PharmaSUG China 2025 - Paper CC-141

Lifecycle Management of R Functions

Longfei LI, Sanofi

ABSTRACT

As R functions evolve, maintaining backward compatibility while introducing improvements becomes a critical challenge for developers. lifecycle offers a robust and standardized approach to managing the deprecation and evolution of functions. This article explores two common scenarios in package development — replacing old functions with new ones and handling changes to function arguments, which will help R developers implement effective lifecycle management strategies that enhance both maintainability and user experience.

INTRODUCTION

In the domain of data analysis within the pharmaceutical industry, the SAS programming language has historically held a dominant position. Many experienced SAS programmers have encountered the following scenario: a well-designed general-purpose SAS macro is developed and subsequently adopted widely within the team. Initially, the macro may be simple, stable, and reliable, gradually becoming an integral component of critical projects. However, as analytical requirements evolve, data complexity increases, and regulatory standards become more stringent, such macros often require bug fixes, additional parameters, or even substantial reimplementation.

These changes introduce several challenges:

1. Lack of mechanisms to ensure that all users are promptly informed of updates;
2. New users may remain unaware of the latest version of the macro;
3. Legacy code continues to invoke outdated versions, resulting in frequent runtime errors;
4. Each update typically triggers numerous inquiries and on-site debugging efforts;
5. More critically, multiple divergent versions of the same macro may coexist across different projects.
6. Such issues not only reduce development efficiency but also pose potential compliance risks—particularly concerning in regulated environments such as clinical trials.

As the industry transitions toward using the R programming language for statistical analysis, developers are no longer constrained by the limitations inherent in the SAS macro system. However, without systematic management and maintenance of functions, similar problems can easily re-emerge. Therefore, it is essential to adopt a structured, traceable, and sustainable approach to managing the full lifecycle of R functions.

This paper introduces how the lifecycle package can be effectively employed to address two common development scenarios: the deprecation and replacement of outdated functions, and the handling of changes to function arguments.

When developing with R, the use of packages is fundamental. An R package is a cohesive collection of functions designed to achieve a specific analytical objective, offering advantages in modularity, reusability, and distribution. More importantly, R packages provide the necessary infrastructure for robust function lifecycle management. Without this structure, tracking the evolution and versioning of individual functions becomes significantly more challenging.

While a detailed introduction to R packages is beyond the scope of this discussion, it is crucial to emphasize that the two application scenarios—function replacement and argument modification—are most meaningful and practically effective when implemented within the framework of an R package. Only through such a structured approach can development workflows benefit from improved maintainability, traceability, and long-term sustainability.

ARGUMENT CHANGES ACROSS FUNCTION VERSIONS

In the development of R packages, it is common for developers to adjust function arguments as part of feature iteration or design optimization—such as deprecating certain old arguments and introducing new ones in their place. To avoid abruptly disrupting user workflows through direct removal or alteration of existing parameters, the lifecycle package provides a standardized and gentle mechanism for achieving this goal, with one of its key functions being lifecycle::deprecate_warn(). This function enables developers to issue structured warnings when deprecated parameters are used, thereby informing users about the availability of a replacement while maintaining backward compatibility.

The following is a typical example of using the lifecycle package to manage the deprecation of a function argument, illustrating the transition from the parameter path to the recommended parameter file:

```
test <- function(file, path = lifecycle::deprecated()){
  if (lifecycle::is_present(path)) {
    lifecycle::deprecate_warn(when = "1.4.0",
                              what = "test(path)",
                              with = "test(file)")
    file <- path
  }
}
```

```
Warning message:
The `path` argument of `test()` is deprecated as of rtoy 0.1.0.
i Please use the `file` argument instead.
This warning is displayed once every 8 hours.
Call `lifecycle::last_lifecycle_warnings()` to see where this warning was generated.
```

If an argument (e.g., path) is to be completely phased out and users are to be compelled to abandon its use immediately, the lifecycle::deprecate_stop() function can be employed to raise an error and halt execution upon detecting its usage. This ensures that the outdated interface is no longer used and enforces immediate migration to the updated alternative.

```
Error:
! The `path` argument of `test()` was deprecated in rtoy 0.1.0
  and is now defunct.
i Please use the `file` argument instead.
Run `rlang::last_trace()` to see where the error occurred.
```

FUNCTION REPLACEMENT IN R PACKAGES

In some cases, in addition to updating or replacing function arguments, developers may choose to fully deprecate an existing function and replace it with a completely new one. This situation is similar to the parameter deprecation discussed earlier, but the scope of change is broader—encompassing not only modifications to the interface structure but also significant evolution in the function's semantics or implementation.

To support such transitions, the lifecycle package provides corresponding mechanisms. The recommended approach is to use lifecycle::deprecate_warn(), which issues a structured warning when the old function is called, advising users to switch to the new replacement. This method maintains backward compatibility while clearly communicating the direction of the API evolution, thereby facilitating a gradual migration path for users.

However, in practical data processing workflows, function replacement is often necessitated by the presence of critical bugs, logical errors, or security concerns in the original implementation. In such cases, continued use of the deprecated function may introduce unacceptable risks. Therefore, we advocate for a stricter deprecation strategy—such as using lifecycle::deprecate_stop()—which not only informs existing users of the rationale behind the deprecation, but also enforces immediate migration to the corrected replacement function. This enhances code safety, stability, and long-term maintainability.

```
test <- function(){

  lifecycle::deprecate_stop(
    when = cli::col_cyan("1.0.0"),
    what = I("test()"),
    with = "test1()",
    details = c("*" = "`test()` has a fundamental logical flaw as well as
significant functional limitations")
  )

}
```

```
Error:
! test() was deprecated in rtoy 1.0.0 and is now defunct.
i Please use `test1()` instead.
• `test()` has a fundamental logical flaw as well as significant
  functional limitations
Run `rlang::last_trace()` to see where the error occurred.
```

It is important to note that lifecycle::deprecate_stop() should be used within the definition of the original (deprecated) function to ensure proper enforcement of the migration.

OTHER METHOD

Setting aside the design philosophy and implementation details of the lifecycle package, its two most common use cases—deprecation of function arguments and deprecation of entire functions—share a core objective: to communicate interface changes to users in a structured manner, and to guide them toward adopting new usage patterns. This ultimately helps reduce communication overhead and long-term maintenance costs.

In practice, developers can achieve similar goals using alternative approaches. For instance, base R provides built-in functions such as message() and warning() for communicating with users, while

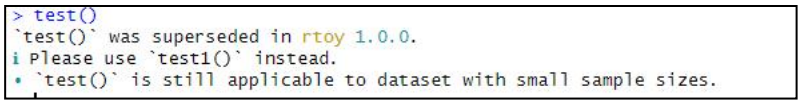
packages like cli offer more advanced capabilities for generating well-formatted, semantically meaningful output.

The following example demonstrates how the cli package can be used to implement a lightweight deprecation mechanism:

```
test <- function(){

  cli::cli_inform(c(
    "`test()` was superseded in {cli::col_yellow('rtoy')}
{cli::col_cyan('1.0.0')}.",
    "i" = "Please use `test1()` instead.",
    "*" = "`test()` is still applicable to dataset with small sample sizes."
  ))

}
```



Although this approach lacks the version-aware semantics and standardized infrastructure provided by lifecycle, it can still serve as a practical and readable alternative in scenarios where a lighter-weight solution is preferred.

CONCLUSION

Function lifecycle management is a meaningful and essential concept in the development of R packages. This article has introduced two commonly used strategies in our workflow — mechanisms for managing deprecations in both function arguments and entire functions. These approaches are straightforward to implement yet highly effective in improving code maintainability and development efficiency.

By consciously applying lifecycle management practices, developers can significantly reduce communication overhead with users and enhance the overall robustness and clarity of their packages. Therefore, it is strongly recommended to incorporate such lifecycle-aware design patterns into R package development, fostering the creation of more sustainable, user-friendly, and well-documented APIs.

REFERENCES

Henry L, Wickham H (2023). lifecycle: Manage the Life Cycle of your Package Functions. R package version 1.0.4, https://github.com/r-lib/lifecycle, https://lifecycle.r-lib.org/

Csárdi G (2025). cli: Helpers for Developing Command Line Interfaces. R package version 3.6.5, https://cli.r-lib.org.

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to Lionel Henry and Hadley Wickham for their development of the lifecycle package, as well as to Gábor Csárdi for creating the cli package. I also acknowledge the broader R community for their outstanding contributions to the development of the R language, which have made R programming increasingly accessible and efficient. As users of these valuable tools, I am honored to promote the concept of function lifecycle in R within the pharmaSUG community.

CONTACT INFORMATION

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

<Name: Longfei LI>
<E-mail: Longfei2.Li@Sanofi.com>