1 Secrets in packages and user protection

Say your package needs an API key for making requests on behalf of users of your package.

- In your package documentation, guide the user so the API key doesn't end up in the .Rhistory/script of users of your package.
  - Encourage the use of environment variables to store the API key (or even remove the possibility to pass it as an argument to the functions?). You could link to this intro to startup files and `usethis::edit_r_environ()`.
  - Or your package could depend on, or encourage the use of, `keyring` to help user store variables in the specific OS' credential stores (more secure than .Renviron): i.e. you'd create a function for setting the key, and have another one for retrieving the key; or the user would write `Sys.setenv(SUPERSECRETKEY = keyring::key_get("myservice"))` at the beginning of their script.
  - Do not print the API key even in verbose mode in any message, warning, error.
- In the GitHub issue template, it should be stated not to share any credentials. If an user of your package accidentally shares credentials in an issue, make sure they're aware of that so they can revoke the key (i.e. ask them explicitly in an answer whether they realized they shared their key).

3.4.2 Secrets in packages and development

You'll need to protect your secrets as you protect secrets of users, but there's more to take into account and keep in mind.

3.4.2.1 Secrets and recorded requests in tests

If you use `vcr` or `httptest` in tests for caching API responses, you need to make sure the recorded requests / fixtures do not contain secrets. Refer to `vcr` security guidance and `httptest` guidance "Redacting and Modifying Recorded Requests", and inspect your recorded requests / fixtures before committing them the first time to be sure you got the setup right.

`vcr` being an rOpenSci package, you can post any question you might have to rOpenSci forum.

3.4.2.2 Share secrets with CI services

Now, you might need to share secrets with continuous integration services.

You could store API keys as environment variables / secrets, referring to the docs of the CI service.

For more details and workflow advice, refer to the gargle article "Managing tokens securely" and the security chapter of the HTTP testing in R book.

Document the steps you made in CONTRIBUTING.md so you, or say a new maintainer, can remember how to do that next time.

3.4.2.3 Secrets and collaborations

What about pull requests from external contributors? On GitHub for instance, secrets are only available for GitHub Actions for pull requests started from the repository itself, not from fork. Tests using your secrets will fail unless you use some sort of mocked/cached response, so you might want to skip them depending on the context. For instance, in your CI account you could create an environment variable called `THIS_IS_ME` and then skip tests based on the presence of this variable. This obviously means the PR checks by the CI are not exhaustive, so you'll need to check out the PR locally to run all tests.

Document the behavior of your package for external PRs in CONTRIBUTING.md for the sake of people making PRs and of people reviewing them (you in a few weeks, and other authors of the package).

3.4.3 Secrets and CRAN

On CRAN, skip any tests (`skip_on_cran()`) and examples (`dontrun`) requiring credentials.

Precompute vignettes requiring credentials.

3.5 Further reading

Useful security resources:

- rOpenSci community call "Security for R" (recording, slides, etc. see in particular the list of resources);
- the security-related projects of unconf18;
- `gargle` article "Managing tokens securely"

Parte II

Software Peer Review of Packages

4 Software Peer Review, Why? What?

This chapter contains a general intro to our software peer review system for packages, reasons to submit a package, reasons to volunteer as a reviewer, why our reviews are open, and acknowledgements of actors of the system.

Our system has recently been expanded to statistical software peer-review.

If you use our standards/checklists etc. when reviewing software elsewhere, do tell the recipients (e.g. journal editors, students, internal code review) that they came from rOpenSci, and tell us in our public forum, or privately by email.

4.1 What is rOpenSci Software Peer Review?

rOpenSci's suite of packages is partly contributed by staff members and partly contributed by community members, which means the suite stems from a great diversity of skills and experience of developers. How to ensure quality for the whole set? That's where software peer review comes into play: packages contributed by the community undergo a transparent, constructive, non adversarial and open review process. For that process relying mostly on volunteer work, associate editors manage the incoming flow and ensure progress of submissions; authors create, submit and improve their package; reviewers, two per submission, examine the software code and user experience. This blog post written by rOpenSci editors is a good introduction to rOpenSci software peer review Other blog posts about review itself and reviewed packages can be find via the "software-peer-review" tag on rOpenSci blog.

You can recognize rOpenSci packages that have been peer-reviewed via a green "peer-reviewed" badge in their README, linking to their reviews (cf this example); and via a blue comment icon near their description on rOpenSci packages page, also linking to the reviews.

Technically, we make the most of GitHub infrastructure: each package review process is an issue in the ropensci/software-review GitHub repository. For instance, click here to read the review thread of the `ropenaq` package: the process is an ongoing conversation until acceptance of the package, with two external reviews as important milestones. Furthermore, we use GitHub features such as the use of issue templates (as submission templates), and labelling which we use to track progress of submissions (from editor checks to approval).

4.2 Why submit your package to rOpenSci?

- First, and foremost, we hope you submit your package for review **because you value the feedback**. We aim to provide useful feedback to package authors and for our review process to be open, non-adversarial, and focused on improving software quality.
- Once aboard, your package will continue to receive **support from rOpenSci members**. You'll retain ownership and control of your package, but we can help with ongoing maintenance issues such as those associated with updates to R and dependencies and CRAN policies.
- rOpenSci will **promote your package** through our webpage, blog, and social media (like Mastodon and LinkedIn). Packages in our suite also get a documentation website that is automatically built and deployed after each push.
- rOpenSci **packages can be cross-listed** with other repositories such as CRAN and BioConductor.
- rOpenSci packages that are in scope for the Journal of Open-Source Software and add the necessary accompanying short paper, would, at the discretion of JOSS editors, benefit from a fast-tracked review process.
- If you write one, rOpenSci will **promote gitbooks related to your package**: the source of such books can be transferred to the `ropensci-books` GitHub organisation for books to be listed at books.ropensci.org.

4.3 Why review packages for rOpenSci?

- As in any peer-review process, we hope you choose to review **to give back to the rOpenSci and scientific communities.** Our mission to expand access to scientific data and promote a culture of reproducible research is only possible through the volunteer efforts of community members like you.
- Review is a two-way conversation. By reviewing packages, you'll have the chance to **continue to learn development practices from authors and other reviewers**.
- The open nature of our review process allows you to **network and meet colleagues and collaborators** through the review process. Our community is friendly and filled with supportive members expert in R development and many other areas of science and scientific computing.
- To volunteer to be one of our reviewers, fill out this short form providing your contact information and areas of expertise. We are always looking for more reviewers with both general package-writing experience and domain expertise in the fields where packages are used.

4.4 Why are reviews open?

Our reviewing threads are public. Authors, reviewers, and editors all know each other's identities. The broader community can view or even participate in the conversation as it happens. This pro-

vides an incentive to be thorough and provide non-adversarial, constructive reviews. Both authors and reviewers report that they enjoy and learn more from this open and direct exchange. It also has the benefit of building a community. Participants have the opportunity to meaningfully network with new peers, and new collaborations have emerged via ideas spawned during the review process.

We are aware that open systems can have drawbacks. For instance, in traditional academic review, double-blind peer review can increase representation of female authors, suggesting bias in non-blind reviews. It is also possible reviewers are less critical in open review. However, we posit that the openness of the review conversation provides a check on review quality and bias; it's harder to inject unsupported or subjective comments in public and without the cover of anonymity. Ultimately, we believe that having direct and public communication between authors and reviewers improves quality and fairness of reviews.

Furthermore, authors and reviewers have the ability to contact privately the editors if they have any doubt or question.

4.5 How will users know a package has been reviewed?

- Your package README will feature a peer-review badge linking to the software review thread.
- Your package will get a `docs.ropensci.org` docs website that you can link from DESCRIPTION.
- Your package repo will be transferred to the rOpenSci organization.
- If reviewers agree to be listed in DESCRIPTION, their metadata will mention the review.

4.6 Editors and reviewers

4.6.1 Associate editors

rOpenSci's Software Peer Review process is run by:

- Noam Ross, EcoHealth Alliance
- Karthik Ram, rOpenSci
- Maëlle Salmon, rOpenSci
- Mark Padgham, rOpenSci
- Anna Krystalli, University of Sheffield RSE
- Melina Vidoni, RMIT University (School of Science)
- Mauro Lepore, 2 Degrees Investing Initiative
- Laura DeCicco, USGS
- Julia Gustavsen, Agroscope
- Emily Riederer, Capital One

- Adam Sparks, Department of Primary Industries and Regional Development
- Jeff Hollister, US Environmental Protection Agency

4.6.2 Reviewers

We are grateful to the following individuals who have offered up their time and expertise to review packages submitted to rOpenSci.

Em Markowitz (NOAA) · Sam Albers · Toph Allen · Kaique dos S. Alves · Alison Appling · Denisse Fierro Arcos · Zebulun Arendsee · Taylor Arnold · Al-Ahmadgaid B. Asaad · Dean Attali · Mara Averick · Suzan Baert · James Balamuta · Vikram Baliga · David Bapst · Joëlle Barido-Sottani · Allison Barner · Cale Basaraba · John Baumgartner · Marcus Beck · Gabriel Becker · Jason Becker · Salvador Jesus Fernandez Bejarano · Dom Bennett · Ken Benoit · Aaron Berdanier · Fred Boehm · Carl Boettiger · Will Bolton · Ben Bond-Lamberty · Anne-Sophie Bonnet-Lebrun · Alison Boyer · Abby Bratt · François Briatte · Eric Brown · Julien Brun · Jenny Bryan · Lukas Burk · Lorenzo Busetto · Maria Paula Caldas · Mario Gavidia Calderón · Carlos Cámara-Menoyo · Brad Cannell · Joaquin Cavieres · Kevin Cazelles · Cathy Chamberlin · Jennifer Chang · Pierre Chausse · Jorge Cimentada · Nicholas Clark · Chase Clark · Jon Clayden · Dena Jane Clink · Will Cornwell · Nic Crane · Enrico Crema · Verónica Cruz-Alonso · Ildiko Czeller · Tad Dallas · Kauê de Sousa · Christophe Dervieux · Amanda Dobbyn · Jasmine Dumas · Dewey Dunnington · Remko Duursma · Mark Edmondson · Paul Egeler · Evan Eskew · Harry Eslick · Alexander Fischer · Kim Fitter · Robert M Flight · Sydney Foks · Stephen Formel · Zachary Stephen Longiaru Foster · Auriel Fournier · Kaija Gahm · Zach Gajewski · Carl Ganz · Duncan Garmonsway · Jan Laurens Geffert · Sharla Gelfand · Monica Gerber · Duncan Gillespie · David Gohel · A. Cagri gokcek · Guadalupe Gonzalez · Rohit Goswami · Laura Graham · Charles Gray · Matthias Grenié · Corinna Gries · Hugo Gruson · Ernest Guevarra · W Kyle Hamilton · Ivan Hanigan · Jeffrey Hanson · Liz Hare · Jon Harmon · Rayna Harris · Ted Hart · Nujcharee Haswell · Verena Haunschmid · Stephanie Hazlitt · Andrew Heiss · Max Held · Anna Hepworth · Bea Hernandez · Jim Hester · Peter Hickey · Roel Hogervorst · Kelly Hondula · Allison Horst · Sean Hughes · James Hunter · Brandon Hurr · Ger Inberg · Christopher Jackson · Najko Jahn · Tamora D James · Veronica Jimenez-Jacinto · Mike Johnson · Will Jones · Max Joseph · Megha Joshi · Krunoslav Juraic · Soumya Kalra · Zhian N. Kamvar · Michael Kane · Andee Kaplan · Tinula Kariyawasam · Hazel Kavılı · Jonathan Keane · Christopher T. Kenny · Os Keyes · Eunseop Kim · Aaron A. King · Michael Koontz · Bianca Kramer · Will Landau · Sam Lapp · Erin LeDell · Thomas Leeper · Sam Levin · Lisa Levinson · Stephanie Locke · Marion Louveaux · Robin Lovelace · Julia Stewart Lowndes · Tim Lucas · Muralidhar, M.A. · Andrew MacDonald · Jesse Maegan · Mike Mahoney · Tristan Mahr · Paula Andrea Martinez · Joao Martins · Ben Marwick · Claire Mason · Miles McBain · Lucy D'Agostino McGowan · Amelia McNamara · Elaine McVey · Bryce Mecum · Nolwenn Le Meur · François Michonneau · Mario Miguel · Helen Miller · Jessica Minnier · Priscilla Minotti · Nichole Monhait · Kelsey Montgomery · Ronny A. Hernández Mora · Natalia Morandeira · Ross Mounce · Athanasia Monika Mowinckel · Lincoln Mullen · Matt Mulvahill · Maria Victoria Munafó · David Neuzerling · Dillon Niederhut · Joel Nitta · Rory Nolan · Kari Norman · Jakub Nowosad · Matt Nunes · Daniel Nüst · Lauren O'Brien · Joseph O'Brien · Paul Oldham · Samantha Oliver · Dan Olner · Jeroen Ooms · Victor Ordu · Luis Osorio · Philipp Ottolinger · Mark Padgham · Marina Papadopoulou · Edzer

Pebesma · Thomas Lin Pedersen · Antonio J. Pérez-Luque · Marcelo S. Perlin · Rafael Pilliard-Hellwig · Rodrigo Neto Pires · Lindsay Platt · Nicholas Potter · Joanne Potts · Josep Pueyo-Ros · Etienne Racine · Manuel Ramon · Nistara Randhawa · David Ranzolin · Quentin Read · Neal Richardson · tyler rinker · Emily Robinson · David Robinson · Alec Robitaille · Sam Rogers · Julia Romanowska · Xavier Rotllan-Puig · Bob Rudis · Edgar Ruiz · Kent Russel · Michael Sachs · Sheila Saia · Chitra M Saraswati · Alicia Schep · Klaus Schliep · Clemens Schmid · Patrick Schratz · Collin Schwantes · Marco Sciaini · Eric Scott · Heidi Seibold · Julia Silge · Peter Slaughter · Mike Smith · Tuija Sonkkila · Øystein Sørensen · Jemma Stachelek · Christine Stawitz · Irene Steves · Kelly Street · Matt Strimas-Mackey · Alex Stringer · Michael Sumner · Chung-Kai Sun · Sarah Supp · Jason Taylor · Filipe Teixeira · Andy Teucher · Jennifer Thompson · Joe Thorley · Nicholas Tierney · Tiffany Timbers · Tan Tran · Tim Trice · Utku Turk · Kyle Ueyama · Ted Underwood · Adithi R. Upadhya · Kevin Ushey · Josef Uyeda · Frans van Dunné · Mauricio Vargas · Remi Vergnon · Jake Wagner · Ben Ward · Elin Waring · Rachel Warnock · Leah Wasser · David Watkins · Lukas Weber · Marc Weber · Karissa Whiting · Stefan Widgren · Anna Willoughby · Saras Windecker · Luke Winslow · David Winter · Witold Wolski · Kara Woo · Marvin N. Wright · Jacob Wujciak-Jens · Bruna Wundervald · Lauren Yamane · Emily Zabor · Taras Zakharko · Hao Zhu · Chava Zibman · Naupaka Zimmerman · Jake Zwart · Felipe · santikka · kasselhingee · Bri · Flury · Vincent · eholmes · Pachá · Rich · Claudia · Jasmine · Zack · Lluís · becarioprecario · gaurav

We are also grateful to the following individuals who have served as guest editors.

Ana Laura Diedrichs · Hao Zhu

5 Software Peer Review policies

This chapter contains the policies of rOpenSci Software Peer Review.

In particular, you'll read our policies regarding software peer review itself: the review submission process including our conflict of interest policies, and the aims and scope of the Software Peer Review system. This chapter also features our policies regarding package ownership and maintenance.

Last but not least, you'll find the code of conduct of rOpenSci Software Peer Review.

5.1 Review process

- For a package to be considered for the rOpenSci suite, package authors must initiate a request on the ropensci/software-review repository.
- Packages are reviewed for quality, fit, documentation, clarity and the review process is quite similar to a manuscript review (see our packaging guide and reviewing guide for more details). Unlike a manuscript review, this process will be an ongoing conversation.
- Once all major issues and questions, and those addressable with reasonable effort, are resolved, the editor assigned to a package will make a decision (accept, hold, or reject). Rejections are usually done early (before the review process begins, see the aims and scope section), but in rare cases a package may also be not onboarded after review & revision. It is ultimately editor's decision on whether or not to reject the package based on how the reviews are addressed.
- Communication between authors, reviewers and editors will first and foremost take place on GitHub, although you can choose to contact the editor by email or Slack for some issues. When submitting a package, please make sure your GitHub notification settings make it unlikely you will miss a comment.
- The author can choose to have their submission put on hold (editor applies the holding label). The holding status will be revisited every 3 months, and after one year the issue will be closed.
- If the author hasn't requested a holding label, but is simply not responding, we should close the issue within one month after the last contact intent. This intent will include a comment tagging the author, but also an email using the email address listed in the DESCRIPTION of the package which is one of the rare cases where the editor will try to contact the author by email.

- If a submission is closed and the author wishes to re-submit, they'll have to start a new submission. If the package is still in scope, the author will have to respond to the initial reviews before the editor starts looking for new reviewers.

5.1.1 Publishing in other Venues

- We strongly suggest submitting your package for review *before* publishing on CRAN or submitting a software paper describing the package to a journal. Review feedback may result in major improvements and updates to your package, including renaming and breaking changes to functions. We do not consider previous publication on CRAN or in other venues sufficient reason to not adopt reviewer or editor recommendations.
- Do not submit your package for review while it or an associated manuscript is also under review at another venue, as this may result on conflicting requests for changes.

5.1.2 Conflict of interest for reviewers/editors

Following criteria are meant to be a guide for what constitutes a conflict of interest for an editor or reviewer. The potential editor or reviewer has a conflict of interest if:

- The potential reviewer/editor are from the same institution or institutional component (e.g., department) as any author with a major role.
- The potential reviewer/editor has been a collaborator or has had other professional relationships with at least one person on the package who has a major role within in the past three years.
- The potential reviewer/editor serves, or has served, as a member of the advisory board for the project under review.
- The potential reviewer/editor would receive a direct or indirect financial benefit if the package were accepted.
- The potential reviewer/editor has significantly contributed to a competitor project.
- There is also a lifetime COI for the family members, business partners, and thesis student/advisor or mentor.

In the case where none of the associate editors can serve as editor, an external guest editor will be recruited.

5.2 Aims and Scope

rOpenSci aims to support packages that enable reproducible research and managing the data lifecycle for scientists. Packages submitted to rOpenSci should fit into one or more of the categories outlined either below. Statistical software may also be submitted for peer review, for which we have

a separate set of guidelines and standards. The categories below are for general, and not statistical, software, while the remainder of this chapter applies to both kinds of software. If you are unsure whether your package fits into one of the general or statistical categories, please open an issue as a pre-submission inquiry (**Examples**).

As this is a living document, these categories may change through time and not all previously onboarded packages would be in-scope today. For instance, data visualization packages are no longer in-scope. While we strive to be consistent, we evaluate packages on a case-by-case basis and may make exceptions.

Note that not all rOpenSci projects and packages are in-scope or go through peer review. Projects developed by staff or at conferences may be experimental, exploratory, address core infrastructure priorities and thus not fall into these categories. Look for the peer-review badge - see below - to identify peer-reviewed packages in the rOpenSci repository.

Figura 5.1: example of a green peer-reviewed badge

5.2.1 Package categories

- **data retrieval**: Packages for accessing and downloading data from online sources with scientific applications. Our definition of scientific applications is broad, including data storage services, journals, and other remote servers, as many data sources may be of interest to researchers. However, retrieval packages should be focused on data *sources* / *topics*, rather than *services*. For example a general client for Amazon Web Services data storage would not be in-scope. (Examples: **rotl**, **gutenbergr**)
- **data extraction**: Packages that aid in retrieving data from unstructured sources such as text, images and PDFs, as well as parsing scientific data types and outputs from scientific equipment. Statistical/ML libraries for modeling or prediction are typically not included in this category, nor are code parsers. Trained models that act as utilities (e.g., for optical character recognition), may qualify. (Examples: **tabulizer** for extracting tables from PDF documents, **genbankr** for parsing files from GenBank, **treeio** for phylogentic reading in phylogentic tree files, **lightr** for parsing files from spectroscopic instruments))
- **data munging**: Packages for processing data from formats above. This area does not include broad data manipulations tools such as **reshape2** or **tidyr**, or tools for extracting data from R code itself. Rather, it focuses on tools for handling data in specific scientific formats generated from scientific workflows or exported from scientific instruments. (Examples: **plateR** for reading in data structured as plate maps for scientific instruments, or **phonfieldwork** for processing annotated audio files for phonics research)

- **data deposition**: Packages that support deposition of data into research repositories, including data formatting and metadata generation. (Example: **EML**)
- **data validation and testing**: Tools that enable automated validation and checking of data quality and completeness as part of scientific workflows. (Example: **assertr**)
- **workflow automation**: Tools that automate and link together workflows, such as build systems and tools to manage continuous integration. Does not include general tools for literate programming. (e.g., R markdown extensions not under the previous topics). (Example: **drake**)
- **version control**: Tools that facilitate the use of version control in scientific workflows. Note that this does not include all tools that interact with online version control services (e.g., GitHub), unless they fit into another category. (Example: **git2rdata**)
- **citation management and bibliometrics**: Tools that facilitate managing references, such as for writing manuscripts, creating CVs or otherwise attributing scientific contributions, or accessing, manipulating or otherwise working with bibliometric data. (Example: **RefManageR**)
- **scientific software wrappers**: Packages that wrap non-R utility programs used for scientific research. These programs must be specific to research fields, not general computing utilities. Wrappers must be non-trivial, in that there must be significant added value above simple `system()` calls or bindings, whether in parsing inputs and outputs, data handling, etc. Improved installation process, or extension of compatibility to more platforms, may constitute added value if installation is complex. This does not include wrappers of other R packages or C/C++ libraries that can be included in R packages. It also does not include packages that are clients for web APIs, which must fall into one of the other categories. We strongly encourage wrapping open-source and open-licensed utilities - exceptions will be evaluated case-by-case, considering whether open-source options exist. (Examples: **babette**, **nlrx**)
- **field and laboratory reproducibility tools**: Packages that improve reproducibility of real-world workflows through standardization and automation of field and lab protocols, such as sample tracking and tagging, form and data sheet generation, interfacing with laboratory equipment or information systems, and executing experimental designs. (Example: **baRcodeR**)
- **database software bindings**: Bindings and wrappers for generic database APIs (Example: **rrlite**)

In addition, we have some *specialty topics* with a slightly broader scope.

- **geospatial data**: We accept packages focused on accessing geospatial data, manipulating geospatial data, and converting between geospatial data formats. (Examples: **osmplotr**, **tidync**).

- **translation**: As part of our work in multilingual publishing, we have a special interest in packages that facilitate the translation and publication of scientific and programming resources into multiple (human) languages so they are accessible to larger and more diverse audiences. These could include interfaces to automated translation programs, frameworks for managing documentation in multiple languages, or programs accessing specialized linguistic resources. This is a new and experimental scope, so please open a pre-submission inquiry if you are interested in submitting a package in this category.

5.2.2 Other scope considerations

Packages should be *general* in the sense that they should solve a problem as broadly as possible while maintaining a coherent user interface and code base. For instance, if several data sources use an identical API, we prefer a package that provides access to all the data sources, rather than just one.

Packages that include interactive tools to facilitate researcher workflows (e.g., shiny apps) must have a mechanism to make the interactive workflow reproducible, such as code generation or a scriptable API.

For packages that are not in the scope of rOpenSci, we encourage submitting them to CRAN, BioConductor, as well as other R package development initiatives (e.g., cloudyr), and software journals such as JOSS, JSS, or the R journal, depending on the current scopes of those journals.

Note that the packages developed internally by rOpenSci, through our events or through collaborations are not all in-scope for our Software Peer Review process.

5.2.3 Package overlap

rOpenSci encourages competition among packages, forking and re-implementation as they improve options of users overall. However, as we want packages in the rOpenSci suite to be our top recommendations for the tasks they perform, we aim to avoid duplication of functionality of existing R packages in any repo without significant improvements. An R package that replicates the functionality of an existing R package may be considered for inclusion in the rOpenSci suite if it significantly improves on alternatives in any repository (RO, CRAN, BioC) by being:

- More open in licensing or development practices
- Broader in functionality (e.g., providing access to more data sets, providing a greater suite of functions), but not only by duplicating additional packages
- Better in usability and performance
- Actively maintained while alternatives are poorly or no longer actively maintained

These factors should be considered *as a whole* to determine if the package is a significant improvement. A new package would not meet this standard only by following our package guidelines while others do not, unless this leads to a significant difference in the areas above.

We recommend that packages highlight differences from and improvements over overlapping packages in their README and/or vignettes.

We encourage developers whose packages are not accepted due to overlap to still consider submittal to other repositories or journals.

5.3 Package ownership and maintenance

5.3.1 Role of the rOpenSci team

Authors of contributed packages essentially maintain the same ownership they had prior to their package joining the rOpenSci suite. Package authors will continue to maintain and develop their software after acceptance into rOpenSci. Unless explicitly added as collaborators, the rOpenSci team will not interfere much with day to day operations. However, this team may intervene with critical bug fixes, or address urgent issues if package authors do not respond in a timely manner (see the section about maintainer responsiveness).

5.3.2 Maintainer responsiveness

If package maintainers do not respond in a timely manner to requests for package fixes from CRAN or from us, we will remind the maintainer a number of times, but after 3 months (or shorter time frame, depending on how critical the fix is) we will make the changes ourselves.

The above is a bit vague, so the following are a few areas of consideration.

- Examples where we'd want to move quickly:
  - Package `foo` is imported by one or more packages on CRAN, and `foo` is broken, and thus would break its reverse dependencies.
  - Package `bar` may not have reverse dependencies on CRAN, but is widely used, thus quickly fixing problems is of greater importance.
- Examples where we can wait longer:
  - Package `hello` is not on CRAN, or on CRAN, but has no reverse dependencies.
  - Package `world` needs some fixes. The maintainer has responded but is simply very busy with a new job, or other reason, and will attend to soon.

We urge package maintainers to make sure they are receiving GitHub notifications, as well as making sure emails from rOpenSci staff and CRAN maintainers are not going to their spam box. Authors of onboarded packages will be invited to the rOpenSci Slack to chat with the rOpenSci team and the greater rOpenSci community. Anyone can also discuss with the rOpenSci community on the rOpenSci discussion forum.

Should authors abandon the maintenance of an actively used package in our suite, we will consider petitioning CRAN to transfer package maintainer status to rOpenSci.

5.3.3 Quality commitment

rOpenSci strives to develop and promote high quality research software. To ensure that your software meets our criteria, we review all of our submissions as part of the Software Peer Review process, and even after acceptance will continue to step in with improvements and bug fixes.

Despite our best efforts to support contributed software, errors are the responsibility of individual maintainers. Buggy, unmaintained software may be removed from our suite at any time.

5.3.4 Package removal

In the unlikely scenario that a contributor of a package requests removal of their package from the suite, we retain the right to maintain a version of the package in our suite for archival purposes.

5.4 Ethics, Data Privacy and Human Subjects Research

rOpenSci packages and other tools are used for a variety of purposes, but our focus is on tools for research. We expect that tools will enable ethical use by research practitioners, who are obligated to adhere to ethical codes such Declaration of Helsinki and The Belmont Report. Researchers bear responsibility for their use of software, but software developers must consider the ethical use of their products, and developers themselves adhere to ethical codes for computer professionals such as those expressed by IEEE and ACM. rOpenSci contributors often play both the role of both researcher and developer.

We ask that software developers place themselves in researchers' role and consider the requirements of an ethical workflow using authors' software. Given the variation and degree of flux of ethical approaches for Internet-based analyses, judgement calls rather than recipes are required. The Ethical Guidelines of The Association of Internet Researchers provides a robust framework and we encourage authors, editors, and reviewers to use this in evaluating their work. In general, adherence to legal or regulatory minimum requirements may not be sufficient, though these (e.g., GDPR), may be relevant. Package authors should direct users to relevant resources for the ethical use of the software.

Some packages, due to the nature of data they handle, may be determined by editors to require enhanced scrutiny. For these, editors may require additional (or reduced) functionality, and robust documentation, defaults, and warnings to direct users to relevant ethical practices. The following topics may merit enhanced scrutiny:

- ***Vulnerable populations***: Authors of packages and workflows that deal with information related to vulnerable populations bear responsibility to protect them from likely harms.
- ***Personally identifiable or sensitive data***: The release of personally identifiable or sensitive data is potentially harmful. This includes "reasonably re-identifiable" data - which a motivated individual could trace back to the owner or creator even if the data are anonymized. This includes both cases where identifiers (e.g., name, date of birth) are available as part of data, and also if unique pseudonyms/screen names are linked with full-text posts, through which one can link back individuals through cross-reference with other data sets.

While the best response to ethical concerns will be context-specific, these general guidelines should be followed by packages where the challenges above arise:

- Packages should adhere to data source's terms of use, as expressed in website Terms and Conditions, "robots.txt" files, privacy policies, and other relevant restrictions, and link to them prominently in package documentation. Packages should provide or document functionality to adhere to such restrictions (e.g., scrape from only allowed endpoints, use appropriate rate limiting in code, examples, or vignettes). Note that while Terms and Conditions, Privacy Policies, etc., may not provide sufficient bounds on ethical usage, they can provide an outer bound.
- A key tool in addressing the risks posed in studying vulnerable populations or using personally identifiable data is ***informed consent***. Package authors should support users' acquisition of informed consent when relevant. This may include providing links to data source's preferred method of acquiring consent, contact information of data providers (e.g. forum moderators), documentation of informed consent protocols, or getting pre-approval for general uses of a package.

  Note that consent is not implicitly granted just because data are accessible. Accessible data are not necessarily public, as different persons and contexts have different normative expectations of privacy (see work by Social Data Lab).
- Packages accessing personally identifiable information should take special care to follow [security best practices][Package Development Security Best Practices] (e.g., exclusive use of secure internet protocols, strong mechanisms for storing credentials, etc.).
- Packages that access or handle personally identifiable or sensitive data should enable, document, and demonstrate workflows for de-identification, secure storage, other best practices to minimize risk of harm.

As standards for data privacy and research continue to evolve, we welcome input from authors on considerations specific to their software and supplemental documentation such as approval from university ethics review boards. These may be attached to issue threads of package submissions or pre-submission inquiries, or conveyed directly to editors if needed. General suggestions may be filed as issues in this book's repository.

5.4.1 Resources

The following resources may be helpful for researchers, package authors, editors and reviewers in addressing ethical questions related to privacy and research software.

- The Declaration of Helsinki and The Belmont Report provide fundamental principles for ethical practice by researchers.
- Several organizations provide guidance on how to translate these principles into the context of internet research. These include the Ethical Guidelines of The Association of Internet Researchers, the NESH Guide to Internet Research Ethics, and BPS' Ethics Guidelines for Internet-Mediated Research. Anabo et al (2019) provide a helpful overview of these.
- The Social Media Lab provides a high-level overview with data on normative expectations of privacy and use on social forums.
- Bechmann A., Kim J.Y. (2019) Big Data: A Focus on Social Media Research Dilemmas. In: Iphofen R. (eds) Handbook of Research Ethics and Scientific Integrity. https://doi.org/10.1007/978-3-319-76040-7_18-1
- Chu, K.-H., Colditz, J., Sidani, J., Zimmer, M., & Primack, B. (2021). Re-evaluating standards of human subjects protection for sensitive health data in social media networks. Social Networks, 67, 41–46. https://dx.doi.org/10.1016/j.socnet.2019.10.010
- Lomborg, S., & Bechmann, A. (2014). Using APIs for Data Collection on Social Media. The Information Society, 30(4), 256–265. https://dx.doi.org/10.1080/01972243.2014.915276
- Flick, C. (2016). Informed consent and the Facebook emotional manipulation study. *Research Ethics*, *12*(1), 14–28. https://doi.org/10.1177/1747016115599568
- Sugiura, L., Wiles, R., & Pope, C. (2017). Ethical challenges in online research: Public/private perceptions. *Research Ethics*, *13*(3–4), 184–199. https://doi.org/10.1177/1747016116650720
- Taylor, J., & Pagliari, C. (2018). Mining social media data: How are research sponsors and researchers addressing the ethical challenges? Research Ethics, 14(2), 1–39. https://doi.org/10.1177/1747016117738559
- Zimmer, M. (2010). "But the data is already public": on the ethics of research in Facebook. Ethics and Information Technology, 12(4), 313–325. https://dx.doi.org/10.1007/s10676-010-9227-5

5.5 Code of Conduct

rOpenSci's community is our best asset. Whether you're a regular contributor or a newcomer, we care about making this a safe place for you and we've got your back. We have a Code of Conduct that applies to all people participating in the rOpenSci community, including rOpenSci staff and leadership and to all modes of interaction online or in person. The Code of Conduct is maintained on the rOpenSci website.

6 Guide for Authors

This concise guide presents the software peer review process for you as a package author.

6.1 Planning a Submission (or a Pre-Submission Enquiry)

- Do you expect to maintain your package for at least 2 years, or to be able to identify a new maintainer?
- Consult our policies see if your package meets our criteria for fitting into our suite and does not overlap with other packages.
  - If you are unsure whether a package meets our criteria, feel free to open an issue as a pre-submission inquiry to ask if the package is appropriate.
  - Example response regarding overlap. Also consider adding some points about similar packages to your package documentation.
- Please consider the best time in your package's development to submit. Your package should be sufficiently mature so that reviewers are able to review all essential aspects, but keep in mind that review may result in major changes.
  - We strongly suggest submitting your package for review *before* publishing on CRAN or submitting a software paper describing the package to a journal. Review feedback may result in major improvements and updates to your package, including renaming and breaking changes to functions.
  - Do not submit your package for review while it or an associated manuscript is also under review at another venue, as this may result in conflicting requests for changes.
- Please also consider the time and effort needed to respond to reviews: think about your availability or that of your collaborators in the next weeks and months following a submission. Note that reviewers are volunteers, and we ask that you respect their time and effort by responding in a timely and respectful manner.
- If you use repostatus.org badges (which we recommend), submit when you're ready to get an *Active* instead of *WIP* badge. Similarly, if you use lifecycle badges, submission should happen when the package is *Stable*.
- For any submission or pre-submission inquiry the README of your package should provide enough information about your package (goals, usage, similar packages) for the editors to assess its scope without having to install the package. Even better, set up a pkgdown website for allowing more detailed assessment of functionality online.

  - At the submission stage, all major functions should be stable enough to be fully documented and tested; the README should make a strong case for the package.
  - Your README file should strive to explain your package's functionality and aims, assuming readers have little to no domain knowledge. All technical tems, including references to other software, should be clarified.

- Your package will continue to evolve after review, the chapter on *Package evolution* provides guidance about the topic.

6.2 Preparing for Submission

- Read and follow our packaging style guide, reviewer's guide to ensure your package meets our style and quality criteria.
- Feel free to ask any questions about the process, or your specific package, in our Discussion Forum.
- All submissions are automatically checked by our pkgcheck system to ensure packages follow our guidelines. All authors are expected to have run the main `pkgcheck` function locally to confirm that the package is ready to be submitted. Alternatively, an even easier way to ensure a package is ready for submission is to use the `pkgcheck` GitHub Action to run `pkgcheck` as a GitHub Action, as described in our blog post.
- If your package requires unusual system dependencies (see *Packaging Guide*) for our GitHub Action to pass, please submit a pull request adding them to our base Dockerfile.
- If there are any aspects of `pkgcheck` which your package is unable to pass, please explain reasons in your submission template.
- If you feel your package is in scope for the Journal of Open-Source Software (JOSS), do not submit it to JOSS consideration until after the rOpenSci review process is over: if your package is deemed in scope by JOSS editors, only the accompanying short paper would be reviewed, (not the software that will have been extended reviewed by rOpenSci by that time). Not all rOpenSci packages will meet the criteria for JOSS.

6.3 The Submission Process

- Software is submitted for review by opening a new issue in the software review repository and filling out the template.
- The template begins with a section which includes several HTML-styled variables (`<!---variable--->`). These are used by our `ropensci-review-bot`, and must be left in place, with values filled between the indicated start and end points, like this:

```
<!---variable--->insert value here<!---end-variable>
```

- Communication between authors, reviewers and editors will first and foremost take place on GitHub so that the review thread can serve as a full record of the review. You may choose to contact the editor by email or Slack if private consultation is needed (e.g., asking how to respond to a reviewer question). Do not contact reviewers off-thread without asking them in the GitHub thread whether they agree to it.
- *When submitting a package please make sure your GitHub notification settings make it unlikely you will miss a comment.*
- Packages are automatically checked on submission by our `pkgcheck` system, which will confirm whether or not a package is ready to be reviewed.
- Submitted packages can be hosted in the main/default branch, or any other non-default branch. In the latter case, it is encouraged, but not required, to submit the package via a dedicated `ropensci-software-review` branch.

6.4 The Review Process

- An editor will review your submission within 5 business days and respond with next steps. The editor may assign the package to reviewers, request that the package be updated to meet minimal criteria before review, or reject the package due to lack of fit or overlap.
- If your package meets minimal criteria, the editor will assign 1-3 reviewers. They will be asked to provide reviews as comments on your issue within 3 weeks.
- We ask that you respond to reviewers' comments within 2 weeks of the last-submitted review, but you may make updates to your package or respond at any time. Your response should include a link to the updated NEWS.md of your package. Here is an author response example. Once the response is commited, submit it using the bot. We encourage ongoing conversations between authors and reviewers. See the reviewing guide for more details.
- Any time package changes are likely to alter the results of the automated `pkgcheck` checks, authors can request a re-check with the command, `@ropensci-review-bot check package`.
- Please notify us immediately if you are no longer able to maintain your package or to respond to reviews. You will then be expected to either retract a submission, or to find alternative package maintainers. You can also discuss maintenance issues in the rOpenSci slack workspace.
- Once your package is approved, we will provide further instructions about the transfer of your repository to the rOpenSci repository.

Our code of conduct is mandatory for everyone involved in our review process.

7 Guide for Reviewers

Thanks for accepting to review a package for rOpenSci! This chapter consists of our guidelines to prepare, submit and follow up on your review.

You might contact the editor in charge of the submission for any question you might have about the process or your review.

Please strive to complete your review within 3 weeks of accepting a review request. We will aim to remind reviewers of upcoming and past due dates. Editors may assign additional or alternate reviewers if a review is excessively late.

rOpenSci's community is our best asset. We aim for reviews to be open, non-adversarial, and focused on improving software quality. Be respectful and kind! See our reviewers guide and code of conduct for more.

If you use our standards/checklists etc. when reviewing software elsewhere, do tell the recipients (e.g. journal editors, students, internal code review) that they came from rOpenSci, and tell us in our public forum, or privately by email.

7.1 Volunteering as a reviewer

Thank you for your desire to participate in rOpenSci software peer-review as a reviewer!

Please fill our volunteering form.

If you see a current submission that is particularly relevant to your interests please email `info@ropensci.org`, including the name of the package, the URL to the submission issue and the name of the assigned editor. However, keep in mind that reviewer invitations are kept at the editor's discretion, and the editor might well have already emailed people. Please do not volunteer for all issues, and do not volunteer via GitHub interface.

For other ways to contribute, refer to rOpenSci contributing guide.

7.2 Preparing your review

Reviews should be based on the latest GitHub version on the default branch, unless otherwise indicated by package authors. All submissions trigger a detailed report on package structure and functionality, generated by our `pkgcheck` package. If the package has changed substantially since the last checks, you may request a re-check with the command `@ropensci-review-bot check package`. Note that when installing the package to review it, you should make sure you have all dependencies available (for instance run `pak::pak()`).

7.2.1 General guidelines

To review a package, please begin by copying our review template (or our review template in Spanish) and using it as a high-level checklist. In addition to checking off the minimum criteria, we ask that you provide general comments addressing the following:

- Does the code comply with general principles in the Mozilla reviewing guide?
- Does the package comply with the rOpenSci packaging guide?
- Are there improvements that could be made to the code style?
- Is there code duplication in the package that should be reduced?
- Are there user interface improvements that could be made?
- Are there performance improvements that could be made?
- Is the documentation (installation instructions/vignettes/examples/demos) clear and sufficient? Does it use the principle of *multiple points of entry* i.e. takes into account the fact that any piece of documentation may be the first encounter the user has with the package and/or the tool/data it wraps?
- Were functions and arguments named to work together to form a common, logical programming API that is easy to read, and autocomplete?
- If you have your own relevant data/problem, work through it with the package. You may find rough edges and use-cases the author didn't think about.

Please be respectful and kind to the authors in your reviews. Our code of conduct is mandatory for everyone involved in our review process. We expect you to submit your review within 3 weeks, depending on the deadline set by the editor. Please contact the editor directly or in the submission thread to inform them about possible delays.

We encourage you to use automated tools to facilitate your reviewing. These include:

- Checking the initial package report generated by our `@ropensci-review-bot`.
- Checking the package's logs on its continuous integration services (GitHub Actions, Codecov, etc.)
- Running `devtools::check()` and `devtools::test()` on the package to find any errors that may be missed on the author's system.

- Seeing whether tests' skipping is justified (e.g. `skip_on_cran()` tests that do real API requests vs. skipping all tests on one operating system).
- If the package is not submitted via the default/main branch, remember to switch to the submitted review branch before starting your review. In this case, you will also have to search the package locally, as GitHub search is limited to the default branch. Further, documentation hosted on a `pkgdown` website is not necessarily up-to-date, and we recommend to inspect the package's documentation locally by running `pkgdown::build_site()`.

Reviewers may also re-generate package check results from `@ropensci-review-bot` at any time by issuing the single comment in a review issue: `@ropensci-review-bot check package`.

7.2.2 Off-thread interactions

If you interact with the package authors and talked about the review outside a review thread (in chats, DMs, in-person, issues in the project repository), please make sure that your review captures and/or links to elements from these conversations that are relevant to the process.

7.2.3 Experience from past reviewers

First-time reviewers may find it helpful to read (about) some previous reviews. In general you can find submission threads of onboarded packages here. Here are a few chosen examples of reviews (note that your reviews do not need to be as long as these examples):

- `rtika` review 1 and review 2
- `NLMR` review 1 and review 2
- `bowerbird` pre-review comment, review 1, review 2.
- `rusda` review (from before we had a review template)

You can read blog posts written by reviewers about their experiences via this link. In particular, in this blog post by Mara Averick read about the "naive user" role a reviewer can take to provide useful feedback even without being experts of the package's topic or implementation, by asking themselves *"What did I think this thing would do? Does it do it? What are things that scare me off?"*. In another blog post Verena Haunschmid explains how she alternated between using the package and checking its code.

As both a former reviewer and package author, and now editor, Adam Sparks wrote "[write] a good critique of the package structure and best coding practices. If you know how to do something better, tell me. It's easy to miss documentation opportunities as a developer, as a reviewer, you have a different view. You're a user that can give feedback. What's not clear in the package? How can it be made more clear? If you're using it for the first time, is it easy? Do you know another R package that

maybe I should be using? Or is there one I'm using that perhaps I shouldn't be? If you can contribute to the package, offer."

7.2.4 Helper package for reviewers

If working in RStudio, you can streamline your review workflow by using the `pkgreviewr` package created by associated editor Anna Krystalli. Say you accepted to review the `refnet` package, you'd write

```
remotes::install_github("ropensci-org/pkgreviewr")
library(pkgreviewr)
pkgreview_create(pkg_repo = "embruna/refnet",
                 review_parent = "~/Documents/workflows/rOpenSci/reviews/")
```

and

- the GitHub repo of the `refnet` package will be cloned.
- a review project will be created, containing a notebook for you to fill, and the review template.
- note that if the package is not submitted via the default/main branch, you need to switch to the submitted branch before starting your review.

7.2.5 Feedback on the process

We encourage you to ask questions and provide feedback on the review process on our forum.

7.3 Submitting the Review

- When your review is complete, paste it as a comment into the package software-review issue.
- Additional comments are welcome in the same issue. We hope that package reviews will work as an ongoing conversation with the authors as opposed to a single round of reviews typical of academic manuscripts.
- You may also submit issues or pull requests directly to the package repo if you choose, but if you do, please comment about them and link to them in the software-review repo comment thread so we have a centralized record and text of your review.
- Please include an estimate of how many hours you spent on your review afterwards.

7.4 Review follow-up

Authors should respond within 2 weeks with their changes to the package in response to your review. At this stage, we ask that you respond as to whether the changes sufficiently address any issues raised in your review. We encourage ongoing discussion between package authors and reviewers, and you may ask editors to clarify issues in the review thread as well.

You'll use the approval template.

8 Guide for Editors

Software Peer Review at rOpenSci is managed by a team of editors. The Editor-in-Chief (EiC) role is rotated (generally quarterly) amongst experienced members of our editorial board. Information on current and recent editors and their activities can be viewed on our editorial dashboard at dashboard.ropensci.org.

This chapter presents the responsabilities of the Editor-in-Chief, of any editor in charge of a submission, how to respond to an out-of-scope submission and how to manage a dev guide release.

If you're a guest editor, thanks for helping! Please contact the editor who invited you to handle a submission for any question you might have.

> Always assume participants in the software review system (fellow editors, submitters, reviewers) are doing their best, and communicate gracefully accordingly, especially when inquiring why a thing is delayed.

8.1 Editors' responsibilities

- In addition to handling packages (about 4 a year), editors weigh in on group editorial decisions, such as whether a package is in-scope, and determining updates to our policies. We generally do this through Slack, which we expect editors to be able to check regularly.
- You only need to keep track of your own submissions, but if you do notice an issue with a package that is being handled by another editor, feel free to raise that issue directly with the other editor, or post the concern to editors-only channel on slack. Examples:
  - You know of an overlapping package, that hasn't been mentioned in the process yet.
  - You see a question to which you have an expert answer that hasn't been given after a few days (such as linking to a blog post which may answer a question).
  - Concerns related to general review progress, including aspects such as the speed of the process, should be directed to the current Editor-in-Chief.

8.2 Handling Editor's Checklist

8.2.1 Upon submission:

- Submission will automatically generate package check output from ropensci-review-bot. The check results should be examined for any outstanding issues (most exceptions will need to be justified by the author in the particular context of their package). Checks can be re-run after any package change with the comment `@ropensci-review-bot check package`.
- For statistical submissions (identifiable as "Submission Type: Stats" in issue template), add the "stats" label to the issue (if not already added).
- Check that issue template has been properly filled out. Most common oversights and omissions should be caught and noted by the bot, but a manual check always helps. Editors can edit templates directly for minor fixes, or may direct authors to fill any mandatory template fields that may be missing.
- The checking system is rebuilt at every Tuesday at 00:01 UTC, and can take a couple of hours. If automatic checks fail around that time, wait a few hours and try again.
- After automatic checks are posted, use the editor template to guide initial checks and record your response to the submission. You can also streamline your editor checks by using the `pkgreviewr` package created by associate editor Anna Krystalli. Please strive to finish the checks and start looking for reviewers within 5 working days.
- Check against policies for fit and overlap. Initiate discussion via Slack #software-review channel if needed for edge cases that haven't been caught by previous checks by the EiC. If reject, see this section about how to respond.
- Ensure that the package gets tested on multiple platforms (having the package built on several operating systems via GitHub Actions for instance; see criteria in this section of the CI chapter for further details and options).
- Wherever possible when asking for changes, direct authors to automatic tools such as `usethis` and `styler`, and to online resources (sections of this guide, sections of the R packages book) to make your feedback easier to use. See this example of editor's checks.
- Ideally, any remarks you make as editor should be addressed before assigning reviewers.
- If initial checks show major gaps, request changes before assigning reviewers. If the author mentions changes might take time, apply the holding label by calling `@ropensci-review-bot put on hold`. You'll get a reminder in the issue every 90 days to check in with the package author(s).
- If the package raises a new issue for rOpenSci policy, start a conversation in Slack or open a discussion on the rOpenSci forum to discuss it with other editors (example of policy discussion).

8.2.2 Look for and assign two reviewers:

8.2.2.1 Tasks

- Comment with `@ropensci-review-bot seeking reviewers`.
- Use the email template if needed for inviting reviewers
  - When inviting reviewers, include something like "if I don't hear from you in a week, I'll assume you are unable to review," so as to give a clear deadline when you'll move on to looking for someone else.
- Assign reviewers with `@ropensci-review-bot assign @username as reviewer`. `add` can also be used instead of `assign`, and `to reviewers` (plural) instead of `as reviewer` (single). The following is thus also valid: `@ropensci-review-bot add @username to reviewers`. One command should be issued for each reviewer. If needed later, remove reviewers with `@ropensci-review-bot remove @username from reviewers`.
- If you want to change the due date for a review use `@ropensci-review-bot set due date for @username to YYYY-MM-DD`.

8.2.2.2 How to look for reviewers

8.2.2.2.1 Where to look for reviewers?

As a (guest) editor, use

- the potential suggestions made by the submitter(s), (although submitters may have a narrow view of the types of expertise needed. We suggest not using more than one of suggested reviewers);
- the Airtable database of reviewers and volunteers (see next subsection);
- and the authors of rOpenSci packages.

When these sources of information are not enough,

- ping other editors in Slack for ideas,
- look for users of the package or of the data source/upstream service the package connects to (via their opening issues in the repository, starring it, citing it in papers, talking about it on Twitter).
- You can also search for authors of related packages on r-pkg.org.
- R-Ladies has a directory specifying skills and interests of people listed.
- You may post a request for reviewers in the #general and/or #software-review channels on the rOpenSci Slack, or on social media.

8.2.2.2.2 Tips for reviewer search in Airtable

You can use filters, sorting, and search to identify reviewers with particular experience:

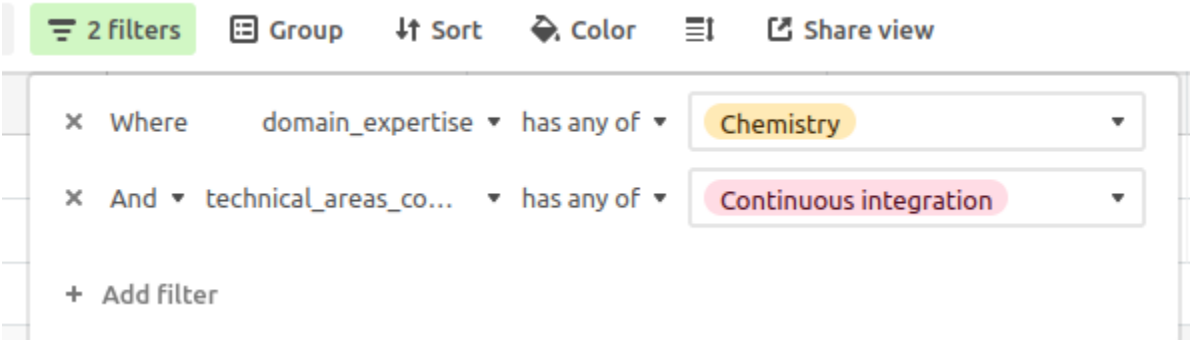

Figura 8.1: Screenshot of the Airtable filters interface with a filter on domain expertise that has to include chemistry and technical areas that have to include continuous integration

Please check the reviewer's most recent review and avoid anyone who has reviewed anyone in the past six months. Also, please check if a first-time reviewers have indicated that they `require_mentorship`. If so, please use the mentorship portion of the email template and be prepared to provide additional guidance.

8.2.2.2.3 Criteria for choosing a reviewer

Here are criteria to keep in mind when choosing a reviewer. You might need to piece this information together by searching CRAN and the potential reviewer's GitHub page and general online presence (personal website, Twitter).

- Has not reviewed a package for us within the last 6 months.
- Some package development experience.
- Some domain experience in the field of the package or data source
- No conflicts of interest.
- Try to balance your sense of the potential reviewer's experience against the complexity of the package.
- Diversity - with two reviewers both shouldn't be cis white males.
- Some evidence that they are interested in openness or R community activities, although cold emailing is fine.

Each submission should be reviewed by *two* package reviewers. Although it is fine for one of them to have less package development experience and more domain knowledge, the review should not be split in two. Both reviewers need to review the package comprehensively, though from their particular perspective. In general, at least one reviewer should have prior reviewing experience, and of course inviting one new reviewer expands our pool of reviewers.

8.2.3 During review:

- Check in with reviewers and authors occasionally. Offer clarification and help as needed.
- In general aim for 3 weeks for review, 2 weeks for subsequent changes, and 1 week for reviewer approval of changes.
- Upon each review being submitted,
  - Write a comment thanking the reviewer with your words;
  - Record the review via typing a new comment `@ropensci-review-bot submit review <review-url> time <time in hours>`. E.g. for the review https://github.com/ropensci/software-review/issues/329#issuecomment-809783937 the comment would be `@ropensci-review-bot submit review https://github.com/ropensci/software-revi time 4`.
- If the author stops responding, refer to the policies and/or ping the other editors in the Slack channel for discussion. Importantly, if a reviewer was assigned to a closed issue, contact them when closing the issue to explain the decision, thank them once again for their work, and make a note in our database to assign them to a submission with high chances of smooth software review next time (e.g. a package author who has already submitted packages to us).
- Upon changes being made, change the review status tag to `5/awaiting-reviewer-response`, and request that reviewers indicate approval with the reviewer approval template.

8.2.4 After review:

- `@ropensci-review-bot approve <package-name>`
- *If the original repository wishes to keep the package in their own GitHub organization rather than transfer to ropensci, add a line with the repository URL to this repos list to ensure the package gets included in rOpenSci package registry.*
- Nominate a package to be featured in an rOpenSci blog post or tech note if you think it might be of high interest. Please note in the software review issue one or two things the author could highlight, and tag `@ropensci/blog-editors` for follow-up.
- If authors maintain a gitbook that is at least partly about their package, contact an rOpenSci staff member so they might contact the authors about transfer to the `ropensci-books` GitHub organisation.

8.2.5 Package promotion:

- Direct the author to the chapters of the guide about package releases, marketing and GitHub grooming.

8.3 EiC Responsibilities

Rotating Editors-in-Chief (EiCs) generally serve for 3 months or a time agreed to by all members of the editorial board. The EiC is entitled to taking scope and overlap decisions as independently as possible (but can still request help and advice). Information on current status of all editorial team members is presented on our *Editorial Dashboard*. The EiC is responsible for the following tasks:

- On assuming EiC rotation, reviewing the status of current open reviews as detailed on the *Dashboard* page, and issuing reminders to other editors or package authors as needed. See the following sub-section for more details
- Watching all new issues posted to the software-review repo, for which the EiC must either subscribe to repo notifications on GitHub, or watch the `#software-peer-review-feed` channel on Slack.
- Tagging each new full submission with `0/editorial-team-prep`
- Calling `@ropensci-review-bot check srr` on pre-submission enquiries for statistical software. See corresponding *Stats Dev Guide* chapter for details.
- Finding an editor (potentially including yourself) to handle each submission. Currently available editors are indicated on the *Editorial Dashboard*, and editorial workloads should be distributed as evenly as possible, through referring to the *Dashboard* charts of recent editorial load.
- Assigning editors by issuing the command:

  ```
  @ropensci-review-bot assign @username as editor
  ```

  This will also add tag `1/editor-checks` to the issue.
- Regularly (for instance weekly) monitoring the pace of all open reviews by keeping an eye on the *Dashboard* page, and reminding other editors to move packages along as needed.
- Responding to issues posted to [the software-review-meta repo
- Making decisions on scope and overlap for pre-submission inquiries, referrals from JOSS or other publication partners, and submissions. Discussions should be initiated in the rOpenSci Slack editors-only channel through summarising the (pre-)submitted/referred software, along with any concerns the EiC might have. If after the EiC feels they haven't received enough answers after a day or two, they can ping all editors.
  - Any editor should feel free to step in on these. See this section about how to respond to out-of-scope (pre-) submissions.
  - After explaining an out-of-scope decision, write an issue comment `@ropensci-review-bot out-of-scope`.

8.3.1 The rOpenSci Editorial Dashboard

The *rOpenSci Editorial Dashboard* is updated daily, primary by extracting information on all software review issues on GitHub, along with additional information from Slack and our Airtable database. The dashboard provides an up-to-date overview of our editorial team, their recent reponsibilities, and the current state of all software review issues. The EiC (or any editors who are interested) can gain an overview of the editorial team status, availability, and recent workloads on the *editors* page. This should be used to find and assign editors for new software review issues. An overview of all current software reviews is on the **Software Review* page. Entries on this page are colored by a measure of "urgency", summarised in the table at the bottom of that page.

Specific tasks for reviews in the specific review stages include:

- Looking over submissions in "0/presubmission" and "1/editorial-team-prep", to check whether any action needs to be taken (such as polling editors, making decisions, putting issues on hold, pinging for updates, or finding and assigning editors).
- Looking over submissions in "2/seeking-reviewer(s)" to ensure things are progressing quickly. If the reviewer search has been going for unusually long (red color), check whether the submission is on hold, read the thread to gather context, and contact the editor in private to ask for more information.
- Looking over submissions in "3/reviewer(s)-assigned". If there are still missing reviews after an unusually long time (red color), check whether the submission is on hold, read the thread to gather context, and contact the editor in private to ask for more information.
- Looking over submissions in "4/review(s)-in-awaiting-changes". If some are still lacking an author response after an unusually long time (red color), check whether the submission is on hold, read the thread, and contact the editor in private to ask for more information.

8.3.2 Asking for more details

In some cases online documentation is sparse. Minimal README, no pkgdown website make assessment harder. In that case please ask for more details: even if the package is deemed out-of-scope, the package docs will have gotten better so we are fine asking for these efforts.

Example text

```
Hello <username> and many thanks for your submission.

We are discussing whether the package is in scope and need a bit more information.

Would you mind adding more details and context to the README?
After reading it someone with little domain knowledge should have been informed about the ai
```

```
<optional>
If a package has overlapping functionality with other packages, we require it to demonstrate
</optional>
```

8.3.3 Inviting a guest editor

After discussion with other editors the EiC might invite a guest editor to handle a submission (e.g. if submission volume is large, if all editors have a conflict of interest, if specific expertise is needed, or as a trial prior to inviting a person to join the editorial board).

When inviting a guest editor,

- Ask about conflicts of interest using the same phrasing as for reviewers,
- Give a link to the guide for editors.

If the person said yes (yay!),

- Make sure they enabled 2FA for their GitHub account,
- Invite them to the `ropensci/editors` team and to the ropensci organization,
- Once they've accepted this repo invitation, assign the issue to them,
- Ensure they're (already) invited to rOpenSci Slack workspace,
- Add their name to the Airtable guest-editor table (so their names might appear in this book and in the software-review README).

After the review process is finished (package approved, issue closed),

- Thank the guest editor again,
- Remove them from the `ropensci/editors` team (but not from the ropensci organization).

8.4 Responding to out-of-scope submissions

Thank authors for their submission, explain the reasons for the decision, and direct them to other publication venues if relevant, and to the rOpenSci discussion forum. Use wording from Aims and scope in particular regarding the evolution of scope over time, and the overlap and differences between unconf/staff/software-review development.

Examples of out-of-scope submissions and responses.

8.5 Answering reviewers' questions

Reviewers might ask for feedback on e.g. the tone of their review. Beside pointing them at general guidance in this guide, asking editors / opening an issue when such guidance is lacking, here are some review examples that might be useful.

- tough-but-constructive example: the part of this review suggesting a re-write of the vignette: ropensci/software-review#191 (comment).
- the `slopes` package, which ended up being fundamentally redesigned in response to the reviews. All reviews/reviewers were at all times entirely constructive, which seems to have played a major role in motivating the authors to embark on such a major overhaul. Comments such as, *"this package does not …"* or *"has not …"* were invariably followed by constructive suggestions for what could be done (there are, for example, several in one of the first reviews).
- tic reviews politely expressed reservations: https://github.com/ropensci/software-review/issues/305#issuecomment-504762517 and https://github.com/ropensci/software-review/issues/305#issuecomment-508271766
- bowerbird useful "pre-review" that resulted in a package split before the actual reviews.

8.6 Managing a dev guide release

If you are in charge of managing a release of the very book you are reading, use the book release guidance as an issue template to be posted in the dev guide issue tracker, and do not hesitate to ask questions to other editors.

8.6.1 Dev guide governance

For very small amendments to the dev guide, no PR review is needed. For larger amendments, request review from at least a few editors (if none participated in the discussion related to the amendment, request a review from all of them on GitHub, and in the absence of any reaction merge after a week).

Two weeks before a dev guide release, once the PR from dev to master **and the release blog post** are ready for review, all editors should be pinged by GitHub ("review request" on the PR from dev to master) and Slack, but the release doesn't need all of them to explicitly approve the release.

8.6.2 Blog post about a release

The blog post about a release will be reviewed by editors, and one of `@ropensci/blog-editors`.

8.6.2.1 Content

Refer to the general rOpenSci blogging guidance, and the more specific guidance below.

First example of such a post; second example.

The blog post should mention all important items from the changelog organized in (sub)sections: e.g. a section about big change A, another one about big change B, and one about smaller changes lumped together. Mention the most important changes first.

For each change made by an external contributor, thank them explicitly using the information from the changelog. E.g. `[Matt Fidler](https://github.com/mattfidler/) amended our section on Console messages [ropensci/dev_guide#178](https://github.com/ropensci/dev_guide/pu`

At the end of the post, mention upcoming changes by linking to open issues in the issue tracker, and invite readers to contribute to the dev guide by opening issues and participating in open discussions. Conclusion template:

```
In this post we summarized the changes incorporated into our book ["rOpenSci Packages: Develo
We are grateful for all contributions that made this release possible.
We are already working on updates for our next version, such as ISSUE1, ISSUE2.
Check out the [the issue tracker](https://github.com/ropensci/dev_guide/issues/) if you'd lik
```

8.6.2.2 Authorship

The editor writing the post is first author, other editors are listed by alphabetical order.

9 Editorial management

Guidance for managing the editorial team.

9.1 Recruiting new editors

Recruiting new editors and maintaining a sufficient and well-balanced editorial board is a responsibility of the Software Review Lead, with support and advice from the editorial board.

Steps:

- Start a private channel for discussion (so that it does not have a history in the editors channel that future editors will join, which could be awkward).
- Ping editors to be sure they get a notification as this is an important topic.
- Wait for a majority of editors to chime in before inviting someone. Leave them one week to respond.

9.2 Inviting a new editor

- Candidates might start by being guest editors. When inviting them as guest editor, invite them as you would invite a guest editor for other reasons.
- If a candidate is guest editor first, assess how the process went after the submission is tackled. Asked other editors for their advice again.
- Send an email.

```
We would like to invite you to join the rOpenSci editorial board as a full member. [SPECIFIC
We think you would make a wonderful addition to the team.

[IF GUEST EDITOR:You are familiar with the editor's role as you've been a guest editor].  We
We ask that editors make an informal commitment of serving for two years, and re-evaluate th
On a short-term basis, any editor can decline to handle a package or say, "I'm pretty busy, l
```

```
In addition to handling packages, editors weigh in on group editorial decisions, such as whet
We generally do this through Slack, which we expect editors to be able to check regularly.
We have editorial board calls annually.
We also rotate Editor-in-Chief responsibilities (first-pass scope decisions and assigning edi
You'll have the opportunity to enter this rotation once you have been on the board for some t
Some of us also take on bigger projects to improve the peer-review process, though this is en

We hope that you'll join the board!
It's an exciting time for peer-review at rOpenSci.

Please give this some thought, ask us any questions you have, and let us know whether you can

Best,
[EDITOR], on behalf of the rOpenSci Editorial Board
```

9.3 Onboarding a new editor

- Inform rOpenSci community manager so that
  - editors are added to the rOpenSci website.
  - an introduction blog post can be put together.
- If they haven't already done so as guest editors, request that the new editor turn on two-factor authentication (2FA) for GitHub.
- Invite editors to the rOpenSci GitHub organization as member, as a member of the `editors` team and the `data-pkg-editors` or `stats-board` sub-team, as appropriate. This will give them appropriate permissions and allow them to get team-specific notifications.
- Editors need access to the AirTable database of software review.
- Editors need access to the private editors channel in rOpenSci Slack workspace (and to the Slack workspace in general if they didn't previously, in that case ask rOpenSci community manager).
- Post a welcome message in the channel, pinging all editors.
- In the Slack workspace they need to be added to the editors team so that `@editors` will ping them too.
- We add editors' names to
  - dev_guide authors list
  - dev_guide chapter introducing software review (at two locations in this file, as editors and a bit below to remove them from the reviewers list)

  - software-review README (in two places in this file as well) Both the dev_guide and software-review README are automatically knit via continuous integration.

- Add editors to https://github.com/orgs/ropensci/teams/editors/members

9.4 Offboarding an editor

- Thank them for their work!
- Remove them from the editors-only channel and the editors Slack team.
- Remove them from https://github.com/orgs/ropensci/teams/editors/members and sub-team.
- Inform rOpenSci community manager or some other staff emember so that they might be moved to alumni team members on the website.
- Remove their access to the Airtable workspace.
- Remove them from

  - dev_guide chapter introducing software review (at two locations in this file, as editors and a bit below to remove them from the reviewers list)
  - software-review README (in two places in this file as well) Both the dev_guide and software-review README are automatically knit via continuous integration.

Parte III

Maintaining Packages

10 Folha de dicas de manutenção do pacote rOpenSci

Um lembrete da infraestrutura e dos canais de contato para mantenedores de pacotes rOpenSci.

10.1 Você precisa de ajuda?

Se você precisar de ajuda pontual (por exemplo, uma revisão de PR; ou alguma solução de problemas de CI), ou ajuda para procurar co-mantenedores ou um novo mantenedor, ou se precisar que retiremos seu pacote, envie-nos um ping no GitHub via `@ropensci/admin` ou envie um e-mail para você `info@ropensci.org`. Você também pode usar nosso canal de manutenção de pacotes no Slack.

Nunca hesite em pedir ajuda.

10.2 Acesso ao repositório do GitHub

Você deve ter acesso administrativo ao repositório do GitHub do seu pacote. Se esse não for mais o caso (por exemplo, o processo automatizado falhou ou você perdeu o acesso depois de ter que desativar temporariamente a autenticação de dois fatores), entre em contato conosco via `info@ropensci.org`.

10.3 Outros tópicos do GitHub

Se tiver alguma pergunta ou solicitação sobre o GitHub (por exemplo, adicionar um colaborador à organização do GitHub), você pode usar um canal público do espaço de trabalho slack do rOpenSci ou enviar um ping para `@ropensci/admin` no GitHub.

10.4 Documentação do pkgdown

Veja Documentos do rOpenSci.

10.5 Acesso ao espaço de trabalho do rOpenSci no Slack

Os mantenedores e desenvolvedores de pacotes devem ter acesso a Slack do rOpenSci. Se você não recebeu o convite ou não o aceitou a tempo, ou se você quiser que um novo colaborador regular receba um convite, envie um e-mail para `info@ropensci.org`, indicando para qual endereço de e-mail você deseja receber o convite.

Você pode achar que o canal #package-maintenance é relevante para perguntas e respostas, bem como para uma comiseração amigável quando necessário.

10.6 Publicações no blog sobre pacotes

Consulte nosso guia do blog.

10.7 Promoção de problemas de pacotes

Rotular as edições com “procura-se ajuda” para obtê-las para que sejam transmitidos à comunidade.

10.8 Promoção de casos de uso de pacotes

Você pode relatar casos de uso do seu pacote ou incentivar os usuários a relatá-los por meio do nosso fórum para obtê-los publicados em nosso site e em nosso boletim informativo.

11 Guia de colaboração

Ter pessoas colaborando vai trazer melhorias para o seu pacote e, se você integrar algumas delas como autoras do pacotes com permissões de escrita no repositório, seu pacote será desenvolvido de forma mais sustentável. Também pode ser muito agradável trabalhar em equipe!

Este capítulo contém nossa orientação para colaboração, em uma seção sobre como tornar o seu repositório amigável para contribuições e colaborações por infraestrutura (código de conduta, diretrizes de contribuição, rótulos de problemas); e uma seção sobre como colaborar com novos colaboradores, em particular no contexto da organização da rOpenSci no GitHub.

Além dessas dicas técnicas, é importante lembrar-se de ser gentil e levar em conta a perspectiva das outras pessoas, especialmente quando as prioridades delas forem diferentes das suas.

11.1 Torne a contribuição e a colaboração do seu repositório amigáveis

11.1.1 Código de conduta

Após a transferência para a nossa organização no GitHub, o Código de Conduta da rOpenSci será aplicado ao seu projeto. Você deve adicionar este texto ao README:

```
Please note that this package is released with a [Contributor
Code of Conduct](https://ropensci.org/code-of-conduct/).
By
contributing to this project, you agree to abide by its terms.
```

Em seguida, exclua o código de conduta atual do pacote, caso exista algum.

11.1.2 Guia de contribuição

Você pode usar a *issue*, a solicitação de *pull request* e as diretrizes de contribuição. Ter um arquivo sobre contribuição como `.github/CONTRIBUTING.md` ou `docs/CONTRIBUTING.md` é obrigatório. Uma maneira fácil de inserir um modelo para um guia de contribuição é usar

a função `use_tidy_contributing()` do pacote `usethis` que insere este modelo como `.github/CONTRIBUTING.md`. Um exemplo mais completo é este modelo de Peter Desmet ou as abrangentes páginas da wiki do pacote `mlr3` no GitHub. Em geral, esses e outros modelos precisarão ser modificados para serem usados com um pacote da rOpenSci, principalmente por meio de referências e links para o nosso Código de Conduta conforme descrito em um outro lugar neste livro. Modificar um guia de contribuição genérico para adicionar um toque pessoal também tende a fazer com que ele pareça menos genérico e mais sincero. As preferências pessoais em um guia de contribuição incluem:

- Preferências de estilo? No entanto, você pode preferir tornar o estilo uma configuração (de lintr, styler) ou para corrigir você mesmo o estilo de código especialmente se você não usar um estilo de código popular como o estilo de código do tidyverse.
- Infraestrutura como a do `roxygen2`?
- Preferências de fluxo de trabalho? *Issue* antes de um PR?
- Uma declaração de escopo, como no pacote skimr?
- Criação de contas de sandbox? *Mocking* em testes? Links para documentos externos?

A rOpenSci também incentiva os guias de contribuição a incluírem uma declaração de ciclo de vida que esclareça as visões e expectativas para o desenvolvimento futuro do seu pacote, como neste exemplo. É necessário que os pacotes estatísticos tenham uma declaração de ciclo de vida, conforme especificado em *Padrões estatísticos gerais* G1.2. Esses links fornecem um modelo para uma declaração de ciclo de vida simples. Os arquivos `CONTRIBUTING.md` também podem descrever como você reconhece as contribuições (consulte esta seção).

Recomendamos que você envie comentários que não sejam um relatório de bug ou uma solicitação de *feature* para o Fórum da rOpenSci após certificar-se de que você verá essas perguntas no fórum. As pessoas usuárias podem usar o fórum para fazer perguntas sobre o uso e relatar seus casos de uso, e você pode se inscrever em categorias e tags individuais para receber notificações sobre o seu pacote. Sinta-se à vontade para mencionar isso nos documentos do seu pacote e/ou nas diretrizes de contribuição/modelo de *issue*. Oriente as pessoas que usam seu pacote a marcar as mensagens com o nome do pacote.

Quando uma pull request estiver mais perto de ser mesclada, você poderá usar um *GitHub Actions PR workflow* para estilizar o código com o pacote `styler`.

11.1.3 Gerenciamento de *issues*

Ao usar os recursos do GitHub em torno dos *issues*, você pode ajudar os possíveis colaboradores a encontrá-los e tornar público o seu roteiro.

11.1.3.1 Modelos de *issues*

Você pode usar um ou vários modelo(s) de *issue(s)* para ajudar as pessoas usuárias a preencherem os relatórios de bugs ou as solicitações de *feature*. Quando há vários modelos de *issues*, os usuários que clicam em abrir uma nova *issue* veem um menu que orienta as suas escolhas.

Você pode até configurar uma das opções para apontar para algum lugar fora do seu repositório (por exemplo, um fórum de discussão).

Consulte a Documentação do GitHub.

11.1.3.2 Rotulagem de *issues*

Você pode usar rótulos como “help wanted” (procura-se ajuda) e “good first issue” (bom primeiro problema) para ajudar colaboradores em potencial, inclusive novatos, a encontrar o seu repositório. Consulte o artigo do GitHub. Você também pode usar o rótulo “Beginner” (Iniciante). Você pode ver exemplos de problemas para iniciantes em todos os repositórios da rOpenSci.

11.1.3.3 Fixando *issues*

Você pode fixar até 3 *issues* por repositório que aparecerão na parte superior do seu rastreador de *issues* como cartões de *issues* bonitos. Isso pode ajudar a divulgar quais são suas prioridades.

11.1.3.4 Marcos

Você pode criar marcos e atribuir problemas a eles, o que ajuda outras pessoas a verem o que você planeja para a próxima versão do seu pacote, por exemplo.

11.1.4 Comunicação com as pessoas usuárias

Você pode indicar para as pessoas usuárias o fórum da rOpenSci se você monitorar ou ativar as Discussões no GitHub para o repositório do seu pacote. Cada discussão do GitHub pode ser convertida em um *issue*, se necessário (e vice-versa, os *issues* podem ser convertidos em discussões).

11.2 Trabalhando com colaboradores

Em primeiro lugar: mantenha contato com o seu repositório do GitHub!

- Não se esqueça de **observar o seu repositório do GitHub** para receber notificações sobre *issues* ou *pull requests* (como alternativa, se você trabalha de forma concentrada em certas épocas, talvez adicione essas informações ao guia de contribuição).
- não se esqueça de enviar as atualizações que você tem localmente.
- desative os testes com falha se você não puder corrigí-los, pois eles criam ruído nos PRs que podem confundir colaboradores iniciantes.

11.2.1 Integração de pessoas colaboradoras

Não existe uma regra geral da rOpenSci sobre como você deve integrar pessoas colaboradoras. Você deve aumentar os direitos delas sobre o repositório à medida que ganhar confiança e, sem dúvida, deve reconhecer as contribuições que fizerem (consulte esta seção).

Você pode pedir a um novo colaborador que crie PRs (consulte a seção a seguir para avaliar um PR localmente, ou seja, além das verificações de CI) para dev/main e avalie-os antes de mesclá-los e, depois de algum tempo, deixe-os enviar para o main, embora você possa querer manter um sistema de revisões de PRs… mesmo para si mesmo quando tiver colegas de equipe!

Um modelo possível para a integração de pessoas colaboradoras é fornecido por Jim Hester em seu `lintr` repo.

Se o seu problema for o recrutamento de pessoas colaboradoras, você pode publicar uma chamada aberta como a de Jim Hester no Twitter, ou no GitHub e, como autor(a) de um pacote da rOpenSci, você pode pedir ajuda no slack da rOpenSci e pedir à equipe da rOpenSci por ideias para recrutar novas pessoas colaboradoras.

11.2.2 Trabalhando com colaboradores (incluindo você)

Branchs (ou ramificações) são baratas. Use-as extensivamente ao desenvolver recursos, testar novas ideias e corrigir problemas.

Uma das *branches* é a *branch* `main` (que é a ramificação padrão/principal), na qual, se você seguir desenvolvimento baseado no tronco você "faz *merge* de atualizações pequenas e frequentes". Veja também as documentações do Fluxo do GitHub e do fluxo do GitLab. Talvez você também queira incrementar frequentemente os números da versão de seu pacote (em `DESCRIPTION`). Um aspecto específico do trabalho com colaboradores é a revisão de *Pull requests*, com algumas orientações úteis em:

- The Art of Giving and Receiving Code Reviews (Gracefully) (A arte de dar e receber revisões de código (graciosamente)), de Alex Hill;
- Documentação do GitHub sobre revisões de PR.

Você pode querer mexer nas configurações do seu repositório do GitHub para, por exemplo, exigir revisões de *pull request* antes de fazer o *merge*. Consulte também os documentos do GitHub sobre "proprietários de código".

Para fazer e revisar *pull requests*, recomendamos que você que você explore essas funções.

Para que você configure o "git remote", consulte Happy Git and GitHub for the useR. Veja também a seção Useful Git patterns for real life no mesmo livro.

11.2.3 Seja generoso(a) com as atribuições

Se alguém contribuir para o seu repositório, considere adicioná-la em `DESCRIPTION`, como contribuinte ("ctb") para pequenas contribuições, e autor ("aut") para contribuições maiores. Tradicionalmente, ao citar um pacote em uma publicação científica, apenas as pessoas autoras "aut" são listados, e não as contribuidores "ctb"; e em `pkgdown` e, nos sites, somente os nomes "aut" são listados na página inicial, e todas as pessoas autoras são listados na página de autores.

No mínimo, considere adicionar o nome das pessoas colaboradoras próximo à linha de correção de *feature*/*bugs* em NEWS.md.

Recomendamos que você seja generoso(a) com esses agradecimentos, porque é uma coisa boa de se fazer e porque isso aumentará a probabilidade de as pessoas contribuírem novamente com o seu pacote ou com outros repositórios da organização.

Como um lembrete de nossas diretrizes de empacotamento se o seu pacote foi revisado e você acha que as pessoas revisoras fizeram uma contribuição substancial para o desenvolvimento do seu pacote, você pode listá-las na seção `Authors@R` com um tipo de contribuinte "Revisor" (`"rev"`), da seguinte forma:

```
person("Bea", "Hernández", role = "rev",
comment = "Bea reviewed the package (v. X.X.XX) for rOpenSci, see <https://github.com/ro
```

Inclua as pessoas revisoras somente após solicitar o consentimento delas. Leia mais nesta postagem do blog "Agradecendo seus revisores: Gratidão por meio de metadados semânticos". Observe que 'rev' gerará uma CRAN NOTE, a menos que o pacote seja criado usando o R v3.5. Certifique-se de que você use `roxygen2` na versão mais recente disponível no CRAN.

Por favor, não liste os editores como colaboradores. Sua participação e contribuição para a rOpenSci são agradecimentos suficientes!

11.2.4 Dando as boas-vindas aos colaboradores da rOpenSci

Se você conceder a alguém permissões de escrita no repositório,

- entre em contato com um membro da equipe para que esse novo colaborador possa receber **um convite para a organização da rOpenSci no GitHub** (em vez de ser um colaborador externo)
- entre em contato com a equipe da rOpenSci ou um outro membro da equipe para que esse novo colaborador possa receber **um convite para o *workspace* do Slack da rOpenSci**.

11.3 Outros recursos

- Chamada da comunidade rOpenSci: Configure seu pacote para promover uma comunidade.
- Para reutilizar respostas gentis e usuais, considere a funcionalidade de respostas salvas do GitHub.

12 Mudando os(as) mantenedores(as) de um pacote

Este capítulo apresenta um guia sobre como assumir a manutenção de um pacote.

12.1 Você quer desistir da manutenção do seu pacote?

Temos uma seção em nosso boletim informativo para chamadas de novos(as) colaboradores(as) que é publicado a cada duas semanas. A seção se chama *Chamada para colaboradores*. Nessa seção, destacamos os pacotes que estão à procura de novos(as) mantenedores(as). Se você deseja desistir da função de mantenedor(a) do seu pacote, entre em contato conosco e poderemos destacar o seu pacote em nosso boletim informativo.

12.2 Você quer assumir a manutenção de um pacote?

Temos uma seção em nosso boletim informativo para chamadas de novos(as) colaboradores(as) que é publicado a cada duas semanas. A seção se chama *Chamada para colaboradores*. Nessa seção, destacamos os pacotes que estão à procura de novos(as) mantenedores(as). Se você ainda não está inscrito no boletim informativo, é uma boa ideia assinar para que você seja notificado quando houver um pacote procurando por um novo(a) mantenedor(a).

12.3 Assumir a manutenção de um pacote

- Adicione você como o(a) novo(a) mantenedor(a) no arquivo DESCRIPTION, com `role = c("aut", "cre")`, e, altere o(a) antigo(a) mantenedor(a) para o papel `aut` apenas;
- Certifique-se de alterar o(a) mantenedor(a) para o seu nome em qualquer outro lugar do pacote, mantendo o(a) antigo(a) mantenedor(a) como autor (por exemplo, em manuais do pacote, CONTRIBUTING.md, CITATION etc.);
- Os Guia de Colaboração tem orientações sobre como adicionar novos(as) mantenedores(as) e colaboradores(as);

- Pacotes que foram arquivados ou órfãos no CRAN não precisam da permissão do(a) mantenedor(a) anterior para serem retomados no CRAN. Nesses casos, entre em contato conosco para que possamos oferecer a ajuda necessária;
- Se o pacote não tiver sido arquivado pelo CRAN e houver uma mudança de mantenedor(a), peça ao(à) mantenedor(a) antigo(a) que envie um e-mail ao CRAN e informe por escrito quem é o(a) novo(a) mantenedor(a). Certifique-se de mencionar esse e-mail sobre a mudança de mantenedor(a) quando você enviar a primeira nova versão ao CRAN. Se o(a) antigo(a) mantenedor(a) estiver inacessível ou não enviar esse e-mail, entre em contato com a equipe do rOpenSci;
- Se o(a) mantenedor(a) anterior estiver acessível, agendar uma reunião ajudará você a conhecer melhor a situação atual do pacote;

12.3.1 Perguntas frequentes para novos(as) mantenedores(as)

- Existem alguns problemas no pacote que ainda não foram resolvidos, e que eu não sei como corrigir. A quem posso pedir ajuda?

  Depende; aqui está o que você deve fazer em diferentes cenários:

  - Se for possível entrar em contato com o(a) antigo(a) mantenedor(a): entre em contato com ele(a) e peça ajuda;
  - Slack do rOpenSci: bom para obter ajuda em problemas específicos ou gerais, consulte o canal #package-maintenance;
  - Fórum de discussão do rOpenSci este forum é uma boa opção, sinta-se à vontade para fazer perguntas;
  - Equipe do rOpenSci sinta-se livre para entrar em contato com um de nós, seja por e-mail ou nos marcando em um problema no GitHub, ficaremos felizes em ajudar;
  - é claro, também existem vários centros de ajuda geral sobre o R, se isso atender às suas necessidades: Fórum da comunidade Posit, StackOverflow, Mastodon #rstats, etc.

- O quanto você pode/deve alterar no pacote?

  Para obter ajuda geral sobre como alterar o código em um pacote, consulte a seção Evolução do pacote de pacotes.

  Quando você pensa nisso, há muitas considerações a se fazer.

  Quanto tempo você tem para dedicar ao pacote? Se você tiver um tempo muito limitado, seria melhor se concentrar nas tarefas mais importantes, sejam elas quais forem para o pacote em questão. Se você tiver bastante tempo, suas metas poderão ter um escopo maior.

  Qual é o grau de maturidade do pacote? Se o pacote for maduro, muitas pessoas provavelmente têm códigos que dependem da API do pacote (ou seja, as funções exportadas e seus parâmetros). Além disso, se houver outros pacotes que dependam do seu pacote no CRAN, você precisa verificar se as suas alterações vão quebrar ou não esses outros pacotes. Quanto

mais maduro for o pacote, mais cuidadoso(a) você precisará ser ao fazer alterações, especialmente com os nomes das funções exportadas, seus parâmetros e a estrutura exata do que as funções exportadas retornam. É mais fácil fazer alterações que afetem apenas os aspectos internos do pacote.

12.4 Tarefas da equipe do rOpenSci

Como organização, a rOpenSci está interessada em garantir que os pacotes de nossa suíte estejam disponíveis enquanto forem úteis para a comunidade. Como os(as) mantenedores(as) precisam seguir em frente, na maioria dos casos tentaremos conseguir um(a) novo(a) mantenedor(a) para cada pacote. Para isso, as tarefas a seguir são de responsabilidade da equipe da rOpenSci.

- Se um repositório não tiver nenhuma atividade (commits, issues, pull requests) há muito tempo, ele pode ser simplesmente um pacote maduro com pouca necessidade de alterações/etc., portanto, leve isso em consideração.
- O(a) mantenedor(a) atual não responde a problemas/solicitações pull há muitos meses, por meio de e-mails, problemas no GitHub ou mensagens no Slack:
  - Entre em contato e veja qual é a situação. Ele(a) pode dizer que gostaria de deixar o cargo de mantenedor(a) e, nesse caso, procure um(a) novo(a) mantenedor(a)
- O(a) mantenedor(a) atual está completamente ausente/não está respondendo
  - Se isso acontecer, tentaremos entrar em contato com o(a) mantenedor(a) por até um mês. No entanto, se a atualização do pacote for urgente, poderemos usar nosso acesso de administrador para fazer alterações em seu nome.
- Faça uma chamada na seção “Call for Contributors” do boletim informativo da rOpenSci para um(a) novo(a) mantenedor(a) - abra um problema no repositório do boletim informativo.

13 Publicação de um pacote

Seu pacote deve ter versões diferentes ao longo do tempo: *snapshots* de um estado do pacote que você pode publicar no CRAN, por exemplo. Essas versões devem ser devidamente *numeradas*, *publicadas* e *descritas em um arquivo NEWS*. Mais detalhes abaixo.

Observe que você pode simplificar o processo de atualização do NEWS e do controle de versão do seu pacote usando [o pacote `fledge`] (https://github.com/cynkra/fledge).

13.1 Controle de versão

- Recomendamos enfaticamente que os pacotes rOpenSci usem controle de versão semântico. Uma explicação detalhada está disponível no capítulo de ciclo de vida do livro *R Packages*.

13.2 Publicação

- Usando `usethis::use_release_issue()` e `devtools::release()` ajudará você a se lembrar de mais verificações.
- Marque cada versão no Git após cada envio ao CRAN. Mais informações
- O CRAN não gosta de atualizações muito frequentes. Dito isso, se você notar um grande problema uma semana após o lançamento do CRAN, explique-o no cran-comments.md e tente lançar uma versão mais recente.

13.3 Arquivo de notícias

Um arquivo NEWS que descreve as alterações associadas a cada versão facilita para os usuários verem o que está mudando no pacote e como isso pode afetar seu fluxo de trabalho. Você deve adicionar um para o seu pacote e torná-lo fácil de ler.

- É obrigatório usar um arquivo `NEWS` ou `NEWS.md` na raiz do seu pacote. Recomendamos que você use o arquivo `NEWS.md` para tornar o arquivo mais navegável.

- Por favor, use nosso exemplo Arquivo NEWS como modelo. Você pode encontrar um bom arquivo NEWS na natureza na seção `taxize` repositório de pacotes por exemplo.
- Se você usar `NEWS` adicione-o a `.Rbuildignore` mas não se você usar `NEWS.md`
- Atualize o arquivo de notícias antes de cada lançamento do CRAN, com uma seção com o nome do pacote, a versão e a data de lançamento, como (como visto em nosso exemplo) Arquivo NEWS):

```
foobar 0.2.0 (2016-04-01)
=========================
```

- Sob esse cabeçalho, coloque as seções conforme necessário, incluindo: `NEW FEATURES`, `MINOR IMPROVEMENTS`, `BUG FIXES`, `DEPRECATED AND DEFUNCT`, `DOCUMENTATION FIXES` e qualquer título especial que agrupe um grande número de alterações. Em cada cabeçalho, liste os itens conforme necessário (como visto em nosso exemplo Arquivo NEWS). Para cada item, forneça uma descrição do novo recurso, melhoria, correção de bug ou função/característica obsoleta. Link para qualquer problema relacionado no GitHub, como `(#12)`. O problema `(#12)` será resolvido no GitHub em Releases para um link para esse problema no repositório.
- Depois que você tiver adicionado um `git tag` e enviado para o GitHub, adicione os itens de notícias para essa versão marcada às notas de versão de uma versão no seu repositório do GitHub com um título como `pkgname v0.1.0`. Você pode ver Documentos do GitHub sobre a criação de uma versão.
- Novos lançamentos do CRAN serão escritos sobre em nosso boletim informativo mas veja próximo capítulo sobre marketing sobre como informar mais usuários em potencial sobre o lançamento.
- Para obter mais orientações sobre o arquivo NEWS, sugerimos que você leia o documento guia de estilo do tidyverse NEWS.

14 Marketing do seu pacote

Ajudaremos você a promover o seu pacote, mas aqui estão mais algumas coisas que você deve ter em mente.

- Se você souber de um caso de uso de seu pacote, incentive o autor a publicar o link em nosso fórum de discussão na categoria *Use Cases*, para um post no Mastodon da rOpenSci e possível inclusão no boletim quinzenal da rOpenSci. Também recomendamos que você adicione um link para o caso de uso em uma seção "use cases in the wild" do seu README.
- Quando você liberar uma nova versão do seu pacote ou você lançá-lo pela primeira vez no CRAN,
  - Faça um *pull request* para a R Weekly com uma linha sobre essa nova versão na seção "New Releases" (ou "New Packages" para a primeira versão do GitHub/CRAN).
  - Tweet sobre isso usando a hashtag "#rstats" e marque rOpenSci! Isso pode ajudar no envolvimento do colaborador/usuário. Exemplo.
  - Considere a possibilidade de enviar um breve post sobre o lançamento para as Notas técnicas da rOpenSci. Entre em contato com o/a gerente da comunidade do rOpenSci (por exemplo, via Slack ou info@ropensci.org). Consulte as diretrizes sobre como contribuir com um *blog post*).
  - Envie seu pacote para listas de pacotes, como a Visão de tarefas do CRAN.
- Se você optar por divulgar o seu pacote dando uma palestra sobre ele em um encontro ou conferência (excelente ideia!) leia este artigo de Jenny Bryan e Mara Averick.

15 GitHub Grooming

rOpenSci packages are currently in their vast majority developed on GitHub. Here are a few tips to leverage the platform in a section about making your repo more discoverable and a section about marketing your own GitHub account after going through peer review.

15.1 Make your repository more discoverable

15.1.1 GitHub repo topics

GitHub repo topics help browsing and searching GitHub repos, are used by R-universe on package pages and for search results, and are digested by `codemetar` for rOpenSci registry keywords.

We recommend:

- Adding "r", "r-package" and "rstats" as topics to your package repo.
- Adding any other relevant topics to your package repo.

We might make suggestions to you after your package is onboarded.

15.1.2 GitHub linguist

GitHub linguist will assign a language for your repo based on the files it contains. Some packages containing a lot of C++ code might get classified as C++ rather than R packages, which is fine and shows the need for the "r", "r-package" and "rstats" topics.

We recommend overriding GitHub linguist by adding or modifying a .gitattributes to your repo in two cases:

- If you store html files in non standard places (not in docs/, e.g. in vignettes/) use the documentation overrides. Add `*.html linguist-documentation=true` to .gitattributes (Example in the wild)
- If your repo contains code you haven't authored, e.g. JavaScript code, add `inst/js/* linguist-vendored` to .gitattributes (Example in the wild)

This way the language classification and statistics of your repository will more closely reflect the source code it contains, as well as making it more discoverable. Notably, if linguist does not correctly recognize your repo as containing mainly R code, your package won't appear in search results with the `language:R` filter. Similarly, your repo cannot be listed among the trending R repos.

More info about GitHub linguist overrides here.

15.2 Market your own account

- As the author of an onboarded package, you are now a member of rOpenSci's "ropensci" GitHub organization. By default, organization memberships are private; see how to make it public in GitHub docs.
- Even after your package repo has been transferred to rOpenSci, you can pin it under your own account.
- In general we recommend adding at least an avatar (which doesn't need to be your face!) and your name to your GitHub profile.

16 Package evolution - changing stuff in your package

This chapter presents our guidance for changing stuff in your package: changing parameter names, changing function names, deprecating functions, and even retiring and archiving packages.

This chapter was initially contributed as a tech note on rOpenSci website by Scott Chamberlain; you can read the original version here.

16.1 Philosophy of changes

Everyone's free to have their own opinion about how freely parameters/functions/etc. are changed in a library - rules about package changes are not enforced by CRAN or otherwise. Generally, as a library gets more mature, changes to user facing methods (i.e., exported functions in an R package) should become very rare. Libraries that are dependencies of many other libraries are likely to be more careful about changes, and should be.

16.2 The lifecycle package

This chapter presents solutions that do not require the lifecycle package but you might still find it useful. We recommend reading the lifecycle documentation.

16.3 Parameters: changing parameter names

Sometimes parameter names must be changed for clarity, or some other reason.

A possible approach is check if deprecated arguments are not missing, and stop providing a meaningful message.

```
foo_bar <- function(x, y) {
    if (!missing(x)) {
        stop("use 'y' instead of 'x'")
    }
    y^2
}

foo_bar(x = 5)
#> Error in foo_bar(x = 5) : use 'y' instead of 'x'
```

If you want to be more helpful, you could emit a warning but automatically take the necessary action:

```
foo_bar <- function(x, y) {
    if (!missing(x)) {
        warning("use 'y' instead of 'x'")
        y <- x
    }
    y^2
}

foo_bar(x = 5)
#> 25
```

Be aware of the parameter `...`. If your function has `...`, and you have already removed a parameter (lets call it `z`), a user may have older code that uses `z`. When they pass in `z`, it's not a parameter in the function definition, and will likely be silently ignored – not what you want. Instead, leave the argument around, throwing an error if it used.

16.4 Functions: changing function names

If you must change a function name, do it gradually, as with any other change in your package.

Let's say you have a function `foo`.

```
foo <- function(x) x + 1
```

However, you want to change the function name to `bar`.

Instead of simply changing the function name and `foo` no longer existing straight away, in the first version of the package where `bar` appears, make an alias like:

```
#' foo - add 1 to an input
#' @export
foo <- function(x) x + 1

#' @export
#' @rdname foo
bar <- foo
```

With the above solution, the user can use either `foo()` or `bar()` – either will do the same thing, as they are the same function.

It's also useful to have a message but then you'll only want to throw that message when they use the old function, e.g.,

```
#' foo - add 1 to an input
#' @export
foo <- function(x) {
    warning("please use bar() instead of foo()", call. = FALSE)
    bar(x)
}

#' @export
#' @rdname foo
bar <- function(x) x + 1
```

After users have used the package version for a while (with both `foo` and `bar`), in the next version you can remove the old function name (`foo`), and only have `bar`.

```
#' bar - add 1 to an input
#' @export
bar <- function(x) x + 1
```

16.5 Functions: deprecate & defunct

To remove a function from a package (let's say your package name is `helloworld`), you can use the following protocol:

- Mark the function as deprecated in package version `x` (e.g., `v0.2.0`)

In the function itself, use `.Deprecated()` to point to the replacement function:

```
foo <- function() {
    .Deprecated("bar")
}
```

There's options in `.Deprecated` for specifying a new function name, as well as a new package name, which makes sense when moving functions into different packages.

The message that's given by `.Deprecated` is a warning, so can be suppressed by users with `suppressWarnings()` if desired.

Make a man page for deprecated functions like:

```
#' Deprecated functions in helloworld
#'
#' These functions still work but will be removed (defunct) in the next version.
#'
#' \itemize{
#'  \item \code{\link{foo}}: This function is deprecated, and will
#'  be removed in the next version of this package.
#' }
#'
#' @name helloworld-deprecated
NULL
```

This creates a man page that users can access like `` ?`helloworld-deprecated` `` and they'll see in the documentation index. Add any functions to this page as needed, and take away as a function moves to defunct (see below).

- In the next version (`v0.3.0`) you can make the function defunct (that is, completely gone from the package, except for a man page with a note about it).

In the function itself, use `.Defunct()` like:

```
foo <- function() {
    .Defunct("bar")
}
```

Note that the message in `.Defunct` is an error so that the function stops whereas `.Deprecated` uses a warning that let the function proceed.

In addition, it's good to add `...` to all defunct functions so that if users pass in any parameters they'll get the same defunct message instead of a `unused argument` message, so like:

```
foo <- function(...) {
    .Defunct("bar")
}
```

Without `...` gives:

```
foo(x = 5)
#> Error in foo(x = 5) : unused argument (x = 5)
```

And with `...` gives:

```
foo(x = 5)
#> Error: 'foo' has been removed from this package
```

Make a man page for defunct functions like:

```
#' Defunct functions in helloworld
#'
#' These functions are gone, no longer available.
#'
#' \itemize{
#'  \item \code{\link{foo}}: This function is defunct.
#' }
#'
#' @name helloworld-defunct
NULL
```

This creates a man page that users can access like `` ?`helloworld-defunct` `` and they'll see in the documentation index. Add any functions to this page as needed. You'll likely want to keep this man page indefinitely.

16.5.1 Testing deprecated functions

You don't have to change the tests of deprecated functions until they are made defunct.

- Consider any changes made to a deprecated function. Along with using `.Deprecated` inside the function, did you change the parameters at all in the deprecated function, or create a new function that replaces the deprecated function, etc. Those changes should be tested if any made.

- Related to above, if the deprecated function is simply getting a name change, perhaps test that the old and new functions return identical results.
- `suppressWarnings()` could be used to suppress the warning thrown from `.Deprecated`, but tests are not user facing, so it is not that bad if the warning is thrown in tests, and the warning could even be used as a reminder to the maintainer.

Once a function is made defunct, its tests are simply removed.

16.6 Archiving packages

Software generally has a finite lifespan, and packages may eventually need to be archived. Archived packages are archived and moved to a dedicated GitHub organization, ropensci-archive. Prior to archiving, the contents of the README file should be moved to an alternative location (such as "README-OLD.md"), and replaced with minimal contents including something like the following:

```
# <package name>

[![Project Status: Unsupported](https://www.repostatus.org/badges/latest/unsupported.svg)](ht
[![Peer-review badge](https://badges.ropensci.org/<issue_number>_status.svg)](https://github

This package has been archived. The former README is now in [README-old](<link-to-README-old
```

The repo status badge should be "unsupported" for formerly released packages, or "abandoned" for former concept or WIP packages, in which case the badge code above should be replaced with:

```
[![Project Status: Abandoned](https://www.repostatus.org/badges/latest/abandoned.svg)](https
```

An example of a minimal README in an archived package is in ropensci-archive/monkeylearn. Once the README has been copied elsewhere and reduced to minimal form, the following steps should be followed:

- ☐ Close issues with a sentence explaining the situation and linking to this guidance.
- ☐ Archive the repository on GitHub (also under repo settings).
- ☐ Transfer the repository to ropensci-archive, or request an rOpenSci staff member to transfer it (you can email `info@ropensci.org`).

Archived packages may be unarchived if authors or a new person opt to resume maintenance. For that please contact rOpenSci. They are transferred to the ropenscilabs organization.

17 Package Curation Policy

This chapter summarizes a proposed curation policy for rOpenSci's ongoing maintenance of packages developed as part of rOpenSci activities and/or under the rOpenSci GitHub organization. This curation policy aims to support these goals:

- Ensure packages provided by rOpenSci are up-to-date and high quality
- Provide clarity as to the development status and and review status of any software in rOpenSci repositories
- Manage maintenance effort for rOpenSci staff, package authors, and volunteer contributors
- Provide a mechanism to gracefully sunset packages while maintaining peer-review badging

Elements of infrastructure described below needed for implementation of the policy are in some cases partly built and in other cases not yet begun. We aim to adopt this policy in part to prioritize work on these components.

17.1 The package registry

- The rOpenSci package registry is a central listing of R packages currently or formerly maintained or reviewed by rOpenSci. It contains essential package metadata including development and review status, and will be the source of data for display on websites, badges, etc. It will allow this listing to be maintained independently of package or infrastructure hosting platforms.

17.2 Staff-maintained packages

Staff-maintained packages are developed and maintained by rOpenSci staff as part of rOpenSci projects. These packages may also be peer-reviewed packages, but are not necessarily peer reviewed. Many are infrastructure packages that fall out of scope for peer review.

- Staff-maintained packages will be listed in the registry with tag "staff_maintained" and listed on rOpenSci's packages web page or similar venues with tag "staff-maintained"
- These packages will be stored in the "ropensci" GitHub organization
- Staff-maintained packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross).
- When the packages fail checks, rOpenSci staff will endeavor to fix changes, prioritizing packages based on user base (downloads), reverse dependencies, or strategic goals.
- On a biannual or annual basis, rOpenSci will review all packages that have been failing for over a month to determine whether to transfer them to the "ropensci-archive" GitHub organization.
- Packages consistently failing and without an ongoing plan to return to active maintenance will move to "archive" status. When archived, staff packages will move to the "ropensci-archive" repository (to be created) and and gain the "archived" type in the registry. They will not be built on rOpenSci system.
- Archived packages will not be displayed by default on the packages web page. A special tab of packages pages will display these with `"type": "archived"` that were either peer-reviewed or staff-maintained.
- Archived packages can be unarchived when the old or a new maintainer is willing to address the problems and wants to revive the package. For that please contact rOpenSci. They are transferred to the ropenscilabs organization.

17.3 Peer-reviewed packages

Peer-reviewed packages are those contributed to the rOpenSci by the community and have passed through peer review. They need to be in-scope at the time of submission to be reviewed.

- Upon acceptance, these peer-reviewed packages are transferred from the author's GitHub to the "ropensci" GitHub organization
- Peer-reviewed packages will be in the registry tagged as "peer-reviewed" and have a peer-reviewed badge in their README.
- Peer-reviewed packages will be listed on rOpenSci's web page or similar venues with tag "peer-reviewed"
- Peer-reviewed packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross).

- Annually or bi-annually, rOpenSci staff will review packages in a failing state or have been failing for extended periods, and contact the authors to determine ongoing maintenance status and expected updates. Based on this exchange, rOpenSci may opt to retain the package's current status with the expectation of an updates, contribute support or seek a new maintainer, or transfer the package to "archived" status.
- Based on user base (measured by downloads), reverse dependencies, or rOpenSci strategic goals, rOpenSci staff may support failing packages via PRs reviewed by package authors, or direct changes (if authors are unresponsive for approximately a month). rOpenSci will also provide support to package authors on request, both by staff and community volunteers, based on time available.
- At the author's request, or if authors are unresponsive to inquiries for approximately a month, rOpenSci may seek a new maintainer for select peer-reviewed packages it deems have high community value based on user base/downloads, reverse dependencies, or rOpenSci strategic goals.
- When archived, these packages will move from the "ropensci" GitHub organization to the "ropensci-archive" organization (or author GitHub accounts if desired), following transfer guidance. They will gain the "archived" type in the registry. They will retain "peer-reviewed" tags and badges. They will not be built on rOpenSci system.
- Archived packages will not be displayed by default. A special tab of packages pages will display these with `"type": "archived"` that were either peer-reviewed or staff-maintained.

17.4 Legacy acquired packages

"Legacy" packages are packages not created or maintained by rOpenSci staff and not peer reviewed, but are under the rOpenSci GitHub organization(s) due to historical reasons. (Prior to establishing the peer review process and its scope, rOpenSci absorbed packages from various developers without well-defined criteria.)

- rOpenSci will transfer legacy packages back to author organizations and repositories. If authors are uninterested, we will transfer them to the "ropensci-archive" repository following transfer guidance. If packages are in-scope, rOpenSci will inquire if authors would like to submit them to the Software Review process.
- Legacy packages will not be listed in the package registry.
- Exceptions may be made for packages that are vital parts of the R and/or rOpenSci package ecosystem which are actively monitored by staff.

17.5 Incubator packages

"Incubator" packages are in-development packages created by staff or community members as part of community projects, such as those created at unconferences

- Incubator packages will live in the "ropenscilabs" organization.
- Incubator packages will appear in the package registry with the "incubator" tag
- Incubator packages will not appear on the website by default, but packages pages will include an "experimental packages" tab.
- Incubator packages and their docs will be built by rOpenSci system. This system does not send notifications but it outputs results as GitHub commit status (red check mark or red cross). The docs will indicate clearly the package is experimental.
- Biannually or annually, rOpenSci will contact incubator maintainers about repositories at least three months old, inquiring about development status and author preferences for migration to peer-review, ropensci-archive, or to author organizations. Based on response, package will be migrated immediately, peer review will be initiated, or migration will be deferred to the next review. Incubator packages will be migrated to ropensci-archive by default after one year, following transfer guidance.
- Archived incubator packages will gain the "archived" type.

17.5.1 Incubator non-R-packages

- The "incubator" organization will also include non-R-package projects.
- These projects will not be listed in the registry or appear on a web page, and will not be automatically built.
- Migration policy will be the same as for R packages, with appropriate migration locations (e.g., ropensci-books)
- If archived, non-R-packages will move to "ropensci-archive" following transfer guidance.

17.6 Books

rOpenSci books are long-form documentation, often bookdown-formatted, related to rOpenSci packages, projects, or themes, created by both rOpenSci staff and community members.

- Books will live in the "ropensci-books" organization
- Books will be hosted at books.ropensci.org

- Books may be mature or in-development, but must have minimal outlines/content before migrating into “ropensci-books” (e.g. from “ropenscilabs”).
- The authorship and development status of a book should be clearly described on its home page and README.
- rOpenSci may provide badges or templates (e.g., “In development,” “Community Maintained,”) for authors to use on book home pages in the future

18 Guia de contribuição

Este capítulo apresenta o nosso Guia de Contribuição, que descreve como você pode fazer contribuições com código e sem código para a rOpenSci.

- Então você quer contribuir para a rOpenSci? Fantástico! Nós desenvolvemos o Guia de contribuição da comunidade rOpenSci para dar as boas-vindas a você na rOpenSci e te ajudar a se reconhecer como um(a) colaborador(a) em potencial. O guia ajudará você a descobrir o que você pode ganhar doando seu tempo, conhecimento e experiência, combinando suas necessidades com coisas que ajudarão a missão da rOpenSci, e conectando você a recursos que podem te ajudar ao longo do caminho.

Nossa equipe e comunidade promovem ativamente um ambiente acolhedor em que pessoas usuárias e desenvolvedoras de diferentes origens e níveis de habilidade aprendem, compartilham ideias e inovam juntas abertamente por meio de normas e software compartilhados. A participação em todas as atividades da rOpenSci é apoiada por nosso Código de Conduta.

Aceitamos contribuições com código e sem código de pessoas programadoras novas ou experientes em qualquer estágio da carreira e em qualquer setor. Você não precisa ser um(a) desenvolvedor(a)! Talvez você queira **passar 30 minutos** compartilhando o caso de uso do seu pacote em nosso fórum público ou relatando um bug, **uma hora** aprendendo e participando de uma chamada da comunidade, **cinco horas** revisando um pacote de R enviado para revisão aberta por pares, **ou talvez você queira assumir um compromisso contínuo** para ajudar a manter um pacote.

Quais são alguns dos benefícios de contribuir?

- Conectar-se com uma comunidade que compartilha do seu interesse em tornar a ciência mais aberta
- Aprenda com pessoas de fora de seu domínio que usam o R com desafios semelhantes aos seus
- Faça e responda a novas perguntas de pesquisa conhecendo novas ferramentas de software e aliados
- Sentir-se confiante e com apoio em seus esforços para escrever código e desenvolver software
- Ganhar visibilidade para seu trabalho de código aberto
- Melhorar o software que você usa ou constrói
- Aprimore suas habilidades em R e ajude outras pessoas a aprimorarem as delas
- Aumente o nível de suas habilidades de escrita
- Obter mais exposição para o seu pacote

Consulte nosso Guia de contribuição e navegue pela seção “O que traz você aqui?” para saber quais são os *Eu quero …* que se encaixam melhor em você e escolha o seu caminho! Para ajudar você a se reconhecer, nós as agrupamos em: Descobrir; Conectar; Aprender; Construir; Ajudar. Para cada categoria, listamos exemplos de como essas contribuições podem ser e colocamos links para os nossos recursos para obter os detalhes de que você precisa.

Parte IV

Appendix

19 NEWS

19.1 0.9.0

- 2024-01-09, update roxygen2 wording (`@vincentvanhees`, #792).
- 2023-12-15, update roxygen2 advice, mainly linking to roxygen2 website (#750).
- 2023-09-15, add suggestions for API packages (#496).
- Translation to Spanish!
- 2023-07-17, Update Aims and Scope to include translation packages, remove experimental text-processing categories, and provide clarifications around API wrappers
- 2023-05-04, Added link to Bioconductor book (#663, `@llrs`).
- 2023-04-26, Changed suggested lifecycle stage in authors guide (#661, `@bart1`).
- 2023-04-25, changed the COI section to use parallel construction (#659, `@eliocamp`).
- 2022-07-04, Add resources around GitHub workflows (#479, `@maurolepore`).
- 2023-02-14, update instructions for CITATION to reflect new CRAN policies (#604, #609).
- 2023-02-14, add package maintainer cheatsheet (#608).
- 2023-01-25, add Mastodon as social media (#592, by @yabellini).
- 2023-01-25, add Mastodon as social media (#592, by `@yabellini`).
- 2023-01-20, fix small formating error (#590 by `@eliocamp`).
- 2022-11-22, mention shinytest2 near shinytest.
- 2022-09-20, add editor instruction to add “stats” label to stats submissions
- 2022-09-20, fixed link to reviewer approval template (#548), and rendering of editor’s template (#547)
- 2022-08-23, add recommendation to document argument default (`@Bisaloo`, #501)
- 2022-08-06, fix link to R Packages book (#498)
- 2022-07-21, mention GitHub Discussions and GitHub issue templates. (#482)

- 2022-07-21, highlight values for reviewing in more places (#481)
- 2022-07-20, Explanation of package submission via non-default branches (#485), added `@s3alfisc` to contributor list.
- 2022-07-20, add how to volunteer as a reviewer (#457).
- 2022-06-23, Expanded explanation of Codecov, added `@ewallace` to contributor list (#484)

19.2 0.8.0

- 2022-06-03, Remove former references to now-archived "rodev" package
- 2022-05-30, Advise that reviewers can also directly call `@ropensci-review-bot check package`
- 2022-05-27, Add Mark Padgham to list of authors
- 2022-05-27, Add `devguider::prerelease_checklist` item to pre-release template (#463)
- 2022-05-13, Align version number in DESCRIPTION file with actual version (#443)
- 2022-05-13, Update guidelines for CONTRIBUTING.md (#366, #462)
- 2022-05-09, Add section on authorship of included code, thanks to `@KlausVigo` (#388).
- 2022-05-09, Remove mention of 'rev' role requiring R v3.5
- 2022-05-05, Move all scripts from local `inst` directory to `ropensci-org/devguider` pkg.
- 2022-05-03, Update package archiving guidance to reduce README to minimal form.
- 2022-04-29, Advise that authors can directly call `@ropensci-review-bot check package`.
- 2022-04-29, Describe `pkgcheck-action` in CI section.
- 2022-04-29, Update scope in policies section to include statistical software.
- 2022-04-29, Add `prelease.R` script to open pre-release GitHub issue & ref in appendix.
- 2022-04-26, Add GitHub 2FA recommendation to package security.
- 2022-03-29, Remove references to Stef Butland, former community manager.
- 2022-03-28, Add comments on submission planning about time commitment.
- 2022-03-24, Remove approval comment template (coz it's automatically generated by the bot now).
- 2022-03-21, rephrase CITATION guidance to make it less strict. Also mentions CITATION.cff and the cffr package.

- 2022-03-08, add links to blogs related to package development (#389).
- 2022-02-17, update redirect instructions (`@peterdesmet`, #387).
- 2022-02-14, link to Michael Lynch's post Why Good Developers Write Bad Unit Tests.
- 2022-02-14, mention more packages for testing like dittodb, vcr, httptest, httptest2, webfakes.
- 2022-01-10, make review templates R Markdown files (`@Bisaloo`, #340).
- 2022-01-14, update guidance on CI services (#377)
- 2022-01-11, update guidance around branches, with resources suggested by `@ha0ye` and `@statnmap`.
- 2022-01-10, divide author's guide into sub-sections, and add extra info including `pkgcheck`.
- 2021-11-30, adds links to examples of reviews, especially tough but constructive ones (with help from `@noamross`, `@mpadge`, #363).
- 2021-11-19, add recommended spatial packages to scaffolding section (software-review-meta#47)
- 2021-11-18, update advice on grouping functions for pkgdown output (#361)

19.3 0.7.0

- 2021-11-04, add mentions of stat software review to software review intro and to the first book page (#342).
- 2021-11-04, mention pkgcheck in the author guide (@mpadge, #343).
- 2021-11-04, add editors' responsibilities including Editor etiquette for commenting on packages on which you aren't handling/reviewing (@jhollist, #354).
- 2021-11-04, give precise examples of tools for installation instructions (remotes, pak, R-universe).
- 2021-11-04, add more bot guidance (less work for editors).
- 2021-10-07, add guidance for editorial management (recruiting, inviting, onboarding, offboarding editors).
- 2021-09-14, add a requirement that there is at least one *HTML* vignette.
- 2021-09-03, add some recommendations around git. (@annakrystalli, #341)
- 2021-07-14, clarify the categories data extraction and munging by adding examples. (@noamross, #337)

- 2021-05-20, add guidance around setting up your package to foster a community, inspired by the recent rOpenSci community call. (with help from `@Bisaloo`, #289, #308)
- 2021-04-27, no longer ask reviewers to ask covr as it'll be done by automatic tools, but ask them to pay attention to tests skipped.
- 2021-04-02, add citation guidance.
- 2021-04-02, stop asking reviewers to run goodpractice as this is part of editorial checks.
- 2021-03-23, launched a new form for reviewer volunteering.
- 2021-02-24, add guidance around the use of `@ropensci-review-bot`.

19.4 0.6.0

- 2021-02-04, add guidance to enforce package versioning and tracking of changes through review (`@annakrystalli`, #305)
- 2021-01-25, add a translation of the review template in Spanish (`@Fvd`, `@maurolepore`, #303)
- 2021-01-25, the book has now better citation guidance in case you want to cite this very guide (`@Bisaloo`, #304).
- 2021-01-12, add some more guidance on escaping examples (#290).
- 2021-01-12, mention the lifecycle package in the chapter about package evolution (#287).
- 2021-01-12, require overlap information is put in documentation (#292).
- 2021-01-12, start using the bookdown::bs4_book() template.
- 2021-01-12, add a sentence about whether it is acceptable to push a new version of a package to CRAN within two weeks of the most recent version if you have just been made aware of, and fixed, a major bug (`@sckott`, #283)
- 2021-01-12, mention the HTTP testing in R book.
- 2021-01-12, mention testthat snapshot tests.
- 2021-01-12, remove mentions of Travis CI and link to Jeroen Ooms' blog post about moving away from Travis.
- 2021-01-12, update the package curation policy: mention a possible exception for legacy packages that are vital parts of the R and/or rOpenSci package ecosystem which are actively monitored by staff. (`@noamross`, #293)

19.5 0.5.0

- 2020-10-08, add help about link checking (@sckott, #281)
- 2020-10-08, update JOSS instructions (@karthik, #276)
- 2020-10-05, add links to licence resources (@annakrystalli, #279)
- 2020-10-05, update information about the contributing guide (@stefaniebutland, #280)
- 2020-09-11, make reviewer approval a separate template (@bisaloo, #264)
- 2020-09-22, add package curation policy (@noamross, #263)
- 2020-09-11, add more guidance and requirements for docs at submission (@annakrystalli, #261)
- 2020-09-14, add more guidance on describing data source in DESCRIPTION (@mpadge, #260)
- 2020-09-14, add more guidance about tests of deprecated functions (@sckott, #213)
- 2020-09-11, update the CI guidance (@bisaloo, @mcguinlu, #269)
- 2020-09-11, improve the redirect guidance (@jeroen, @mcguinlu, #269)

19.6 0.4.0

- 2020-04-02, give less confusing code of conduct guidance: the reviewed packages' COC is rOpenSci COC (@Bisaloo, @cboettig, #240)
- 2020-03-27, add section on Ethics, Data Privacy and Human Subjects Research to Policies chapter
- 2020-03-12, mention GitHub Actions as a CI provider.
- 2020-02-24, add guide for inviting a guest editor.
- 2020-02-14, add mentions of the ropensci-books GitHub organisation and associated subdomain.
- 2020-02-10, add field and laboratory reproducibility tools as a category in scope.
- 2020-02-10, add more guidance about secrets and package development in the security chapter.
- 2020-02-06, add guidance about Bioconductor dependencies (#246).
- 2020-02-06, add package logo guidance (#217).
- 2020-02-06, add one CRAN gotcha: single quoting software names(#245, @aaronwolen)

- 2020-02-06, improve guidance regarding the replacement of “older” pkgdown website links and source (#241, @cboettig)
- 2020-02-06, rephrase the EiC role (#244).
- 2020-02-06, remove the recommendation to add rOpenSci footer (https://github.com/ropensci/software-review-meta/issues/79).
- 2020-02-06, remove the recommendation to add a review mention to DESCRIPTION but recommends mentioning the package version when reviewers are added as “rev” authors.
- 2020-01-30, slightly changes the advice on documentation re-use: add a con; mention `@includeRmd` and `@example`; correct the location of Rmd fragments (#230).
- 2020-01-30, add more guidance for the editor in charge of a dev guide release (#196, #205).
- 2020-01-22, add guidance in the editor guide about not transferred repositories.
- 2020-01-22, clarify forum guidance (for use cases and in general).
- 2020-01-22, mention an approach for pre-computing vignettes so that the pkgdown website might get build on rOpenSci docs server.
- 2020-01-22, document the use of mathjax with rotemplate (@Bisaloo, #199).
- 2020-01-20, add guidance for off-thread interaction and COIs (@noamross, #197).
- 2020-01-20, add advice on specifying dependency minimum versions (@karthik, @annakrystalli, #185).
- 2020-01-09, start using GitHub actions instead of Travis for deployment.
- -2019-12-11, add note in Documentation sub-section of Packaging Guide section about referencing the new R6 support in roxygen2 (ropensci/dev_guide#189)
- 2019-12-11, add new CRAN gotcha about having ‘in R’ or ‘with R’ in your package title (@bisaloo, ropensci/dev_guide#221)

19.7 0.3.0

- 2019-10-03, include in the approval template that maintainers should include link to the docs.ropensci.org/pkg site (ropensci/dev_guide#191)
- 2019-09-26, add instructions for handling editors to nominate packages for blog posts (ropensci/dev_guide#180)
- 2019-09-26, add chapter on changing package maintainers (ropensci/dev_guide#128) (ropensci/dev_guide#194)

- 2019-09-26, update Slack room to use for editors (ropensci/dev_guide#193)
- 2019-09-11, update instructions in README for rendering the book locally (ropensci/dev_guide#192)
- 2019-08-05, update JOSS submission instructions (ropensci/dev_guide#187)
- 2019-07-22, break "reproducibility" category in policies into component parts. (ropensci/software-review-meta#81)
- 2019-06-18, add link to rOpenSci community call "Security for R" to security chapter.
- 2019-06-17, fix formatting of Appendices B-D in the pdf version of the book (bug report by `@IndrajeetPatil`, #179)
- 2019-06-17, add suggestion to use R Markdown hunks approach when the README and the vignette share content. (ropensci/dev_guide#161)
- 2019-06-17, add mention of central building of documentation websites.
- 2019-06-13, add explanations of CRAN checks. (ropensci/dev_guide#177)
- 2019-06-13, add mentions of the `rodev` helper functions where relevant.
- 2019-06-13, add recommendation about using `cat` for `str.*()` methods. RStudio assumes that `str` uses `cat`, if not when loading an R object the `str` prints to the console in RStudio and doesn't show the correct object structure in the properties. ([`@mattfidler`] (https://github.com/mattfidler/) #178)
- 2019-06-12, add more details about git flow.
- 2019-06-12, remove recommendation about `roxygen2` dev version since the latest stable version has what is needed. (`@bisaloo`, #165)
- 2019-06-11, add mention of usethis functions for adding testing or vignette infrastructure in the part about dependencies in the package building guide.
- 2019-06-10, use the new URL for the dev guide, https://devguide.ropensci.org/
- 2019-05-27, add more info about the importance of the repo being recognized as a R package by linguist (`@bisaloo`, #172)
- 2019-05-22, update all links eligible to HTTPS and update links to the latest versions of Hadley Wickham and Jenny Bryan's books (`@bisaloo`, #167)
- 2019-05-15, add book release guidance for editors. (ropensci/dev_guide#152)

19.8 0.2.0

- 2019-05-23, add CRAN gotcha: in the Description field of your DESCRIPTION file, enclose URLs in angle brackets.
- 2019-05-13, add more content to the chapter about contributing.
- 2019-05-13, add more precise instructions about blog posts to approval template for editors.
- 2019-05-13, add policies allowing using either <- or = within a package as long as the whole package is consistent.
- 2019-05-13, add request for people to tell us if they use our standards/checklists when reviewing software elsewhere.
- 2019-04-29, add requirement and advice on testing packages using `devel` and `oldrel` R versions on Travis.
- 2019-04-23, add a sentence about why being generous with attributions and more info about ctb vs aut.
- 2019-04-23, add link to Daniel Nüst's notes about migration from XML to xml2.
- 2019-04-22, add use of rOpenSci forum to maintenance section.
- 2019-04-22, ask reviewer for consent to be added to DESCRIPTION in review template.
- 2019-04-22, use a darker blue for links (feedback by `@kwstat`, #138).
- 2019-04-22, add book cover.
- 2019-04-08, improve formatting and link text in README (`@katrinleinweber`, #137)
- 2019-03-25, add favicon (`@wlandau`, #136).
- 2019-03-21, improve Travis CI guidance, including link to examples. (`@mpadge`, #135)
- 2019-02-07, simplify code examples in Package Evolution section (maintenance_evolution.Rmd file) (`@hadley`, #129).
- 2019-02-07, added a PDF file to export (request by `@IndrajeetPatil`, #131).

19.9 0.1.5

- 2019-02-01, created a .zenodo.json to explicitly set editors as authors.

19.10 First release 0.1.0

- 2019-01-23, add details about requirements for packages running on all major platforms and added new section to package categories.
- 2019-01-22, add details to the guide for authors about the development stage at which to submit a package.
- 2018-12-21, inclusion of an explicit policy for conflict of interest (for reviewers and editors).
- 2018-12-18, added more guidance for editor on how to look for reviewers.
- 2018-12-04, onboarding was renamed Software Peer Review.

19.11 place-holder 0.0.1

- Added a `NEWS.md` file to track changes to the book.

20 Modelo de revisão

Você pode salvar isso como um arquivo RMarkdown ou excluir o YAML e salvá-lo como um arquivo Markdown.

20.1 Revisão do pacote

Marque as caixas conforme aplicável e elabore nos comentários abaixo. Sua avaliação não se limita a estes tópicos, conforme descrito no guia do(a) revisor(a)

- **Descreva resumidamente qualquer relacionamento de trabalho que você tem (teve) com as pessoas autoras do pacote.**

☐ Como revisor(a), confirmo que não há conflitos de interesse para que eu revise este trabalho (se você não tiver certeza se está em conflito, fale com seu(sua) editor(a) *antes* de iniciar sua revisão)

20.1.0.1 Documentação

O pacote inclui todas as seguintes formas de documentação:

☐ **Uma declaração de necessidade:** declarando claramente os problemas que o software foi projetado para resolver além do seu público-alvo no README
☐ **Instruções de instalação:** para a versão de desenvolvimento do pacote e quaisquer dependências que sejam fora do padrão no README
☐ **Vignette(s):** demonstrando as principais funcionalidades com exemplos que são executados localmente com sucesso
☐ **Documentação de funções:** para todas as funções exportadas
☐ **Exemplos:** (que são executados localmente com sucesso) para todas as funções exportadas
☐ **Diretrizes da comunidade:** incluindo diretrizes de contribuição no README ou CONTRIBUTING, e DESCRIPTION com `URL`, `BugReports` e `Maintainer` (o qual pode ser gerado automaticamente via `Authors@R`).

20.1.0.2 Funcionalidade

- ☐ **Instalação:** Processo de instalação documentado conclui com sucesso.
- ☐ **Funcionalidade:** Qualquer funcionalidade que foi assumida pelo software foi confirmada.
- ☐ **Desempenho:** Qualquer desempenho a mais que foi assumido pelo software foi confirmado.
- ☐ **Testes automatizados:**Os testes unitários cobrem funções essenciais do pacote e uma gama razoável de *inputs* e condições. Todos os testes passam na máquina local.
- ☐ **Diretrizes de empacotamento**: O pacote está em conformidade com as diretrizes de empacotamento da rOpenSci.

Horas estimadas gastas na revisão:

- ☐ Caso as pessoas autoras do pacote considerem apropriado, concordo em ser reconhecido(a) como revisor(a) do pacote (função “rev”) no arquivo DESCRIPTION do pacote.

20.1.1 Comentários da revisão

21 Modelo para o(a) editor(a)

21.0.1 Checks do editor:

- ☐ **Documentação**: O pacote possui documentação suficiente e está disponível online (`README`, `pkgdown docs`), de modo que permita um estudo de suas funcionalidades e escopo sem a necessidade de instalar o pacote. Em particular,
  - ☐ O caso de uso do pacote é bem feito?
  - ☐ A página principal (*index*) da documentação é clara (agrupada por tópicos se necessário)?
  - ☐ As documentações longas (*vignettes*) são legíveis, e detalhadas o suficiente ao invés de serem muito superficiais?
- ☐ **Adequação**: O pacote atende aos critérios de adequação e sobreposição.
- ☐ **Instruções de instalação**: As instruções de instalação são claras o suficiente para um ser humano?
- ☐ **Testes**: Caso o pacote possua algum produto interativo / HTTP / gráfico etc. os seus testes estão utilizando ferramentas em estado-de-arte?
- ☐ **Instruções para contribuição**: A documentação para contribuição é clara o suficiente (e.g. tokens para testes e áreas de playground)?
- ☐ **Licença:** O pacote possui uma licença aceita no CRAN ou OSI.
- ☐ **Gerenciamento do projeto**: O monitoramento de problemas (*issues*) e PRs (*pull requests*) está em bom estado, e.g. existem bugs muito críticos, está claro quando um pedido de *feature* está planejado para ser tratado?

21.0.1.1 Comentários do(a) editor(a)

---

22 Modelo de solicitação de revisão

Os editores podem usar o modelo de e-mail abaixo para recrutar revisores.

Caro [REVISOR(A)]

Olá, aqui é [EDITOR(A)]. [BRINCADEIRA AMIGÁVEL]. Estou escrevendo para perguntar se você gostaria de revisar um pacote para a rOpenSci. Como você provavelmente sabe, a rOpenSci realiza revisão por pares de pacotes de R contribuídos para nossa coleção, de maneira semelhante aos periódicos.

O pacote, [PACOTE] de [AUTOR(ES)], faz [FUNÇÃO]. Você pode encontrá-lo no GitHub aqui: [LINK PARA REPOSITÓRIO]. Também conduzimos nosso processo de revisão aberta via GitHub, aqui: [ISSUE DE ONBOARDING]

Se você aceitar, observe que pedimos aos revisores que concluam as avaliações em três semanas. (Descobrimos que a revisão de um pacote leva um tempo semelhante ao de um trabalho acadêmico.)

Nosso guia para revisores detalha o que procuramos em uma revisão de pacote e inclui links para exemplos de revisão. Nossos padrões estão detalhados em nosso [guia de pacotes] e fornecemos um modelo de revisão para você usar. Certifique-se de que não haja um conflito de interesses que lhe impeça de revisar este pacote. Se você tiver dúvidas ou comentários, sinta-se à vontade para me perguntar ou postar no fórum da rOpenSci.

A comunidade da rOpenSci é o nosso melhor ativo. Nosso objetivo é que as revisões sejam abertas, não adversas e focadas na melhoria da qualidade do software. Seja respeitoso(a) e gentil! Consulte nosso guia para revisores e o nosso código de conduta para mais informações.

[SE MENTORIA FOR REQUISITADA: Você indicou em seu formulário que prefere ter uma orientação para sua primeira revisão. Você é livre para me usar como recurso durante esse processo, incluindo fazer perguntas por e-mail e Slack (você receberá um convite para o Slack da rOpenSci) e compartilhar os rascunhos de sua revisão para feedback antes de postá-los. Também estarei feliz de fazer uma breve videochamada para lhe explicar o processo. Por favor, me avise em sua resposta se você deseja agendar uma videochamada dessas.]

Você consegue revisar? Caso não consiga, você tem alguma sugestão de revisor(a)? Se eu não receber uma resposta sua dentro de uma semana, vou presumir que você não pode revisar neste momento.

Agradeço pelo seu tempo.

Atenciosamente

[EDITOR(A)]

23 Modelo de comentário de aprovação do(a) revisor(a)

23.1 Resposta do(a) revisor(a)

23.1.0.1 Aprovação final (pós-revisão)

- ☐ **O(a) autor(a) respondeu à minha revisão e realizou as mudanças requisitadas. Eu recomendo a aprovação deste pacote.**

Estimativa de horas dedicadas à revisão:

24 Modelo de notícias

```
foobar 0.2.0 (2016-04-01)
=========================

### NOVAS FUNCIONALIDADES

  * Nova função adicionada `do_things()` para fazer coisas (#5)

### MELHORIAS PEQUENAS

  * Documentação foi aprimorada para a função `things()` (#4)

### CORREÇÕES DE BUGS

  * Correção de um bug de parseamento em `parse()` (#3)

### DEPRECADO E EXTINTO

  * `hello_world()` está deprecada agora e será removida em
     uma futura versão, utilize `hello_mars()`

### CORREÇÕES EM DOCUMENTAÇÃO

  * Papel de `hello_mars()` versus `goodbye_mars()` está melhor esclarecido.


### (um especial: qualquer cabeçalho que agrupa um número grande de mudanças sobre uma única

    * blablabla.

foobar 0.1.0 (2016-01-01)
=========================

### NOVAS FUNCIONALIDADES
```

```
* lançamento no CRAN
```

25 Orientação para o lançamento de livros

Os(as) editores(as) que estão se preparando para um lançamento podem executar o script `prelease.R` na pasta `inst` deste repositório para abrir automaticamente um problema no GitHub com os pontos de verificação para todos os problemas atuais atribuídos ao marco da próxima versão, juntamente com a seguinte lista de verificação. Antes de executar o script, verifique manualmente a atribuição de problemas ao marco. Isso deve ser executado um mês antes do lançamento planejado.

25.1 Versão de lançamento do livro

25.1.1 Manutenção do repositório entre lançamentos

- ☐ Consulte a página de problemas para o guia dev e também para o repositório de revisões de software, procure por mudanças que ainda devem ser feitas no guia dev. Atribua os problemas encontrados no guia dev ao marco correspondente às versões, seja esta a próxima versão, ou, às versões seguintes, e.g versão 0.3.0. Encoraje novos PRs e revise eles.

25.1.2 1 mês antes do lançamento

- ☐ Lembre os editores de abrirem problemas/PRs para itens que desejam ver na próxima versão.
- ☐ Execute a função `devguide_prerelease()` do pacote `devguider`.
- ☐ Peça aos(às) editores(as) por qualquer feedback que você precise antes do lançamento.
- ☐ Para cada contribuição/alteração verifique se as NOTÍCIAS no arquivo `Appendix.Rmd` foram atualizadas.
- ☐ Planeje uma data para o lançamento e se comunique com o/a gerente da comunidade da rOpenSci, que lhe dará uma data para publicar uma postagem no blog (ou nota técnica).

25.1.3 2 semanas antes do lançamento

- ☐ Escreva um rascunho para uma postagem de blog (ou nota técnica) sobre o lançamento com antecedência suficiente para que os(as) editores(as) e, em seguida, o(a) gerente da comunidade, possam revisá-lo (2 semanas). Exemplo, instruções gerais para a postagem no blog, instruções específicas para as postagens de lançamento.
- ☐ Crie um PR a partir da branch `dev` para a branch `master` e, em seguida, comunique aos editores através do GitHub e do Slack. Mencione o rascunho da postagem do blog em um comentário dentro deste PR.

25.1.4 Lançamento

- ☐ Verifique as URLs usando a função `devguide_urls()` do pacote {devguider}
- ☐ Verifique a ortografia usando a função `devguide_spelling()` do pacote {devguider}. Atualize também a WORDLIST conforme necessário.
- ☐ Realize um squash sobre os seus commits para o PR de `dev` para `master`.
- ☐ Atualize a página de *release* do GitHub, e confira a página de *release* do Zenodo.
  - [ ] Reconstrua (para atualizar os metadados do livro no Zenodo) ou aguarde o processo diário de construção do livro.
- ☐ Crie novamente a branch `dev`.
- ☐ Conclua o PR com a sua postagem de blog (ou nota técnica). Destaque os aspectos mais importantes a serem destacados em tweets (e publicações) como parte da discussão do PR.

26 Como definir um redirecionamento

26.1 Site que não seja de páginas do Github Pages (por exemplo, Netlify)

Substitua o conteúdo do site atual por dois arquivos chamados `index.html` e `404.html`. Ambos os arquivos devem conter o seguinte conteúdo:

```
<html>
<head>
<meta http-equiv="refresh" content="0;URL=https://docs.ropensci.org/<pkgname>/">
</head>
</html>
```

26.2 Páginas do GitHub

Você pode configurar o redirecionamento no repositório gh-pages do seu usuário principal:

- crie um novo repositório (se você ainda não tiver um): `https://github.com/<username>/<username>.githu`
- Nesse repositório, crie um diretório `<pkgname>` contendo 2 arquivos: um `index.html` e `404.html` que redirecionam para o novo local (consulte a subseção anterior).
- Teste o endereço `https://<username>.github.io/<pkgname>/index.html` que vai redirecionar.

27 Comandos do bot

27.1 Para todos

Vale ressaltar que nós limpamos os tópicos de problemas ao remover todo conteúdo estranho, portanto, o registro de que você solicitou ajuda de bots será rapidamente apagado ou ocultado.

27.1.1 Veja a lista de comandos disponíveis para você

Se você precisar de um lembrete rápido!

```
@ropensci-review-bot help
```

27.1.2 Veja o código de conduta

```
@ropensci-review-bot code of conduct
```

27.2 Para autores

27.2.1 Verificar o pacote com o pkgcheck

Quando seu pacote tiver mudado substancialmente.

```
@ropensci-review-bot check package
```

27.2.2 Enviar resposta aos revisores

Para registrar sua resposta aos revisores.

```
@ropensci-review-bot submit response <response-url>
```

onde `<response_url>` é o link para o comentário de resposta no tópico do problema.

27.2.3 Finalizar a transferência do repositório

Depois que você aceitar o convite para a organização do GitHub do rOpenSci e transferir seu repositório do GitHub para ela, execute este comando para obter novamente o acesso de administrador ao seu repositório.

```
@ropensci-review-bot finalize transfer of <package-name>
```

27.2.4 Obter um novo convite após a aprovação

Se você perdeu o prazo de uma semana para aceitar o convite para a organização do rOpenSci no GitHub, execute isso para receber um novo convite.

```
@ropensci-review-bot invite me to ropensci/<package-name>
```

27.3 Para o editor-chefe

27.3.1 Atribua um (a) editor (a)

```
@ropensci-review-bot assign @username as editor
```

27.3.2 Colocar o envio em espera

Veja política editorial.

```
@ropensci-review-bot put on hold
```

27.3.3 Indique que o envio está fora do escopo

Não se esqueça de publicar primeiro um comentário explicando a decisão e agradecendo ao(s) autor(es) pelo envio.

```
@ropensci-review-bot out-of-scope
```

27.4 Para o editor designado

27.4.1 Colocar o envio em espera

Veja política editorial.

```
@ropensci-review-bot put on hold
```

27.4.2 Verificar o pacote com o pkgcheck

Geralmente, apenas em consultas pré-submissão ou quando os autores indicam que o pacote foi substancialmente alterado.

```
@ropensci-review-bot check package
```

27.4.3 Verificar padrões estatísticos

Geralmente, apenas em consultas pré-submissão ou quando os autores indicam que o pacote foi substancialmente alterado.

```
@ropensci-review-bot check srr
```

27.4.4 Verifique se o README tem o selo de revisão de software

No final do processo de envio.

```
@ropensci-review-bot check readme
```

27.4.5 Indique que você está procurando revisores

```
@ropensci-review-bot seeking reviewers
```

27.4.6 Atribuir um (a) revisor (a)

```
@ropensci-review-bot assign @username as reviewer
```

ou

```
@ropensci-review-bot add @username as reviewer
```

27.4.7 Remover um (a) revisor (a)

```
@ropensci-review-bot remove @username from reviewers
```

27.4.8 Ajustar a data de vencimento da revisão

```
@ropensci-review-bot set due date for @username to YYYY-MM-DD
```

27.4.9 Registre que uma revisão foi enviada

```
@ropensci-review-bot submit review <review-url> time <time in hours>
```

27.4.10 Aprovar o pacote

```
@ropensci-review-bot approve <package-name>
```