CODE beyond FAIR

Roberto Di Cosmo, Sabrina Granger, Konrad Hinsen, Nicolas Jullien, Daniel Le Berre, Violaine Louvet, Camille Maumet, Clémentine Maurice, Raphaël Monat, Nicolas P. Rougier

CODE beyond FAIR

Roberto Di Cosmo[1,2], Sabrina Granger[6], Konrad Hinsen[3,4], Nicolas Jullien[5], Daniel Le Berre[7], Violaine Louvet[8], Camille Maumet[9], Clémentine Maurice[10], Raphaël Monat[10] & Nicolas P. Rougier[11]

[1]Inria Paris, Paris, France
[2]Université Paris Cité, Paris, France
[3]Centre de Biophysique Moléculaire (CNRS), Orléans, France
[4]Synchrotron SOLEIL, Saint Aubin, France
[5]IMT Atlantique, Brest, France
[6]Inria Lyon, Lyon, France
[7]CRIL, Université d'Artois, Lens, France
[8]Laboratoire Jean Kuntzmann, Grenoble, France
[9]Inria, Univ Rennes, CNRS, Inserm, Rennes, France
[10]Univ. Lille, CNRS, Inria, UMR 9189 CRIStAL, Lille, France
[11]Centre Inria de l'université de Bordeaux, Bordeaux, France
Corresponding Author: Nicolas.Rougier@inria.fr

Abstract

FAIR principles are a set of guidelines aiming at simplifying the distribution of scientific *data* to enhance reuse and reproducibility. This article focuses on *research software*, which significantly differs from data through its living nature, and its relationship with free and open-source software. Based on the second French plan for Open Science, we provide a tiered roadmap to improve the state of research software, which is inclusive to all stakeholders in the research software ecosystem: scientific staff, but also institutions, funders, libraries and publishers.

Contents

1 Introduction

Software diversity. Software plays both a central and critical role in almost every aspect of modern science. It can be used to process data, perform analyses, model complex phenomena, drive various apparatus, design surveys, setup experiments, write documents, and it can itself be a topic of research. This list has dramatically expanded during the last few decades, to the point that it would be now extremely difficult or impossible to conduct research without the proper software. These software can take many different forms, ranging from a full fledged binary executable to a mere script of only a few lines. If we were to scrutinize more closely all these software, we would discover an ever greater variety. According to the historical encyclopaedia of programming languages, there exist approximately 9,000 programming languages. Most of them are hardly used outside specific niches and of course, some languages are dominant. To name just a few, we can mention C/C++, Fortran, Matlab, Python, or R that seem to dominate the contemporary scientific landscape. If we now consider the cross product of language, form, usage and epistemic diversity, we get a small glimpse of the many forms of scientific software and their immense diversity.

A living nature. This diversity is comparable to some extent to research diversity. However, unlike many other scientific objects (research papers & data), software packages can be complex living entities that are continuously evolving under the direct action of core developers and various contributors [16]. This means that there does not exist something like a single final state that could be considered the *version of record* to be archived once and for all for later reuse. It is actually not rare to have many versions of the same software (e.g. stable, unstable, latest, deprecated, etc.) that co-exist for some specific reasons. Software evolves and continues to do so as long as there is one person willing to contribute some code, tests, documentation, bug reports or even simple ideas. This mutable nature makes it a quite singular object in the research landscape, requiring more a *record of versions* than the *version of record* which is central for publications. Furthermore, software cannot be easily separated from its biotope, which corresponds to the operating systems it runs on, taking advantage of the huge stack of software libraries that possess their very own lives. No environment is like another such that the smallest change can have dramatic effects on the re-usability, or worse, on the correctness of a piece of software. This makes preserving software while maintaining its functionalities quite a tricky operation.

A porous frontier. Software development is less than a century old, but it has already undergone a series of dramatic changes, one of which is the explosive growth of *open source*, that has taken by storm the software industry, traditionally based on *closed source*. Open source means the source code of a piece of software is made freely available for possible modification and redistribution, which is not the case of closed source software that is generally distributed as non-modifiable binary executables. The actual term "open source" has been popularized by Eric S. Raymond and Bruce Perrens at the end of the 1990s [38], even though the free software movement founded by Richard Stallman is more than a decade older [41]. Free/Libre Open-Source Software (FLOSS) has a global scope, with academia being a small part of it since the beginning.

From that perspective, trying to enforce some newly defined principles (even with best intentions) onto a community that has a long tradition of sharing might not only be considered dubious and pedantic, but also misses the opportunity to build on the shoulders of giants. The FLOSS community had to develop from the start good practices on how to find, access, evolve and re-use

software, and has faced for decades issues like governance, recognition and sustainability, such that instead of trying to set our own agenda onto this community, a smarter move would be learn from their experience and adopt and adapt their philosophy to our own needs.

Open Science to the rescue, but. In 2016, Wilkinson et al. [42] published the article "The FAIR Guiding Principles for scientific data management and stewardship" that encouraged academia, industry, funding agencies, and scholarly publishers to endorse a set of principle to promote the reuse of scholarly data. These FAIR principles (for Findability, Accessibility, Interoperability, and Reusability) have since then become highly popular in academia and a majority of stakeholders now defend these principles at various levels. The FAIR principles were clearly designed for research data, where the term *data* was intended to denote precisely *the input for a processing that leads to a result*, hence the strong focus on metadata that is put forward there. In principle, anything can be seen as data for a particular kind of processing: articles are data for Text and Data Mining, DNA strands are data for sequencing, software source code are data for vulnerability scanning tools, personal entertainment habits are data for recommendation algorithms. But if we look closer, it's easy to see that many of these objects are not *just data*, and most of them are *mainly not data*, in the sense that their intended use is not to serve as input for a processing pipeline. As a consequence, looking at these objects as *just data* is unsatisfactory, as it misses the key issues at stake. Software and source code are a clear example of this phenomenon, leading to several efforts to get the focus back on software as *not just data*: Lamprechet et al. [32] published in 2020 a proposal to have dedicated FAIR principles for software, following several discussions and workshops whose assessment can be found in [22]; more recently, the Research Data Alliance (RDA) published a similar set of principles [11] advocating for the improvement of sharing and reuse of research software, recently extended with [40].

All these efforts are laudable and will definitely help to make software a first class citizen in research. And yet, we believe they both evade the core problems of software complexity and fail to provide an actionable implementation path for what is proposed. For example, the idea that "software reads, writes and exchanges data in a way that meets domain-relevant community standards" cannot be always enforced when for example your software depends on a closed-source software whose format cannot be easily changed. Actually, in some dramatic cases, this is the other way around, and we see a nomenclature changed to cope with software misbehavior [6]. Similarly, the website `fair-software.eu` puts forward the need to register our code on a software registry and yet, as researchers or developers, we hardly use such registry when looking for a specific software.

A tentative roadmap. To this end, we propose a comprehensive roadmap towards a standard that is inclusive of all stakeholders in the research software ecosystem. While authors and contributors of research software remain pivotal, it is imperative to recognize the significant roles played by funders, research institutions, publishers, libraries, and policymakers. Building on the recommendations of the Journal of Open Source Software, we suggest that, in particular for authors and contributors, this roadmap must be seen as gradual: while a few steps are foundational and mandatory, others can be seen as recommended or optional. This tiered approach is adopted for several reasons:

1. A too-stringent standard can deter engagement. The tiers are aimed at not discouraging participants who might feel overwhelmed by seemingly unreachable high-level principles.

2. The approach acknowledges the diverse backgrounds of scientist-developers, many of whom may not have formal training in software engineering but are deeply committed to scientific software development [9].

3. It is difficult to impose stringent rules without appropriate recognition of the effort that they require: in the current landscape, with software development still underrated, this can be counterproductive.

4. The level of maturity about software practice in research vary significantly across disciplines and from country to country: a gradual roadmap seems the best way to bring everybody up to speed over time.

Section 2 details the proposed roadmap that can be summarized as Open, Document, Execute, Collaborate, each element representing a key pillar for maximizing the value of research software in the realm of open science (see Figure 1). Even tough we do not necessarily aim to promote a specific acronym that may hinder the actual philosophy, it is to be noticed that CODE summarizes the key points of the roadmap quite well. Section 3 proposes a call for action from the main stakeholders, i.e., institutions, funders, libraries and publishers (see Figure 2).

2 Open, Document, Execute & Collaborate

Figure 1 summarizes the CODE gradual roadmap. The different levels are closely inspired by the ACM badges system [2], where:

- The *mandatory* points correspond to the *Artifact Available* badge, which mainly mandates that the artifact is publicly accessible and archived;

- The *recommended* points of the Document and Execute blocks correspond to the *Artifact Evaluated: Functional* badge, which mandates that the artifact is documented and exercisable with included scripts to generate the results in the paper;

- The *optional* points of the Document and Execute blocks correspond to the *Artifact Evaluated: Reusable* badge, which mandates that the quality of the artifact significantly exceeds minimal functionality. While, due to the varying nature of artifacts, no concrete requirements are made for this badge, we can map points such as describing the API or implementing a test suite.

Note that the Collaborate block has no equivalent in Artifact Evaluation, and therefore in badges, as it focuses in the life of software after the publication of a research article.

OPEN

- **Publish your source code on a public forge** — **mandatory**
- **Save your repository on dedicated archive** — **mandatory**
- **License your code with an open license** — **strongly recommended**
- Declare authorship & rightholders — recommended

DOCUMENT

- Choose meaningful names — recommended
- Comment code — recommended
- Provide examples, notebooks & tutorials — recommended
- Describe API — optional

EXECUTE

- List software & hardware dependencies — recommended
- Provide a computational environment — optional
- Implement a test suite — optional
- Show real-life usage example with expected results — optional

COLLABORATE

- Respond to issues & feature requests — optional
- Describe maintenance, features & support limits — optional
- Explain how to contribute — optional
- Build and animate a community — optional

Figure 1: *Summary of the CODE progressive road map organized in four blocks: Open, Document, Execute and Collaborate. The different levels (mandatory/recommended/optional) are closely inspired by the ACM badges system.*

2.1 Open

For the sake of reproducibility and progress in science, software will heavily benefit from being opened up to the research community, that may need to study and verify it, and wish to modify and reuse it. There are two very different sides to opening research code: one is the technical aspect of making the software accessible to the research community, ensuring this access will stand the test of time; the other is the legal question of choosing a license that supports the appropriate level of sharing and reuse and attributing authors and rightholders.

- **Publish your source code on a public forge** — **mandatory**
- **Save your repository on dedicated archive** — **mandatory**
- **License your code with an open license** — **strongly recommended**
- Declare authorship & rightholders — recommended

Publish. There are many options for making software accessible to the community, but only a few of them are recommended and an even smaller number will pass the test of time. Among the worst options to distribute software are the email option ("code available upon request" [29]) and the personal website option ("code available on my webpage" [19]), both of which are very fragile since researchers can move from one lab to another, invalidating both their email and website. A safer option are forges which are dedicated websites whose goal is precisely to facilitate the dissemination of source code using version control tools such as git. Version control represents a barrier for researchers not yet familiar with these tools, but this barrier is worth overcoming because forges provide some guarantee of durability and offer additional services, such as the preservation of the software's development history, support for collaboration among developers, and interaction between developers and users.

Archive. Forges have become today essential collaborative development platforms, but they are not archives: repositories on forges can be damaged or deleted by their owners, including by accident, and the forges themselves may be closed down, as it happened to two popular forges that were shut down in 2015 (Gitorious) and 2016 (Google code), invalidating all the URLs that link to them. Ensuring the long-term availability of software therefore requires depositing it in an *archive*, whose mission is long term preservation, in addition to publishing it via a forge. As we will explain later, archiving software requires a specific kind of archive, one that is aware of the living nature of software development, and today we have one such archive available: Software Heritage.

License. While most research software is shared publicly with the intention to make it accessible and reusable for all, following the principles of Open Science, very often this intention is not formalised by the explicit addition of a licence into the code or the repository. If the right holder wants to make possible the use of their software, a license that explicit the term and conditions to this use is mandatory. Indeed, in the absence of a license, the right holder keeps exclusive rights (see Choose a license): even if the right holder has no intention to exercise them, users of the code will be legally forbidden from reusing it, reducing its usefulness. Thus, we strongly recommend to provide a license, and preferably an open-source one for the sake of Open Science.

Authorship. The identification of authors and copyright holders is an essential part of opening software, both for its legal aspects and for citing software in journal articles. Authors and copyright holders may be individuals, groups of individuals, a collective entity (development team), an institution, etc. Ultimately, authors and copyright holders are the only ones who can define the conditions of reuse for a software, via a license, so it is important that they can be identified.

2.2 Document

Documentation is important for software in general, but even more so for research software, because it often implements highly specialized methods that readers might not be very familiar with. Good documentation of software is every bit as important as good explanations in a scientific paper.

- Choose meaningful names — recommended
- Comment code — recommended
- Provide examples, notebooks & tutorials — recommended
- Describe API — optional

The audience for software documentation is diverse. It includes users, who need to understand what the software does exactly and how it is used correctly. It also includes users who wish to modify the software for different applications, and who need to be aware in particular of any assumptions built into the software that may not be valid for the application they have in mind. For larger and long-lived software packages, the audience for documentation includes current and future contributors and maintainers. And in preparation for a hopefully near future in which scientific software will be peer reviewed, the audience for documentation includes reviewers, who need to be convinced that the software is appropriate for the use case they are reviewing.

Choose meaningful names. Documenting a code starts with naming variables and functions. When done wisely, these names convey a lot of information to the reader (including the author herself when she looks at her own code some months or years later). For example, naming a function `da` vs `discrete-analysis` makes a real difference in terms of understanding and re-usability. For short scripts, good names can be sufficient to make the code understandable.

Comment code. When names are no longer sufficient, comments added to the code become important. Their goal is not to rephrase the code, but to explain the reasoning that lead to the code. For example, if the code computes the square root of a number, that number must be positive, and a comment may be required to explain why the number is always positive in this specific context. Moreover, non-trivial algorithms require comments that provide a higher-level view of what is going on, to help the reader who might otherwise be lost in the technical details. The description of the variables that have to be provided to a function, with their type and the reason for this type, is another example of a necessary comment.

Provide examples. API documentation is necessary for precision, but not always sufficient to help new users get started, in particular for large APIs. Learning how to use a complex software package from its API documentation is like learning a foreign language from a dictionary and a grammar. Beginners need pedagogical documentation, such as tutorials and documented usage examples.

Document the API. For larger software packages, users cannot be expected to read the code in order to understand what it does and how to use it. Code readability and good comments still matter, for the benefit of contributors, but users require a separate document that focuses on the API (Application Programming Interface) that they interact with. API documentation lists and describes the functions, classes, shell commands, and other entities in the software that users refer

to in their own code that orchestrates the computation at a higher level.

The transition from commented code to a software package with an API documentation is a major one. Users of commented code have to read and understand the code, and adapt it to their own needs by modifying it. Users of a software package with an API expect to learn everything they need to know from the documentation, and don't even attempt to read the code. Much scientific software is in a dangerous intermediate state: there is an API documentation, but it may not be up to date, differ from the actual behavior of the code, or be incomplete, leaving out details that can be crucial for some applications.

2.3 Execute

> One of the core principles of science is reproducibility, meaning that results can be checked by others, so that they can in particular build upon previous research. As software can be used to produce research results, it is important to allow other teams to execute it in order to make the results reproducible.
>
> - List software & hardware dependencies — recommended
> - Provide a computational environment — optional
> - Implement a test suite — optional
> - Show real-life usage example with expected results — optional

Dependency hell. Running software requires a computational environment, which consists of a hardware (the computer) and other software items, called the software's dependencies. If you wish to run a piece of software published by someone else, you must figure out how to construct a suitable environment, by starting with a compatible computer system and installing all the software's dependencies. Unless an explicit list of dependencies is supplied, this task may turn out to be prohibitively difficult or time consuming. This is why we strongly recommend authors to provide a complete and precise description of the dependencies with each published piece of software.

Unfortunately, providing such a description is not always easy. You may be able to figure out the components of the environment that you use yourself, although even that can be a difficult task. But not all the components of your environment are actually required. If you run a piece of software on a desktop computer, there is a good chance that it has a Web browser installed, but it is unlikely that this Web browser is required to run the software under question. It is also likely that the software will work equally well in an environment that has somewhat different versions of the dependencies. These uncertainties make it difficult for software authors to describe the environmental requirements, and even more difficult for software users to set up an environment that conforms to these requirements. In the worst case, users succeed in constructing an environment in which the software works without obvious problems, but produces different results than in its original environment. This is one of the root causes of computational irreproducibility [26].

Recipes and containers for computational environments. One way to help users set up a suitable

environment is to provide either an archived environment in the form of a container image (through Docker or virtual machines), or a recipe for constructing an identical environment for a reproducible software manager, such as Guix or Nix. Both approaches require technical expertise that many researchers do not have, which is why we consider this step optional within the current state of the art. A good overview to get started writing Guix recipes is given in the document *A guide to reproducible research papers*. Such recipes are the technically best way to provide the description of a computational environment: they are small text files that are easy to publish and archive, and they can be easily modified to explore alternative computational environments for the software. In contrast, container images, created and run using container management software such as Docker or Apptainer, are large files that can be used as-is, but not easily modified. rworkflows [39] is an example for providing both an easy and an advanced solution, for statistical and social sciences and related fields.

Unit tests and integration tests. Given the inevitable variability of computational environments, another helpful step is to provide a test suite for the software. A test suite consists of simple example use cases for the software for which the correct results are known and recorded. Executing the test suite verifies that the results are indeed the expected ones, providing the user with reassurance, though not proof, that the software works correctly in its current environment.

Real use cases. A final optional step for helping users to get started with running the software is a collection of usage examples from real-life applications, with the expected results. We have already mentioned examples as documentation, but examples become much more valuable if they are executable. Such examples differ from tests in two ways: first, they are meant to be studied and modified by the user, whereas a test suite is typically executed blindly. Second, they illustrate common usage scenarios of the software, whereas tests tend to focus on simple scenarios and on edge cases that are important for correctness even though they rarely occur in practice.

2.4 Collaborate

We are concerned with open collaboration, as it has been practiced in FOSS communities for decades: a long-term collaboration that anyone interested can join, and which nobody owns in any legal or moral sense. We believe that this form of collaboration is what Open Science should aim for more generally.

- Respond to issues & feature requests — optional
- Describe maintenance, features & support limits — optional
- Explain how to contribute — optional
- Build and animate a community — optional

Traditionally, collaboration in research means that a group of researchers works together on a research project and in the end publishes a joint paper on which everyone who contributed to it is a co-author. Such collaborative research can include the production of software, usually in the form of scripts and notebooks that are specific to a project. That is not the kind of collaboration we are discussing in this section. We focus on *open* collaboration that anyone interested can participate in. Open collaborations are typically long-lived and broad-spectrum efforts, which in the realm of

software means tools of interest to entire communities.

Respond to issues. The first level of open collaboration on a software package is inviting feedback. Are there any problems with the software? Does it behave incorrectly, or is it too difficult to use? Is there functionality that could be usefully added to it? Issue trackers in software forges were designed to collect such feedback and support open discussions about them.

Set limits. The experience with FOSS projects has shown that when a piece of software attracts a critical mass of users, such feedback can easily overwhelm the development team. After all, criticism (even when fully justified) is easy, whereas dealing with it is a lot of work. An important step for open collaboration is thus defining limits and stating them clearly. What level of maintenance and support are the developers able and willing to provide? Which kind of feature requests will be considered, and which will be considered out of scope?

Engage users. The second level of open collaboration is permitting and soliciting not only feedback, but also contributions, be it to the code itself or to its documentation, or to the governance structure of the project. This requires careful thought. Contributions need to be reviewed, which takes time. Contributions that make the code larger or more complex increase the maintenance effort in the future. It is therefore necessary to decide on a policy for handling contributions, define a workflow, and document it publicly. Furthermore, it is good practice to think about *onboarding*, i.e. integrating new contributors into the collaboration. For example, many projects maintain a list of easy problems ("good first issue") that are suitable as entry points to become familiar with the code and with the culture of its community.

Build a community. The third level of open collaboration is to build a community around the software. This involves organizing tutorials, workshops, and hackathons, and mentoring new contributors. For very large projects, it can also involve communicating to a wider public, and establishing relations with decision makers in industry or society. The goal of community building is not growth for growth's sake. Success of a community should rather be evaluated by the satisfaction of its members, its willingness to welcome everyone who wishes to participate, and its integration into a wider scientific ecosystem that ensures epistemic diversity and critical evaluation of each other's work.

3 A call to action

Beside researchers and engineers, there are several stakeholders that have prominent roles to play in order to support scientific software in all its dimensions. Researchers and engineers are the main actors, but they require support and recognition for their work, both of which are still lacking in a large number of countries and institutions. This has been dramatically revealed in the case of the heartbleed bug in 2014[1], and masterfully captured by XKCD #2347. *The OpenSSL Software Foundation typically receives about US$2000 a year in donations* [35] for a piece of code that is supporting 17% of all trusted web servers, and the core team consists of a single person. This example illustrates quite faithfully the current situation for scientific software and the necessity to

[1] see https://en.wikipedia.org/wiki/Heartbleed

INSTITUTIONS

• **Support development**	**mandatory**
• Promote software	recommended
• Build institutional forges	optional

FUNDERS

• **Long term support**	**mandatory**
• Promote reproducibility	recommended
• Facilitate collaborations	optional

LIBRARIES

• **Archive software**	**mandatory**
• **Curate software (metadata)**	**mandatory**
• Build catalogs	recommended

PUBLISHERS

• **Enforce open source**	**mandatory**
• Archive software	recommended
• Review software	optional

Figure 2: *Summary of recommendations for the different stakeholders.*

take action.

3.1 Institutions

The role of institutions for supporting and promoting software is critical and they can act on several different levels.

- **Support development** — **mandatory**
- Promote software — recommended
- Build institutional forges — optional

Software as a pillar of research: the data is in! The fact that software plays a key role *in all fields of modern research* has recently moved from folklore knowledge to well established fact, thanks to the extensive analysis of software mentions in research articles that has been published in early 2023 by the French ministry of research [31, 4]. This analysis of over 160,000 research articles shows the the percentage of articles mentioning explicitly the use of software in their research ranges from 21% in the social sciences to a whopping 63% in earth, ecology, energy and applied biology. This study stresses the importance to develop a clear strategy to properly support the development, use, reuse and recognition of software in academia. It has been materialized in France through the Second National Plan for Open Science published in July 2021 [21], and through the work of the Software college of the French national committee for Open Science, that is tasked to implement it (most of the authors of the article are part of this committee).

Support development by investing in the needed human resources. Nowadays, software development is a prevalent part of scientific work. As such, we advocate for first-class recognition and support of software development activities by institutions. This support is provided by various means. The primary mean is to hire research software engineers, who bring the necessary expertise to develop, maintain and deploy research software. This need is not new: in many countries research engineers working on software have been around for decades, well before the recent grassroot effort that led to the creation of research software engineer societies [2], but it is important to recognise them as a specific body, with skills and requirements of their own. These research software engineers are the backbone of software development process and can help raise awareness among research staff about basic software development practices.

Support development by creating legal frameworks that scientists and engineers can find their way around. A team wishing to publish a piece of software must be able to find out easily who the right holders are and which person to contact for inevitable legal arrangements such as choosing a licence. Ideally, institutions should make it possible for scientists and engineers to publish research software under an Open Source license without imposing a lengthy bureaucratic procedure for obtaining permission.

Promote software. Institutions can help promote software through different actions acknowledging software creation and development as contributions, which can be taken into account during

[2] https://researchsoftware.org/

hiring and promotion campaigns of scientific and engineering staff. The creation of software-related awards can create visibility and awareness on the importance of software development within institutions. France has led the way by creating a national Open Science Research Software Award [20] in 2022, following the Second National Plan for Open Science. These prizes reward projects and teams working to develop and disseminate open source research software, and are presented by the Ministry of Higher Education and Research in a high profile annual ceremony. The award has several categories aimed at showcasing projects according to their scientific and technical dimensions, their capacity to train and lead a community and the quality of their documentation. A full description of the goals, difficulties, approach and lessons learned has been published in [10], providing actionable blueprints for a broader adoption of the concept.

Build better institutional forges. Software forges were created some 25 years ago as a means to facilitate collaboration among teams of developers. With the fast growth of Open Source, and the advent of modern version control tools, some of them have progressively become large scale social networks, with the associate network effects that lead to concentration into a handful of big players. Research software developers face a difficult decision about which forge to choose, based on the target audience and network. The options range from institutional forges (i.e. hosted on institutional servers) to publicly available, general purposes commercial ones (i.e. GitHub, GitLab, BitBucket), with open-source community forges as an alternative if the software fits the requirements of one such community. The report [33] presents a comprehensive description of the issues at stake when building institutional forges, based on a broad analysis of over 70 such forges used in French Higher Education and Research. One key issue is the tension between the need to maintain control over the use of the resources, and the desire to enable contributions with minimal friction from unknown individuals worldwide. Currently, access to most of the analysed forges is available only to the institution's own members, which drastically limit external contributions, and even becomes an obstacle for the automated workflow of journals dedicated to software like JOSS, which requires submissions to be available on a forge which is open to anybody (without registration approval).

It seems that this is the reason why so much software is currently developed on commercial forges. This situation may change if a coordinated effort is put in place to remove the technical and administrative obstacles that make it currently difficult to accept external contributions on institutional forges (e.g. via federation of identities and appropriate access control to the different resources).

3.2 Funders

- **Long-term support** — **mandatory**
- Promote reproducibility — recommended
- Facilitate collaborations — optional

Include maintenance of valuable software in the scope of funding As recently reiterated by Coelho [12], *most of [research] software is designed and built by research groups funded by short-term research grants.* These short-term grants are both beneficial and detrimental to a software. They are

beneficial because researchers can hire people to work on the software they need, and sometimes start to build a community. Detrimental because once the grant is finished, researchers are on their own to maintain the software to the best of their skills. Maintaining a software is difficult and requires advanced skills in software development to deal with software collapse [27]. Funders have a role to play by expanding their granting scheme to include maintenance of existing software with a proven value, and not only development of new tools and systems. The Chan Zuckerberg Initiative has shown one way to do this with its *Essential Open Source Software for Science* program: we believe more research should go into exploring the different mechanisms that can be put into play to balance between funding innovation and supporting maintenance with the necessarily limited resources that are available [5].

Promote reproducibility Being able to reproduce and replicate results is a cornerstone of the scientific approach. However, numerous studies have highlighted in the past years a reproducibility crisis in scientific research [37, 7, 8, 3]. The significant and successful role of funders such as the European Research Council in the promotion of open access publication and open data, two key open science practices, indicates that funders have a key role to play in promoting reproducibility of research software too. Funders should set as a default the publication of research software under free and open-source licenses, with limited exceptions for the cases when keeping it closed is necessary (e.g. security, confidentiality, or when needed for commercial exploitation or industrial collaborations).

Facilitate collaborations We've explained previously that a scientific software depends on a software stack whose components have very diverse origins. These components can be themselves highly specialized scientific software, but they can be also much more generic and serve broader goals. To name but one example, the ubiquitous NumPy library [24] is *the primary array programming library for the Python language* and *has an essential role in research analysis pipelines in fields as diverse as physics, chemistry, astronomy, geoscience, biology, psychology, materials science, engineering, finance and economics*. This library is community driven and its development *still depends heavily on contributions made by graduate students and researchers in their free time*. This situation is problematic because it means a large part of science depends on the free time of a few people. This is the reason why funders need to consider alternative financing models, where, for example, a subset of a grant could be dedicated to the support of a core library or a core developer could be supported through the grant even though her work would be only indirectly linked to the grant.

3.3 Libraries

Sharing software on well established forges is better than using a personal website, but it is important to understand that forges are meant for collaboration, not for archival.

- **Archive software** — **mandatory**
- **Curate software (metadata)** — **mandatory**
- Build catalogs — recommended

Forges are not archives The evolution of functionalities available on forges led some developers to use forges also as distribution platforms, and we see many researchers doing the same, even

though learning to use version control tools like `git` represents a barrier for those not familiar with them. Sharing software on well established forges is better than using a personal website, but it is important to understand that *forges are meant for collaboration, not for archival*. On the one hand, the owner of a project hosted on a forge may *alter, move or remove it at any time* (millions of projects have been removed from GitHub over the past years). To try and mitigate this issue, document archives like Zenodo or Figshare have been adapted to allow researchers to store copies of specific versions of their own source code hosted on GitHub (integration is not provided for other platforms), but they are not software oriented which brings severe limitations (loss of metadata and version control history, no control of granularity, lack of an intrinsic identifier, etc.). On the other hand, and most importantly, *forges make no commitment to long-term maintenance*, independently of the financial clout of their owners: as a reminder of this fact, two popular forges were closed in 2015 (Gitorious) and 2016 (Google code), invalidating all their URLs (over 800.000 according to the Software Heritage archive) that link to them, and more recently the removal of support for the Mercurial version control system from Bitbucket in 2020 invalidated an additional 300.000 URLs.

Software Heritage is an archive The shutdown of such popular and well funded forges was a shock to many, and made clear the need for a long term archive *specifically designed for software* that addresses all the above issues, leading to the creation of Software Heritage[3], a non profit international multi-stakeholder initiative started in 2016 in partnership with UNESCO, which is today the *state of the art solution* to address the long term availability of *all software source code* together with *its full development history* [15, 1]. It proactively harvests code hosting and distribution platforms, so that a lot of research software is already archived, and it also provides seamless mechanisms to trigger archival on demand. As of June 2024, its archive contains over 19 billion unique source files collected from over 300 million projects, with all their version history, including a full copy of the projects that were removed from Gitorious, Google Code and Bitbucket, which can be used to repair the broken link in the web of academic knowledge. It also provides for all archived software artifacts the Software Hash persistent intrinsic identifier (SWHID) that allows to precisely pinpoint the version of source code at all levels of granularity[4] [14]. As such, Software Heritage provides the core infrastructural layer that links the academic ecosystem [25] with the many other ecosystems that rely on software, like industry and public administration [13, 18].

Curate and catalog Software is a very complex and fragile object and therefore, metadata is needed for providing provenance information as well as to properly credit authors. For software, this is especially challenging because the roles of the different contributors are diverse and may change during the project. Fully automated processing may not provide accurate answers, making metadata curation necessary. Furthermore, citing the article mentioning a software is a common practice among researchers but is not compatible with the software lifecycle that makes this citation eventually inaccurate. This is the reason why (Chue Hong et al., 2019) highlighted the need to cite the software itself, acknowledging is as a legitimate product of research. It is to be noted that referencing and citing are related to distinct needs. For example, Software Heritage Hash Identifiers (SWHID) are permanent identifiers that allow to reference a specific version of the source code of a project, at different levels of granularity. Citing software refers to the authorship: the purpose is to attribute the credits. Institutional repositories such as HAL, the

[3]https://www.softwareheritage.org/howto-archive-and-reference-your-code/

[4]https://swhid.org

French multidisciplinary open archive, provide the adequate tooling to describe software. Thanks to the articulation between HAL and Software Heritage, one can generate a citable software description including a SWHID. The set of mandatory metadata is minimal: document type, authors, software name, academic field, license and the end-user receives some guidance from a moderator in order to enrich the description. At the end of the day, building a catalog upon HAL and Software Heritage enable an institution to showcase its software production.

3.4 Publishers

Publishers have a role to play in order to guarantee code is truly shared at the time of publication. We cannot be satisfied anymore with current recommendations where *availability upon request* is still an option.

- **Enforce open source** **mandatory**
- Linking publications and codes recommended
- Review software optional

Enforce open source *Ugly* code is always better than no code [34]. Sharing code has been made extremely easy using Zenodo or Software Heritage, even though it still requires a minimal experience with version control tools. In that context, publishers have a role to play in order to guarantee code is truly shared at the time of publication. We cannot be satisfied anymore with current recommendations where *availability upon request* is still an option. In that regard, the Nature journal received severe criticism [23] from the machine learning community, following the publication of "International evaluation of an AI system for breast cancer screening" [36]. Haibe-Kains et al. explains that the lack of details of the methods and algorithm code undermines its scientific value.

Linking publications and codes (and data) In order to make research more transparent, accessible and inclusive, it is essential to have access not only to publications but also to all scientific productions used to write them, especially codes and data. Publishers have a significant responsibility to integrate ways to link all these scientific objects together. The publication ecosystem must now integrate sources codes. Some editors have already started this process. For example, Dagstuhl Artifact Series (DARTS) [17] and IACR's Artifact Archive [30] enable publication of peer-reviewed software artifacts (cf. next paragraph) alongside papers that have been accepted in a conference or journal from the same editor. Long-term archiving of source code can be obtained by interoperating with code archives such as Software Heritage.

Software Review. As described in detail in the EOSC report on Scholarly Infrastructures for Research Software [18], various communities have over time put in place mechanisms for reviewing software associated to publications, ranging from a simple check of existence, to quality of metadata, to quality of software and adequacy w.r.t. what is stated in the publications. As an example, some computer science communities[5] have put in place an *artifact evaluation* process, with a dedicated program committee, which started in 2011. In these communities, authors are

[5]to the best of our knowledge, it concerns the formal methods, software engineering, systems, security, and architecture communities

encouraged to submit a software artifact in addition to their paper. Artifacts are built and evaluated to ensure that all the software-related experimental claims of the paper can be reproduced by anyone. An artifact consists of a snapshot of the computational environment and the software used for the experimental evaluation of the paper – distributed for example as a docker container or an image of a virtual machine. Reviewers may grant different badges to an artifact, depending on its various qualities[6]. Publishers should consider getting together to identify the various levels of review that have been field-tested, and propose a unified approach to enable the diverse research community adopt the one most appropriate to them.

4 Discussion

Not all scientific software is created equal. Some are meant to be widely spread in the scientific community, to grow and to be maintained in the long term, while others serve solely as a one-shot illustration of a concept or an idea. But both are important for science. Independently of the goal, the size, or the form of the software, we advocate first and foremost to share the code and archive it permanently even when authors may think it's not worth to be shared. Then, and only then, more can be asked from the authors. In the meantime, institutions, funders, libraries and publishers have a fundamental role to play and we propose a set of recommendations for each of them. Some of these recommendations have already been put in practice in France, the U.K., or by private foundations, such as the Sloan Foundation, the Chan Zuckerberg Initiative in the USA, that has been engaged with the recognition, support and promotion of scientific software for a few years already, with policies that start to have long lasting impacts in several dimensions. At both the European and international levels, a number of actors are also present and work similarly for the recognition of scientific software in science. To name just a few, we can cite the Research Data Alliance, UNESCO, Software Heritage, the Research Software Alliance, the European Open Science Cloud and Knowledge Exchange. We are happy to see these improvements, but there is still a long way to go, and we think that it is important to re-assess that research software is special: it has become over the years a critical ingredient of science and yet, it is still not recognized widely enough as a valuable scientific contribution [28]. If cannot be reduced to its "just data" form, and imposing some high-level principles that ignore this fact might be counter-productive: FOSS developers and researchers have created and shared code for many decades and their experience and practice, formed well before the term Open Science became popular, must be properly taken into account.

References

[1] J.-F. Abramatic, R. D. Cosmo, and S. Zacchiroli. "Building the Universal Archive of Source Code". In: *Communications of the ACM* 61.10 (Oct. 1, 2018). Ed. by ACM, pp. 29–31. DOI: 10.1145/3183558. published.

[6]The Association for Computer Machinery provides a comprehensive explanation of the badges used in this process[2]

[2] ACM. *Artifact Review and Badging*. 2020. URL: https://www.acm.org/publications/policies/artifact-review-and-badging-current (visited on 08/11/2023).

[3] M. Baker. "Reproducibility crisis". In: *nature* 533.26 (2016), pp. 353–66.

[4] A. Bassinet, L. Bracco, A. L'Hôte, E. Jeangirard, P. Lopez, and L. Romary. "Large-scale Machine-Learning analysis of scientific PDF for monitoring the production and the openness of research data and software in France". working paper or preprint. 2023. URL: https://hal.science/hal-04121339.

[5] I. Blanc, R. Di Cosmo, M. Giraud, D. Le Berre, V. Louvet, N. P. Rougier, S. Granger, and F. Pellegrini. *Highlights of the "Software Pillar of Open Science" workshop*. Tech. rep. Comité pour la Science Ouverte, June 2024. DOI: 10.52949/53.

[6] E. A. Bruford, B. Braschi, P. Denny, T. E. M. Jones, R. L. Seal, and S. Tweedie. "Guidelines for human gene nomenclature". In: *Nature Genetics* 52.8 (Aug. 2020), pp. 754–758. DOI: 10.1038/s41588-020-0669-3.

[7] K. S. Button, J. P. Ioannidis, C. Mokrysz, B. A. Nosek, J. Flint, E. S. Robinson, and M. R. Munafò. "Power failure: why small sample size undermines the reliability of neuroscience". In: *Nature reviews neuroscience* 14.5 (2013), pp. 365–376.

[8] C. F. Camerer, A. Dreber, E. Forsell, T.-H. Ho, J. Huber, M. Johannesson, M. Kirchler, J. Almenberg, A. Altmejd, T. Chan, et al. "Evaluating replicability of laboratory experiments in economics". In: *Science* 351.6280 (2016), pp. 1433–1436.

[9] J. C. Carver, N. Weber, K. Ram, S. Gesing, and D. S. Katz. "A survey of the state of the practice for research software in the United States". In: *PeerJ Computer Science* 8 (May 2022), e963. DOI: 10.7717/peerj-cs.963.

[10] I. B. Catala, R. Di Cosmo, M. Giraud, D. Le Berre, V. Louvet, and S. Renaudin. "Establishing a national research software award". In: *Open Research Europe* 3 (Oct. 2023), p. 185. DOI: 10.12688/openreseurope.16069.1.

[11] N. P. Chue Hong et al. "FAIR Principles for Research Software (FAIR4RS Principles)". en. In: (2021). DOI: 10.15497/RDA00068.

[12] L. P. Coelho. "For long-term sustainable software in bioinformatics". In: *PLOS Computational Biology* 20.3 (Mar. 2024). Ed. by J. A. Papin, e1011920. DOI: 10.1371/journal.pcbi.1011920.

[13] R. D. Cosmo, M. Gruenpeter, B. Marmol, A. Monteil, L. Romary, and J. Sadowska. "Curated Archiving of Research Software Artifacts: Lessons Learned from the French Open Archive (HAL)". In: *International Journal of Digital Curation* 15.1 (Jan. 1, 2020), p. 16. DOI: 10.2218/ijdc.v15i1.698. published.

[14] R. D. Cosmo, M. Gruenpeter, and S. Zacchiroli. "Identifiers for Digital Objects: the Case of Software Source Code Preservation". In: *iPRES 2018 - 15th International Conference on Digital Preservation*. Sept. 1, 2018. DOI: 10.17605/OSF.IO/KDE56. published.

[15] R. D. Cosmo and S. Zacchiroli. "Software Heritage: Why and How to Preserve Software Source Code". In: *iPRES 2017: 14th International Conference on Digital Preservation*. Kyoto, Japan, Sept. 25, 2017. URL: https://www.softwareheritage.org/wp-content/uploads/2020/01/ipres-2017-swh.pdf%20https://hal.archives-ouvertes.fr/hal-01590958. published.

[16] G. Courbebaisse, B. Flemisch, K. Graf, U. Konrad, J. Maassen, and R. Ritz. "Research Software Lifecycle". en. In: (2023). DOI: 10.5281/ZENODO.8324827.

[17] Dagstuhl. *Dagstuhl Artifact Series*. URL: https://drops.dagstuhl.de/entities/journal/DARTS (visited on 06/21/2024).

[18] R. Di Cosmo et al. *Scholarly Infrastructures for Research Software. Report from the EOSC Executive Board Working Group (WG) Architecture Task Force (TF) SIRS*. European Commission. Directorate General for Research and Innovation., 2020. DOI: 10.2777/28598.

[19] D. V. Dimitrova and M. Bugeja. "The half-life of internet references cited in communication journals". In: *New Media & Society* 9.5 (Oct. 2007), pp. 811–826. DOI: 10.1177/1461444807081226.

[20] French Committee for Open Science. *The 2023 Open Science Free Research Software Awards*. URL: https://www.ouvrirlascience.fr/the-2023-open-science-free-research-software-awards/ (visited on 06/21/2024).

[21] French Ministry of Reseach and Higher Education. *Second National Plan for Open Science*. 2021. URL: https://www.ouvrirlascience.fr/second-national-plan-for-open-science/.

[22] M. Gruenpeter, R. Di Cosmo, H. Koers, P. Herterich, R. Hooft, J. Parland-von Essen, J. Tana, T. Aalto, and S. Jones. "M2.15 Assessment report on 'FAIRness of software'". en. In: (2020). DOI: 10.5281/ZENODO.5472911.

[23] B. Haibe-Kains et al. "Transparency and reproducibility in artificial intelligence". In: *Nature* 586.7829 (Oct. 2020), E14–E16. DOI: 10.1038/s41586-020-2766-y.

[24] C. R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020), pp. 357–362. DOI: 10.1038/s41586-020-2649-2.

[25] M. of Higher Education and Research. *French National Strategy on Research Infrastructures 2021*. Accessed: August 2023. 2021. URL: https://www.enseignementsup-recherche.gouv.fr/sites/default/files/2023-02/french-national-strategy-on-research-infrastructures---2021-26489.pdf.

[26] K. Hinsen. "Computational reproducibility". In: *Computation in Science (Second Edition)*. 2053-2563. IOP Publishing, 2020, 6-1 to 6–20. DOI: 10.1088/978-0-7503-3287-3ch6.

[27] K. Hinsen. "Dealing With Software Collapse". In: *Computing in Science & Engineering* 21.3 (May 2019), pp. 104–108. DOI: 10.1109/mcse.2019.2900945.

[28] A. Hocquet et al. "Software in science is ubiquitous yet overlooked". In: *Nature Computational Science* (July 2024). DOI: 10.1038/s43588-024-00651-2.

[29] I. Hussey. "Data is not available upon request". In: (May 2023). DOI: 10.31234/osf.io/jbu9r.

[30] IACR. *IACR Artifact Archive*. URL: https://artifacts.iacr.org/ (visited on 06/21/2024).

[31] E. Jeangirard. "Monitoring Open Access at a national level: French case study". In: *ELPUB 2019 23rd edition of the International Conference on Electronic Publishing*. Vol. Academic publishing and digital bibliodiversity. Marseille, France: INRIA, June 2019. DOI: 10.4000/proceedings.elpub.2019.20.

[32] A.-L. Lamprecht et al. "Towards FAIR principles for research software". In: *Data Science* 3.1 (June 2020). Ed. by P. Groth, P. Groth, and M. Dumontier, pp. 37–59. DOI: 10.3233/ds-190026.

[33] D. Le Berre, J.-Y. Jeannas, R. Di Cosmo, and F. Pellegrini. *Higher Education and Research Forges in France - Definition, uses, limitations encountered and needs analysis*. Tech. rep. Comité pour la science ouverte, Nov. 2023. DOI: 10.52949/37.

[34] R. J. Leveque. In: *SIAM News* 46 (Apr. 2013). URL: https://sinews.siam.org/Details-Page/top-ten-reasons-to-not-share-your-code-and-why-you-should-anyway.

[35] S. Marquees. *Of Money, Responsibility, and Pride*. http://veridicalsystems.com/blog/of-money-responsibility-and-pride/. 2020.

[36] S. M. McKinney et al. "International evaluation of an AI system for breast cancer screening". In: *Nature* 577.7788 (Jan. 2020), pp. 89–94. DOI: 10.1038/s41586-019-1799-6.

[37] F. Prinz, T. Schlange, and K. Asadullah. "Believe it or not: how much can we rely on published data on potential drug targets?" In: *Nature reviews Drug discovery* 10.9 (2011), pp. 712–712.

[38] E. S. Raymond. *The cathedral and the bazaar: musings on Linux and open source by an accidental revolutionary*. eng. 2., überarb. und erw. A. With a foreword by Bob Young. Beijing; Cambridge; Farnham; Köln; Paris; Sebastopol; Taip: O'Reilly Media, 2001, p. 241. URL: http://www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/.

[39] B. M. Schilder, A. E. Murphy, and N. G. Skene. "rworkflows: automating reproducible practices for the R community". In: *Nature Communications* 15.1 (Jan. 2024). DOI: 10.1038/s41467-023-44484-5.

[40] R. Sonabend, H. Gruson, L. Wolansky, A. Kiragga, and D. S. Katz. "FAIR-USE4OS: Guidelines for creating impactful open-source software". In: *PLOS Computational Biology* 20.5 (May 2024). Ed. by J. A. Papin, e1012045. DOI: 10.1371/journal.pcbi.1012045.

[41] R. M. Stallman and J. Gay. *Free Software, Free Society: Selected Essays of Richard M. Stallman*. Scotts Valley, CA: CreateSpace, 2009.

[42] M. D. Wilkinson et al. "The FAIR Guiding Principles for scientific data management and stewardship". In: *Scientific Data* 3.1 (Mar. 2016). DOI: 10.1038/sdata.2016.18.