PharmaSUG 2023 - Paper AP-130

# Strategies for Code Validation at Statistical Center for HIV/AIDS Research and Prevention (SCHARP)

Marie C Vendettuoli, Xuehuan (Emily) Zhang, and Rodger F Zou,
SCHARP at Fred Hutchinson Cancer Center, Seattle WA

## ABSTRACT

Code validation presents a common challenge to programming teams embedded in the pharmaceutical data sphere. How much effort is enough? Is it possible to have a lightweight and nimble approach to validation that accommodates the variety that statistical programmers see in our daily codebase?

At Statistical Center for HIV/AIDS Research and Prevention, Fred Hutchinson Cancer Center (SCHARP) we have adopted the R Validation Framework (PHUSE, 2021) as a language-agnostic paradigm for validating code at various stages of maturity, during both internal development efforts and when adopting community-authored resources. We will share examples from three major use categories: (1) Validation of community-authored resources (2) Validation integrated into development, and (3) Validation separate from R packages.

A brief review of the R Validation Framework will be provided. Examples are drawn from the R language environment with additional discussion exploring the intersection of a risk-based strategy for adoption and development of an R codebase, with a particular emphasis to empower reproducibility and R package development. While examples shown in this paper and accompanying talk are in the R statistical programming language, the concepts are not dependent on programming language and no audience fluency is expected.

## INTRODUCTION

Data verification and code validation are two sides of ensuring data quality when performing statistical analysis of clinical data for regulatory purposes. This paper focuses on the code validation aspect. When performing code validation, it is easy to focus on the practical challenge of implementing this process. However, significant value lies in capturing decisions implicit in the code base. For what activity and task is this code intended? What is the likelihood that using this code will fail to accomplish the designated task? What are the consequences of a failure and are there features of implementation that flag successes and failures for human attention? Answering these questions for a given codebase collectively identifies the requirements and risk assessment that is the first step of code validation.

### CODE VALIDATION IS NOT DATA VERIFICATION

The nuance when distinguishing between code validation and data verification lies in whether one is considering the reproducibility of the *process* versus a comparison of the *product*, e.g. an individual dataset. A *process* will include definitions of input, output, and transformations. The *product* is simply an output. In the following code example, we see two implementations that deliver the same output, with different definitions of input and transformations:

```
combine_text1 <- function(x, y){
    paste(x, y)
}
combine_text2 <- function(x){
    paste(x, collapse = " ")
}

combine_text1("Hello", "world!")
# [1] "Hello world!"

input_vector_a <- c("Hello", "world!")
combine_text2(input_vector_a)
```

```
# [1] "Hello world!"
```

In the above example, although data verification would demonstrate equivalence for both cases, the differences in implementation will have downstream implications for code reuse and extensibility:

```
combine_text1(input_vector_a)
# Error in paste(x, y) : argument "y" is missing, with no default

combine_text1("Hello", "new", "world")
# Error in combine_text1("Hello", "new", "world!") : unused argument
# ("world!")


combine_text2("Hello", "world!")
# Error in combine_text2("Hello", "world!") : unused argument ("world!")

input_vector_b <- c("Hello", "new", "world!")
combine_text2(input_vector_b)
# [1] "Hello new world!"
```

Some questions for programmers and stakeholders alike in this set of examples is whether potential use cases include inputs of differing lengths or if the length of input arguments is a rigid expectation of length two. While the "Hello world!" implementation is trivial, it is easy to imagine use cases where a function includes evaluating whether inputs meet specified criteria, and the transformations may vary based on that initial evaluation.

## CODE VALIDATION IS SOFTWARE VALIDATION

Under real-world constraints of resource allocation, analytical complexity and emphasis on quality, an investment in statistical programming demonstrates greatest value-add when code can be reused with clear documentation to capture details of that input, transformation, and output. Collectively these documentation elements make up a set of *specifications* that define user expectations regarding the scope of software code, with a particular emphasis on fitness for purpose and illustrative scenarios to capture generalizability. The set of expectations can be referred to as *requirements,* while the scenarios make up a set of *test cases.* For readers coming from a background of software development, this stage of evaluation may be known as "user site testing" and it allows programmers to confirm that they are building the right software. This is done by documenting the tasks that the software code is intended to address while evaluating for appropriate responses to a range of use cases.

Looking towards regulatory guidance on this topic, we see that software validation is "confirmation by examination and provision of objective evidence that software specifications conform to user needs and intended uses, and that the particular requirements implemented through software can be consistently fulfilled." (FDA, 2002). Of note is that the practice of validation is agnostic to language and task, but a particular implementation is focused on meeting specific business needs and generalized beyond a particular study deliverable. What is essential is that a validation effort is both reproducible and traceable, and the test cases address risks of the particular use cases.

## RISK AS A STRUCTURED OPINION

Risk is often measured on the orthogonal axes of impact and likelihood of an event occurring (Table 1). This approach assumes that events of increasing complexity are more likely to happen. Higher risk events are both more likely to happen and have greater negative impact when they occur.

Where opinion comes into play is in the subject matter expert assessment of events along either of these axes. It is not possible or efficient to be exhaustive when identifying possible events. However, features such as: adoption rate, historical usage, documentation, developer testing and previous validation are metrics that justify categorization of lower risk when assessing community-authored sources. Open-source projects in the community such as {riskmetric} (R Validation Hub, et al, 2023) and R Validation Hub collaborators are seeking to formalize this assessment in a manner that is consistent and sharable across the pharmaceutical industry. Such efforts are paralleled by open-source development of R packages to facilitate submission activities by industry collaborators (Herbert et. al, 2022) and reinforced by FDA via guidance (FDA, 2003) and clarifying statement (FDA, 2015).

**Table 1: Risk assessment matrix.**

| **Overall Risk** | *Likelihood* | | |
|---|---|---|---|
| *Impact* | **Low** | **Medium** | **High** |
| **Low** | L | L | M |
| **Medium** | L | M | H |
| **High** | M | H | H |

Internally authored code or software authored by the wider community may also benefit from validation efforts implemented using a just in time strategy to advance efforts with a modular approach. This is the iterative method that we describe in later sections. The general rule of thumb is for events of high risk to be considered with greatest scrutiny when defining requirements and test cases, while controlling for events of low risk which may be deferred to general control checkpoints. Justification of whether to include or exclude medium risk events from the validation process is a decision that is specific to the organization, with an understanding that risk categorization may change over time. The possibility that a risk category may change over time, due to new applications or additional details suggests that it is pragmatic to review the validation plan periodically, and to be open to the possibility of re-validation using an agile, iterative approach. Historically, organizations have avoided iterative validation due to the extensive effort of documenting manual testing. Since the primary activity of statistical programming involves running of code, there exists the opportunity to reuse and extend a suite of *test code* to execute test cases in a validation framework that logs the results for audit and review purposes. The authoring of these test code scripts are subject to the controls and review that is traditionally applied to manual validation, with the adoption of some technical elements to facilitate extensibility.

## CONCEPTS OF A VALIDATION FRAMEWORK

The R Validation Framework (PHUSE, 2021) is a system-agnostic and flexible paradigm to identify, implement and distribute the decisions and tests of validation. While the framework may be deployed without consideration of development stage or programming language, the primary value lies in a modular ability to extend validation as the code base evolves, whether that growth is in the extension of existing code, adoption of community code or a new development effort. The R package {valtools} (Hughes, et.al. 2021) implements this framework specifically for R.

A new implementation of the validation framework should start with evaluation and categorization of risk of a particular event (see: Table 1). While some events may be more likely to occur, the potential negative impact of such events may be minimal. Other events may be less likely to occur but have the potential for greater negative effect. For example, if a variable contains censored data instead of the raw numeric observations, how may that bias later derived values? Does the code alert users of that detail with warnings or informative messaging? And does that impact apply to the use case at hand? While it may be tempting to simply test for all potentially negative events, limitations of resources may curtail what is possible at a practical level. A more efficient and pragmatic strategy is to quantify the likelihood and impact – including thresholds – of risk tolerance for the use cases.

## ORGANIZATION OF VALIDATION ELEMENTS

A key aspect of the validation framework is that elements are organized consistently, and content follows a defined structure. Within the top-level validation folder, one folder exists each for requirements, test cases and test code scripts. Individual files in both the requirements and test cases folders can be markdown (Cone, 2018) or rmarkdown (Xie, 2018) extensions. Test code scripts are written with the file extension ".R". The R package {valtools} (Hughes, et.al., 2021) includes documentation and functions to help with this preparation and may be found at: https://github.com/phuse-org/valtools.

```
- validation/
|   |- requirements/
|   |   |- req1.md
|   |   |- req2.md
|   |   |- ...
|   |- test_cases/
|   |   |- tcase1.md
|   |   |- tcase2.md
|   |   |- ...
|   |- test_code/
|   |- validation.yml
|   |- change_log.md
|
|- report.Rmd
```

**Figure 1: Directory structure of R Validation Framework showing validation elements.**

### Syntax of eye-readable files

Each requirement or test case is captured in an individual file. At the top of the file, common {roxygen2} (Wickham et. al. 2022) style tags are used to capture the document title and details regarding author. The motivation for using these tags is multifactorial: (1) quick to skim-read (2) consistent with documentation of other R package elements for ease of use by programmers (3) plays nicely with version control such as git or svn and (4) leverages auto-complete features of Posit IDE.

{valtools} implements custom tags to map requirements to test cases and identify risk levels for each requirement set. A single requirement may be supported by multiple test cases. Risk levels are a free text field.

### Syntax of test code files

Within each test code file, a single set of tests is R code wrapped within a call to the `test_that` function. Evaluated code is compared to expected results via the `expect_*` set of functions. Both functions may be found as part of the existing {testthat} package (Wickham, 2011).

## VALIDATION ELEMENT CONTENT

### Capturing risk via requirements

FDA guideline (FDA, 2002) instructs industry to determine the level of software validation by assessing the safety risk posed by the system. Typically, SCHARP considers both the likelihood and impact of a negative event, prioritizing high risk use cases for immediate assessment (Table 1). For example, when validating code to generate a statistical report, it may be essential that the necessary math fonts are installed. Since the mathematical characters are displayed in every report, accessing these font libraries is both highly likely (every time) and has enormous negative impact (the report is blocked) if an error occurs, so it may be considered "high risk". At minimum, requirements that carry high risk should be included in the validation plan. Organizations may also choose to include requirements of medium risk as part of initial validation or defer these events to a later validation phase. Unless there is a clear justification for exhaustive validation, it is unlikely that most teams will find it efficient to validate low risk scenarios. When considering risk, it is important to give weight to stakeholder perspectives outside of the immediate software user group. In the high-risk example, statistical analysts may grade risk of formatting errors lower because this user group traditionally interacts with data directly. However, for a stakeholder, failure in the process of generating a report may delay or block access to analysis that is routinely shared by email correspondence.

What is also of considerable value is to consider that requirements, risk assessment and use cases may drift over time. This supports the idea that the requirement list should be organized in a modular manner and be accessible to stakeholders without need for special or proprietary software. Using markdown files and separating each requirement into its own file facilitates these dual goals and introduces the flexibility to reuse (source) this information for multiple audiences. Of additional interest to reviewers and future automation is capturing author/edit details of “who” and “when” directly in the file using headers and meaningful tags. Once the requirements have been drafted, it is possible to compile an initial requirement report as a single PDF and be confident the same source information will be incorporated into the final validation report.

#### Test cases to identify use cases

While requirements are generic high-level descriptors of use cases, “test cases” refer to code-specific usage and is referred to in literature as “User Site Testing”. Note that while test cases describe software specific usage on the intended system, these cases are written in a format that is both human-readable and ready for automation.

#### Test code for execution

Although the purpose of test code files are to document technical details of action taken as part of validation, it is also necessary to consider whether it is possible for an unbiased user to arrive at that course of action. Authors of the codebase being validated have extremely detailed understanding of the codebase and may choose or avoid actions that are undocumented due to that deep and nuanced insight. Likewise, authors of the validation requirements or validation test cases may assume a standard practice that is not-so-standard and warrants capturing through standard operating procedures (SOPs). The solution to addressing this human variability is to simply designate the test code author as someone who was not part of the development team nor is part of authoring other elements of the validation project. Thus, one way of describing the test code is: “what a reasonable programmer” would choose. “Reasonable” meaning that the individual has access to organizational SOPs and the documentation i.e. tutorials and help pages published by developers.

### PULLING IT TOGETHER INTO REPORTS

To execute validation and generate the report involves creating a rmarkdown template with sections that iterate over files in the requirement, test case and test code folders. Both {valtools} and {knitr} offer functions that facilitate this process. Of particular interest is that as the files in each of the folders are updated, new files are added and/or deprecated, re-running the template will update validation, generating a new report without overwriting the existing pdf. Additionally, it is possible to update the template structure, for example if corporate logos change.

## VALIDATION EXAMPLES FROM USE CATEGORIES

### COMMUNITY-AUTHORED RESOURCES: BASE R FUNCTIONS VIA THE R PQ PROJECT

Community-authored R packages are often recognized as the most stable and well-maintained resources. The attributes that contribute to their quality include: a large base of experienced developers, wide adoption across many use cases in production, extensive unit testing, mature documentation and/or tutorials, a robust issue tracking system with rapid response, and open-source code that is under scrutiny by academic and industry experts. For example, {ggplot2} (Wickham et.al., 2016), {kableExtra} (Zhu et.al., 2021), {data.table} (Dowle et.al., 2022), {knitr} (Xie, 2023), and {Rcpp} (Eddelbuettel et.al., 2023) R packages have been maintained over several years, and over a decade in some cases. Other community-authored R packages are less robust, perhaps due to an early developmental stage, niche application, or position in emerging practice.

During the validation process, a risk assessment may address the package as a single unit or explore application of individual functions. Similar tasks may be classified with a common risk level. Alternatively, it may be wise to consult programmers to qualitatively evaluate for developers' consistency with best practices.

## Defining requirements

One reality of validation testing is that it is simply not an efficient use of resources to exhaustively test all scenarios. At SCHARP, our initial interest focuses on identifying basic operations in the categorical areas of command line usage, data ingestion, data output, plot generation, statistical distributions, statistical modeling, statistical tests, and summary statistics. The RPQ project at SCHARP captures the specifications, test cases and test code related to this effort. The following code example explores an individual category e.g., statistical modeling – correlation. For this category statistical analysts identified frequently used evaluations such as Pearson's correlation and Spearman's rank correlation coefficient from two samples. A risk level was assigned to the category, and these were saved as a single requirement file in markdown format:

```
#' @title Statistical Modeling - correlation
#' @editor A Statistical Programmer
#' @editDate 2023-01-01
#' @riskAssessment
#' 1.1: Low/Medium/High
#' 1.2: Low/Medium/High

## Statistical Modeling - correlation

+ 1.1 Calculate Pearson's correlation coefficient between two samples.
+ 1.2 Calculate Spearman's rank correlation coefficient between two
samples.
```

This is repeated for each category. With a complete set of requirements, we can circulate the requirement report for review. Team members confident reviewing markdown files directly leveraged the GitHub interface. After an initial review is complete, a pdf is compiled for circulation by email to individuals who are only available by email. Note that it is also possible to compile these requirements into Word document format to allow for track changes.

## Defining test cases

From these requirements test cases are defined. While requirements are reviewed by stakeholders who may not be technical staff, test cases allow for explicit technical details to be captured, describing the use cases that statistical programmers employ. First, we identify the two input numeric vectors for consideration. Second, we identify which R function are being tested and the output value expected. In this case, both coefficients make use of the `cor` function, but we are interested in different parts of the output. In the test case file, the expected output values are also identified:

```
#' @title Test Case - Statistical Modeling - correlation
#' @editor A Statistical Programmer
#' @editDate 2023-03-24
#' @coverage
#' TC1.1: Req1.1
#' TC1.2: Req1.2

## Statistical Modeling - correlation
```

```
+ Setup: create a vector `v1` with values: 1:5 and the vector `v2` with
values c(1, 1, 3, 6, 2)

+ TC1.1 Test that Pearson's correlation using the `cor` function, setting
the x argument to `v1` and the y argument to `v2`, and assigning the result
to `cor_p`.
    + TC1.1.1 Confirm that the value of `cor_p` is 0.534

+ TC1.2 Test that Spearman's rank correlation using the `cor` function,
setting the x argument to `v1`, the y argument to `v2`, the `method`
argument to "spearman", and assigning the result to `cor_s`.
    + TC1.2.1 Confirm that the value of `cor_s` is 0.667
```

Test cases are written for full coverage of all requirements. Full coverage simply means that all requirements have one or more test cases that demonstrate the use scenario. Because test case files are also authored individually, it is possible to review each set of test case individually. This allows the testing programmer (next section) to start authoring code before all test cases are finalized. In a busy team, this allows for an agile approach and reduces overall elapsed time by introducing parallelization.

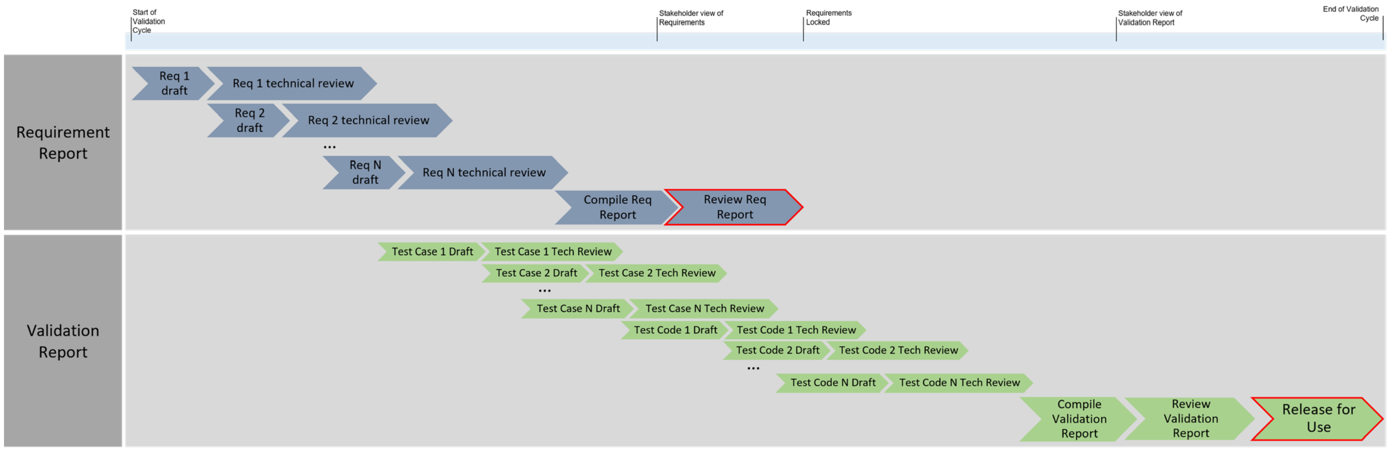


**Figure 2: Gantt chart of hypothetical validation cycle. Parallelization opportunities exist between authoring/updating each requirement. Opportunity for parallelization also exists within the set of test case and within the set of test code. A dependency exists between Requirement 1 -> Test Case 1 -> Test Code 1. Each element is reviewed twice: once at the technical level e.g. peer review, and once at non-technical level e.g. stakeholder review. Elements may be reused for multiple deliverables e.g. Requirement Report and Validation Report. Individual elements may form basis of next iteration of the validation cycle. Deliverables are outlined in red.**

## Programming test code

To ensure that validation is independent of bias, a key feature of validation is that the author of a particular test code script may not be the same person who authored the associated test case, or the function(s) being tested. In many organizations, this task is left to the most novice programmer as an opportunity for training and a sanity check to evaluate the quality of documentation. Each test case is a separate call to the `test_that` function:

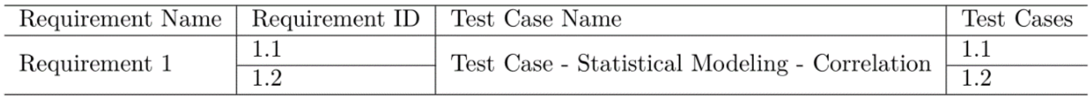
```
# Test setup
v1 <- 1:5
 v2 <- c(1, 1, 3, 6, 2)


#' @editor Test Programmer
#' @editDate 2023-05-14
test_that("TC.1.1", {
  # Pearson's correlation
  cor_p <- cor(x = v1, y = v2)
  expect_equal(cor_p, 0.534)
})

#' @editor Test Programmer
#' @editDate 2023-05-14
test_that("##tc:1.2", {
    # Spearman correlation
    cor_s <- cor(x = v1, y = v2, method = "spearman")
    expect_equal(cor_s, 0.667)
})
```

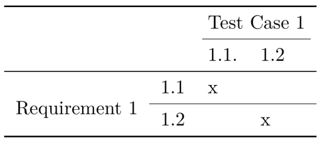
## The validation report

Once all the validation elements are finalized, compiling the validation template executes all the test code scripts and generates the final report ready for signatures. In this project the SCHARP validation team also includes a coverage matrix to demonstrate mapping between requirements, test cases and test code.



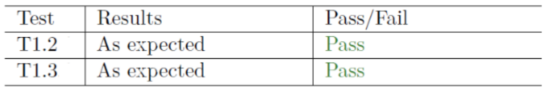
| Requirement Name | Requirement ID | Test Case Name | Test Cases |
|---|---|---|---|
| Requirement 1 | 1.1 | Test Case - Statistical Modeling - Correlation | 1.1 |
|  | 1.2 |  | 1.2 |

**Figure 3: Coverage table shows immediately that each requirement is addressed by one or more test cases.**



|  |  | Test Case 1 |  |
|---|---|---|---|
|  |  | 1.1. | 1.2 |
| Requirement 1 | 1.1 | x |  |
|  | 1.2 |  | x |

**Figure 4: Matrix display of requirement mapping to individual test cases.**



| Test | Results | Pass/Fail |
|---|---|---|
| T1.2 | As expected | Pass |
| T1.3 | As expected | Pass |

**Figure 5: Evaluation of test code cases displaying that the results are as expected and pass validation. If the test code script generates an error or warning, it is a hard stop and the validation report will not compile, e.g. no PDF is generated. To complete compilation of a validation report, all test code scripts must pass.**

## INTEGRATED INTO DEVELOPMENT: {SCHARPYVERSE}

In-house development of R packages presents a unique risk, if only because the software codebase is opaque to testing via implementation by the larger pharmaceutical industry. The validation framework implemented via {valtools} may be incorporated into package development. As statistical programmers at SCHARP create R package to share code across internal teams, we identify requirements during an early stage of development. The validation process for package development is similar to the process of validation of base R functions. Test cases to support the requirements are authored. An independent person is tasked with authoring the test code. The validation report is compiled from the collection of requirements, test cases and test code.

A benefit of validation as part of development is that during distribution of the validated package, a copy of the validation report pdf and the validation raw files are included. This allows an individual to re-validate if they seek to use the package in a different use case. From a traceability perspective the entire package code base, including validation efforts, are tracked and distributed as a single unit. The key to this functionality is that the entire validation framework directory structure is saved to the "vignettes" folder within the R package directory. Individuals can choose to re-validate by setting `build_vignettes = TRUE` during initial setup for the user. If the subject-matter expert does not need to re-validate, they can set `build_vignettes = FALSE` and can depend on the existing validation report. Additionally, {valtools} facilitates the process of re-executing validation test code at any time subsequently.

## SEPARATE FROM PACKAGE: CUSTOMIZING THE FRAMEWORK

### Validation of annual reporting code

Another example of validation in use at SCHARP involves the validation of R scripts that prepare standard annual reports. The goal of this validation is to reduce the risk associated with this codebase and minimize the risk for use in production without data verification. Usually, the output created using R scripts will be verified by performing double programming to make sure two outputs match i.e. *data verification*. However, data verification can be labor-intensive, especially in cases where the R script is complex, but the use case is highly consistent.

While the validation process for these R scripts begins with collecting requirements, the test cases focus on capturing that the code demonstrates fitness for purpose. For instance, summarization and formatting of values that may be evaluated by visual inspection instead of programmatic test code for a defined set of input cases. While the evaluation of output is manual, the expectations and input are captured using version-controlled csv files.

In this application, we are using the directory structure as identified in Figure 1 and many of the concepts of the Validation Framework as proposed in the PHUSE whitepaper (PHUSE, 2021). Specifically, we preserve the definition of requirements, test case and test code. We track and maintain elements in a modular approach that supports the process visualized in Figure 2. However, what is different is that the Validation Report template is pulling details from files in csv form for requirements and test cases and is evaluating code at command-line. This is consistent with the approach that is currently employed by the {Tplyr} package (Atorus Research, 2023).

### SAS validation

Validation of base SAS functions (Duran, Etikala, Rammohan, 2023, p.1) includes approaches used in both the R PQ project and the customization of annual reporting code described earlier. The risk and usage evaluation are consistent the categorization in use when validating base R functionality. The file format and listing structure of individual requirements and test cases, however, is specific to this validation project. SAS code is run according to organizational standards for that language.

Future work to support this validation use category involves developing functions that may be incorporated into {valtools} to handle csv input as an alternative to the (r)markdown file format currently expected. In preparation for this, a consensus regarding the structure of csv input must be addressed.

## DIFFICULTIES WITH IMPLEMENTATION AND LESSONS LEARNED

Implementation of validation is not without its hurdles. Consider that the definition of validation includes the idea the software must perform consistently for all intended users, and that differences in performance between use cases must be clearly identified and accommodated during production. We encountered difficulties in validating base R for SCHARP given our goal of demonstrating that base R is "consistent" across platforms and environments among various users at SCHARP.

It was necessary to first define a standard environment for validation, especially when considering the different packages each user might have loaded for different data analysis steps. We decided on base R for this reason, so that we would have a foundation for validation that could theoretically perform consistently for all users. By defining a standard environment, regardless of packages loaded, and performing tests in that environment, we are better able to offer the confidence needed for a successful validation project.

In another instance, the initial iteration of test code returned different results between local machines and servers running a different operating system. For example, the simple test case to read in a file with specified, Greek letters such as sigma and alpha are printed in a .csv file:

```
greek_csv <- read.csv("example_csv_encoding.csv", fileEncoding = "windows-1253")
```

Users accessing the R via an instance of RStudio Integrated Development Environment (IDE) expect to see the same performance regardless of whether the application is run locally or hosted via a server instance, despite the two scenarios being run on different operating systems. With the assumption that fileEncoding of type “windows-1253” is appropriate, in some scenarios the special character is encoded as “S” and in others, the same special character is encoded as Ó. Discovering this during review of the validation test code, prior to executing the final validation report allowed the team to identify an encoding that is usable across the computational environments that programmers encounter daily.

Another use case that identified performance differences across operating environments was captured during identifying the appropriate test case for invoking “R CMD” tools from within R. Again, was to demonstrate the usage that can be consistently used across operating systems. Due to assumptions of the function `tools::Rcmd`, SCHARP stakeholders determined that the function call to `system("R CMD BATCH", )` was a better fit for purpose.

Network/user differences that must be anticipated in advance poses even more difficulties for validation. While we can meet the needs and specifications of one network at SCHARP with simple requirements and test cases, here we have multiple people with different needs across a variety of networks that must be captured via validation efforts. This obviously leads to increased complexity in scope and requires increased effort to discern. In multiple networks within SCHARP, assay data formats are continually evolving. Validation of R packages that depend on these data formats must be updated due to differences in needs and expectations that have changed. A common change we anticipate will lead to the addition of columns (variables) and data points (records) for unique entries that must be accounted for in processing that data. While requirements, test cases and test code will need to be updated or developed, many of the existing elements will simply need confirmation that continue to execute as expected. While the R Validation Framework reduces the rework needed for future validation, it does not eliminate need for coordination and collaboration across teams. If a requested calculation needs to be created for one network and is forgone for a different network, test cases would need to capture both events. At SCHARP, similarities and differences in code usage are discussed as part of our joint-network efforts to develop packages for shared use.

A resourcing difficulty is the necessary separation of labor during the validation process. To avoid conflicts such as a programmer reviewing their own functions, or programmers changing specifications retroactively in accordance with their own experience, project collaborators must be divided smartly into appropriate categories. Given for example, 3 programmers and 2 research associates, we can potentially divide the labor as follows:

**Table 2: Example of role assignment during validation. The modular structure of validation elements means that these roles may fluctuate by identifier. That is "Alex A. Programmer" may act as "Programmer 1" for module id 1, and in the capacity of “Programmer 2” for module id 2, where module identifiers 1:N enumerate a single linked set of Requirement/Test Case/Test Code files.**

| Module | Author | Reviewer |
|---|---|---|
| Requirements | Researcher 1 | Researcher 2, Programmers |
| Test Case | Programmer 1 | Researcher 1 and 2 |
| Test Case Code | Programmer 2 | Programmer 3, Researchers |

With fewer programmers or research associates, problems can arise with improper division of labor. Adding more staff can cause duties to become fragmented, and more tracking is needed. For large validation teams, or if multiple validation efforts are simultaneously ongoing, there may be operational benefit by designate a project manager to coordinate.

## FUTURE WORK

At SCHARP, our future efforts are focused in three areas: (1) increasing the number of community-authored R Packages that are validated (2) ongoing validation of code that has been previously validated by the team and (3) developing new R Packages that incorporate the validation framework at the start. This involves defining an ongoing maintenance cycle for code that is currently validated, with the expectation that while a future validation cycle should take less time than initial validation, the total sum of all ongoing validation efforts will increase. It will not be possible to let this accumulate indefinitely, so a clear strategy for risk mitigation that allows designation of resources in a parsimonious manner will be well received. Additionally, any increase to the number of community-authored packages that undergo validation will need assessment from a risk-based perspective, considering efforts and progress towards this goal in the wider R in pharma community and addressing the accumulation of validation demands at a resourcing level.

Given the interest in use of the Validation Framework with requirements and test cases that are captured via CSV files, an opportunity exists to extend the {valtools} package to transform the content into validation files consistently.

## CONCLUSION

In this paper we have described three categories of code validation use cases, with examples from production use at SCHARP. We have shown that the R Validation Framework first proposed by PHUSE contributors (PHUSE, 2021) has been successfully implemented via the R package {valtools} (Hughes, et.al., 2021). We have identified opportunities to extend that implementation for greater flexibility via custom code implemented both at SCHARP and in the community.

Validation of code has historically been a labor-intensive process that is idiosyncratic to an organization. The R Validation Framework and technical implementations thereof present a systematic and modular approach to focus efforts on facilitating risk mitigation. By emphasizing standard concepts, the primary challenges that occur during validation can focus on resolving questions pertaining to use cases of data processing, analysis, and reporting in production. Once these challenges are addressed it is possible to summarize the intersection of requirements, test cases and test code scripts for future reference both in programmatic and eye-readable displays. This approach also creates opportunity to parallelize validation efforts and iterate the validation cycle with greater frequency and minimal rework. At SCHARP, we are excited that this approach allows us to operationalize code validation, increase transparency and promote rigor.

## REFERENCES

1. Atorus Research LLC, Miller, E., Stackhouse, M., Tarasiezecz, A., Kosiba, N., Mascary, S. "A Traceability Focused Grammar of Clinical Data Summary". 2023. https://github.com/atorus-research/Tplyr
2. Cone, M., "The Markdown Guide". 2018. https://www.markdownguide.org/
3. Dowle, M., Srinivasan, A. "data.table: Extension of 'data.frame'". 2022. https://CRAN.R-project.org/package=data.table
4. Duran, V., Etikala E., Rammohan H., "Integrating Practices: How Statistical Programmers Differ and Align Within User Groups". Proceedings of PharmaSUG 2023 (accepted). SI-139.
5. Eddelbuettel D., "Rcpp: Seamless R and C++ Integration", https://cran.r-project.org/web/packages/Rcpp/index.html
6. FDA. "Electronic Signatures – Scope and Application: Guidance for Industry Part 11, Electronic Records". September 2003. Last updated July 2018. Accessed March 17, 2023. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/part-11-electronic-records-electronic-signatures-scope-and-application
7. FDA. "General Principles of Software Validation: Guidance for Industry and FDA Staff". January 2002. Last updated May 2019. Accessed March 17, 2023. https://www.fda.gov/regulatory-information/search-fda-guidance-documents/general-principles-software-validation
8. FDA. "Statistical Software Clarifying Statement". May 2015. Accessed March 17, 2023. https://www.fda.gov/media/161196/download
9. Hartford, A., Pang, H., Wan, L., Griffiths, K.L., Eds. "Biopharmaceutical Report". Fall 2022. 29(3)
10. Hughes, E., Vendettuoli, M., Miller, E., Peyman, E. "Automate Validated Package Creation". 2021. https://github.com/phuse-org/valtools
11. R Validation Hub, Kelkhoff, D., Gotti, M., Miller, E., Kevin, K., Zhang, Y., Milliman, E., Manitz, J., "riskmetric: Risk Metrics to Evaluating R Packages". 2023. https://CRAN.R-project.org/package=riskmetric
12. PHUSE. R Package Validation Framework. 2021. WP059. Accessed March 17, 2023. https://stage.phuse.global/Deliverables/1
13. Wickham H., "ggplot2: Elegant Graphics for Data Analysis." 2016. Springer-Verlag New York. https://ggplot2.tidyverse.org
14. Wickham, H., "testthat: Get Started with Testing". 2011. The R Journal https://journal.r-project.org/archive/2011-1/RJournal_2011-1_Wickham.pdf
15. Wickham, H., Danenberg, P., Csárdi, G., Eugster, M. "roxygen2: In-Line Documentation for R". 2022. https://CRAN.R-project.org/package=roxygen2
16. Wickham, H., et.al. "Welcome to the {tidyverse}". 2019. Journal of Open Source Software, 4(43).
17. Xie Y, "knitr: A General-Purpose Package for Dynamic Report Generation in R". 2022. https://cran.r-project.org/web/packages/knitr/index.html
18. Xie, Y., Allaire, J.J., Grolemund, G., "R Markdown: The Definitive Guide". 2018. https://bookdown.org/yihui/rmarkdown
19. Zhu, H., "kableExtra: Construct Complex Table with 'kable' and Pipe Syntax. https://CRAN.R-project.org/package=kableExtra

## ACKNOWLEDGMENTS

The authors would like to express gratitude to our SCHARP coworkers and collaborators, past and present, who have generously invested time and wisdom in support of our continuing professional development. We declare that no conflict of interests exists.

## RECOMMENDED READING

- *R Validation Hub https://www.pharmar.org/*
- *valtools cheat sheet https://github.com/phuse-org/valtools#cheat-sheet*

## CONTACT INFORMATION

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

Marie C Vendettuoli, PhD
mvendett@scharp.org