Originally published in: Lüngen, Harald/Kupietz, Marc/Bański, Piotr/Barbaresi, Adrien/Clematide, Simon/Pisetta, Ines (Eds.): Proceedings of the Workshop on Challenges in the Management of Large Corpora (CMLC-9) 2021. Limerick, 12 July 2021 (Online-Event). – Mannheim: Leibniz-Institut für Deutsche Sprache, 2021. Pp. 20-26.
DOI: *https://doi.org/10.14618/ids-pub-10467*

Lessons learned in Quality Management for Online Research Software Tools in Linguistics

Nils Diewald and **Eliza Margaretha** and **Marc Kupietz**
Leibniz Institute For The German Language, Mannheim, Germany
{diewald|margaretha|kupietz}@ids-mannheim.de

Abstract

In this paper, we present our experiences and decisions in dealing with challenges in developing, maintaining and operating online research software tools in the field of linguistics. In particular, we highlight reproducibility, dependability, and security as important aspects of quality management – taking into account the special circumstances in which research software is usually created.

1 Introduction

Research data and its management has been very much in the focus of linguistics and related disciplines in the digital humanities over the last 15 years. Although research tools were often mentioned in this context, they have played a subordinate role in terms of development, maintenance, operations, and requirements such as management plans. This discrepancy is probably unfavourable in general, however, it is especially problematic in the case of research based on language data, since there the data often cannot be used, analyzed or interpreted at all without specialized research software. In addition, in the software software, methodology and implementation are often intertwined, so that errors affect findings in ways that are not immediately apparent and often hard to reproduce. Furthermore, there are general problems with software developed in academic contexts, such as lack of training, lack of community support, lack of reputational incentives (Cohen et al., 2021), and lack of structures for sustainable maintenance and operation. Hence we focus on ensuring reproducibility, dependability, and security regarding quality management.

In the following, we report on our experiences with these challenges in operating and further developing the corpus query and analysis platforms KorAP and Cosmas II, which are used, among others, by the more than 40,000 users of the German Reference Corpus DeReKo (Kupietz et al., 2010, 2018) for a broad spectrum of corpus linguistic research.

2 Online Research Software Tools

In this article we distinguish between *research software* in general and *research software tools* in particular. While research software can be the result of a research project (especially in the field of computer science) or specific to the achievement of a single research result (see Hasselbring et al., 2020), we understand research software tools to be products that can be used to conduct or support research, i.e. which can be reused in a research area to a wide field of research objects and therefore requires constant *maintenance*[1]. By focusing on online tools, we refer to platforms that provide remote access to research data, and in the field of corpus linguistics, we are referring in particular to corpora that may be exclusively accessible via these platforms (e.g. for legal reasons, as is often the case for contemporary corpora; see Kupietz et al., 2010). These tools therefore require not only to be maintained but also to be *operated*.[2]

An important aspect of the use of computer-assisted methods in research is the reproducibility of results. Especially in the case of quantitative studies, not only the data are relevant for the reproducibility by different teams[3], but also the software used for the data analysis. Reproducibility including research software is a challenge (see Stodden and Miguez, 2014, regarding *best practices* on reproducibility in computational sciences), particularly in the case of online tools, since neither the underlying data nor the software used are usually in a state that is easy for the user to preserve and

[1] For a taxonomy on and criteria for research software, see https://rseng.github.io/rseng/.

[2] All other aspects listed here also predominantly apply to locally operated research software tools.

DOI: *https://doi.org/10.14618/ids-pub-10469*

bundle with the research findings.

An additional aspect when providing online tools for (scientific) work is that operation must be guaranteed to be secure, uninterrupted and failure-free (in the best case), since on the one hand users depend on them for their research and on the other hand the validity of the results depend on their correctness.

3 Reproducibility

In order to make scientific studies with computer-assisted methods reproducible, not only prerequisites have to be fulfilled by the study design and the underlying data, but also the software should be designed and developed in such a way that it can be run on other systems in the form used for the initial study. This poses numerous challenges (Ivie and Thain, 2018), especially for online tools, but first and foremost, it requires

1. **licensing** that is as open as feasible,
2. transparent **versioning** of the software, and
3. a high degree of **portability**, so the software can be run independent of its environment and time.

It should be noted, of course, that reproducibility can basically only be achieved according to the *best effort* principle. Full control over and complete documentation of the environment can rarely be guaranteed (i.e. full control over the hardware, the operating system, the compiler or interpreter used, etc.).

3.1 Licensing

To enable autonomous reproduction of a study using computer-assisted methods, the software used must be accessible for everyone without restriction in the best case. In order for the implementation to be completely transparent, publication as open source is essential (Hasselbring et al., 2020). This not only helps with reproducibility, but can also reveal problems in the analysis, originating from errors in the software used (Goldacre et al., 2019).

The decisive criterion in the selection of software licences for the publication of KorAP modules was to restrict their use as little as possible and not without reason. Therefore, we have published most of the KorAP modules under the very liberal *BSD 2-clause license*[4] as open-source software on our Gerrit server[5] and on GitHub[6]. The biggest concern we had with our license choice was that the BSD licence does not exclude the subsequent removal of externally developed code. Our compromise solution to this consisted of pointing out in the licence notes (certainly not completely legally secure) that externally contributed code would also automatically fall under the BSD licence. For the case that substantial code parts were contributed by external developers, we also planned to introduce Contribution License Agreements (CLA) independently via corresponding hooks in GitHub and Gerrit. So far, we have not had any bad experiences with our licensing policy.

The decision on Cosmas II licensing was made against opening up the source code in the mid-1990s. Therefore all aspects of reproducibility were in the responsibility of the project owner.

3.2 Versioning

As software continues to be developed, there are differences that may disrupt reproducibility. At times backwards incompatibility is also inevitable. This is where versioning plays an important role. Versioning ensures consistent behaviour of a software by identifying and recording immutable states of a software (i.e. that are called versions) over time. By using version control systems, older versions of a software can be rebuilt. We use Git for versioning KorAP and SVN for Cosmas II. Moreover, we use GitHub for hosting the KorAP Git repository.

A version number or hierarchy (e.g. “1.5.2”) is often used to communicate changes between states. While the different levels of a version number can indicate different forms of change (compare with *semantic versioning*[7]), it is crucial that the behaviour of a released version of software is immutable, and that it can be restored at any time. In the case of KorAP, different components are in play (Diewald et al., 2016), which are operated and released independently of each other. In order to provide users with information about the whole software stack in use, a central API endpoint was designed that returns the individual version numbers of the com-

[3] We follow the terminology by the Association for Computing Machinery (2020). Please note, that the definitions for “reproducibility” and “replicability” were revised in version 1.1; see Plesser (2018) for an overview on the terminology.

[4] Also called *Simplified BSD License* or *FreeBSD License*

[5] https://korap.ids-mannheim.de/gerrit/admin/repos

[6] https://github.com/KorAP

[7] https://semver.org/

ponents involved.

Hashing and tagging can be used for identifying and naming particular changes respectively (Ivie and Thain, 2018). Git uses a hash function to create a unique identifier for each change or commit (Chacon and Straub, 2014). An accurate versioning system would involve the transparent communication of git commit hashes. Tags on these commits are often used to mark releases, but they are not necessarily unique. In KorAP, we take advantage of Git commit hashes as commit references, and Gerrit change-id (see sec. 4.1) to group commits belonging to the same review. Moreover, we use tags to mark releases both in KorAP and Cosmas II.

Releases can be made citable by archiving them in Zenodo, an open access repository for depositing research resources, as supported by GitHub[8]. Zenodo will issue a Digital Object Identifiers (DOI) for each release in a GitHub repository connected to it. We have published recent KorAP releases in Zenodo.

Beside software versioning, it is important to maintain API versioning to support clients using older APIs, especially when there are breaking changes in the newer APIs such as changes in the request or response formats and types. API versioning is commonly achieved by including the API version number in the service URL (i.e. URI versioning), adding a custom header or using the Accept header indicating the API version number. For KorAP API, we support API versioning by including the API version number within its service URL path.

3.3 Portability

Exact repetition of a computer-assisted scientific study would require "building the same program with the same compiler running on the same hardware and the same operating system" (Ivie and Thain, 2018, p. 63:4), which is rarely possible[9]. In the case of online tools, this is further complicated because the server architecture (both hardware and software) is seldom communicated to make attacks more difficult (see sec. 4). This requirement is even more ambitious to meet if a study is to be reproduced far in the future, when common hardware and software have changed to a great extent.

Therefore we instead aim at a high degree of portability of the system while ensuring equivalence of the final result. As a single criterion for equivalent behaviour, we consider the error-free run of all test suites of the system – including their dependencies (see sec. 4.3). To test the error-free operation in different environments, we use *Continuous Integration* for some components via GitHub (see sec. 4.3). To facilitate full building of the overall system locally, we provide both Vagrant[10] and Docker[11] files as our way to enable a virtualization approach (Howe, 2012).

At the beginning of the development of COSMAS II (Bodmer, 1996), the only requirement in terms of portability was the use of GNU-C, so it was usually necessary to access the existing environment to reproduce behaviour.[12]

3.4 Replicability

To additionally enable replicability of a computer-assisted study (i.e. re-implementation of the design by a different team using methods developed independently; see Footnote 3), further detailed documentation of the methods used is necessary. In the case of research software tools, this is sometimes part of the official documentation and thus does not require repeated explanation in publications. In many cases, such as the use of collocation measures, independently implemented methods already allow for the replicability of results (at least from the software point of view).

In KorAP, to facilitate replicability of studies, different query languages were implemented to allow comparison of results across multiple corpus analysis platforms. The use of virtual corpora enables the replicability of studies with different data bases (for example, considering comparable corpora; Kupietz et al., 2020); APIs, URL-encoded queries and various client libraries should help facilitate this. The functionality of KorAP and Cosmas II is documented in scientific publications, in manuals, in GitHub Readmes and Wikis, and commented in the code.

4 Dependability and Security

Avizienis et al. (2004) provide a taxonomy of dependable and secure computing, whose individual

[8] https://guides.github.com/activities/citable-code/

[9] This may still lead to different results in case of non-deterministic behaviour.

[10] https://github.com/KorAP/KorAP-Vagrant

[11] https://hub.docker.com/u/korap

[12] Only due to multiple migrations of the software, for instance from Solaris to a modern 64-bit Linux architecture, did the aspect of portability come to the front, albeit not in the context of facilitating reproducibility.

parts can be seen as cornerstones of quality management in the provision of software. One definition of dependability is "the ability of a system to avoid service failures that are more frequent or more severe than is acceptable", attributed with the *Availability*, *Reliability*, *Safety*, *Integrity* and *Maintainability* of the system. When taking security concerns into account, the *confidentiality* of the system is another important attribute (Avizienis et al., 2004, ch. 2.3).[13]

4.1 Availability and Reliability

Availability is defined as the "readiness for correct service" and reliability as its "continuity". With respect to security, this means a limitation to authorized actions only.

In order to keep the availability of KorAP at a high level, we use the service monitoring tool Icinga[14], which monitors the availability of the web services themselves and the status of the servers involved in order to be able to recognise emerging problems early on. To indicate planned downtimes, we currently use the start page of KorAP's web interface. In order to also be able to notify API users in the future, a corresponding message is planned for the functions for establishing connections in KorAP's client libraries. A fail-safe server structure with load-balancing and automatic switching between servers is not yet implemented for KorAP. This is also because with limited resources and in the context of research tools, we do not prioritise availability over reliability.

Concerning reliability in corpus linguistic research, in particular, it is a commonplace that interesting corpus findings are often initially artefacts of corpus composition and that corpus sampling and analysis cycles should therefore be regarded as an iterative process (Kupietz, 2016). One could add that the findings that remain after the elimination of confounders may also not represent true properties of the language domain under study, but also results of software bugs.

A proven means of reducing software errors is the use of code review (McIntosh et al., 2016), which in the context of research tools can also often take on the role of a classic peer review, at a fine granularity level. Among assisting systems, Gerrit Code Review[15] has become particularly well established in recent years.[16] Gerrit is an open-source team collaboration tool that is typically operated via a web interface. Developers can use it to review others' changes to their source code and comment, improve, augment, approve or reject those changes. It is tightly integrated with Git and can be considered an interface layer on top of Git.

The multiple-eyes principle not only helps to avoid errors and to be able to make design decisions collaboratively, but also ensures that code knowledge is distributed among several people, even if only individuals are responsible for a code base. In this way, personnel failures or absences do not necessarily lead to serious disruptions in operations, maintenance and development.

Admittedly, the use of Gerrit means an increase in the entry threshold, especially if the users are not yet very experienced in dealing with Git either. In addition, the review effort is certainly not to be neglected and the maintenance of Gerrit also involves a certain additional effort. Nevertheless, in view of the direct comparison with projects running in parallel without code review and the advantages already mentioned above, we are convinced that in the case of research tools, the use of a code reviewing system is worthwhile at least from a project size of 2-3 people. The initial and recurring costs incurred are more than made up for by the avoidance of errors and the distributed code knowledge. In our experience, another positive side effect of code reviews in terms of reliability is that commits are typically smaller and that pieces of parallel development strains can be more often combined without conflicting merges. This also increases the readability and traceability of the commit history.

To prevent unauthorized activities, we use integrate the OAuth 2.0 framework (Hardt, 2012) allowing users to grant their applications access to the KorAP APIs[17]. These applications may thus perform operations within the scope of their grants, e.g. searching and retrieving annotations, on behalf of the users.

4.2 Safety, Confidentiality, and Integrity

Safety is defined as the "absence of catastrophic consequences on the user(s) and the environment", confidentiality – as an attribute to security only – as "the absence of unauthorized disclosure of inform-

[13] The following definition quotes are taken from Avizienis et al. (2004).

[14] https://icinga.com/

[15] https://www.gerritcodereview.com/

[16] Gerrit is used by several prominent companies and projects, such as Android, SAP, LibreOffice, Volvo, Skia, TYPO3, ARM, and Wikimedia.

[17] https://github.com/KorAP/Kustvakt/wiki

ation" and integrity as the "absence of improper system alterations", which in regard to security includes unauthorized alterations.

Such security risks are in particular a threat to unmaintained online tools and can bring a very quick end to their operation. When potentially serious security vulnerabilities of an online tool become known and there are no longer any responsible parties, an academic institution usually has no choice but to take the tool offline immediately. Especially since, unlike research data management plans, research software management plans are probably a rarity. But even with tools that are still in development or maintenance, it is not obvious how to identify security problems with reasonable effort. In the case of KorAP, however, the publication of the source code on GitHub already helped us a lot without further ado. GitHub has an integrated security scanner that is enabled by default for public repositories. It detects so called *Common Vulnerabilities and Exposures (CVE)* in used dependencies and notifies the repository owners. Similar code scanner plugins for IDEs can serve as a supplement or alternative. There also seem to be open source approaches for such scanners, but we have no experience with them.

After having received a notification about CVE of a library and there is already an update to this that resolves the vulnerability, a common problem is that the update often also requires the update of other libraries, which again depend on other library updates and so on. This can mean that fixing a security vulnerability in one used library ultimately requires significant work to adapt the code to all the interface and behaviour changes of all the necessarily updated libraries (see sec. 4.3). Doing this quickly without taking the tool offline in the meantime can be a major challenge, even for software that is still under active maintenance.

Unless the use of external libraries is dropped, which however causes other costs and issues (see following section), the only secure option is to update library dependencies regularly and to factor this in from the outset as permanent maintenance costs for the operation of the online tool.

Tools that can help with the continuous updating of library dependencies have proven useful for certain projects (Mirhosseini and Parnin, 2017; Wessel et al., 2018). Dependabot[18], a so called dependency scanner, not only informs about updated dependencies, but also automatically makes merge requests to update, in the case of Java[19], Maven or Gradle project files. Following GitHub's acquisition of Dependabot in May 2019, the feature was added natively to GitHub. In the meantime, there is also an open-source project based on the original Dependabot core, that makes Dependabot available for GitLab.

We have been using Dependabot with GitHub since July 2020 for the KorAP component Kustvakt and have since received an average of 15 update pull requests per month. These update requests are particularly useful and easy to handle with corresponding continuous integration workflows, which can be used to automatically check whether the software is still buildable and operational with the updated library (see 4.3, below).

4.3 Maintainability

Maintainability is the "ability to undergo modifications, and repairs" of a system. Continuous maintenance is necessary not only to fix bugs, to accommodate changes in demand and to address security issues (sec. 4.2), but also to accommodate changes in the behaviour of client or server environments.

Modularity has proved to be useful to simplify a complex system by breaking it down into smaller independent modules or components. Smaller modules are easier to maintain and reuse than a complex system, since they are easier to understand, test and restructure independently of others. KorAP is composed of small independent components, both the service (Diewald et al., 2016) as well as the preprocessing pipeline.

Due to the increased complexity of the system, the maintenance of software dependency trees requires a great deal of effort, so that a constant trade-off must be made between reuse and reimplementation of functions (i.e. *re-inventing the wheel* vs *dependency hell*; see Abdalkareem et al., 2020). In KorAP, we decided to use both approaches, namely reusing functions and libraries as far as possible (as long as indirect dependencies are manageable) and re-implementing only when necessary (e.g. when existing functions are not adequate to cope with new requirements).

All KorAP components are equipped with comprehensive test suites. The test suites help us on the one hand to check new functions for proper beha-

[18] https://dependabot.com/

[19] Besides Java, various other programming languages are also supported, such as JavaScript and Python.

viour, and on the other hand to automatically ensure that program changes do not alter any previous behaviour (cf. Rafi et al., 2012). It is also significant for checking if updated dependencies break or modify the system flow. We also use the automatic detection of test coverage to identify deficiencies and gaps in the test suites. It should be noted that the maintenance of the test suites involves significant extra costs. In some areas, we recently employ additional *fuzzing* techniques (Miller et al., 1990) to test unexpected input to address shortcomings in manually crafted tests.

An important automatable instrument to control the functionality of software in the last instance are continuous integration (CI) tests. We now apply such CI test workflows to the production branches of almost all KorAP components, using *GitHub Actions*[20]. The workflows check if the tool can be built and apply all its available tests partly in different operating system environments (see sec. 3.3). Our CI workflows are usually configured in such a way that they are also automatically apply merge requests submitted via GitHub, so that it is immediately apparent if these affect the functionality of the software [21].

5 Discussion

Developing, maintaining, and operating online research software tools presents numerous challenges including ensuring reproducibility, dependability and security, as it requires knowledge and skills in many different areas. In fact, however, like research software in general, these tools are predominantly developed by individuals from academia – and less with a background in software development (Cohen et al., 2021). In our case, this background is in linguistics.

Cohen et al. (2021) introduce a model of four pillars considered essential to develop sustainable research software in such an environment, with *software development* being only one of the pillars. As additional pillars, they introduce *community* involvement for collaborative problem solving, *training* of researchers in software development techniques, and *policy* development for institutional support. We consider these points to be essential for the development of online research software tools as well, whereby we would add another pillar for maintaining and operating the system.

While the perception of the importance of software for scientific work is growing, development, maintenance, and operation is rarely associated with gaining scientific merit: "many activities are software maintenance – new functionalities or endless bug fixing – and hardly publishable" (Goble, 2014, p. 6). Anzt et al. (2021) therefore propose to accept contributions to open source projects (so-called "pull requests") as a new form of academic contributions to conferences[22], in order to increase the motivation to participate in the development of research software tools, which is beneficial for the wider scientific community.

Our remarks regarding quality management to ensure reproducibility, dependability and security of online research software tools should in no way be misunderstood as *best practices*. They are only meant to reflect our choices and experiences in these areas running the corpus analysis platforms KorAP and Cosmas II, whereby all these decisions were based on a cost-benefit calculation. Especially when creating research software, the effort to run a practicable quality management is often not compatible with the circumstances. We are however convinced that the aforementioned aspects are worth to be considered in the development, operation, and maintenance – maybe even in the planning – of online research software tools in general.

Acknowledgements

We thank our four anonymous reviewers for helpful comments on earlier drafts of the manuscript.

References

Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: an empirical case study on npm and PyPI. *Empirical Software Engineering (EMSE)*, 25:1168–1204.

Hartwig Anzt, Eileen Kuehn, and Goran Flegar. 2021. Crediting pull requests to open source research software as an academic contribution. *Journal of Computational Science*, 49:101278.

Association for Computing Machinery. 2020. Artifact review and badging version 1.1. Technical report, Association for Computing Machinery.

[20] https://docs.github.com/en/actions

[21] Consisting of multiple components developed in various programming languages and frameworks having distinct formats and structures, KorAP is too complex and not suitable for uniform code and comment styles and conventions.

[22] With a focus on High Performance Computing.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.*, 1(1):11–33.

Franck Bodmer. 1996. Aspekte der Abfragekomponente von COSMAS II. *LDV-INFO*, 8:142–155.

Scott Chacon and Ben Straub. 2014. *Pro Git*, 2nd edition. Apress, USA.

Jeremy Cohen, Daniel S. Katz, Michelle Barker, Neil Chue Hong, Robert Haines, and Caroline Jay. 2021. The four pillars of research software engineering. *IEEE Software*, 38(1):97–105.

Nils Diewald, Michael Hanl, Eliza Margaretha, Joachim Bingel, Marc Kupietz, Piotr Banski, and Andreas Witt. 2016. KorAP architecture – Diving in the deep sea of corpus data. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, pages 3586–3591, Portorož/Paris. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2016/pdf/243_Paper.pdf.

Carole Goble. 2014. Better Software, Better Research. *IEEE Internet Computing*, 18(5):4–8. Conference Name: IEEE Internet Computing.

Ben Goldacre, Caroline E. Morton, and Nicholas J. DeVito. 2019. Why researchers should share their analytic code. *BMJ (Clinical research ed.)*, 367:l6365.

Dick Hardt. 2012. The OAuth 2.0 Authorization Framework. RFC 6749.

Wilhelm Hasselbring, Leslie Carr, Simon Hettrick, Heather Packer, and Thanassis Tiropanis. 2020. Open source research software. *Computer*, pages 84–88.

Bill Howe. 2012. Virtual appliances, cloud computing, and reproducible research. *Computing in Science Engineering*, 14(4):36–41.

Peter Ivie and Douglas Thain. 2018. Reproducibility in scientific computing. *ACM Computing Surveys*, 51:1–36.

Marc Kupietz. 2016. Constructing a corpus. In Philip Durkin, editor, *The Oxford Handbook of Lexicography*, pages 62 – 75. Oxford University Press, Oxford.

Marc Kupietz, Cyril Belica, Holger Keibel, and Andreas Witt. 2010. The German Reference Corpus DeReKo: A Primordial Sample for Linguistic Research. In *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10)*, pages 1848–1854, Valletta/Paris. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2010/pdf/414_Paper.pdf.

Marc Kupietz, Nils Diewald, Beata Trawiński, Ruxandra Cosma, Dan Cristea, Dan Tufiş, Tamás Váradi, and Angelika Wöllstein. 2020. Recent developments in the European Reference Corpus (EuReCo). In *Translating and Comparing Languages: Corpus-based Insights*, Corpora and Language in Use, pages 257–273, Louvain-la-Neuve. Presses universitaires de Louvain.

Marc Kupietz, Harald Lüngen, Paweł Kamocki, and Andreas Witt. 2018. The German Reference Corpus DeReKo: New Developments – New Opportunities. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC'18)*, pages 4353–4360, Miyazaki/Paris. European Language Resources Association (ELRA). http://www.lrec-conf.org/proceedings/lrec2018/pdf/737.pdf.

Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. 2016. An empirical study of the impact of modern code review practices on software quality. *Empirical Software Engineering*, 21(5):2146–2189.

Barton P. Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44.

Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 84–94.

Hans Plesser. 2018. Reproducibility vs. replicability: A brief history of a confused terminology. *Frontiers in Neuroinformatics*, 11.

Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42.

Victoria Stodden and Sheila Miguez. 2014. Best practices for computational science: Software infrastructure and environments for reproducible and extensible research. *Journal of Open Research Software*, 2:1–6.

Mairieli Wessel, Bruno Mendes de Souza, Igor Steinmacher, Igor S. Wiese, Ivanilton Polato, Ana Paula Chaves, and Marco A. Gerosa. 2018. The power of bots: Characterizing and understanding bots in oss projects. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW).