IJDC | *Conference Paper*

How Long Can We Build It? Ensuring Usability of a Scientific Code Base

Klaus Rechert
University of Freiburg

Jurek Oberhauser
University of Freiburg

Rafael Gieschke
University of Freiburg

Abstract

Software and in particular source code became an important component of scientific publications and henceforth is now subject of research data management. Maintaining source code such that it remains a usable and a valuable scientific contribution is and remains a huge task. Not all code contributions can be actively maintained forever. Eventually, there will be a significant backlog of legacy source-code. In this article we analyse the requirements for applying the concept of long-term reusability to source code. We use simple case study to identify gaps and provide a technical infrastructure based on emulator to support automated builds of historic software in form of source code.

Received 08 April 2021 ~ *Accepted* 19 April 2021

Correspondence should be addressed to Klaus Rechert, Hermann Herder Str 10, 79104 Freiburg, Germany. Email: klaus.rechert@rz.uni-freiburg.de

An earlier version of this paper was presented at the 16th International Digital Curation Conference.

The *International Journal of Digital Curation* is an international journal committed to scholarly excellence and dedicated to the advancement of digital curation across a wide range of sectors. The IJDC is published by the University of Edinburgh on behalf of the Digital Curation Centre. ISSN: 1746-8256. URL: http://www.ijdc.net/

Introduction

Software and software citation has become an important ingredient for research data publications and henceforth research data management. The growing importance of software developed as part of research projects makes it an integral part of a research paper, necessary for review, validation, reuse and collaboration (Katz et al., 2021; Smith et al., 2016). The importance of software as part of a scientific publication is increasingly recognized by researchers, funders and publishers (e.g., Barker et al., 2020).

In particular, in the context of scientific publications, cited software needs to remain findable, accessible and usable, and thus needs to be preserved and managed over time. Concepts that are applied to research data, e.g., FAIR data principles (Wilkinson et al., 2016) need to be mirrored for software too (e.g., FAIR4RS[1], Lamprecht et al., 2020; Hasselbring et al., 2021). To support these practices, guides (e.g., Chue Hong et al., 2019) and polices (Monteil et al., 2020) have been published as well as usable infrastructure for archiving (scientific) software and source code became available. For instance, Software Heritage, a non-profit organization, offers, together with other source code preservation services, a self-archiving option[2], which allows individuals to submit a code repository for archiving (Di Cosimo, 2020).

The currently available infrastructure covers well most of the proposed FAIR principles for software, however, the *reusable* property is still a difficult one and we argue, there is a gap to be filled – in particular, ensuring long-term re-use of software is a growing concern.

Currently, a common recommendation is to focus on community engagement, software design and good software engineering (Collberg and Proebsting, 2016) and thus, sharing the maintenance burden, e.g., by porting code to new platforms and upgrading or replacing software and hardware dependencies (Katz and McHenry, 2021). Mimicking the successful open-source ecosystem of the past 30+ years seems a viable approach for popular software with a significant user community. However, scientific communities are scattered and considerably smaller, creating specialized scientific software only of interest for a small number of specialists. But even rather popular software will be eventually abandoned by its community, moving on with scientific and technical progress.

In this article we analyze the requirements for applying the concept of long-term reusability to source code, a very common form of scientific software. We use a technical implementation to identify gaps in current practices.

Reusability of Scientific Source Code

Software is more vulnerable to a technical life-cycle than research data, as it poses more complex requirements on its technical environment. Data, for instance, might be re-used in the future with then contemporary software, or if its format and structure is well documented and understood, it may be migrated to a then contemporary format. In contrast, software, specifically in form of binaries require a suitable runtime platform,

1 FAIR 4 Research Software Working Group: https://rd-alliance.org/groups/fair-research-software-fair4rs-wg

2 Software Heritage: https://save.softwareheritage.org

e.g., a specific CPU, implementing a specific instruction set architecture[3] (ISA), to execute machine instruction, but additionally a specific operating system (and version) as well as a number of compatible libraries. For long-term usage of software in form of executable binaries, virtualization (Larsen et al., 2012) and eventually emulation can be a suitable solution when accompanied with a comprehensive collection of software dependencies. There are frameworks and initiatives fostering the use of emulation and various aspects of structured software collection (Rechert et al., 2020; Rosenthal, 2015).[4]

Software developed in the context of research projects is usually published and shared in the form of source code to support reproduction. In contrast to binaries, source code differs significantly with respect to future re-use. Source code is available in text form and remains human readable over time. It contains exact information about how a specific program has been made, usually supported through *inline* comments made by the developers, important for scientific transparency but also to allow others to improve and develop the code further.

For some re-use purposes, accessing source code in text form is sufficient, i.e., to extract parameters or other specific aspects of a scientific model. However, non-trivial programs easily reach ten thousand or more lines of code and thus, reaching a complexity that is difficult to follow and especially very difficult to understand and predicting its behavior in context of a complex task. In order to use software (e.g., to process large data sets) in form of source code, it either needs an appropriate interpreter (e.g., for scripts written in Perl or Python) or needs to be compiled to a native ISA (e.g., in case of C/C++ source code).

The foreseeable obsolescence of contemporary technology poses an even bigger threat to source code. To build (compile) software from source code, additional software dependencies (e.g., the so called toolchain) are necessary. These dependencies are typically distinct from the binary's dependencies required to run and may consist of various types of libraries, source code (e.g., header files) as well as tools to (pre-)process to code, translate the code into machine instruction and tools to automate and orchestrate this process. Furthermore, the build process itself may be non-trivial and error prone, such that it can be difficult to guarantee a successful build and even more important, that the binary works as intended.

(Re-)Building Software from Source Code

Building software from source code can be a challenging task. For this, build tools are required, e.g., a compiler usually accompanied by tools for pre-processing, dependency analysis, build orchestration and packaging. Additionally, computer programs are built on top of existing functionality, available in form of libraries. These libraries are usually independently developed and maintained. Updated versions of a library may introduce incompatible changes how a function should be called, i.e., the number and kind of arguments the caller needs to pass, but also the function's semantics may change, e.g., the result returned may change structurally or semantically. The same applies to how build-tools process code and in particular, what is assumed implicitly and what not.

For instance, even when using a standardized programming language like "C" with a long tradition of carefully maintained open-source tools (here the GNU compiler suite

3 E.g., Intel's x86 architecture is currently one of the most popular instruction set architecture (ISA).

4 Emulation as a Service Infrastructure (EaaSI): https://www.softwarepreservationnetwork.org/emulation-as-a-service-infrastructure/

(GCC)) building ancient code is not trivial, as demonstrated by a simple experiment[5] rebuilding the GCC compiler in version 1.27 released 1988 in a contemporary environment. Even though the code itself was syntactically quite compatible with contemporary compilers and the experiment was successful in the end, still modifications of the original source code (even though in a small number) were required and thus, the availability of sufficient knowledge on the specific programming language and other build process specifics. For the purpose of improved reproducibility, ideally a successfully built software should reference then the respective build environment, which should be captured and archived in addition to the author's narrative to prevent another future trial and error approach.

Hence, a further problem is to document and archive not only a program's *runtime* dependencies but also its build environment. Source code is usually published with a "README" or similarly named file with instructions to build the code, accompanied with some hints on required build and runtime dependencies – typically libraries. These descriptions often refer to systems available when the software was released, i.e., describing package names for different operating systems or Linux distributions but usually without explicit versioning. If version information is provided, a minimum version is referenced but usually not the last compatible version. More importantly, these instructions are written for humans and are not machine actionable. The Software Heritage archive recommends adding additional, structured metadata when archiving source code (Di Cosimo, 2020). Current practice, e.g., using the proposed CodeMeta schema (Jones at al., 2017) aims mostly towards improving findability but not yet supporting a re-build of source code. Future versions of archived versions may close this gap, however, as an intermediate solution and in particular for already archived and now unmaintained code, maintain the ability to recreate the "original" build environments, i.e., keeping the original Linux environment accessible and usable, is already a good starting point, not only for a "basic" re-build, but also as a starting point to improve metadata and documentation as well as a starting point to explore re-use of the software in other scenarios.

Rebuilding a program from source and re-run it without an error does not always imply successful reproduction, e.g., of scientific result. Benureau and Rougier (2018) provide more nuanced criteria for reproducibility of scientific code contribution, ranging from re-runnable (R1) over reproducible (R3) to being fully replicable (R5). They show by example that a small, subtle variation of a Python version (3.2 vs. 3.3) will lead to different outcomes, even though the code itself has been executed successfully. With growing complexity of scientific workflows, the complexity of the dependency chain and the number of different libraries and tools used will also grow. To ensure reproducible or even fully replicable results of software-based process, being able to instantiate all software components in a defined version is an important precondition. These dependencies might not always be available in the exact specified version in a pre-compiled (binary) archived version and therefore, may have to be compiled from source code.

Finally, the most difficult task when re-building source code is to verify that a binary program originates from a given source code snapshot, i.e., one can reproduce the exact same binary for instance, by comparing the hash values of the binary sample and the binary built from source. For instance, initiatives like the Reproducible Builds project[6] aim to develop guidelines and infrastructure for developers to ensure reproducibility of

5 See: https://web.archive.org/web/20210124204244/ and https://miyuki.github.io/2017/10/04/gcc-archaeology-1.html

6 Reproducible Builds project: https://reproducible-builds.org/

their builds to proof integrity and provenance of a published binary. This is not only important for security research, e.g., that a distributed binary does not contain additional (malicious) code and is a result of (peer-)reviewed source code. Reproducible builds are also important to allow traceability of development and binary management. If a binary is lost or it has not been preserved, it can be rebuilt exactly as it was. Additionally, different stages of the development cycle can be re-staged, tracing regressions, bugs or major improvements. Thus, deterministic reproduction of research results also relies on a deterministic software toolchain, which allows for introspection and links source code and program behavior. While general concepts and solutions for reproducible builds are outside of the scope of this paper, to ensure such properties in the long-term, not only the necessary build environment and tools need to be preserved and be re-usable, it is also necessary to ensure a stable and controlled emulated build environment.

Automating Historic Builds – A Case Study

Regardless of how the code will be (re-)used, infrastructure and automation are crucial to ensure that a large code-base can be used and to mitigate a widening knowledge gap over time.

To illustrate how workflows for rebuilding and re-using source code can be implemented, we chose the simple, aforementioned Python example from Benureau and Rougier (2018):

```
import random
random.seed(1) # RNG initialization
x = 0
walk = []
for i in range(10):
	step = random.choice([-1,+1])
	x += step
	walk.append(x)
	print(walk)
# Saving output to disk
with open('results-R2.txt', 'w') as fd:
	fd.write(str(walk))
```

The code snippet implements a random walk. A random number generator (RNG) chooses ten times either +1 or -1. In order to make this code deterministic and long-term reproducible, the random generator is initialized with the fixed seed value '1' (line 2). Due to changes in the Python RNG, from version 3.3, the output of the code above is `[-1, -2, -1, -2, -1, 0, 1, 2, 1, 0]` instead of the expected output of `[-1, 0, 1, 0, -1, -2, -1, 0, -1, -2]` prior to that change. In order to replicate scientific code contributions built on prior version of Python, either the code of the contribution needs to be adapted, which may introduce additional problems, or the contribution is executed with the original tool chain. In the following, we demonstrate usage of a specific Python version installed from source code.

Preparations

In order to build an archived software project both the source code needs to be retrieved and a suitable build environment needs to be prepared. For the aforementioned use-case we require a specific Python version. A complete development history of Python is available from the Software Heritage archive in form of an archived Git repository. As a first step, the relevant snapshot or version of the source code has to be identified. Currently it remains a manual task to identify the commit hash, for instance, based on a symbolic version number.

The Software Heritage archive offers a RESTful API[7] to retrieve content in an automated way. It is important to note that the Git commit hashes are globally stable identifiers, i.e., the commit hashes found on public Git repositories hosted by providers such as GitHub, GitLab, etc., remain valid if retrieved through the Software Heritage API. For this case study, the relevant commit hash is *ccc4ffe7a1e2ae151959446722bf1b58834f5e9f* (the last commit for version 3.1)[8].

The EaaS(I) framework has been extended with a plug-in to retrieve source code from the Software Heritage archive by commit hash[9]. An alternative solution could be accessing the archive through a FUSE filesystem, which became recently available (Allançon, Pietri, and Zacchiroli, 2021) and is currently evaluated. The result is a snapshot of the source-tree as a compressed archive, ready for further processing.

The second preparatory step is to identity a suitable build environment. If the build instructions are not machine-actionable or they contain no detailed information on build dependencies (and versions thereof), a viable approach is to provide the user with a basic build environment from that time as an interactive emulation session (e.g., a preinstalled historic Linux distribution). Depending on the chosen environment, additional software, libraries or build tools are required. This step can be carried out manually within an interactive session allowing users to experiment with source-code in a historic software environment. Installing software dependencies requires a software archive, e.g., in case of a packaged-based distribution like Debian, a working package repository. If pre-packaged build- and runtime-dependencies are not available, a re-build from source code might also be necessary, adding an additional recursive workflow step.

In order to support stable, automated re-builds, references to external repositories have to be carefully managed and maintained. For instance, Debian distributions prior to release 8 (2015-2018) are not available anymore under the initially (pre-configured through the installation media) configured URLs but are moved to a dedicated archive.[10] Similar archives exist for other Linux distributions. If archived prepackaged software dependencies are available (and the use of an archived version is necessary), these are usually not directly usable, but have to be manually configured in the build system. In order to ensure automated (re-)builds and to reduce maintenance overhead of archived build environments, the EaaS framework offers a "managed" network, i.e., emulated machines in such a network are isolated from the "live" internet and access to external network resources are managed. A dedicated Linux repository proxy service transparently keeps old repository URLs working, while serving the requests for contemporary archives. These mappings are generic and apply to any available

7 Software Heritage API: https://archive.softwareheritage.org/api/1/

8 This URL points to the last v3.1 commit: https://archive.softwareheritage.org/browse/revision/ccc4ffe7a1e2ae151959446722bf1b58834f5e9f/?origin_url=https://github.com/python/cpython

9 See: https://github.com/Aeolic/swh-downloader

10 Debian archive server: http://archive.debian.org/debian/

emulated environment, such that installed Linux systems do not have to be updated every time package repositories are deprecated or moved. Additionally, these mappings can also be used for other archived network resources, e.g., pip, CPAN, npm, etc.

The preparation workflow is primarily designed for initial setup, experiments and quality assurance and eventually should yield re-usable environments. Once a system has been configured, the user has two options to preserve the ready-made build environment. One option is to preserve a snapshot of the emulated machine's disk with all required dependencies installed and configured. This pragmatic option, however, limits re-use of the build environment to a specific case. Re-use in similar cases requires additional manual adaption and may require individual maintenance in the future. A second, more re-useable solution, would keep the base environment unchanged and maintain all object-specific installation and configuration steps as an executable shell script or, if possible, in the form of abstract machine-actionable metadata. Ideally, the resulting *recipe* can be executed in different environments, e.g., to find the latest compatible version.

For the purposes of this example, we chose a plain Debian 9 (released in 2017) with Python v3.5.3 installed as well as additional dependencies, e.g., a C/C++ compiler and essential build utilities.

Build Automation

Starting with a reference to a source code snapshot (e.g., Software Heritage commit hash) and an identified build environment, ideally, all build steps are orchestrated and executed by the framework without any additional user-interaction in a similar way of automated builds offered by today's public code repositories (i.e., continuous integration (CI) jobs). To support this, two technical problems are to be solved: firstly, to "inject" data into the emulated machine and, secondly, to reliably "autostart" the build process. Both functionalities should be as generic as possible, such that they can be applied to a wide range of different emulated systems. However, both are tied (at least to some degree) to the guest operating system.

In case of injecting data into an emulated system, one can consider multiple options. A quite generic option to transport data into a guest system are CD-ROMs, as their filesystem (ISO9660) is widely supported. However, in order to build code, the filesystem usually needs to be writeable, thus leaving the task of organizing the data within the guest environment to the user's build script. Another option could be to create a secondary virtual hard disk on the fly and format the disk with a common, widely supported file system (e.g., FAT32). But this solution also has similar drawbacks because the secondary disk has to be detected by the guest operating system and *mounted* as user-visible directory (Linux) or drive (Windows). In both cases, the user has to anticipate the guest operating system's behavior and has to prepare the build recipe accordingly. Additionally, this complicates the task of automated executions within an emulated guest. While the current EaaS(I) framework supports both ways, we have implemented a third option to improve deterministic behavior and re-use of build recipes with different build environments. For this, the emulated system's main disk (e.g., the disk and partition the operating systems is installed on) is modified and the user (or the system's metadata) has to specify the target partition, filesystem, and the final destination of the source code on the disk.

Additionally, the user has to provide the recipe, i.e., a script that carries out the actual build, together with additional build parameters (e.g., a result of the preparation process).

Finally, the framework needs to provide a generic and deterministic way for automated runs of the build process in order to support scheduled, non-interactive build-jobs. As a quite generic solution for Linux environments, we utilize the *cron* subsystem, a built-in job scheduler developed in 1975 and still supported in almost any UNIX-like operating system[11], to initiate the build after the guest system has booted. For this, the EaaS(I) framework modifies the crontab of the build environment accordingly.

Any modifications of the build system (source code and build script injection, autostart, etc.) are carried out at execution time and are non-persistent by default. Figure 1 shows the user interface to setup an automated historic build.

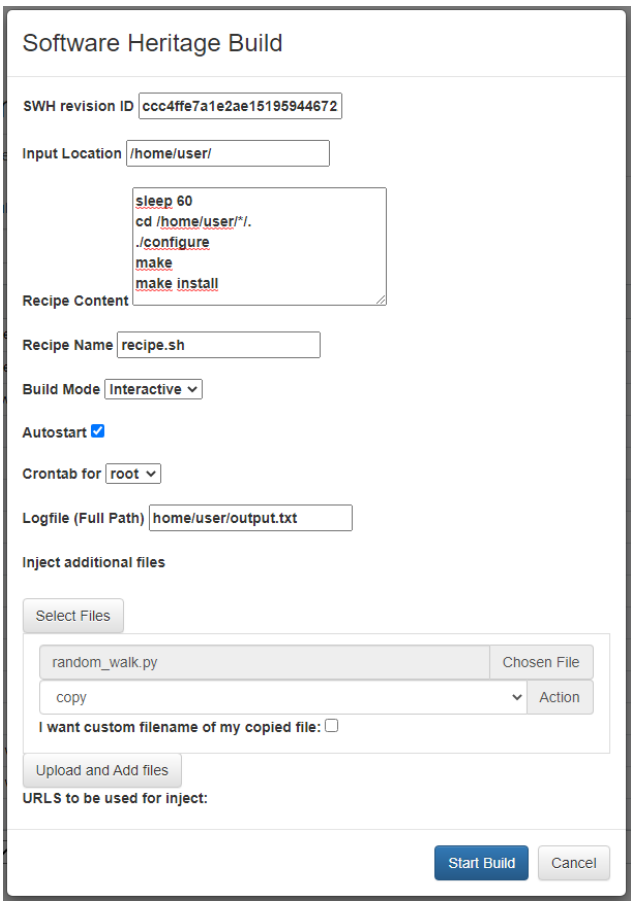

Figure 1. UI to setup an automated historic information.

(Re-)using Build Results

The build process can be partly interactive, allowing users to observe the pre-defined build steps and interact with the system if necessary, e.g., to debug or improve the build recipe. However, the service is designed to run batch jobs without user interaction, orchestrated through a RESTful API.

As a result, two different outcomes are possible, both designed to use the result in other workflows, either in a different workflow (system) or as a pipelined workflow within EaaS(I). One possible result is an emulated machine with a modified disk image which contains the (installed) build result. This machine can be used either for a further build

11 See: https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html

job, e.g., in case of a dependency build, or for the build result to be used with actual data. The alternative is a downloadable archive containing a user-defined result directory. The user's build *recipe* should then copy any desired output to the predefined directory. In case of a scheduled, non-interactive workflow, the result is kept available for the user to download.

The description of the build environment (in this example additional installation steps on top of a default Debian 9 installation) and, optionally, the build image together with the build recipe can be archived and linked to the archived source. This does not only support future re-use of the code itself but can be a valuable starting point for similar code.

Results and Discussion

Maintaining source code such that it remains a usable and valuable scientific contribution is and remains a huge task. Not all code contributions can be actively maintained forever. Eventually, there will be a significant backlog of legacy source-code.

A major step is to maintain access to suitable technical ecosystems. Emulation (and emulation frameworks) provide a basic infrastructure. Still, adapted workflows are required to provide simplified, ideally fully automated, access to source-code, automated builds, and simple re-use of built code with data. We have demonstrated a technical solution for such workflows within the EaaS(I) emulation framework, however, we still lack machine-actionable metadata, both to describe the build requirements and necessary build steps.

Furthermore, in order to ensure long-term builds, all inputs to the build process need to be controlled. For instance, automated, networked package management systems and build systems, which utilize online repositories, e.g., Debian repositories, pip, CPAN, npm, etc., require attention. Firstly, these need to be archived. Additionally, these archives need to be integrated into the staged build environment. The archives may be hosted under different domains, cryptographic keys and certificates may have expired. We have demonstrated the usage of a virtual network environment providing the necessary services. Our next steps will focus on the formalization of this process as structured machine-actionable metadata.

References

Allançon, T., Pietri, A., & Zacchiroli, S. (2021, May). The Software Heritage Filesystem (SwhFS): Integrating Source Code Archival with Development. In ICSE 2021: the 43rd International Conference on Software Engineering.

Barker, M., Katz, D. S., & Gonzalez-Beltran, A. (2020, June 8). Evidence for the importance of research software. Zenodo. doi:10.5281/zenodo.3884311

Benureau, F., & Rougier, N. P. (2018). Re-run, repeat, reproduce, reuse, replicate: Transforming code into scientific contributions. *Frontiers in Neuroinformatics, 11*, 69. doi:10.3389/fninf.2017.00069

Di Cosimo, R. (2020, July). Archiving and referencing source code with Software Heritage. In International Congress on Mathematical Software (pp. 362-373). Springer, Cham.

Chue Hong, N. P., Allen, A., Gonzalez-Beltran, A., de Waard, A., Smith, A. M., Robinson, C., … Pollard, T. (2019, October 15). Software Citation Checklist for Authors (Version 0.9.0). Zenodo. doi:10.5281/zenodo.3479199

Collberg, C. & Proebsting, T. A. (2016). Repeatability in computer systems research. *Communications of the ACM, 59*(3):62–69, February 2016. doi:10.1145/2812803

Hasselbring, W., Carr, L., Hettrick, S., Packer, H., & Tiropanis, T. (2020). From FAIR research data toward FAIR and open research software. *it – Information Technology, 62*(1), 39-47. doi:10.1515/itit-2019-0040

Jones, M. B., Boettiger, C., Cabunoc Mayes, A., Slaughter, P., Gil, Y., Chue Hong, N., & Goble, C. (2017). CodeMeta. Retrieved from http://ssi1.eprints-hosting.org/id/eprint/2/

Katz, D.S., Chue Hong, N.P., Clark, T. et al. (2021). Recognizing the value of software: A software citation guide [version 2; peer review: 2 approved]. F1000Research 2021, 9:1257. doi:10.12688/f1000research.26932.2

Katz, D., & Mchenry, K. (2021). Research Software Sustainability: Lessons Learned at NCSA. In Proceedings of the 54th Hawaii International Conference on System Sciences (p. 7249).

Lamprecht, A. L., Garcia, L., Kuzak, M., Martinez, C., Arcila, R., Martin Del Pico, E., ... & Capella-Gutierrez, S. (2020). Towards FAIR principles for research software. *Data Science, 3*(1), 37-5

Larsen, D. T., Blomer, J., Buncic, R., Charalampidis, I., & Haratyunyan, A. (2012). Long-term preservation of analysis software environment. *Journal of Physics: Conference Series, 396*(3):1–8, December 2012.

Monteil, A., Gonzalez-Beltran, A., Ioannidis, A., Allen, A., Lee, A., Bandrowski, A., ... & Morrell, T. (2020). Nine best practices for research software registries and repositories: A concise guide. arXiv preprint. arXiv:2012.13117.

Rechert K., Stobbe O., Zharkow O., Gieschke R., & Wehrle D. (2020). CITAR – Preserving software-based research. *International Journal of Digital Curation, 15*(1).

Rosenthal, D. S. (2015). Emulation and Virtualization as Preservation Strategies. Retrieved from https://web.stanford.edu/group/lockss/resources/2015-10_Emulation_&_Virtualization_as_Preservation_Strategies.pdf

Smith, A. M., Katz, D. S., Niemeyer, K. E., & FORCE11 Software Citation Working Group. (2016). Software citation principles. *PeerJ Computer Science 2*:e86 doi:10.7717/peerj-cs.86

Wilkinson, M. D., Dumontier, M., Aalbersberg, I. J., Appleton, G., Axton, M., Baak, A., ... & Mons, B. (2016). The FAIR Guiding Principles for scientific data management and stewardship. *Scientific data*, *3*(1), 1-9.