12-2021

Software Engineering and R Programming: A Call for Research

Melina Vidoni

Software Engineering and R Programming: A Call for Research

by Melina Vidoni

Abstract Although R programming has been a part of research since its origins in the 1990s, few studies address scientific software development from a Software Engineering (SE) perspective. The past few years have seen unparalleled growth in the R community, and it is time to push the boundaries of SE research and R programming forwards. This paper discusses relevant studies that close this gap Additionally, it proposes a set of good practices derived from those findings aiming to act as a call-to-arms for both the R and RSE (Research SE) community to explore specific, interdisciplinary paths of research.

Introduction

R is a multi-paradigm statistical programming language, based on the S statistical language (Morandat et al., 2012), developed in the 1990s by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand. It is maintained by the R Development Core Team (Thieme, 2018). Though CRAN (Comprehensive Archive Network) was created for users to suggest improvements and report bugs, nowadays, is the official venue to submit user-generated R packages (Ihaka, 2017). R has gained popularity for work related to statistical analysis and mathematical modelling and has been one of the fastest-growing programming languages (Muenchen, 2017). In July 2020, R ranked 8th in the TIOBE index, which measures of popularity of programming languages; as a comparison, one year before (July 2019), TIOBE ranked R in the 20th position (TIOBE, 2020). According to (Korkmaz et al., 2018), *"this has led to the development and distribution of over 10,000 packages, each with a specific purpose"*. Furthermore, it has a vibrant end-user programming community, where the majority of contributors and core members are *"not software engineers by trade, but statisticians and scientists"*, with diverse technical backgrounds and application areas (German et al., 2013).

R programming has become an essential part of *computational science*–the "application of computer science and Software Engineering (SE) principles to solving scientific problems" (Hasselbring et al., 2019). As a result, there are numerous papers discussing R packages explicitly developed to close a particular gap or assist in the analysis of data of a myriad of disciplines. Regardless of the language used, the development of software to assist in research ventures in a myriad of disciplines, has been termed as 'research SE' (RSE) (Cohen et al., 2021). Overall, RSE has several differences with traditional software development, such as the lifecycles used, the software goals and life-expectancy, and the requirements elicitation. This type of software is often "constructed for a particular project, and rarely maintained beyond this, leading to rapid decay, and frequent 'reinvention of the wheel" (Rosado de Souza et al., 2019).

However, both RSE and SE for R programming remain under-explored, with little SE-specific knowledge being tailored to these two areas. This poses several problems, given that in computational science, research software is a central asset for research. Moreover, although most RSE-ers (the academics writing software for research) come from the research community, a small number arrive from a professional programming background (Cohen et al., 2021; Pinto et al., 2018). Previous research showed R programmers do not consider themselves as developers (Pinto et al., 2018) and that few of them are aware of the intricacies of the language (Morandat et al., 2012). This poses a problem since the lack of formal programming training can lead to lower quality software (Hasselbring et al., 2019), as well as less-robust software (Vidoni, 2021a). This is problematic since ensuring sustainable development focused on code quality and maintenance is essential for the evolution of research in a myriad of computational science disciplines, as faulty and low-quality software can potentially affect research results (Cohen et al., 2018).

As a result, this paper aims to provide insights into three core areas:

- Related works that tackle both RSE and R programming, discussing their goals, motivations, relevancy, and findings. This list was curated through an unstructured review and is, by no means, complete or exhaustive.
- Organising the findings from those manuscripts into a list of good practices for developers. This is posed as a baseline, aiming to be improved with time, application, and experience.
- A call-to-arms for RSE and R communities, to explore interdisciplinary paths of research, covering not only empirical SE topics but also further developing the tools available to R programmers.

The rest of this paper is organised as follows. Section 2.2 presents the related works, introducing them one by one. Section 2.3 outlines the proposed best practices, and Section 2.4 concludes this work with a call-to-action for the community.

Related Works

This Section discusses relevant works organised in four sub-areas related to software development: coding in R, testing packages, reviewing them, and developers' experiences.

Area: Coding in R

Code quality is often related to technical debt. Technical Debt (TD) is a metaphor used to reflect the implied cost of additional rework caused by choosing an easy solution in the present, instead of using a better approach that would take longer (Samarthyam et al., 2017).

Claes et al. (2015) mined software repositories (MSR) to evaluate the *maintainability* of R packages published in CRAN. They focused on *function clones*, which is the practice of duplicating functions from other packages to reduce the number of dependencies; this is often done by copying the code of an external function directly into the package under development or by re-exporting the function under an alias. Code clones are harmful because they lead to redundancy due to code duplication and are a code smell (i.e., a practice that reduces code quality, making maintenance more difficult).

The authors identified that cloning, in CRAN packages only, is often caused by several reasons. These are: coexisting package versions (with some packages lines being cloned in the order of the hundreds and thousands), forked packages, packages that are cloned more than others, utility packages (i.e., those that bundle functions from other packages to simply importing), popular packages (with functions cloned more often than in other packages), and popular functions (specific functions being cloned by a large number of packages).

Moreover, they analysed the cloning trend for packages published in CRAN. They determined that the ratio of packages impacted by cloning appears to be stable but, overall, it represents over quarter-million code lines in CRAN. Quoting the authors, *"those lines are included in packages representing around 50% of all code lines in CRAN."* (Claes et al., 2015). Related to this, Korkmaz et al. (2019) found that the more dependencies a package has, the less likely it is to have a higher impact. Likewise, other studies have demonstrated that scantily updated packages that depend on others that are frequently updated are prone to have more errors caused by incompatible dependencies (Plakidas et al., 2017); thus, leading developers to clone functions rather than importing.

Code quality is also reflected by the comments developers write in their code. The notion of *Self-Admitted Technical Debt* (SATD) indicates the case where programmers are aware that the current implementation is not optimal and write comments alerting of the problems of the solution Potdar and Shihab (2014). Vidoni (2021b) conducted a three-part mixed-methods study to understand SATD in R programming, mining over 500 packages publicly available in GitHub and enquiring their developers through an anonymous online survey. Overall, this study uncovered that:

- Slightly more than 1/10th of the comments are actually "commenting out" (i.e., nullifying) functions and large portions of the code. This is a code smell named *dead code*, which represents functions or pieces of unused code that are never called or reached. It clogs the files, effectively reducing the readability Alves et al. (2016).
- About 3% of the source code comments are SATD, and 40% of those discuss code debt. Moreover, about 5% of this sample discussed *algorithm debt*, defined as *"sub-optimal implementations of algorithm logic in deep learning frameworks. Algorithm debt can pull down the performance of a system"* Liu et al. (2020).
- In the survey, developers declared adding SATD as "self reminders" or to "schedule future work", but also responded that they rarely address the SATD they encounter, even if it was added by themselves. This trend is aligned with what happens in traditional object-oriented (OO) software development.

This work extended previous findings obtained exclusively for OO, identifying specific debt instances as developers perceive them. However, a limitation of the findings is that the dataset was manually generated. For the moment, there is no tool or package providing support to detect SATD comments in R programming automatically.

Area: Testing R Packages

Vidoni (2021a) conducted a mixed-methods MSR (Mining Software Repositories) that combined mining GitHub repositories with a developers survey to study *testing technical debt* (TTD) in R programming–the test dimension of TD.

Overall, this study determined that R packages testing has poor quality, specifically caused by the situations summarised in Table 1. A finding is with regards to the type of tests being carried out. When designing test cases, good practices indicate that developers should test *common cases* (the "traditional" or "more used" path of an algorithm or function) as well as *edge cases* (values that require special handling, hence assessing boundary conditions of an algorithm or function) (Daka and Fraser, 2014). Nonetheless, this study found that almost 4/5 of the tests are common cases, and a vast majority of alternative paths (e.g., accessible after a condition) are not being assessed.

Moreover, this study also determined that available tools for testing are limited regarding their documentation and the examples provided (as indicated by survey respondents). This includes the usability of the provided assertions (given that most developers use custom-defined cases) and the lack of tools to automate the initialisation of data for testing, which often causes the test suits to fail due to problems in the suite itself.

Smell	Definition (Samarthyam et al., 2017)	Reason (Vidoni, 2021a)
Inadequate Unit Tests	The test suite is not ideal to ensure quality testing.	Many relevant lines remain untested. Alternative paths (i.e., those accessible after a condition) are mostly untested. There is a large variability in the coverage of packages from the same area (e.g., biostatistics). Developers focus on common cases only, leading to incomplete testing.
Obscure Unit Tests	When unit tests are obscure, it becomes difficult to understand the unit test code and the production code for which the tests are written.	Multiple asserts have unclear messages. Multiple asserts are mixed in the same test function. Excessive use of user-defined asserts instead of relying on the available tools.
Improper Asserts	Wrong or non-optimal usage of asserts leads to poor testing and debugging.	Testing concentrated on common cases. Excessive use of custom asserts. Developers still uncover bugs in their code even when the tests are passing.
Inexperienced Testers	Testers, and their domain knowledge, are the main strength of exploratory testing. Therefore, low tester fitness and non-uniform test accuracy over the whole system accumulate residual defects.	Survey participants are reportedly highly-experienced, yet their most common challenge was lack of testing knowledge and poor documentation of tools.
Limited Test Execution	Executing or running only a subset of tests to reduce the time required. It is a shortcut increasing the possibility of residual defects.	A large number of mined packages (about 35%) only used manual testing, with no automated suite (e.g., `testthat`). The survey responses confirmed this proportion.
Improper Test Design	Since the execution of all combination of test cases is an effort-intensive process, testers often run only known, less problematic tests (i.e., those less prone to make the system fail). This increases the risk of residual defects.	The study found a lack of support for automatically testing plots. The mined packages used `testthat` functions to generate a plot that was later (manually) inspected by a human to evaluate readability, suitability, and other subjective values. Survey results confirmed developers struggle with plots assessment.

Table 1: Problems found by Vidoni (2021a) regarding unit testing of R packages.

Křikava and Vitek (2018) conducted an MSR to inspect R packages' source code, making available a tool that automatically generates unit tests. In particular, they identified several challenges regarding testing caused by the language itself, namely its extreme dynamism, coerciveness, and lack of types, which difficult the efficacy of traditional test extraction techniques.

In particular, the authors worked with *execution traces*, "the sequence of operations performed by a program for a given set of input values" (Křikava and Vitek, 2018), to provide genthat, a package to optimise the unit testing of a target package (Krikava, 2018). **genthat** records the execution traces of a target package, allowing the extraction of unit test functions; however, this is limited to the public interface or the internal implementation of the target package. Overall, its process requires installation, extraction, tracing, checking and minimisation.

Both `genthat` and the study performed by these authors are highly valuable to the community since the *minimisation* phase of the package checks the unit tests and discards those failing, and records to coverage, eliminating redundant test cases. Albeit this is not a solution to the lack of edge cases detected in another study (Vidoni, 2021a), this **genthat** assists developers and can potentially reduce the workload required to obtain a baseline test suite. However, this work's main limitation is its emphasis on the coverage measure, which is not an accurate reflection of the tests' quality.

Finally, Russell et al. (2019) focused on the *maintainability quality* of R packages caused by their *testing* and *performance*. The authors conducted an MSR of 13500 CRAN packages, demonstrating that "reproducible and replicable software tests are frequently not available". This is also aligned with the findings of other authors mentioned in this Section. They concluded with recommendations to improve the long-term maintenance of a package in terms of testing and optimisation, reviewed in Section 2.3.

Area: Reviewing Packages

The increased relevance of software in data science, statistics and research increased the need for reproducible, quality-coded software (Howison and Herbsleb, 2011). Several community-led organisations were created to organize and review packages - among them, *rOpenSci* (Ram et al., 2019; rOpenSci et al., 2021) and *BioConductor* (Gentleman et al., 2004). In particular, *rOpenSci* has established a thorough peer-review process for R packages based on the intersection of academic peer-reviews and software reviews.

As a result, Codabux et al. (2021) studied *rOpenSci* open peer-review process. They extracted completed and accepted packages reviews, broke down individual comments, and performed a card sorting approach to determine which types of TD were most commonly discussed.

One of their main contributions is a taxonomy of TD extending the current definitions to R programming. It also groups debt types by *perspective*, representing 'who is the most affected by a type of debt". They also provided examples of rOpenSci's peer-review comments referring to a specific debt. This taxonomy is summarised in Table 2, also including recapped definitions.

Perspective	TD Type	Reason
User	Usability	In the context of R, test debt encompasses anything related to usability, interfaces, visualisation and so on.
	Documentation	For R, this is anything related to **roxygen2** (or alternatives such as the Latex or Markdown generation), readme files, vignettes and even **pkgdown** websites.
	Requirements	Refers to trade-offs made concerning what requirements the development team needs to implement or how to implement them.
Developer	Test	In the context of R, test debt encompasses anything related to coverage, unit testing, and test automation.
	Defect	Refers to known defects, usually identified by testing activities or by the user and reported on bug tracking systems.
	Design	For R, this debt is related to any OO feature, including visibility, internal functions, the triple-colon operator, placement of functions in files and folders, use of **roxygen2** for imports, returns of objects, and so on.
	Code	In the context of R, examples of code debt are anything related to renaming classes and functions, $<-$ vs. $=$, parameters and arguments in functions, FALSE/TRUE vs. F/T, print vs warning/message.
CRAN	Build	In the context of R, examples of build debt are anything related to Travis, Codecov.io, GitHub Actions, CI, AppVeyor, CRAN, CMD checks, `devtools::check`.
	Versioning	Refers to problems in source code versioning, such as unnecessary code forks.
	Architecture	for example, violation of modularity, which can affect architectural requirements (e.g., performance, robustness).

Table 2: Taxonomy of TD types and perspectives for R packages, proposed by Codabux et al. (2021).

Additionally, they uncovered that almost one-third of the debt discussed is *documentation debt*–related to how well packages are being documented. This was followed by *code debt*, providing a different distribution than the one obtained by Vidoni (2021b). This difference is caused by rOpenSci reviewers focusing on documentation (e.g., comments written by reviewers' account for most of the *documentation debt*), while developers' comments concentrate their attention in *code debt*. The entire classification process is detailed in the original study Codabux et al. (2021).

Area: Developers' Experiences

Developers' perspectives on their work are fundamental to understand how they develop software. However, scientific software developers have a different point of view than 'traditional' programmers (Howison and Herbsleb, 2011).

Pinto et al. (2018) used an online questionnaire to survey over 1500 R developers, with results enriched with metadata extracted from GitHub profiles (provided by the respondents in their answers). Overall, they found that scientific developers are primarily self-taught but still consider peer-learning a second valuable source. Interestingly, the participants did not perceive themselves as programmers, but rather as a member of any other discipline. This also aligns with findings provided by other works (German et al., 2013; Morandat et al., 2012). Though the latter is understandable, such perception may

pose a risk to the development of quality software as developers may be inclined to feel 'justified' not to follow good coding practices Pinto et al. (2018).

Additionally, this study found that scientific developers work alone or in small teams (up to five people). Interestingly enough, they found that people spend a significant amount of time focused on coding and testing and performed an ad-hoc elicitation of requirements, mostly 'deciding by themselves' on what to work next, rather than following any development lifecycle.

When enquiring about commonly-faced challenges, the participants of this study considered the following: cross-platform compatibility, poor documentation (which is a central topic for reviewers (Codabux et al., 2021)), interruptions while coding, lack of time (also mentioned by developers in another study (Vidoni, 2021b)), scope bloat, lack of user feedback (also related to validation, instead of verification testing), and lack of formal reward system (e.g., the work is not credited in the scientific community (Howison and Herbsleb, 2011)).

Area	Main Problem	Recommended Practice
Lifecycles	The lack of proper requirement elicitation and development organisation was identified as a critical problem for developers (Wiese et al., 2020; Pinto et al., 2018), who often resort to writing comments in the source to remind themselves of tasks they later do not address (Vidoni, 2021b).	There are extremely lightweight agile lifecycles (e.g., Extreme Programming, Crystal Clear, Kanban) that can be adapted for a single developer or small groups. Using these can provide a project management framework that can also organise a research project that depends on creating scientific software.
Teaching	Most scientific developers do not perceive themselves as programmers and are self-taught (Pinto et al., 2018). This hinders their background knowledge and the tools they have available to detect TD and other problems, potentially leading to low-quality code (German et al., 2013).	Since graduate school is considered fundamental for these developers (Pinto et al., 2018), providing a solid foundation of SE-oriented R programming for candidates whose research relies heavily on software can prove beneficial. The topics to be taught should be carefully selected to keep them practical and relevant yet still valuable for the candidates.
Coding	Some problems discussed where functions clones, incorrect imports, non-semantic or meaningful names, improper visibilitiy or file distribution of functions, among others.	Avoid duplicating (i.e., copy-pasting or re-exporting) functions from other packages, and instead use proper selective import, such as **roxygen2**'s `@importFrom` or similar Latex documentation styles. Avoid leaving unused functions or pieces of code that are 'commented out' to be nullified. Proper use of version control enables developers to remove the segments of code and revisit them through previous commits. Code comments are meant to be meaningful and should not be used as a planning tool. Comments indicating problems or errors should be addressed (either when found, if the problem is small or planning for a specific time to do it if the problem is significant). Names should be semantic and meaningful, maintaining consistency in the whole project. Though there is no pre-established convention for R, previous works provide an overview (Baath, 2012), as well as packages, such as the tidyverse's style guide.
Testing	Current tests leave many relevant paths unexplored, often ignoring the testing of edge cases and damaging the robustness of the code packaged (Vidoni, 2021a; Russell et al., 2019)	All alternative paths should be tested (e.g., those limited by conditionals). Exceptional cases should be tested; e.g., evaluating that a function throws an exception or error when it should, and evaluating other cases such as (but not limited to), nulls, `NA`s, `NaN`s, warnings, large numbers, empty strings, empty variables (e.g., `character(0)`, among others. Other specific testing cases, including performance evaluation and profiling, discussed and exemplified by Russell et al. (2019).

Table 3: Recommendations of best practices, according to the issues found in previous work and good practices established in the SE community.

This study (Pinto et al., 2018) was followed up to create a taxonomy of problems commonly faced by scientific developers (Wiese et al., 2020). They worked with over 2100 qualitatively-reported problems and grouped them into three axes; given the size of their taxonomy, only the larger groups are summarised below:

- *Technical Problems:* represent almost two-thirds of the problems faced. They are related to software design and construction, software testing and debugging, software maintenance and evolution, software requirements and management, software build and release engineering, software tooling' and others (e.g., licensing, CRAN-related, user interfaces).
- *Social-Related Problems:* they represent a quarter of the problems faced by developers. The main groups are: publicity, lack of support, lack of time, emotional and communication and collaboration.
- *Scientific-Related Problems:* are the smaller category related to the science supporting or motivating the development. The main groups are: scope, background, reproducibility and data handling, with the latter being the most important.

These two works provide valuable insight into scientific software developers. Like other works mentioned in this article, albeit there are similarities with traditional software development (both in terms of programming paradigms and goals), the differences are notable enough to warrant further specialised investigations.

Towards Best Practices

Based on well-known practices for traditional software development (Sommerville, 2015), this Section outlines a proposal of best practices for R developers. These are meant to target the weaknesses found by the previous studies discussed in Section 2.2. This list aims to provide a baseline, aiming that (through future research works) they can be improved and further tailored to the needs of scientific software development and the R community in itself.

The practices discussed span from overarching (e.g., related to processes) to specific activities. They are summarised in Table 3.

Call to Action

Scientific software and R programming became ubiquitous to numerous disciplines, providing essential analysis tools that could not be completed otherwise. Albeit R developers are reportedly struggling in several areas, academic literature centred on the development of scientific software is scarce. As a result, this Section provides two calls to actions: one for R users and another for RSE academics.

Research Software Engineering Call: SE for data science and scientific software development is crucial for advancing research outcomes. As a result, interdisciplinary works are increasingly needed to approach specific areas. Some suggested topics to kickstart this research are as follows:

- *Lifecycles and methodologies for project management.* Current methodologies focus on the demands of projects with clear stakeholders and in teams of traditional developers. As suggested in Section 2.3, many agile methodologies are suitable for smaller teams or even uni-personal developments. Studying this and evaluating its application in practice can prove highly valuable.
- *Specific debts in scientific software.* Previous studies highlighted the existence of specific types of debt that are not often present in traditional software development (e.g., algorithm and reproducibility) (Liu et al., 2020) and are therefore not part of currently accepted taxonomies (Alves et al., 2016; Potdar and Shihab, 2014). Thus, exploring these specific problems can help detect uncovered problems, providing viable paths of actions and frameworks for programmers.
- *Distinct testing approaches.* R programming is an inherently different paradigm, and current guidance for testing has been developed for the OO paradigm. As a result, more studies are needed to tackle specific issues that may arise, such as how to test visualisations or scripts (Vidoni, 2021a), and how to move beyond coverage by providing tests that are optimal yet meaningful Křikava and Vitek (2018).

R Community Call: The following suggestions are centred on the abilities of the R community:

- Several packages remain under-developed, reportedly providing incomplete tools. This happens not only in terms of functionalities provided but also on their documentation and examples. For instance, developers disclosed that lack of specific examples was a major barrier when properly

testing (Vidoni, 2021a). Extending the examples available in current packages can be achieved through community calls, leveraging community groups' reach, such as R-Ladies and RUGs (R User Groups). Note that this suggestion is not related to package development guides but to a community-sourced improvement of the documentation of existing packages.

- Additionally, incorporating courses in graduate school curricula that focus on "SE for Data Science" would be beneficial for the students, as reported in other works (Pinto et al., 2018; Wiese et al., 2020). However, this can only be achieved through interdisciplinary work that merges specific areas of interest with RSE academics and educators alike. Once more, streamlined versions of these workshops could be replicated in different community groups.

There is a wide range of possibilities and areas to work, all derived from diversifying R programming and RSE. This paper highlighted meaningful work in this area and proposed a call-to-action to further this area of research and work. However, these ideas need to be repeatedly evaluated and refined to be valuable to R users.

Acknowledgements

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors. The author is grateful to both R-Ladies and rOpenSci communities that fostered the interest in this topic and to Prof. Dianne Cook for extending the invitation for this article.

Packages Mentioned

The following packages were mentioned in this article:

- **covr**, for package coverage evaluation. Mentioned by Křikava and Vitek (2018) and Codabux et al. (2021). Available at: https://cran.r-project.org/web/packages/covr/index.html.
- **genthat**, developed by Křikava and Vitek (2018), to optimise testing suits. Available at https://github.com/PRL-PRG/genthat.
- **pkgdown** for package documentation. Mentioned by Codabux et al. (2021) as part of documentation debt. Available at: https://cran.r-project.org/web/packages/pkgdown/index.html.
- **roxygen2**, for package documentation. Recommended in Section 2.3, and mentioned as examples of design and documentation debt by Codabux et al. (2021). Available at https://cran.r-project.org/web/packages/roxygen2/index.html.
- **testthat**, most used testing tool, according to findings by Vidoni (2021a). Mentioned when discussing testing debt by Codabux et al. (2021). Available at https://cran.r-project.org/web/packages/testthat/index.html.
- **tidyverse**, bundling a large number of packages and providing a style guile. Mentioned in Section 2.3. Available at: https://cran.r-project.org/web/packages/tidyverse/index.html.

Bibliography

N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology*, 70:100–121, 2016. ISSN 0950-5849. doi: https://doi.org/10.1016/j.infsof.2015.10.008. [p7, 11]

R. Baath. The state of naming conventions in r. *The R Journal*, 4:74–75, 12 2012. doi: 10.32614/RJ-2012-018. [p10]

M. Claes, T. Mens, N. Tabout, and P. Grosjean. An empirical study of identical function clones in CRAN. In *2015 IEEE 9th International Workshop on Software Clones (IWSC)*, pages 19–25, Mar. 2015. doi: 10.1109/IWSC.2015.7069885. [p7]

Z. Codabux, M. Vidoni, and F. Fard. Technical Debt in the Peer-Review Documentation of R Packages: a rOpenSci Case Study. In *2021 International Conference on Mining Software Repositories*, pages 1–11, Madrid, Spain, 2021. IEEE. doi: https://arxiv.org/abs/2103.09340. [p9, 10, 12]

J. Cohen, D. S. Katz, M. Barker, R. Haines, and N. Chue Hong. Building a sustainable structure for research software engineering activities. In *2018 IEEE 14th International Conference on e-Science (e-Science)*, pages 31–32, 2018. doi: 10.1109/eScience.2018.00015. [p6]

J. Cohen, D. S. Katz, M. Barker, N. Chue Hong, R. Haines, and C. Jay. The four pillars of research software engineering. *IEEE Software*, 38(1):97–105, 2021. doi: 10.1109/MS.2020.2973362. [p6]

E. Daka and G. Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014. [p8]

R. C. Gentleman, V. J. Carey, D. M. Bates, B. Bolstad, M. Dettling, S. Dudoit, B. Ellis, L. Gautier, Y. Ge, J. Gentry, K. Hornik, T. Hothorn, W. Huber, S. Iacus, R. Irizarry, F. Leisch, C. Li, M. Maechler, A. J. Rossini, G. Sawitzki, C. Smith, G. Smyth, L. Tierney, J. Y. Yang, and J. Zhang. Bioconductor: open software development for computational biology and bioinformatics. *Genome Biology*, 5(10):R80, Sep 2004. ISSN 1474-760X. doi: 10.1186/gb-2004-5-10-r80. URL https://doi.org/10.1186/gb-2004-5-10-r80. [p9]

D. M. German, B. Adams, and A. E. Hassan. The Evolution of the R Software Ecosystem. In *2013 17th European Conference on Software Maintenance and Reengineering*, pages 243–252, Mar. 2013. doi: 10.1109/CSMR.2013.33. ISSN: 1534-5351. [p6, 9, 10]

W. Hasselbring, L. Carr, S. Hettrick, H. Packer, and T. Tiropanis. Fair and open computer science research software, 2019. [p6]

J. Howison and J. D. Herbsleb. Scientific software production: Incentives and collaboration. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, page 513–522, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450305563. doi: 10.1145/1958824.1958904. URL https://doi.org/10.1145/1958824.1958904. [p9, 10]

R. Ihaka. The r project: A brief history and thoughts about the future, 2017. URL https://www.stat.auckland.ac.nz/~{}ihaka/downloads/Massey.pdf. [p6]

G. Korkmaz, C. Kelling, C. Robbins, and S. A. Keller. Modeling the Impact of R Packages Using Dependency and Contributor Networks. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 511–514, Aug. 2018. doi: 10.1109/ASONAM.2018.8508255. ISSN: 2473-991X. [p6]

G. Korkmaz, C. Kelling, C. Robbins, and S. Keller. Modeling the impact of Python and R packages using dependency and contributor networks. *Social Network Analysis and Mining*, 10(1):7, Dec. 2019. ISSN 1869-5469. doi: 10.1007/s13278-019-0619-1. URL https://doi.org/10.1007/s13278-019-0619-1. [p7]

F. Krikava. fikovnik/ISSTA18-artifact: ISSTA'18 Artifact release, July 2018. URL https://doi.org/10.5281/zenodo.1306437. [p8]

F. Křikava and J. Vitek. Tests from traces: Automated unit test extraction for r. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 232–241, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213863. URL https://doi.org/10.1145/3213846.3213863. [p8, 11, 12]

J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li. Is Using Deep Learning Frameworks Free? Characterizing Technical Debt in Deep Learning Frameworks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Society*, ICSE-SEIS '20, page 1–10, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371244. doi: 10.1145/3377815.3381377. URL https://doi.org/10.1145/3377815.3381377. [p7, 11]

F. Morandat, B. Hill, L. Osvald, and J. Vitek. Evaluating the Design of the R Language. In J. Noble, editor, *ECOOP 2012 – Object-Oriented Programming*, Lecture Notes in Computer Science, pages 104–131, Berlin, Heidelberg, 2012. Springer. ISBN 978-3-642-31057-7. doi: 10.1007/978-3-642-31057-7_6. [p6, 9]

B. Muenchen. R's growth continues to accelerate, 2017. URL https://www.r-bloggers.com/rs-growth-continues-to-accelerate/. [p6]

G. Pinto, I. Wiese, and L. F. Dias. How do scientists develop scientific software? an external replication. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 582–591, 2018. doi: 10.1109/SANER.2018.8330263. [p6, 9, 10, 11, 12]

K. Plakidas, D. Schall, and U. Zdun. Evolution of the r software ecosystem: Metrics, relationships, and their impact on qualities. *Journal of Systems and Software*, 132:119–146, 2017. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2017.06.095. URL https://www.sciencedirect.com/science/article/pii/S0164121217301371. [p7]

A. Potdar and E. Shihab. An Exploratory Study on Self-Admitted Technical Debt. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 91–100, 2014. doi: 10.1109/ICSME.2014.31. [p7, 11]

K. Ram, C. Boettiger, S. Chamberlain, N. Ross, M. Salmon, and S. Butland. A community of practice around peer review for long-term research software sustainability. *Computing in Science Engineering*, 21(2):59–65, 2019. doi: 10.1109/MCSE.2018.2882753. [p9]

rOpenSci, B. Anderson, S. Chamberlain, L. DeCicco, J. Gustavsen, A. Krystalli, M. Lepore, L. Mullen, K. Ram, N. Ross, M. Salmon, and M. Vidoni. rOpenSci Packages: Development, Maintenance, and Peer Review, Feb. 2021. URL https://doi.org/10.5281/zenodo.4554776. [p9]

M. Rosado de Souza, R. Haines, M. Vigo, and C. Jay. What makes research software sustainable? an interview study with research software engineers. In *2019 IEEE/ACM 12th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 135–138, 2019. doi: 10.1109/CHASE.2019.00039. [p6]

S. Russell, T. D. Bennett, and D. Ghosh. Software engineering principles to improve quality and performance of R software. *PeerJ Computer Science*, 5:e175, Feb. 2019. ISSN 2376-5992. doi: 10.7717/peerj-cs.175. Publisher: PeerJ Inc. [p9, 10]

G. Samarthyam, M. Muralidharan, and R. K. Anna. *Understanding Test Debt*, pages 1–17. Springer Singapore, Singapore, 2017. ISBN 978-981-10-1415-4. doi: 10.1007/978-981-10-1415-4_1. URL https://doi.org/10.1007/978-981-10-1415-4_1. [p7, 8]

I. Sommerville. *Software Engineering*. Pearson, 10th edition, 2015. ISBN 0133943038. [p11]

N. Thieme. R generation. *Significance*, 15(4):14–19, 2018. doi: 10.1111/j.1740-9713.2018.01169.x. [p6]

TIOBE. Tiobe index - the software quality company, 2020. URL https://www.tiobe.com/tiobe-index/. [p6]

M. Vidoni. Evaluating unit testing practices in r packages. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, pages 1–12, Madrid, Spain, 2021a. IEEE. [p6, 8, 10, 11, 12]

M. Vidoni. Self-Admitted Technical Debt in R Packages: An Exploratory Study. In *2021 International Conference on Mining Software Repositories*, pages 1–11, Madrid, Spain, 2021b. IEEE. [p7, 9, 10]

I. Wiese, I. Polato, and G. Pinto. Naming the pain in developing scientific software. *IEEE Software*, 37(4):75–82, 2020. doi: 10.1109/MS.2019.2899838. [p10, 11, 12]

Melina Vidoni
Australian National University, School of Computing
Canberra, Australia
0000-0002-4099-1430
melina.vidoni@anu.edu.au