PharmaSUG 2025 - Paper OS-293

Achieving Reliable Data Verification with R: Proven Tools, Best Practices, and Innovative Workflows

Valeria Duran and Xuehan (Emily) Zhang, SCHARP at Fred Hutchinson Cancer Center, Seattle, Washington

ABSTRACT

In the pharmaceutical industry, ensuring the accuracy and reliability of data is critical, particularly when the output data can influence the analysis result. For SAS® programmers transitioning to R or current R programmers exploring the language's capabilities in verification, identifying specific tools and best practices can be challenging. This paper presents the tools we can use for effective verification. We will focus on three levels of data verification: complete independent verification (double programming), targeted independent verification, and peer review. We discuss how these levels align with varying risk levels associated with data complexity and the criticality of study endpoints. We will highlight our verification practices in R, showcasing how R functions from packages such as {testthat} and {purrr} can act as alternatives to SAS's *PROC COMPARE*. Additionally, we will explore a proposed peer review verification process, offering possible approaches to code and data review procedures. This paper aims to inform both R and SAS programmers about the nuances of data verification and provide practical guidance for integrating R effectively into their data verification workflows.

INTRODUCTION

Verification is a critical component of ensuring the accuracy, reliability, and reproducibility of statistical programming results. The concepts of verification in statistical programming are language agnostic, but their application may be approached differently depending on the programming language and environment. Double programming and comparison using the SAS COMPARE Procedure is considered to be the gold standard for verification in most of the pharmaceutical industry. Open source software options, such as R, are now challenging that long held belief. In R verification is facilitated by various tools and packages, which can enhance efficiency and consistency in the validation workflow. As an open-source language, R benefits from a large community of contributors, resulting in many freely available packages. These tools enable programmers to customize the verification process to their specific needs, whether through packages that assert data expectations, enable interactivity, or streamline comparisons. Compared to the more structured and fixed approach of SAS, R provides a more flexible and user-friendly environment to programmers. In this paper, we will discuss verification approaches including independent programming, targeted independent programming, data review, and code review. We will also provide examples and corresponding code using R packages that can be applied for verification tasks.

R TOOLS

Several R packages are particularly useful for supporting verification workflows, whether through data comparison, assertion checks, or summarization. Table 1 lists the R packages covered and their intended uses.

R Package	Useful For	Verification Level/Step
assertr	Data verification in programming	Self/Inline Verification
diffdf	Comparing datasets	Independent/Targeted Programming
purrr	Comparing datasets	Independent/Targeted Programming

testthat	Comparing datasets	Independent/Targeted Programming, Self/Inline, Data Review
arsenal	Comparing datasets	Independent/Targeted Programming
skimr	Data summary	Data Review

Table 1. R packages and recommended use during verification.

Below is a more detailed overview of the packages listed in Table 1:

- {assertr}: Useful for asserting expectations in datasets, making it easier to catch unexpected values or structures during data processing (Barrett, 2023).
- {diffdf}: Useful for comparing datasets and identifying differences in content or structure (Gower-Page & Martin, 2024).
- {purrr}: Useful for verifying multiple datasets or variables due to its efficiency in iterating over datasets or columns (Wickham & Henry, 2025).
- {testthat}: Useful for writing test cases to verify output at different stages of data processing, as well as testing that outputs are equal (Wickham, 2011).
- {arsenal}: Useful for comparing datasets and providing detailed summaries of the differences (Heinzen et al., 2021).
- {skimr}: Useful for providing summaries of datasets, helping to flag issues such as missing values and variable completion rates (Waring et al., 2025).

All packages mentioned in this paper are freely available through CRAN (the Comprehensive R Archive Network), which can be accessed at https://cran.r-project.org/ and installed in R using the *install.packages()* function. In addition to packages that support verification, there are also packages useful for processing data. Programmers can leverage packages such as those included in the {tidyverse} (Wickham et al., 2019). The {tidyverse} provides a cohesive and consistent framework for data manipulation, visualization, and analysis, making it an essential tool for efficient data processing. The R community supports this collection of packages by contributing, maintaining, and reviewing code, helping ensure that the {tidyverse} remains compliant with CRAN's package policies, passes regular checks, and, as such, may be considered a trusted resource (PharmaR, 2023). As with any third-party tools used in regulated environments, organizations may want to perform internal validation of these packages to ensure they meet internal standards before use.

VERIFICATION RISK ASSESSMENT

A risk-based framework can be utilized to classify levels of programming risk into three categories: low, medium, and high risk. The risk level could be determined, as shown in this example, based on the following criteria: the impact on the study and endpoint, as well as the complexity of the programming task. This risk-based approach enables the efficient allocation of verification resources, ensuring that higher-risk analyses receive more rigorous attention. Organizations should define their risk categories and criteria based on their own operational needs.

Below is an example of how we could categorize our risk assessment matrix with the lowest level of verification required.

OVERALL RISK		Programming/Complexity Risk		
		Low	Medium	High
Study, Safety, and Endpoint Risk	Low	Data review	Code review + Data review	Independent verification + Data review
	Medium	Code review + Data review	Code review + Data review	Independent verification + Data review
	High	Independent verification + Data review	Independent verification + Data review	Independent verification + Data review

Table 2. Minimum verification risk assessment matrix.

We will now discuss types of verification in more detail, along with the tools that can be used to implement them.

INDEPENDENT PROGRAMMING WITH {ASSERTR}, {DIFFDF}, {PURRR}, AND {TESTTHAT}

Independent programming is conducted by two programmers who independently create an analysis dataset or data display/TLF based on the same set of data specifications or table mocks. The output from both programmers is then compared for agreement. The Production Programmer generates analysis results based on the mock TLFs and data specifications provided or approved by the Lead Biostatistician. Verification can begin after or concurrently with the production programming depending on the specific tasks. Once both programmers have completed their outputs, they compare results and collaborate with the Lead Statistician to address any mismatches.

For both SAS and R, if any validated R packages or institutional SAS macros are used in production or verification programming, a completely different or independently programmed set of code should be used by the other programmer in order to ensure independent output generation. If the verification is conducted in R, the verification program should avoid the use of R packages that have been used by the Production Programmer, perhaps with the exception of those required to load and output datasets. This prevents errors related to the incorrect application of macros or existing code. The Verification Programmer may work more exclusively in base R to accomplish this. Base R are the core functions and tools provided with R by default and do not rely on external packages. Since R packages are developed by various programmers from the R community, there are unforeseen risks of incorrect results when using them. These risks depend on package maintenance and usage. To mitigate them, verification is preferably done using base R.

COMPARE SAS AND R OUTPUT

Both SAS and R can provide information about mismatches, including variable names and mismatched values between variables. SAS users can use PROC COMPARE to compare two SAS datasets and generate the required output file, as shown in Figure 1. Programming teams and study leadership should decide if all variable values and types must match exactly for the verification to be considered complete or if any mismatches are acceptable. This may also be driven by the overall risk assessment for the programming task. Any discrepancies will also be displayed in the SAS log, as shown in Output 1.

```
proc compare base = production_data compare = verifier_data; run;
```

Figure 1. Using PROC Compare in SAS.

```
                    The COMPARE Procedure
      Comparison of WORK.PRODUCTION_DATA with WORK.VERIFIER_DATA
                        (Method=EXACT)

             Value Comparison Results for Variables

__________________________________________________________
           ||  Base Value           Compare Value
       Obs ||  arm                   arm
 ________  ||  ___________________+  ___________________+
           ||
         1 ||  Treatment:T8          Treatment:T7
         2 ||  Treatment:T8          Treatment:T7
         4 ||  Treatment:T8          Treatment:T7
         5 ||  Treatment:T8          Treatment:T7
        32 ||  Treatment:T8          Treatment:T7
        36 ||  Treatment:T8          Treatment:T7
        78 ||  Treatment:T8          Treatment:T7
        79 ||  Treatment:T8          Treatment:T7
        81 ||  Treatment:T8          Treatment:T7
        84 ||  Treatment:T8          Treatment:T7
__________________________________________________________
```

Output 1. Comparing datasets in SAS using PROC COMPARE.

R packages can also be used to compare datasets. R packages {testthat}, {arsenal}, and {purrr} are commonly used in the comparison scripts to compare datasets, as shown in Figure 2.

```
# Define key fields for dataset comparison
.keyfields <- c("studyid", "usubjid", "subjid")

# Wraps expect_equal with error handling using purrr::safely
quiet_test <- purrr::safely(testthat::expect_equal)

# Perform dataset comparison using arsenal::comparedf
compare_diff <- arsenal::comparedf(production_data, verifier_data, by =
.keyfields, tol.num.val = 0.0001)

# Extract mismatch details
mismatch_data <- arsenal::diffs(compare_diff)
```

Figure 2. Using {testthat}, {arsenal} and {purrr} packages in R.

In the code above, the *safely()* function from {purrr} is used to wrap the *expect_equal()* function from the {testthat} package, preventing it from generating an error. Instead, it captures the error and returns it as part of a structured output. This approach is useful for handling potential failures in a controlled manner. The *expect_equal()* function from the {testthat} package is used for checking if two objects are equal.

Lastly, the *diff()* function from the {arsenal} package extracts details of the mismatches from the comparison results generated from arsenal's *comparedf()* function.

If the two datasets match, based on the code in Figure 2, object *compare_diff* is expected to have a data.frame as the output. If the datasets do not match, the mismatches will be stored in *mismatch_data* as shown in Output 2.

```
> # Extract mismatch details
> arsenal::diffs(compare_diff)
   var.x var.y studyid usubjid    subjid     values.x     values.y row.x row.y
1    arm   arm Study A    1015 601881432 Treatment:T8 Treatment:T7    36    36
2    arm   arm Study A    1024 250148677 Treatment:T8 Treatment:T7     5     5
3    arm   arm Study A    1035 824826705 Treatment:T8 Treatment:T7     2     2
4    arm   arm Study A    1072 458864965 Treatment:T8 Treatment:T7    79    79
5    arm   arm Study A    1080 804004663 Treatment:T8 Treatment:T7    81    81
6    arm   arm Study A    1082 159361359 Treatment:T8 Treatment:T7    84    84
7    arm   arm Study A    1121 698873880 Treatment:T8 Treatment:T7    78    78
8    arm   arm Study A    1150 356729620 Treatment:T8 Treatment:T7     1     1
9    arm   arm Study A    1157 418465525 Treatment:T8 Treatment:T7     4     4
10   arm   arm Study A    1158 648167005 Treatment:T8 Treatment:T7    32    32
```

Output 2. Comparing datasets in R using {arsenal}.

In addition to {arsenal}, another R package that can be used to compare datasets is {diffdf}. Similar to {arsenal}, {diffdf} also provides comparison results and corresponding mismatches, as shown in Figure 3.

```
# Define key fields for dataset comparison

.keyfields <- c("studyid", "usubjid", "subjid")

# Perform dataset comparison using diffdf::diffdf
diff <- diffdf(production_data, verifier_data, keys = .keyfields,
suppress_warnings = T)
```

Figure 3. Using {diffdf} packages in R.

The function *diffdf()* in {diffdf} can extract the details of mismatches from the comparison results, as shown in Output 3.

```
> diff
Differences found between the objects!

Summary of BASE and COMPARE
  ================================================================
    PROPERTY              BASE                        COMP
  ----------------------------------------------------------------
      Name           production_data             verifier_data
     Class      "tbl_df, tbl, data.frame"  "tbl_df, tbl, data.frame"
    Rows(#)                114                         114
   Columns(#)               6                           6
  ----------------------------------------------------------------


Not all Values Compared Equal
  ============================
   Variable  No of Differences
  ----------------------------
     arm             10
  ----------------------------


  ========================================================================
   VARIABLE   studyid    usubjid   subjid        BASE         COMPARE
  ------------------------------------------------------------------------
     arm     "Study A"   1015    601881432  Treatment:T8  Treatment:T7
     arm     "Study A"   1024    250148677  Treatment:T8  Treatment:T7
     arm     "Study A"   1035    824826705  Treatment:T8  Treatment:T7
     arm     "Study A"   1072    458864965  Treatment:T8  Treatment:T7
     arm     "Study A"   1080    804004663  Treatment:T8  Treatment:T7
     arm     "Study A"   1082    159361359  Treatment:T8  Treatment:T7
     arm     "Study A"   1121    698873880  Treatment:T8  Treatment:T7
     arm     "Study A"   1150    356729620  Treatment:T8  Treatment:T7
     arm     "Study A"   1157    418465525  Treatment:T8  Treatment:T7
     arm     "Study A"   1158    648167005  Treatment:T8  Treatment:T7
  ------------------------------------------------------------------------
```

Output 3. Comparing datasets in R using {diffdf}.

Comparing the three examples from both SAS and R, the verification results are easily obtained, and all mismatches are displayed in the log. In SAS, mismatches in the *arm* variable are shown in a table, comparing production data with the Verification Programmer data. One advantage of R is that both the {arsenal} and {diffdf} packages can provide key variable information for mismatched records. In R, the programmer can select key variables that help identify the affected records. In contrast, the COMPARE Procedure in SAS does not provide detailed record-level information beyond the mismatched variables and values. To obtain similar details in SAS, programmers need to write additional code or create a macro.

TARGETED INDEPENDENT PROGRAMMING WITH {DIFFDF}

In targeted independent programming, the Verification Programmer creates a new script to produce an output independently. It is similar to independent programming, but the goal is to align with specific

content rather than the entire output. For example, targeted content may include a selected group of participants with specific symptoms or one or two columns within the output. Similar to independent programming, R packages such as {arsenal} and {diffdf} can be used to provide more detailed information when the content of datasets differs. Below is an example where the production data and verifier data match when checked against a specific site ID. The {diffdf} package offers a convenient TRUE/FALSE output, which can be helpful for a quick initial check before diving into a more detailed review.

```
# compare variables from site 139

library(tidyverse)

production_data <- production_data %>%
  filter(new_siteid == 139)

verifier_data <- verifier_data %>%
  filter(new_siteid == 139)

diff <- diffdf(production_data, verifier_data, suppress_warnings =
TRUE)
diffdf_has_issues(diff)
```

Figure 4. Using {diffdf} package in R for targeted independent verification.

```
> diffdf_has_issues(diff)
[1] FALSE
```

Output 4. *diffdf_has_issues()* output when no issues are found.

CODE AND DATA REVIEW WITH {ASSERTR}, {TESTTHAT}, AND {SKIMR}

Code review is the process of examining the programming script for errors, code quality, and maintainability. Data review is the process of examining datasets to confirm accuracy and correctness, identify any inconsistencies, and ensure the data aligns with expectations. Additionally, data review may involve verifying tables, listings, or figures that require more visual inspection. In both code and data review, the Verification Programmer assesses the Production Programmer's script and dataset output against the specifications. R has a wide range of tools that make it easier to check, test, and validate both code and data in a way that's flexible and easy to adapt to different workflows.

VERIFICATION PROGRAMMER

Two designated verifiers are involved in the process: a verifying statistical programmer and the biostatistician working with the dataset. The level of review varies between both: statistical programmers concentrate on reviewing the code to ensure readability and code quality, while their data review involves checking the data against the expectations outlined in the data specification, including verifying accepted values, expected formats, identifying any missing data, ensuring that counts for expected samples are present, and confirming that key fields are included in the dataset. Statisticians may focus more on data review, and in addition to the checks performed by statistical programmers, they can conduct specific reviews relevant to their analysis. If discrepancies are identified in the code or data, the Verification Programmer will follow up with the Production Programmer. Our emphasis will be on the review conducted by the Verification Programmer.

PRODUCTION PROGRAMMER

Code and data reviews can be quite nuanced, and we believe that script owners in all languages should have a high level of confidence in their code and generated dataset. Therefore, we recommend that the Production Programmers insert checks throughout their code to validate the data prior to sending it for

verification by another team member. In SAS, these checks can be implemented using custom macros. In R, packages such as {assertr} and {testthat} are available on CRAN and can be used to assist with these checks. This approach not only enables the Production Programmer to perform self-checks on their script but also provides further assurance that the data meets expectations. This effectively adds the Production Programmer as another contributor to the verification process with the Verification Programmer and statistician. Below, we will illustrate an example of how a Production Programmer can confirm that the data is structured as expected within their programming script using R.

Programmer Self-Checks

Our example will entail merging clinical and demographic data. Every participant is expected to have a treatment arm assigned to them. Additionally, the dataset is expected to have 114 participants. We can use the {assertr} and {testthat} packages to verify these expectations. Figure 5 shows the example datasets used, and Figure 6 displays an example code that can be used while programming to check the data using the *not_na()* and *assert()* functions from {assertr} and the *expect_equal*() function from {testthat}.

```
> sample_info
# A tibble: 114 × 5
   studyid usubjid    subjid new_siteid new_site
   <chr>     <dbl>     <dbl>      <dbl> <chr>
 1 Study A    1150 356729620        158 Site158
 2 Study A    1035 824826705        158 Site158
 3 Study A    1071 276891215        103 Site103
 4 Study A    1157 418465525        103 Site103
 5 Study A    1024 250148677        107 Site107
 6 Study A    1155 448859312        107 Site107
 7 Study A    1193 310962356        107 Site107
 8 Study A    1142 543886072        107 Site107
 9 Study A    1106 789075683        107 Site107
10 Study A    1028 788339824        139 Site139
# i 104 more rows
# i Use `print(n = ...)` to see more rows
```

```
> demo_info
# A tibble: 114 × 2
      subjid arm
       <dbl> <chr>
 1 356729620 Treatment:T8
 2 824826705 Treatment:T8
 3 276891215 Treatment:T3
 4 418465525 Treatment:T8
 5 250148677 Treatment:T8
 6 448859312 Treatment:T2
 7 310962356 Treatment:T2
 8 543886072 Treatment:T7
 9 789075683 Treatment:T2
10 788339824 Treatment:T3
# i 104 more rows
# i Use `print(n = ...)` to see more rows
```

Figure 5. Participant dataset (left) and demographic dataset (right).

```
library(tidyverse)


merged_data <- sample_info %>%
  full_join(demo_info) %>%
  # EXPECT MERGE TO NOT CREATE ADDITIONAL ROWS
  assertr::verify(nrow(.) == nrow(sample_info)) %>%
  # EXPECT ALL PARTICIPANTS TO HAVE A TREATMENT ASSIGNMENT
  assertr::assert(assertr::not_na, arm)
```

Figure 6. Programming checks using {assertr} and {testthat}.

If an assertion fails, {assertr} will stop execution and will output a helpful table indicating where the assertion failed. In the case where the table *demp_info* did not include treatment arms for all participants in the *sample_info* table, Output 5 shows an example of the *not_na* arm assertion failing, indicating that there is missingness in the *arm* variable. In the case where the assertion passes, the program would continue executing.

```
> merged_data ← sample_info %>%
+   full_join(demo_info) %>%
+   # EXPECT MERGE TO NOT CREATE ADDITIONAL ROWS
+   assertr::verify(nrow(.) == nrow(sample_info)) %>%
+   # EXPECT ALL PARTICIPANTS TO HAVE A TREATMENT ASSIGNMENT
+   assertr::assert(assertr::not_na, arm)
Joining with `by = join_by(subjid)`
Column 'arm' violates assertion 'not_na' 54 times
    verb redux_fn predicate column index value
1 assert       NA    not_na    arm    61  <NA>
2 assert       NA    not_na    arm    62  <NA>
3 assert       NA    not_na    arm    63  <NA>
4 assert       NA    not_na    arm    64  <NA>
5 assert       NA    not_na    arm    65  <NA>
  [omitted 49 rows]

Error: assertr stopped execution
```

Output 5. {assertr} error output.

Tasks such as checking for extra rows, checking for missing values, validating value ranges, and more can be accomplished within the code using {assertr} by using the three major functions: *assert()*, *verify()*, and *insist()*. Other checks, like counts, are better performed with the {testthat} package. Although {testthat} is primarily intended for unit testing, particularly in areas such as R package development, it is also useful for testing our data against the expectations we hold.

An example of how {testthat} would assist in checking for missing participants in the dataset is shown below.

```
> # EXPECT 114 PARTICIPANTS
> merged_data %>%
+   distinct(subjid) %>%
+   nrow() %>%
+   testthat::expect_equal(114)
Error: `.` not equal to 114.
1/1 mismatches
[1] 60 - 114 == -54
```

Output 6. {testthat} error output.

Utilizing a combination of functions from the {assertr} and {testthat} packages during processing is effective in catching issues in real-time, as execution and informative error messages can assist the programmer in quickly identifying and addressing problems.

CODE REVIEW

Code reviews can consist of many elements and may vary based on the needs of the organization or team. Should the code follow a specific structure? Is code style important? Are certain programming principles necessary? Should functions be tested to ensure that expectations are met? Does the code comply with the expectations outlined in the data specification? We encourage readers to consider what is most important to them and their team and create a checklist that can serve as a standard guideline.

Below we share an example checklist that may be useful during code reviews. Keep in mind that what is deemed important can evolve as the organization's needs change.

Code Review Checklist:

- ☐ Code follows programming principles (meaningful variable and function names, consistent formatting, etc.)
- ☐ Code is well commented to explain why
- ☐ Code runs without error
- ☐ Functions consistently yield expected results
- ☐ Code matches data specification expectations
- ☐ Code does not have any hard-coded variables
- ☐ Unit tests are used to test functions

R can assist with code review by offering tools that support programmatic checks of code quality. Packages such as {lintr} (Hester, 2023) and {styler} (Müller & Wiernik, 2023) help ensure consistent formatting and that code follows good programming practices. The **{**testthat**}** package can be used to write unit tests that confirm functions behave as expected, and the *check()* function from **{**devtools**}** (Wickham, Hester, & Chang, 2023) can help identify errors or issues before code is finalized. Together, these tools support a more efficient and reliable code review process.

OUTPUT REVIEW

Data Review

Data review can be considered more straightforward than code review when it comes to knowing what to check. We encourage readers to create a checklist that can serve as a standard guide. In R, the Verification Programmer can write and reuse programmatic checks using R scripts or functions to validate the data. These checks can be saved as testing scripts that can be rerun when data is refreshed and modified, helping ensure the outputs remain aligned with the data's expectations. We also suggest that readers develop a standardized testing script for data verification, which can be adjusted for each use case. Below is an example data review checklist, and Figure 7 is an example of a data review testing script written in R.

Data Review Checklist:

- ☐ Data hash of dataset matches hash shared by production programmer
- ☐ Required variables are present
- ☐ Variables have legal values
- ☐ Missingness is anticipated and allowed by the data specification
- ☐ A unique set of key field values as documented in the data specification must match the number of records in the data
- ☐ Data review script runs without outputting warning messages

```
## Data Review

# confirm data hash given by STP is correct. Confirms I'm looking at the
correct data
testthat::expect_equal("c2db07b214703496c314030b253ef5dd",
                       digest::digest(merged_data))

# confirm key fields subjid, studyid, and arm are in data
testthat::expect_contains(names(merged_data),
                          c("subjid", "studyid", "arm"))

# confirm subjid is a character variable
testthat::expect_is(merged_data$subjid,
                    "character")

# confirm treatment arm is not NA
testthat::expect_false(any(is.na(merged_data$subjid)))
```

Figure 7. Data review example code.

Testing scripts can vary depending on what the team considers most necessary. In the example above, functions from the {testthat} package are used to programmatically check the data, and the example is meant as a script that can be quickly run. Reports can also be generated, and data review can also consist of more visual checks. If needing to inspect the data in a high-level manner, the *skim()* function from the {skimr} package can be of use as it provides more detailed information compared to base R's *summary()* function. Helpful information provided by the function includes the count of rows and columns,

the number of missing values, completion rates, and unique values. Below is an example of the code and output.

```
> skimr::skim(merged_data)
── Data Summary ────────────────────────
                           Values
Name                       merged_data
Number of rows             114
Number of columns          6

___________________________
Column type frequency:
  character                3
  numeric                  3

___________________________
Group variables            None

── Variable type: character ──────────────────────────────────────────────────
  skim_variable n_missing complete_rate min max empty n_unique whitespace
1 studyid               0             1   7   7     0        1          0
2 new_site              0             1   7   7     0       10          0
3 arm                   0             1  12  12     0        8          0

── Variable type: numeric ────────────────────────────────────────────────────
  skim_variable n_missing complete_rate       mean          sd        p0
1 usubjid               0             1      1108.        54.8      1001
2 subjid                0             1 507663348. 223650186.  107150929
3 new_siteid            0             1       124.        13.9       103
         p25        p50        p75      p100 hist
1      1064.      1115       1152.      1199 ▆▆▆▆▆
2 325891936. 509282974. 705370448. 884539942 ▆▆▆▆▆
3       108        133        134        158 ▆▁▁▁▁
```

Output 7. {skimr} *skim()* output.

Report Review

Report review mainly involves two parts: checking the headers and footers and verifying the numbers within and between tables. Some checks in the report review process are automated based on the report content. For example, the number of enrolled participants should be consistent within the same table. Below we illustrate how R code using {tidyverse}, embedded in an R Markdown (Allaire et al., 2024) report, can automate checks that flag inconsistencies in the number of enrolled participants within the same table. In this example, the total expected enrolled participants is 26, but the total enrollment in the table is 23.

TABLE 2

DEMOGRAPHICS

Enrollment data as of November 17, 2025

Number of Enrolled Participants = 26

	Sex at Birth					
	Female		Male			
Ethnic Category[1]	N	%	N	%	Total	%
Hispanic or Latino	1	3.8	4	15.4	5	19.2
Not Hispanic or Latino	1	3.8	17	65.4	18	69.2
Total	2	7.7	21	80.8	23	88.5

Check 2.1 Fail: The number of enrolled particpants does NOT match the number in Total column in each Tables in Table 2. Please check.

Check 2.2 Fail: Some Participants have SEX = 'Unknown' in standard enrollment dataset. Please check.

Output 8. Demographics table automated check.

Another common scenario is verifying the same data point across different tables. For example, in Output 9 and Output 10, there are two participants with a termination reason of "Death." Output 10 is expected to list both participants, but only one is shown. The checks are displayed under each table, providing useful information to help programmers verify the numbers.

TABLE 5

REASONS FOR EARLY STUDY TERMINATION

Data as of November 17, 2025

Participants who Terminated Study Early	8
Death	2
Participant refused further participation	0
Participant is unwilling or unable to comply with required study procedures	1
Lost to follow-up	2
Investigator decision	1
Participant refused further study product use	0
Early study closure	0
Protocol deviation	0
Adverse event	1
Other, specify	1

Check 5.1 Fail: Counts of Deaths in Table 5 does NOT match the number of records listed in Table 8 Death.

Output 9. Reasons for early study termination table automated check.

TABLE 8
DEATHS
Data as of November 17, 2025

Number of Participants with Death = 1

Participant ID	Date Treatment Started	Date Treatment Ended	Date of Death	Cause of Death
993-85814-5	01FEB2021	10FEB2021	10FEB2021	heart arrest

Check 5.1 Fail: Counts of Deaths in Table 5 does NOT match the number of records listed in Table Death.

Check 8.1 Fail:Total number of participants listed in Table Death does NOT match Death counts in standard enrollment dataset. Please check both termination and adverse event datasets

Output 10. Deaths table automated check.

FUTURE WORK

As we continue to refine and expand our options for verification processes, we aim to evaluate the efficiency of different R approaches—such as the {tidyverse}, base R, and {data.table} (Barrett et al., 2025)—in dataset verification. Each of these paradigms offers unique strengths in terms of readability, speed, and scalability, and understanding their relative advantages can help optimize verification workflows. Additionally, we plan to investigate how these approaches can be integrated with existing R packages, such as {arsenal} and {diffdf}, to further enhance the accuracy, automation, and reproducibility of the verification process. Apart from this, we aim to explore the reporting capabilities of the package {pointblank} (Iannone et al., 2025), which provides similar functionalities to those of {assertr} and {diffdf} with the added feature of outputting interactive reports. We aim to expand our collection of validated internal R packages by creating new ones that work with different types of data. Standardizing these packages will enhance data accuracy and streamline the verification process. Finally, we plan to combine the useful features of the R packages mentioned in this paper with our testing scripts to develop an internal R verification package. This package will help check different datasets across our institution in a consistent and reliable way.

CONCLUSION

The insights from this paper provide practical guidance to R and SAS users on how to integrate R into their verification workflows. For programmers transitioning from SAS to R, identifying equivalent verification tools can be challenging. We demonstrated how the COMPARE procedure in SAS can be effectively replicated in R using packages like {arsenal}, {diffdf}, and {testthat}. Unlike SAS, R offers enhanced flexibility in selecting key variables for mismatches, while SAS's built-in procedures provide robust, predefined comparison methods.

Furthermore, we discussed code and data review as a crucial part of verification, emphasizing best practices for programmer self-checks, statistical programmer reviews, and statistician oversight. The integration of automated checks using {assertr} and {testthat} within programming scripts improves error detection and ensures data consistency prior to analysis.

Looking forward, future work will focus on evaluating the strengths and weaknesses of various R approaches—{tidyverse}, base R, and {data.table}—for verification processes, particularly in terms of scalability and usability, as well as developing new R packages for data processing and verification. This paper serves as a guide for both SAS and R programmers, providing practical insights into verification strategies and fostering a structured approach to data verification in the pharmaceutical industry.

REFERENCES

Allaire, J. J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., & Iannone, R. (2024). *rmarkdown: Dynamic documents for R* (Version 2.26) [Computer software]. https://github.com/rstudio/rmarkdown

Barrett, T. (2023). *assertr: Assertive programming for R data analysis pipelines* [Vignette]. Comprehensive R Archive Network (CRAN). https://cran.r-project.org/web/packages/assertr/vignettes/assertr.html

Barrett, T., Dowle, M., Srinivasan, A., Gorecki, J., Chirico, M., Hocking, T., Schwendinger, B., & Krylov, I. (2025). *Introduction to data.table*. https://cran.r-project.org/web/packages/data.table/vignettes/datatable-intro.html

Duran, V., Etikala E., Rammohan H., "*Integrating Practices: How Statistical Programmers Differ and Align Within User Groups*". Retrieved from PharmaSUG 2023 Conference. https://www.lexjansen.com/pharmasug/2023/SI/PharmaSUG-2023-SI-139.pdf.

Gower-Page, C., & Martin, K. (2024). *diffdf: Dataframe Difference Tool* (Version 1.1.1) [R package]. Comprehensive R Archive Network (CRAN). https://cran.r-project.org/package=diffdf

Heinzen, E., Sinnwell, J., Atkinson, E., Gunderson, T., & Dougherty, G. (2021). *arsenal: An Arsenal of 'R' Functions for Large-Scale Statistical Summaries* (Version 3.6.3) [R package]. Comprehensive R Archive Network (CRAN). https://cran.r-project.org/package=arsenal

Hester, J. (2023). *lintr: A 'Linter' for R Code* (R package version 3.0.2) [R package]. https://CRAN.R-project.org/package=lintr

Iannone R, Vargas M, Choe J (2025). *pointblank: Data Validation and Organization of Metadata for Local and Remote Tables*. R package version 0.12.2.9000, https://github.com/rstudio/pointblank, https://rstudio.github.io/pointblank/

Müller, K., & Wiernik, B. M. (2023). *styler: Non-Invasive Pretty Printing of R Code* (R package version 1.10.2) [R package]. https://CRAN.R-project.org/package=styler

PharmaR. (2023). *White paper: Production programming best practices.* https://www.pharmar.org/white-paper/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

Waring, E., Quinn, M., McNamara, A., Arino de la Rubia, E., Zhu, H., & Ellis, S. (2025). *skimr: Compact and Flexible Summaries of Data* (Version 2.2.0) [R package]. rOpenSci. https://docs.ropensci.org/skimr/

Wickham H (2011). "testthat: Get Started with Testing." *The R Journal*, **3**, 5–10. https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf.

Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., ... Yutani, H. (2019). *"Welcome to the tidyverse". Journal of Open Source Software*, 4(43), 1686. https://doi.org/10.21105/joss.01686

Wickham, H., Hester, J., & Chang, W. (2023). *devtools: Tools to Make Developing R Packages Easier* (R package version 2.4.5) [R package]. https://CRAN.R-project.org/package=devtools

Wickham H, Henry L (2025). *purrr: Functional Programming Tools*. (R package version 1.0.4) [R package]. https://github.com/tidyverse/purrr, https://purrr.tidyverse.org/

ACKNOWLEDGMENTS

The authors would like to express gratitude to our SCHARP colleagues and collaborators, past and present, who have generously invested time and wisdom in support of our continuing professional development. We also extend our heartfelt thanks to Amber Randall and Radhika Etikala for their encouragement, guidance, and thoughtful review.

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:

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

Xuehan (Emily) Zhang
Statistical Center for HIV/AIDS Research & Prevention (SCHARP) at Fred Hutchinson Cancer Center
xzhang27@fredhutch.org