

Automated R package validation

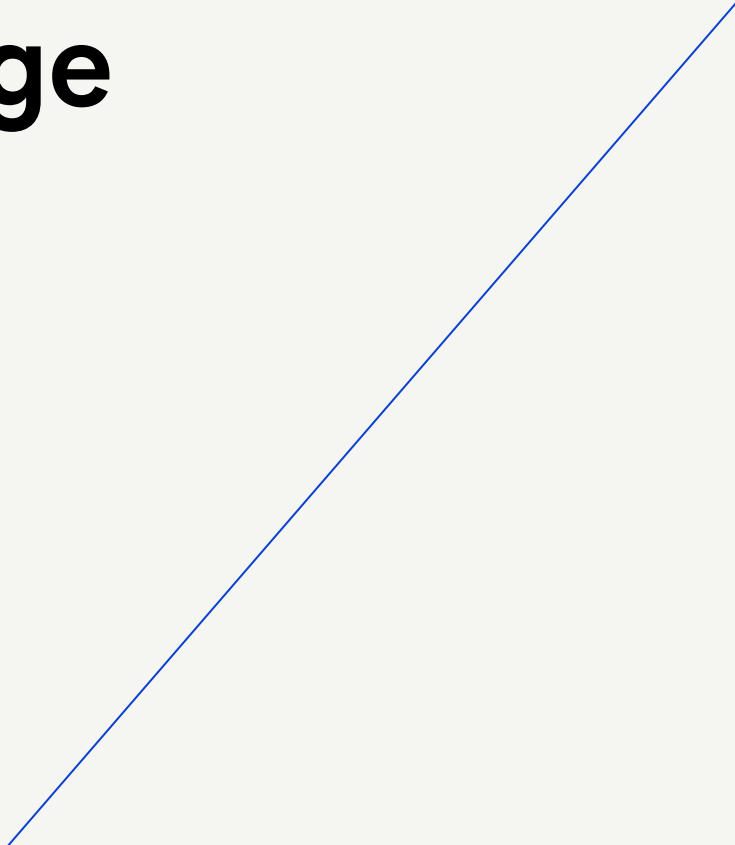
Roche R Package Validation Team

Coline Zeballos

Doug Kelkhoff

Szymon Maksymiuk

Lorenzo Braschi

A thin blue diagonal line extending from the bottom right corner towards the top right edge of the slide.

Design Priorities

Leverage best practices for R package development

Prefer automation of gating criteria

Avoid special cases for classes of packages

Minimize package scope

- For example, packages which interface with peripheral systems (database, API) are responsible only for the R language interface. The system that they talk to is managed by a separate system validation.

Balance risk mitigation against automation

- Some risk is best mitigated by end users, who have responsibility to escalate observed issues
- Some risk is best mitigated by system administration, particularly around data access and security

Human-in-the-middle approach to reconcile automated check gaps

Target ephemeral, reproducible systems and composable packages (containers + package repo)

Transparent review process to encourage in-house development & iteration

Process overview

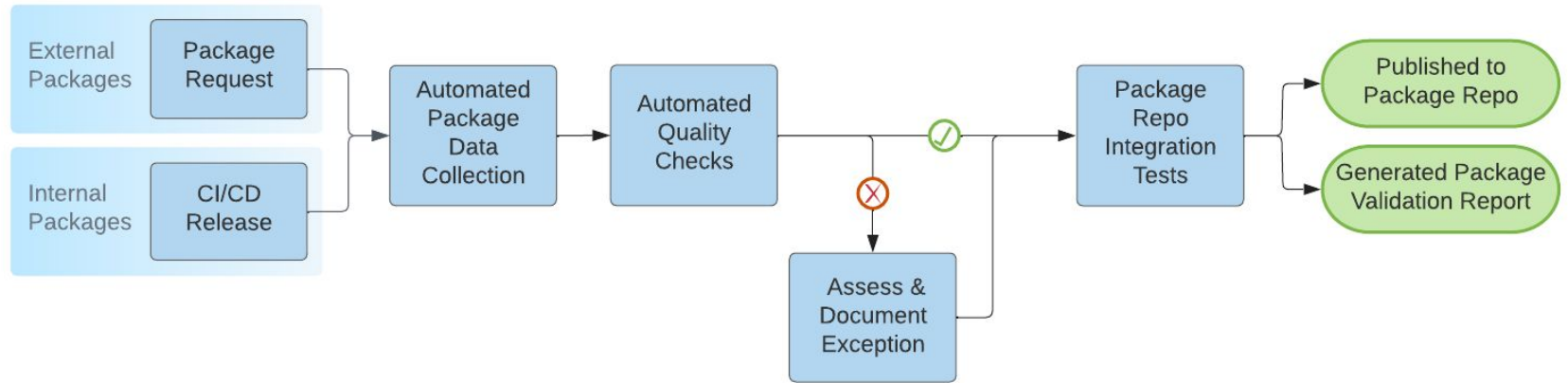


Image 1: Graphical overview of the submission process

👁️ human-in-the-middle component

Step 1: Submission

- Packages start with a submission
- The validation process has only two prerequisites:
 - Dependencies must to be validated first (unmet dependencies will alert support team)
 - The source of the package has to be reproducible either with git hash or a tar.gz checksum
- Submissions trigger CI/CD workflows, managed through merge requests
inspired by [Julia language general registry](#)
 - Requester/Developer can monitor CI/CD pipelines
 - Review & discussion can happen in shared, (internally) public location
- Submissions are overseen by the support team who are ready to help at the any stage

Step 2: Package data collection

Automatically perform basic package tests, and record output to produce a report, or provide for review if the package has any quality gaps

- We automatically record
 - Basic package metadata (package name, version, authors, description, documentation)
 - **R CMD check** results
 - Test execution & coverage
 - Traceability matrix (mapping tests to R package documentation, using [covtracer](#)¹⁾)
 - Test system information
- How they are collected?
 - Execution in controlled reproducible analytic environments (using containers)
 - Strict dependencies management, leveraging a validated R package repository

[1]  **covtracer open test-docs mapping tools**
github.com/genentech/covtracer

Step 3: Package quality checks

Theme	Description
Source Control	Reproducible source code ensured with git hash or a tar.gz checksum
Documentation	The package has clear ownership, documented as Authors & Maintainer fields in DESCRIPTION All exported objects are documented. These comprise our software requirements
R CMD check	The package passes R CMD check without ERRORS R CMD check WARNINGS or NOTEs are justified in cran-comments.md
Testing	All evaluated unit tests succeed Code coverage is at or above 80%
Traceability	All exported functionality is evaluated by at least one unit test

Table 1: Summary of metadata types collected to generate a comprehensive report.

Step 4: Addressing gaps

Packages may not meet our automatically enforced standards.

If packages have gaps, they are often have one of

- Insufficient testing / coverage
- R CMD check issues which require approval
- Unique failure mode specific to internal systems

There are two possible paths for addressing the packages' gaps

- cran-comments.md file
- Remediation procedure

Step 4: Addressing gaps using cran-comments

- Leverage standard communication channel for package authors to address known R CMD check issues
- Enforced for internal packages, preferred for external packages
- Every check issue (either warning or note) has to be addressed in this file
- The purpose (allow developer communicate that they are aware of the issues reported by check and took their time to investigate them - note in this file is a result of this investigation)

cran-comments.md

from {rbmi¹}

R CMD check results

There were no ERRORS or WARNINGS.

There were 2 NOTES:

```
> checking installed package size ... NOTE
  installed size is 57.4Mb
  sub-directories of 1Mb or more:
    libs 56.0Mb
```

```
> checking for GNU extensions in Makefiles ... NOTE
  GNU make is a SystemRequirements.
```

Both of the above notes are a consequence of using rstan in the package following the usage steps as described by the stan developers [here](<https://cran.r-project.org/web/packages/rstantools/vignettes/minimal-rstan-package.html>). Our understanding from the [developers](<https://discourse.mc-stan.org/t/using-rstan-in-an-r-package-generates-r-cmd-check-notes/26628>) is that they are acceptable to ignore.

[1] **rbmi** Reference Based Multiple Imputation
github.com/insightsengineering/rbmi

Step 4: Addressing gaps documenting remediation

Any gaps represent a lapse in expected package behaviors or quality. We document observed gaps, and itemize why they are permissible

- Many R packages (especially older R packages), don't hold up to modern development expectations. This is often an indicator of the time that they were developed, not the quality of the package
- Every issue reported by set up checks (either R CMD check, tests or any of our custom edge cases checks) can be a subject to remediation **except** for check ERRORS
- The view and opinion of experienced R developers & members of the support team is major part of this process

Step 4: Addressing gaps documenting remediation

Gaps are evaluated on a case-by-case basis.

Rationale to justify package gaps often falls in four major themes:

Consideration	Rationale
Complex Systems	When testing is minimal presumably because of system complexity, other indicators of quality may be more informative
Decision Impact	Packages that are unlikely to impact critical decision making requires less stringency
Adoption & Longevity	Wide adoption or historical stability may be good indicator of quality
Developer Trust	Packages may be developed by established members of the R community, trusted institutions or have corresponding peer-reviewed publications

Table 2: Common remediation considerations & strategies

If the package fails? - Reiterate!

- The validation of internal packages is similar to the iterative review process
- Gaps identified in internal packages are underlined and presented to maintainers, alongside suggestions to improve them
- Once maintainers address those gaps, the package is re-submitted and the process repeats
- The process ensures not only higher quality of packages produced within the company but also improves collective developer best practices
- External packages that fail to meet our expectations and cannot be remediated are rejected due to the limitations in affecting changes

What comes after validation?

R package repository

```
> options(repos = "repo/validated/latest")
> install.packages("package")
```

- Leverage CRAN-style rolling package release (with historical snapshots for reproducibility)
- Users are free to use canonical R package management tools
- In-house developers can easily validate & release updates

Report generation

PDF

Validation Report
package v1.2.3

- Documents findings, including all automated checks and support team comments
- Uploaded to document management service

Summary

- Automate a strict set of rules, highlighting gaps to support team experts
- Validation applies only to the specific version of the packages run with the specific system (eg, Architecture, OS, R version)
- We believe the process we've developed
 - ✓ Assures high quality of packages for regulatory analysis
 - ✓ Reinforces internal developers familiarity with best R programming practices
 - ✓ Reduces risk by making our analysts and developers more capable contributors to the R ecosystem more broadly