# rOpenSci Packages: Development, Maintenance, and Peer Review

rOpenSci software review editorial team (current and alumni): Brooke Anderson, Scott C

# Tabla de contenidos

# rOpenSci Dev Guide



```
@software{ropensci_2021_6619350,
  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        = nov,
  year         = 2021,
  publisher    = {Zenodo},
  version      = {0.7.0},
  doi          = {10.5281/zenodo.6619350},
  url          = {https://doi.org/10.5281/zenodo.6619350}
}
```

You can also read the PDF version of this book.

# Prefacio

¡Te damos la bienvenida! Este libro es una guía para autoras/es, encargadas/os, revisoras/es y editoras/es de rOpenSci.

La primera sección del libro contiene nuestras recomendaciones para crear y probar los paquetes de R.

La segunda sección está dedicada al proceso de revisión por pares de software de rOpenSci: en qué consiste, nuestras políticas y las guías específicas para orientar a quienes crean, mantienen, revisan y editan a lo largo del proceso. Para la *revisión de software estadístico*, consulta la página web del proyecto y los recursos asociados.

La tercera y última sección presenta nuestras buenas prácticas para hacer crecer tu paquete una vez que ha sido incorporado a rOpenSci: cómo colaborar con otras/os desarrolladoras/es, cómo documentar cada versión, cómo promover tu paquete y cómo aprovechar GitHub como plataforma de desarrollo. La tercera sección también incluye un capítulo para quienes desean comenzar a contribuir a los paquetes de rOpenSci.

Esperamos que esta guía te resulte útil y clara, y agradecemos tus sugerencias en las issues del libro.
¡Mucho éxito desarrollando paquetes de R!

El equipo editorial de rOpenSci.

Este libro es un documento vivo. Puedes ver las actualizaciones de nuestras buenas prácticas y políticas a través de las notas de cada versión.
Puedes citar este libro utilizando sus metadatos de Zenodo y su DOI.

“bibtex @software{ropensci_2021_6619350, 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 = nov, year = 2021, publisher = {Zenodo}, version = {0.7.0}, doi = {10.5281/zenodo.6619350}, url = {https://doi.org/10.5281/zenodo.6619350} }

```
*Si quieres contribuir sugerencias o correcciones a este libro, visita [el repositorio de Gi

*Agradecemos a las/os autoras/es, encargadas/os, revisoras/es y editoras/es invitadas/os por
```

```
[Hao Ye](https://github.com/ha0ye),
[Sébastien Rochette](https://github.com/statnmap),
[Edward Wallace](https://github.com/ewallace/),
[Alexander Fischer](https://github.com/s3alfisc/).
Por favor, avísanos si nos olvidamos de reconocer tu contribución.*



::: {.quarto-book-part}


`<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4ifQ== -->`{=html}

```{=html}
<!-- quarto-file-metadata: eyJyZXNvdXJjZURpciI6Ii4iLCJib29rSXRlbVR5cGUiOiJwYXJ0IiwiYm9va0lOZ
```
```

# Building Your Package

:::

# 1 Guía de empaquetado

rOpenSci acepta paquetes que cumplen nuestras recomendaciones a través de un proceso fluido de Revisión por pares de software. Para garantizar un estilo coherente en todas nuestras herramientas, hemos escrito este capítulo en el que se destacan recomendaciones para el desarrollo de paquetes. Por favor, también lee y aplica nuestro capítulo sobre integración continua (CI). En la tercera sección de este libro, que comienza con un capítulo sobre la colaboración, se ofrece una guía para continuar luego del proceso de revisión.

Recomendamos fuertemente a quienes desarrollen paquetes que lean el libro de Hadley Wickham y Jenny Bryan sobre el desarrollo de paquetes. Es conciso pero exhaustivo y está disponible gratis en línea. Algunas partes de nuetra guía son redundantes con respecto a otros recursos, pero destaca las recomendaciones de rOpenSci.

Para leer por qué vale la pena enviar un paquete a rOpenSci siguiendo las recomendaciones, echa un vistazo a razones para enviar un paquete.

## 1.1 Nombre del paquete y metadatos

### 1.1.1 Cómo elegir el nombre de tu paquete

- Recomendamos fuertemente nombres cortos y descriptivos en minúsculas. Si tu paquete tiene que ver con uno o más servicios comerciales, asegúrate de que el nombre no infringe las recomendaciones de la marca. Puedes comprobar si el nombre de tu paquete está disponible, es informativo y no es ofensivo utilizando el paquete `available`. En particular, *no* elijas un nombre de paquete que ya se utilice en CRAN o en Bioconductor.
- Un nombre de paquete menos común podría ser más fácil de encontrar (para que tú y en rOpenSci evaluemos el uso del paquete) y de buscar (para que quienes lo usen lo encuentren y busquen en Google sus preguntas). Evidentemente, un nombre *demasiado* único puede hacer que el paquete sea más difícil de encontrar (por ejemplo, podría ser un argumento para nombrar tu paquete geojson).
- Este artículo web de Nick Tierney lista otro aspectos interesantes de los nombres de los paquetes de R. En caso de que cambies de opinión sobre el nombre de tu paquete, descubre cómo cambiar el nombre de tu paquete en este otro artículo del blog de Nick.

### 1.1.2 Crear metadatos para tu paquete

Te recomendamos que utilices el paquete `codemetar` para crear y actualizar un JSON CodeMeta para tu paquete a través de `codemetar::write_codemeta()`. Agregará automáticamente toda la información útil, incluyendo temas de GitHub. CodeMeta utiliza conceptos de Schema.org por lo que, a medida que gane popularidad, los metadatos JSON de tu paquete podrían ser utilizados por servicios de terceros, incluso por motores de búsqueda.

## 1.2 Plataformas

- Los paquetes deben funcionar en todas las plataformas principales (Windows, macOS, Linux). Puede haber excepciones para paquetes que interactúen con funciones específicas del sistema, o wrappers para utilidades que sólo funcionan en plataformas limitadas, pero se debe hacer todo lo posible para garantizar la compatibilidad entre plataformas, incluyendo la compilación específica en cada sistema, o la contenerización de utilidades externas.

## 1.3 API del paquete

### 1.3.1 Nombres de funciones y argumentos

- Los nombres de las funciones y los argumentos deben elegirse para que funcionen juntos y formen una API de programación común y lógica que sea fácil de leer y autocompletar.
  - Considera un esquema de nomenclatura `objecto_verbo()` para las funciones de tu paquete que toman un tipo de datos común o interactúan con una API común. `objecto` se refiere a los datos/API y `verbo` a la acción principal. Este esquema ayuda a evitar conflictos de nombres con paquetes que pueden tener verbos similares, y hace que el código sea legible y fácil de autocompletar. Por ejemplo, en **stringi** las funciones que empiezan con `stri_` manipulan cadenas de texto (`stri_join()`, `stri_sort()`y en **googlesheets** las funciones que empiezan con `gs_` son llamadas a la API de Google Sheets (`gs_auth()`, `gs_user()`, `gs_download()`).
- Para las funciones que manipulan un objeto o dato y devuelven un objeto/dato del mismo tipo, haz que el objeto o dato sea el primer argumento de la función para mejorar la compatibilidad con el operador pipe (%>% o |>).
- Recomendamos fuertemente usar `snake_case` por sobre todos los otros estilos, a menos que estés usando funciones de un paquete que ya se utilice ampliamente.
- Evita los conflictos de nombres de funciones con los paquetes de R base u otros populares (por ejemplo `ggplot2`, `dplyr`, `magrittr`, `data.table`)

- Los nombres y el orden de los deben ser coherentes entre las funciones que utilizan inputs similares.
- Las funciones del paquete que importan datos no deben importar los datos al entorno global, sino que deben devolver objetos. Las asignaciones al entorno global deben evitarse en general.

### 1.3.2 Mensajes de la consola

- Utiliza `message()` y `warning()` en tus funciones para comunicarte con la persona en frente al teclado. No utilices `print()` o `cat()` a menos que sea para un método `print.*()` o `str.*()` ya que estos métodos de impresión de mensajes son más difíciles de silenciar.

### 1.3.3 Interfaces interactivas o gráficas

Si proporcionas una interfaz gráfica de usuario (GUI) (como una aplicación Shiny), para facilitar el flujo de trabajo incluye un mecanismo para reproducir automáticamente los pasos realizados en la GUI. Esto podría incluir la autogeneración de código para reproducir los mismos resultados, la generación de valores intermedios producidos en la herramienta interactiva, o simplemente un mapeo claro y bien documentado entre las acciones de la GUI y las funciones usadas. (Ver también "Tests" más abajo).

El paquete `tabulizer` por ejemplo, tiene un flujo de trabajo interactivo para extraer tablas, pero también puede extraer sólo coordenadas, por lo que se pueden volver a ejecutar los mismos pasos como un script. Otros dos ejemplos de aplicaciones shiny que generan código son https://gdancik.shinyapps.io/shinyGEO/y https://github.com/wallaceEcoMod/wallace/.

## 1.4 Estilo del código

- Para más información sobre cómo dar estilo a tu código, nombrar funciones y scripts de R dentro de la carpeta `R` te recomendamos que leas el capítulo Code del libro R Packages. Recomendamos el paquete `styler` para automatizar parte del estilo del código. También te sugerimos que leas la Guía de estilo de Tidyverse.
- Puedes optar por utilizar = en lugar de <- siempre que seas coherente con esa elección dentro de tu paquete. Recomendamos evitar el uso de -> para la asignación dentro del paquete. Si utilizas <- en todo tu paquete, y también utilizas `R6` en ese paquete, te verás obligado a utilizar = para la asignación dentro de la generación de la clase `R6Class` - esto no se considera una incoherencia porque no puedes usar <- en este caso.

## 1.5 Archivo CITATION

- Si tu paquete aún no tiene un archivo CITATION, puedes crear uno con `usethis::use_citation()` y llenarlo con los valores generados por la función `citation()`.
- Si archivas cada versión de tu repositorio de GitHub en Zenodo, añade el DOI principal de Zenodo al archivo CITATION.
- Si un día **luego de** la revisión en rOpenSci publicas un artículo sobre tu paquete, añádelo al archivo CITATION.
- Menos relacionado con tu paquete en sí mismo, pero con lo que lo soporta: si tu paquete incluye un recurso concreto, como una fuente de datos o, por ejemplo, un algoritmo estadístico, recuérdale a quien lo use cómo citar ese recurso mediante, por ejemplo, `citHeader()`. Quizás incluso añade la referencia a ese recurso.

Como ejemplo, revisa el archivo CITATION de nasapower que hace referencia tanto al manual como a un artículo. Lo único que le falta es un DOI de Zenodo para el manual, aunque la mayoría de los usuarios probablemente acabarán citando el artículo de JOSS.

```
citHeader("While nasapower does not redistribute the data in any way,\n",
          "we encourage users to follow the requests of the POWER\n",
          "Project Team:\n",
          "\n",
          "'When POWER data products are used in a publication, we\n",
          "request the following acknowledgment be included:\n",
          "These data were obtained from the NASA Langley Research\n",
          "Center POWER Project funded through the NASA Earth Science\n",
          "Directorate Applied Science Program.'\n",
          "\n",
          "To cite nasapower in publications, please use:")
citEntry(
  entry = "Article",
  author = as.person("Adam H Sparks"),
  title = "nasapower: A NASA POWER Global Meteorology, Surface Solar Energy and Climatolog
  doi = "10.21105/joss.01035",
  year = 2018,
  month = "oct",
  publisher = "The Open Journal",
  volume = 3,
  number = 30,
  pages = 1035,
  journal = "The Journal of Open Source Software",
  textVersion = paste("Sparks, Adam (2018). nasapower: A NASA POWER Global Meteorology, Su
year <- sub("-.*", "", meta$Date)
```

```
note <- sprintf("R package version %s", meta$Version)
bibentry(bibtype = "Manual",
         title = "{nasapower}: NASA-POWER Data from R",
         author = c(person("Adam", "Sparks")),
         year = year,
         note = note,
     url = "https://CRAN.R-project.org/package=nasapower")
 textVersion = paste0("Adam H Sparks, (", year, ").",
  " nasapower: A NASA POWER Global Meteorology, Surface Solar Energy and Climatology Data
  note, ".",
  " https://CRAN.R-project.org/package=nasapower")
```

- También puedes crear y almacenar un archivo `CITATION.cff` gracias al paquete cffr que también proporciona un flujo de trabajo con GitHub Actions para mantener el archivo `CITATION.cff` actualizado.

## 1.6 README

- Todos los paquetes deben tener un archivo *README*, llamado `README.md` en la raíz del repositorio. El *README* debe incluir, en orden:
  - El nombre del paquete.
  - Etiquetas de integración continua y la cobertura de tests, la etiqueta de revisión por pares de rOpenSci una vez que haya comenzado (ver más abajo), una etiqueta de repostatus.org, y cualquier otra (por ejemplo R-universe). Si el *README* tiene muchas etiquetas, puedes considerar el uso de una tabla de etiquetas, ver este ejemplo, este otro o este otro. Esta tabla debe ser más ancha que alta.
  - Breve descripción de los objetivos del paquete con enlaces descriptivos a todas las viñetas (renderizadas, es decir, legibles, por ejemplo el sitio web de documentación) a menos que el paquete sea pequeño y sólo haya una viñeta que repita el *README*.
  - Instrucciones de instalación utilizando, por ejemplo, el paquete remotes, paquete pak o R-universe.
  - Cualquier configuración adicional que se requiera (tokens de autenticación, etc).
  - Breve demostración de uso.
  - Si aplica, cómo se compara el paquete con otros paquetes similares y/o cómo se relaciona con otros paquetes.
  - Información sobre cómo citar, es decir, muestra a las personas el formato de cita preferida en el *README* añadiendo el texto “así es como se puede citar mi paquete”. Véase, por ejemplo el *README* de ecmwfr.

Si utilizas otra etiqueta de estado del repo, como el ciclo de vida por favor, añade también una etiqueta de repostatus.org . Ejemplo de un *README* de un repo con dos insignias de estado.

- Una vez que hayas enviado el paquete y haya completado la revisión editorial, añade la insignia de revisión por pares como

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

donde issue_id es el número del _issue _en el repositorio donde se hizo la revisión del software. Por ejemplo, la etiqueta de revisión de `rtimicropem` utiliza el número 126, ya que es el número de *issue* de revisión. La etiqueta indicará primero "en revisión" y luego "revisado" una vez que tu paquete haya sido incorporado (*issue* etiquetado como "aprobado" y cerrado), y estará vinculado con el *issue* de la revisión.

- Si tu *README* tiene muchas etiquetas, considera la posibilidad de ordenarlas en una tabla HTML para que sea más fácil ver la información de un vistazo. Consulta los ejemplos en el repo de `drake` y en el repo de `qualtRics`. Las secciones posibles son:
  - Desarrollo (estados de CI, ver el capítulo CI, canal de Slack para la discusión, repostatus)
  - Edición o publicación (etiquetas de la versión de CRAN y de la fecha de publicación de METACRAN, etiqueta de la API de chequeos de CRAN, etiqueta de Zenodo)
  - Estadísticas o uso (descargas, por ejemplo etiquetas de descarga de METACRAN)
  - La tabla debe ser más ancha que larga para enmascarar el resto del *README*.
- Si tu paquete se conecta a una fuente de datos o a un servicio en línea, o utiliza otro software, ten en cuenta que el *README* de tu paquete puede ser el punto de entrada para alguien que lo usa por primera vez. Debe proporcionar suficiente información para poder entender la naturaleza de los datos, el servicio o el software, y proporcionar enlaces a otros datos y documentación relevantes. Por ejemplo un *README* no debe limitarse a decir: "Proporciona acceso a GooberDB", sino que también debe incluir "…, un repositorio online de avistamientos de Goober en Sudamérica. Se puede encontrar más información sobre GooberDB, y la documentación de la estructura de la base de datos y metadatos en *este enlace*".
- Recomendamos no crear el `README.md` directamente, sino a partir de un archivo `README.Rmd` (un archivo R Markdown) si incluye código de ejemplo. La ventaja del archivo `.Rmd` es que puedes combinar el texto con el código que puede actualizarse fácilmente cada vez que se actualice tu paquete.
- Considera utilizar `usethis::use_readme_rmd()` para generar una plantilla para el archivo `README.Rmd` y para configurar automáticamente un chequeo que garantice que `README.md` sea siempre más reciente que `README.Rmd` antes de hacer un commit.
- Los ejemplos largos deben incluirse sólo en las viñetas. Si quieres que las viñetas sean más accesibles antes de instalar el paquete, te sugerimos crear un sitio web para tu paquete.

- Añade un código de conducta y una guía de contribución.
- Consulta el *README* de `gistr` para ver un buen ejemplo de *README* de un paquete pequeño, y el *README* de `bowerbird` para un buen ejemplo de *README* de un paquete más grande.

## 1.7 Documentación

### 1.7.1 General

- Todas las funciones exportadas del paquete deben estar completamente documentadas con ejemplos.
- Si existe un posible solapamiento o confusión con otros paquetes que ofrezcan una funcionalidad similar o tengan un nombre parecido, añade una nota en el *README*, en la viñeta principal y, potencialmente, en el campo descripción de archivo DESCRIPTION. Ejemplos en rtweet README, rebird README, y el paquete no-rOpensci slurmR.
- El paquete debe contener la documentación general del paquete para `?paquete` (o `` ?`paquete-package` `` si hay un conflicto de nombres). Opcionalmente, puedes utilizar tanto `?paquete` como `` ?`paquete-package` `` para el archivo del manual del paquete utilizando la etiqueta `@aliases` de roxygen. `usethis::use_package_doc()` añade la plantilla para generar la documentación general.
- El paquete debe contener al menos una viñeta en formato **HTML** que cubra una parte importante de las funciones del paquete, ilustrando casos de uso realistas y cómo se supone que las funciones interactúen entre ellas. Si el paquete es pequeño, la viñeta y el *README* pueden tener un contenido muy similar.
- Al igual que el *README*, la documentación general o las viñetas puede ser el punto de entrada para alguien que lo usa por primera vez. Si tu paquete se conecta a una fuente de datos o a un servicio en línea, o utiliza otro software, debe proporcionar suficiente información para poder entender la naturaleza de los datos, el servicio o el software, y proporcionar enlaces a otros datos y documentación relevantes. Por ejemplo, la introducción de una viñeta o la documentación no debería limitarse a decir: "Proporciona acceso a GooberDB", sino incluir también: "…, un repositorio online de avistamientos de Goober en Sudamérica. Puedes encontrar más información sobre GooberDB y la documentación de la estructura de la base de datos y los metadatos en *este enlace*". Cualquier viñeta debe incluir que conocimientos previos son necesarios para poder entenderla.

La viñeta general debe presentar una serie de ejemplos que progresen en complejidad desde el uso básico al avanzado.

- La funciones de uso avanzado o que sean usadas sólo para desarrollo pueden incluirse en una viñeta separada (por ejemplo, la programación usando evaluación no estándar con dplyr).

- El *README*, la documentación del paquete de nivel superior, las viñetas, los sitios web, etc., deben tener suficiente información al principio para obtener una visión general del paquete y de los servicios o datos a los que se conecta, y proporcionar navegación a otras partes de la documentación relevante. Esto es para seguir el principio de *múltiples puntos de entrada* es decir, tener en cuenta el hecho de que cualquier pieza de documentación puede ser el primer encuentro de la persona con el paquete y/o la herramienta o los datos que accede.
- La(s) viñeta(s) debe(n) incluir citas a software y de artículos cuando corresponda.
- Si tu paquete proporciona acceso a una fuente de datos, requerimos que la DESCRIPCIÓN contenga (1) Una breve identificación y/o descripción de la organización responsable de la generación de los datos; y (2) La URL que vincula con la página pública que proporciona, describe o permite el acceso a los datos (que a menudo puede diferir de la URL que conduce directamente a la fuente de datos).
- Utiliza mensajes de inicio en el paquete sólo cuando sea necesario (cuando algunas funciones son enmascaradas, por ejemplo). Evita los mensajes de inicio del paquete tales como "Esto es paquete 2.4-0" o la guía de cómo citarlo porque pueden ser molestos para el/la usuario/a. Utiliza la documentación para dar ese tipo de información.
- Puedes optar por tener una sección en el *README* sobre casos de uso de tu paquete (otros paquetes, artículos de blog, etc.), ejemplo.

### 1.7.2 Uso de roxygen2

- Pedimos que todos los paquetes que se presenten a revisión utilicen `roxygen2` para generar la documentación. `roxygen2` es un paquete de R que compila automáticamente archivos `.Rd` en la carpeta `man` del paquete utilizando etiquetas incluidas antes de cada función.
- Más información sobre el uso de documentación con roxygen2 está disponible en el libro *R Packages*.
- Si escribiste los archivos .Rd a mano, el paquete Rd2roxygen contiene funciones para convertir la documentación de Rd a roxygen.
- Una ventaja clave de utilizar `roxygen2` es que tu archivo `NAMESPACE` siempre se generará automáticamente y estará actualizado.
- Todas las funciones deben documentar el tipo de objeto devuelto bajo el encabezamiento `@return`.
- El valor por defecto de cada parámetro debe estar claramente documentado. Por ejemplo, en lugar de escribir `Un valor lógico que determina si ...` deberías escribir `Un valor lógico (por defecto` `` `TRUE` ``) `que determina si ...`. También es una buena práctica indicar los valores por defecto directamente en la definición de tu función:

```r
f <- function(a = TRUE) {
  # código de la función
}
```

- La documentación debe ayudar a la navegación incluyendo links entre funciones relacionadas y agrupando la documentación de funciones relacionadas en página de ayuda comunes. Para esto recomendamos las etiquetas `@seealso` y `@family` (esta última crea automáticamente enlaces "See also" y pueden agrupan las funciones en sitios web generados con pkgdown). Consulta la sección "manual" del libro *R Packages* y la sección "Agrupación de funciones" de este capítulo para más detalles.
- Puedes reutilizar partes de la documentación (por ejemplo, detalles sobre la autenticación, paquetes relacionados) en las viñetas, *README* y páginas de manual. Una forma de hacerlo es usando fragmentos de R Markdown. Usando "documentos hijos" con kintr puedes guardar fragmentos de código en la carpeta `man` y reutilizarlos en el *README*, las viñetas y la documentación (usando la etiqueta `@includeRmd`, disponible desde la versión 0.7.0 de roxygen2).
- Añade `#' @noRd` a las funciones internas.
- Si prefieres no abarrotar el código con una documentación muy larga, puedes colocar más ejemplos fuera del *script* de R. Coloca la documentación extra en archivos en la carpeta `man-roxygen` en la raiz de tu paquete e insértala en el archivo de ayuda con `@template <nombre de archivo>` debajo de la etiqueta `@example`.
  - Coloca la documentación de cualquier objeto en un archivo `.R` en la carpeta `man-roxygen` (en la raíz de tu paquete). Por ejemplo este archivo. Vincula ese archivo de plantilla desde el código de tu función (por ejemplo) con la etiqueta `@template` (por ejemplo). El contenido de la plantilla se insertará cuando se construya la documentación en el archivo resultante `.Rd` que se mostrará al acceder a la documentación de la función.
  - Ten en cuenta que actualmente el formato de markdown no funciona en las plantillas, así que asegúrate de utilizar formato LaTeX.
  - En la mayoría de los casos puedes ignorar las plantillas y `man-roxygen` pero hay dos casos en los que aprovecharlos será de gran ayuda:
    1. Si tienes mucha documentación en una función, clase u objeto, separar ciertos trozos de esa documentación puede mantener tu archivo `.R` ordenado. Esto es especialmente útil cuando tienes mucho código en ese archivo `.R`. Por otro lado, significa que la documentación no será legible "en el código fuente", ya que estará en otro archivo.
    2. Cuando utilizas la misma documentación en muchas funciones y archivos `.R` es útil utilizar una plantilla. Esto reduce el texto duplicado y ayuda a evitar que por error se actualice la documentación de una función pero no la de otra. Utilizar un archivo de plantilla para la documentación de un parámetro es una alternativa a definir el

parámetro para una función y utilizar `@inheritParams` para otras funciones que utilicen el mismo parámetro.

- A partir de la versión 7.0.0 de `roxygen2`, las clases `R6` son oficialmente compatibles. Consulta la documentación de roxygen2 para saber cómo documentar las clases `R6`.

### 1.7.3 URLs en la documentación

Esta subsección es especialmente relevante para quienes deseen enviar su paquete a CRAN. CRAN comprobará las URLs incluidas en la documentación y no permitirá páginas que redirijan a códigos 301. Puedes utilizar el paquete urlchecker para reproducir los checks de CRAN y, en particular, sustituir las URLs originales por las URLs a las que redirigen. Otra opción es mostrar algunas URLs de manera explícita sin link (cambiar `<https://ropensci.org/>` por `https://ropensci.org/` o `\url{https://ropensci.org/}` por `https://ropensci.org/`), pero si lo haces, tendrás que implementar algún tipo de comprobación de URL para evitar que se rompan sin que te des cuenta. Además, no se podrá hacer click en los enlaces desde la documentación local.

## 1.8 Sitio web de la documentación

Te recomendamos que crees un sitio web con la documentación de tu paquete utilizando `pkgdown`. Neal Richardson escribió un tutorial para dar los primeros pasos con `pkgdown` y, cómo no es de extrañar, el paquete `pkgdown` tiene su propio sitio web de documentación.

Hay algunos elementos que nos gustaría subrayar aquí.

### 1.8.1 Construcción automática del sitio web de documentación

Sólo tendrás que preocuparte por la construcción automática de tu sitio web cuando se apruebe y se transfiera el repositorio de tu paquete a la organización ropensci; de hecho, después de eso se construirá un sitio web con pkgdown para tu paquete luego de cada *push* al repositorio de GitHub. Puedes encontrar el estado de estas acciones en `https://dev.ropensci.org/job/nombre_paquete`, por ejemplo para `magick` y el sitio web en `https://docs.ropensci.org/nombre_paquete`, por ejemplo para `magick`. La construcción del sitio web utilizará el archivo de configuración pkgdown si tienes uno, excepto para el estilo, ya que utilizará el paquete `rotemplate`. El sitio web resultante tendrá una barra de búsqueda local. Por favor, informa de los errores, preguntas y pedido de nuevas características para la construcción del sitio centralizada en https://github.com/ropensci/docs/ y sobre la plantilla en https://github.com/ropensci/rotemplate/.

*Si las viñetas de tus paquetes necesitan credenciales (claves de la API, tokens, etc.) para poder generarlas, es posible que quieras pregenerar las viñetas ya que las credenciales no se pueden utilizar en el servidor de documentos.*

Antes de la presentación y transferencia de tu paquete, puedes utilizar el enfoque documentado por `pkgdown` o el paquete `tic` para la construcción automática del sitio web del paquete. Esto te ahorrará el trabajo de ejecutar (y recordar ejecutar) `pkgdown::build_site()` cada vez que haya que actualizar el sitio. Primero consulta nuestro capítulo sobre integración continua si ésto no te resulta familiar. En cualquier caso, no olvides actualizar todas las apariciones de la URL del sitio web después de hacer la transferencia a la organización ropensci.

### 1.8.2 Agrupar funciones en el índice

Cuando tu paquete tenga muchas funciones, conviene agruparlas en el índice de la documentación, lo cual se puede hacer de forma más o menos automática.

Si utilizas `roxygen2` versión 6.1.1 o mayor, puedes utilizar la etiqueta `@family` en la documentación de tus funciones para indicar su grupo. Esto generará enlaces entre la documentación local de las funciones del paquete instalado (sección "See also") *y* te permitirá utilizar la función `has_concept` de `pkgdown` en el archivo de configuración de tu sitio web. Puedes ver un ejemplo (no rOpenSci) cortesía de `optiRum`: etiqueta family, archivo de configuración de `pkgdown` y la sección en el índice resultante. Para personalizar el texto del título de la referencia cruzada creada por roxygen2 (`Other {family}:`), puedes consultar la documentación de roxygen2 sobre cómo proporcionar una lista `rd_family_title` en `man/roxygen/meta.R`.

Menos automáticamente, puedes ver el ejemplo de sitio web de `drake` y el archivo de configuración asociado.

### 1.8.3 Marca de autores/as

Puedes hacer que los nombres de (algunos) autores/as sean links añadiendo su URL, e incluso puedes sustituir sus nombres por un logo (por ejemplo rOpenSci… ¡o tu organización/empresa!). Ver la documentación de `pkgdown`.

### 1.8.4 Configurar la barra de navegación

Puedes hacer que el contenido de tu sitio web sea más fácil de navegar organizando la barra de navegación, consulta `pkgdown` documentación. En particular, ten en cuenta que si nombras la viñeta principal de tu paquete como "pkg-name.Rmd", será accesible desde la barra de navegación como `Para empezar` en lugar de a través de `Artículos > Título de la Vigñeta`.

### 1.8.5 Mathjax

Una vez que tu paquete sea transferido y obtenga un sitio web utilizando nuestra plantilla de `pkgdown`, si quieres utilizar Mathjax tendrás que especificarlo en el archivo de configuración de `pkgdown` de la siguiente manera

```
template:
  params:
    mathjax: true
```

### 1.8.6 Logo del paquete

Para utilizar el logo de tu paquete en la página de inicio de pkgdown, consulta `usethis::use_logo()`. Si tu paquete no tiene ningún logotipo, el generador de documentación de rOpenSci utilizará el logo de rOpenSci en su lugar.

## 1.9 Autoría

El archivo `DESCRIPTION` de un paquete debe enumerar las personas que participaron de la autoría y quienes colaboraron en el paquete, utilizando la sintaxis `Authors@R` para indicar sus funciones (*author*/*creator*/*contributor*, etc.) si hay más de una persona listada. Utiliza el campo de comentarios para indicar el ID ORCID de cada persona si tuviera (ver este post). Revisa esta sección de "Writing R Extensions" para más detalles. Si crees que quienes revisaron tu paquete han hecho una contribución sustancial al desarrollo de tu paquete, puedes agregarlas en el campo `Authors@R` usando el tipo de contribución `"rev"`, por ejemplo:

```
person("Bea", "Hernández", role = "rev",
comment = "Bea revisó el paquete (v. X.X.XX) para rOpenSci, ver <https://github.com/roper
```

No incluyas a nadie en tu paquete sin antes pedir su consentimiento. Lee más en esta entrada del blog *Thanking Your Reviewers: Gratitude through Semantic Metadata* (Agradeciendo las revisiones: La gratitud a través de los metadatos semánticos). Por favor, no agregues al equipo editorial. ¡Su participación y contribución a rOpenSci es suficiente agradecimiento!

### 1.9.1 Autoría del código incluido en el paquete

Muchos paquetes incluyen código de otro software. Tanto si se incluyen archivos enteros como funciones individuales de otros paquetes, los paquetes de rOpenSci deben seguir la *Política de Repositorio* de CRAN:

> La propiedad de los derechos de autor y de propiedad intelectual de todos los componentes del paquete debe ser clara e inequívoca (incluso en la especificación de autoría en el archivo DESCRIPTION). Cuando el código se copie (o se derive) del trabajo de otras personas (incluso del propio R), hay que tener cuidado de que se conserven las declaraciones de derechos de autor/licencia y no se tergiverse la autoría.
>
> Preferiblemente, se utilizaría un campo "Autores@R" con funciones "ctb" para quienes tiene la autoría de dicho código. Como alternativa, el campo "Autor" puede incluir a estas personas en el rol de colaboración.
>
> Cuando los derechos de autor los tenga una entidad distinta de quienes mantienen la autoría del paquete, se indicará preferentemente mediante el rol "cph" en el campo "Autores@R", o utilizando un campo "Derechos de autor" (si es necesario, redirigiendo a un archivo inst/COPYRIGHTS).
>
> Deben respetarse las marcas comerciales.

## 1.10 Licencia

El paquete debe tener una licencia aceptada por CRAN u OSI. Para más detalles sobre las licencias, consulta el libro R packages.

## 1.11 Testeo

- Todos los paquetes deben pasar `R CMD check`/`devtools::check()` en las principales plataformas.
- Todos los paquetes deben tener un conjunto de *tests* que cubran la funcionalidad principal del paquete. Los *tests* deben cubrir también el comportamiento del paquete en caso de error.
- Es una buena práctica escribir *tests* unitarios para todas las funciones, y para todo el código del paquete en general, asegurando que se cubra las funcionalidades clave. Si la cobertura de los *tests* en tu paquete está por debajo del 75% probablemente requerirá *tests* adicionales o una explicación de la baja cobertura antes de ser enviado para su revisión.

- Recomendamos utilizar testthat para escribir los *tests*. Intenta escribir *tests* a medida que escribes cada nueva función. Esto responde a la necesidad obvia de tener *tests* adecuados para el paquete, pero te permite pensar en varias formas en las que una función puede fallar, y *defensivamente* programar en contra esas fallas. Más información.
- Los *tests* deben ser fáciles de entender. Te sugerimos que leas el artículo de blog *Why Good Developers Write Bad Unit Tests* (Por qué las personas que son buenas desarrollando escriben malos tests) de Michael Lynch.
- Los paquetes con aplicaciones Shiny deberán generar *tests* unitarios usando `shinytest2` o `shinytest` para comprobar que las interfaces interactivas se comportan como se espera.
- Para testear las funciones que crean gráficos, sugerimos utilizar vdiffr una extensión del paquete testthat; o pruebas instantáneas de testthat.
- Si tu paquete interactúa con recursos de la web (APIs de la web y otras fuentes de datos en la web) podrías encontrar el libro *HTTP testing in R* (Testeando HTTP en R) de Scott Chamberlain y Maëlle Salmon relevante. Paquetes que ayudan a realizar *tests* HTTP (clientes HTTP correspondientes):
  - httptest2 (httr2);
  - httptest (httr);
  - vcr (httr, crul);
  - webfakes (httr, httr2, crul, curl).
- El paquete testthat tiene una función `skip_on_cran()` que puedes utilizar para no ejecutar algunos *tests* en CRAN. Recomendamos utilizarla en todas las funciones que tengan llamadas a APIs, ya que es muy probable que fallen en CRAN. Estos *tests* deberán ejecutarse en la integración continua. Ten en cuenta que a partir de testthat 3.1.2, `skip_if_offline()` llama automáticamente a `skip_on_cran()`. Más información en *CRAN preparedness for API wrappers* (Preparación de CRAN para utilización de APIs).
- Si tu paquete interactúa con una base de datos, dittodb puede resultarte útil.
- Una vez que hayas configurado la integración continua (CI), utiliza el informe de cobertura de tu paquete (revisa esta sección de nuestro libro) para identificar las líneas no testeadas y añadir más *tests*.
- Aunque utilices la integración continua te recomendamos que ejecutes los *tests* localmente antes de enviar tu paquete ya que a menudo se omiten algunos *tests* (puede que tengas que establecer `Sys.setenv(NOT_CRAN="true")` para asegurarte de que se ejecutan todos los *tests*). Además, te recomendamos que antes de enviar tu paquete utilices el paquete de MangoTheCat `goodpractice` para testear tu paquete en busca de posibles fuentes de errores, y ejecutar `spelling::spell_check_package()` para encontrar errores ortográficos en la documentación.

## 1.12 Ejemplos

- Incluye ejemplos extensos en la documentación. Además de demostrar cómo se utiliza el paquete, pueden ser una forma fácil de testear la funcionalidad del paquete antes de que haya *tests* adecuados. Sin embargo, ten en cuenta que exigimos *tests* en los paquetes contribuidos.
- Puedes ejecutar los ejemplos con `devtools::run_examples()`. Ten en cuenta que los ejemplos que no estén rodeados de `\dontrun{}` o `\donttest{}` serán ejecutados cuando ejecutes R CMD CHECK o su equivalente (por ejemplo, `devtools::check()`). Consulta la tabla de resumen en la documentación de roxygen2.
- Para evitar que los ejemplos se ejecuten en CRAN (por ejemplo, si requieren autenticación), tienes que utilizar `\dontrun{}`. Sin embargo, para un primer envío, CRAN no te permitirá que todos los ejemplos se salteen así. En este caso podrías añadir algunos pequeños ejemplos de juguete, o incluir el código de los ejemplos en `try()`. Consulta también la etiqueta `@exampleIf`, que al momento de escribir este artículo, se encuentra en la versión de desarrollo de roxygen2.
- Además de ejecutar los ejemplos localmente en tu propia computadora, te aconsejamos fuertemente que ejecutes los ejemplos en uno de los sistemas de integración continua. De nuevo, los ejemplos que no estén incluidos en `\dontrun{}` o `\donttest{}` no se ejecutarán. Puedes configurar la integración continua para que se ejecuten a través de los argumentos R CMD `--run-dontrun` y/o `--run-donttest`.

## 1.13 Dependencias del paquete

- Utiliza `Imports` en lugar de `Depends` para los paquetes que proporcionan funciones de otros paquetes. Asegúrate de enumerar los paquetes utilizados para los *tests* (`testthat`), y la documentación (`knitr`, `roxygen2`) en tu sección `Suggests` de dependencias de paquetes (si utilizas `usethis` para añadir la infraestructura de *tests* con `usethis::use_testthat()` o una viñeta mediante usethis::use_vignette() los paquetes necesarios se añadirán a `DESCRIPTION`). Si utilizas algún paquete en los ejemplos o *tests* de tu paquete, asegúrate de listarlo en `Suggests` si no aparece ya en `Imports`.
- Si tu paquete no está en Bioconductor pero depende de paquetes de Bioconductor, asegúrate de que las instrucciones de instalación en el *README* y la viñeta son lo suficientemente claras incluso para una persona que no esté familiarizada con el ciclo de publicación de Bioconductor.
  - ¿Hay que usar `BiocManager` (recomendado)? Documenta esto.

  - ¿La instalación automática de paquetes de Bioconductor con `install.packages()` es suficiente? En ese caso, menciona que se debe ejecutar `setRepositories()` si aún no han configurado los repositorios de Bioconductor necesarios.
  - Si tu paquete depende de Bioconductor a partir de una determinada versión, menciónalo en el archivo DESCRIPTION y en las instrucciones de instalación.

- Especificar las dependencias mínimas (por ejemplo `glue (>= 1.3.0)` en lugar de sólo `glue`) debería ser una elección consciente. Si sabes con certeza que tu paquete se romperá con una dependencia debajo de una determinada versión, especifícalo explícitamente.

Pero si no lo sabes, no es necesario especificar una dependencia mínima. En ese caso, si una persona informa de un fallo que está explícitamente relacionado con una versión antigua de una dependencia, entonces revísalo. Considerar las versiones de paquetes locales como la versiones mínimas necesarias para las dependencias es una mala práctica. Eso obligaría a todo el mundo a actualizar las dependencias innecesariamente (causando problemas con otros paquetes) cuando no hay una buena razón detrás de esa elección de versiones.

- En la mayoría de los casos en los que debes exponer funciones que vienen de alguna dependencia, debes importar y reexportar esas funciones individuales en lugar de enumerarlas en el campo `Depends`. Por ejemplo, si las funciones de tu paquete producen objetos `raster` podrías reexportar sólo las funciones de impresión y graficado del paquete raster.
- Si tu paquete utiliza una dependencia de *sistema*, debes
  - Indicarla en el archivo DESCRIPTION;
  - Comprobar que aparece en la lista de `sysreqsdb` para permitir que las herramientas automáticas lo instalen, o enviar una contribución si no es así;
  - Comprueba que está en un script `configure` (ejemplo) y devuelve un mensaje de error útil si no se encuentra (ejemplo). Los scripts `configure` pueden ser un reto, ya que a menudo requieren soluciones rebuscadas. para hacer que diversas dependencias del sistema funcionen en todos los sistemas. Utiliza ejemplos como punto de partida (más aquí), pero ten en cuenta que es habitual encontrar errores y casos límite y que a menudo violan las políticas de CRAN. No dudes en pedir ayuda en nuestro foro.
- Ten en cuenta las ventajas y desventajas de tener dependencias con un paquete. Por un lado, el uso de dependencias reduce el esfuerzo de desarrollo, y permite construir sobre funcionalidades útiles desarrollada por otras personas, especialmente si la dependencia realiza tareas complejas, es de alto rendimiento y/o está bien revisada y probada. Por otro lado, tener muchas dependencias supone una carga de mantenimiento ya que requiere estar al día con los cambios en esos paquetes, con riesgo para la sostenibilidad de tu paquete a largo plazo. También aumenta el tiempo y el tamaño de la instalación, lo que supone principalmente una consideración en el ciclo de desarrollo tuyo y del resto, y en los

sistemas de compilación automatizados. Los paquetes “pesados” -los que tienen muchas dependencias, y los que tienen grandes cantidades de código compilado- aumentan este costo. He aquí algunos enfoques para reducir las dependencias:

- Las funciones pequeñas y sencillas de una dependencia pueden copiarse en tu propio paquete si sólo utilizas unas pocas funciones de esa dependencia grande o pesada. (Ver la sección *Autoría* para saber cómo reconocer la autoría del código copiado). Por otro lado, las funciones complejas con muchos casos especiales (por ejemplo, los analizadores sintácticos) requieren considerable testeo y revisión.
    * Un ejemplo común de esto es devolver “tibbles” usadas por el tidyverse en las funciones del paquete que proporcionan datos. Se puede evitar el uso del paquete **tibble** devolviendo un tibble creado modificando un data.frame de la siguiente manera

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

      (Ten en cuenta que este enfoque no está universalmente aceptado.)
- Asegúrate de que utilizas la función del paquete donde está definida originalmente y no de un paquete que la re-exporta. Por ejemplo, muchas funciones de **devtools** pueden encontrarse en paquetes especializados más pequeños, como **sessioninfo**. La función %>% debe importarse de **magrittr** donde está definida, en lugar de **dplyr** que la reexporta y es mucho más pesado.
- Algunas dependencias proporcionan nombres de funciones y sintaxis más fáciles de interpretar que los nombres de las funciones y la sintaxis de R base. Si ésta es la razón principal para usar una función en una dependencia pesada, considera incluir el código de R base en una función interna bien nombrada en tu paquete. Consulta, por ejemplo, el script de R de rlang que proporciona funciones con una sintaxis similar a las funciones de purrr.
- Si las dependencias tienen funcionalidades que se solapan, comprueba si puedes depender sólo de una.
- Puedes encontrar más consejos sobre la gestión de dependencias en este artículo de Scott Chamberlain.

## 1.14 Estructura de base recomendada

- Para las consultas HTTP recomendamos utilizar curl, crul, httr o httr2 en vez de RCurl. El paquete curl es mejor es mejor si prefieres los clientes de bajo nivel de HTTP, mientras que crul o httr son mejores para un control de mayor nivel. El paquete crul es mantenido por rOpenSci. Recomendamos los paquetes mantenidos por rOpenSci webmockr para imitar las consultas HTTP, y vcr para almacenar en caché las consultas HTTP en los tests del paquete.

- Para parsear JSON, utiliza jsonlite en lugar de rjson o RJSONIO.
- Para parsear, crear y manipular XML, recomendamos fuertemente xml2 en la mayoría de los casos. Puedes consultar las notas de Daniel Nüst sobre la migración de XML a xml2.
- Para datos espaciales, el paquete sp debe considerarse obsoleto en favor de sf. Los paquetes rgdal, maptools y rgeos se retirarán a finales de 2023. Recomendamos el uso de las suites espaciales desarrolladas por las comunidades r-spatial y rspatial. Consulta este issue de GitHub para ver las discusiones pertinentes.

## 1.15 Control de versiones

- Los archivos fuente de tu paquete tienen que estar bajo control de versiones, más concretamente versionados con Git. Puede que el paquete gert te resulte útil, así como algunas de las funciones de usethis relacionadas con Git/GitHub; sin embargo, puedes utilizar git como quieras.
- Asegúrate de listar archivos innecesarios, como `.DS_Store`, en .gitignore. Puede el paquete gitignore es útil para esto.
- Una sección posterior de este libro incluye algunos consejos sobre el flujo de trabajo con git.

## 1.16 Problemas comunes en CRAN

Esta es una colección de problemas en CRAN que vale la pena evitar desde el principio.

- Asegúrate de que las palabras del título de tu paquete comiencen con mayúsculas (*Title Case*).
- No pongas un punto al final de tu título.
- No pongas “en R” o “con R” en tu título, ya que esto es obvio en los paquetes alojados en CRAN. Si a pesar de todo quieres que esta información aparezca en tu sitio web, revisa la documentación de `pkgdown` para saber cómo anular esta restricción.
- Evita empezar la descripción con el nombre del paquete o “Este paquete…”.
- Asegúrate de incluir enlaces a sitios web si utilizas una API, obtienes datos de una página web, etc… en el campo `Description` de tu archivo `DESCRIPTION`. Las URL deben ir entre simbolos <>, por ejemplo `<https://www.r-project.org>`.
- Tanto en el campo `Title` como en `Description` los nombres de los paquetes u otro software externo deben ir entre comillas simples (por ejemplo *Integración de ‘Rcpp’ para la Biblioteca de Álgebra Lineal Armadillo*).
- Evita tests y ejemplos que tarden en correr. Considera usar `testthat::skip_on_cran()` en los test que toman mucho tiempo para que se omitan en CRAN pero sigan corriendo localmente y en la integración continua.

- Incluye archivos de alto nivel como `paper.md` y archivos de configuración de integración continua en tu archivo `.Rbuildignore`.

### 1.16.1 Comprobaciones en CRAN

Una vez que tu paquete esté en CRAN, será comprobado regularmente en diferentes plataformas. Los fallos de estas comprobaciones, cuando no son falsos positivos, pueden hacer que el equipo de CRAN se ponga en contacto contigo. Puedes revisar el estado de las comprobaciones de CRAN con:

- el paquete `foghorn`.
- la API de comprobaciones de CRAN mantenida por rOpenSci, que proporciona etiquetas de estado.

## 1.17 Guía para Bioconductor

Si deseas que tu paquete se envíe a Bioconductor, o si tu paquete está en Bioconductor, consulta las Directrices de paquetes de Bioconductor.

## 1.18 Otras recomendaciones

- Si envías un paquete a rOpenSci a través del repo de software-review puedes dirigir tus preguntas al equipo de rOpenSci a través de issues o en nuestro foro de discusión.
- Antes de enviar un paquete, utiliza estas **buenas prácticas** (`goodpractice::gp()`) como guía para mejorar tu paquete, ya que la mayoría de las excepciones a estas tendrán que justificarse. Por ejemplo, usar `foo` no se recomendado generalmente y, por tanto, marcado como erróneo por `goodpractice`, pero podrías tener una buena razón para utilizarlo en tu paquete.
- Lee, incorpora y actúa según los consejos del capítulo *Guía de Colaboración*.

### 1.18.1 Aprender sobre el desarrollo de paquetes

#### 1.18.1.1 Libros

- *R Packages* (Paquetes de R) de Hadley Wickham y Jenny Bryan es un recurso excelente y fácil de leer sobre el desarrollo de paquetes que está disponible gratis en línea (e impreso - link a la versión anterior de Hadley Wickham, ya que, hasta junio de 2022, la nueva versión aún no se ha publicado).
- *Writing R Extensions* (Escribiendo extensiones de R) es la referencia canónica, normalmente la más actualizada, para crear paquetes de R.
- *Mastering Software Development in R* (Dominar el desarrollo de software en R) por Roger D. Peng, Sean Kross y Brooke Anderson.
- *Advanced R* (R avanzado) de Hadley Wickham

#### 1.18.1.2 Tutoriales

- El famoso artículo de Hilary Parker *Writing an R package from scratch* (Escribiendo tu propio paquete de R desde cero) o su versión actualizada por Tomas Westlake que muestra cómo hacer lo mismo de forma más eficiente utilizando `usethis`.
- esta descripción del flujo de trabajo por Emil Hvitfeldt.
- Esta ilustración de Matthew J Denny.

#### 1.18.1.3 Blogs

- Blog de R-hub.
- Algunos posts del blog de rOpenSci por ejemplo *"How to precompute package vignettes or pkgdown articles"* (Cómo pregenerar viñetas de paquetes o artículos de pkgdown).
- Sección *El rincón del desarrollo de paquetes* del boletín de rOpenSci.
- Algunos posts del blog de tidyverse por ejemplo *"Upgrading to testthat edition 3"* (Actualizando a la 3ra edición de testthat).

#### 1.18.1.4 MOOCs

Existe una especialización en Coursera correspondiente al libro de Roger Peng, Sean Kross y Brooke Anderson con un curso específico sobre paquetes de R.

# 2 Buenas prácticas de integración continua

Este capítulo explica qué es la integración continua (CI, por sus siglas en inglés) y luego resume nuestras recomendaciones sobre cómo usarla.

Junto con el capítulo anterior, forma parte de nuestras directrices para la revisión por pares de software.

## 2.1 ¿Qué es la integración continua?

La integración continua ejecuta tests sobre el software automáticamente. En el caso de rOpenSci, CI significa, en la práctica, que un conjunto de test se ejecutará automáticamente a través de GitHub, cada vez que hagas un *commit* o *pull request* a GitHub.

La CI automatiza la ejecución de tests globales de los paquetes, como `R CMD check` (ver tests). Es posible configurar la CI antes de escribir tus tests, entonces CI los ejecutará a medida que los envíes al repositorio.

## 2.2 ¿Por qué utilizar la integración continua (CI)?

Todos los paquetes de rOpenSci deben utilizar alguna forma de integración continua. Esto asegura que todos los *commits*, *pull requests* y nuevas *branches* pasan por `R CMD check`. Los resultados de todos los tests se muestran en la página del *pull request* en GitHub, lo cual proporciona otra capa de información sobre los problemas y protección para no aceptar cambios que rompan tu paquete. La integración continua de los paquetes de rOpenSci también debe estar vinculada a un servicio de cobertura de código que indique cuántas líneas son chequeadas por los tests unitarios.

Tanto el estado de los tests como el porcentaje de cobertura del código deben informarse mediante etiquetas en el *README* de tu paquete.

Los paquetes de R deben tener CI para todos los sistemas operativos (Linux, Mac OSX, Windows) si contienen:

- Código compilado
- Dependencias de Java

- Dependencias de otros lenguajes
- Paquetes con llamadas al sistema
- Procesamiento de texto, por ejemplo obtener nombres de personas (para encontrar problemas de codificación).
- Cualquier cosa que incluya llamadas al sistema de archivos / rutas de acceso

Ante la duda de si estos criterios se aplican a tu paquete, es mejor añadir CI para todos los sistemas operativos. La mayoría de los servicios de CI para paquetes de R lo permiten sin mucha complicación utilizando la configuración estándar.

## 2.3 ¿Qué servicio/s de integración continua usar?

Existen muchos servicios de integración continua. Algunos son servicios independientes (CircleCI, AppVeyor), mientras que otros están integrados a servicios de alojamiento del código o relacionados (GitHub Actions, GitLab, AWS Code Pipeline). Distintos servicios permiten distintas configuraciones de sistemas operativos.

GitHub Actions es una opción conveniente para quienes desarrollan paquetes de R y ya utilizan GitHub, ya que está integrado en la plataforma y es compatible con todos los sistemas operativos necesarios.
Hay acciones compatibles con el ecosistema R así como soporte de primera clase con el paquete{usethis}. Todos los paquetes enviados a rOpenSci para su revisión por pares son comprobados por nuestro propio sistema, `pkgcheck`, el cual se describe con más detalle en la Guía para autores/as. Estos tests también están disponibles como acción de GitHub en el repositorio `ropensci-review-tools/pkgcheck-action`. Es recomendable utilizar esta acción para confirmar que el paquete pasa todos los tests antes de enviarlo. Consulta nuestro blog para más información.

usethis se puede usar con otros sistemas de CI, aunque estas funciones están siendo deprecadas. rOpenSci también provee el paquete circle, el cual ayuda a configurar cadenas de CircleCI, y el paquete tic para la construcción de cadenas de CI más complicadas.

#### 2.3.0.1 Tests con diferentes versiones de R

Requerimos que los paquetes de rOpenSci no sólo se testeen usando la versión más reciente de R, sino también con la anterior y con la versión en desarrollo. Esto garantiza compatibilidad con R base, tanto con versiones futuras como con versiones pasadas.

Los detalles de cómo ejecutar tests utilizando diferentes versiones de R localmente se pueden encontrar en la viñeta de R-hub sobre la ejecución de Checks locales en Linux con Docker.

Puedes definir los detalles de los checks para cada una de las versiones utilizando una matriz de tests.

Si desarrollas un paquete que depende o está destinado para Bioconductor, puede que el paquete biocthis te resulte relevante.

#### 2.3.0.2 Minimizar los tiempos de construcción en CI

Puedes utilizar estos consejos para minimizar los tiempos de construcción en CI:

- Guardar los paquetes instalados en una caché. La acción r-lib/actions lo hace por defecto.

### 2.3.1 Travis CI (Linux y Mac OSX)

Recomendamos dejar de utilizar Travis.

### 2.3.2 AppVeyor CI (Windows)

Para la integración continua en Windows, consulta R + AppVeyor. Configúralo con `usethis::use_appveyor()`.

Aquí tienes consejos para minimizar el tiempo de compilación de AppVeyor:

- Guarda los paquetes instalados en una caché. Mira, por ejemplo, este archivo de configuración. AppVeyor usará la caché automáticamente si lo configuras con `usethis::use_appveyor()`.
- Activa rolling builds.

Ya no transferimos los proyectos de AppVeyor a la cuenta de AppVeyor de rOpenSci, así que después de transferir tu repo a la organización de GitHub "ropensci", la etiqueta será `[![AppVeyor Build Status](https://ci.appveyor.com/api/projects/status/github/ropensci/pkgname?branch=master&sv`

### 2.3.3 Circle CI (Linux y Mac OSX)

Circle CI es utilizado, por ejemplo, por el paquete rOpenSci `bomrang` como servicio de integración continua.

## 2.4 Cobertura de tests

La integración continua también debe incluir información de la cobertura de los tests a través de un servicio de testing como Codecov o Coveralls.

Recomendamos utilizar Codecov. Para activar Codecov para tu repo, ejecuta `usethis::use_github_action("test-` el cual va a crear un archivo `.github/workflows/test-coverage.yaml`. También tienes que dar acceso a Codecov a tu repositorio de GitHub, ver la guía de inicio rápido de Codecov para saber cómo hacerlo. Luego añade una etiqueta de estado de Codecov en la parte superior de tu README.md, puedes consultar *Status Badges* (etiquetas de estado) en la documentación de Codecov.

Actualmente, Codecov tiene acceso a todos los repositorios de GitHub de la organización ropensci por defecto. Cuando tu repositorio sea aceptado y transferido a ropensci, el acceso de Codecov debería transferirse automáticamente. Tendrás que actualizar la URL de la etiqueta para que apunte al repositorio alojado en ropensci.

Para más detalles, consulta el README del paquete **covr**, así como la documentación de `usethis::use_coverage()` y `usethis::use_github_action()`.

Si ejecutas la cobertura en varios servicios de CI los resultados se unirán.

## 2.5 Más servicios de CI: OpenCPU

Después de transferir el repo a la organización GitHub "ropensci" de rOpenSci, cada *push* se chequeará en OpenCPU y la persona que hace el *commit* recibirá una notificación por correo electrónico. Este es un servicio adicional de CI para que permite correr funciones en paquetes de R de forma remota a través de https://ropensci.ocpu.io/ utilizando la API de opencpu. Para más detalles sobre este servicio, consulta la página de ayuda de OpenCPU que también indica dónde hacer preguntas.

## 2.6 Aún más servicios de CI: rOpenSci docs

Después de la transferencia a la organización GitHub "ropensci" de rOpenSci, cada *push* al repo de GitHub disparará la construcción (o actualización) del sitio web de tu paquete usando pkgdown. Puedes encontrar el estado de este proceso en la URL `https://ropensci.r-universe.dev/ui#packages` y en el estado del commit. El sitio web estará en `https://docs.ropensci.org/package_name` (por ejemplo para `magick`). Si tu paquete tiene un archivo de configuración de pkgdown, rOpenSci docs lo usará para crear el sitio web, excepto para el tema, que se utilizará `rotemplate` paquete.

Por favor, informa sobre errores, haz preguntas y solicita nuevas funcionalidades sobre este servicio y sobre la plantilla en https://github.com/ropensci-org/rotemplate/.

# 3 Buenas prácticas de seguridad en el desarrollo de paquetes

Este es un capítulo en proceso que incluye consejos sobre la gestión de información confidencial en los paquetes y enlaces a lecturas adicionales.

## 3.1 Miscelaneos

Recomendamos el artículo Diez consejos rápidos sobre seguridad en Internet de Danielle Smalls y Greg Wilson.

## 3.2 Seguridad al acceder a GitHub

- Te recomendamos proteger tu cuenta de GitHub con autenticación de dos factores (2FA). Esto es *obligatorio* para miembros de la organización ropensci en GitHub y quienes colaboren de manera externa, así que asegúrate de habilitarla antes de que se apruebe tu paquete.
- También te recomendamos que compruebes regularmente quién tiene acceso al repositorio de tu paquete, y que elimines cualquier acceso no utilizado (como el de antiguos/as colaboradores/as).

## 3.3 https

- Si el servicio web al que se conecta tu paquete tiene opciones de https y http, usa https.

## 3.4 Credenciales dentro de paquetes

Esta sección incluye consejos para cuando desarrolles un paquete que interactúa con un recurso web que requiere credenciales (claves API, tokens, etc.). Consulta también la viñeta de `httr` sobre compartir credenciales.

### 3.4.1 Credenciales dentro de paquetes y la protección de usuarios/as

Supongamos que tu paquete necesita una clave para hacer consultas a una API en nombre de quien usa el paquete.

- En la documentación de tu paquete, orienta a la persona para que la clave de la API no se almacene en su archivo .Rhistory o en su script.
- Fomenta el uso de variables de entorno para almacenar la clave de la API (o incluso elimina la posibilidad de pasarla como argumento a las funciones). Puedes mencionar esta introducción a los archivos de arranque y la función `usethis::edit_r_environ()`.
- Otra posibilidad es usar o fomentar el uso del paquete `keyring` para almacenar variables en los sistemas de almacenamiento de credenciales del sistema operativo (que son más seguros que guardarlas en el archivo .Renviron). Para esto tu paquete tendría una función para guardar la clave en el keyring y una para recuperarla; alternativamente quien use tu paquete podría escribir `Sys.setenv(CLAVESUPERSECRETA = keyring::key_get("miservicio"))` al principio de su script.
- No imprimas la clave de la API en ningún mensaje, advertencia o error, ni siquiera en modo verboso.
- La plantilla de issues en GitHub debe indicar no compartir ninguna credencial. Si alguien comparte accidentalmente las credenciales en un issue, asegúrate de avisarle para que revoque la clave (es decir, pregúntale explícitamente si se ha dado cuenta de que ha compartido su clave).

### 3.4.2 Credenciales en paquetes y desarrollo

Durante el desarrollo del paquete tendrás que proteger tus credenciales igual que proteges las credenciales de quienes usan tu paquete, pero hay más cosas a considerar y tener en cuenta.

#### 3.4.2.1 Credenciales y consultas guardadas en los tests

Si utilizas `vcr` o `httptest` en los test para almacenar las respuestas de la API en caché, tienes que asegurarte de que las consultas grabadas o fixtures no contengan credenciales. Consulta la guía de seguridad de `vcr` y la guía "Redacting and Modifying Recorded Requests" de `httptest` e inspecciona tus consultas grabadas y fixtures antes de hacer un primer commit para asegurarte de que tienes la configuración correcta.

Como `vcr` es un paquete de rOpenSci, puedes publicar cualquier pregunta que tengas en el foro de rOpenSci.

#### 3.4.2.2 Compartir credenciales con servicios de integración contínua

Si utilizas un servicio de integración continua (CI por sus siglas en inglés), es posible que necesites compartir credenciales con el mismo.

Puedes almacenar las claves de la API como variables de entorno/credenciales consultando la documentación del servicio de CI.

Para más detalles y consejos sobre el flujo de trabajo, consulta al artículo de gargle "Managing tokens securely" y el capítulo de seguridad del libro HTTP testing in R.

Documenta los pasos que has seguido en CONTRIBUCIÓN.md para que tú, o alguien en el futuro, pueda recordar cómo hacerlo la próxima vez.

#### 3.4.2.3 Credenciales y colaboraciones

¿Qué pasa con los pull requests de colaboradoras/es externas/os? En GitHub, por ejemplo, los "secrets" sólo están disponibles para las acciones de GitHub para los pull requests iniciados desde el propio repositorio, no desde el fork. Los test que utilicen tus credenciales fallarán a menos que utilices algún tipo de respuesta simulada/en caché, por lo que es posible que quieras omitirlos dependiendo del contexto. Por ejemplo, en tu cuenta CI podrías crear una variable de entorno llamada `ESTE_SOY_YO` y luego omitir los test en función de la presencia de esta variable. Obviamente, esto significa que los tests del PR por parte del CI no serán exhaustivos, por lo que tendrás que comprobar el PR localmente para ejecutar todos los tests.

Documenta el comportamiento de tu paquete frente a PRs externos en CONTRIBUCIÓN.md por el bien de la gente que hace PRs y de la gente que los revisa (tú dentro de unas semanas, y otras/os autoras/es del paquete).

### 3.4.3 Credenciales y CRAN

En CRAN, omite las pruebas y los ejemplos que requieran credenciales utilizando `skip_on_cran()` y `dontrun` respectivamente.

También omite las viñetas que requieran credenciales.

## 3.5 Lecturas adicionales

Recursos útiles sobre seguridad:

- Encuentro de la comunidad rOpenSci sobre "Seguridad para R" (grabación, diapositivas, etc…). Ver en particular la lista de recursos);

- los proyectos relacionados con la seguridad de la unconf18;
- el paquete `notary`;
- el artículo de `gargle` “Managing tokens securely”

# Parte I

# Software Peer Review of Packages

# 4 Revisión por pares del software, ¿por qué? ¿qué?

Este capítulo contiene una introducción general a nuestro sistema de revisión por pares de software para paquetes, razones para enviar un paquete, razones para ofrecerse para revisar, por qué nuestras revisiones son abiertas, y agradecimientos a quienes participan del sistema.

Nuestro sistema se ha ampliado recientemente a la revisión por pares de software estadístico.

*Si utilizas nuestros estándares, listas de tareas, etc… al revisar software en otro lugar, informa a las personas a las que está destinado (por ejemplo, editoras/es de revistas, estudiantes, revisión interna de código) que estos materiales proceden de rOpenSci, y también cuéntanoslo en nuestro foro público, o por correo electrónico, si prefieres comentarlo en forma privada.*

## 4.1 ¿Qué es la revisión por pares de software de rOpenSci?

La suite de paquetes de rOpenSci recibe aportes tanto de personal de la organización, como por miembros de la comunidad, lo que significa que se nutre de una gran diversidad de habilidades y experiencias. Pero, ¿cómo asegurar un buen nivel de calidad? Ahí es donde entra en juego la revisión por pares de software: los paquetes aportados por la comunidad se someten a un proceso de revisión transparente, constructivo, no conflictivo y abierto. El mismo se sustenta principalmente en el trabajo de personas voluntarias: editoras/os asociadas/os gestionan el flujo de información y garantizan que los envíos progresen; personas que crean, envían y mejoran su paquete; revisoras/es (dos por envío), examinan el código y la experiencia de usuario. Esta entrada del blog, escrito por editoras/es de rOpenSci es una buena introducción a la revisión por pares de software de rOpenSci. Puedes encontrar más artículos en el blog de rOpenSci sobre el proceso y los paquetes que pasaron por el mismo en la etiqueta "software-peer-review".

Puedes reconocer los paquetes de rOpenSci que han sido revisados por pares mediante una etiqueta verde que dice "peer-reviewed" en su *README* (ej. el paquete restez), y por el icono azul en su descripción en la página de paquetes de rOpenSci, ambos contienen links a la revisión.

En cuanto a la parte técnica, aprovechamos GitHub al máximo: cada proceso de revisión es un *issue* en el repositorio ropensci/software-review. Por ejemplo, haz click aquí para leer el hilo de revisión del paquete `ropenaq`: el proceso es una conversación activa hasta que el paquete es aceptado, con dos revisiones externas como hitos importantes. Además, utilizamos funciones de GitHub como las

plantillas de *issues* (como plantilla de envío), y el etiquetado que utilizamos para seguir el progreso de los envíos (desde las revisiones de quien hace la edición hasta la aprobación).

## 4.2 ¿Por qué enviar tu paquete a rOpenSci?

- En primer lugar, y sobre todo, esperamos que envíes tu paquete para su revisión **porque valoras la devolución**. Nuestro objetivo es proporcionar una devolución útil a las personas que crearon el paquetes y que nuestro proceso de revisión sea abierto, no conflictivo y centrado en la mejora de la calidad del software.
- Una vez a aceptado, tu paquete seguirá recibiendo **apoyo de los miembros de rOpenSci**. Mantendrás la propiedad y el control de tu paquete, pero podemos ayudarte con los problemas de mantenimiento en curso, como los relacionados con las actualizaciones de R y las dependencias, y las políticas de CRAN.
- rOpenSci **difundirá tu paquete** a través de nuestra página web, blog y redes sociales. Los paquetes de nuestra suite también tienen un sitio web de documentación que se construye y despliega automáticamente después de cada *push*.
- Los paquete de rOpenSci **se pueden incluir** en otros repositorios como CRAN y BioConductor.
- Los paquetes de rOpenSci que son relevantes para el Journal of Open-Source Software y añaden un artículo corto de acompañamiento pueden beneficiarse, a discreción de las/os editoras/es de JOSS, de un proceso de revisión acelerado.
- Si escribes un **gitbook relacionado a tu paquete**, rOpenSci lo difundirá: el código fuentel mismo puede ser transferido a la organización de GitHub `ropensci-books` para ser listado en books.ropensci.org.

## 4.3 ¿Por qué revisar paquetes para rOpenSci?

- Como en cualquier proceso de revisión por pares, esperamos que elijas revisar **para contribuir a rOpenSci y a las comunidades científicas**. Nuestra misión de ampliar el acceso a los datos científicos y promover una cultura de investigación reproducible sólo es posible gracias a los esfuerzos voluntarios de miembros de la comunidad como tú.
- La revisión es una conversación bidireccional. Al revisar los paquetes, tendrás la oportunidad de **seguir aprendiendo buenas prácticas de desarrollo de las personas que crean paquetes y otras/os revisoras/es**.
- La naturaleza abierta de nuestro proceso de revisión te permite **establecer redes y conocer colegas y personas que colaboran**. Nuestra comunidad es amigable y está llena de miembros con conocimiento en el desarrollo de R y en muchas otras áreas de la ciencia y la informática científica con ganas de ayudar.
- Para ofrecerte para revisar paquetes, completa este breve formulario con tu información de contacto y tus áreas de experiencia. Siempre buscamos más personas con experiencia en el

desarrollo de paquetes en general y con experiencia en los campos en los que se utilizan los paquetes.

## 4.4 ¿Por qué las revisiones son abiertas?

Nuestros hilos de revisión son públicos. Todas las personas que participan (en roles de autoría, revisión y edición) conocen la identidad del resto y la comunidad en general puede ver o incluso participar en la conversación a medida que se produce. Esto proporciona un incentivo para ser minucioso y proporcionar revisiones constructivas y no conflictivas. Tanto autoras/es como revisoras/es afirman que disfrutan y aprenden más de este intercambio abierto y directo. También tiene el beneficio de construir una comunidad, ya que quienes participan tienen la oportunidad de interactuar de manera significativa con nuevas personas. Nuevas colaboraciones han nacido gracias a las ideas surgidas durante el proceso de revisión.

Somos conscientes de que los sistemas abiertos pueden tener inconvenientes. Por ejemplo, en la revisión académica tradicional la revisión por pares doblemente ciega puede aumentar la representación de autoras femeninas, lo que sugiere un sesgo en las revisiones abiertas. También es posible que quienes hacen la revisión sean menos críticos en la revisión abierta. Sin embargo, proponemos que la apertura de la conversación de la revisión proporciona un control de la calidad de la revisión y del sesgo; es más difícil hacer comentarios no fundamentados o subjetivos en público sin la cobertura del anonimato. En definitiva, creemos que tener una comunicación directa y pública entre las personas que desarrollaron el paquete y quienes lo revisan mejora la calidad y la imparcialidad de las revisiones.

Además, tanto autoras/es como revisoras/es tienen la posibilidad de contactar al equipo editorial por privado si tienen alguna duda o pregunta.

## 4.5 ¿Cómo se distingue un paquete que fue revisado?

- El *README* de tu paquete incluirá una etiqueta de revisión por pares que enlaza con el hilo de revisión.
- Tu paquete tendrá un sitio web de documentación en `docs.ropensci.org` que podrás enlazar en el archivo *DESCRIPTION*.
- El repositorio de tu paquete será transferido a la organización rOpenSci.
- Si quines realizan la revisión aceptan aparecer en el archivo *DESCRIPTION* sus metadatos mencionarán la revisión.

## 4.6 Editoras/es y revisoras/es

### 4.6.1 Equipo editorial asociado

El proceso de revisión por pares del software de rOpenSci está dirigido por

- 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 Equipo de revisión

Agradecemos a las siguientes personas que han ofrecido su tiempo y experiencia para revisar los paquetes enviados a rOpenSci.

Sam Albers · Toph Allen · Kaique dos S. Alves · Brooke Anderson · Alison Appling · 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 · 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 · Mario Gavidia Calderón · Brad Cannell · Kevin Cazelles · Scott Chamberlain · Cathy Chamberlin · Jennifer Chang · Pierre Chausse · Jorge Cimentada · Nicholas Clark · Chase Clark · Jon Clayden · Will Cornwell · Nic Crane · Enrico Crema · Ildiko Czeller · Kauê de Sousa · Christophe Dervieux · Amanda Dobbyn · Jasmine Dumas · Remko Duursma · Mark Edmondson · Paul Egeler · Evan Eskew · Salvador Fernandez · Alexander Fischer · Kim Fitter · Robert M Flight · Sydney Foks · Zachary Stephen Longiaru Foster · Auriel Fournier · Carl Ganz · Duncan Garmonsway · Sharla Gelfand · Monica Gerber · Duncan Gillespie · David Gohel · Rohit Goswami · Laura Graham · Charles Gray · Matthias Grenié · Corinna Gries · Hugo Gruson · W Kyle Hamilton · Ivan Hanigan · Jeffrey Hanson · 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 · Mike Johnson · Will Jones

· Max Joseph · Krunoslav Juraic · Soumya Kalra · Zhian N. Kamvar · Michael Kane · Andee Kaplan · Tinula Kariyawasam · Hazel Kavılı · Jonathan Keane · Christopher T. Kenny · Os Keyes · Aaron A. King · Michael Koontz · Bianca Kramer · Will Landau · Sam Lapp · Erin LeDell · Thomas Leeper · Sam Levin · Stephanie Locke · Marion Louveaux · Robin Lovelace · Julia Stewart Lowndes · Tim Lucas · Andrew MacDonald · Jesse Maegan · Tristan Mahr · Paula Andrea Martinez · Joao Martins · Ben Marwick · Claire Mason · Miles McBain · Lucy D'Agostino McGowan · Amelia McNamara · Elaine McVey · Bryce Mecum · François Michonneau · Mario Miguel · Helen Miller · Beatriz Milz · Jessica Minnier · Priscilla Minotti · Nichole Monhait · Kelsey Montgomery · Paula Moraga · Ross Mounce · Athanasia Monika Mowinckel · Lincoln Mullen · Matt Mulvahill · Maria Victoria Munafó · David Neuzerling · Dillon Niederhut · Rory Nolan · Kari Norman · Jakub Nowosad · Matt Nunes · Daniel Nüst · Joseph O'Brien · Paul Oldham · Samantha Oliver · Dan Olner · Jeroen Ooms · Luis Osorio · Philipp Ottolinger · Mark Padgham · Marina Papadopoulou · Edzer Pebesma · Thomas Lin Pedersen · Marcelo S. Perlin · Rafael Pilliard-Hellwig · Rodrigo Neto Pires · Lindsay Platt · Nicholas Potter · Etienne Racine · Manuel Ramon · Nistara Randhawa · David Ranzolin · Quentin Read · Neal Richardson · tyler rinker · Emily Robinson · David Robinson · Francisco Rodriguez-Sanchez · Julia Romanowska · Xavier Rotllan-Puig · Bob Rudis · Edgar Ruiz · Kent Russel · Alicia Schep · Klaus Schliep · Clemens Schmid · Patrick Schratz · Marco Sciaini · Heidi Seibold · Julia Silge · Margaret Siple · Peter Slaughter · Mike Smith · Tuija Sonkkila · Jemma Stachelek · Christine Stawitz · Irene Steves · Kelly Street · Alex Stringer · Michael Sumner · Chung-Kai Sun · Sarah Supp · Emi Tanaka · Jason Taylor · Filipe Teixeira · Andy Teucher · Jennifer Thompson · Joe Thorley · Tiffany Timbers · Tan Tran · Tim Trice · 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 · Lukas Weber · Marc Weber · Karissa Whiting · Stefan Widgren · Anna Willoughby · Saras Windecker · Luke Winslow · David Winter · Witold Wolski · Kara Woo · Marvin N. Wright · Bruna Wundervald · Lauren Yamane · Emily Zabor · Taras Zakharko · Hao Zhu · Chava Zibman · Naupaka Zimmerman · Jake Zwart · NA · Bri · Flury · NA · Pachá · Rich · Claudia · Jasmine · Lluís · NA · gaurav · NA

También agradecemos a las siguientes personas que han actuado como editoras/es invitadas/os.

Ana Laura Diedrichs · Hao Zhu

# 5 Políticas de la Revisión por Pares de Software

Este capítulo contiene las políticas de la Revisión por Pares de Software de rOpenSci.

En particular, encontrarás nuestras políticas sobre a la revisión por pares de software en si misma: el proceso de presentación de revisiones (incluyendo nuestras políticas sobre conflictos de intereses) y los objetivos y alcance del sistema de revisión por pares de software. Este capítulo también presenta nuestras políticas sobre propiedad y mantenimiento de los paquetes.

Por último, pero no menos importante, encontrarás el código de conducta de la Revisión por Pares de Software de rOpenSci.

## 5.1 Proceso de revisión

- Para que un paquete sea considerado para ingresar a rOpenSci, quienes crearon el paquete deben iniciar una solicitud en el repositorio ropensci/software-review.
- Los paquetes se revisan en cuanto a su calidad, adecuación, documentación y claridad, y el proceso de revisión es bastante similar al de la revisión de un artículo científico (véase nuestra guía de empaquetado y guía de revisión para más detalles). A diferencia de la revisión de un artículo científico, este proceso será una conversación continua.
- Una vez que se hayan resuelto todas las cuestiones y preguntas importantes que puedan abordarse haciendo esfuerzo razonable, la persona responsable de la edición tomará una decisión (aceptar, esperar o rechazar). Los rechazos suelen hacerse pronto (antes de que comience el proceso de revisión; ver la sección de objetivos y alcance), pero en raras ocasiones un paquete puede no ser aceptado después de la revisión. En última instancia, es decisión de quien hace la edición rechazar o no el paquete en función de cómo se aborden las revisiones.
- La comunicación entre quienes enviaron el paquete, quienes lo revisan y editan tendrá lugar principalmente en GitHub, aunque puedes optar por ponerte en contacto con quien hace la edición por correo electrónico o por Slack para algunas cuestiones. Cuando envíes un paquete, asegúrate de que tu configuración de notificaciones de GitHub sea correcta para que no te pierdas un comentario.

- Quien hace el envío del paquete puede elegir que su envío se ponga en espera (se le aplica la etiqueta *holding*). El estado de espera se revisará cada 3 meses, y después de un año el *issue* se cerrará.
- Si quien sea responsable del paquete no ha solicitado una etiqueta de espera, sino que simplemente no responde, podremos cerrar el tema en el plazo de un mes desde el último intento de contacto. Este intento incluirá un comentario etiquetando a la persona, pero también un correo electrónico utilizando la dirección de correo electrónico que aparece en el archivo *DESCRIPTION* del paquete. Este es uno de los raros casos en los que quien hace la edición intentará ponerse en contacto con el autor por correo electrónico.
- Para volver a enviar un paquete cuyo envío fue cerrado hay que iniciar un nuevo envío. Si el paquete sigue siendo relevante, la persona tendrá que responder a las revisiones iniciales antes de que la persona encargada de la revisión empiece a buscar nuevas personas para la revisión.

### 5.1.1 Publicar en otros lugares

- Te recomendamos fuertemente que envíes tu paquete para su revisión *antes de* publicarlo en CRAN o enviar un artículo de software que describa el paquete a una revista. Las devoluciones de las revisiones pueden dar lugar a importantes mejoras y actualizaciones de tu paquete, incluyendo el cambio de nombre y funcionalidad en las funciones. No consideramos que la publicación previa en CRAN o en otros lugares sea razón suficiente para no adoptar las recomendaciones de quienes hacen la revisión o edición.
- No envíes tu paquete para su revisión mientras éste o un artículo científico asociado también esté siendo revisado en otro lugar, ya que esto puede dar lugar a pedidos de cambios conflictivas.

### 5.1.2 Conflicto de intereses con quienes revisan o editan

Los siguientes criterios pretenden ser una guía de lo que constituye un conflicto de intereses para quien hace la edición o revisión. Existe un conflicto de intereses si:

- quienes tienen un papel importante en el paquete pertenecen a la institución o a una componente institucional (por ejemplo, un departamento) de quien potencialmente revisará o editará;
- en los últimos tres años, quien potencialmente revisará o editará ha colaborado o ha tenido cualquier otra relación profesional con cualquier persona que tenga un papel importante en el paquete;
- quien potencialmente revisará o editará es miembro del consejo asesor del proyecto que se está revisando;

- quien potencialmente revisará o editará recibiría un beneficio económico directo o indirecto si se acepta el paquete;
- quien potencialmente revisará o editará ha contribuido significativamente a un proyecto que se considera competencia;
- también hay un conflicto de interés de por vida para familiares, socias/os comerciales y estudiantes o quien le asesora o mentorea.

En el caso de que ninguno de los miembros del equipo editorial pueda realizar la edición, se invitará a una persona externa para que la realice.

## 5.2 Objetivos y alcance

rOpenSci tiene como objetivo apoyar paquetes que permiten la investigación reproducible y la gestión del ciclo de vida de los datos para quienes hacen ciencia. Los paquetes enviados a rOpenSci deben encajar en una o más de las categorías indicadas a continuación. El software estadístico también puede ser enviado para su revisión por pares, para lo cual tenemos un conjunto de directrices y normas. Mientras que el resto de este capítulo se aplica a ambos tipos de software, las categorías que aparecen a continuación son para el software general, y no para el estadístico. Si te queda claro si tu paquete encaja en una de las categorías generales o estadísticas, abre un *issue* como consulta previa a la presentación (**Ejemplos**).

Como se trata de un documento vivo, estas categorías pueden cambiar con el tiempo y no todos los paquetes incluidos anteriormente estarían en el ámbito de aplicación en la actualidad. Por ejemplo, los paquetes de visualización de datos ya no están incluidos. Aunque nos esforzamos por ser coherentes, evaluamos los paquetes caso por caso y podemos hacer excepciones.

Ten en cuenta que no todos los proyectos y paquetes de rOpenSci están incluidos en el ámbito de aplicación o pasan por una revisión por pares.

Los proyectos desarrollados por el personal o en conferencias pueden ser experimentales, exploratorios, abordar las prioridades de la infraestructura básica y, por tanto, no entrar en estas categorías. Busca la etiqueta de revisión por pares -ver más abajo- para identificar los paquetes revisados por pares en el repositorio de rOpenSci.


Figura 5.1: ejemplo de insignia verde de revisión por pares

### 5.2.1 Categorías de paquetes

- **obtención de datos**: Paquetes para acceder y descargar datos de fuentes en línea con aplicaciones científicas.

  Nuestra definición de "aplicaciones científicas" es amplia. Incluye servicios de almacenamiento de datos, revistas y otros servidores remotos, ya que muchas fuentes de datos pueden ser de interés para quienes hacen ciencia. Sin embargo, los paquetes de obtención de datos deben centrarse en las *fuentes* / *temas* de los datos y no en *servicios*. Por ejemplo, un cliente general para el almacenamiento de datos de *Amazon Web Services* no estaría en nuestro ámbito. (Ejemplos: **rotl**, **gutenbergr**)

- **extracción de datos**: Paquetes que ayudan a recuperar datos de fuentes no estructuradas, como texto, imágenes y PDF, así como a analizar tipos de datos científicos y resultados de equipamiento científico. Las bibliotecas estadísticas o de *machine learning* para el modelado o la predicción no suelen incluirse en esta categoría, como tampoco los analizadores de código. Los modelos entrenados que actúan como utilidades (por ejemplo, para el reconocimiento óptico de caracteres), podrían calificar. (Ejemplos: **tabulador** para extraer tablas de documentos PDF, **genbankr** para segmentar archivos de GenBank, **treeio** para la lectura filogenética en archivos de árboles filogenéticos, **lightr** para segmentar archivos de instrumentos espectroscópicos)

- **manipulación de datos**: Paquetes para procesar datos obtenidos de los formatos anteriores. Esta área no incluye las herramientas de manipulación de datos en general, como **reshape2** o **tidyr** o herramientas para extraer datos del propio código R. Más bien, se centra en herramientas para manipular datos en formatos científicos específicos generados a partir de flujos de trabajo científicos o exportados desde instrumentos científicos. (Ejemplos: **plateR** para leer datos estructurados como mapas de placas de instrumentos científicos, o **phonfieldwork** para procesar archivos de audio anotados para la investigación fonética)

- **depósito de datos**: Paquetes que apoyan el depósito de datos en repositorios de investigación, incluyendo el proceso de dar formato a los datos y la generación de metadatos. (Ejemplo: **EML**)

- **validación y comprobación de datos**: Herramientas que permiten la validación y comprobación automatizada de la calidad e integridad de los datos como parte de flujos de trabajo científico. (Ejemplo: **assertr**)

- **automatización del flujo de trabajo**: Herramientas que automatizan y encadenan flujos flujos de trabajo, como los sistemas de construcción y las herramientas para gestionar la integración continua. No incluye las herramientas generales para la programación literiaria (por ejemplo, las extensiones de R markdown que no están en los temas anteriores). (Ejemplo: **drake**)

- **control de versiones**: Herramientas que facilitan el uso del control de versiones en los flujos de trabajo científico. Ten en cuenta que esto no incluye todas las herramientas que interactúan con los servicios de control de versiones en línea (por ejemplo, GitHub), a menos que encajen en otra categoría. (Ejemplo: **git2rdata**)
- **gestión de citas y bibliometría**: Herramientas que facilitan la gestión de las referencias, como por ejemplo para escribir publicaciones científicas, crear CVs o atribuir de otra manera las contribuciones científicas, o acceder, manipular o trabajar de otra manera con datos bibliométricos. (Ejemplo: **RefManageR**)
- **capa de interfaz de software científico**: Paquetes que permiten correr programas no escritos en R utilizados para la investigación científica. Estos programas deben ser específicos de los campos de investigación, no utilidades informáticas generales. Las capas deben ser no triviales, en el sentido de que debe haber un valor añadido significativo por encima del uso de `system()` o de vincular el programa, ya sea en darle formato a las entradas entradas y salidas, en el manejo de datos, etc. La mejora del proceso de instalación, o la ampliación de la compatibilidad a más plataformas, puede constituir un valor añadido si la instalación es compleja. Esto no incluye las capas de interfaz de otros paquetes de R o bibliotecas de C/C++ que puedan incluirse en los paquetes de R. Animamos encarecidamente crear capas de interfaz a utilidades de código abierto y de licencia abierta; las excepciones se evaluarán caso por caso, teniendo en cuenta si existen opciones de código abierto. (Ejemplos: **babette**, **nlrx**)
- **herramientas de reproducibilidad de campo y laboratorio**: Paquetes que mejoran la reproducibilidad de los flujos de trabajo del mundo real mediante la estandarización y la automatización de los protocolos de campo y de laboratorio. Por ejemplo, el seguimiento y el etiquetado de las muestras, la generación de formularios y hojas de datos, la interconexión con los equipos de laboratorio o los sistemas de información, y la ejecución de diseños experimentales. (Ejemplo: **baRcodeR**)
- **enlaces a software de bases de datos**: Enlaces y capas de interfaz para las API genéricas de bases de datos (Ejemplo: **rrlite**)

Además, tenemos algunos *temas especializados* con un alcance algo más amplio.

- **datos geoespaciales**: Aceptamos paquetes centrados en el acceso a datos geoespaciales, la manipulación de datos geoespaciales y la conversión entre formatos de datos geoespaciales. (Ejemplos: **osmplotr**, **tidync**).
- **datos de texto**: Incluimos paquetes que procesan y manejan datos de texto y de lenguaje. Esto se limita al procesamiento/manipulación/gestión de textos (por ejemplo, tokenización, origen, conversión a y entre formatos de texto estructurados, metadatos de texto y acceso a repositorios, etc. ). El aprendizaje automático y los paquetes que implementan algoritmos de análisis de Procesamiento del Lenguaje Natural deben presentarse bajo la revisión por pares del software *estadístico*. El alcance de este tema no está totalmente definido, por lo que te

rogamos que abras una consulta previa a la presentación si estás pensando en presentar un paquete que entre dentro de este tema. (Ejemplo: **tokenizers**)

### 5.2.2 Otras consideraciones sobre el ámbito de aplicación

Los paquetes deben ser *generales* en el sentido de que deben resolver un problema de la forma más amplia posible, manteniendo una interfaz de usuario y una base de código coherentes. Por ejemplo, si varias fuentes de datos utilizan una API idéntica, preferimos un paquete que proporcione acceso a todas las fuentes de datos, en lugar de una sola.

Los paquetes que incluyan herramientas interactivas para facilitar los flujos de trabajo científico (por ejemplo, las shiny apps) deben tener un mecanismo para que el flujo de trabajo interactivo sea reproducible, como la generación de código o una API que permita la creación de *scripts*.

Para los paquetes que no están en el ámbito de rOpenSci, animamos a presentarlos en CRAN, BioConductor, así como en otras iniciativas de desarrollo de paquetes de R (por ejemplo cloudyr), y a las revistas de software como JOSS, JSS o la revista R, dependiendo del alcance actual de esas revistas.

Ten en cuenta que los paquetes desarrollados internamente por rOpenSci, a través de nuestros eventos o mediante colaboraciones, no están todos en el ámbito de nuestro proceso de revisión por pares de software.

### 5.2.3 Superposición de paquetes

rOpenSci fomenta la competencia entre paquetes, la bifurcación y la reimplementación, dado que éstas mejoran las opciones de quienes los usan en general. Sin embargo, como queremos que los paquetes del universo de rOpenSci sean nuestras principales recomendaciones para las tareas que realizan, pretendemos evitar la duplicación de la funcionalidad de los paquetes de R existentes en cualquier repo sin mejoras significativas. Un paquete que replica la funcionalidad de un paquete existente puede ser considerado para su inclusión en rOpenSci si mejora significativamente las alternativas en cualquier repositorio (RO, CRAN, BioC) al ser

- más abierto en cuanto a licencias o prácticas de desarrollo;
- más amplio en funcionalidad (por ejemplo, proporcionando acceso a más conjuntos de datos, proporcionando un mayor conjunto de funciones), pero no sólo duplicando paquetes adicionales;
- mejor en cuanto a usabilidad y rendimiento;
- activamente mantenido si las alternativas tienen poco o ningún mantenimiento activo.

Estos factores deben considerarse *en su conjunto* para determinar si el paquete es una mejora significativa. Un nuevo paquete no cumpliría esta norma sólo por seguir nuestras directrices sobre paquetes mientras otros no lo hacen, a menos que esto suponga una diferencia significativa en las áreas mencionadas.

Recomendamos que los paquetes destaquen en su *README* y/o en sus viñetas las diferencias con respecto a los paquetes que se solapan y las mejoras que introducen.

Animamos a quienes desarrollaron paquetes que no son aceptados debido al solapamiento a que sigan considerando su envío a otros repositorios o revistas.

## 5.3 Propiedad y mantenimiento de los paquetes

### 5.3.1 Rol del equipo de rOpenSci

Quienes crearon los paquetes mantienen esencialmente la misma propiedad que tenían antes de que su paquete se uniera al conjunto paquetes de rOpenSci. Estas personas seguirán manteniendo y desarrollando su software luego de su aceptación en rOpenSci. A menos que se les añada explícitamente con rol de colaboración, el equipo de rOpenSci no interferirá mucho en las actividades cotidianas. Sin embargo, el equipo puede intervenir con correcciones de errores críticos, o abordar cuestiones urgentes si quienes son responsables de los paquetes no responden de manera oportuna (ver la sección sobre la capacidad de respuesta de quienes mantienen los paquetes).

### 5.3.2 Capacidad de respuesta de quienes mantienen los paquetes

Si quienes matienen los paquetes no responden a tiempo a las solicitudes de corrección de de CRAN o de rOpenSci, enviaremos recordatorios varias veces, pero después de 3 meses (o un plazo más corto, dependiendo de lo crítica que sea la corrección) haremos los cambios.

Lo anterior es un poco impreciso, por lo que a continuación se presentan algunos ejemplos a considerar.

- Ejemplos en los que querríamos actuar con rapidez:
  - El paquete `foo` es importado por uno o más paquetes en CRAN, y `foo` tiene problemas, y por tanto generará problemas en los paquetes que dependan de `foo`.
  - El paquete `bar` puede no ser usado por otros paquetes que se encuentran en CRAN, pero es muy utilizado, por lo que arreglar rápidamente los problemas es muy importante.
- Ejemplos en los que podemos esperar más:

- El paquete `hello` no está en CRAN, o está en CRAN, pero no tiene dependencias inversas (otros paquetes no dependen de `hello`).
- El paquete `world` necesita algunas correcciones. La persona responsable ha respondido, pero simplemente está muy ocupada con un nuevo trabajo, u otra razón, y atenderá el problema pronto.

Instamos a quienes mantienen paquetes a que se aseguren de que reciben las notificaciones de GitHub, y de que los correos electrónicos del equipo de rOpenSci y de CRAN no van a su bandeja de correo no deseado. Invitaremos al Slack de rOpenSci a quienes hayan desarrollado paquetes que son incorporados a rOpenSci para charlar con el equipo y la comunidad de rOpenSci en general. Cualquier persona puede sumarse a la conversación con la comunidad rOpenSci en el foro de discusión de rOpenSci.

Si quienes desarrollaron un paquete abandonan su mantenimiento y se trata de un paquete utilizado activamente, consideraremos la posibilidad de solicitar a CRAN que transfiera el estatus de mantenedor del paquete a rOpenSci.

### 5.3.3 Compromiso de calidad

rOpenSci se esfuerza por desarrollar y promover software de investigación de alta calidad. Para asegurarnos de que tu software cumple nuestros criterios, revisamos todos los envíos como parte del proceso de revisión por pares del software, e incluso después de la aceptación seguiremos interviniendo con mejoras y correcciones de errores.

A pesar de nuestros esfuerzos por apoyar el software contribuido, los errores son responsabilidad de quien mantiene el paquete. El software con errores y sin mantenimiento puede ser eliminado de nuestra suite de paquetes en cualquier momento.

### 5.3.4 Eliminación de paquetes

En el improbable caso de que quien colabora con de un paquete solicite que su paquete sea retirado de la suite, nos reservamos el derecho de mantener una versión del paquete en nuestra suite con fines de archivo.

## 5.4 Ética, privacidad de los datos e investigación con sujetos humanos

Los paquetes de rOpenSci y otras herramientas se utilizan para diversos fines, pero nuestro foco está en las herramientas para la investigación. Esperamos que las herramientas permitan un uso ético por parte de quienes hacen investigación y tienen la obligación de adherirse a códigos éticos como la Declaración de Helsinki y el Informe Belmont. Quienes investigan son responsables del

uso que hacen del software, pero quienes desarrollan el software deben considerar el uso ético de sus productos y deben adherir a los códigos éticos para profesionales de la informática, como los expresados por el IEEE y ACM. Quienes colaboran en rOpenSci a menudo desempeñan el papel de investigación y desarrollo de software.

Pedimos que quienes desarrollan software se pongan en el papel de quienes investigan y consideren los requisitos de un flujo de trabajo ético utilizando el software. Dada la variación y el grado de cambios de los enfoques éticos para los análisis basados en Internet, es necesario evaluar cada caso en lugar de elaborar recetas. La Guía de Ética de la Asociación de Investigadores de Internet proporciona un marco sólido y animamos tanto a quienes mantienen paquetes como a quienes los revisan y editan, a utilizarlo a la hora de evaluar su trabajo. En general, la adhesión a los requisitos legales o reglamentarios mínimos puede no ser suficiente, aunque éstos (p. ej, GDPR), pueden ser relevantes. Quienes crean los paquetes deben dirigir a quienes los usan a los recursos pertinentes para el uso ético del software.

Algunos paquetes, debido a la naturaleza de los datos que manejan, pueden ser elegidos por quienes hacen la edición para pasar por un mayor escrutinio. Para éstos, quienes hacen la edición pueden exigir una funcionalidad adicional (o reducida), y una sólida documentación, valores por defecto y advertencias para dirigir a quienes usan el paquete a las prácticas éticas pertinentes. Los siguientes temas pueden merecer un mayor escrutinio:

- ***Poblaciones vulnerables***: Quienes crean paquetes y flujos de trabajo que tratan con información relacionada con poblaciones vulnerables tienen la responsabilidad de protegerlas de posibles daños.
- ***Datos personales o sensibles***: La divulgación de datos identificables o sensibles es potencialmente perjudicial. Esto incluye los datos "razonablemente identificables", que una persona motivada podría rastrear hasta la persona propietaria o creadora, incluso si los datos son anónimos. Esto incluye tanto los casos en los que los datos identificadores (por ejemplo, nombre, fecha de nacimiento) están disponibles como parte de los datos, y también si los seudónimos/nombres de usuario están vinculados a mensajes de texto completo, a través de los cuales se puede puede vincular a las personas individuales mediante referencias cruzadas con otros conjuntos de datos.

Aunque la mejor respuesta a los problemas éticos dependerá del contexto, se deben seguir estas directrices generales cuando se presenten los desafíos anteriores:

- Los paquetes deben adherirse a las condiciones de uso de la fuente de datos, expresadas en los términos y condiciones del sitio web, archivos "robots.txt", políticas de privacidad y otras otras restricciones relevantes, y enlazarlas de forma destacada en la documentación del paquete. Los paquetes deben proporcionar o documentar la funcionalidad para cumplir restricciones (por ejemplo, el *scrapeado* sólo de los servicios permitidos, el uso de limitación de velocidad adecuada en el código, los ejemplos o las viñetas). Ten en cuenta que aunque los Términos y Condiciones, las políticas de privacidad, etc…, pueden no ofrecer límites suficientes uso ético, pueden proporcionar un límite superior.

- Una herramienta clave para abordar los riesgos que plantea el estudio de poblaciones vulnerables o utilizar datos de identificación personal es ***el consentimiento informado***. Quienes crean los paquetes deben permitir la obtención del consentimiento informado cuando sea pertinente. Esto puede incluir proveer enlaces al método preferido por la fuente de datos para adquirir el consentimiento, información de contacto de quienes proveen los datos (por ejemplo, quienes moderan un foro), documentación de los protocolos de consentimiento informado, u obtener la aprobación previa para los usos generales de un paquete.

  Ten en cuenta que el consentimiento no se concede implícitamente sólo porque los datos sean accesibles. Los datos accesibles no son necesariamente públicos, ya que diferentes personas y contextos tienen diferentes expectativas normativas de privacidad (véase el trabajo de *Social Data Lab*).

- Los paquetes que acceden a información personal identificable deben tener especial cuidado en seguir las buenas prácticas de seguridad (por ejemplo, el uso exclusivo de protocolos de Internet seguros, mecanismos fuertes para almacenar credenciales, etc…).

- Los paquetes que acceden o manejan datos personales identificables o sensibles deben habilitar, documentar y demostrar los flujos de trabajo para la desidentificación el almacenamiento seguro, y otras buenas prácticas para minimizar el riesgo de daños.

A medida que las normas de privacidad de los datos y la investigación siguen evolucionando, agradeceremos los aportes de quienes desarrollan paquetes sobre las consideraciones específicas de su software y la documentación complementaria, como la aprobación de los comités de revisión ética de las universidades. Estos pueden adjuntarse a los *issues* de los envíos de paquetes o a las consultas previas al envío, o transmitirse directamente a quienes se encargan de la edición si es necesario. En general, sugerencias pueden enviarse como *issue* en el repositorio de este libro.

### 5.4.1 Recursos

Los siguientes recursos pueden ser útiles para quienes investigan, desarrollan paquetes, hacen la edición o revisión a la hora de abordar cuestiones éticas relacionadas con la privacidad y el software de investigación.

- El sitio web Declaración de Helsinki y el Informe Belmont proporcionan principios fundamentales para la práctica ética de quienes hacen investigación.
- Varias organizaciones ofrecen orientación sobre cómo trasladar estos principios en el contexto de la investigación en Internet. Entre ellas se encuentra la Guía de Ética de la Asociación de Investigadores de Internety la Guía de la NESH sobre Eica de la Investigación de Internet, y Directrices éticas de BPS para la investigación a través de Internet.
- Anabo et al (2019) proporcionan una visión general útil de estas guías.
- El Laboratorio de Redes Sociales ofrece una revisión general de alto nivel con datos sobre las expectativas de privacidad y uso en los foros sociales.

- Bechmann A., Kim J.Y. (2019) Big Data: A Focus on Social Media Research Dilemas. En: 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., y 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., y 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., y 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., y 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 Código de conducta

La comunidad de rOpenSci es nuestro mejor recurso. Tanto si colaboras habitualmente como si recién llegas, nos preocupamos por hacer de éste un lugar seguro para ti y te apoyamos en lo que necesites. Tenemos un Código de Conducta que se aplica a todas las personas que participan en la comunidad de rOpenSci, incluidos el equipo y la dirección de rOpenSci y a todos las formas de interacción en línea o en persona. El Código de Conducta se mantiene en el sitio web de rOpenSci.

# 6 Guía para quienes crean paquetes

Esta guía condensa el proceso de revisión por pares desde el punto de vista de quienes crean paquetes.

## 6.1 Planificar un envío (o una consulta previa al envío)

- ¿Esperas mantener tu paquete durante al menos 2 años, o poder identificar a una persona para que lo haga?
- Consulta nuestras políticas para evaluar si tu paquete cumple los criterios para ser incluido en nuestro conjunto de paquetes y no se superpone con otros.
  - Si no sabes si un paquete cumple con nuestros criterios, no dudes en abrir un issue para preguntar si el paquete es apropiado antes de presentarlo.
  - Ejemplo de respuesta sobre solapamiento](https://github.com/ropensci/software-review/issues/199#issuecomment-375358362). Considera también añadir algunos puntos sobre paquetes similares a tu documentación de paquetes.
- Por favor, considera cuál es el mejor momento en el desarrollo de tu paquete para presentarlo. Tu paquete debe estar lo suficientemente maduro como para que quienes lo revisen puedan evaluar todos los aspectos esenciales, pero ten en cuenta que la revisión puede resultar en cambios importantes.
- Te recomendamos fuertemente que envíes tu paquete para su revisión *antes* de publicarlo en CRAN o enviar un artículo de software describiéndolo a una revista. Es posible que tu paquete sufra mejoras importantes y actualizaciones fruto de la revisión, incluyendo cambios en el nombre de las funciones o incompatibilidades con versiones anteriores.
- No envíes tu paquete para su revisión mientras éste o un manuscrito asociado esté también en proceso de revisión en otro lugar, ya que esto puede dar lugar a solicitudes de cambios contradictorias.
- Ten en cuenta también el tiempo y el esfuerzo necesarios para responder a las revisiones: piensa en tu disponibilidad o en la de quienes colaboran con tu paquete en las semanas y meses siguientes al envío. Ten en cuenta que las revisiones son realizadas por personas voluntarias, y te pedimos que respetes su tiempo y esfuerzo respondiendo puntual y respetuosamente.

- Si utilizas etiquetas de repostatus.org (que recomendamos), envía tu paquete cuando esté listo para obtener la etiqueta *Active* en vez de *WIP*. Si utilizas etiquetas de ciclo de vida, el envío debe producirse cuando el paquete esté al menos en el estado *Maturing*.
- Para cualquier envío o consulta previa al envío, el *README* de tu paquete debería proporcionar suficiente información sobre el mismo (objetivos, uso, paquetes similares) para que quienes revisan el paquete puedan evaluar su alcance sin tener que instalarlo. Mejor aún, crea un sitio web de pkgdown para que puedan evaluar las funcionalidads detalladamente en línea.
- En la fase de envío, todas las funciones principales deben ser lo suficientemente estables como para estar completamente documentadas y testeadas. El *README* tiene que dar una buena impresión de tu paquete.
- El archivo *README* debe esforzarse por explicar las funcionalidades y los objetivos de tu paquete asumiendo poco o ningún conocimiento del dominio. Además, debe aclarar todos los temas técnicos, incluidas las referencias a otros software.
- Tu paquete seguirá evolucionando después de la revisión, el capítulo sobre *Evolución de paquetes* proporciona orientación sobre el tema.

## 6.2 Preparación para el envío

- Lee y sigue nuestra guía de estilo para paquetes y nuestra guía para la revisión para asegurarte de que tu paquete cumple nuestros criterios de estilo y calidad.
- No dudes en hacer cualquier pregunta sobre el proceso en general, o sobre tu paquete en particular en nuestro Foro de debate.
- Todos los envíos son revisados automáticamente por nuestro propio sistema para garantizar que los paquetes sigan nuestras directrices. Se espera que hayas corrido la función `pkgcheck` localmente para confirmar que el paquete está listo para ser enviado. Otra forma aún más fácil de asegurarse de que un paquete está listo para su envío es utilizar la acción de GitHub `pkgcheck` para ejecutar `pkgcheck` como una Acción de GitHub, como se describe en esta publicación en nuestro blog.
- Si hay algún test de `pkgcheck` que tu paquete no pueda pasar, explica los motivos en la plantilla de envío.
- Si crees que tu paquete es relevante para el Journal of Open Source Software (JOSS), no lo sometas a consideración de JOSS hasta que haya finalizado el proceso de revisión de rOpenSci: si los/as editores/as de JOSS consideran que tu paquete está dentro de su ámbito de aplicación, sólo se revisará el breve artículo que lo acompaña, (no el software que habrá sido revisado en extensión por rOpenSci en ese momento). No todos los paquetes de rOpenSci pueden aplicar a JOSS.

## 6.3 El proceso de envío

- Para presentar tu paquete a revisión tienes que abrir un nuevo *issue* en el repositorio de revisión de software y completar la plantilla.
- La plantilla comienza con una sección que incluye varias variables de estilo HTML (`<!---variable--->`). Éstas son utilizadas por nuestro bot (`ropensci-review-bot`) y no deben modificarse. Los valores de las variables deben completarse ente los puntos de inicio y fin, así:

```
<!---variable--->insertar valor aquí<!---end-variable>
```

- La comunicación entre quines envian el paquete, quienes lo revisen y quienes hagan la edición se dará principalmente en GitHub, para que el *issue* de revisión sirva de registro completo de la misma. Puedes contactarte con tu editor/a por correo electrónico o Slack si es necesario realizar una consulta privada (por ejemplo, preguntar cómo responder a una pregunta de quien está haciendo la revisión). No te pongas en contacto con los/as revisores/as fuera del *issue* sin antes preguntarles, dentro del mismo, si están de acuerdo con ello.
- *Cuando envíes un paquete, por favor asegúrate de tener configuradas las notificaciones de GitHub para que no te pierdas ningún comentario.*
- Los paquetes se checkean automáticamente con nuestro sistema `pkgcheck` sistema al ser enviados, el cual confirmará si está listo para ser revisado o no.
- Los paquetes enviados pueden estar en la rama *main/default*, o en cualquier otra rama no predeterminada. En este último caso, es recomendable, aunque no obligatorio, enviar el paquete a través de una rama dedicada llamada `ropensci-software-review`.

## 6.4 El proceso de revisión

- Un/a editor/a revisará tu envío en un plazo de 5 días laborables y te responderá con los siguientes pasos a seguir. Puede asignar el paquete a personas para que lo revisen, solicitar que el paquete se actualice para cumplir los criterios mínimos antes de la revisión, o rechazar el paquete porque el mismo no encaja en rOpenSci o porque se solapa con uno ya existente.
- Si tu paquete cumple los criterios mínimos, se le asignará de 1 a 3 personas para hacer la revisión, a quienes se les pedirá proporcionar revisiones en forma de comentarios sobre tu issue en un plazo máximo de 3 semanas.
- Te pedimos que respondas a estos comentarios en un plazo máximo de 2 semanas desde la última revisión presentada, pero puedes actualizar tu paquete o responder en cualquier momento. Tu respuesta debe incluir un enlace a la actualización del *NEWS.md* de tu paquete. Aquí tienes un ejemplo de respuesta. Fomentamos las conversaciones continuas entre autores/as y revisores/as. Consulta la guía de revisión para más detalles.

- Si algún cambio en el paquete puede modificar los resultados de `pkgcheck`, se puede solicitar un nuevo chequeo con el comando `@ropensci-review-bot check package`.
- Por favor, notifícanos inmediatamente si ya no puedes mantener tu paquete o responder a las revisiones. En ese caso, se espera que retractes el envío o que encuentres responsables alternativos para el mantenimiento del paquete. También puedes discutir los problemas de mantenimiento en el Slack de rOpenSci.
- Una vez que tu paquete sea aceptado, te proporcionaremos más instrucciones sobre la transferencia de tu repositorio al repositorio de rOpenSci.

Nuestro código de conducta es obligatorio para la participación en nuestro proceso de revisión.

# 7 Guía para quienes hacen una revisión

¡Gracias por aceptar revisar un paquete para rOpenSci! Este capítulo incluye nuestras recomendaciones para preparar, enviar y completar tu revisión.

Puedes ponerte en contacto con la persona encargada de la edición del paquete si tienes dudas sobre el proceso en general o tu revisión en particular.

Trata de completar tu revisión en un plazo de 3 semanas desde aceptar participar de la misma. Intentaremos recordar a quienes hacen una revisión las fechas de vencimiento próximas y pasadas. La persona encargada de la edición puede asignar revisores/as adicionales o alternativos/as si una revisión se retrasa excesivamente.

**La comunidad de rOpenSci es lo más importante. Nuestro objetivo es que las revisiones sean abiertas, no conflictivas y con el objetivo de mejorar la calidad del software. ¡Sé respetuoso/a y amable! Consulta nuestra guía para quienes realizan una revisión y el [código de conducta] (https://ropensci.org/code-of-conduct/) para más información.**

*Si utilizas nuestros estándares, listas de tareas, etc… al revisar software en otro lugar, informa a las/os destinatarias/os (por ejemplo, editoras/es de revistas, estudiantes, revisión interna de código) que proceden de rOpenSci, y cuéntanoslo en nuestro foro público, o en privado por correo electrónico.*

## 7.1 Voluntariarte para revisar

Gracias por tu deseo de participar en la revisión de software de rOpenSci como revisor/a.

Por favor, completa nuestro formulario de voluntariado.

Si ves una presentación vigente que es particularmente relevante para tus intereses, por favor envía un correo electrónico a `info@ropensci.org`, incluyendo el nombre del paquete, la URL del número de presentación y el nombre de la persona responsable de la edición. Sin embargo, ten en cuenta que las invitaciones a revisar quedan a discreción de la persona a cargo de la edición, y es muy posible que ya haya realizado invitaciones. Por favor, no te ofrezcas como voluntario/a para todos los temas, y no lo hagas a través de la interfaz de GitHub.

Para otras formas de contribuir, consulta la Guía de contribución de rOpenSci.

## 7.2 Preparar tu revisión

Todos los envíos generan un informe detallado sobre la estructura y funcionalidad del paquete, generado por nuestro paquete `pkgcheck`. Si el paquete ha cambiado sustancialmente desde las últimas revisiones, puedes solicitar una nueva revisión con el comando `@ropensci-review-bot check package`. Ten en cuenta que, al instalar el paquete para revisarlo, debes utilizar el argumento `dependencies = TRUE` de `remotes::install()` para asegurarte de que tienes todas las dependencias necesarias.

### 7.2.1 Directrices generales

Para revisar un paquete, empieza por copiar nuestra plantilla de revisión (o nuestra plantilla de revisión en español) y utilízala como lista de verificación general. Además de marcar los criterios mínimos, te pedimos que hagas comentarios generales sobre lo siguiente:

- ¿Cumple el código con los principios generales de la Guía de revisión de Mozilla?
- ¿Cumple el paquete con la guía de empaquetado de rOpenSci?
- ¿Puede mejorarse el estilo del código?
- ¿Puede reducirse el código duplicado, de haberlo?
- ¿Podría mejorarse la interfaz de usuario?
- ¿Podría aumentarse el rendimiento?
- ¿La documentación (instrucciones de instalación, viñetas, ejemplos, demos) es clara y suficiente? ¿Utiliza el principio de *múltiples puntos de entrada*? Es decir, tiene en cuenta que cualquier parte de la documentación puede ser la primera exposición de una persona con el paquete y/o la herramienta o datos que utiliza?
- ¿Los nombres de funciones y argumentos forman una API de programación común y lógica que sea fácil de leer y que se beneficie de herramientas de automcompletado?
- Si tienes datos propios relevantes o un problema relacionado, trabaja con el paquete. Puede que encuentres asperezas y/o casos de uso no contemplados.

Por favor, en tus revisiones dirígete con respeto y amabilidad hacia las personas que presentaron el paquete. Nuestro código de conducta es obligatorio para quienes participan en nuestro proceso de revisión. Esperamos que envíes tu revisión en un plazo de 3 semanas, según el plazo establecido por la persona a cargo de la edición. Por favor, ponte en contacto con esta persona directamente o mediante el *issue* de envío para informarle de posibles retrasos.

Te recomendamos usar herramientas automatizadas para facilitar tu revisión. Éstas incluyen

- Comprobar el informe inicial del paquete generado por nuestro bot `@ropensci-review-bot`.
- Comprobar los registros del paquete en sus servicios de integración continua (GitHub Actions, Codecov, etc.)
- Ejecutar `devtools::check()` y `devtools::test()` en el paquete para encontrar cualquier error que haya pasado desapercibido en la computadora de quien desarrolló el paquete.

- Revisar si los tests salteados son justificados (por ejemplo usar `skip_on_cran()` en tests que usan una API versus saltearse todos los tests en un determinado sistema operativo).
- Si el paquete no se envía a través de la *branch* principal o por defecto, recuerda cambiar a la *branch* enviada para revisar antes de comenzar tu revisión. En este caso, tendrás que usar la búsqueda local para revisar el código, ya que la búsqueda en GitHub está limitada a la *branch* por defecto. Además, la documentación alojada en un sitio web de `pkgdown` no va a estar necesariamente actualizada, por lo que recomendamos revisar la documentación del paquete localmente ejecutando `pkgdown::build_site()`.

Puedes volver a correr la comprobación del paquete de `@ropensci-review-bot` en cualquier momento enviando un comentario en el *issue* de revisión con el contenido "`@ropensci-review-bot check package`".

### 7.2.2 Interacciones fuera del hilo

Si interactúas con quienes desarrollaron el paquete y hablas de la revisión fuera de un *issue* de revisión (en chats, DMs, en persona, issues en el repositorio del proyecto), asegúrate de que tu revisión refleje y/o enlace los elementos de estas conversaciones que sean relevantes para el proceso.

### 7.2.3 Experiencia de revisiones anteriores

Si es la primera vez que revisas un paquete, te puede ser útil leer algunas revisiones anteriores. En general, puedes encontrar *issues* de presentación de paquetes aceptados aquí. Estas son algunas revisiones seleccionadas en inglés (ten en cuenta que tus reseñas no tienen que ser tan largas como estos ejemplos):

- `rtika` revisión 1 y revisión 2
- `NLMR` revisión 1 y revisión 2
- `bowerbird` comentario previo a la revisión, revisión 1, revisión 2.
- `rusda` revisión (previo a que tuviéramos una plantilla de revisión)

Puedes leer los artículos del blog sobre experiencias al revisar paquetes a través de este enlace. En particular esta publicación de Mara Averick (en inglés) habla como puedes proporcionar una devolución útil aún careciendo de experticia en el tema o en la implementación del paquete adoptando el rol de "usuario/a ingenuo/a" y preguntándote *"¿Qué creo que hace esta cosa? ¿lo hace? ¿qué cosas me causan rechazo?"* En otra entrada del blog Verena Haunschmid explica cómo alternó entre usar el paquete y revisar el código.

Como antiguo revisor y autor de paquetes, Adam Sparks escribió lo siguiente "[escribe] una buena crítica de la estructura del paquete y de las mejores prácticas de escritura de código. Si sabes

cómo hacer algo mejor, dímelo. Como desarrollador es fácil pasar por alto oportunidades de documentación, pero como revisor/a, tienes una visión diferente. Eres una persona que puede dar devolución. ¿Qué no está claro en el paquete? ¿Cómo puede hacerse más claro? Si lo utilizas por primera vez, ¿es fácil? ¿Conoces otro paquete R que quizás debería utilizar? ¿O hay alguno que esté utilizando y que quizás no debería? Si puedes contribuir al paquete, ofrécete".

### 7.2.4 Paquete de ayuda para quienes hacen una revisión

Si trabajas en RStudio, puedes agilizar tu flujo de trabajo de revisión utilizando el paquete `pkgreviewr` creado por la editora asociada Anna Krystalli. Digamos que aceptas revisar el paquete `refnet`, escribirías

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

Esto

- clonará el repositorio de GitHub del paquete `refnet`,
- creará un proyecto de revisión, que contendrá una notebook para que rellenes, y la plantilla de revisión.

Ten en cuenta que si el paquete no se envía a través de la *branch* principal o por defecto, tienes que cambiar a la *branch* usada para enviar antes de comenzar tu revisión.

### 7.2.5 Devoluciones sobre el proceso

Te animamos a que preguntes y des tu opinión sobre el proceso de revisión en nuestro foro.

## 7.3 Enviar la revisión

- Cuando tu revisión esté completa, agrégala como comentario en el *issue* de revisión del paquete.
- Comentarios adicionales son bienvenidos en el mismo *issue*. Aspiramos a que las revisiones de paquetes funcionen como una conversación continua entre las partes en lugar de una única ronda de revisiones típica de los manuscritos académicos.

- También puedes crear *issues* o *pull requests* directamente al repositorio del paquete si lo deseas, pero si lo haces, por favor, menciónalo y vincúlalo en el hilo de revisión de software para que tengamos un registro y un texto centralizados de tu revisión.
- Por favor, al terminar incluye una estimación de cuántas horas le has dedicado a tu revisión.

## 7.4 Seguimiento de la revisión

Las personas que enviaron el paquete a revisón deben responder en un plazo de 2 semanas con sus cambios en el paquete en respuesta a tu revisión. En esta fase, te pedimos que respondas si los cambios abordan suficientemente las cuestiones planteadas en tu revisión. Fomentamos el debate continuo entre autores/as y revisores/as, y también puedes pedir que quiene sestán a cargo de la revisión aclaren cosas en el mismo hilo.

- Utilizarás la plantilla de aprobación.

# 8 Guide for Editors

Software Peer Review at rOpenSci is managed by a team of editors. We are piloting a system of a rotating Editor-in-Chief (EiC).

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.

## 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.
- We also rotate Editor-in-Chief responsibilities (first-pass scope decisions and assigning editors) amongst the board about quarterly.
- You do not have to keep track of other 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 (e.g. you know of a blog post tackling how to add images to package docs).
  - Concerns related to e.g. the speed of the process should be tackled by the editor-in-chief so that's who you'd turn to for such questions.

## 8.2 Handling Editor's Checklist

### 8.2.1 Upon submission:

- If you're the EiC or the first editor to respond, assign an editor with a comment of `@ropensci-review-bot assign @username as editor`. This will also add tag `1/editor-checks` to the issue.
- Submission will automatically generate package check output from ropensci-review-bot which should be examined for any outstanding issues (most exceptions will need to be justified by the author in the particular context of their package.). If you want to re-run checks after any package change post a comment `@ropensci-review-bot check package`.
- 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 that template has been properly filled out.
- 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.
- Check that mandatory parts of template are complete. If not, direct authors toward appropriate instructions.
- For packages needing continuous integration on multiple platforms (cf criteria in this section of the CI chapter) make sure the package gets tested on multiple platforms (having the package built on several operating systems via GitHub Actions for instance).
- 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. Example of editor's checks.
- Ideally, the remarks you make should be tackled before reviewers start reviewing.
- If initial checks show major gaps, request changes before assigning reviewers. If the author mentions changes might take time, apply the holding label via typing `@ropensci-review-bot put on hold`. You'll get a reminder every 90 days (in the issue) 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 might tweet about the reviewer search.

##### 8.2.2.2.2 Tips for reviewer search in Airtable

Your tools for searching reviewers in the Airtable are

- filters (please remove them once you're done), see example below.
- creating a duplicated view, depending on your Airtable skills.

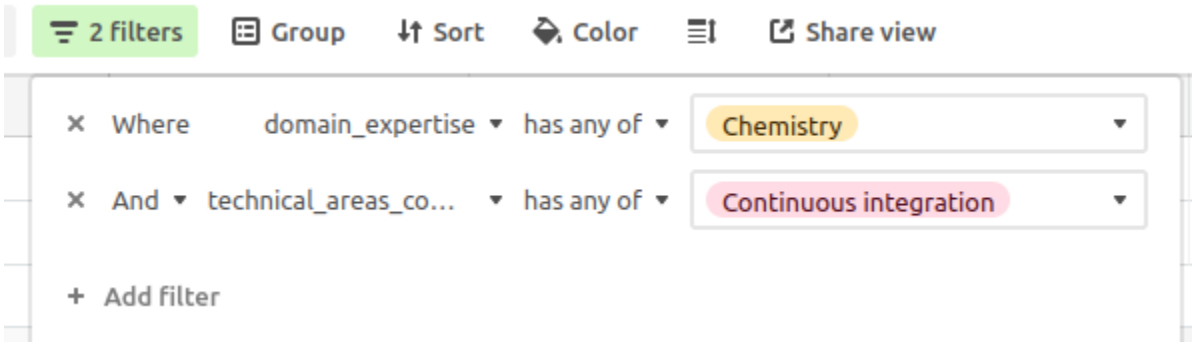


Figura 8.1: Screenshot of the Airtable filters interface with a filter on domain expertise that has to include chemistry and technical areas that have to include continuous integration

Have a look at the `last_time_contacted` and `last_answer` columns before contacting someone! If someone recently refused because they were busy, it might be best to abstain, whereas someone who refused because of a COI could be contacted again without waiting too long.

#### 8.2.2.2.3 Storing information from the reviewer search in Airtable

For people listed in the Airtable, if you contact them about a review please update the `last_time_contacted` column, and enter the category corresponding to their answer in `last_answer`.

Only add people to Airtable if they accept to review (otherwise, people should volunteer themselves by filling the Airtable form). Do not enter any information other than GitHub username, name and email address yourself. Point reviewers to the Airtable form.

#### 8.2.2.2.4 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 blind 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/softwa review/issues/329#issuecomment-809783937 the comment would be `@ropensci-review-bot submit review https://github.com/ropensci/software-review/issues/329#issuecomment-80 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 owner opposes transfer, add a line with its address 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

The EiC serves 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/advice). In details, the EiC plays the following roles

- Watches all issues posted to the software-review repo (either subscribe to repo notifications on GitHub, or watch the `#software-peer-review-feed` channel on Slack).
- Tags issue with `0/editorial-team-prep`
- Calls `@ropensci-review-bot check srr` on pre-submission enquiries for statistical software. See corresponding *Stats Dev Guide* chapter for details.
- Assigns package submissions to other editors, including self, to handle. Mostly this just rotates among editors, unless the EiC thinks an editor is particularly suited to a package, or an editor declines handling the submission due to being too busy or because of conflicting interests.

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

- Monitors pace of review process and reminds other editors to move packages along as needed.
- On assuming EiC rotation, reviews status of current open reviews and reminds editors to respond or update status as needed.
- Responds to issues posted to the software-review-meta repo
- Makes decisions on scope/overlap for pre-submission inquiries, referrals from JOSS or other publication partners, and submissions if they see an ambiguous case (This last case may also be done by handling editors (see below)). To initiate discussion, this is posted to the rOpenSci Slack editors-only channel along with a small summary of what the (pre-)submitted/referred submission is about, what doubts the EiC has i.e. digesting information a bit. If after one day or two the EiC feels they haven't received enough answers, 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 the out-of-scope decision, write an issue comment `@ropensci-review-bot out-of-scope`.

- Requests a new EiC when their rotation is up (set a calendar reminder ahead of your expected end date and ask for volunteers in the editors' Slack channel)

### 8.3.1 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

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

### 8.3.2 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: Dev
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
```

#### 8.6.2.2 Authorship

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

# 9 Gestión editorial

Guía para gestionar el equipo editorial.

## 9.1 Reclutar nuevas personas para la edición

Reclutar nuevas personas y mantener un equipo editorial funcional y equilibrado es responsabilidad de quien Lidere la Revisión de Software con el apoyo y asesoramiento del comité editorial.

Estos son los pasos a seguir:

- Inicia un canal privado para discutir la invitación; de este modo, el historial de conversación no queda disponible en el canal al que luego se unirán los nuevos miembros del equipo, lo que podría ser incómodo.
- Haz un *ping* a las personas del equipo editorial para asegurarte de que reciban una notificación, ya que se trata de un tema importante.
- Espera a que la mayoría de del equipo intervenga antes de invitar a alguien. Dales una semana para que respondan.

## 9.2 Invitar a una nueva persona al equipo

- Quienes sean candidatas/os pueden empezar siendo editoras/es invitadas/os. Haz la invitación como invitarías a cualquier editora/o invitada/o por otros motivos.
- Si una candidata/o fue editora invitada/o, evalúa cómo ha sido el proceso una vez abordada la presentación. Vuelve a pedir consejo a otras personas del equipo.
- Envía un correo electrónico:

```
Nos gustaría invitarte a formar parte del consejo editorial de rOpenSci como miembro en plen
Creemos que serías una magnífica incorporación al equipo.

[SI FUE EDITORA INVITADA/O: Ya conoces el rol de edición ya que has sido editora invitada/o]
Pedimos a las/os editoras/es que se comprometan de manera informal a prestar servicio durant
```

```
A corto plazo, cualquier persona puede rechazar encargarse de un paquete o decir: "Estoy muy

Además de encargarse de los paquetes, las personas en el equipo editorial opinan sobre las de
Generalmente usamos Slack, donde esperamos que los miembros del equipo participen regularment
Tenemos convocatorias anuales al equipo editorial.
También rotamos las responsabilidades de Editora en Jefe (quien toma decisiones de alcance en
Tendrás la oportunidad de entrar en esta rotación cuando lleves un tiempo en el equipo, norma
Algunos mimebros del equipo también asumimos proyectos más grandes para mejorar el proceso de

¡Esperamos que te unas al equipo!
Es un momento emocionante para la revisión por pares en rOpenSci.

Por favor, reflexiónalo, pregúntanos cualquier duda que tengas y haznos saber si puedes unirt

Te deseamos lo mejor,
[NOMBRE], en nombre del Equipo Editorial de rOpenSci.
```

## 9.3 Incorporar una nueva persona al equipo editor

- Informa a la/el *community manager* de rOpenSci para
  - agregar a la/o editora al sitio web de rOpenSci.
  - preparar una artículo de presentación en el blog.
- Solicita que la nueva persona active la autenticación de dos factores (2FA) para GitHub si aún no lo han hecho.
- Invítala a la organización GitHub de rOpenSci, y a los equipos `editors` y `data-pkg-editors` o `stats-board`, según corresponda. Esto le dará los permisos adecuados y le permitirá recibir notificaciones específicas del equipo.
- Dale acceso a la base de datos AirTable de revisión de software.
- Dale acceso al canal privado del Equipo Editorial en el espacio de trabajo de Slack de rOpenSci (y al espacio de trabajo de Slack en general si no lo tenía previamente, en ese caso pregunta a la/el *community manager* de rOpenSci).
- Publica un mensaje de bienvenida en el canal, haciendo ping a `@editors`.
- Agrégala al equipo `editors` de Slack para que `@editors` también le envíe una notificación.
- Añade su nombre a:
  - la lista de autoras/es de dev_guide.

  - el capítulo dev_guide que introduce la revisión de software (en dos lugares de este archivo: como editoras/es y un poco más abajo para eliminar su nombre de la lista de revisoras/es).
  - el *README* de revisión de software (también en dos lugares de este archivo!) Tanto la `dev_guide` como el *REAMDE* de revisión de software se renderizan automáticamente con integración continua.

## 9.4 Dar de baja a un miembro del Equipo Editorial

- Agradece su trabajo.
- Elimína a la persona del canal de editoras/es y del equipo `@editors` de Slack.
- Elimínala a la persona del equipo `editors` y del sub-equipo correspondiente en GitHub.
- Informa a la/el *community manager* de rOpenSci o a algún otro miembro del personal para mover a la persona a la sección de ex-integrantes en el sitio web.
- Elimina su acceso al espacio de trabajo de Airtable.
- Elimína a la persona de:
  - el capítulo dev_guide que introduce la revisión de software (en dos lugares de este archivo: como editoras/es y un poco más abajo para eliminar su nombre de la lista de revisoras/es).
  - el *README* de revisión de software (también en dos lugares de este archivo!) Tanto la `dev_guide` como el *REAMDE* de revisión de software se renderizan automáticamente con integración continua.

# Parte II

# Maintaining Packages

# 10 Guía de colaboración

Colaborar con otras personas mejorará tu paquete, y si les das a algunas de ellas rol de autoría y permisos de escritura en el repositorio, tu paquete se desarrollará de forma más sostenible. Además, ¡trabajar en equipo puede ser muy agradable!

Los consejos para colaboración de este capítulo están divididos en una sección sobre cómo hacer que la infraestructura de tu repo sea accesible para la contribución y la colaboración (código de conducta, directrices de contribución, etiquetas de *issue*) y una sección sobre cómo colaborar con nuevas personas que quieran contribuir, en particular en el contexto de la organización "ropensci" de rOpenSci en GitHub.

Además de estos consejos, en su mayoría técnicos, es importante recordar que hay que ser amable y tener en cuenta la perspectiva del resto, especialmente cuando sus prioridades difieren de las tuyas.

## 10.1 Haz que tu repositorio sea accesible a las contribuciones y colaboración

### 10.1.1 Código de conducta

Tras el traslado a nuestra organización de GitHub, el Código de Conducta de rOpenSci se aplicará a tu proyecto. Por favor, añade este texto al *README*

```
Ten en cuenta que este paquete se publica con un [Código de Conducta para contribuciones](
Al contribuir a este proyecto, te comprometes a cumplir sus condiciones.
```

Y eliminar el código de conducta actual del paquete, si lo hubiera.

### 10.1.2 Guía de contribución

Puedes utilizar guías de *issue*, *pull request* y de contribución. Tener un archivo de contribución como `.github/CONTRIBUTING.md` o `docs/CONTRIBUTING.md` es obligatorio. Una forma fácil de insertar una plantilla para una guía de contribución es la función `use_tidy_contributing()` del paquete `usethis` que inserta esta plantilla como `.github/CONTRIBUTING.md`. Un ejemplo más extenso es

esta plantilla de Peter Desmet o las completas páginas wiki de GitHub para el paquete mlr3. Estas y otras plantillas generalmente tendrán que ser modificadas para su uso con un paquete rOpenSci, en particular haciendo referencia y enlazando con nuestro Código de Conducta como se describe en otra sección de este libro. Modificar una guía de contribución genérica para añadir un toque propio también tiende a hacer que parezca menos impersonal y más sincera. Las preferencias personales en una guía de contribución incluyen

- Preferencias de estilo. Sin embargo, puedes preferir que el estilo sea uno automático (a través de lintr o styler) o bien arreglar el estilo de código por tu cuenta especialmente si no utilizas un estilo de código popular como el estilo de código tidyverse.
- Infraestructura, como roxygen2.
- Preferencias de flujo de trabajo. Por ejemplo, abrir un *issue* antes de hacer un *pull request*.
- Una declaración de alcance, como en el paquete skimr.
- Creación de una cuenta de prueba, tests simulados, enlace a documentación externa.

rOpenSci recomienda que las guías de contribución incluyan una declaración del ciclo de vida, que aclare la visión y las expectativas para el desarrollo futuro del paquete, como en este ejemplo. Los paquetes estadísticos deben tener una declaración de ciclo de vida, como se especifica en Normas Generales de Estadística G1.2. Ese enlace proporciona una plantilla para una declaración de ciclo de vida sencilla. Los archivos CONTRIBUTING.md también pueden describir cómo se reconocen las contribuciones (ver esta sección).

Te animamos a que dirijas las consultas y comentarios que no sean informes de errores o pedidos de nueva funcionalidad al foro de rOpenSci después de asegurarte de que verás esas preguntas en el foro. El foro puede usarse para hacer preguntas sobre cómo usar el paquete e informar de sus casos de uso, y puedes suscribirte a categorías y etiquetas individuales para recibir notificaciones sobre tu paquete. No dudes en mencionar esto en los documentos de tu paquete, ya sea en la guía de contribución o en la plantilla de *issue*.
Pide etiquetar las publicaciones con el nombre del paquete.

Una vez que un *pull request* está más cerca de ser aceptado, puedes utilizar un flujo de trabajo de *GitHub Actions PR* para estilizar el código con styler.

### 10.1.3 Gestión de *issues*

Utilizando las funciones de GitHub en torno a los *issues* puedes hacer que sean más encontrables y a hacer pública tu hoja de ruta.

#### 10.1.3.1 Plantillas de *issue*

Puedes utilizar una o varias plantilla(s) de *issue* para que los informes de errores o pedido de funcionalidad sean más fáciles de completar. Cuando hay varias plantillas de *issue*, éstas se muestran en un menú al hacer click en el botón de "nuevo *issue*"

Incluso puedes configurar una de las opciones para que apunte a algún lugar fuera de tu repositorio (por ejemplo, un foro de discusión).

Consulta la Documentación de GitHub.

#### 10.1.3.2 Etiquetado de *issues*

Puedes utilizar etiquetas como "se necesita ayuda" y "buen primer *issue*" para que seas más fáciles de encontrar, incluso por gente que llega por primera vez a tu repo. Consulta este artículo de GitHub. También puedes utilizar la etiqueta "Principiante". Consulta ejemplos de issues para principiantes en todos los repos de ropensci.

#### 10.1.3.3 Resaltado de *issues*

Puedes resaltar hasta 3 issues por repositorio que aparecerán en la parte superior de tu gestor de *issues* como bonitas tarjetas. Puede ayudar a anunciar cuáles son tus prioridades.

#### 10.1.3.4 Hitos

Puedes crear hitos y asignarles *issues*, lo que ayuda a ver lo que planeas para la próxima versión de tu paquete, por ejemplo.

### 10.1.4 Comunicación

Puedes dirigir a quienes usan tu paquete al foro de rOpenSci, si lo supervisas, o habilitar Discusiones de GitHub para el repositorio de tu paquete. Las discusiones de GitHub pueden convertirse en *issues* si fuera necesario (y al revés, los *issues* pueden convertirse en discusiones).

## 10.2 Trabajar en equipo

Lo primero lo primero: ¡estate al tanto de tu repositorio de GitHub!

- no te olvides de **vigilar tu repositorio de GitHub** para que te notifiquen los *issues* o *pull requests* (si trabajas por períodos, puedes informarlo en la guía de contribución).
- no olvides hacer *push* de las actualizaciones que tengas localmente.
- desactiva los test que fallan si no puedes arreglarlos, ya que crean ruido en los *pull requests* que pueden desconcertar quienes comienzan a colaborar.

### 10.2.1 Incorporación de miembros al equipo

No hay una regla general en rOpenSci sobre cómo debes incorporar a quienes colaboran en tu paquete al equipo. Deberías aumentar sus derechos en el repositorio a medida que ganes confianza, y definitivamente deberías reconocer las contribuciones (ver esta sección).

Puedes pedir a las nuevas personas que se incorporen que hagan PRs (ver la siguiente sección para evaluar un PR localmente, es decir, más allá de las comprobaciones de CI) a *dev/main* y evaluarlos antes de aceptarlos, y después de un tiempo dejar que hagan *push* a *main*. También puede que quieras mantener un sistema de revisión de PRs… ¡incluso para ti una vez que tengas miembros de equipo!

Jim Hester ofrece un posible modelo de incorporación de miembros de equipo en su repositorio de `lintr`.

Si tu problema es *reclutar* nuevas personas, puedes publicar una convocatoria abierta como la de Jim Hester en Twitter, GitHub y, como responsable de un paquete de rOpenSci, puedes pedir ayuda en el Slack de rOpenSci y solicitar al equipo de rOpenSci ideas para reclutar ayuda.

### 10.2.2 Trabajar con otras personas (y incluyendo tú en el futuro)

Las *branches* son baratas. Utilízalas mucho cuando desarrolles funciones, pruebes nuevas ideas o arregles problemas.

Si sigues el desarrollo troncal, "agregarás pequeñas y frecuentes actualizaciones" a tu *branch* principal. Ver también la documentación del Flujo de GitHub y el flujo de GitLab. Es posible que quieras incrementar los números de versión (en `DESCRIPTION`) frecuentemente. Un aspecto particular del trabajo con otras personas es la revisión de *pull requests*. Pueden ver algunas guías útiles en:

- *The Art of Giving and Receiving Code Reviews (Gracefully)* (El arte de dar y recibir revisiones de código (con elegancia)), por Alex Hill

- Documentación de GitHub sobre revisiones de PR

Para hacer y revisar *pull requests* recomendamos explorar las funciones del paquete usethis.

### 10.2.3 Atribuye con generosidad

Si una persona contribuye a tu repositorio, considera añadirla en el archivo DESCRIPTION, con el rol de contribución ("ctb") para pequeñas contribuciones o autoría ("aut") para contribuciones mayores. Tradicionalmente, cuando se cita un paquete en una publicación científica, sólo se enumeran a quienes están como "aut", no quienes están como "ctb". Los sitios web creados con `pkgdown` listan en la página de inicio sólo a las personas con rol de autoría ("aut"). El resto están listadas en una página dedicada.

Como mínimo, considera la posibilidad de añadir el nombre de quien contribuyó cerca de la línea de la función o corrección de errores en el archivo NEWS.md.

Te recomendamos reconozcas estas contribuciones generosamente, ya que es algo agradable y porque hará que sea más probable que la gente vuelva a contribuir a tu paquete o a otros repos de la organización.

Como recordatorio de nuestra guía de empaquetado si tu paquete fue revisado y crees que quienes lo hicieron han hecho una contribución sustancial al desarrollo de tu paquete, puedes usar el campo `Authors@R` con el rol de revisión (`"rev"`), así

```
persona("Bea", "Hernández", rol = "rev",
comment = "Bea revisó el paquete (v. X.X.XX) para rOpenSci, véase <https://github.com/ro
```

Sólo incluye a quienes hicieron la revisión con su consentimiento. Lee más en esta entrada del blog *Thanking Your Reviewers: Gratitude through Semantic Metadata* (Agradeciendo a quienes revisaron tu paquete: La gratitud a través de los metadatos semánticos). Ten en cuenta que 'rev' generará una NOTA de CRAN a menos que el paquete esté construido con R v3.5. Asegúrate de que utilizas la ultima versión de `roxygen2` en CRAN.

Por favor, no incluyas a quienes se encargaron de la edición. ¡Tu participación y contribución a rOpenSci es suficiente agradecimiento!

### 10.2.4 Bienvenida a nuevas personas en rOpenSci

Si das a alguien permisos de escritura en el repositorio

- por favor, ponte en contacto con un miembro del quipo para **invitar a la organización GitHub "ropensci" de rOpenSci** a este nuevo miembro del equipo (en lugar de una colaboración externa)

- ponte en contacto con quien gestiona la comunidad de rOpenSci o con otro miembro del equipo para que el nuevo miembro pueda ser **invitado al espacio de trabajo de Slack de rOpenSci**.

## 10.3 Otros recursos

- Llamada comunitaria de rOpenSci *Set Up Your Package to Foster a Community* (Configura tu paquete para fomentar una comunidad).
- Para reutilizar respuestas amables y habituales, considera las respuestas guardadas.

# 11 Cambio de mantenedores de paquetes

Este capítulo presenta nuestra guía para asumir el mantenimiento de un paquete.

## 11.1 ¿Quieres dejar de mantener tu paquete?

Nuestro newsletter quincenal tiene una sección de pedidos de colaboración denominada *"Call for Contributors"* (*Pedidos de contribuciones*, en inglés). En la misma se destacan paquetes que buscan nuevas personas para encargarse del mantenimiento. Si quieres dejar de mantener tu paquete, contáctanos para que lo incluyamos en esta sección. Hasta ahora hemos tenido mucho éxito, encontrando responsables para todos los paquetes que lo necesitaron.

## 11.2 ¿Quieres encargarte del mantenimiento de un paquete?

Nuestro newsletter quincenal tiene una sección de pedidos de colaboración denominada *Pedidos de contribuciones*. En la misma se destacan paquetes que buscan nuevas personas para encargarse del mantenimiento. Si aún lo hiciste, es una buena idea subscribirte al newsletter para recibir notificaciones cuando hay un paquete buscando ayuda.

## 11.3 Asumir el mantenimiento de un paquete

- Agrégate en el archivo DESCRIPTION, con rol `role = c("aut", "cre")` y cambia el rol de la persona anterior a `role = c("aut")`.
- Asegúrate reemplazar el nombre anterior en cualquier otra parte del paquete por el tuyo, manteniendo a la persona anterior como autora (por ejemplo, en el archivo de manual a nivel de paquete, CONTRIBUTING.md, CITATION, etc…)
- La Guía de Colaboración tiene consejos sobre cómo añadir nuevos miembros al equipo de mantenimiento y colaboración.
- Para paquetes que han sido archivados o han quedado huérfanos en CRAN, no es necesario el consentimiento de quien lo mantenía anteriormente para hacerse responsable del mismo en CRAN. En estos casos, contáctanos para que podamos ofrecerte la ayuda que necesites.

- Si el paquete no ha sido archivado por CRAN y cambia quien lo mantiene, haz que la persona responsable anterior envíe un correo electrónico a CRAN y ponga por escrito quién se hará cargo del mantenimiento de ahí en adelante. Asegúrate de mencionar que se envío ese correo cuando envíes la próxima versión a CRAN. Si quien mantenía al paquete no es localizable o no envía ese correo electrónico, ponte en contacto con el personal de rOpenSci.
- Si puedes contactarte con quien te está transfiriendo el paquete, puedes organizar una reunión para que te explique cómo funciona.

### 11.3.1 Preguntas frecuentes

- Hay algunos *issues* no resueltos del paquete que no sé cómo arreglar. ¿A quién puedo pedir ayuda? Depende; esto es lo que hay que hacer en diferentes situaciones:
- si se puede contactar con la persona anterior, contáctate con ella y pídele ayuda.
- el slack de rOpenSci: es bueno para obtener ayuda sobre problemas específicos o generales, consulta el canal #package-maintenance;
- Foro de discusión de rOpenSci: este foro es una buena opción, siéntete libre de hacer cualquier pregunta allí;
- Equipo de rOpenSci: no dudes en contactarte con cualquier integrante del equipo por correo electrónico o enviando *issues* a GitHub, nos encantará ayudarte;
- por supuesto, también hay ayuda general de R si eso se ajusta a tus necesidades: Foro de la comunidad de RStudio, StackOverflow, Twitter y Mastodon #rstats, etc.
- ¿Cuánto puedes/debes cambiar en el paquete?

Para obtener ayuda general sobre cómo cambiar el código de un paquete, consulta la sección Evolución del paquete.

Al pensar en esto, hay varias cosas a considerar.

¿Cuánto tiempo tienes que dedicar al paquete? Si tienes un tiempo muy limitado, lo mejor será que te centres en las tareas más críticas, sean las que sean para el paquete en cuestión. Si tienes mucho tiempo, tus objetivos pueden ser más ambiciosos.

¿Qué grado de madurez tiene el paquete? Si el paquete es maduro, es probable que mucha gente tenga código que dependa de su API (es decir, las funciones exportadas y sus parámetros). Además, si hay paquetes que dependen de tu paquete en CRAN, entonces tienes que comprobar que cualquier cambio que hagas no rompa esos paquetes. Cuanto más maduro sea el paquete, más cuidado deberás tener a la hora de hacer cambios, especialmente con los nombres de las funciones exportadas, sus parámetros y la estructura exacta de lo que devuelven las funciones exportadas. Es más fácil hacer cambios que sólo afecten a los aspectos internos del paquete.

## 11.4 Tareas para el personal de rOpenSci

Como organización, a rOpenSci le interesa asegurarse de que los paquetes de nuestra suite estén disponibles mientras sean útiles para la comunidad. Si un paquete se queda sin personas que lo mantengan, en la mayoría de los casos intentaremos conseguir nuevas personas que se encarguen del mismo. Para ello, las siguientes tareas son responsabilidad del personal de rOpenSci.

- Ten en cuenta que el hecho que un repositorio no haya ninguna actividad (*commits*, *issues*, *pull requests*) en mucho tiempo puede significar simplemente que se trata de un paquete maduro con poca necesidad de cambios.
- Si no hay respuestas a reportes de problemas o *pull requests* en muchos meses, ya sea por email, *issues* en GitHub o mensajes de Slack:
  - Ponte en contacto para averiguar cuál es la situación. Puede que la persona encargada diga que le gustaría dejar de mantener el paquete, en cuyo caso busca un reemplazo.
- Si no recibes respuesta o no puedes establecer el contacto:
  - Si esto ocurre, insistiremos con el intento de contacto durante un mes como máximo. Sin embargo, si la actualización del paquete es urgente, podemos utilizar nuestro acceso de administrador para realizar los cambios en su nombre.
- Haz un llamado en la sección “Pedidos de contribuciones” del boletín rOpenSci para buscar reemplazo - abre un *issue* en el repo del boletín.

# 12 Releasing a package

Your package should have different versions over time: snapshots of a state of the package that you can release to CRAN for instance. These versions should be properly *numbered*, *released* and *described in a NEWS file*. More details below.

Note that you could streamline the process of updating NEWS and versioning your package by using the `fledge` package.

## 12.1 Versioning

- We strongly recommend that rOpenSci packages use semantic versioning. A detailed explanation is available in the description chapter.

## 12.2 Releasing

- Using `usethis::use_release_issue()` and `devtools::release()` will help you remember about more checks.
- Git tag each release after every submission to CRAN. more info
- CRAN does not like too frequent updates. That said, if you notice a major problem one week after a CRAN release, explain it in cran-comments.md and try releasing a newer version.

## 12.3 News file

A NEWS file describing changes associated with each version makes it easier for users to see what's changing in the package and how it might impact their workflow. You must add one for your package, and make it easy to read.

- It is mandatory to use a `NEWS` or `NEWS.md` file in the root of your package. We recommend using `NEWS.md` to make the file more browsable.
- Please use our example NEWS file as a model. You can find a good NEWS file in the wild in the `taxize` package repo for instance.

- If you use `NEWS`, add it to `.Rbuildignore`, but not if you use `NEWS.md`
- Update the news file before every CRAN release, with a section with the package name, version and date of release, like (as seen in our example NEWS file):

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

- Under that header, put in sections as needed, including: `NEW FEATURES`, `MINOR IMPROVEMENTS`, `BUG FIXES`, `DEPRECATED AND DEFUNCT`, `DOCUMENTATION FIXES` and any special heading grouping a large number of changes. Under each header, list items as needed (as seen in our example NEWS file). For each item give a description of the new feature, improvement, bug fix, or deprecated function/feature. Link to any related GitHub issue like `(#12)`. The `(#12)` will resolve on GitHub in Releases to a link to that issue in the repo.
- After you have added a `git tag` and pushed up to GitHub, add the news items for that tagged version to the Release notes of a release in your GitHub repo with a title like `pkgname v0.1.0`. See GitHub docs about creating a release.
- New CRAN releases will be tweeted about automatically by roknowtifier and written about in our biweekly newsletter but see next chapter about marketing about how to inform more potential users about the release.
- For more guidance about the NEWS file we suggest reading the tidyverse NEWS style guide.

# 13 Marketing your package

We will help you promoting your package but here are some more things to keep in mind.

- If you hear of an use case of your package, please encourage its author to post the link to our discussion forum in the Use Cases category, for a tweet from rOpenSci and possible inclusion in the rOpenSci biweekly newsletter. We also recommend you to add a link to the use case in a "use cases in the wild" section of your README.
- When you release a new version of your package or release it to CRAN for the first time,
  - Make a pull request to R Weekly with a line about the release under the "New Releases" section (or "New Packages" for the first GitHub/CRAN release).
  - Tweet about it using the "#rstats" hashtag and tag rOpenSci! This might help with contributor/user engagement. Example.
  - Consider submitting a short post about the release to rOpenSci tech notes. Contact rOpenSci Community Manager, (e.g. via Slack or info@ropensci.org). Refer to the guidelines about contributing a blog post).
  - Submit your package to lists of packages such as CRAN Task Views, and rOpenSci non-CRAN Task Views.
- If you choose to market your package by giving a talk about it at a meetup or conference (excellent idea!) read this article of Jenny Bryan's and Mara Averick's.

# 14 GitHub Grooming

Los paquetes de rOpenSci se desarrollan actualmente en su gran mayoría en GitHub. Aquí tienes algunos consejos para aprovechar la plataforma en una sección sobre hacer que tu repo sea más descubrible y una sección sobre publicitar tu propia cuenta de GitHub después de pasar por la revisión por pares.

## 14.1 Haz que tu repositorio sea fácil de descubrir

### 14.1.1 Temas de repositorio en GitHub

Los temas del repositorio de GitHub ayudan a navegar y buscar en los repositorios de GitHub, y son usados por `codemetar` en las palabras clave del registro de rOpenSci.

Recomendamos:

- Añadir "r", "r-package" y "rstats" como temas al repositorio de tus paquetes.
- Añadir cualquier otro tema relevante al repositorio de tus paquetes.

Es posible que te hagamos sugerencias una vez que tu paquete esté incorporado.

### 14.1.2 Lingüista de GitHub

El lingüista de GitHub asignará un lenguaje a tu repo en función de los archivos que contenga. Algunos paquetes que contienen mucho código en C++ pueden ser clasificados como paquetes de C++ en lugar de R, lo cual está bien y muestra la necesidad de los temas "r", "r-package" y "rstats".

Recomendamos anular el lingüista de GitHub añadiendo o modificando un archivo .gitattributes a tu repositorio en dos casos:

- Si almacenas archivos html en lugares no estándar (no en docs/, por ejemplo en vignettes/) omite los archivos de documentación. Añade `*.html linguist-documentation=true` a .gitattributes (como en este ejemplo).
- Si tu repo contiene código que no es de tu autoría, por ejemplo, código JavaScript, añade `inst/js/* linguist-vendored` a .gitattributes (como en este ejemplo).

De este modo, la clasificación lingüística y las estadísticas de tu repositorio reflejarán mejor el código fuente que contiene, además de hacerlo más descubrible. En particular, si el lingüista no reconoce correctamente que tu repositorio contiene principalmente código R, tu paquete no aparecerá en los resultados de búsqueda con el filtro `language:R`. Del mismo modo, tu repositorio no podrá aparecer entre los repos de R en tendencia.

Más información sobre las anulaciones del lingüista de GitHub aquí.

## 14.2 Publicita tu propia cuenta

- Como responsable de un paquete incorporado, ahora eres miembro de la organización GitHub "ropensci" de rOpenSci. Por defecto, la pertenencia a la organización es privada; consulta cómo hacerla pública en la documentación de GitHub.
- Incluso después de que el repositorio de tu paquete haya sido transferido a rOpenSci, puedes destacarlo en tu propia cuenta.
- En general, recomendamos añadir al menos una imagen de perfil (¡que no tiene por qué ser tu cara!) y tu nombre a tu perfil de GitHub.

# 15 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.*

## 15.1 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.

## 15.2 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.

## 15.3 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.

## 15.4 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
```

## 15.5 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.

### 15.5.1 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.

## 15.6 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)]
[![Peer-review badge](https://badges.ropensci.org/<issue_number>_status.svg)](https://gith

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

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)](htt
```

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.

# 16 Política de gestión de paquetes

Este capítulo resume la política de gestión propuesta para el mantenimiento continuo de los paquetes desarrollados como parte de las actividades de rOpenSci y/o bajo la organización rOpenSci en GitHub. Esta política de gestión pretende apoyar estos objetivos:

- Garantizar que los paquetes proporcionados por rOpenSci estén actualizados y sean de alta calidad.
- Proporcionar información clara sobre el estado de desarrollo y revisión de cualquier software en los repositorios de rOpenSci.
- Gestionar el esfuerzo de mantenimiento para el equipo de rOpenSci, quienes mantienen paquetes y quienes contribuyen voluntariamente.
- Proporcionar un mecanismo para el retiro gradual de paquetes, manteniendo al mismo tiempo las etiquetas de revisión por pares

Los elementos de infraestructura necesarios para la aplicación de esta política están, en algunos casos, parcialmente construidos y en otros casos aún no se han iniciado. Nuestro objetivo es adoptar esta política en parte para priorizar el trabajo en estos componentes.

- El registro de paquetes rOpenSci registro es una lista centralizada de paquetes de R mantenidos o revisados por rOpenSci. Contiene metadatos esenciales de los paquetes, incluyendo el estado de desarrollo y revisión, y es la fuente de los datos mostrados en sitios web, etiquetas, etc… Esto permite mantener este listado independientemente de la infraestructura o plataforma de alojamiento de los paquetes.

## 16.1 Paquetes mantenidos por el equipo

Los paquetes mantenidos por el equipo son desarrollados y mantenidos por el equipo de rOpenSci como parte de los proyectos de rOpenSci. Estos paquetes pueden ser revisados por pares o no. Muchos son paquetes de infraestructura que están fuera del alcance de la revisión por pares.

- Los paquetes mantenidos por el equipo aparecerán en el registro con la etiqueta “staff_maintained” y aparecerán en la página web de paquetes de rOpenSci o en lugares similares con la etiqueta “staff-maintained”.

- Estos paquetes se almacenarán en la organización GitHub “ropensci”.
- Los paquetes mantenidos por el equipo serán comprobados, y su documentación sera generada por el sistema de rOpenSci. Este sistema no envía notificaciones pero guarda los resultados en forma de estado de *commit* en GitHub (marca de verificación roja o cruz roja).
- Cuando las comprobaciones de un paquete fallen, el equipo de rOpenSci se esforzará en corregir los problemas, priorizando los paquetes en función de la base de las personas que lo usan (descargas), las dependencias inversas o los objetivos estratégicos.
- Cada uno o dos años, rOpenSci revisará todos los paquetes que lleven más de un mes con comprobaciones fallidas para determinar si se transfieren a la organización “ropensci-archive”.
- Los paquetes que fallan consistentemente y que no tienen un plan para volver al mantenimiento activo pasarán al estado de “archivados”. Estos paquetes serán transferidos a la organización “ropensci-archive” y se les asignará la etiqueta “archived” en el registro. No se comprobarán en el sistema de rOpenSci.
- Los paquetes archivados no se mostrarán por defecto en la página web de paquetes. Una pestaña especial de páginas de paquetes los mostrará con `"type": "archived"`, hayan sido revisados por pares o mantenidos por el personal.
- Los paquetes archivados pueden desarchivarse cuando alguien nuevo/a, o quien lo mantenía anteriormente, esté dispuesto/a a solucionar los problemas y quiera revivirlo. Para ello, ponte en contacto con rOpenSci. Se transferirán a la organización “ropenscilabs”.

## 16.2 Paquetes revisados por pares

Los paquetes revisados por pares son aquellos aportados a rOpenSci por la comunidad y han pasado por la revisión por pares. Tienen que estar en el ámbito de aplicación en el momento del envío para ser revisados.

- Una vez aceptados, estos paquetes revisados por pares se transfieren desde el GitHub original a la organización GitHub “ropensci”.
- Los paquetes revisados por pares estarán en el registro etiquetados como “peer-reviewed” y tendrán una etiqueta de revisado por pares en su archivo README.
- Los paquetes revisados por pares aparecerán en la página web de rOpenSci o lugares similares con la etiqueta “peer-reviewed”.
- Los paquetes revisados por pares serán comprobados, y su documentación sera generada por el sistema de rOpenSci. Este sistema no envía notificaciones pero guarda los resultados en forma de estado de *commit* en GitHub (marca de verificación roja o cruz roja).

- Si los paquetes están en CRAN, quienes lo mantienen pueden subscribirse a la API de notificaciones de comprobaciones de CRAN.
- Cada uno o dos años, el equipo de rOpenSci revisará los paquetes que estén fallando o que han estado fallando durante largos períodos y contactará a quienes los mantienen para determinar los planes de mantenimiento y las actualizaciones previstas. En base a esto, rOpenSci puede optar por mantener el estado actual del paquete con la expectativa de actualizaciones, contribuir a los arreglos, ayudar a buscar a una persona nueva para mantenerlo, o transferir el paquete al estado "archived".
- En función de la base de personas que usan el paquete (medida por las descargas), las dependencias inversas u objetivos estratégicos de rOpenSci, el equipo de rOpenSci puede asistir a los paquetes que fallan a través de PRs revisados por quienes mantienen el paquete, o cambios directos (si quienes lo mantienen no responden por aproximadamente un mes). rOpenSci también proporcionará apoyo a pedido, tanto del equipo como de personas voluntarias de la comunidad, en función del tiempo disponible.
- Si quien mantienen un paquete no responde consultas por un mes, o si ésta persona lo pide, rOpenSci puede buscar una nueva persona para mantener determinados paquetes revisados por pares que considere que tienen un alto valor para la comunidad en función de la base de personas que lo usan (descargas), dependencias inversas o los objetivos estratégicos de rOpenSci.
- Cuando se archiven, estos paquetes pasarán de la organización "ropensci" de GitHub a la organización "ropensci-archive" (o a las cuentas de GitHub quienes los mantienen si así lo desean), siguiendo la guía de transferencia. Obtendrán la etiqueta "archived" en el registro. Conservarán las etiquetas e insignias de "peer-reviewed". No se comprobará en el sistema rOpenSci.
- Los paquetes archivados no se mostrarán por defecto en la página web de paquetes. Una pestaña especial de páginas de paquetes los mostrará con `"type": "archived"`, hayan sido revisados por pares o mantenidos por el equipo.

## 16.3 Paquetes heredados

Los paquetes "heredados" son paquetes ni creados ni mantenidos por el equipo de rOpenSci ni revisados por pares, pero que están bajo la organización de rOpenSci GitHub por razones históricas. (Antes de establecer el proceso de revisión por pares y su alcance, rOpenSci absorbía paquetes de la comunidad sin criterios bien definidos).

- rOpenSci transferirá los paquetes heredados de nuevo a las organizaciones y repositorios de quienes los mantienen, si lo desean. Caso contrario, serán transferidos al repositorio "ropensci-archive" siguiendo la guía de transferencia. Si los paquetes están en el ámbito de aplicación, rOpenSci les preguntará si desean someterlos al proceso de revisión del software.

- Los paquetes heredados no se incluirán en el registro de paquetes.
- Se pueden hacer excepciones para los paquetes que son partes vitales del ecosistema de paquetes de R y/o rOpenSci que son supervisados activamente por el equipo.

## 16.4 Paquetes incubados

Los paquetes "incubados" son paquetes en desarrollo creados por el equipo o miembros de la comunidad como parte de proyectos comunitarios, como los creados en las desconferencias

- Los paquetes incubados vivirán en la organización "ropenscilabs".
- Los paquetes incubados aparecerán en el registro de paquetes con la etiqueta "incubator".
- Los paquetes incubados no aparecerán por defecto en el sitio web, pero las páginas de paquetes incluirán una pestaña de "paquetes experimentales".
- Los paquetes incubados serán comprobados, y su documentación sera generada por el sistema de rOpenSci. Este sistema no envía notificaciones pero guarda los resultados en forma de estado de *commit* en GitHub (marca de verificación roja o cruz roja). La documentación indicará claramente que el paquete es experimental.
- Semestral o anualmente, rOpenSci se pondrá en contacto con quienes mantienen los paquetes incubados con repositorios de al menos tres meses de antigüedad para consultar por el estado de desarrollo y las preferencias para la migración a revisión por pares, ropensci-archive, o a sus organizaciones. En base a la respuesta, el paquete se migrará inmediatamente, se iniciará la revisión por pares, o la migración se aplazará hasta la próxima revisión. Los paquetes incubados se migrarán a ropensci-archive por defecto después de un año, siguiendo la guía de transferencia.
- Los paquetes incubados archivados ganarán la etiqueta "archived".

### 16.4.1 Paquetes incubados que no son de R

- La organización "incubator" también incluirá proyectos que no sean paquetes de R.
- Estos proyectos no aparecerán en el registro ni en una página web y no se comprobarán automáticamente.
- La política de migración será la misma que para los paquetes R, con ubicaciones de migración adecuadas (por ejemplo, "ropensci-books")
- Si se archivan, los paquetes que no son de R se trasladarán a "ropensci-archive" siguiendo la guía de transferencia.

## 16.5 Libros

Los libros de rOpenSci son documentación larga, a menudo con formato de libro, relacionada con los paquetes, proyectos o temas de rOpenSci, creados por el personal de rOpenSci o por miembros de la comunidad.

- Los libros vivirán en la organización “ropensci-books
- Los libros se alojarán en books.ropensci.org
- Los libros pueden estar maduros o en desarrollo, pero deben tener un mínimo de contenido antes de migrar a “ropensci-books” (por ejemplo desde “ropenscilabs”).
- La autoría y el estado de desarrollo de un libro deben estar claramente descritos en su página de inicio y en el archivo README.
- rOpenSci puede proporcionar etiquetas o plantillas (por ejemplo, “En desarrollo”, “Mantenido por la comunidad”) para utilizarse en las páginas de inicio de los libros en el futuro

# 17 Guía de Contribución

Este capítulo describe nuestra Guía de Contribución, la cual indica cómo puedes contribuir al proyecto rOpenSci, ya sea con código o no.

¿Quieres contribuir a rOpenSci? ¡Fantástico! Hemos desarrollado la Guía de Contribución a la Comunidad rOpenSci para darte la bienvenida a rOpenSci y animarte a colaborar. Te ayudará a averiguar lo que podrías ganar aportando tu tiempo, tus conocimientos y tu experiencia, a coincidir tus necesidades con las cosas que ayudarán a la misión de rOpenSci y a ponerte en contacto con los recursos que te ayudarán en el camino.

Nuestro equipo y comunidad fomentan activamente un entorno acogedor donde gente de diferentes orígenes y niveles de habilidad aprenden, comparten ideas e innovan juntos abiertamente a través de normas y software compartidos, ya sea que usen o que desarrollen software. La participación en todas las actividades de rOpenSci está respaldada por nuestro Código de Conducta.

Aceptamos contribuciones de código y de otros tipos de personas con o sin experiencia, en cualquier etapa de su carrera, y en cualquier sector. ¡No es necesario que desarrolles software! Tal vez quieras **pasar 30 minutos** compartiendo el caso de uso de tu paquete en nuestro foro público o informando de un error, **una hora** a aprender asistiendo a una Llamada Comunitaria, **cinco horas** revisando un paquete de R enviado para una revisión abierta por pares, **o tal vez quieras comprometerte de forma continuada** para ayudar a mantener un paquete.

¿Cuáles son algunos de los beneficios de contribuir?

- Conectar con una comunidad que comparte tu interés por hacer la ciencia más abierta
- Aprender de personas ajenas a tu ámbito que utilizan R con retos similares a los tuyos
- Plantear y responder a nuevas preguntas de investigación conociendo nuevas herramientas de software y aliados
- Sentir confianza y apoyo en tus esfuerzos por escribir código y desarrollar software
- Ganar visibilidad para tu trabajo de código abierto
- Mejorar el software que utilizas o construyes
- Mejorar tus conocimientos de R y ayudar a otros a mejorar los suyos
- Aumentar tus habilidades de escritura
- Consigue una mayor exposición para tu paquete

Consulta nuestra Guía de Contribución y explora “¿Qué te trae por aquí?” para encontrar qué declaraciones *Quiero …* te representan mejor y elige tu camino.
Para ayudarte a reconocerte, las hemos agrupado en: Descubrir; Conectar; Aprender; Construir; Ayudar. Para cada categoría, enumeramos ejemplos de cómo podrían ser esas contribuciones y enlazamos con nuestros recursos para obtener los detalles que necesitas.

# Parte III

# Appendix

# 18 NEWS

## 18.1 dev (0.9.0)

- 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)

## 18.2 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)

## 18.3 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`.

## 18.4 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)

## 18.5 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)

## 18.6 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)

## 18.7 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)

## 18.8 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).

## 18.9 0.1.5

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

## 18.10 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.

## 18.11 place-holder 0.0.1

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

# 19 Review template

You can save this as an R Markdown file, or delete the YAML and save it as a Markdown file.

## 19.1 Package Review

*Please check off boxes as applicable, and elaborate in comments below. Your review is not limited to these topics, as described in the reviewer guide*

- **Briefly describe any working relationship you have (had) with the package authors.**

☐ As the reviewer I confirm that there are no conflicts of interest for me to review this work (if you are unsure whether you are in conflict, please speak to your editor *before* starting your review).

#### 19.1.0.1 Documentation

The package includes all the following forms of documentation:

☐ **A statement of need:** clearly stating problems the software is designed to solve and its target audience in README
☐ **Installation instructions:** for the development version of package and any non-standard dependencies in README
☐ **Vignette(s):** demonstrating major functionality that runs successfully locally
☐ **Function Documentation:** for all exported functions
☐ **Examples:** (that run successfully locally) for all exported functions
☐ **Community guidelines:** including contribution guidelines in the README or CONTRIBUTING, and DESCRIPTION with `URL`, `BugReports` and `Maintainer` (which may be autogenerated via `Authors@R`).

#### 19.1.0.2 Functionality

☐ **Installation:** Installation succeeds as documented.
☐ **Functionality:** Any functional claims of the software have been confirmed.
☐ **Performance:** Any performance claims of the software have been confirmed.

- ☐ **Automated tests:** Unit tests cover essential functions of the package and a reasonable range of inputs and conditions. All tests pass on the local machine.
- ☐ **Packaging guidelines**: The package conforms to the rOpenSci packaging guidelines.

Estimated hours spent reviewing:

- ☐ Should the author(s) deem it appropriate, I agree to be acknowledged as a package reviewer ("rev" role) in the package DESCRIPTION file.

---

### 19.1.1 Review Comments

# 20 Editor's template

Dear [REVIEWER]

Hi, this is [EDITOR]. [FRIENDLY BANTER]. I'm writing to ask if you would be willing to review a package for rOpenSci. As you probably know, rOpenSci conducts peer review of R packages contributed to our collection in a manner similar to journals.

The package, [PACKAGE] by [AUTHOR(S)], does [FUNCTION]. You can find it on GitHub here: [REPO LINK]. We conduct our open review process via GitHub as well, here: [ONBOARDING ISSUE]

If you accept, note that we ask reviewers to complete reviews in three weeks. (We've found it takes a similar amount of time to review a package as an academic paper.)

Our reviewers guide details what we look for in a package review, and includes links to example reviews. Our standards are detailed in our packaging guide, and we provide a reviewer template for you to use. Please make sure you do not have a conflict of interest preventing you from reviewing this package. If you have questions or feedback, feel free to ask me or post to the rOpenSci forum.

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.

Are you able to review? If you can not, suggestions for alternate reviewers are always helpful. If I don't hear from you within a week, I will assume you are unable to review at this time.

Thank you for your time.

Sincerely,

[EDITOR]

# 21 Reviewer approval comment template

## 21.1 Reviewer Response

#### 21.1.0.1 Final approval (post-review)

- ☐ **The author has responded to my review and made changes to my satisfaction. I recommend approving this package.**

Estimated hours spent reviewing:

# 22 NEWS template

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

### NEW FEATURES

  * New function added `do_things()` to do things (#5)

### MINOR IMPROVEMENTS

  * Improved documentation for `things()` (#4)

### BUG FIXES

  * Fix parsing bug in `stuff()` (#3)

### DEPRECATED AND DEFUNCT

  * `hello_world()` now deprecated and will be removed in a
     future version, use `hello_mars()`

### DOCUMENTATION FIXES

  * Clarified the role of `hello_mars()` vs. `goodbye_mars()`


### (a special: any heading grouping a large number of changes under one thing)

    * blablabla.

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

### NEW FEATURES
```

```
* released to CRAN
```

# 23 Book release guidance

Editors preparing for a release can run the `prelease.R` script in the `inst` directory of this repository to automatically open a GitHub issue with checkpoints for all current issues assigned to the upcoming release milestone, along with the following checklist. Before running the script, please manually check the assignment of issues to the milestone. This should be run one month prior to planned release.

## 23.1 Release book version

### 23.1.1 Repo maintenance between releases

- ☐ Look at the issue tracker for the dev guide and for software review meta for changes still to be made in the dev guide. Assign dev guide issues to milestones corresponding to versions, either the next one or the one after that, e.g. version 0.3.0. Encourage PRs, have them reviewed.

### 23.1.2 1 month prior to release

- ☐ Remind editors to open issues/PRs for items they want to see in the next version.
- ☐ Run the `devguide_prerelease()` function from the `devguider` package.
- ☐ Ask editors for any feedback you need from them before release.
- ☐ For each contribution/change make sure the NEWS in Appendix.Rmd were updated.
- ☐ Plan a date for release in communication with rOpenSci's Community Manager who will give you a date for publishing a blog post / tech note.

### 23.1.3 2 weeks prior to release

- ☐ Draft a blog post / tech note about the release with enough advance for editors and then Community Manager to review it (2 weeks). Example, General blog post instructions, specific instructions for release posts.
- ☐ Make a PR from the dev branch to the master branch, ping editors on GitHub and Slack. Mention the blog post draft in a comment on this PR.

### 23.1.4 Release

- ☐ Check URLs using the `devguide_urls()` function from the {devguider} package
- ☐ Check spelling using the `devguide_spelling()` function from the {devguider} package. Update the WORDLIST as necessary.
- ☐ Squash and merge the PR from dev to master.
- ☐ GitHub release, check Zenodo release.
  - [ ] Re-build (for Zenodo metadata update in the book) or wait for daily build
- ☐ Re-create the dev branch
- ☐ Finish your blog post / tech note PR. Underline the most important aspects to be highlighted in tweets as part of the PR discussion.

# 24 How to set a redirect

## 24.1 Non GitHub pages site (e.g. Netlify)

Replace the content of the current website with a `index.html` and `404.html` files both containing:

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

## 24.2 GitHub pages

You can setup the redirect from your main user gh-pages repository:

- create a new repository (if you don't have one yet): `https://github.com/<username>/<username>.github.i`
- In this repository create a directory `<pkgname>` containing 2 files: a `index.html` and `404.html` file, which both redirect to the new location (see previous subsection).
- Test that `https://<username>.github.io/<pkgname>/index.html` now redirects.