PharmaSUG 2026 - Paper ET-397

# Trusting Your R Packages: A Practical, Risk-Based Approach to External Package Validation

Radhika Etikala, Valeria Duran, SCHARP at Fred Hutchinson Cancer Center, Seattle, Washington

## ABSTRACT

The increasing use of R in regulated clinical research environments introduces new challenges for ensuring reliability, reproducibility, and traceability of analytical workflows. While internally developed R packages can be validated against formal specifications, external packages present unique challenges due to their open-source nature, varying development practices, and complex dependency structures.

This paper presents a practical, risk-based framework for Performance Qualification (PQ) of external R packages within a validated scientific computing environment. The proposed Risk-Based Performance Qualification (RPQ) approach leverages existing repository checks, package metadata, and automated testing evidence to provide structured, scalable assessment of package reliability.

Rather than attempting full validation of all external packages and dependencies, RPQ focuses on generating transparent evaluation outputs that support downstream decision-making. The approach integrates with study-level processes, where additional targeted validation may be applied for high-risk use cases.

This framework demonstrates how organizations can balance regulatory expectations with practical constraints, enabling responsible and reproducible use of external R packages without imposing unsustainable validation burden. The approach is demonstrated through implementation within a validated environment, including structured validation outputs and interpretation workflows.

## INTRODUCTION

### BACKGROUND

In high-stakes environments such as pharmaceutical research, it is essential to maintain reliable workflows and tools for data analysis. When analytical results must be vetted and reproducible, the tools used to generate those results must also be reliable and reproducible. For this reason, a validated Scientific Computing Environment (SCE) is critical.

Validation of an SCE typically includes Installation Qualification (IQ), Operational Qualification (OQ), and Performance Qualification (PQ). IQ and OQ verify that software is installed correctly and operates as intended, while PQ ensures that systems perform as expected under real-world analytical conditions. This paper focuses specifically on the PQ component as it applies to external R packages.

R presents unique considerations in validation contexts. As an open-source language, R is highly transparent and supported by a collaborative community with frequent feature updates. Despite its open-source nature, base R and its recommended packages follow a structured Software Development Lifecycle (SDLC) overseen by the R Foundation and undergo rigorous checks before being released to the Comprehensive R Archive Network (CRAN). However, most analytical workflows depend on external packages that vary widely in development practices, testing coverage, and maintenance. This variability introduces a central challenge: how to assess the reliability of external R packages in regulated environments without incurring excessive validation burden.

### METRICS IN THE COMMUNITY

The R Validation Hub white paper serves as a foundational reference for validating R packages and outlines key principles that are particularly relevant in regulated environments (R Validation Hub, 2020). These include accuracy, reproducibility, and traceability, which together ensure that analytical results are correct, consistent, and auditable. Accuracy refers to the expectation that a system produces correct results for its intended purpose. Reproducibility ensures that results can be consistently regenerated

under the same conditions, while traceability allows outputs to be linked back to specific software versions and configurations.

Building on these principles, the R Validation Hub proposes a structured framework for evaluating R packages across four domains: purpose, maintenance good practice, community usage, and testing. The purpose domain considers whether a package is statistical or non-statistical and evaluates its impact on analytical outcomes. Maintenance good practice assesses whether developers follow an SDLC, provide documentation, and make their practices transparent, which can reduce the likelihood of defects. Community usage evaluates how widely a package is used, with broader usage often indicating informal validation through user feedback. Testing focuses on the extent and quality of test coverage, which directly influences confidence in package reliability.

Several organizations have adopted variations of this framework, tailoring it to their operational needs. Some implement structured qualification workflows that incorporate expert review, while others rely on automated systems that generate risk scores and flag high-risk packages for additional scrutiny. In some cases, packages may require enhanced testing before approval, while in others, they may be restricted entirely. These examples demonstrate that external package validation can be achieved using a range of approaches, from manual review to fully automated pipelines.

One notable tool supporting this process is the *riskmetric* package, which collects quantitative indicators such as CRAN check results, code coverage, open issue rates, and repository metadata (R Validation Hub et al., 2026). These indicators can be combined to produce an overall risk score. However, in practice, we found that relying solely on aggregated scores can be challenging to construct and interpret requiring standardized weighting schemes across projects and packages, along with defined risk thresholds and may obscure important contextual factors. As a result, our approach emphasizes transparent evidence rather than reducing package assessment to a single numerical value.

### CONSTRAINTS

Given the regulatory context, one might assume that validating external R packages would require extensive effort. However, several practical constraints shaped our approach. At our organization, SAS remains the primary tool for regulatory submissions, and the number of R programmers is relatively small. As a result, the time and resources available for validating external R packages are limited.

In contrast, internally developed R packages undergo rigorous validation against formal specifications using tools such as *valtools* (R Validation Hub, 2023). This process can take several months for a single package. In our most recent validation project, extending this level of validation to approximately two hundred external packages was not feasible, given the substantial time and resource requirements per package. Unlike internal packages, for which we have greater control over development and implementation, external packages vary widely in structure, documentation, and maintenance practices, further complicating the application of a uniform validation approach. The disparity in team size between those validating internal versus external packages further reinforced the need for a scalable approach.

These constraints led to the development of a pragmatic framework focused on Performance Qualification rather than full validation. The goal was to provide sufficient confidence in package behavior while remaining aligned with available resources.

## OVERVIEW OF OUR RISK-BASED PERFORMANCE QUALIFICATION (RPQ) APPROACH

The Risk-Based Performance Qualification (RPQ) framework was developed to provide a structured and scalable approach for evaluating R packages installed on the production server. The framework was designed in response to the practical and regulatory challenges associated with qualifying externally maintained open-source packages in a controlled analytical environment. While the original project charter focused on Performance Qualification of an internally defined list of R/CRAN packages, subsequent project discussions led to adoption of an alternative model that combines centralized, high-level (i.e., repository-based checks and automated evidence gathering) package qualification with study-specific review when packages are used in critical analyses. The resulting RPQ framework therefore supports both package-level qualification and downstream risk-informed decision-making.

At a high level, the RPQ workflow begins with package metadata supplied by the IT team. This metadata provides the authoritative inventory of packages and versions installed on the server and defines the scope of evaluation. Using this inventory, automated scripts run to perform repository and package-level checks. These checks generate a structured output dataset that captures package status across the installed environment. The output is then reviewed through the RPQ evaluation process, which summarizes package results and identifies packages requiring follow-up. In parallel, a separate risk assessment output is generated to provide additional package characteristics that may inform future study-level review. When issues (e.g., failed checks, version mismatches, or missing repository data) are identified, they are documented and evaluated through discrepancy handling procedures. Final package-level results are then documented in RPQ reporting and may be referenced later during preparation of study-specific R Usage Plans. The overall workflow is illustrated in Figure 1.

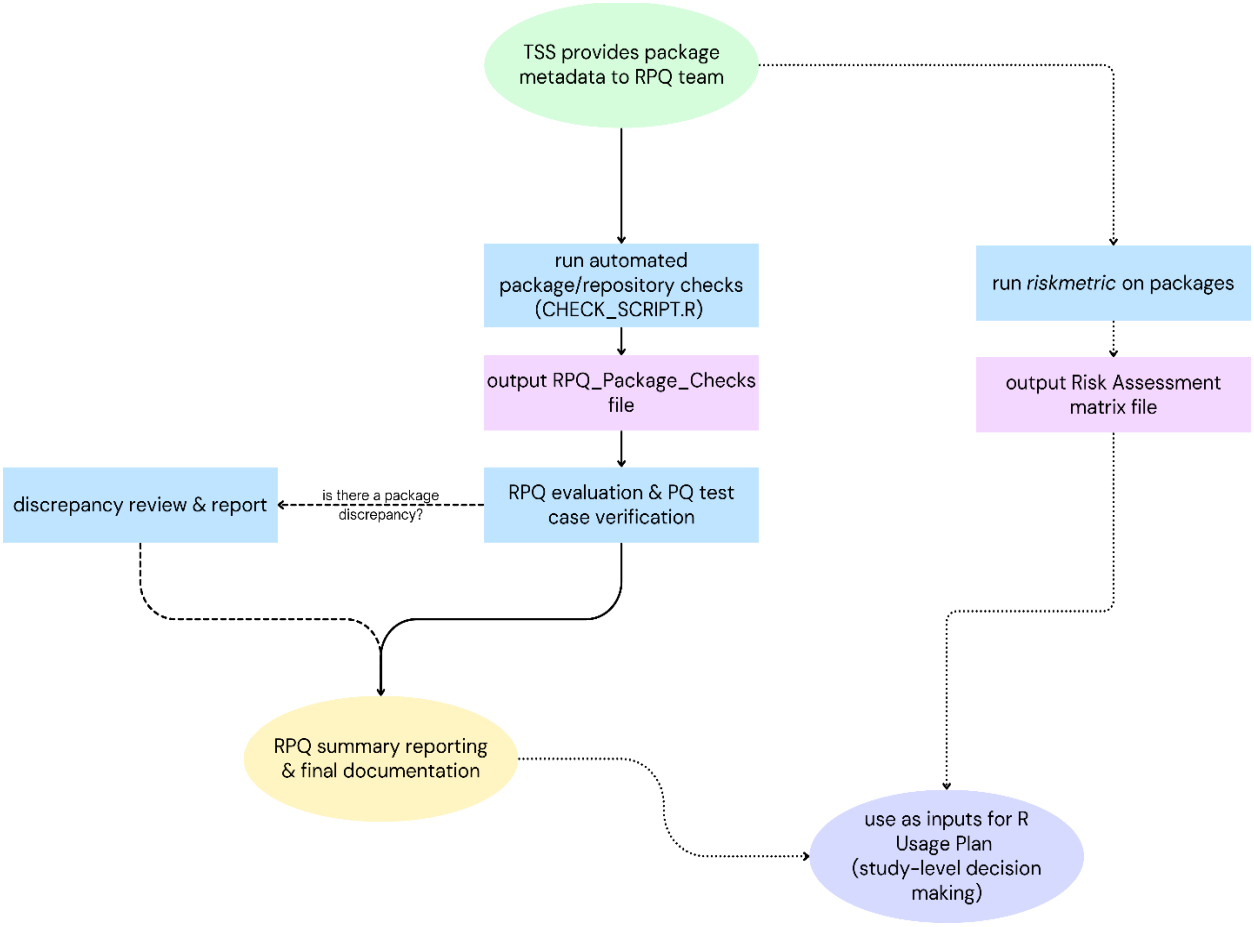


**Figure 1. RPQ Framework Workflow.**

## RPQ APPROACH IN PRACTICE

In practice, the RPQ framework is implemented as a programmer-driven, evidence-based process that relies on publicly available package information rather than attempting to recreate full validation from first principles for every package. Instead of treating all packages as requiring identical, exhaustive qualification, RPQ gathers structured evidence from relevant package sources and evaluates that evidence in a consistent manner.

The process begins by identifying installed packages and retrieving version information directly from the execution environment. Packages are then evaluated according to their source, including base, CRAN, Bioconductor, and GitHub repositories. Each source type is assessed appropriately to ensure that

repository-specific characteristics are consistently accounted for. The evaluation script retrieves repository-level check results, verifies package installation and behavior, and performs additional validation where required, such as R CMD check for selected packages. The complete implementation of this script is provided in Appendix A for reproducibility and reference.

Base packages are evaluated through environment-level installation verification, while CRAN and Bioconductor packages are assessed using repository-level checks and build reports. For GitHub packages and version-mismatched CRAN packages, additional R CMD check-based validation is performed. These checks are consolidated into a structured dataset, which serves as the primary RPQ output. A representative example of the package-check output is shown in Table 1.

| category | source | package | version | check_status | check_output | version_matches |
|---|---|---|---|---|---|---|
| PKG-BIOCONDUCTOR-01 | bioc | Biobase | 2.66.0 | NO ERROR | NA | TRUE |
| PKG-BIOCONDUCTOR-02 | bioc | BiocGenerics | 0.52.0 | NO ERROR | NA | TRUE |
| PKG-BIOCONDUCTOR-03 | bioc | flowWorkspace | 4.18.0 | NO ERROR | NA | TRUE |
| PKG-BIOCONDUCTOR-04 | bioc | COMPASS | 1.44.0 | NO ERROR | NA | TRUE |
| PKG-BIOCONDUCTOR-05 | bioc | CytoML | 2.18.1 | NO ERROR | NA | FALSE |
| PKG-BIOCONDUCTOR-06 | GitHub | MIMOSA | 1.29.1 | NO ERROR | checking for hidden files | NA |
| PKG-BIOCONDUCTOR-07 | cran | MASS | 7.3.64 | NO ERROR | NA | FALSE |
| PKG-SCHARP-001 | cran | abind | 1.4.8 | NO ERROR | NA | TRUE |
| PKG-SCHARP-002 | cran | alakazam | 1.3.0 | NO ERROR | checkRd: (-1) | TRUE |
| PKG-SCHARP-003 | cran | aod | 1.3.3 | NO ERROR | NA | TRUE |
| PKG-SCHARP-004 | cran | arrow | 18.1.0.1 | NO ERROR | checking installed | FALSE |
| PKG-SCHARP-005 | cran | arsenal | 3.6.3 | NO ERROR | NA | TRUE |

**Table 1. RPQ Package Check Output**

This dataset standardizes heterogeneous evidence across package ecosystems into a consistent format, enabling systematic review and interpretation. Each row represents a package-level evaluation record, allowing reviewers to trace results back to specific package versions and sources. The derived variable, “check_status”, as shown in Table 1 provides a simplified classification of package results, distinguishing packages with no identified issues from those requiring further review.

In parallel to the package-check output, a separate risk assessment process is used to characterize packages using metadata and quality indicators. This process is implemented through a separate script that leverages the *riskmetric* framework to derive package-level attributes, including documentation coverage, dependency structure, maintenance indicators, and adoption metrics. The full implementation of this process is provided in Appendix B.

The resulting dataset includes metrics that summarize documentation, maintenance, dependency structure, adoption, and other package characteristics. A representative excerpt is shown in Table 2.

| package | version | news_current | r_cmd_check | has_vignettes | has_website | has_maintainer | bugs_status | has_source_control | has_bug_reports_url | downloads_1yr | reverse_dependencies | dependencies |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| compiler | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | AlgebraicHaploPackage ;;; AsynchLong ;;; Ba | |
| datasets | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | acid ;;; adductomicsR ;;; ANOVAIREVA ;;; AN | |
| graphics | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | a4Base ;;; a4Classif ;;; ABarray ;;; ABC.RAP ;; | |
| grDevices | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | ABarray ;;; aberrance ;;; absegR ;;; abtest ;;; | |
| methods | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | a4Base ;;; a4Classif ;;; a4Core ;;; a4Reportin | |
| parallel | 4.4.2 | | NA | 1 | | R Core Team <do | Error in | | 0 | 0 | abclass ;;; abcrf ;;; abctool c("tools", "compil | |
| splines | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 518 | accept ;;; ACEt ;;; aCGH ;;; c("graphics", "sta | |
| stats4 | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | AHSurv ;;; AICcmodavg ;;; c("graphics", "me | |
| stats | 4.4.2 | | NA | 1 | | R Core Team <do | Error in | | 0 | 0 | a4Classif ;;; a4Core ;;; ABarray ;;; ABC.RAP ;; | |
| tcltk | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 238 | ABarray ;;; ade4TkGUI ;;; a utils ;;; Imports | |
| tools | 4.4.2 | | NA | 0 | | R Core Team <do | Error in | | 0 | 0 | absegR ;;; accessr ;;; aceEditor ;;; act ;;; Acui | |
| utils | 4.4.2 | | NA | 1 | | R Core Team <do | Error in | | 0 | 0 | a4Classif ;;; ABarray ;;; ABC.RAP ;;; ABCoptir | |

**Table 2. Risk Assessment Output**

This output is not used as a direct qualification decision tool. Instead, it provides supporting information for downstream study-level evaluation, particularly for R usage planning (Grant & Neradilek, 2026). These metrics are not interpreted in isolation but are considered in conjunction with package-check results as

well as other considerations produced outside of the RPQ process.

## RPQ IMPLEMENTATION AND TEST CASE VALIDATION

To ensure the reliability and reproducibility of RPQ outputs, the framework incorporates both automated execution and formal validation through standardized test cases. The package-check workflow is executed through CHECK_SCRIPT.R, which generates the structured output dataset described above.

Across the full collection of installed packages, the correctness and usability of this output are verified through a Performance Qualification test case. An example of the executed test case is shown in Figure 2.

| Req # | Step | Instruction | Expected Result | Actual Result | Pass/Fail | Initial / Date |
|---|---|---|---|---|---|---|
| 6.1.1 | **Confirm Package Status** | | | | | |
| | 1 | Execute the R code "*checks_df <- readr::read_csv(file.path(here::here(), "outputs", "RPQ_Package_Checks_YYYYMMDD.csv"))*" where YYYYMMDD is the date which the file was generated. | Code should run without error. | | | |
| 6.1.2 | 2 | Count the number of packages in the file by executing code "*nrow(checks_df)*" | 169 | | | |
| 6.1.3 | 3 | Get the unique status entries by executing the R code "*unique(checks_df$check_status)*" | Output is "OK", NA, "NOTE", and "ERROR" | | | |
| 6.1.4 | 4 | Count the number of packages in the file that have a status of "OK" by executing the R code "*length(checks_df$package[checks_df$check_status == "OK" & !is.na(checks_df$check_status)])*" | 146 | | | |
| 6.1.5 | 5 | Count the number of packages in the file that have a status of "NOTE" by executing the R code "*length(checks_df$package[checks_df$check_status == "NOTE" & !is.na(checks_df$check_status)])*" | 20 | | | |
| 6.1.6 | 6 | Count the number of packages in the file that have a status of "ERROR" by executing the R code "*length(checks_df$package[checks_df$check_status == "ERROR" & !is.na(checks_df$check_status)])*" | 1 | | | |
| 6.1.7 | 7 | Check the package that has a status of ERROR by executing the R code "*checks_df$package[checks_df$check_status == "ERROR" & !is.na(checks_df$check_status)]*" | "sessioninfo" | | | |
| 6.1.8 | 8 | Check the package that has a status of NA by executing the R code "*length(checks_df$package[is.na(checks_df$check_status)])*" | 2 | | | |
| 6.1.9 | 9 | Check the packages that have a status of NA by executing the R code "*checks_df$package[is.na(checks_df$check_status)]*" | "MIMOSA", "CytoTree" | | | |
| | **Tester Comments** - Note any discrepancies or annotations that occurred while executing the test. | | | | **Initial / Date** | |

**Figure 2. Performance Qualification Test Case for RPQ Output Verification**

Within this test case, the tester verifies that the output dataset can be successfully loaded, confirms the total number of packages, and evaluates package status summaries across sources. The expected result is that all packages return a 'NO ERROR' status, with any exceptions documented and evaluated through discrepancy handling procedures. These steps ensure that the dataset is complete, consistent, and reproducible, and that the RPQ process executes as expected within the validated environment.

During execution, any discrepancies identified are documented within the test case record. Where required, additional documentation may be completed to formally capture and evaluate such discrepancies. An example template for discrepancy documentation is provided in Figure 3.

# Discrepancy Report Form

**System Name:** R 4.4.2

**Short Name:** R

**System Version:** 4.4.2

**Validation Number:** VAL - 0089

**Documentation Version:** 1.0

| **Discrepancy number:** 1 | **Test case number:** 01 | **Step number:** 07 |
|---|---|---|
| **Description:** The test step 7 is expected to have no packages failed, but package “sessioninfo” is failed. | | |
| **Investigation:** | | |
| **Severity level:** ☒ Minor ☐ Major ☐ Critical ☐ Procedural | | |
| **Action plan:** | | |
| **Re-test required?** ☐ Yes ☒ No | | |
| **Re-test outcome:** ☐ Issue resolved ☐ Issue not resolved ☒ Not applicable (no re-test required) | | |

**Figure 3. Discrepancy Report Form**

This structured validation approach ensures traceability, reproducibility, and audit readiness of the RPQ process, aligning it with established PQ validation principles while adapting to the context of open-source software evaluation.

## INTERPRETATION OF RPQ OUTPUTS

The outputs generated through this process form the basis for downstream evaluation and decision-making. The RPQ framework produces two primary outputs: a package-check dataset and a risk assessment dataset. These outputs serve complementary but distinct roles within the overall evaluation process.

The package-check dataset provides a high-level qualification perspective based on repository checks, installation verification, and associated testing evidence. Within this dataset, package results are summarized into a standardized “check_status” variable, as shown in Table 1, which classifies packages as either having no error or requiring further review. Packages categorized as having no error are considered to have successfully passed the streamlined RPQ qualification checks. Packages with identified issues, including failed checks or notable repository outputs, are flagged for additional evaluation and potential follow-up through discrepancy documentation.

The risk assessment dataset serves a different purpose. Rather than functioning as a qualification decision tool, it includes package characteristics, such as maintenance, documentation, dependency structure, and adoption, using the *riskmetric* package by the R Validation Hub (R Validation Hub et al., 2026). This output is primarily intended to support study-level evaluation, particularly in downstream R usage planning for analyses involving primary endpoints or other critical functions. As such, the risk assessment dataset should be interpreted as contextual supporting evidence rather than as a pass/fail determination.

The RPQ evaluation framework is designed to support subsequent structured interpretation rather than enforce rigid acceptance criteria. While package-check results provide an initial indication of package reliability, these results are not interpreted in isolation. For example, a package with a minor repository note or non-critical warning may still be considered acceptable if supported by strong historical usage, adequate maintenance, or available mitigation strategies. Conversely, a package that passes high-level checks but exhibits limited testing, weak documentation, or complex dependency structures may warrant closer scrutiny, particularly if it is intended for use in critical analyses such as primary endpoint or interim analyses that could stop a trial early. The risk assessment is carried out by study leads in our R Usage Plan, a process separate from the RPQ process (Grant & Neradilek, 2026).

This approach reflects the understanding that package reliability cannot, in our team's opinion, be well reduced to a single metric or classification. Instead, RPQ outputs are evaluated in combination with contextual factors such as intended analytical use, role in primary or secondary endpoints, dependency complexity, and availability of alternative methods or risk mitigation strategies, and study leads' risk appetite. This interpretation layer ensures that the RPQ framework remains both flexible and aligned with real-world analytical and regulatory requirements.

To support traceability and auditability, the interpretation of RPQ outputs and any resulting decisions are documented within RPQ summary reports and, where applicable, associated sign-off records. These records provide a transparent link between package-level evidence, identified risks or observations, and final disposition. This structured interpretation approach ensures that RPQ outputs support consistent and defensible decision-making while avoiding over-reliance on purely automated classifications.

## HANDLING DEPENDENCIES AND PACKAGE VERSIONS

One of the primary challenges in validating R packages is managing dependencies. R operates within a dynamic ecosystem where packages depend on one another, often forming complex dependency chains. A single workflow may rely on dozens of interconnected packages maintained by different developers.

While base R and recommended packages follow controlled development practices, many external packages are updated frequently. This rapid evolution, while beneficial for innovation, complicates validation efforts. Fully validating all dependencies would require substantial and ongoing effort, making it impractical in most settings.

Figure 4 illustrates how a single package can introduce multiple layers of dependencies, highlighting the impracticality of full validation across all components.

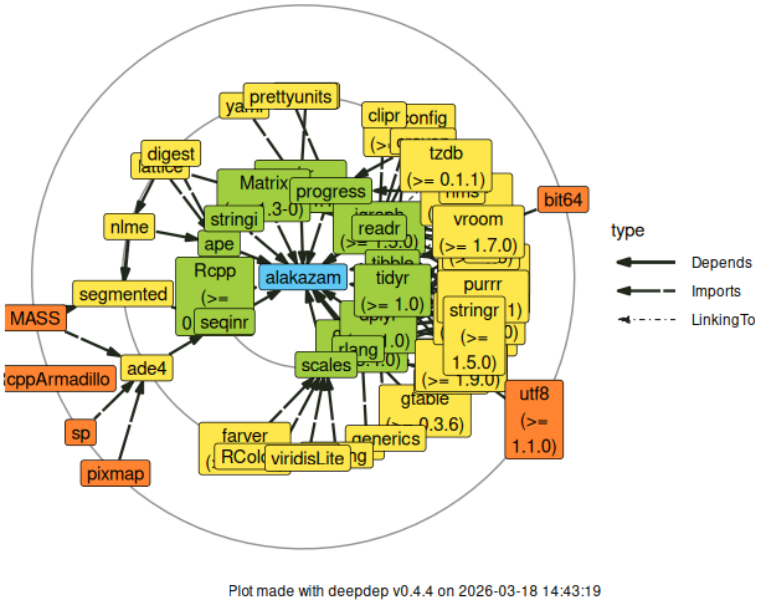


**Figure 4. Example of R Package Dependency Structure for package *alakazam*.**

As a result, our approach focuses on validating primary packages –those explicitly loaded and directly used in analysis workflows –while relying on successful CRAN and Bioconductor checks to provide a baseline level of quality assurance for dependencies, rather than performing full validation. To further support reproducibility, a curated repository of approximately two hundred commonly used packages was established, with versions fixed within the validated environment. This allows validation activities to focus on specific package versions, reducing variability and improving consistency.

However, version drift remains a potential risk. As newer package versions are installed, sometimes indirectly through dependency updates, subtle changes in defaults, bug fixes, or algorithms can impact workflows. Over time, these changes in package behavior, particularly across interconnected dependencies, may affect reproducibility, emphasizing the importance of thorough documentation, version control, and explicit dependency management. Study leads responsible for R usage are required to consider this drift and document how best to handle it in the reproducibility section of our R Usage Plan.

## SOURCE-SPECIFIC CONSIDERATIONS (CRAN, BIOCONDUCTOR, GITHUB)

R packages are distributed across multiple platforms, each with different validation expectations. CRAN packages must pass automated checks related to documentation, build integrity, and compatibility before release. Bioconductor packages follow a more structured process with coordinated releases and integrated testing frameworks.

In contrast, packages hosted on GitHub do not follow standardized validation processes. During our validation efforts, we encountered packages that were no longer available on Bioconductor but remained accessible on GitHub. These packages had a long history of internal use, providing confidence in their functionality.

To mitigate risk of GitHub packages, additional checks were performed, including execution of R CMD check and verification that unit tests were present and passed. Greater emphasis was placed on functional validation rather than strict metadata compliance, particularly for packages with established usage and documentation.

## INTEGRATION WITH STUDY-LEVEL RISK ASSESSMENT

RPQ is designed as a platform-level process that generates evidence rather than making final decisions. Study teams use RPQ outputs to determine package suitability within specific analytical contexts. For high-risk studies, additional validation may be performed through targeted approaches such as R Usage Plans.

This layered model ensures a clear separation of responsibilities. Programmers focus on generating consistent and reproducible evidence, while statistical teams evaluate risk and make study-specific decisions. This approach allows validation effort to be aligned with the level of risk.

In practice, the integration between RPQ outputs and study-level decision-making requires clear communication of both strengths and limitations of the RPQ framework. Study teams must understand that RPQ provides evidence of package behavior within the validated environment but does not guarantee correctness for all possible analytical use cases. This distinction is particularly important for packages used in primary endpoint analyses, where additional verification or risk mitigation may be required. By clearly defining these boundaries, the RPQ framework enables informed and risk-based decisions without overextending the scope of platform-level validation. In some cases, RPQ outputs may be further evaluated through structured approaches such as R Usage Plans, which apply targeted validation strategies for high-risk analytical components.

## LESSONS LEARNED FROM IMPLEMENTATION

The implementation of RPQ highlighted several important considerations. Automation significantly improved efficiency, particularly when leveraging repository checks and scripted workflows. However, variability in package characteristics, such as testing, documentation, and dependency structure, required flexibility in evaluation criteria.

During initial development, we anticipated that repository-based metadata would streamline validation by allowing us to directly retrieve CRAN check results for installed package versions. In practice, this approach proved insufficient. As validation timelines progressed, some packages had newer versions available on CRAN, leading to discrepancies between installed versions and those represented in the repository. Because CRAN primarily maintains check results for current package versions, historical check information for previously installed versions was not consistently accessible.

To address this limitation, we developed a custom workflow to perform local R CMD check evaluations on the installed package versions. This required reconstructing source packages from installed libraries and re-running checks within the validated environment. While more computationally intensive, this approach ensured that validation was performed against the exact package versions in use. This experience highlighted that repository-based validation alone may be insufficient in controlled environments and that direct validation of installed versions is critical for reproducibility.

Communication between programmers and study teams was also critical. Ensuring that RPQ outputs were clearly understood and appropriately interpreted required careful documentation. Maintaining a clear distinction between evidence generation and decision-making was also essential to avoid confusion regarding roles and responsibilities. In this framework, evidence generation refers to using RPQ to assess package risk and document the rationale, while decision-making refers to how study teams interpret that evidence to determine how a package may be used.

Another important lesson was the need to balance standardization with flexibility. While a consistent evaluation framework is essential for scalability, overly rigid criteria can fail to account for the diversity of the R ecosystem. Allowing for controlled flexibility in interpreting results enabled the team to accommodate packages with different characteristics while maintaining overall consistency. Additionally, early collaboration with IT teams proved critical in establishing a controlled package repository, which significantly reduced variability and simplified validation efforts.

## REPRODUCIBILITY

Reproducibility is supported through controlled package libraries and fixed versions within the validated environment. This ensures that analyses can be repeated under consistent conditions. At the study level,

reproducibility depends on proper documentation of package versions and awareness of changes over time. Programmers play a key role in maintaining this consistency.

### FUTURE DIRECTIONS

The current RPQ framework establishes a structured and reproducible approach for evaluating external R packages through a combination of automated checks and manual validation. While this approach ensures traceability and audit readiness, it also introduces operational overhead associated with execution of test cases and documentation of results.

Future enhancements to the RPQ framework will focus on further automation and standardization of the qualification process. One key objective is to evolve the current implementation toward a more fully integrated validation model, aligning with Installation Qualification (IQ) and Operational Qualification (OQ) principles. This includes standardizing core evaluation scripts so they can be reused across environments, particularly during server upgrades, with minimal modification.

In addition, opportunities exist to automate the generation of RPQ outputs into structured reports, such as PDF-based summaries of package check results and risk indicators. Such automation would reduce reliance on manual test case execution while preserving traceability of results.

Importantly, while automation may reduce manual effort, the need for oversight and review remains critical, particularly for packages used in primary analyses. Future iterations of the RPQ framework will therefore aim to balance efficiency with appropriate governance, ensuring that automated outputs continue to support reliable and defensible decision-making.

## CONCLUSION

The RPQ framework provides a practical and scalable approach to Performance Qualification of external R packages within a regulated environment. By leveraging existing repository-level evidence, incorporating structured validation procedures, and enabling context-driven interpretation, the approach addresses key challenges associated with open-source software validation. Importantly, the framework distinguishes between evidence generation and decision-making, allowing validation effort to be aligned with study-specific risk. This separation ensures both scalability at the platform level and flexibility at the study level. As the use of R continues to grow in clinical research, approaches such as RPQ offer a sustainable path forward for integrating open-source tools into validated analytical workflows. Future enhancements aimed at increasing automation and standardization are expected to further improve efficiency while maintaining compliance with validation expectations.

## REFERENCES


Gillespie, C. S. (2025). *Can I trust that package?* [Conference presentation]. posit::conf. https://www.youtube.com/watch?v=g6QU16jOt0g

International Council for Harmonisation of Technical Requirements for Pharmaceuticals for Human Use. (1998). *Good statistical principles for clinical trials (ICH E9).* https://database.ich.org/sites/default/files/E9_Guideline.pdf

International Council for Harmonisation of Technical Requirements for Pharmaceuticals for Human Use. (2025). *Guideline for good clinical practice (ICH E6(R3)).* https://database.ich.org/sites/default/files/ICH_E6%28R3%29_Step4_FinalGuideline_2025_0106.pdf

Novikova, M. (2023). *deepdep: Visualise and explore R package dependencies* (Version 0.2.0) [R package]. https://cran.r-project.org/package=deepdep

R Foundation for Statistical Computing. (2021). *R: Regulatory compliance and validation issues: A guidance document for the use of R in regulated clinical trial environments.* https://www.r-project.org/doc/R-FDA.pdf

R Validation Hub. (2023). *valtools: Tools for validation frameworks* (Version 0.3.0) [R package]. https://cran.r-project.org/package=valtools

R Validation Hub, Kelkhoff, D., Gotti, M., Miller, E., K, K., Zhang, Y., Milliman, E., & Manitz, J. (2026). *riskmetric: Risk metrics for evaluating R packages* (Version 0.2.6) [R package]. https://cran.r-project.org/package=riskmetric

R Validation Hub, Nicholls, A., Bargo, P. R., & Sims, J. (2020). *A risk-based approach for assessing R package accuracy within a validated infrastructure.* https://pharmar.org/white-paper/

Grant, S., & Neradilek, B. M. (2026). *Handle with care: A study-specific assessment of R package risk*. In Proceedings of PHUSE US Connect 2026. PHUSE.

Vendettuoli, M., Zhang, E., & Zou, R. (2023). "*Strategies for Code Validation at Statistical Center for HIV/AIDS Research and Prevention (SCHARP)*". Retrieved from PharmaSUG 2023 Conference. https://www.lexjansen.com/pharmasug/2023/AP/PharmaSUG-2023-AP-130.pdf

## ACKNOWLEDGMENTS

The authors would like to thank the SCHARP R PQ team and paper reviewers Shannon Grant, Moni Neradilek, Dave Slager, Xuehan (Emily) Zhang, and Amy Harper.

## RECOMMENDED READING

- *R package {validate} website: https://cran.r-project.org/web/packages/validate/vignettes/cookbook.html*
- *R installation and administration: https://cran.r-project.org/doc/manuals/r-release/R-admin.html*

## CONTACT INFORMATION

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

Radhika Etikala
Statistical Center for HIV/AIDS Research & Prevention (SCHARP) at Fred Hutchinson Cancer Center
retikala@fredhutch.org

Valeria Duran
Statistical Center for HIV/AIDS Research & Prevention (SCHARP) at Fred Hutchinson Cancer Center
vduran@fredhutch.org

# APPENDIX

## APPENDIX A

```
## PROGRAM: CHECK_SCRIPT.R
## PURPOSE: Runs package checks on installed packages based on package category

# Load Libraries ----------------------------------------------------------

library(tidyverse)
library(janitor)
library(rcmdcheck)

source("rcmd_functions.R")

# Load Package List -------------------------------------------------------

pkgs <- read_excel("Package_List.xlsx", col_types = "text") %>%
  mutate(installed_version = sapply(package, function(pkg) {
    if (pkg %in% rownames(installed.packages())) as.character(packageVersion(pkg)) else NA
  }))

# BASE Package Checks -----------------------------------------------------

# Base packages are bundled with R. Verify successful installation via system commands.
base_pkgs <- pkgs %>% filter(grepl("BASE", category)) %>% pull(package)

base_pkgs <- data.frame(
  package = base_pkgs,
  tools_status = ifelse(<system_installation_check_condition>, "OK", "ERROR")
)

# CRAN Package Checks -----------------------------------------------------

# Pull CRAN check results and match to installed package versions.
CRAN_pkgs <- tools::CRAN_check_details(flavors = "r-release-linux-x86_64") %>%
  clean_names() %>%
  filter(package %in% pkgs$package) %>%
  group_by(package, version, status) %>%
  summarise(CRAN_checks = paste(check, collapse = "; "), .groups = "drop")

# Bioconductor Package Checks ---------------------------------------------

# Identify Bioconductor packages and pull build reports.
bioc_pkgs <- setdiff(pkgs[!grepl("BASE", pkgs$category), ]$package, CRAN_pkgs$package)

BioC_pkgs <- BiocPkgTools::biocBuildReport() %>%
  filter(pkg %in% bioc_pkgs, stage == "install") %>%
  select(package = pkg, bioc_version, bioc_result = result)

# GitHub Package Checks ---------------------------------------------------

# For GitHub-sourced packages, clone and run R CMD check on each repository.
repo_urls <- c("https://github.com/sample-org/package-one.git")

gh_results <- lapply(repo_urls, function(url) {
  temp_repo <- tempfile(); dir.create(temp_repo, recursive = TRUE)
  git2r::clone(url = url, local_path = temp_repo)
  check_result <- devtools::check(temp_repo, vignettes = FALSE, error_on = "never")
  unlink(temp_repo, recursive = TRUE)
  list(package = check_result$package, version = check_result$version,
```

```
       status = check_result$status)
}) %>%
    do.call(rbind, .) %>%
    as.data.frame()

# Manual R CMD Check ------------------------------------------------------

# For packages where installed version does not match CRAN, download source
# tarballs from the internal repository and run R CMD check manually.
local_pkg_files <- <download_packages_from_internal_repo>

rcmd_results <- lapply(local_pkg_files, function(pkg_tar) {
  check_result <- rcmdcheck::rcmdcheck(pkg_tar, error_on = "never")
  list(package = check_result$package, version = check_result$version,
       status = check_result$status, platform = check_result$platform)
}) %>%
    do.call(rbind, .) %>%
    as.data.frame()

# Combine and Save Results ------------------------------------------------

pkg_all_checks <- pkgs %>%
  left_join(base_pkgs) %>%
  left_join(CRAN_pkgs) %>%
  left_join(BioC_pkgs) %>%
  left_join(gh_results) %>%
  left_join(rcmd_results) %>%
  mutate(check_status = case_when(
    status %in% c("OK", "NOTE") ~ "NO ERROR",
    TRUE ~ "ERROR")) %>%
  select(category, package, installed_version, check_status)

readr::write_csv(pkg_all_checks, paste0("Package_Checks_", Sys.Date(), ".csv"))
```

## APPENDIX B

```
## PROGRAM: Risk_Assessment.R
## PURPOSE: Runs riskmetric assessments on installed packages to evaluate maintenance, usage,
and quality risk metrics

# Load Libraries ------------------------------------------------------------

library(tidyverse)
library(readxl)
library(janitor)
library(riskmetric)

# Load Package List ---------------------------------------------------------

pkgs <- read_excel("Package_List.xlsx") %>%
  filter(!grepl("BASE", category))

# riskmetric Assessment -----------------------------------------------------

# Assess each package and compute two derived metrics:
# - exported_prop: proportion of exported objects with documentation
# - rev_deps: count of reverse dependencies
risk_assess <- riskmetric::pkg_ref(pkgs$package) %>%
  riskmetric::pkg_assess() %>%
  mutate(across(where(is.list), ~ map_chr(., ~ paste(.x, collapse = " ;;; "))))

# Save Output ---------------------------------------------------------------

readr::write_csv(risk_assess,
                 paste0("RiskAssessment_Output_", Sys.Date(), ".csv"))
```