Paper OS14

Unveiling R Package Risk Assessment: Comparative Analysis of Metadata Extraction Techniques

Malte Stein, Cytel, Berlin, Germany
Sebastia Barcelo, Cytel, Geneva, Switzerland

ABSTRACT

In this paper, we delve into our innovative approach for assessing risk associated with R public packages from CRAN, focusing on the extraction of metadata crucial for informed decision-making. We elucidate our methodology for accessing R packages metadata, shedding light on the intricacies of metadata extraction. Through a comparative study, we showcase the disparities in metadata extracted using our approach versus conventional methods like the R riskmetric package.

By highlighting these differences, we provide valuable insights into the limitations of existing techniques and the advantages of our approach in comprehensively assessing package risk. Join us to gain a deeper understanding of R package risk assessment and discover new avenues for enhancing package evaluation practices.

INTRODUCTION

CRAN (Comprehensive R Archive Network) is the main source for R packages, widely used in data science, statistics, and research. While these packages offer valuable tools, they can present risks like poor maintenance, bugs, or security issues, making risk assessment crucial for safe and reliable use. For industries requiring GxP (Good Practice) compliance, like pharmaceuticals, controlled and well-documented use of software packages is essential to ensure quality and regulatory standards.

Current methods tend to focus only on the latest versions of packages, but many users still rely on older versions, which may have their own risks. Our goal is to assess the risk for all CRAN packages, including older versions, to provide a comprehensive view of potential issues over time.

In this paper, we introduce a method to extract metadata from CRAN packages for risk assessment. We compare our approach with existing tools, such as the riskmetric package, and demonstrate how it provides better insight into package risks, supporting GxP-compliant practices.

BACKGROUND: RISK REQUIREMENTS FOR CYTEL'S USE OF R PACKAGES

Cytel utilizes R across multiple business units for a range of tasks, each with different requirements for the CRAN packages used. Some tasks may even demand a validation process in addition to risk assessment to ensure compliance with regulatory standards and internal processes. However, this paper will focus exclusively on the risk assessment aspect of CRAN packages, without delving into validation processes or other repositories.

Risk tolerance can vary significantly, even for the same task, depending on the context. For example, using R to program datasets in a production environment, with subsequent review in SAS, might require a lower level of risk assessment compared to using the same R package for both programming and reviewing. (Note: Check also differences in SAS and R on https://psiaims.github.io/CAMIS/). Different settings and tasks demand tailored approaches to risk, and understanding these nuances is key to effectively managing package use in a controlled and compliant manner.

BACKGROUND: RISK ASSESSMENT WORKFLOW

The following workflows layout the general risk-based approach without going into detail.

Planning:

- Define the required risk categories based on all relevant use cases. (E.g., C1 for "low risk" up to C5 for "don't use")
- Identify key package characteristics for evaluation and establish mapping rules to align them with the defined risk categories.

Data Collection:

- Automate the ongoing collection of package characteristics, ensuring data is gathered for all necessary snapshot dates or package versions.
- Optionally, allow manual overrides of automatically gathered data.

Single Package Risk Assessment:

- Assign risk categories to individuals or groups of characteristics independently, without considering the risk of the dependent packages. The worst risk category among these will determine the overall risk level for the package.

Overall Package Risk Assessment

- Since many packages depend on other packages, the overall risk assessment must account for both the package itself and its recursive dependencies. The worst single-package risk category, including dependencies, will determine the overall risk category.

This risk assessment workflow allows the identification of specific package characteristics contributing to the overall risk. This enables corrective actions to be taken as needed. For example, if a package lacks test coverage, internal tests can be developed, or external validation, such as papers or supporting documentation, can be used to mitigate the risk. This flexible approach ensures that risks are addressed efficiently, without compromising the use of valuable packages. Any manual updates to package characteristics will be reflected in the automated risk report and recorded in the database.

METADATA OF INTEREST:

When assessing the risk of R packages from CRAN, several key metadata characteristics are crucial to consider:

- **Version**: Each version of a package can fix bugs or introduce new ones. Evaluating both the current and historical versions helps assess stability and risk.
- **Dependence**: Packages often depend on other packages, and each dependency introduces potential risk, especially if those dependencies are outdated or no longer actively maintained.
- **License**: A package's license governs its permissible use. Some licenses may restrict usage in specific tasks, which can pose compliance challenges.
- **Maintainer Activity**: Packages with active maintainers are generally more reliable. In contrast, packages with little or no recent activity may indicate a lack of updates or support, increasing risk.
- **CRAN Checks**: CRAN performs checks to ensure packages are installed and run without errors, work across multiple platforms, and are well-documented, providing a baseline of reliability.
- **Test Coverage**: The extent to which a package's functionality is tested gives insight into its robustness and reliability.
- **Download Rates**: High download rates suggest community trust and frequent use, which often correlates with package quality and reliability.
- **Supporting Documentation**: Academic papers or technical documentation supporting the package can further validate its quality and use in specialized settings.

Most characteristics can be found already in the package description or source files of packages of the CRAN repository. But not all the metadata is stored in CRAN (e.g., bug pages or supporting documentation) and must be collected outside of CRAN.

METADATA IN CRAN

The key information of a package is stored in the package description file and controlled by the CRAN checks. The following elements are commonly included:

- **Package:** The name of the package.
- **Version:** The current version of the package.
- **Title:** A brief title summarizing the package's purpose.
- **Description:** A more detailed explanation of the package, including its functionality and usage.
- **Authors**: Information about the author(s) and maintainer(s) of the package
- **Maintainer**: The person responsible for package updates and bug fixes, with an email address.
- **License**: The license under which the package is distributed (e.g., GPL-3, MIT).
- **Depends**: Specifies which version of the R and other packages the package depends on to work properly.
- **Imports**: Lists of other packages that the package uses internally and needs to be installed, but they are not directly attached by the user.
- **Suggests**: Packages that are suggested for additional functionality or examples but are not required for the package to work.
- **Enhances**: Lists packages that the package can optionally interact with, enhancing functionality.
- **LinkingTo**: For packages that need to compile C, C++, or Fortran code, this section indicates which packages need to be linked against.
- **Repository**: The location of the package repository (usually CRAN).
- **URL**: URLs to the package website, GitHub repository, or documentation.
- **BugReports**: URL for reporting bugs (e.g., GitHub Issues page).
- **VignetteBuilder**: Specifies packages required to build vignettes (detailed documentation).
- **Collate**: The order in which the R files in the package should be sourced.

ACCESSING CRAN METADATA

The package description will be shared with additional information on CRAN and is part of the source file of the package.
Example: https://cran.r-project.org/web/packages/dplyr/index.html.

CRAN is providing also an additional database for the latest version of each package on https://cran.r-project.org/web/packages/packages.rds (or with function tools::CRAN_package_db())
This dataset contains the following columns:

```
 [1] "Package"                "Version"              "Priority"               "Depends"              "Imports"
 [6] "LinkingTo"              "Suggests"             "Enhances"               "License"              "License_is_FOSS"
[11] "License_restricts_use"  "OS_type"              "Archs"                  "MD5sum"               "NeedsCompilation"
[16] "Additional_repositories" "Author"              "Authors@R"              "Biarch"               "BugReports"
[21] "BuildKeepEmpty"         "BuildManual"          "BuildResaveData"        "BuildVignettes"       "Built"
[26] "ByteCompile"            "Classification/ACM"   "Classification/ACM-2012" "Classification/JEL"   "Classification/MSC"
[31] "Classification/MSC-2010" "Collate"             "Collate.unix"           "Collate.windows"      "Contact"
[36] "Copyright"              "Date"                 "Date/Publication"       "Description"          "Encoding"
[41] "KeepSource"             "Language"             "LazyData"               "LazyDataCompression"  "LazyLoad"
[46] "MailingList"            "Maintainer"           "Note"                   "Packaged"             "RdMacros"
[51] "StagedInstall"          "SysDataCompression"   "SystemRequirements"     "Title"                "Type"
[56] "URL"                    "UseLTO"               "VignetteBuilder"        "ZipData"              "Path"
[61] "X-CRAN-Comment"         "Published"            "Reverse depends"        "Reverse imports"      "Reverse linking to"
[66] "Reverse suggests"       "Reverse enhances"     "Deadline"               "DOI"
```

METADATA OUTSIDE OF CRAN

Some important package information is not stored within the package itself, but is hosted on external webpages, such as test coverage reports or links to the package's GitHub repository. To collect this data, two common methods are used: APIs and web scraping.

API AND WEB SCRAPING: AN EXPLANATION

An API (Application Programming Interface) is a tool provided by websites or services that allows users to request specific pieces of data in a structured format, such as JSON or XML, without needing to interact with the website itself. APIs are typically well-documented and provide a reliable, controlled way to access data directly from a server.
In contrast, web scraping involves extracting data directly from the HTML content of a webpage. Instead of requesting specific data through a defined interface, as with an API. Web scraping parses the webpage's structure to manually collect the information on the site. This method is more flexible, allowing access to any visible data on the page, but is less structured and might fail if the webpage layout changes.

CHALLENGES WITH API

Using an API offers several advantages: it is efficient, easy to automate, and provides data in a structured format. APIs are also more stable over time, as they are specifically designed for external use and are less likely to undergo sudden changes. However, there are limitations. For instance, when collecting information for a large number of CRAN packages, potential request limits on APIs are quickly reached. To get around this, requests can be spread out over time, slowing down the process. Additionally, APIs often provide limited data. Typically, only the current status of the latest version of a package is accessible and may not offer historical information by published CRAN versions.

CHALLENGES WITH WEB SCRAPING

Web scraping, while more manual, offers greater flexibility. It allows you to collect any data visible on a webpage, including historical data or information not available via an API. It also bypasses the request limits that APIs impose, making it ideal for large-scale data collection. However, web scraping comes with its own challenges. It is more fragile, as even small changes to a webpage's layout can break the scraping script. Web scraping is also more work-intensive to set up and maintain, requiring custom code for each website being scraped. Additionally, dynamic content (such as data that only appears after running JavaScript) requires more complex scraping techniques, further complicating the process.

Note: We only scraped publicly available, non-personal data related to open-source R packages. However, it is important to be aware that web scraping may violate local laws or website policies. Some websites actively block scraping attempts, so it is crucial to review and comply with legal and policy guidelines before proceeding.

Example for API, web scraping and source

For instance, to collect the test coverage of the dplyr package, both API and web scraping can be used. But for both methods additional information must be extracted from the package description first.

API:
The API call on Codecov.io is https://api.codecov.io/api/v2/{service}/{owner_username}/repos/{repo_name}/totals.
The R code for dplyr:

jsonlite::read_json("https://api.codecov.io/api/v2/gh/tidyverse/repos/dplyr/totals")$totals$coverage

Result: 90.95%
This approach is straightforward, using the Codecov API to return the coverage results in a structured format. The benefits include its reliability and ease of automation, though it is limited to the current state of the package and is subject to API rate limits. Also, the information needed for the API about the service, owner, repo name and the publishing to Codecov is hidden in the linked webpages or readme.

Web Scraping:
The link to the test coverage is included in the readme file of the dplyr package and can be extracted with the following R code:

xml2::xml_text(xml2::read_html("https://codecov.io/gh/tidyverse/dplyr/branch/main/graph/badge.svg"))

Result: "codecovcodecov91%91%"

Web scraping provides similar results, but it requires manually extracting the text from the HTML structure of the page. While scraping is flexible and does not have the limitations of an API, it requires careful coding and is more sensitive to changes in the webpage's layout or content.

CONCLUSION FOR API AND WEB SCRAPPING

We were able to extract the most needed information from the package description or database of the package repository itself. But analyzing different source pages (e.g., GitHub, GitLab, Bitbucket, R-forge.r-project.org) and their supporting pages (e.g., Codecov, Coverall, bug pages outside of source page) was not stable over time and reliable. So, we tried to simplify the process on limited data sources and define rules, which can be applied to all CRAN packages. Web scraping and API should be limited as much as possible, performed ideally on controlled internal systems (e.g., Posit package manager) or at stable on consistent webpages across CRAN packages.

R-hub is hosting an unofficial GitHub CRAN mirror on https://github.com/cran. The source of all CRAN packages is available as a GitHub repository. Web scraping or using the GitHub API only of this page is more robust and easier than trying to analyze all diverse types of source page. It contains only the package source from CRAN, i.e., the bugs are not reported here. But checking source content information (e.g., existing tests folder, vignettes) can be checked here without installing or downloading package first.

CASE STUDY

In the following sections we describe our experience extracting different R package information order by importance. Whenever possible we used our internal R system (e.g., Posit package manager) to collect data, but for this paper we often replaced the data source with public CRAN or public Posit package manager to make it usable for everyone.

Some metadata extraction methods require the source code of a package. The following example code download and store the source code for dplyr package:

```
package_name = "dplyr"
temp_dir <- tempfile("temp_source")
dir.create(temp_dir)
package_tarball_url <- https://cran.r-project.org/src/contrib/dplyr_1.1.4.tar.gz
tarball_path <- file.path(temp_dir, paste0(package_name, ".tar.gz"))
download.file(package_tarball_url, tarball_path, quiet = TRUE)
untar(tarball_path, exdir = temp_dir)
pkg_source_path<-file.path(temp_dir, package_name)
```

The following sections are using the variable pkg_source_path as the location source code of an R package.

CRAN CHECK

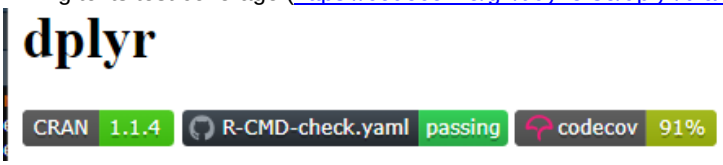
CRAN checks are automated tests that ensure R packages meet quality, compatibility, and policy standards before being accepted on CRAN. These checks verify that packages have correct structure, documentation, and metadata; run without errors across multiple platforms (Linux, Windows, macOS); and adhere to CRAN's guidelines, such as avoiding deprecated functions and non-portable code. They also evaluate examples, unit tests, and vignettes to ensure functionality and performance. CRAN Repository Policy is shared at https://cran.r-project.org/web/packages/policies.html.

The CRAN checks results are shared on https://cran.r-project.org/web/checks/check_summary_by_package.html and the database can be downloaded from https://cran.r-project.org/web/checks/check_details.rds. CRAN checks are performed for different systems (e.g., Linux or Windows).

An important aspect of CRAN checks is the R CMD check, which ensures packages are free of errors and uncommented warnings before being added to the CRAN repository. These checks are not only conducted upon initial submission but are also repeated continuously, as changes in new R versions, operating systems, or dependencies may introduce issues after a package is published. More details can be found in the CRAN Repository Policy.
The R CMD check can also be run locally within a specific R environment and package library using several user-friendly functions. However, running the R CMD check for all CRAN packages is computationally intensive and often unnecessary, as CRAN already performs these checks and publishes the results in a public database. Nevertheless, running R CMD checks within a particular environment and setup may be essential for critical tasks or packages.
The function devtools::check(pkg_source_path) runs the R CMD check for a package using its source, and the same functionality is available in the riskmetric package via riskmetric::assess_r_cmd_check(riskmetric::pkg_ref(pkg_source_path)). During testing, several issues arose related to our specific R environment, making the automated process for all CRAN packages challenging. This remains an area for future investigation.

TEST COVERAGE FROM README:
Test coverage, while not included in the package DESCRIPTION file, is a crucial quality metric for R packages. Although test folders are typically part of the source package, they are excluded by default from the binary versions during package creation. To simplify data extraction, we initially attempted to search for shared test coverage pages (e.g., Codecov.io) within the URLs listed in the package descriptions. While this approach showed some success, it proved unstable over time and across all packages. Consequently, we shifted our focus to the README files, where test coverage is frequently displayed as a badge, providing a more reliable source for this information.

A badge is a small visual indicator, usually embedded in a README file, which displays dynamic information about the package, such as test coverage, build status, or version. For example, the dplyr README contains a badge linking to its test coverage (https://codecov.io/gh/tidyverse/dplyr/branch/main/graph/badge.svg):



Extracting the coverage from these badges is a straightforward and efficient process, applicable across all CRAN packages. However, this method only retrieves the latest test coverage; older versions of a package will still show the most current coverage in their README files.

By automating this process through an API, such as in an internal Posit Package Manager, we ensure a robust data source for extracting test coverage from each new release. The README files are part of the Posit package manager database (e.g., jsonlite::read_json("https://packagemanager.posit.co/__api__/repos/cran/packages/dplyr")). The following table provides a snapshot of test coverage extracted from badges included in README files:

Test Coverage from README Files (Snapshot of September 19, 2024)

Test coverage	All (incl. N/A)	>=0%	>=50%	>=75%	>=90%	>=95%
Number of packages	21438	1762	1537	1257	839	545

Note: Test coverage refers to the percentage of code lines executed during testing. However, it does not provide any information about the quality or thoroughness of the tests themselves.

TEST COVERAGE CALCULATION
A more precise and version-specific method involves recalculating test coverage by running the tests included in a package's source code using tools like covr::package_coverage("dplyr"). The riskmetric package also uses this function internally. Several approaches were explored for recalculating test coverage.

Setup
The R package must be installed with its tests and all dependencies. This requires the installation option --install-tests to be set. For example, to install the dplyr package with tests included:

utils::install.packages(pkgs = "dplyr",type="source",repos = "https://cloud.r-project.org/", quiet= F, INSTALL_opts = c("--install-tests"))

Test coverage calculation also requires the package's source code, as it measures the percentage of code lines tested.

Using riskmetric

In riskmetric version 0.2.4, test coverage is calculated for package references of type "pkg_score". The path or package reference to the source is required, not just the package name:

riskmetric::assess_covr_coverage(riskmetric::pkg_ref(pkg_source_path)) #Correct

Incorrect usage, such as providing only the package name ("dplyr"), will not work:

#Not working# riskmetric::assess_covr_coverage(riskmetric::pkg_ref("dplyr"))

Result:

```
> ref<-riskmetric::pkg_ref(pkg_path)
> riskmetric::assess_covr_coverage(ref)
$filecoverage
          R/across.R          R/all-equal.R           R/arrange.R          R/bind-cols.R          R/bind-rows.R                R/by.R        R/case-match.R
               97.88                  82.93                 94.74                 100.00                 100.00                100.00                100.00
        R/case-when.R          R/coalesce.R    R/colwise-arrange.R   R/colwise-distinct.R     R/colwise-filter.R       R/colwise-funs.R   R/colwise-group-by.R
              100.00                100.00                100.00                 100.00                 100.00                 92.50                 94.44
    R/colwise-mutate.R     R/colwise-select.R          R/colwise.R     R/compat-dbplyr.R R/compat-name-repair.R    R/compute-collect.R        R/conditions.R
               97.65                 98.04                 82.28                   3.57                  59.82                 33.33                 95.48
    R/consecutive-id.R          R/context.R          R/copy-to.R         R/count-tally.R          R/data-mask.R           R/dbplyr.R           R/defunct.R
              100.00                 86.27                 84.62                  87.67                  97.20                  3.70                100.00
    R/deprec-combine.R     R/deprec-context.R        R/deprec-dbi.R          R/deprec-do.R        R/deprec-funs.R    R/deprec-lazyeval.R   R/deprec-src-local.R
              100.00                100.00                 18.75                  96.30                 100.00                 76.16                 58.82
     R/deprec-tibble.R              R/desc.R           R/distinct.R        R/doc-methods.R             R/error.R           R/explain.R            R/filter.R
               45.45                100.00                100.00                   0.00                  76.92                  0.00                100.00
             R/funs.R           R/generics.R           R/group-by.R         R/group-data.R          R/group-map.R         R/group-nest.R        R/group-split.R
              100.00                 97.01                 96.51                 100.00                  90.16                100.00                 78.12
       R/group-trim.R        R/grouped-df.R        R/groups-with.R            R/if-else.R             R/join-by.R          R/join-cols.R     R/join-common-by.R
              100.00                 93.39                100.00                 100.00                  93.95                 98.36                  0.00
       R/join-cross.R         R/join-rows.R               R/join.R           R/lead-lag.R            R/locale.R             R/mutate.R          R/n-distinct.R
              100.00                 98.94                 97.74                 100.00                 100.00                 92.42                100.00
            R/na-if.R              R/near.R           R/nest-by.R          R/nth-value.R          R/order-by.R             R/pick.R          R/progress.R
              100.00                100.00                100.00                 100.00                 100.00                 98.55                 30.51
            R/pull.R              R/rank.R             R/recode.R            R/reframe.R           R/relocate.R           R/rename.R               R/rows.R
               42.11                100.00                 86.82                 100.00                 100.00                100.00                 98.38
         R/rowwise.R            R/sample.R     R/select-helpers.R             R/select.R              R/sets.R              R/slice.R          R/src-dbi.R
               97.92                 94.74                 94.12                  96.00                 100.00                 98.54                100.00
             R/src.R           R/summarise.R              R/tbl.R            R/top-n.R          R/transmute.R              R/ts.R     R/utils-format.R
               42.86                 93.64                 62.50                 100.00                 100.00                100.00                 28.57
           R/utils.R             R/vctrs.R     R/vec-case-match.R     R/vec-case-when.R              R/zzz.R         src/chop.cpp          src/dplyr.h
               94.23                100.00                100.00                 100.00                  19.35                100.00                100.00
       src/filter.cpp          src/funs.cpp       src/group_by.cpp       src/group_data.cpp       src/imports.cpp         src/init.cpp         src/mask.cpp
               97.50                100.00                 93.16                 100.00                 100.00                100.00                 86.11
       src/mutate.cpp     src/reconstruct.cpp          src/slice.cpp       src/summarise.cpp
               94.23                 83.93                100.00                  95.33

$totalcoverage
[1] 90.94
```

The recalculated test coverage may differ from the published value on Codecov.io (e.g., 90.95%) due to variations in test skipping options or R environment settings (e.g., Linux vs. Windows).

Using covr directly

Test coverage can also be computed directly using the covr package, with similar methods to those in riskmetric. This allows for adjustments specific to the environment, which might not be possible within riskmetric. Example code for dplyr:

```
> coverage<-covr::package_coverage(
+   path = pkg_source_path,
+   type = "none",
+   stop_on_error = FALSE,
+   code = "tools::testInstalledPackage('dplyr', types = 'tests')"
+ )
> covr::percent_coverage(coverage)
[1] 90.94111
```

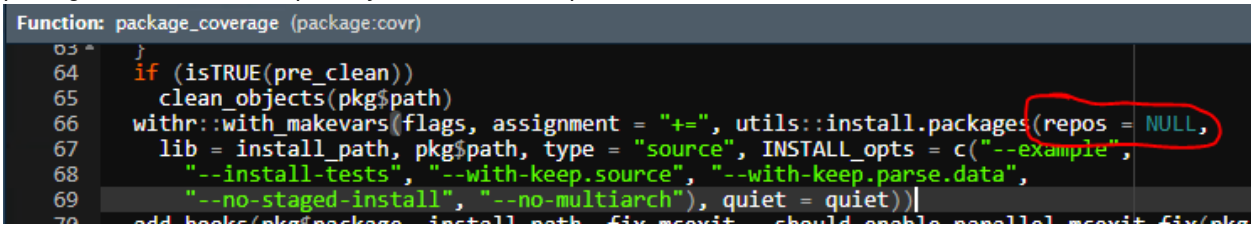
Note:

Test coverage can be calculated without explicitly defining the test code (code="..."), but this approach was unstable across different packages. Test coverage can only be calculated if the test code runs without errors, which was not always the case without specifying the code.

Conclusion:

Automating this process for all CRAN packages is feasible but demands significant computational resources. It is better suited for high-priority packages or scheduled intervals (e.g., daily for new releases). Limited manual updates may still be needed, as different library setups (e.g., using older or newer dependency versions) can produce varying results or issues.

Note for potential issues:
The default option in covr::package_coverage is “pre_clean = TRUE”, which will trigger a re-installation of the package from the default repository with the correct options:



We were facing some issue that the package was installed or re-installed without tests, because the default repository was a binary repository or the package (without tests) was installed/linked from the renv cache. Therefore we recommend to check carefully the default repository and not using renv for the first time of calculation the test coverage. Of cource using renv or working with temporary libraries and settings (e.g., with withr) is supportive, once testing code in the specific R system is running correctly.

DOWNLOAD RATES

Apart from test coverage, download rates serve as one of the best indicators of a package's trustworthiness. Packages that are well-documented and produce reliable results are used more frequently and recommended within the R community. Higher usage leads to more users reporting bugs, suggesting improvements, and rigorously testing the package, which contributes to its robustness over time.
Download data from the RStudio CRAN mirror (cran.rstudio.com) are available for data since October 2012 and can be accessed through the cranlogs::cran_downloads function, with logs stored at http://cran-logs.rstudio.com. Below is a table showing the download rates within one year using date from this function:

Download Rates (Snapshot of September 10, 2024)

Downloads within 1 year	>=0	>=2500	>=5k	>10k	>20k	>=500k	>=1m
Number of packages	21274	14742	6732	3769	2598	485	306

Currently, R functions and APIs provide download data for a package over a selected time period, but they do not offer download counts for specific versions of a package. However, the raw logs include detailed information, such as the package version, operating system, and R version used, allowing for deeper analysis if necessary.

Note:
Riskmetric is using the cranlogs::cran_downloads function to extract the download rates of the last 365 days.

PACKAGE DEPENDENCIES, REQUIREMENTS AND LICENSE

Package dependencies, system requirements, and license information are included in the package description files and can be extracted directly, for example, from CRAN package metadata (https://cran.r-project.org/web/packages/packages.rds) or using the tools::CRAN_package_db() function.

For a thorough risk assessment, it is essential to check all recursive dependencies, as a package may have 100% test coverage, but critical calculations could rely on untested functions from other packages. This highlights the need to assess not just the package itself, but also the risks associated with its dependencies.

DOCUMENTATION, BUGS, VIGNETTES AND OTHER

Several papers suggest evaluating risk based on the number of vignettes, reported bugs, and function documentation. The current riskmetric package (version 0.2.4) performs these checks effectively for selected packages, but it remains experimental. Additionally, API rate limits restrict its use across a large number of packages simultaneously. Consequently, alternative methods were explored. While it is technically possible to analyze installed packages or source files directly, this approach is impractical given the vast number of CRAN packages available.
The decision was made not to automatically assess documentation, bugs, or vignettes for three key reasons:
It is challenging to create meaningful, general mapping rules for these characteristics across all CRAN packages. For instance, a package with no reported bugs might not be bug-free. Bugs may simply not be reported, or the package may have very few users. Conversely, a package with many open bugs could indicate a large user base, experimental features, or an inactive maintainer.

CRAN imposes minimal documentation requirements, and packages are archived if they fail checks during new R version releases. These requirements, combined with the use of download rates and test coverage, already provide a solid foundation for assessing risk.

Packages with significant issues, such as important bugs, poor documentation, or lack of user-friendliness, tend to have fewer users and lower recommendations within the community. Since these factors correlate with download rates, which are already part of the risk assessment, there is less need to assess them separately.

CONCLUSION

The source and method of collecting package metadata are crucial for accurate risk assessment. While information on the latest version of a limited number of packages can be extracted relatively easily, gathering comprehensive data on all current CRAN packages presents challenges due to computational demands and API limitations. Moreover, historical data on earlier package versions is often unavailable, difficult to retrieve, or requires manual reconstruction.

To address this, we have begun collecting data on an ongoing basis to facilitate retrospective analysis in the future. For specific time points and packages, we plan to explore historical data, enabling risk assessments for older R versions in selected projects. We are currently maintaining an internal database of key characteristics to support this effort. It would be advantageous to establish an official public database or enhance package documentation (e.g., by incorporating test coverage details in the DESCRIPTION file).

While the riskmetric R package provides much of the necessary information for installed packages, it has limitations in performing risk assessments across all CRAN packages or their historical versions. Furthermore, as the package is still in its experimental phase, it cannot yet be used for our official risk assessments.

REFERENCES

CRAN webpage: https://cran.r-project.org
CRAN packages policy: https://cran.r-project.org/web/packages/policies.html
CRAN package download logs: http://cran-logs.rstudio.com
CRAN package check results: https://cran.r-project.org/web/checks
Unofficial GitHub CRAN mirror: https://github.com/cran

RECOMMENDED READING

PHUSE paper about R Package Validation Framework: https://phuse.s3.eu-central-1.amazonaws.com/Deliverables/Data+Visualisation+%26+Open+Source+Technology/WP059.pdf
PHUSE working group about discrepancies in statistical analysis results in different programming languages: https://psiaims.github.io/CAMIS/

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Author Name: Malte Stein
Company: Cytel
Address: Potsdamer Straße 58, 10785 Berlin, Germany
Email: Malte.Stein@Cytel.com
Website: Cytel.com

Co-Author Name: Sebastia Barcelo
Company: Cytel
Address: 2nd Floor, Block H, International Center Cointrin, 1215 Geneva 15, Switzerland
Email: Sebastia.Barcelo@Cytel.com
Website: Cytel.com