Empirical Software Engineering (2023) 28:106
https://doi.org/10.1007/s10664-023-10345-4

Towards a taxonomy of Roxygen documentation in R packages

Melina Vidoni[1] · Zadia Codabux[2]

Accepted: 25 May 2023 / Published online: 15 July 2023

Abstract
Software documentation is often neglected, impacting maintenance and reuse and leading to technical issues. In particular, when working with scientific software, such issues in the documentation pose a risk to producing reliable scientific results as they may cause improper or incorrect use of the software. R is a popular programming language for scientific software with a prolific package-based ecosystem, where users contribute packages (i.e., libraries). R packages are intended to be reused, and their users rely extensively on the available documentation. Thus, understanding what information developers provide in their packages' documentation (generally, through a system known as Roxygen, based on Javadoc) is essential to contribute to it. This study mined 379 GitHub repositories of R packages and analysed a sample to develop a taxonomy of natural language descriptions used in Roxygen documentation. This was done through hybrid card sorting, which included two experienced R developers. The resulting taxonomy covers parameters, returns, and descriptions, providing a baseline for further studies. Our taxonomy is the first of its kind for R. Based on previous studies in pure object-oriented languages, our taxonomy could be extensible to other dynamically-typed languages used in scientific programming.

Keywords R Programming · Package documentation · Scientific software · Documentation taxonomy

1 Introduction

Currently, software development relies on integrating existing features by writing client code interfacing Application Programming Interfaces (APIs) [48]. To do that, developers

Communicated by: Walid Maalej

✉ Melina Vidoni
melina.vidoni@anu.edu.au

Zadia Codabux
zcodabux@ieee.org

1 Australian National University, CECC School of Computing, Canberra, Australia

2 Department of Computer Science, University of Saskatchewan, Saskatoon, Canada

increasingly rely on proper, complete documentation [12], which aids them during evolution/maintenance activities, reuse of libraries, and external components [1].

Software documentation often has insufficient and inadequate content, obsolete or ambiguous information, and unexplained examples [1;12]. This is concerning since poor documentation affects software maintenance over time, leading to technical issues, limited reusability [26], and inconsistent and incomplete documentation [2]. Nevertheless, the production and maintenance of up-to-date software documentation continue to be neglected [63].

Documentation research has focused mainly on Object-Oriented Programming (OOP) languages producing commercial/traditional software [45;60;10;59]. The difference between 'traditional' and 'scientific' software is not caused by the age or name of the programming language but the purpose of the software itself, including who works on the project (fewer people and more junior developers [44]), the money invested in its maintenance [3], and the development lifecycle [52].

The lack of research in scientific software documentation contributes to the 'gap' expressed by [58]: "the 'chasm' between Software Engineering (SE) and scientific programming is a serious risk to the production of reliable scientific results." This risk happens because scientific software is generally coded in package-based environments requiring interfacing multiple components [29]. Likewise, research-intensive software packages are generally highly specialised and targeted to specific research niches, effectively requiring more in-depth documentation to be appropriately selected and used [33].

The above generates two issues. First, if only the core maintainers understand the package due to poor documentation [26], it will likely be discontinued rather than subsequently updated. Second, producing quality documentation requires understanding its *patterns of knowledge*–namely, what knowledge it contains and how it is organised [39].

Currently, recommendations of patterns of knowledge for scientific software documentation are limited to incomplete, high-level descriptions[1] that provide no grounded guidance on what or how to document software. For example, prior studies demonstrated that poor R package documentation might lead to incorrect usage, affecting all the code depending on it and threatening the validity of experiments and analyses that eventually use these packages [19].

In particular, R originated as a special-purpose language [31] with extensive features for statistical analysis, whose developers rely more heavily on its functional features [28;66]. R is more susceptible to poor documentation than other languages because most R package contributors are less likely to be software engineers by trade [25], and only a few apply sound software engineering practices during development [46;52;67].

Therefore, we chose to work only with R both to keep the scope of this work manageable but also due to three motivators: 1) *R is dynamically typed* (meaning that a variable can take multiple types at different moments without reserved words for types [28;34], forcing developers to 'guess' what types to pass), 2) *R blends OOP and functional programming* (thus rendering current OOP-exclusive taxonomies inapplicable), and 3) *R documentation has been acknowledged as an ongoing issue* [19;74].

For this purpose, we mined 379 R packages from GitHub, including popular and newer packages maintained since 2019. We performed card sorting on three critical elements of the documentation–parameters, returns, and description. We produced an initial taxonomy of what information should be included in the documentation of R packages regarding param-

[1] https://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html

eters, returns, and descriptions. The extracted documentation was mined from source code files, providing 8,670 comments, with over 860,000 lines of package documentation.

As a result, this paper contributes to the need for better documentation standards in R programming [45] by extending prior work in library documentation to the R domain. Our other contributions are:

– This is the first study conducted to explore and understand R packages documentation practices.
– A taxonomy of Roxygen directives for parameters, returns, and description elements. It is structured, including examples (taken from mined GitHub repositories), good practices, and anti-patterns. Our taxonomy is more detailed and complete than Roxygen's own package documentation.
– An analysis of the documentation directives, discussing frequencies, anti-patterns, and comparatives to existing taxonomies. The 'documentation directives' are natural-language statements explaining constraints and guidelines about correctly using a piece of code [45].
– We make available an extensive, well-documented replication package. See Data Availability.

Paper structure. Section 2 presents the related work. Section 3 describes the study's setup, repository selection, dataset generation, and the protocol for the taxonomy generation. Section 4 summarises the taxonomy generated and the relationships between directive kinds. Section 5 discusses the implications, and Section 6 addresses the threats to the validity of this study. Section 7 concludes this study and outlines future works.

2 Related Work

R's Software Ecosystem. R's structure motivated investigations regarding its ecosystem, including the correlation between downloads and package citations to determine impact [34] and its effects on publication activities [78]. Others explored the differences in the growth and expansion of several package-based communities [9], the influence of outdated dependency versioning [49], metrics to quantify the R ecosystem regarding package activity and lifecycle [53], and the maintainability capabilities of CRAN packages [18].

However, R's documentation practices have yet to be approached with the same interest. [57] assessed markdown-generated documentation but was more concerned with R Markdown and other literate-programming formats. [63] focused on documentation quality in R programming but only explored the R language manual, README files, tutorials, articles, and threads in StackOverflow. [76] investigated how the R community creates and curates knowledge in StackOverflow and mailing list, determining that in the former, the participation tends to be individual; in the latter, it builds up on other responses. The types of responses also differ; the mailing list offers suggestions and alternatives, while StackOverflow offers tutorial-like responses.

[67] evaluated self-admitted technical debt in R programming through source code comments, purposefully excluding package documentation. Finally, [19] analysed the technical debt in the peer-review of R packages from rOpenSci and determined that package reviewers give more importance to documentation, being more inclined to manage documentation debt compared to developers.

API Documentation. Several works studied API documentation regarding taxonomies and quality alike [60]. There are large-scale manual explorations of API documentation in Java and

.NET to generate a taxonomy and patterns of knowledge, semantically parsing Javadoc tags [61]. [39] assessed patterns of knowledge in Java and .NET API documentation, assessing the patterns' frequency and co-occurrences while comparing both languages. Also, automatically analysing code assertions with semantic knowledge [10], and proof-of-concept tools to detect outdated documentation [59].

[54] perused Microsoft developers and identified five obstacles to learning API, highlighting that documentation quality is an issue when learning API. A follow-up study with Microsoft developers uncovered five factors to consider when designing API documentation [55]. Later, [65] reported API documentation issues categorised as content or presentation, prioritising the former over the latter.

Existing Taxonomies. [23] defined documentation directives as natural language statements notifying other developers about how to use a software library; however, most are exclusive to Object Oriented (OO) (e.g., related to sub-classes). [45] analysed Java documentation to determine and classify directives, extended prior work, and provided a systematic, organised representation of the taxonomy of Java projects.

Roxygen's official documentation[2], it is *not a taxonomy* but a high-level description of the tag's intended use, written by the developers behind the package `roxygen2`. Therefore, it is not based on systematically-gathered evidence and was considered by Roxygen's own developers as the worst package documentation available [74].

3 Methodology

The mining was completed by following a systematic methodology [68] and taking into account the perils of mining GitHub [32].

3.1 Source Selection

Although CRAN distributes packages, GitHub has risen as a distribution platform for R packages. For this study, we analysed GitHub packages since the perils and problems of mining GitHub are better known than CRAN's, and there are clear strategies to mitigate them [32]. Moreover, GitHub is "increasingly used as a distribution platform for R packages" [22], given that CRAN reserves the right to remove packages without warning[3]. Previous studies demonstrated that about 20% of the most downloaded CRAN packages are on GitHub and that GitHub has a more diverse sample of R packages[21]. The sections below will discuss how the perils of GitHub were mitigated during the process.

We also considered other sources, such as StackOverflow and GitHub issues, but disregarded them, because 1) GitHub issues are exclusive to the repository in which they are open and do not generally discuss documentation unless it caused a development problem, 2) `roxygen2`'s own issues report problems with the package but not with documentation, and 3) the population of StackOverflow posts touching documentation in R programming was too small to be meaningful.

[2] https://cran.r-project.org/web/packages/roxygen2/vignettes/rd.html and https://roxygen2.r-lib.org/index.html

[3] https://cran.r-project.org/web/packages/policies.html

3.2 Repository Selection

The mining process followed the recommendations outlined by [68] and described below.

STEP 1. We defined inclusion and exclusion criteria to determine which packages to consider. The **included R packages** had to be public, open-source repositories written in R and with English as their main language, with a basic structure (as defined by [75]); packages that provided minimal R code but wrapped other languages were also allowed.

Several **exclusion** criteria were defined and are presented in comparison to [32] perils: 1) forked packages (to avoid duplicated samples), 2) packages created before 2010 or with no commits after 2019 (to avoid inactive and low-activity projects), 3) personal, deprecated, archived, or unmaintained packages (to avoid personal projects), and 4) books, data packages, or collections of other packages (to avoid non-software projects).

Finally, the peril *GitHub is continuously evolving* could be related to our use of GitHub's 'best match' algorithm. However, given that we are providing a comprehensive replication package, the names of the packages mined and assessed are publicly available, mitigating this threat and enabling reproducibility.

STEP 2. We used GitHub's advanced search[4] to filter the exclusion criteria through the provided form. We applied the following search string: *package NOT personal NOT archived NOT superseded archived:false created:>2010-01-01 pushed:>2019-01-01 language:R*, searched using GitHub's 'best match' sorting[5], which "combines multiple factors to boost the most relevant item to the top of the result list." However, its algorithm is not publicly available, and its result (just like any other GitHub search) changes as repositories evolve. Nevertheless, as [32] pointed out, all searches in GitHub are prone to change (this is Peril XIII). Since this affects the search replicability, the associated threats to validity are discussed in Section 6. The reproducibility package includes all package names and data used in this study to mitigate the natural variability of the search results.

This search returned 22,308 results, i.e., repositories (as of November 2020). However, regardless of how specific the query terms may be, it is possible that some packages were not properly filtered, producing false positives (i.e., packages that should have been excluded but were shown in the search) or false negatives (i.e., packages that should be included but were left out).

STEP 3. Because we used a manual hybrid card-sorting, it was not feasible to work with the total number of packages returned from the search (namely, $> 22k$) and achieve results within a reasonable timeframe. Therefore, we applied a sample size calculation of 95% confidence and 5% error to determine how many packages should be *effectively mined*; for 22,308 repositories, the sample size was 379.

The search result list produced in STEP 2 was manually inspected by both authors in the order of the results (starting from the first result of the first page) until acquiring the 379 packages that fit the inclusion/exclusion criteria. While doing this, we recorded how many packages were excluded per exclusion criterion (Table 1). **We inspected 432 packages, discarded 53, and kept 379 packages.** Only the 379 packages were effectively mined; this

4 https://github.com/search/advanced

5 https://docs.github.com/en/rest/reference/search

Table 1 Number of repositories excluded by criteria

Exclusion Criterion	# Repos
Collections of Packages	6
Packages as Course Material	3
Book-related packages	2
Data Packages	18
Websites or ShinyApps	1
Tutorial Packages	10
Utilities (e.g., badges, dependencies)	12
Superseded (not filtered)	1
Incorrect Package Structure	0
Forks, mirrors, similar	0
Total	**53**

means that the ‘false positive’ packages were not mined and, therefore, not considered in this study.

The difference between 432 (scouted) packages and the final 379 (selected) packages is due to *false positives*, namely, packages still listed as a search result that did not fit the inclusion criteria. The decision to exclude a package was taken after reading the README file, and perusing the package’s code structure, to assess it against the inclusion/exclusion criteria; a typical example of why the false negatives were listed as search results is that many superseded/personal projects do not use GitHub’s tags, and list the status on the README file. Therefore, if a package was excluded, we had to inspect an additional one. As mentioned before, this process continued until we reached the sample size.

Every searched package was reviewed by both authors individually, and disagreements were discussed until a consensus was reached to mitigate researcher bias. We calculated the inter-rater reliability using Cohen’s Kappa coefficient–a test measuring the raters’ agreement in studies with two or more raters responsible for labelling a categorical scale variable [41]. Cohen’s Kappa results in a number κ between $[-1, +1]$, indicating the highest disagreement and agreement, respectively; nevertheless, the threshold cut-off for deciding on the high agreement varies based on the fields [41]. We considered $\kappa \geq 0.79$ as a high agreement rate, as used in software engineering studies [37]. We obtained a $\kappa = +0.92$, indicating a high agreement rate and reliability for our coding.

Note that this approach is considered standard and systematic, matching current methodologies for MSR selection [68], and does not run into any peril from GitHub [32]. Additionally, the GitHub URLs of the selected R packages were kept in a CSV file alongside those filtered by criteria. It is available in the **replication package** presented in Section 1.

We conducted another analysis to determine the overlap of our selected packages with CRAN, BioConductor, and rOpenSci. Both BioConductor and rOpenSci enforce extensive, well-regarded peer-review processes. In particular, BioConductor is a sub-framework inside the R environment with its automated installation using `BioC` [4]. Table 2 presents the number of selected packages in each package directory. Note that rOpenSci thoroughly peer-reviews its packages but does not require them to be uploaded to CRAN [19]; likewise, it is standard for BioConductor packages not to be available in CRAN due to the intrinsic dependencies with other BioConductor packages. As can be seen, 72.5% of our selected packages were available in CRAN, effectively mitigating validity threats regarding not mining directly from CRAN.

Table 2 Overlap of GitHub packages to other R-related environments

Source	# of Packages	% of Total
BioConductor	31	8.2
rOpenSci	1	0.3
CRAN	275	72.5
GitHub Only	71	19.0

3.3 Data Extraction

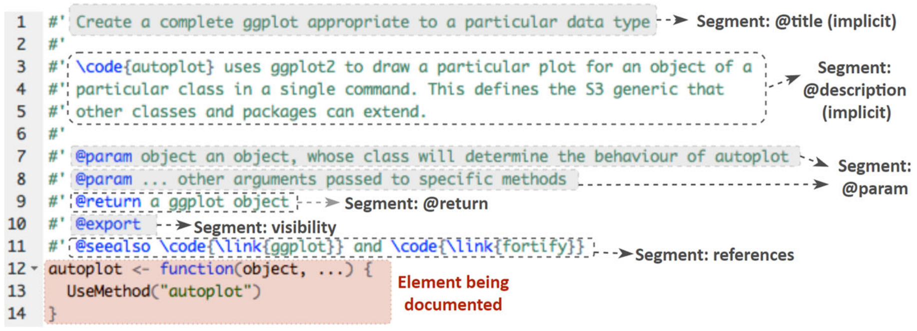
is based on Javadoc and has a similar structure and functionality [75]. As with Javadoc (or similar systems), Roxygen allows detailing specific elements for every function, class, object, or data type existing within a package. To do that, it provides 'tags' (equivalent to Javadoc's `@tag`) to indicate that a 'segment' of the documentation refers to a given attribute of a specific function[6]. However, some can be *implicit* (i.e., recognised by position and not by tag) and are automatically detected when parsing a file to create the online version. It is possible to leave the segment 'blank' (i.e., empty), provide no information, or remove a segment/tag.

The tag structure was used to extract and organise the Roxygen documentation of the mined packages. Using the GitHub repositories' URL obtained from the *repository selection*, we used an R script to download the source code and extracted the lines corresponding to Roxygen. This process consisted of the following steps, completed for each package:

1. Read all R files located in the `/R` folder (equivalent to Java's `src` folder). This allowed us to work only with the source code files and ignore unit testing files since they are located in a different folder, named `/tests` (at the same level as the source code folder) [75].
2. For each file, the script extracted all the lines of the Roxygen documentation using a regular expression. It trimmed starting white spaces and searched for lines beginning with `#'`, the Roxygen comment symbol. The approach was straightforward and uncomplicated, mitigating possible threats caused by the unnecessary complexity of a different tool. This generated a dataset including package name, file name, comment ID number, start and end line, current line number, Roxygen text, and function signature. This produced one large Roxygen block per function, class, data type, or object; henceforth, these four types are called 'elements.'
3. Each extracted comment was divided into *segments* (i.e., consecutive lines that belong to a tag). An auto-incremental 'segment id' field was added to the dataset (in spreadsheet format). Multi-line comments were kept together by reading lines until (a) a blank line or (b) a new tag began. Every row of this dataset had a key composed of the package's name, auto-incremental comment, and segment number.
4. Like in Javadoc, Roxygen tags are identified with an **@** symbol. These were extracted to a new column using regular expressions and purposefully checking against non-Roxygen tags or words (e.g., email addresses); in particular, this check was done by comparing directly to Roxygen's official tag list[7]. Synonym or alias tags were kept together under the main tag only. Figure 1 showcases a Roxygen comment, highlighting the segments it contained. This simplified example is illustrative and does not include all the possible tags.

[6] Roxygen tags are available online: https://roxygen2.r-lib.org/articles/rd.html

[7] https://roxygen2.r-lib.org/reference/index.html



```r
#' Create a complete ggplot appropriate to a particular data type
#'
#' \code{autoplot} uses ggplot2 to draw a particular plot for an object of a
#' particular class in a single command. This defines the S3 generic that
#' other classes and packages can extend.
#'
#' @param object an object, whose class will determine the behaviour of autoplot
#' @param ... other arguments passed to specific methods
#' @return a ggplot object
#' @export
#' @seealso \code{\link{ggplot}} and \code{\link{fortify}}
autoplot <- function(object, ...) {
  UseMethod("autoplot")
}
```

Fig. 1 Example documentation and its segments

Of the 379 selected packages, only 342 had Roxygen documentation. The existence of Roxygen in the package was not part of the exclusion criteria to allow an accurate representation of poorly documented packages. Some 'undocumented' packages had a `man` folder for documentation, but its functions had no comments. Others used regular comments (written as #) instead of Roxygen's (`#'`) to write minimal documentation; these were not structured, limited, and sometimes written under a method's signature. Some used regular comments to clarify code ownership or contribution without information about the function. Only four packages (of the 37 without Roxygen) used the original Latex-inspired documentation style exclusively[8].

Since Roxygen was the primary documentation type, the study centred on its analysis; this is also supported by the literature [75;73]. **The complete dataset has 8,670 Roxygen comments, totalling** 86, 1601 **lines.** The packages had a mean of 38 Roxygen documents, with about 4.3 comments per file describing a function. The dataset was extracted only from the latest commit of the 'master' branch of each repository since it is often used as the main 'release' branch, as per standard R programming books [75;14;13]. This is also addressed as a threat to validity in Section 6.

3.4 Taxonomy Generation

The following subsections present the methodology for generating the taxonomy.

3.4.1 Tag Selection & Study Scope

Roxygen documentation is provided by the R package `roxygen2`[9], produced by RStudio, the most used Integrated Development Environment (IDE) in the R community. Like Javadoc, it has many tags to separate specific parts of the documentation. The tags are classified as *namespace* (to export elements on a package by setting visibility or to import dependencies) and *documentation* (to provide explanations about the elements being documented).

The group of tags considered the minimal 'skeleton' for a Roxygen comment [75] is title and description (which can be implicit, i.e., detected by position rather than tag), parameters, return, visibility, and examples. The description section can have multiple paragraphs organised into individual sections. Likewise, the examples could reference a code page (embedded

8 https://r-pkgs.org/man.html

9 https://roxygen2.r-lib.org

Table 3 Total number of identified segments per tag, and its mean and standard deviation (SD) per package

		Per Package	
Tag	Segments	Mean	SD
@description	2,577	29.61	92.34
@param	79,321	238.92	1,393.08
@return	8,255	26.63	37.14

in the parsed documentation) or be written directly in the Roxygen comment. Given the dataset's size, the large variability most tags present, and the time-consuming manual card-sorting process, we analysed a sample.

As a result, we focused only on three key elements of the *documentation tags*, discussed below; see Table 3 for statistics of counts:

- **`@param paramName`** describes an argument of a function. It can be multi-line but requires the tag and the parameter name. Because R is dynamically-typed, arguments can receive data of multiple types depending on the value of another parameter and have a default value to use if omitted during the invocation. Argument type(s) are not enforced on the signature (the language has no reserved words for types either), are not visible, and may not be internally checked by a function. Finally, it is possible to invoke a function with arguments written in a different order than the specified in the signature by simply writing `paramName = value`. Finally, parameters are essential in functional programming as they allow functions' abstraction and reuse [27].
- **`@return`** describes a function's return and its conditions. Because R is dynamically-typed, functions' signatures do not disclose any type as there are no reserved words for a return and no equivalent Java/C's `void`. A developer can a) use `return(...)` to stop the execution and rebound the value passed there, b) use `invisible(...)` to rebound values that can be assigned but do not print when unassigned[10], or c) return implicitly, by letting the function finish and return whatever was the last in-scope variable assigned. This, plus the fact that returns are essential to functional programming, led us to study this segment type [27].
- **`@description`** is an optional tag for the main explanation of a function. If omitted, the second paragraph of the documentation is considered the description, while the remainder creates the 'details' sections [75]. Therefore, we narrowed the scope to a manageable size by considering only the first paragraph of descriptions explicitly tagged (namely, those with the corresponding tag). Additionally, descriptions can be explicitly formatted using `@section`, items, and markdown notations. Thus, a description section may be lengthy and is not limited to a particular format; this variability adds unnecessary complexity to a manual study. An analysis of how descriptions are documented and formatted is outside the scope of this work.

The remaining tags were excluded from the analysis. *Titles* were disregarded as they are recommended to be a single short sentence [75]. Visibility, aliases, links, families, and related tags were left as future work since they either require extensive triangulation of documents (e.g., links and related) or do not have explanations and only perform actions (e.g., visibility or import tags). Additionally, `@examples` segments were not considered, given the relevance the R community puts on such elements [75;73;16], and how long they can be, they are suitable for a study on its own.

[10] https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/invisible

Additionally, a companion study by [69] focused on analysing common *issues* on documentation, crossing the findings with a developers' survey to determine cases of outdated, incorrect, and incomplete documentation. It reviewed comments distribution and elements generally documented alongside systematic Roxygen tags that do not depend on natural language (i.e., visibility, dependencies management, disclosure of references, author ownership, examples availability, aliases, and keywords). In terms of parameters, it compared the matching between names in the documentation and the code. The current paper performed an analysis that is *complementary* to the one already published and used a *newly mined* dataset. Calculating a statistical sample size yielded the exact sample size; despite being the same number coincidentally (379 repositories), they are different datasets. The main reason to mine new R packages was due to [69]'s dataset being restricted by an Ethical Protocol that limited data available to protect the survey's respondents' identities. The sampling process of the current manuscript was detailed in Section 3.2.

Note that this taxonomy only attempts to be exhaustive but to tackle the most critical natural-language Roxygen tags.

3.4.2 Sampling & Card Sorting

Once we decided which segments to study, the segmented dataset (see Section 3.3) was divided into three parts: D_r containing all the **`@return`** segments, D_p for the **`@param`**, and D_d for the **`@description`** (only those that fit the criteria). However, due to the large datasets, we analysed a representative sample for each segment, using the total number of segments and not the total number of lines. E.g., when comparing two return segments, one can be three lines long, while the other can have 15, but they accounted for two segments and not 18 lines.

Note that topic-modelling algorithms such as LDA are intended to extract topics from a collection of documents [71]; however, their purpose is not to extract or develop taxonomies–namely, a systematic classification and categorisation of content. Likewise, advanced classification techniques (i.e., machine and deep learning, pre-trained models) are not intended to develop taxonomies but to assist in classifying data through *supervised learning*, thus requiring gold datasets of previously manually labelled data [56;24]. As a result, the evaluation of automated classification techniques and their use to expand the taxonomy was left as future work.

Table 4 summarises the original size (i.e., how many segments per type were mined for the 379 packages), the conditions for the sample calculation, and the resulting sample size (i.e., segments to be manually analysed). The **`@description`** have a larger error margin, as per the recommendation of a statistician, since most descriptions were at least twice as long as the other segments.

Table 4 Extracted tagged segments of the original dataset, sample calculation details, and final sample size

Tag	Segments	Sample Calculation	Sample Size
@return	8,255	99% conf., 2% error	2,836
@param	79,321	99% conf., 2% error	3,953
@description	2,577	99% conf., 3% error	1,078
Total			**7,867**

For the manual analysis, we used the *method call directives* proposed by [45] as 'starting directives.' A **directive** is *a natural-language statement that makes developers aware of constraints and guidelines related to the correct and optimal use of an R Package*. A **directive kind** is *a set of directives that share the same sort of constraints or guidelines* [45].

Though some of the 'starting directives' were generic and could be applied to functional programming, many were exclusively OO (e.g., related to object inheritance). Given that R's OO functionalities are limited and not fully embraced by most practitioners [75], these 'starting directives' were excluded. Then, before commencing the sorting process, we organised the 'starting directives' into parameters, returns, errors, and others (i.e., those used in more than one group). This was done in a brainstorming session by both authors, in which each 'starting directive' was discussed and added to one of the groups to be used. Note that a directive may fit into multiple groups.

We applied a *hybrid card-sorting* [72], which is commonly used to derive taxonomies from data and has been previously used in the R domain [19]. Hybrid card-sorting combines working from existing categories (closed card-sorting, in which data is associated with the existing categories) with defining new categories as they emerge during the categorisation (open card-sorting, where the categories are extracted from the data while organising it) [72]. Hybrid card-sorting has been applied to many software engineering studies for taxonomy generation, including a taxonomy of functionality deletion in mobile apps [47], a bug-characterisation taxonomy in cyber-physical systems [77], and a taxonomy of mechanics and gamification parameters [70].

Therefore, we started with two lists of predefined concepts:

(A) The selected 'starting directives'[45]. From "Method Call Directives:" Not Null and Null Allowed, Return Value, String Format, Number Range, Method Parameter Type, Method Parameter Correlation, Exception Rising, Parameter History (later renamed and expanded), and Lifecycle (extended to be on the method and not only the parameters). From "State Directives", Method Call Sequence.

(B) A list of data types derived from R programming books [75;73]: Return Style (none, fixed, variable, normal, invisible), Return Type (primitive, non-primitives, collection, dataframe, object, entry details), Parameter Type (same as Return Type), Extended Restrictions (to consider NAs, defaults), Format Restrictions (extending POINT A's to include Number Format, Date Format, Size/Length). These groups were discussed with two R developers (with 10+ years of experience) during an open brainstorming session, and they suggested including Return Correlation (which extended POINT A's Parameters Correlation) and References (as Roxygen allows adding links to local elements or URLs).

Starting with these sets, the samples of each dataset were explored iteratively to obtain the complete taxonomy. The inter-rater reliability was calculated using Cohen's Kappa coefficient [41], which measures the raters' agreement in studies with two or more raters responsible for measuring a categorical scale variable. The steps were conducted as follows.

PHASE 1. Both authors performed independent manual classifications. Each sampled dataset was perused, and the comments were classified into the directives stated in POINT A. The disagreements were discussed during a peer review session. We calculated the Cohen's Kappa inter-rater for each dataset, averaging $+0.83$, and ensuring a solid classification compared to similar studies [37;30].

PHASE 2. Both authors worked on the agreed partial classification of PHASE 1. Each sampled dataset was reread without being aware of the other author's classification and categorised into the directives stated in POINT B. The disagreements were discussed during a peer-review session; the resulting Cohen's Kappa averaged +0.82, ensuring a sound classification.

PHASE 3. A new discussion emerged between the authors as several 'shared' directives emerged. For example, given that R is dynamically typed, both return and parameters define the data type (i.e., primitive, non-primitive). The only difference was that parameters often added format restrictions (e.g., number ranges). Another case was 'references' (i.e., links pointing to websites or other documented elements) since the same directive appeared in all three datasets. Thus, given that the directives were shared, the authors established unified names and renamed these classifications in all three datasets. This process did not imply reclassification, but simply *renaming* and extracting labels into new columns in the dataset. Thus, there was no need to calculate Cohen's Kappa.

PHASE 4. After the renaming and file restructuring, we noticed that non-predefined directives were repeated in the dataset. Examples were parameters explicitly stating a 'deprecated' status and others mentioning they were mandatory or 'required.' Both authors reviewed each dataset again (individually and independently), marking the segments believed to have an 'emergent directive' (highlighting the sentence or word that prompted the marking). No label was produced at this point. Then, we reviewed each case and: a) removed duplicates per dataset, b) grouped them by similarity (e.g., dataset, words used)[11], and c) read each segment per group to create a new directive.

PHASE 5. We extracted a random sample of 20 example segments for each category and discussed them with the R developers that advised on POINT B. The R developers were aware of the classification, as it was not possible to keep the label hidden while aiming to validate the classification. Since no new suggestions were made, we concluded the taxonomy generation.

To define the good practices and anti-patterns, we followed the definition provided by [45], where **good practices** are "explanation of good practices to achieve clarity and completeness when describing corresponding directives" and **anti-patterns** are defined as ineffective trends that "should be avoided when describing directives in the documentation." To obtain this, we performed three additional phases:

PHASE 6. A script searched for empty tags (e.g., writing just `@param p1` but without adding any explanation), unlinked links (i.e., mentioning a webpage or section of the documentation without providing a working, clickable link), and incomplete citations (i.e., without DOI, or publication details). This automatically determined specific *anti-patterns*.

PHASE 7. A peer-review process was conducted between both authors. Here, we agreed to convert some of the directives already labelled into anti-patterns, such as TYPE>UNDEFINED, RESTRICTIONS>IGNORED. In other situations, the lack of a directive was considered an anti-pattern, such as having a TYPE>NON- PRIMITIVE without explaining the entries (or referencing to a document).

PHASE 8. The remainder of the anti-patterns and good practices were manually explored, per directive. Each author individually and independently selected 2-3 example cases that lacked information (for the anti-patterns) and those considered complete for all the labels involved. This step implied rereading almost all 7800 segments. These were later discussed during a) another peer-review session between the authors and b) a second peer review with the R developers consulted before. None of these two sub-steps produced any changes to the selections. Following the practices of [45], we omitted the calculation of inter-rater

[11] Here, a segment could belong to more than one group, each relation linked to a different 'part' of that comment; this was why we highlighted the parts.

reliability. Some anti-patterns or good-practices occurrences may be subjective; however, the risk of missing an anti-pattern or good practice and leading to unreliable results is negligible.

We chose to work with samples, focusing only on specific documentation elements because the process was extremely long and time-consuming. For instance, PHASE 1 to PHASE 8 required over a year (14 months) to complete. We performed multiple meetings, and in particular PHASE 5 alone required about two months to complete, with another two for PHASE 8. Moreover, coordinating with external developers required additional time. Furthermore, it was not possible to perform automatic classification through machine learning or deep learning, given that no gold data was available for supervised approaches (namely, pre-labelled data to use as a training dataset). Given the text complexity, unsupervised approaches were not recommended [59]. Therefore, this first approach was completed manually, and the labelled datasets were publicly shared in the replication package to enable future automation.

3.4.3 Anti-Patterns Extraction

Automated techniques have been used to extract anti-patterns *from source code*, such as when dealing with code and design smells through static code analysis [8;11]. Prior works also investigated the automated detection of anti-patterns in API functions and variable names [50]. However, those are often called 'linguistic anti-patterns.' They do not refer to *free text*[12], but the wording chosen for class, functions, and variable names and their effects in readability [6].

Since we were working with *free text* instead of code sections or elements' names, detecting anti-patterns required combining knowledge of (i) how R works and what it allows, (ii) understanding the documentation's domain, (iii) comparing that to the positive patterns already found, and (iv) crossing those details to known anti-patterns in traditional OOP documentation. As a result, designing an automated approach to detect anti-patterns was out of the scope of this work.

4 Taxonomy

The taxonomy derived following the methodology explained in Section 3.4 is presented in Fig. 2, which is colour-coded. In the Figure, green shapes represent the segments studied in this manuscript, while the blue ones are the groups of directives (and those shared between multiple segment types are indicated with a blue share icon). Grey shapes represent possible directives (if exclusive, they are indicated with a |, or a red lock if restricted to a segment).

For example, if a developer needs to document a parameter, they should first find the green tag PARAMETERS on the taxonomy of Fig. 2. They can 'navigate' to the required directives using the dashed arrows. If the example parameter is a date, they will need to 'navigate' to FORMAT>PRIMITIVE>DATE FORMAT; if it can be NA, the directive to include would be RESTRICTIONS>NA ALLOWED. Finally, having identified these directives, the developer can use the fully documented taxonomy (in Appendix 1) to learn about each selected directive. We redirect the reader to Section 4.2 for a more in-depth explanation.

[12] *Free text* is information represented in an electronic data storage medium in an ordinary language without any constraint of format.

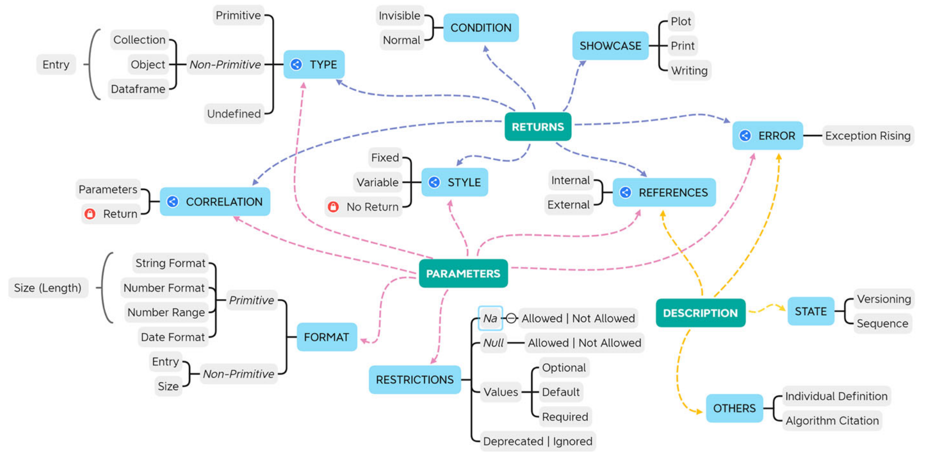


Fig. 2 Taxonomy of Roxygen directives

4.1 Directives Summary

The following subsections summarise the taxonomy's directives. Given the taxonomy's size, we outline *directive kinds*, mentioning each grouped *directive*, and discuss relationships and frequencies. The complete documentation can be found in Appendix 1. Comparisons and discussions will be presented in Section 5.

4.1.1 Shared Directives

These directives were found in multiple segments. In Fig. 2, they have a blue share symbol. For example, given that R is dynamically typed, both returns and parameters must explain their type (e.g., integer, string, list) on the documentation; others are generic and appear across all segments (namely, references and error). Thus, the shared directives are:

Style. These directives are related to the dynamically-typed nature of R programming, which does not provide reserved words for data types, thus allowing variables to hold different types at different times during their lifecycle [73]. As a result, the directives FIXED or VARIABLE make a distinction between parameters or returns that always accept/provide a single type (with the same internal structure in the case of non-primitives) or those under different conditions will accept/provide a different type. Note that NO RETURN refers to functions that provide no return value (i.e., that would be `void` for statically-typed languages such as Java). Therefore, it has a 'lock' symbol because it was only detected in the return segment.

Type. This refers to the type of data being passed; they can be either primitive, non-primitive, or undefined. PRIMITIVES are generally characters, logicals, or numbers (in all its variations), but no subdirectives were created for each type. Meanwhile, NON- PRIMITIVES are COLLECTION (factors, lists, vectors, arrays) a DATAFRAME (matrix, dataframe, tibble, table) or an OBJECT (defined as an R object). Non-primitives can be accompanied by ENTRY, which details the individual values of that non-primitive. Note that the UNDEFINED type is an anti-pattern in itself, as it was used in cases without meaningful information to infer the type being passed (e.g., an ellipsis argument without description or a vague description that highlights no type).

References. This refers to EXTERNAL sources (e.g., websites) not generated by the current documentation to clarify constraints on an element. Otherwise, they can be INTERNAL and generated in other parts of the documentation. For example, when mentioning another object, a shared page, or a section of the same document. In both cases, a non-working link is considered an anti-pattern.

Error. The segment describes cases in which errors are thrown as part of an exception not being handled. It is also valid when describing errors printed on the console or logged in a file.

Correlation. In R, arguments are not enforced and can be omitted when invoking a function. As a result, parameters are often used to change the type of a RETURN. Likewise, they can be used to alter other PARAMETERS by using, enforcing, or ignoring them (related to RESTRICTIONS) or by changing the type of value they accept (related to STYLE and TYPE). Therefore, these were detected only in parameters and returns. Note that the RETURN correlation has a ‘lock’ icon because it was only found in the parameters.

4.1.2 Return-Exclusive Directives

These are found exclusively on the **`@return`** segments. They express constraints and guidelines when documenting a function’s return–the term ‘function’ is preferred as this tag can be used for regular functions or R’s OO methods. They can be:

Condition. These express how a return is rebounded, as discussed in Section 3.4. They can be NORMAL (either when a developer uses use `return(...)` to stop the execution and rebound the value passed there, or the function finishes and returns the last in-scope assigned variable), or INVISIBLE (when using `invisible(...)` to rebound values which can be assigned, but which do not print when not assigned). They can happen on the same return if the return is TYPE>VARIABLE.

Showcase. These are possible, given how R works with its console. It may refer to WRITING (partial output saved as a file at a specific path), PLOT or PRINT (a part of the return is printed or logged on the console or plotted into the inspector).

4.1.3 Description-Exclusive Directives

These were found exclusively on the **`@description`** segments. Although the tag is optional, this part of the taxonomy only covered the segments adequately tagged. Therefore, these directives only cover part of what can be discussed in the Roxygen function description.

State. The first subdirective is SEQUENCE, derived from the work of [45], and specifies the order of method calls (e.g., other functions that should be called before or after the current one). Meanwhile, VERSIONING indicates the lifecycle of a function, such as being experimental, stable, or other custom-made labels. These are not exclusive, and both can appear in a description.

Others. Packages often implement (or apply) algorithms previously developed in scientific papers. The directive ALGORITHM CITATION specifies an algorithm implemented in the function; it can mention the name (for a well-known and established algorithm) or provide a citation. Meanwhile, the INDIVIDUAL DEFINITION clarifies the individual behaviour of every function in a family or group and only applies to shared or grouped documents. These two are not exclusive, and both can appear in a description.

4.1.4 Parameter-Exclusive Directives

These directives appear in the **`@param [name]`** segments. They express constraints and guidelines when documenting a specific argument for a function; the term 'function' is preferred as this tag can be used for regular functions or R's OO methods.

Format. Related to specific details regarding formatting requirements of a parameter. They can be String Format (as derived from the work of [45]) regarding correct string structures, or Date Format when they refer to dates; the latter includes dates passed as strings. Regarding numbers, the taxonomy includes Number Range (when either or both minimal and maximum values are stated) and Number Format (when there is a clarification of formats, such as integer or floating-point, meaning, calculation). Primitives include size to refer to the length (e.g., a string of no more than ten characters, a number with no more than five decimals). Non-Primitives can declare Entry or Size (e.g., a matrix's dimensions). As a result, these directives also share a level (Primitive/Non-Primitive) with Type.

Restrictions. These refer to multiple restrictions enforced on a parameter, either by documentation only or by implementing a particular logic inside a function. Derived from the work of [45], there is Null Allowed or Not Allowed, with the equivalent R-exclusive NA Allowed or Not Allowed, which restricts the usage of `null` and `NA` in a particular argument. As explained in the Correlation section, some arguments can be Optional (in which case they may offer a Default value that would be used if nothing is received) or can be Required if they must be present. A parameter with a correlation can be both simultaneously (i.e., optional under some conditions, mandatory under others). Lastly, some parameters can be Deprecated (no longer used) or Ignored (not implemented yet or irrelevant), which are anti-patterns themselves. However, some deprecated parameters are kept for backward compatibility purposes.

4.2 Directives Relationships

In some cases, we detected that a specific directive 'limited' the usage of entire *directive kinds* or dependent *directives*. These special cases are summarised below. Detailed plots on the relationships are available in the Replication Package (see Section 1.

- In the **`@return`**, if a segment had Style>No Return (meaning that it rebounded no value, like a `void` function), then it did not have a Condition or Type. This is reasonable, given that nothing is returned. However, it may have Showcase, Error, Reference or Correlation. As a result, it was possible for the *return* segments not to have a condition.
- In the **`@return`**, there were cases in which a segment explicitly mentioned that the function always returned `null`. Although this flag could be considered as a case of Style>No Return, we labelled it as Type>Primitive>Null. We considered this a special case of the former, which behaves similarly.
- As seen on Fig. 2, Only Type>Non- Primitive could have a particular type, such as Collection, Object, Dataframe or Entry. Thus, Type>Primitive did not include this division. Likewise, a similar hierarchy happened in Format (with the primitive-exclusive formats), and in Restrictions (with the allowance for `NA` and `null` values).

There were also two cases of **mutually exclusive** *directive kinds*; namely, those whose individual directives cannot be overlapped (i.e., it is either one or the other). This is the case of Style. Although Restrictions>NA>... and Restrictions>Null are not found together, the existence of a Correlation>Parameters may allow them to coexist in the same segment (e.g., given a correlation a parameter cannot be `null`, while in the other cases, it is allowed or default). However, RESTRICTIONS>DEPRECATED OR IGNORED were exclusive, as many developers used the word 'ignored' but meant 'deprecated' (inferred by the remaining words).

Other *directive kinds* were **conditionally shareable**; namely, under specific conditions, they can appear together (e.g., a parameter being both optional and required depending on the values of another). These were: RESTRICTIONS (generally because of a correlation), FORMAT (e.g., a string could not have a number format, but a number could have format and range), TYPE (generally a variable type, sometimes caused by a correlation), and CONDITION (because of a style, a correlation, or on its own due to a function's behaviour).

Moreover, other directive kinds did not require a correlation to be shared, thus being **fully shareable**; these are CORRELATION, OTHERS, STATE, ERROR, REFERENCES, and SHOWCASE.

4.3 Directives Frequency

Depending on the relationships between *directive kinds* mentioned in Section 4.2, we drew insights on what is being documented and how functions work, which are summarised in Table 5. Percentages are always calculated regarding the sample size of the corresponding dataset. A directive not appearing in the dataset does not mean it is infrequently used; however, not having a directive may be an anti-pattern. However, given that the code was not inspected when reading the segments, it was not possible to determine whether this was the case.

Regardless of R's flexibility, about 89.5% of segments returned a fixed type (e.g., always the same type), and only 0.37% provided no return (or always `null`). This was similar to the return condition, with 90% being normal and only 0.8% using invisible returns (either alone or combined). The showcase was minimal, as barely 2.4% of the total returns included a showcase.

For CORRELATIONS, only 152 (about 5.4%) of returns made an explicit mention or a CORRELATION>PARAMETERS, while about 355 parameters did it (about 9%). Parameters also have about 374 records (close to 9.5%) of CORRELATION>RETURNS. Thus, stating a correlation seems more common in parameters than returns. We did not perform a matching study to see how many parameters and returns belonged to the same function.

Regarding TYPE, almost 92.4% of the returns were TYPE>NON- PRIMITIVE, and the order of popularity for types were objects, dataframes, and collections (with 43%, 33%, and 22.8% of the total, respectively). About 49.5% disclosed the ENTRY of their non-primitives (of the total). This trend was considerably different in the parameters sample. About 71.5% were TYPE>PRIMITIVE, and only 4.2% were TYPE>UNDEFINED; for the non-primitives in parameters, the order was collections, objects, and finally, dataframes. Through this, we can confirm that parameters are often used as configurations and entries for the algorithms or analysis performed in a function, hence the different types of returns.

About FORMAT, the most common was FORMAT>STRING FORMAT, appearing in about 10% of the parameters' segments. RESTRICTIONS>DEFAULT and RESTRICTIONS>REQUIRED were the most common constraints (14.7% and 14.2% of parameters segments, respectively). On a positive note, only 0.65% of parameters were stated to be RESTRICTIONS>IGNORED, and only 0.33% were stated as RESTRICTIONS>DEPRECATED. The latter indicates positive practices about updated parameters, albeit it is possible for the documentation to be outdated and not disclose such situations; further explorations regarding version control changes are needed.

Table 5 Frequencies of directives per segment. Percentages are always calculated regarding the sample size of the corresponding dataset, and some may be co-occurring (e.g., a variable parameter can be both primitive and non-primitive), so not every value accounts for 100%; likewise, empty cases are not counted)

Segment	Directive	Frequency %
Description	Others→Algorithm Citation	6.78
	Others→Individual Definition	5.29
	State→Sequence	5.01
	State→Versioning	2.41
	Reference→Internal	19.59
	Reference→External	7.61
Returns	Reference→External	0.21
	Reference→Internal	11.22
	Error→Exception Raising	0.71
	Showcase→Plot	0.63
	Showcase→Print	0.78
	Showcase→Writing	0.39
	Style→Fixed	90.16
	Style→No Return	3.14
	Style→Variable	6.70
	Condition→Normal	90.05
	Condition→Invisible	8.04
	Type→Primitive	4.97
	Type→Non-Primitive	92.06
	Type→Non-Primitive→Dataframe	30.69
	Type→Non-Primitive→Collection	22.96
	Type→Non-Primitive→Object	40.56
	Type→Non-Primitive→Entry	49.52
Parameters	Reference→Internal	13.36
	Reference→External	2.58
	Error→Exception Raising	0.99
	Correlation→Parameters	8.98
	Correlation→Returns	9.46
	Style→Fixed	95.24
	Style→Variable	4.76
	Restrictions→Default	14.67
	Restrictions→Deprecated	0.33
	Restrictions→Ignored	0.63
	Restrictions→Optional	2.66
	Restrictions→Required	14.24
	Restrictions→Null Allowed	3.36
	Restrictions→Null Not Allowed	0.08
	Restrictions→NA Allowed	0.38
	Restrictions→NA Not Allowed	0.10
	Type→Primitive	71.21

Table 5 continued

Segment	Directive	Frequency %
	Type→Non-Primitive	24.39
	Type→Non-Primitive→Dataframe	3.67
	Type→Non-Primitive→Collection	12.17
	Type→Non-Primitive→Object	6.96
	Format→Non-Primitive→Entry	1.92
	Format→(Any)→Size	1.80
	Format→Primitive→String Format	10.47
	Format→Primitive→Number Format	2.48
	Format→Primitive→Number Range	1.54
	Format→Primitive→Date Format	0.46

Overall, 32.9% of returns had REFERENCE>INTERNAL (932 records), with only four records registering an explicit EXTERNAL REFERENCE. Finally, parameters had 13.% of REFERENCE>INTERNAL (but given the sample was larger, this meant 526 records), and 2.6% (102 records) of REFERENCE>EXTERNAL. Descriptions had the least references, with 19.6% and 7.6% respectively; however, of the total, only 6.77% descriptions mentioned an academic citation. Returns had the most internal references, but parameters had the most externals; we hypothesise this may be due to parameters often used as configuration, hence linked to the papers that created the method implemented in a function.

Finally, the usage of ERROR>EXCEPTION RAISING is not common, as it appeared on barely 0.96% of the parameters (38 records), 3.15% of descriptions (exactly 34 records), and 0.7% of returns (20 records).

5 Discussion

5.1 Taxonomy vs. Roxygen Documentation

Roxygen's current documentation[13] is a high-level description of some tags, substantiated with `roxygen2`'s developers' perception ("guidelines") of how Roxygen should be used. For example, in the official Roxygen documentation, only the following information is provided regarding the tag **`@param`**[13].

> @param name description describes the inputs to the function. The description should provide a succinct summary of the parameter type (e.g. a string, a numeric vector) and what the parameter does if it is not evident from the name. The description should start with a capital letter and end with a full stop. It can span multiple lines (or even paragraphs) if necessary. All parameters must be documented. You can document multiple arguments in one place by separating the names with commas (no spaces). For example, to document both x and y, you can say @param x,y Numeric vectors.
> @param tags that appear before the class definition are automatically inherited by all methods if needed.

[13] https://roxygen2.r-lib.org/index.html

As can be seen regarding parameters, there are no suggestions regarding how to document CORRELATION (between parameters or with a return), FORMAT or TYPE (neither for Primitives nor non-Primitives), or RESTRICTIONS. Cross-link documentation (for REFERENCES>INTERNAL) is mentioned in isolation through the **@seealso** tag, which is not necessarily the only option for this [69].

Similarly, only the following information regarding the tag **@return**[13] is available in Roxygen's official documentation:

> @return description describes the output from the function. This is not always necessary, but is a good idea if you return different types of outputs depending on the input, or you're returning an S3, S4 or RC object.

One of the examples in the documentation hints at a correlation between parameters and returns but does not describe how to document *none* of the return- or parameter-related directive kinds presented in this work. Moreover, regardless of R being dynamically typed, Roxygen's official documentation[13] does not mention how to document variable parameters or returns (i.e., the ones whose type changes given a condition)–which our taxonomy explains in detail through the STYLE correlation.

Therefore, our taxonomy provides a more detailed and extensive guidance for package documentation regarding the analysed tags. Given that this work is a companion to a previously published study regarding semi-automated Roxygen tags [69], it is reasonable to assume that the contribution of this paper is relevant for R developers and Roxygen documentation.

Given the above, we provide some suggestions grounded in the taxonomy to extend Roxygen's current documentation regarding *parameters* and *returns*:

- Clarify how the TYPE of each parameter or return must be documented and why this is required. Previous studies indicates that R developers do not consider themselves as developers, e.g., [52], but this perception is no reason to overly simplify the information provided to them.
- Using the taxonomy's dashed arrows to establish the "suggested order" in which to document an element. For example, a parameter's documentation could follow this order: TYPE, FORMAT, STYLE, RESTRICTIONS, CORRELATIONS, REFERENCES; meanwhile, a return could use: STYLE, CONDITION, TYPE, SHOWCASE, CORRELATION, and REFERENCES. The order suggested could be altered by the community. However, the primary purpose is to act as a 'mnemotechnics' to 1) aid developers to remember what to document and 2) establish common ground across R programming.
- Once the order above is presented, Roxygen documentation could include specific examples showing different combinations of directives. This is because only some parameters or returns would need all the directives (e.g., a fixed return may only say it implicitly).
- Although Roxygen provides tags for the references, these are not regulated, and citations of external works (especially academic works) are done in various formats and citation styles. Roxygen documentation could strongly suggest a particular format to establish a common ground between developers.
- Similar to the above, PARAMETERS>RESTRICTIONS should be further elaborated in Roxygen's documentation, providing specific guidelines on: how to document them, why they are needed, suggested practices for deprecation, and ignored parameters/values that are kept in the function signature for backward compatibility.

Beyond these suggested steps, many of the directives uncovered through this work could be used to generate *additional tags* for Roxygen documentation; for example: **@notNull** or **`@notNA`** (to automatically document the corresponding RESTRICTIONS), or *strongly typed hint* styles (akin to Python's PEP-0484[14]).

5.2 Taxonomy vs. Other Taxonomies

The taxonomy generated in this study was derived from the taxonomy established by the 'Method Call Directives' (MCD) from [45], as explained in Section 3.4.2 (referred to as 'starting directives'); moreover, some of our directives have multiple 'patterns of knowledge,' according to the definition provided by [39]. This is summarised in Table 6.

In particular, as explained in Section 3.4.2, we only used the MCD from [45], given that were were not assessing R's OO features or documentation. Nevertheless, our Roxygen Taxonomy is more detailed, both in the granularity of the information provided (e.g., we included both NUMBER FORMAT and NUMBER RANGE as separate directives), but also adds additional groups to 'cluster' directives (e.g., CORRELATION or FORMAT) and links to which each element is allowed to use it. The above presents a two-fold improvement and key differences to [45]'s work.

1. Our Roxygen taxonomy's structure reduces duplication and allows developers to observe the commonalities between different elements in specific package documentation (e.g., Returns and Parameters); an example of duplication in [45] directives are 'Return Value' and 'Parameter Type.'
2. Our Roxygen taxonomy extends some directives to multiple elements. Although this is primarily due to R's dynamically-typed nature, which requires further clarifications of types, multiple directives benefited from this. For example, while [45] only associated 'Correlations' and 'String Formats' to parameters, our exploration demonstrated that, in R packages, they are also related to returns. This helped us create a more extensive taxonomy.

We also compared the content found on the Roxygen taxonomy to the 'patterns of knowledge' [39]. Although two 'patterns of knowledge' were out of scope for this study (namely, 'Code Examples' and 'Patterns'), we found the rest among the Roxygen directives. In particular, some of our directives intersect multiple knowledge patterns. For example:

– CORRELATIONS were considered 'directives' (specifies what the users can/cannot do), 'control-flow' (describes how the package triggers events/behaviours based on the correlation), and 'functionality' (describes the package's function). For example, **`@param 'add' boolean that determines if all items should be added to the travis yaml file or printed on screen`**, is a CORRELATION>RETURN; the following is a correlation between parameters CORRELATION>PARAMETERS: **`@param dpi Input the dpi. If the imageFormat is "pdf," users need not define the dpi. For "png" images, the default dpi is 72. It is suggested that for high-resolution images, select a dpi of 300.`**
– TYPES (namely, primitives or non-primitives) were classified both as 'directives' (because they are a clear contract on what type to pass, regardless of R's dynamic nature) and 'functionality' (because they clarify how the function uses the parameter or generates the return).
– RETURN>STYLE was considered a 'directive' (because it explains what the user can expect to receive/do with a return), but also 'quality attributes and internal aspects,' because

[14] https://peps.python.org/pep-0484/

Table 6 Comparison of the developed Roxygen Taxonomy (central column) to [45] (used as 'starting directives,' as per Section 3.4.2), and the patterns of knowledge by [39]

[45]	Roxygen Taxonomy	[39]
MCD→Not Null/Allowed	Restrictions→NA, Null→Allowed, Not Allowed	Directive, QA& Internal
	Restrictions→Values, Deprecated, Ignored	Directive, QA&Internal
MCD→Return Value	Returns→Type→...	Directive, Functionality
	Returns→Condition→...	Functionality, QA&Internal, Purpose
	Returns→Showcase→...	Functionality
	Returns→Style→...	Directive, QA&Internal
MCD→Post Call		
MCD→Method Visibility		
MCD→Exception Rising	Error→Exception Rising	QA&Internal Aspects
MCD→String Format	Format→Primitive→String Format	Structure, Purpose
	Format→Primitive→Number Format, Date Format	Structure, Purpose
	Format→Non-Primitive→...	Structure, Purpose
MCD→Number Range	Format→Primitive →Number Range	Structure, Purpose
MCD→Method Parameters Type	Parameters→Type→...	Directive, Functionality
MCD→Method Parameters Correlation	Parameters→Correlations	Control-Flow, Directive, Functionality
	Returns→Correlations	Control-Flow, Directive, Functionality
MCD→Miscellaneous	Descriptions→State→...	
	References→...	References, Concepts
	Descriptions→Others→...	References, Concepts
	[Out of Scope]	Code Examples
	[Out of Scope]	Patterns

they often explain the conditions for those returns. For example, for a variable return: **`@return igraph_options returns a list with the old values of the updated parameters, invisibly. Without any arguments, it returns the values of all options. For igraph_opt, the current value is set for option x, or NULL if the option is unset.`**
- Although Table 6 labels REFERENCES and others as a pattern of 'references' and 'concepts,' we still consider them directives because, in many cases, the work that is being referred to (e.g., a citation) will often explain what values to pass/expect, effectively constraining how the developer interacts with the package's functions.

5.3 Implications

For Researchers. This study provides exploratory insights into package documentation in R programming. It paves the way for future studies in Documentation Debt, such as the evolution of directives over time (i.e., outdated documentation), the impact of anti-patterns, and the extension of the taxonomy into other segments, among others. This can be done using the provided directs for close-coding segments of mined documentation to train automatic classifiers, large-scale predictions, and assessments across time (namely, throughout a project's git history). Likewise, this taxonomy enables future human-centric studies, such as those related to challenges and usages [42].

Additionally, our study presents evidence of the differences with documentation made for statically-typed OO languages, supporting further research in this domain. A clear example of this is the need for R developers to make explicit the *type* of a variable or return (by stating it in the text), given that R (as a dynamically-typed language) does not provide typed variables and type-constraints cannot be added in function signatures. Our taxonomy's directives were created for R but could be extrapolated to other languages to enable comparative studies across programming languages or even 'idioms' within R.

Although some intrinsic characteristics presented in this work may have been 'intuitively' known, our results provide systematic evidence of their existence and impact, supporting future investigations by detecting specific patterns. For example, investigating how parameters are correlated and what are the most common correlations.

For R Developers & Data Scientists. The taxonomy generated in this manuscript (and available in Appendix 1) can be used as a guideline for R developers to decide what to include in their documentation, how to avoid anti-patterns, and which practices to uphold. In particular, the "discussion" section of each directive often mentions specific functions or configurations (e.g., `invisible()` for RETURN>CONDITION>INVISIBLE) to assist developers in determining what to document.

As discussed in Section 5.1, our taxonomy is more detailed and extensive than the current guidelines provided by Roxygen[13]. The level of specificity of our taxonomy will provide more guidance to the R developers–especially to those that never crafted software documentation on their own, or never considered if their writing was readable and understandable [69].

Our focus on the 'good practices' and 'anti patterns' of each directive can improve real-world practices, eventually leading to educational trends and practical advice that may contribute to the reduction of documentation debt in R packages [19;69], by making it more explicit (for example, by creating tools that will allow detecting segments and providing hints and suggestions on what to document). Such future works would be possible because our taxonomy establishes relationships between directives, simplifying the decision of *what* to include for each element.

Developers often do not understand how to use a package or “debug" code based on that poorly-documented package and resort to building their own. With proper documentation, such issues can become less common, the usability of R packages beyond the original developers will be increased, and contributing to a repository (instead of creating new software) will be more straightforward.

Additionally, given that organisations for peer-reviewing R packages focus extensively on documentation [19], our results can assist them in establishing a standard of how to document a package and what to look for when reviewing packages prior to acceptance/publication.

For Educators. R is often taught as part of mathematics and data science courses without a solid perspective on code quality or in matters essential to traditional software developers [62;35;20;7]. The taxonomy proposed in this article can be used as a teaching and learning resource to establish the baseline of quality documentation. It is presented in the structured format proposed by [45], with detailed explanations and accompanied by examples. It can also assist educators in directing their curricula, planning classroom activities, and teaching the relevance of proper documentation to avoid accumulating Documentation Debt in R packages.

5.4 Future Works

avenues can be derived from this work, some of which were already mentioned in prior sections.

Taxonomy Extension. Extending the taxonomy to cover other segments (e.g., aliases, titles, sections, visibility, functions families) will be a priority, which can be achieved by reusing our dataset. A cross-source comparison between different documentation sources (such as `pkgdown` websites, developers’ tutorials, and blog posts) remains a future work. Likewise, future works on Roxygen documentation could use an iterative, stratified sampling based on our emerging taxonomy.

The current proposal is not validated; however, it was systematically extracted from real-world data, thus generating knowledge grounded on current, actual R developers’ practices. Although this mitigates some threats, a more thorough assessment will be required before using the taxonomy to ensure its rigour for future work. Once the taxonomy is extended, another avenue will be conducting a developers’ survey to validate our results from a human-centric perspective; nevertheless, given the extensive work completed in this article, such a survey was out of scope for this work.

Human-Centred Analyses. Our taxonomy will enable future mining software repository studies, such as evaluating social aspects (e.g., who completes the documentation and when) and understanding how the directives identified in this work evolve in time (e.g., inspecting commits to uncover which types of changes are done, who does them, and when). Finally, it will allow further comparison with documentation practices in other programming languages, such as Python and Julia, especially given that R shares some general similarities (e.g., scientific approach, dynamically-typed).

Automated Classification. Using both our directives and open dataset, it will be possible to train machine and/or deep learning models as done by [56;24] and develop tool support for comment-assistance, either automatically generating the comments or advising developers of anti-patterns. Although some studies have been conducted in Jupyter Notebooks (which, just like Python, is also dynamically-typed) [43], they require an understanding of common patterns and anti-patterns [38], and manually-labelled datasets which, before our study, were not available for R programming. Machine learning for knowledge identification in API has been previously used [24]. However, they also require gold-standard datasets for the supervised training of the different algorithms; our taxonomy will enable similar works for R. For example, a future avenue of research would be to investigate how directives and anti-patterns are used from a practitioner's perspective to determine their impact and perception.

6 Threats to Validity

Internal Validity. This aspect examines whether the data treatment affected the outcome [79; 5]. The manual study limited how many comments it could explore within a reasonable time frame; thus, we worked with representative sample sizes. To minimise researcher bias, the entire classification was performed by both authors independently and discussed at different stages (Section 3.4). Both authors have extensive experience in technical debt, years of programming experience, and are versed in R. Moreover, we discussed and validated our findings with expert R developers. A validation performed with more raters remains a future work.

Though using GitHub's 'best match' sorting approach is a common standard, it has no clearly defined algorithm. As with any other GitHub search, its use threatens the validity, reproducibility, and generalisability of the data collection and sampling approach, given that the order of packages obtained for this paper may not be precisely reproducible in the future. However, the search results represented the current state-of-the-art when the data was mined, which evolves as software continues to be maintained. To mitigate the presence of false negatives or false positives provided by the 'best match' sorting, we manually analysed each package provided by the search (in the order of the results) to double-check the inclusion/exclusion criteria. The entire process is explained in Section 3.2.

The data was only mined from the latest commit of the 'master' branch of each repository; this decision was made because the 'master' (alternatively known as 'main') branch is suggested as the core 'release' branch in the standard primer R programming books [75;14;13]. This may have led us to analyse incomplete documentation or documentation currently being written. However, we did not produce a completeness/correctness analysis, given that its impact on the quality of the taxonomy is negligible.

External Validity. These threats refer to the generalisability of results. Mining packages from GitHub instead of CRAN enabled future works and ensured that packages under study would be varied enough to depict better what the community offers. It is well-known that not all packages go to CRAN and that significant overlap exists between both [21;22]. Several strategies were used to ensure generalisability: the inclusion and exclusion criteria were defined following accepted standards [32], and a best-match approach was first used to obtain the packages that aligned with the criteria. A representative number of suitable packages was selected from GitHub, and packages were inspected to ensure they fit the criteria. At different points, and to keep the work manageable, random samples of the data were used to generate the taxonomy while maintaining the generalisability of the results (Section 3).

Regarding generalisability to other languages, some directive kinds are applicable to other dynamically-typed languages (e.g., STYLE, TYPE), and others derived from R's own capabilities (e.g., CONDITION, SHOWCASE, RESTRICTIONS). While the former may be applied to other dynamically-typed languages, further evaluation is needed to assess the latter in other contexts. Nevertheless, generalising the results of this work into other languages was out of the scope of this study, and it is considered a future work (Section 5.4).

Likewise, given the extent of the work required for this taxonomy, further validation steps were left as future works; for example, manually checking categories against a new set of mined packages, and/or performing surveys and real-life assessments of the taxonomy. These are discussed in Section 5.4.

To ensure the taxonomy was complete, we worked with representative samples of a large, diverse dataset that included multiple projects of diverse sizes and characteristics. Moreover, we are making public the names of the packages and the labelled dataset to enable further studies in this area (Section 1). Our results are based on real-world data because we mined repositories of existing packages that continue to be worked on. Additionally, the continuous consultation with experienced R developers during the methodology (Section 3.4) contributed to its coherence with real-world practices.

7 Conclusion

This study conducted an MSR of 379 repositories of R packages from GitHub, systematically parsed to extract the Roxygen documentation. We used a hybrid card-sorting to explore generalisable samples of three key segments: parameters, returns, and functions' descriptions to determine which directives (i.e., types of natural language statements) of documentation are used. The paper introduces a taxonomy of directives for R functions (systematically documented in the Replication Package) alongside the coded dataset, which is publicly available. We also provided an analysis of the relationships between directives and frequencies.

Although the proposed taxonomy can be extended, the data provided is a valid and helpful construct: it can support R programmers to identify critical elements to include in their documentation and direct researchers to new opportunities for investigation regarding Documentation Debt in scientific software. This study aims to serve as an empirical foundation for future works.

Acknowledgements The authors gratefully acknowledge the involved developers (O.G. and S.P. who requested their names to be blinded) for the feedback provided on the taxonomy's emerging results. This study was partly supported by the Natural Sciences and Engineering Research Council of Canada, RGPIN-2021-04232 and DGECR-2021-00283 at the University of Saskatchewan.

Funding Open Access funding enabled and organized by CAUL and its Member Institutions

Data availability The Replication Package accompanying this manuscript is available at https://doi.org/10.5281/zenodo.7734769. First, we have included a list of *selected and excluded repositories*, previously described in Section 3.2. This includes the 432 packages scouted and the 53 excluded (Table 1). This is also shared to mitigate the threats to the validity caused by relying on GitHub's 'best match' sorting (discussed in Section 6). The commit used to mine the repositories was always the `master` branch as of November 2020. Second, a spreadsheet with the samples categorised as described in Section 3.4 is also included. Only the resulting categorisations (after finalising PHASE 5) are included here. Third, the taxonomy is presented in Appendix 1. As discussed in Section 4, the taxonomy is not exclusive (some directives belong to multiple categories), and we did not explore tags used for R's OO capabilities (namely, the S3, S4, and R6 systems). Each *directive* is presented in the Replication Package using the same pattern proposed by [45], including name, definition, discussion, examples, good practices, and anti-patterns. This Appendix also includes graphs and explanations of the relationships found and conditions. Fourth, regarding the taxonomy, we are also providing the complete frequency and count calculation that supports the discussion of Section 4.3 and are limited by the relationships discussed in Section 4.2.

Declarations

Conflict of interest The authors declared that they have no conflict of interest.

A. Structured Taxonomy

This taxonomy is presented in Fig. 2 and is colour-coded. In the Figure, green labels represent the segments studied in this manuscript, while the blue ones are the groups of directives (those shared between multiple segment types are indicated with a blue share icon). Grey squares represent possible directives (if they are exclusive, they are indicated with a |, or with a red lock if restricted to which segment uses it).

Based on previous definitions [45], a **directive** is *a natural-language statement that makes developers aware of constraints and guidelines related to the correct and optimal use of an R Package*. In contrast, a **directive kind** is *a set of directives that share the same kind of constraints or guidelines.*

Each of the *directive kinds* is presented using the same pattern proposed by [45]:

– **Name:** A short name to identify the directive kind that summarises what it represents.
– **Definition:** The explanation of the directive. Usually, the first paragraph after the title, with no additional sub-title.
– **Discussion:** The rationale behind the existence of the directive kind, relevant observations, and any concerns related exclusively to R.
– **Example:** Derived from the dataset, that are a prime example for this type. If omitted, they are given in the sections corresponding to good practices or anti-patterns.
– **Good Practices:** Optional. Represents good practices that clarify information about the directive.
– **Anti-Patterns:** Optional. These are trends that are usually ineffective and risk being highly counterproductive when using a directive kind [15].

A.1 Return-Exclusive Directives

These are found exclusively on the `@return` segments. They express constraints and guidelines when documenting a function's return–the term 'function' is preferred as this tag can be used for regular functions or R's OO methods alike. These relationships between directives and *directive kinds* are summarised in Fig. 3, which is colour-coded. Some directives can appear in combinations (highlighted as **+combo**); while other directives determined which could be used in another *directive kind* (e.g., STYLE and CONDITION) and the sequence is indicated with dashed arrows. Some **conditionally shared** appeared in combination with other directives; this is highlighted as **>may be affected by** and a dotted arrow to the related directive.

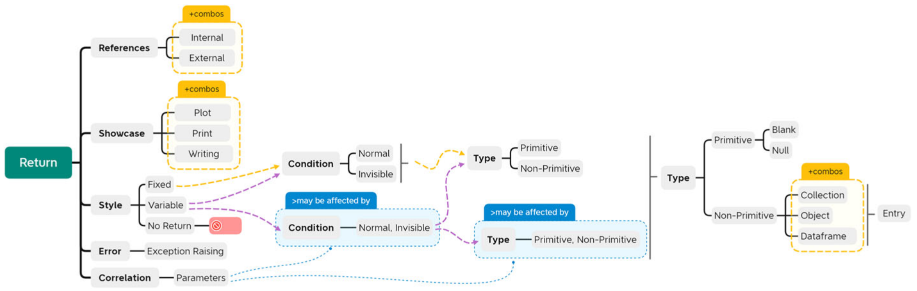


Fig. 3 Return directives relationships and limitations

A.1.1 Condition

These express how a return is being rebounded and are exclusive to the returns.

Invisible
A CONDITION directive. It states that a return is invisible. They are sometimes described as a 'silent return.'

Discussion. The developer must use the `invisible()` expression [75], which is temporarily prevented from being printed out [73]. An invisible return does not stop the execution of a function unless combined with a regular return. As R scripts and markdown reports rely on printing values, making this explicit is critical. It is possible to be combined with STYLE>VARIABLE, to have an invisible return happening only under specific conditions. It can also be related to the parameter CORRELATION>RETURN.

Examples. As a clarification: **`@return the name of the temporary table created (invisibly)`**. As a clear explanation: **`@return invisibly returns the given karyoplot object`**. With fixed returns, such as **`@return TRUE (invisibly)`**.

Anti-Patterns. Not specifying that all returns are invisible or not; e.g.,**`@return invisibly returns the order of rows, if clustfun is provided and/or order=TRUE`**, since there is no information about what is returned in other cases, and if it is invisible.

Normal
A CONDITION directive. It states that a return is not invisible, regardless of how it was returned.

Discussion. It happens when a developer uses use `return(...)` to stop the execution and rebound the value passed there, or lets the function finish and return the last in-scope assigned variable. This is not related to the returned type and can be combined with TYPE to explain that. It is possible to be combined with STYLE>VARIABLE, to have a normal return happening only under specific conditions, or even change the type being rebounded. It can also be related to the parameter CORRELATION>RETURN.

Examples. A clarification on the conditions for a variable return (primitive and non-primitive, but always visible): **`@return A POSIXct object if successful, otherwise failure`**. An example of fixed primitive return: **`@return A string giving the complete mime type, with all parameters stripped off`**.

Anti-Patterns. Leaving the return blank where there are returned values. Not specifying all types of return, or providing incomplete information: **`@return A list of graph attributes, or a single graph attribute`**; another example, but STYLE>FIXED, is **`@return A new graph object`** or **`@return a tibble`**.

A.1.2 Showcase

In R, it is often common to provide alternative returns that are not traditionally rebounded (using the CONDITION directives). These may be complementary or completely replace the above (e.g., a function with STYLE>NO RETURN can still SHOWCASE).

Plot and Print
A SHOWCASE directive. It states that a specific part of the return is written in the console (either printed or logged) or plotted into the inspector.

Discussion. Developers can print or log by using expressions such as `print()`, `message()` or `error()` [75]. Though most plots can be returned as objects (e.g. 'ggplot2'), if not assigned, they are displayed immediately in the inspector [73]. Since many functions rely on this, using this directive is essential to clarify the default behaviour.

Examples. To indicate that something is plotted: **`@return None. Function produces a plot`**. To indicate that the output is printed: **`@return None. Results are printed`**. Combined with directives of the *Style* group: **`@return Prints the PharmacoSet object to the output stream, and returns invisible null`**.

Good Practices. It must be used in combination with STYLE and TYPE directives to clarify any other returns or lack thereof (style) and the type that is being used. If the function has only a showcase in combination with a STYLE>NO RETURN directive, then the *Type* group can be disregarded. Must clarify if the showcase is printed and returned simultaneously.

Anti-patterns. In R, plot objects will be printed if they are called; thus, the directive should be clear regarding automatically plotting and returning the object. For instance, **`@return ggplot object that if called, will print`** is stating a property of a plot object, instead of what the function returns and showcases.

Writing
A SHOWCASE directive, where part (or all) of the output is saved as a file at a specific path.

Discussion. Writing an outcome at a specific path is often helpful to avoid losing a processed data set due to an error on the IDE (Integrated Development Environment).

Examples. Combining writing a file and returning an output (STYLE and TYPE): **`@return Returns a list of metrics derived from the simulated full waveform. A text file (txt) containing the metrics will be saved in the output folder (outRoot)`**.

Good Practices. Explain if any directories are produced or must already exist; e.g., **`@return For type='sparse', a directory is produced at 'path' [...]`**. If the writing path is a default one, it should also be clarified here. If a parameter is the path, the full explanation should be given in the parameter segment. Since this can occur alongside a return, a WRITING must be used with STYLE and TYPE directives. **`@return An invisible data list, and a file is written to the disk if an entry other than the default of NULL is provided for outfile`**, shows that, if writing the file depends on parameters, this information must be given.

Anti-Patterns. Not clarifying what path is used to save the file. Unclear wording about the file being written, returned as an object, or both: **`@return A modified SS .dat file, and that file returned invisibly (for testing) as a vector of character lines`**.

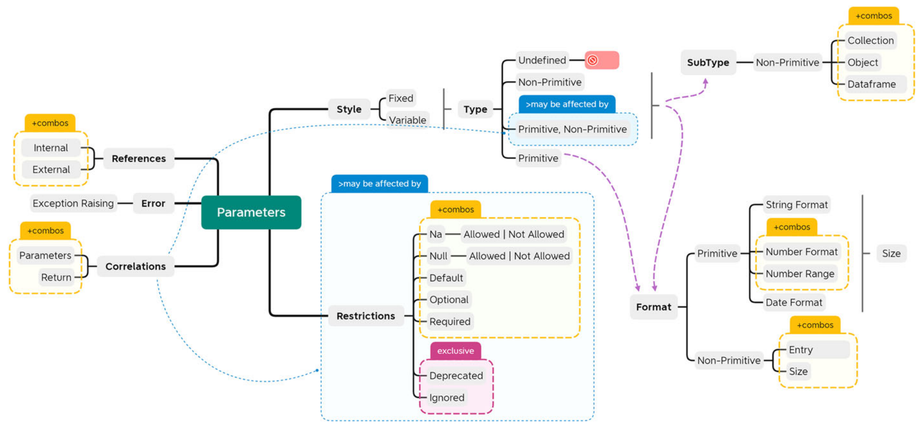


Fig. 4 Parameters directives relationships and limitations

A.2 Parameters-Exclusive Directives

These are directives found on the **`@param [name]`** segments. They express constraints and guidelines when documenting a specific argument for a function; the term 'function' is preferred as this tag can be used for regular functions or R's OO methods. These are summarised in Fig. 4, which is colour-coded. Some directives can appear in combinations (highlighted as **+combo**); while other directives determined which could be used in another *directive kind* (e.g., STYLE and CONDITION) and the sequence is indicated with dashed arrows. Some **conditionally shared** appeared in combination with other directives; this is highlighted as **>may be affected by**, and a dotted arrow to the related directive. Additionally, muatually **exclusive** directives are boxed in.

A.2.1 Restrictions

These refer to multiple restrictions enforced in a parameter, either by documentation or through a function's internal logic. While some of these can be added to the signature of a function (i.e., a DEFAULT value), most are only enforced through internal behaviour that must be clarified for the sake of correct usage.

Null Allowed, Null Not Allowed
A RESTRICTION directive, specifies if a parameter is allowed (or not) to be `null`. It also explains the specific semantics of the `null` value for the respective parameter [45], and any impact it may have on the function's behaviour and return. It is derived from the work of [45].

Discussion. `null` represents a value that does not exist, such as an empty object, indicating an undefined value [75]. This directive states if a `null` value is allowed (and its specific semantics and impact), or if it is not. It combines with TYPE (it can be a TYPE>PRIMITIVE that allows a `null` value, or a TYPE>NON- PRIMITIVE that allows `null` values on its entries). A parameter can allow NA and `null` at the same time.

It is also possible to combine this with other restrictions, such as CORRELATION>PARAMETERS (e.g., can or cannot be `null` when another argument takes specific values or is used), and also with CORRELATION>RETURNS (i.e., a `null` can affect a result). In some cases, it may be related to ERROR>EXCEPTION RAISING as if a `null` value is received but not allowed, the function may throw an error. All these cases should be properly documented and explained, or they become anti-patterns.

Example. Stating that `null` is allowed and its effect, and used in combination with NA RESTRICTIONS directive: **`@param title The title of the plot. null eliminates the title. NA uses the title attribute of the Network object.`**. The NULL ALLOWED can be used in combination with a DEFAULT (e.g., the default value is `null`), such as: **`@param container [...] Defaults to 'NULL'.`** or by parsing its value: **`@param title.cex Character expansion factor for the title. NULL and NA are equivalent to 1.0`**.

Good Practices & Anti-Patterns. Clarify the effect of a `null` argument value: **`@param x ff object where data will be appended to. If x==NULL a new ff object will be created`**. An anti-pattern is allowing `null` without explaining the semantics nor effects: **`@param vp a grid viewport object (or NULL)`**.

NA Allowed, NA Not Allowed

A RESTRICTION directive. It specifies if a method parameter is allowed (or not) to be NA. It also explains the specific semantics of the NA value for the respective parameter and its impact on the function's behaviour or return. This was extended from the NULL ALLOWED, NOT ALLOWED directive from the work of [45], given that it acts in the same way but only with R's unique value NA.

Discussion. In R, NA is a reserved word, with a constant that indicates missing values in any type[15]. It is not the same as a `null` value because NA can be accessed and managed [75]. This directive clarifies if the parameter can receive a NA value or if it is forbidden. If combined with TYPE>NON- PRIMITIVE, it should detail if some elements can be NA and what happens to the whole element; it may also refer to inner NA (e.g., a matrix that accepts missing values). Hence, developers also refer to NA as 'missing' or 'empty values' (according to how it is presented in books and samples).

Like with the NULL ALLOWED, NOT ALLOWED directive, it can be combined with both CORRELATION, with RESTRICTION>DEFAULT or even ERROR>EXCEPTION RAISING. All these cases should be properly documented and explained, or they become anti-patterns.

Example. If the text is clear enough, the reserved word does not need to be included: **`@param Data A numeric matrix or data frame (which may contain missing values)`**.

Good Practices. Clearly stating what happens in the function behaviour when a NA value is passed, such as in :**`@param dates [...] If NA, this means use the lower/upper limit as appropriate`**).

Anti-Patterns. Stating that a NA is allowed, but not explaining the semantics of passing it, nor its effect on the results. When dealing with NON- PRIMITIVE directives, not clarifying if the object as a whole or its elements can be NA.

Default

A RESTRICTION directive, stating that an argument has a default value to be used if nothing is passed there. It explains the semantics of the 'default' and its impact on the function's behaviour and return. It may appear alongside RESTRICTIONS>OPTIONAL or complement it, but that is not always the case.

Discussion. The default value is stated in the signature of a function, as `argumentName = 'defaultValue'`, and means that if the argument is not explicitly passed upon invoca-

15 https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/NA

tion, the functions' behaviour will revert to using whatever was set as the default value [75]. Default values can be of any type, NA or `null` (thus being combined with the corresponding directives). In those cases, the practices and anti-patterns of NA and NULL directive restrictions apply. A default value may not be explicit in the signature but calculated in the function if the argument was empty or null; in this case, it will not appear in the documentation and must be documented.

Example. When specifying a value: **`@param saTemp1 Final temperature for SA (default = 0.01)`**. When the default value is `null`: **`@param grouping.var The grouping variables. Default NULL generates one word list for all text`**.

Good Practices & Anti-Patterns. If the default is internally calculated and cannot be explicitly written in the signature, it should be explained. An anti-pattern is not including the semantics and impact of a default: **`@param legend Plot legend, default = TRUE`**. Detailing the behaviour of the default value will be considered good practice; for example **`@param is_EM Is this an estimation model? Defaults to NULL, which will look for the letters "em" (lower or uppercase) to decide if this is an estimation model or operating model`**.

Optional

This is a RESTRICTION directive that often appears with RESTRICTIONS>DEFAULT but may also appear on its own. It refers to parameters that may be omitted for many reasons.

Discussion. In R arguments are optional, and a function does not need to be invoked with all of them as long as the few used arguments are explicitly named on the invocation [75]. Ideally, when an argument is OPTIONAL, it will also have a DEFAULT value that would be used if nothing is received. However, it is possible to have optional arguments not used if nothing is passed. Moreover, given that there are CORRELATION>PARAMETERS (e.g., using a parameter affects how another argument of the same function is used), some arguments can become optional if another is passed. Likewise, they can become REQUIRED, making it possible for an argument to be labelled as both OPTIONAL and REQUIRED at the same time.

Examples. A case of an optional argument without default that is ignored if not passed: **`@param report Optional name of an HTML file for generating reports`**.

Good Practices. If the argument is optional under some conditions but required in others, the conditions (or related parameters, if existent) must be explicitly stated and clarified. For example: **`@param longitude, latitude optional signed numbers indicating the longitude in degrees East and latitude in degrees North. These values are used if 'type="sippican"', but ignored if 'type="noaa1"', because those files contain location information`**.

Anti-Patterns. If the argument is optional but has a default value, all the anti-patterns of DEFAULT also apply here. If multiple parameters can be passed into a single argument, not providing detailed information about them is considered an anti-pattern; e.g., **`@param Filters Optional parameters that let you set criteria the text must meet to be included in your response`**. Not explaining what happens if the argument is not passed is an anti-pattern, e.g., **`@param Clim Optional limit for conductivity axis`**; in this case, the lack of effect or the argument being ignored should be stated.

When the OPTIONAL is applied to the ellipsis (`...`) argument, providing an REFERENCE>INTERNAL without properly linking it, is an anti-pattern; e.g., **`@param ...`**

`Optional arguments. See Details`. In that case, all anti-patterns of this additional directive also apply here.

Required

This is a RESTRICTION directive that indicates that an argument is mandatory and must always have a value. However, it can be conditionally mandatory and enforced only under some conditions.

Discussion. This is the opposite of RESTRICTIONS>OPTIONAL as this argument must always receive a value. However, it can appear alongside it in situations where there is a CORRELATION>PARAMETERS, as this parameter is mandatory or not depending on what is being passed. In many cases, it was found alongside RESTRICTION>NA|NULL not being allowed, given the function parsed `NA` or `null` as mandatory. Likewise, it can appear with ERROR>EXCEPTION RAISING as the function may halt, throw or print an error if a mandatory parameter is missing.

Example. The mandatory condition should be explicit; e.g., **`@param Identifier [required] The identifier for the resource server.`**

Good Practices. If this is conditionally required, the condition should be explicit: **`@param f A function to use for model fitting. Only required for GLM models at the moment`**. If there are CORRELATION>PARAMETERS that also affect the FORMAT, this should be made explicit: **`@param PrivateKey [required] The private key that matches the public key in the certificate`**.

Anti-Pattern. Stating a requirement without explaining the FORMAT directives: **`@param DomainId [required] The domain ID`**. Not explaining what happens to the functions' invocation when a RESTRICTION>REQUIRED parameter is not passed or erroneous, e.g., **`@param name [required] The name of the filter to create`**. Likewise, if `NA` or `null` are considered as empty values, not explaining such interpretation in the documentation of the parameter is an anti-pattern. Finally, just stating that the argument is required without providing information on what should be passed, such as **`@param GatewayARN [required]`**.

Deprecated or Ignored

These are two RESTRICTION directives that often appear together but are not the same. However, both point to arguments that are no longer used.

Discussion & Examples. DEPRECATED parameters are arguments that were used before but are kept for backwards compatibility or as a warning that they will soon be removed. In other cases, the arguments were replaced by new ones and disclosed in the description. In many cases, we found that deprecated arguments had the description deleted and only kept the status; for example, **`@param method Deprecated`**.

IGNORED arguments are not necessarily deprecated but provide no information about what should be passed. In a few cases, this directive was used to highlight an argument to be implemented in the future while describing what it would use: **`@param parShrinkMN a list for squeezeVar(). (NOT IMPLEMENTED)`**; determining if this is an anti-pattern or not requires more occurrences and deeper analysis.

Good Practices & Anti-Patterns. If DEPRECATED, explaining whether this will be removed in the future or is kept for backwards compatibility should be clarified, e.g., **`@param ... Ignored, included for S3 compatibility`**; otherwise, it becomes an anti-pattern. If replaced by a new argument, the situation should be noted: **`@param document **Deprecated** Use 'devel' instead`**.

Most cases of IGNORED were ellipsis arguments (namely, `...`) explicitly ignored without providing an explanation, such as **`@param ... other arguments (ignored)`**; this is an anti-pattern as it clogs the signature and documentation, and provides no usability.

A.2.2 Format

These directives prescribe formats of particular arguments. As a result, they are exclusive to the parameters.

In general, they can appear alongside TYPE directives (which is why they share the first level). They may be combined with STYLE and have different requirements if they are STYLE>VARIABLE, which should be explicit and detailed not to be an anti-pattern. Overall, they can also be combined with REFERENCES for format guidelines (e.g., redirecting to an external website where the formats are listed, or to a shared document). Finally, they can also be combined with CORRELATION>PARAMETERS (in case the format varies depending on when it is used), and ERROR>EXCEPTION RAISING in case the format is not respected.

Combined anti-patterns. Given the number of combinations, all the good practices and anti-patterns of the associated or related directives will also apply to the FORMAT directives. Additionally, updated or removed formats not reflected in the documentation, incomplete description of the constraint (e.g. specific values without describing their effects), or typos when listing options (thus rendering them incorrect).

String Format
Prescribes the format of a string argument and was derived from the work of [45].

Discussion & Examples. It constraints: lower/upper character, specific values to be accepted, accepted format (example: **`@param URL Character. URL.`**). Specific characters need to be escaped, such as **`@param x Rd string. Backslashes must be double-escaped.`**. Requiring some character types to be included is still part of this directive (e.g., a password with at least one special character). However, length restrictions (i.e., no more or no less than a number) is considered SIZE (LENGTH); e.g., **`@param Description A description of the device. Length Constraints: Maximum length of 256 characters`**.

Number Format
Prescribes the format of a number, including the type.

Discussion & Examples. In R, numbers can be integers, floating-point, precision, calculated, or even part of a factor. This directive constrains the type itself, e.g., **`@param limit A integer. A limit of data in request`**. It can also be used to prescribe specific values to be passed, such as **`@param show Show labels, 1 or 0`**. It may limit how many decimals a number may have, how the value is calculated or obtained, and it may refer to external documents; e.g., **`@param number_format Format for numbering. See [number_format()] for details`**.

Note that length or ranges limitations are part of the FORMAT>NUMBER RANGE directive. Prescriptions of maximum digits can also be interpreted as FORMAT>SIZE(LENGTH), such as: **`@param domainOwner The 12-digit account number of the AWS account that owns the domain`**.

Number Range
Used to delimit ranges in numbers [45], and it is often combined either with FORMAT>NUMBER FORMAT or FORMAT>SIZE(LENGTH).

Discussion & Examples. It can prescribe both sizes of the range, with minimum and maximum values; e.g., `@param level A numerical value between 0 and 1 giving the confidence level`. However, it can also be used to specify only one of the ranges (e.g., by stating it must be positive); for example: `@param GlobalNetworkIds The IDs of one or more global networks. The maximum is 10`. It can also apply to individual elements of a collection of numbers, such as `@param probs numeric vector of probabilities with values in [0,1]`.

Sometimes, the rage was not written as an explicit range but embedded in the explanation of the calculation of the values; for example `@param maxJobDurationInSeconds [required] The maximum simulation job duration in seconds (up to 14 days or 1,209,600 seconds).`

Date Format

R dates have multiple formats, and different functions require unique structures [73]; in many cases, these formats depend on the packages that were used to parse and handle dates. This FORMAT directive states a particular structure for a date. This directive covers date alone, time alone, and date-time formats.

Discussion & Examples. Stating the format by referring to the date object that should be passed, e.g., `@param DeferMaintenanceStartTime A timestamp indicating the start time for the deferred maintenance window`. Strings that only contain dates are considered under the DATE FORMAT directive, and not the STRING FORMAT directive; for example: `@param date Date for which to get schedule (YYYY-MM-DD)`.

Good-Practices. In the case of complex structures, providing an example of a good value is a good practice. For example: `@param endTime The end time of the time period for the returned time series values. This is specified using the ISO 8601 format. For example, 2020-06-01T13:15:02.001Z represents 1 millisecond past June 1, 2020, 1:15:02 PM UTC.`

Anti-Pattern. When referring to objects that represent dates, not providing a reference to a said object is an anti-pattern. Moreover, dates sometimes require timezones to be properly handled; not stating whether a timezone is relevant or not, and if they are converted or manipulated as-is, is also an anti-pattern, as it may provide incorrect results. If incorrect formats are forcefully parsed, this should be stated alongside their effects to disclose them and allow debugging if needed.

Size or Size (Length)

This directive takes different names depending on its association to TYPE>PRIMITIVE (in which it refers to the length, and was explained in the other formats), or TYPE>NON-PRIMITIVE.

Description & Examples. When appearing with TYPE>NON- PRIMITIVE, it refers to limits to collections, bytes, matrix dimensions (without naming the columns), and similar. It can also be combined with FORMAT>NON- PRIMITIVE>ENTRY.

In the case of a TYPE>PRIMITIVE, `@param season: A 4-digit year associated with a given NFL season`. When limiting by size: `@param Text [...] Each string must contain fewer than 20,000 bytes of characters`. When limiting the size of a collection: `@param jobDefinitions A list of up to 100 job definition names or full Amazon Resource Name (ARN) entries`. When stating the size of a matrix: `@param network matrix n1*n2`.

Entry

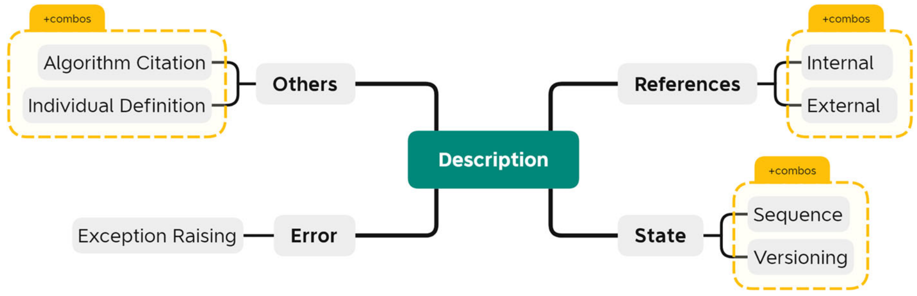


Fig. 5 Description directives relationships and limitations

Similar to TYPE>NON- PRIMITIVE>ENTRY, it refers to the particular format of non-primitive parameters (e.g., vector variable names, dataframe columns, object attributes).

A.3 Description-Exclusive Directives

These were found exclusively on the `@description` segments, and are summarised in Fig. 5, which is colour-coded. Some directives can appear in combinations (highlighted as **+combo**). Although the tag `@description` is optional, this part of the taxonomy only covered those segments properly tagged; as a result, these directives only cover part of what can be discussed in the description of a function when using Roxygen. Note that descriptions can be quite long. As a result, the examples will be excerpts, using a "[...]" to indicate when a text fragment has been extracted.

A.3.1 State

The name for this group of directives was chosen since it encompasses both the version (e.g., a version in a 'stable' state) and the sequence of use (e.g., a function used in the 'preliminary' state of the sequence).

Sequence
It specifies the order of method calls and derives from the work of [45].

Discussion. The sequence does not need to be mandatory, and methods can be optionally related. It can indicate which internal method will be used. Using an INTERNAL REFERENCE directive is implied, and it is affected by those good practices and anti-patterns. It can be used with other directives.

Examples. A case of mandatory sequence, with an unliked internal reference: `@description Workhorse for posterior adaptive grid approximation. Called from cause_grid_adapt`. Indicating an optional sequence: `@description Creates a learning curve object, which can be plotted using the plotLearning-Curve() function`.

Anti-Patterns. Unclear sequence: `@description A helper function prepares a working directory for running an analysis with CAUSE`.

Versioning
It indicates the lifecycle state of a function. It does not require an explanation of changes or bugs but should provide replacement functions.

Discussion. A function may be deprecated or 'experimental.' This directive does not require additional information about the state but contributes to the dependencies' stability (e.g., moving away from deprecated or unstable functions).

Examples. An experimental function may not work properly: `@description \lifecycle {experimental} Shows the variable names that are in common between two or more tibbles.`

Good Practices. Roxygen provides several tags for this, such as `r lifecycle::badge ("stateName")` or the latex-like command `\lifecycle{state}`. Since this is automatically parsed when building the documentation available in the IDEs, its use is encouraged [75]. If a function is deprecated a new one is available, the directive should include a link.

Anti-Patterns. Not mentioning nor linking to alternative functions if deprecated: `@description DEPRECATED`.

A.3.2 Others

These are additional directives that were detected in the analysed description fragments.

Algorithm Citation
It specifies an algorithm implemented in the function. It can mention the name (for a well-known and established algorithm) or provide a citation.

Discussion. Package citations and scientometrics of packages are important [78;36], and given R's scientific use, many packages implement existing algorithms [64;28]; e.g., many academic works compare different implementations in R of the same algorithms [17;40;51]. This directive indicates the algorithm used and how it works. For instance, `@description Calculates the McCune & Keon (2002) Heat Load Index`.

Good Practices. Using an academic citation with linked DOI or referencing to a 'citation' internal documentation page, where all relevant bibliographies are linked.

Anti-Patterns. Using academic citations and not providing a DOI hinders understandability and reproducibility. Mentioning an algorithm by name without providing a link or citation is taxing for developers working on disciplines with many algorithms available. Moreover, finding the specific manuscript may not be possible when papers are cited only by authors' names and publication dates.

Individual Definition
Clarifies individual behaviour of every function in a family or group. It is only applicable to shared or grouped documents.

Discussion. Roxygen allows summarising the documentation of a family or group of functions that have similar behaviour, arguments or return, in order to simplify its readability and browsing [75]. A proper example is the official documentation of the base family of `lapply`[16].

Good Practices. Detailing each function while summarising commonalities. Using formatted syntax is encouraged. For getters and setters, a summarised explanation is acceptable if there is no data manipulation; e.g., `@description 'maxFeatures','maxFeatures <-': getter and setter for the'maxFeatures' slot of the object.`

Anti-Patterns. Many functions are listed, but the individual description replicates other functions without new information. When a group is composed of several functions, only some of them are explained.

[16] https://tinyurl.com/lapplydoc

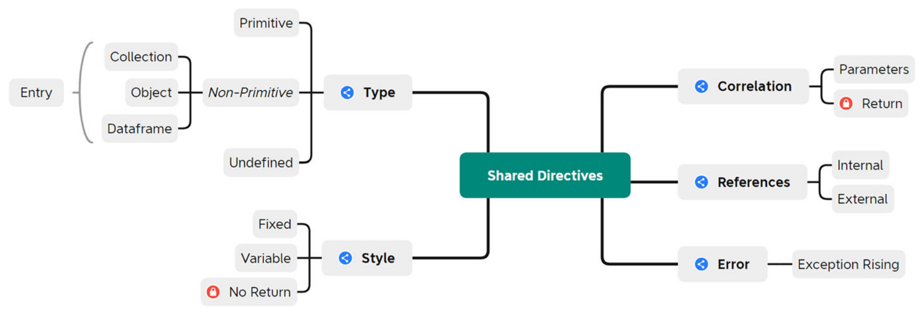


Fig. 6 Shared directives detected in the taxonomy

A.4 Shared Directives

These directives were found in two or three segments and are summarised in Fig. 6. This figure does not explain the relationships between them, as they were mentioned in the Sections above.

A.4.1 References

These directives refer to the cases where there are pointers to additional resources. They can be either INTERNAL or EXTERNAL and were detected among the parameters, returns, and descriptions.

External
A REFERENCE directive points to an external source (not generated by the current documentation) to clarify constraints on an argument. This is a dual directive to INTERNAL reference directive.

Discussion. Roxygen allows linking to other packages, which is essential given the nature of dependencies networks in R, how they affect package growth [78], and the widespread use of functions clones [18]. It can be used to indicate an external source (i.e. using `\url{...}` or markdown syntax `[]()`) that clarifies which arguments can be accepted; this is useful for packages connecting to external APIs.

Examples. To indicate a list of accepted argument values, determined by an API: `@param ... Additional named values that are interpreted as Quandl API parameters. Please see (working URL) for a full list of parameters`. To indicate sources of data, such as `[...] The data set resembles the [chondro](https://...) data set but is entirely synthetic`.

Anti-Patterns. Due to Roxygen's alternative syntaxes available to embed a working link, writing a plain text link is discouraged. Thus, the anti-pattern mentions the source but without a working link. If this is included in the description of a cloned function, making an external reference to the original package is ethical. It is possible for external pages to be modified, no longer providing information accurate to the package; in this case, not updating such links is an anti-pattern.

Internal
A REFERENCE directive, dual to EXTERNAL reference. It can also appear in parameters and descriptions and refer to internal sources inside the same documentation package.

Discussion. It can refer to either another subsection of the documentation of the same function (e.g., redirecting to the examples) or to a general page (e.g., the documentation of a class). A working link can be provided with **`\code{\link{location}}`** [75]. This study found occurrences in the parameters and descriptions segments.

Example. To indicate which function produced the data to be passed as argument: **`@param np Data frame returned by nuts_params()`**. When indicating that internal behaviour uses a specific type of object: **`@description Read simple OTU tables, mapping and taxonomy files into a \code{\link{phyloseq-class}} object`**.

Anti-Patterns. Not providing a working link to an internal reference is an anti-pattern, as it may lead to misunderstandings; example (in parameters): **`@param mSetObj Input the name of the created mSetObj (see InitDataObjects)`**. Likewise, providing a link that does not work is also an anti-pattern.

A.4.2 Error

This has a single directive regarding disclosure of exception handling, and it was detected in parameters, returns, and descriptions.

Exception Raising
Derived from the work of [45], it "states a requirement on the exceptions thrown by a method implementation." It concerns the exception that may be thrown and the situations in which it is thrown. In R, this also refers to errors printed or logged in the console.

Discussion. In R, there are no distinctions between exception types as in other languages [75]. R does have a limited exception handling throw try-catch blocks, but the developer can use `stop()` to terminate a function and display an error, `warning()` to display a problem in the console, or `message()` to show a log on the console [73]; in these cases, the functions receive a message as an argument. This directive should be disclosed when whenever these methods are used.

In many cases, when there are incompatible parameters (RESTRICTION>OPTIONAL parameters with a CORRELATION to another), the developer may document the EXCEPTION RAISING. It can appear in the parameters when used to indicate reattempts before erroring, e.g., **`param n_retries 'numeric' number of times the access to the HTTP server should be retried in case of error before quitting.`**

Anti-Patterns. Not disclosing an exception being thrown when the function does have implicit handling of exception (i.e., omitting this directive when there is an exception being raised). Likewise, stating there is an error but not providing details: **`@return [...] If the sheet does not exist, return an error`**.

A.4.3 Style

These are related to R's dynamically-typed nature and were only found in the parameters and returns.

Fixed
This applies to returns and parameters. It states a required property returned by a specific function, where the type and internal structure are always the same, regardless of the function's behaviour. In a parameter, it explicitly states that an argument only receives variables of a specific type.

Discussion.

– *Return*. Because R is dynamically typed, a function can return different types on each alternative path. A fixed return implies that the return type will always be the same, but the variable's content can change. For example, if it returns a string, it will always be a string; if it returns a matrix of 2x2, it will always be the same matrix. Returning a type or `null` is still considered a FIXED return. This is the most common return, equivalent to a traditional *return value directive* in the work of [45].
– *Parameter*. Because R is dynamically typed, the signature of a function cannot restrict the type of the arguments [75]. Developers often resort to checking type at the start of a function and erroring if it is incorrect [73]. Therefore, this primitive makes explicit that an argument can be of a single type. It should be combined with other *type directives* such as PRIMITIVE or NON- PRIMITIVE (and its variants) to provide details of the expected format (beyond the type).

Example. When a specific value is returned: **`@return true if the request fails (status code 400 or above), otherwise false`**. To add information about a collection's dimensions: **`@return a numeric vector of the same length as x`**. To clarify a specific type of custom object: **`@return a ggmap object (a classed raster object with a bounding box attribute)`**. To specify that the returned object is a modification of an input: **`@return Modified phyloseq object`**. Used without making the type explicit if the description is not ambiguous: **`@param InstanceId (required); The instance IDs for which you want association status information`**. Mixing the explanation with the type: **`@param TP number of true positives`**. Mixed with NON- PRIMITIVES: **`@param object A select qdap object that stores a plot`**.

Good Practices and Anti-patterns. Should be combined with TYPE to explain the exact type of the variable being returned or passed; thus, good practices and anti-patterns of types and showcases also apply here. An R function can SHOWCASE and send a return back alongside this. Providing the type with no additional description is misleading if the variable name is an acronym or shortened name: **`@param pch Numeric`**. When working with numbers, the directive does not clarify the specific type (i.e., number, integer, complex): **`@param eq_price The equilibrium price`**.

Variable
This directive should be used when the type of variable to *return* is conditional: delivering different types (namely, a string and a matrix) after specific conditions. In *parameters*, this directive only appears if the parameter is not of a specific type. It is used to state that the argument can vary in type.

Discussion.

– *Return*. Understanding which return will be provided and under which circumstances is fundamental. Since the function's signature provides no clarification, the documentation should cover it. For a conditional return, at least two different types of variables are returned, i.e., a matrix and a list or a logical and a dataframe. A `null` value will not count as a conditional return. Note that R lists can be composed of any combination of types [75]. This directive only applies to lists if the inner types change, e.g., if in one case it returns a list of data-frames and in another a logical list. A list varying in size is fixed unless a list of size one is returned as the element (extracted from the list), e.g., **`@return An integer, or list of integers`**. As before, it can be used in combination of TYPE and SHOWCASE directives.
– *Parameters*. If the argument is always the same type but changes value, it is a FIXED directive. A VARIABLE directive can change between primitives, non-primitives, or between

them. No maximum limit exists on how many types an argument may accept, but each type's conditions and effects should be clearly stated. Suppose the argument accepts a list or vector of any size; in that case, if the single-element list can be passed as the element itself (without being in the list or vector), it is a VARIABLE directive.

Example. Combined with PRIMITIVES and NON- PRIMITIVES, to explain what each accepted type is: **`@param Rho Required. It can be a single value (correlation common among all variables), a vector of the lower triangular values (vech) of a correlation matrix, or a symmetric matrix of correlation coefficients`**. To clarify between two NON- PRIMITIVES: **`@param a numeric vector or matrix`**.

Good Practices. If an argument determines the change, the largest explanation should be in the **`@param`** segment, with a small reference in the return; e.g., **`@return A tibble (or dataframe), or ggplot2 object if plot = TRUE`**. If many types are NON-PRIMITIVES, it needs an explanation of their composition or a reference to the document that explains it. If the type of return changes according to the results of an internal calculation, it should be explained.

Anti-patterns. Not explaining the conditions upon which a specific type is returned: **`@return A numerical vector or a time series object of class ts`**. This is challenging since developers cannot know when or why a type of return is produced.

Not mentioning all the valid types for a *parameter*. This is acceptable in the case of the ellipsis argument, used when a function can have a variable number of arguments [73]; e.g., the argument was not an ellipsis: **`@param obj A vector, matrix etc.`**; the use of 'etc.' when listing types is strongly discouraged. As this type should be combined with PRIMITIVES and NON- PRIMITIVES, it is affected by their good practices and anti-patterns.

No Return
R functions do not have to declare a return type, so this directive clarifies that a function produces no return (neither regular nor invisible). It is equivalent to Java's `void` return. Thus, it only applies to the **`@return`** segment; note that it was placed here since it could be considered a specialisation (or particular case) of STYLE>FIXED when applied to results.

Discussion. Generally, it is not required to write the `return()` expression at the end of an R function–R returns the value of the last expression evaluated [75]. This can cause a problem if said value is a flag, an intermediate or inconsistent state, an internal value, or even unusable flags such as `NA` or `null`. This directive is used to notify that no intended return is provided and ignore it if any is automatically returned.

Example. A simple explanation with no additional comments, such as **`@return None`**. Or in combination with showcase: **`@return None. Results are printed`**.

Good Practices. If the function produces a showcase, that additional information should be clarified. Good practices and anti-patterns related to showcases are explained in the corresponding directive and are applicable here.

A.4.4 Type

Like the STYLE, these are needed because R is dynamically-typed and has no reserved words to enforce types in variables. It was found only on parameters and returns.

Primitive
It states that the *argument* receives a primitive, or the return rebounds a primitive value. It may have several TYPE>PRIMITIVE directive kinds if it is of STYLE>VARIABLE and allows for

a different primitive. It explains what the value(s) means and how it impacts the behaviour. Collections of primitive types (i.e., vectors or lists) are non-primitives. Always returning `null` can be considered a primitive return.

Discussion. The name 'primitive type' is not used in R; instead, they are defined as 'basic' [75]; this name was chosen for readability purposes. A primitive is either a character, logic, or a number (i.e., integer, numeric, or complex). The explanation must mention the accepted type. In some cases, this is not required (i.e. when describing a logical). It can be used with STYLE, and alongside other TYPE directives (e.g. collection) if the return or argument is STYLE>VARIABLE. In return, it can be used alongside SHOWCASE directives. In the parameters, it can be accompanied by a FORMAT directive.

Good Practices. Explaining the meaning; for example, **`@param ntrees the number of trees in the population`**. Specify if some values modify a function's behaviour: **`@param num.iter Integer scalar specifying the number of iterations to use for the grid search`**.

In the returns, it clarifies the conditions for each value of a logical: **`@return Logical indicating whether a write occurred, invisibly`**. If a character or number is returned and its structure or ranges can change according to the function's internal process, the directive should provide this information. No type requires an explicit mention of the type (e.g. 'returns a logical') if it can be safely assumed from the explanation; for instance**`@return Returns (invisibly) the old root path`**.

Anti-patterns. No description at all (e.g. **`@param maxResults`**), or a vague description that does not clarify the usage (e.g. **`@param my.id Company's ID`**). When a parameter has unexplained format restrictions. Not clarifying the conditions for each value when a logical is returned: **`@return TRUE or FALSE`**. Unclear wording that does not convey the type variable being returned: **`@return 'bib' - invisibly`**.

Non-Primitive
It indicates that a parameter or return accepts/rebounds a non-primitive type as a value. It explains what the argument's value(s) means and how it impacts the function's behaviour.

Discussion. A non-primitive is considered to be a COLLECTION (factors, lists, vectors, arrays) a DATAFRAME (matrix, dataframe, tibble, table) or an OBJECT (defined as an R object). In all of those cases, it can be accompanied by ENTRY, which details the individual values of that non-primitive. A null or NA return is a NON- PRIMITIVE TYPE directive if it happens variably. It can be combined with STYLE directives, or SHOWCASE directives (the latter for returns only). In the parameters, it can be accompanied by FORMAT directives such as FORMAT>ENTRY or FORMAT>SIZE.

Good Practices. Primitive values included inside the non-primitive should include the PRIMITIVE extension when specific formats are needed for a parameter. Linking to type documentation is accepted because it avoids redundancy, but any deviation from the generic document should be explained.

ENTRIES should be explained. This implies a structure of the columns (for frames), a vector's values, a list's structure, and object type. Roxygen allows creating documentation for an object; linking to said document (if it exists) instead of repeating the explanation is a good practice. For example **`@return A data frame with three columns: httr (The short name used in httr), libcurl (The full name used by libcurl), and type (The type of R object that the option accepts)`**. In cases like this, since Roxygen allows markdown commands, its use is encouraged.

Additional information such as encoding (if characters) or length: **`@return For 'text', a character vector of length 1. The character vector is always`**

`re-encoded to UTF-8. If this encoding fails (usually because the page declares an incorrect encoding), 'content()' will return 'NA'`. If it always returns an object, but the class changes conditionally, it should be a VARIABLE directive kind (whichever corresponds), explaining the conditions in which is class is returned.

Anti-Patterns. Not providing insight into the composition of the TYPE>NON- PRIMITIVE can hinder the developers as they struggle to determine what is acceptable. Using the words 'list' or 'vector' as interchangeable can lead to misunderstandings due to differences in manipulating both types. Generic descriptions that do not clarify the type, content, or use act as an anti-pattern: **`@param object object`**.

Unclear wording that does not clarify the non-primitive entries without providing links; e.g., **`@return A list of OAuth parameters`**. Listing the entries without additional explanation about types: **`@return a list containing: scheme, hostname, port, path, params, fragment, query (a list), username, password`**. Lack of clarification regarding several objects being returned as a collection or as conditional returns. If a function returns a 'tibble' [75]. However, the directive describes it as a 'dataframe,' the documentation is misleading since they are technically different types of data, and some functions or packages work with one but not the other; not using the proper word for a type is an anti-pattern.

Undefined

This is an **anti-pattern** in itself, found only on the *parameters*. It was commonly detected on the ellipsis argument (namely, `...`) when the argument was written but not used. For example, **`@param ... Additional arguments, currently ignored`**. Sometimes it was used generically on grouped elements, but without providing enough details to infer the types, such as **`@param ... arguments passed to other methods`**.

In other cases, they specified partial information, as in the following case: **`@param ... Options to set, with the form name = value`**. Albeit it is feasible to infer multiple arguments are passed, this is not a collection, and there are no details regarding what the names are, nor what values (types, formats) are acceptable.

A.4.5 Correlation

These were expanded from the work of [45]. In R, arguments are not enforced, and they can be omitted by default. As a result, often parameters are used either to change the type of a RETURN. Likewise, they can be used to alter other PARAMETERS by using, enforcing, or ignoring them (related to RESTRICTIONS) or by changing the type of value they accept (related to STYLE and TYPE).

Parameters

A CORRELATION directive, describing inter-dependencies involving two or more arguments [45]. The arguments must belong to the same function. This was found on both parameters and returns.

Discussion. R does not provide any syntax for cases when values are only accepted after given conditions in another argument. This directive indicates a correlation between the parameters of the same function. Suppose the parameter is of VARIABLE, and some types can only be passed when another parameter meets a specific condition, then a PARAMETER correlation directive should be used. For a return, it is often used to indicate all types of returns and which parameters affect them.

Example. An argument can only be used if another is used: **`@param longitude The longitude of observation (only used if eos="gsw"; see Details)`**. One argument's characteristics are linked to another's **`@param weights Numeric weights vector. Must be the same length as x`**. Passing one parameter makes another compulsory: **`@param scale A logical value: whether to return standard deviations or 1s. Don't use scale without using centre`**. In a return, depending on an optional parameter: **`@return The value of the edge attribute, or the list of all edge attributes if name is missing`**.

Anti-Patterns. Not explaining the correlations, stating incomplete correlations, or not updating the correlations upon changing them. If combined with other directives, such as RESTRICTIONS, other CORRELATIONS and ERRORS, all the anti-patterns of those directives also apply here.

Return

A CORRELATION directive, it is a dual for PARAMETER. It describes inter-dependencies between the parameter of a method and the return it will provide. This was found only on the parameters.

Discussion. Many arguments are used to configure a function's behaviour, affecting its results. This correlation cannot be written in the method signature and must be clarified through a directive. The return of a different function may be forced as the parameter of another; in that case, the description should also have a STATE>SEQUENCE directive.

Example. To affect how many (and which) results will be returned: **`@param nextToken The token for the next set of items to return. (You received this token from a previous call)`**. To affect showcase returns (i.e. printing or plotting): **`@param silent keep output as silent as possible. Defaults to true`**.

Anti-Patterns. Stating incomplete correlations or not updating them after a change. Not mentioning the correlation on the return segment of the documentation. If combined with other directives, such as RESTRICTIONS, other CORRELATIONS and ERRORS, all the anti-patterns of those directives also apply here.

References

1. Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software Documentation Issues Unveiled. In: IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, Montreal, Canada, pp 1199–1210, 10.1109/ICSE.2019.00122
2. Aghajani E, Nagy C, Linares-Vásquez M, Moreno L, Bavota G, Lanza M, Shepherd DC (2020) Software documentation: The practitioners' perspective. 42nd International Conference on Software Engineering (ICSE). IEEE/ACM, South Korea, pp 590–601
3. Ahalt S, Band L, Christopherson L, Idaszak R, Lenhardt C, Minsker B, Palmer M, Shelley M, Tiemann M, Zimmerman A (2014) Water Science Software Institute: Agile and Open Source Scientific Software Development. Computing in Science Engineering 16(3):18–26. https://doi.org/10.1109/MCSE.2014.5
4. Amezquita RA, Lun ATL, Becht E, Carey VJ, Carpp LN, Geistlinger L, Marini F, Rue-Albrecht K, Risso D, Soneson C, Waldron L, Pagès H, Smith ML, Huber W, Morgan M, Gottardo R, Hicks SC (2020) Orchestrating single-cell analysis with bioconductor. Nature Methods 17(2):137–145. https://doi.org/10.1038/s41592-019-0654-x
5. Ampatzoglou A, Bibi S, Avgeriou P, Verbeek M, Chatzigeorgiou A (2019) Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. Information and Software

Technology 106:201–230. https://doi.org/10.1016/j.infsof.2018.10.006, https://www.sciencedirect.com/science/article/pii/S0950584918302106

6. Arnaoudova V, Di Penta M, Antoniol G, Guéhéneuc YG (2013) A new family of software anti-patterns: Linguistic anti-patterns. In: 17th European Conference on Software Maintenance and Reengineering, pp 187–196, 10.1109/CSMR.2013.28
7. Auker LA, Barthelmess EL (2020) Teaching r in the undergraduate ecology classroom: approaches, lessons learned, and recommendations. Ecosphere 11(4):e03060. https://doi.org/10.1002/ecs2.3060
8. Barbez A, Khomh F, Guéhéneuc YG (2020) A machine-learning based ensemble method for anti-patterns detection. Journal of Systems and Software 161:110486. https://doi.org/10.1016/j.jss.2019.110486, https://www.sciencedirect.com/science/article/pii/S0164121219302602
9. Blanthorn OA, Caine CM, Navarro-López EM (2019) Evolution of communities of software: using tensor decompositions to compare software ecosystems. Applied Network Science 4(1):120. https://doi.org/10.1007/s41109-019-0193-5
10. Blasi A, Kuznetsov K, Goffi A, Castellanos SD, Gorla A, Ernst MD, Pezzè M (2017) Semantic-based analysis of javadoc comments. In: Proceedings of the Seminar Series on Advanced Techniques and Tools for Software Evolution SATToSE 2017, SATToSE, Madrid, Spain, pp 1–5
11. Brabra H, Mtibaa A, Petrillo F, Merle P, Sliman L, Moha N, Gaaloul W, Guéhéneuc YG, Benatallah B, Gargouri F (2019) On semantic detection of cloud api (anti)patterns. Information and Software Technology 107:65–82. https://doi.org/10.1016/j.infsof.2018.10.012
12. Broy M (2022) Software system documentation: Coherent description of software system properties. In: Margaria T, Steffen B (eds) Leveraging Applications of Formal Methods, Verification and Validation. Software Engineering, Springer Nature Switzerland, Cham, pp 10–27
13. Bryan J (2018) Excuse me, do you have a moment to talk about version control? The American Statistician 72(1):20–27. https://doi.org/10.1080/00031305.2017.1399928
14. Bryan J (2021) Happy Git and GitHub for the useR. RStudio, https://happygitwithr.com/index.html
15. Budgen D (2003) Software Design, 2nd edn. Addison-Wesley Longman Publishing Co., Inc, USA
16. Chambers JM (2008) Software for Data Analysis: Programming With R, vol 2. Springer, CA, USA
17. Cho W, Lim Y, Lee H, Varma MK, Lee M, Choi E (2014) Big Data Analysis with Interactive Visualization using R packages. In: Proceedings of the 2014 International Conference on Big Data Science and Computing, Association for Computing Machinery, Beijing, China, BigDataScience '14, pp 1–6, 10.1145/2640087.2644168
18. Claes M, Mens T, Tabout N, Grosjean P (2015) An empirical study of identical function clones in CRAN. In: 2015 IEEE 9th International Workshop on Software Clones (IWSC), IEEE, Montreal, Canada, pp 19–25, 10.1109/IWSC.2015.7069885
19. Codabux Z, Vidoni M, Fard F (2021) Technical Debt in the Peer-Review Documentation of R Packages: a rOpenSci Case Study. 2021 International Conference on Mining Software Repositories. IEEE, Madrid, Spain, pp 195–206
20. Datta S, Nagabandi V (2017) Integrating data science and r programming at an early stage. In: 2017 IEEE 4th International Conference on Soft Computing Machine Intelligence (ISCMI), IEEE, Mauritius, pp 1–5, 10.1109/ISCMI.2017.8279587
21. Decan A, Mens T, Claes M, Grosjean P (2015) On the development and distribution of r packages: An empirical analysis of the r ecosystem. In: Proceedings of the 2015 European Conference on Software Architecture Workshops, Association for Computing Machinery, New York, NY, USA, ECSAW '15, pp 1–6, 10.1145/2797433.2797476
22. Decan A, Mens T, Claes M, Grosjean P (2016) When github meets cran: An analysis of inter-repository package dependency problems. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, Osaka, Japan, vol 1, pp 493–504, 10.1109/SANER.2016.12
23. Dekel U, Herbsleb JD (2009) Improving api documentation usability with knowledge pushing. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, USA, ICSE '09, p 320-330, 10.1109/ICSE.2009.5070532, https://doi.org/10.1109/ICSE.2009.5070532
24. Fucci D, Mollaalizadehbahnemiri A, Maalej W (2019) On using machine learning to identify knowledge in api reference documentation. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, USA, ESEC/FSE 2019, p 109-119, 10.1145/3338906.3338943
25. German DM, Adams B, Hassan AE (2013) The Evolution of the R Software Ecosystem. In: 17th European Conference on Software Maintenance and Reengineering, IEEE, Genova, Italy, pp 243–252, 10.1109/CSMR.2013.33, iSSN: 1534-5351
26. Groher I, Weinreich R (2017) An interview study on sustainability concerns in software development projects. In: 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, Austria, pp 350–358, 10.1109/SEAA.2017.70

27. Hinsen K (2009) The promises of functional programming. Computing in Science Engineering 11(4):86–90. https://doi.org/10.1109/MCSE.2009.129
28. Hornik K (2012) Are there too many r packages? Austrian Journal of Statistics 41(1):59–66
29. Howison J, Deelman E, McLennan MJ, Ferreira da Silva R, Herbsleb JD (2015) Understanding the scientific software ecosystem and its impact: Current and future measures. Research Evaluation 24(4):454–470. https://doi.org/10.1093/reseval/rvv014
30. Huang Q, Shihab E, Xia X, Lo D, Li S (2018) Identifying self-admitted technical debt in open source projects using text mining. Empirical Software Engineering 23(1):418–451
31. Ihaka R (2017) The r project: A brief history and thoughts about the future. https://www.stat.auckland.ac.nz/~ihaka/downloads/Massey.pdf
32. Kalliamvakou E, Gousios G, Blincoe K, Singer L, German DM, Damian D (2014) The promises and perils of mining github. In: Proceedings of the 11th Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR 2014, p 92-101, 10.1145/2597073.2597074
33. Königstorfer F, Thalmann S (2021) Software documentation is not enough! requirements for the documentation of ai. Digital Policy, Regulation and Governance 23(5):475–488. https://doi.org/10.1108/DPRG-03-2021-0047
34. Korkmaz G, Kelling C, Robbins C, Keller SA (2018) Modeling the Impact of R Packages Using Dependency and Contributor Networks. In: 2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM), IEEE/ACM, Barcelona, Spain, pp 511–514, 10.1109/ASONAM.2018.8508255, iSSN: 2473-991X
35. Kross S, Peng RD, Caffo BS, Gooding I, Leek JT (2020) The democratization of data science education. The American Statistician 74(1):1–7. https://doi.org/10.1080/00031305.2019.1668849
36. Li K, Chen PY, Yan E (2019) Challenges of measuring software impact through citations: An examination of the lme4 R package. Journal of Informetrics 13(1):449–461. https://doi.org/10.1016/j.joi.2019.02.007
37. Liu J, Huang Q, Xia X, Shihab E, Lo D, Li S (2020) Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society, IEEE, Seoul, South Korea, pp 1–10
38. Liu X, Wang D, Wang A, Hou Y, Wu L (2021) HAConvGNN: Hierarchical attention based convolutional graph neural network for code documentation generation in Jupyter notebooks. In: Findings of the Association for Computational Linguistics: EMNLP 2021, Association for Computational Linguistics, Punta Cana, Dominican Republic, pp 4473–4485, 10.18653/v1/2021.findings-emnlp.381
39. Maalej W, Robillard MP (2013) Patterns of knowledge in api reference documentation. IEEE Transactions on Software Engineering 39(9):1264–1282. https://doi.org/10.1109/TSE.2013.12
40. Maddumage C, Dhanushika MP (2018) R programming for Social Network Analysis - A Review. In: 2018 3rd International Conference on Information Technology Research (ICITR), IEEE, Moratuwa, Sri Lanka, pp 1–5, 10.1109/ICITR.2018.8736142
41. McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia medica: Biochemia medica 22(3):276–282
42. Meng M, Steinhardt S, Schubert A (2019) How developers use api documentation: An observation study. Commun Des Q Rev 7(2):40–49. https://doi.org/10.1145/3358931.3358937
43. Miceli Barone AV, Sennrich R (2017) A Parallel Corpus of Python Functions and Documentation Strings for Automated Code Documentation and Code Generation. In: Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers), Asian Federation of Natural Language Processing, Taipei, Taiwan, pp 314–319, https://aclanthology.org/I17-2053
44. Milewicz R, Pinto G, Rodeghero P (2019) Characterizing the Roles of Contributors in Open-Source Scientific Software Projects. In: 16th International Conference on Mining Software Repositories (MSR), IEEE/ACM, Canada, pp 421–432, 10.1109/MSR.2019.00069
45. Monperrus M, Eichberg M, Tekes E, Mezini M (2012) What should developers be aware of? an empirical study on the directives of api documentation. Empirical Software Engineering 17(6):703–737. https://doi.org/10.1007/s10664-011-9186-4
46. Morandat F, Hill B, Osvald L, Vitek J (2012) Evaluating the Design of the R Language. In: Noble J (ed) ECOOP-Object-Oriented Programming. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 104–131
47. Nayebi M, Kuznetsov K, Chen P, Zeller A, Ruhe G (2018) Anatomy of functionality deletion: An exploratory study on mobile apps. In: Proceedings of the 15th International Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '18, p 243-253, 10.1145/3196398.3196410
48. Nybom K, Ashraf A, Porres I (2018) A systematic mapping study on api documentation generation approaches. In: 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), IEEE, Czech Republic, pp 462–469, 10.1109/SEAA.2018.00081

49. Ooms J (2013) Possible Directions for Improving Dependency Versioning in R. The R Journal 5(1):197, 10.32614/RJ-2013-019, https://journal.r-project.org/archive/2013/RJ-2013-019/index.html
50. Palma F, Gonzalez-Huerta J, Moha N, Guéhéneuc YG, Tremblay G (2015) Are restful apis well-designed? detection of their linguistic (anti)patterns. In: Barros A, Grigori D, Narendra NC, Dam HK (eds) Service-Oriented Computing. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 171–187
51. Perperoglou A, Sauerbrei W, Abrahamowicz M, Schmid M (2019) A review of spline function procedures in R. BMC Medical Research Methodology 19(1):46. https://doi.org/10.1186/s12874-019-0666-3
52. Pinto G, Wiese I, Dias LF (2018) How do scientists develop scientific software? an external replication. In: IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE, Campobasso, Italy, pp 582–591, 10.1109/SANER.2018.8330263
53. Plakidas K, Schall D, Zdun U (2017) Evolution of the R software ecosystem: Metrics, relationships, and their impact on qualities. Journal of Systems and Software 132:119–146. https://doi.org/10.1016/j.jss.2017.06.095
54. Robillard MP (2009) What makes apis hard to learn? answers from developers. IEEE software 26(6):27–34
55. Robillard MP, DeLine R (2011) A field study of api learning obstacles. Empirical Software Engineering 16(6):703–732
56. Shyam R, Singh R (2021) A taxonomy of machine learning techniques. Journal of Advancements in Robotics 8(3):18–25p
57. Souza R, Oliveira A (2017) Guideautomator: Continuous delivery of end user documentation. In: 39th International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track (ICSE-NIER), pp 31–34, 10.1109/ICSE-NIER.2017.10
58. Storer T (2017) Bridging the chasm: A survey of software engineering practice in scientific programming. ACM Comput Surv 50(4), 10.1145/3084225
59. Stulova N, Blasi A, Gorla A, Nierstrasz O (2020) Towards detecting inconsistent comments in java source code automatically. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM), IEEE, Adelaide, Australia, pp 65–69, 10.1109/SCAM51674.2020.00012
60. Tan L (2015) Chapter 17 - code comment analysis for improving software quality**this chapter contains figures, tables, and text copied from the author's phd dissertation and the papers that the author of this chapter coauthored [[3], [1], [35], [7]]. sections 17.2.3, 17.4.3, 17.5, and 17.6 are new, and the other sections are augmented, reorganized, and improved. In: Bird C, Menzies T, Zimmermann T (eds) The Art and Science of Analyzing Software Data, Morgan Kaufmann, Boston, pp 493 – 517, https://doi.org/10.1016/B978-0-12-411519-4.00017-3
61. Tan SH, Marinov D, Tan L, Leavens GT (2012) @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, IEEE, Montreal, Canada, pp 260–269, 10.1109/ICST.2012.106
62. Thieme N (2018) R generation. Significance 15(4):14–19. https://doi.org/10.1111/j.1740-9713.2018.01169.x
63. Treude C, Middleton J, Atapattu T (2020) Beyond accuracy: Assessing software documentation quality. In: 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, p 1509-1512, https://doi.org/10.1145/3368089.3417045
64. Turcotte A, Vitek J (2019) Towards a Type System for R. In: Proceedings of the 14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems, Association for Computing Machinery, London, United Kingdom, ICOOOLPS '19, pp 1–5, 10.1145/3340670.3342426
65. Uddin G, Robillard MP (2015) How api documentation fails. IEEE software 32(4):68–75
66. Vidoni M (2021a) Evaluating unit testing practices in r packages. In: Proceedings of the 43rd International Conference on Software Engineering (ICSE), IEEE, Madrid, Spain, pp 1–12
67. Vidoni M (2021b) Self-Admitted Technical Debt in R Packages: An Exploratory Study. In: International Conference on Mining Software Repositories, IEEE, Madrid, Spain, pp 179–189, https://doi.ieeecomputersociety.org/10.1109/MSR52588.2021.00030
68. Vidoni M (2022a) A Systematic Process for Mining Software Repositories: Results from a Systematic Literature Review. Information and Software Technology p 106791, https://doi.org/10.1016/j.infsof.2021.106791, https://www.sciencedirect.com/science/article/pii/S0950584921002317
69. Vidoni M (2022) Understanding Roxygen Package Documentation in R. Journal of Systems and Software 188:111265. https://doi.org/10.1016/j.jss.2022.111265, https://www.sciencedirect.com/science/article/pii/S0164121222000310
70. Villegas E, Labrador E, Fonseca D, Fernández-Guinea S (2019) Validating game mechanics and gamification parameters with card sorting methods. In: Rocha Á, Adeli H, Reis LP, Costanzo S (eds) New

Knowledge in Information Systems and Technologies. Springer International Publishing, Cham, pp 392–401
71. Wang X, Lee M, Pinchbeck A, Fard F (2019) Where does lda sit for github? In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW), pp 94–97, 10.1109/ASEW.2019.00037
72. Whitworth B, Ahmad A, Soegaard M, Dam R (2006) Encyclopedia of Human Computer Interaction. IGI Publishing, Hershey, PA, Information Science Reference - Imprint of
73. Wickham H (2015) Advanced R. Chapman & Hall, CRC The R Series, CRC Press, Boca Raton, Florida
74. Wickham H (2019) roxygen2 7.0.0. https://www.tidyverse.org/blog/2019/11/roxygen2-7-0-0/
75. Wickham H, Grolemund G (2017) R for Data Science: Import, Tidy, Transform, Visualize, and Model Data, 1st edn. O'Reilly Media Inc, USA
76. Zagalsky A, German DM, Storey MA, Teshima CG, Poo-Caamaño G (2018) How the r community creates and curates knowledge: an extended study of stack overflow and mailing lists. Empirical Software Engineering 23(2):953–986
77. Zampetti F, Kapur R, Di Penta M, Panichella S (2022) An empirical characterization of software bugs in open-source cyber-physical systems. Journal of Systems and Software 192:111425. https://doi.org/10.1016/j.jss.2022.111425
78. Zanella G, Liu CZ (2020) A Social Network Perspective on the Success of Open Source Software: The Case of R Packages. In: Hawaii International Conference on System Sciences, HICSS, HAwaii, pp 471–480, 10.24251/HICSS.2020.058
79. Zhou X, Jin Y, Zhang H, Li S, Huang X (2016) A map of threats to validity of systematic literature reviews in software engineering. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), IEEE, Hamilton, New Zealand, pp 153–160, 10.1109/APSEC.2016.031