10 Quick tips for making your code last beyond your current job

This is a draft: the goal is to submit a version of this to PLoS, and put a preprint somewhere like Zenodo or arXiv.

The current state of the paper is now here: https://github.com/RichardLitt/10-simple-rules-for-code/. This document is closed to future edits. Edits to this document are still welcome, although we're closed to new authors at this time.

Past agenda for planning call Thursday Apr 17: Agenda for 10 Simple Rules doc

Help that's needed:

- Add references in the doc. It would be great to be able to back up claims with research.
- Add resources. There's surely more that could be in here!
- Add or refine points: What is missing? What shouldn't be here? What would you add?

Authors:
Richard Littauer, CURIOSS, SustainOSS, GNOME Foundation, and Te Herenga Waka Victoria University of Wellington 0000-0001-5428-7535
Clare Dillon, CURIOSS, Lero, University of Galway 0009-0008-6205-0296
Priyanka Ojha, 0000-0002-6844-6493
Ian McInerney, 0000-0003-2616-9771
Georg Link, Bitergia, 0000-0001-6769-7867
Mala Kumar
Daniel S. Katz, University of Illinois Urbana-Champaign, 0000-0001-5934-7525
Greg Wilson, 0000-0001-8659-8979
Eman Abdullah AlOmar 0000-0003-1800-9268
Mohamed Wiem Mkaouer 0000-0001-6010-7561
David Lippert 0009-0003-6444-9595
Daniel R. McCloy, 0000-0002-7572-3241
Bill Branan, 0000-0002-4735-6624
David Pérez-Suárez, 0000-0003-0784-6909
Sam Cunliffe, 0000-0003-0167-8641
Chang Liao, 0000-0002-7348-8858
Christoph Treude, 0000-0002-6919-2149
Tobias Augspurger

Sylwester Arabas, 0000-0003-2361-0082
Ethan P. White, Department of Wildlife Ecology and Conservation, University of Florida, Gainesville, FL, USA 0000-0001-6728-7745
Fang Liu 0000-0002-3383-2191
Geoffrey Lentner, 0000-0001-9314-0683
David Eyers, 0000-0002-7284-8006
Jouni Helske, INVEST Research Flagship Centre, University of Turku, 0000-0001-7130-793X
Elena Findley-de Regt
Kris Bubendorfer 0000-0003-4315-8337
Paola Corrales 0000-0003-1923-9129
Pieter Huybrechts, Research Institute for Nature and Forest (INBO), Brussels, Belgium 0000-0002-6658-6062
Phani Velicheti, 0009-0004-2580-3624
Daniel Morillo-Cuadrado, Universidad Nacional de Educación a Distancia (UNED), Spain, 0000-0003-3021-3878
Tommy Guy 0009-0003-3652-5036
Jan Ainali 0000-0001-8747-1670

If you make a comment, add your name here. Please use editing mode to make suggestions. ***Also, if you could, send a quick email to richard.littauer@gmail.com with your email address. It'll make it easier for me to publish this.***

The state of academic research is often precarious, particularly for those who work on research software, an asset that is often underfunded, undercited, and overlooked (Carver *et al*., 2022), but essential to modern research (Pearson *et al*. 2025). The situation is worsening due to the global political climate and the breakdown of established norms and practices. Academic researchers in all disciplines are facing losses of funding or jobs, a smaller pool of incoming students (Mallapaty, 2025), and institutional policies that demand immediate returns on investment. Researchers in non-academic organizations, such as government offices, NGOs, or non-profits (Woodward and Leeder, 2025), are also impacted. The scope of instability is global; institutional agreements between universities are tenuous, national research strategies are being upended, and the political environment and associated public funding are rapidly changing (Nature, 2025).

In this environment it is important that research software is resilient. Code should be built to outlast situations where the author changes workplace or industries, where partnernships fail or a lab or institution closes down, and where digital infrastructures shift and tooling ecosystems change. Further, as part of the global push for more responsible research assessment (DORA, coARA, ADORE.software), research software is increasingly being recognized as a significant and impactful component of researchers' profiles. The process of building resilient code can help the author as well as the code, even if their job transition is expected or welcomed.

Here are ten concrete tips to ensure your code remains accessible and usable by others even if your job situation is uncertain. The aim is to help you leave your project in a state where you or

anyone else can pick up where you left off if they need to. They are ordered by priority: the first few will help you quickly ensure preservation of your code in its current state. The rest will help ensure the ongoing use and development of the code in the long run. You do not need to follow each one to make an impact: do what you can where you are with what you have.

Even if your job is not in danger, following this guide will help you avoid legal, technical and practical obstacles for others to access and use your code, to contribute code back to your project, and to reproduce and cite your work. While there are times when retaining code or making it widely available is not ideal – for example, with some health or military software – overall, these tips will apply to most research software and will improve its auditability, and hence credibility. As well, the goal of this work is not to keep you on the hook for maintaining all of your code going forward, nor to put you at risk of litigation for releasing work out of copyright. Instead, tis guide will help leave your code in a better state, so that you can step away knowing you did what you could.

This guide focuses on software. Other resources already exist for data – archives such as Zenodo or FigShare, or, more broadly, the End Of Term Archive or the Internet Archive, and guides on sharing data. There are no projects that we have found that focus solely on hardening code in the face of job instability. Some general resources exist, such as the FAIR Principles for Research Software. Aligning your software with FAIR principles is an important long-term goal, but short-term work to preserve research software is also crucial given the current environment.

Before doing any of these things: Do not break the law or policies at your institution. Your work is not worth putting yourself at legal risk.

1. Consider your threat model.

When making plans, it's useful to know what you're planning *for*. Being explicit about **threat models** helps you prioritize, build consensus with colleagues, and check whether you've forgotten something important.

1. **Individual threats** are those that affect one or a few members of your team, such as a foreign student having their visa revoked without notice or a contributor taking extended leave. Threat modelling is from the perspective of the asset – especially for software under the maintenance of a single author, a common individual threat can also come from more mundane career or life changes, when the main developer lacks resources to maintain or loses access to a project, while the software is still valuable to others. The most common way to prepare for this is to require everyone to document their work thoroughly, but that rarely works in practice:
    a. The hours spent writing those descriptions are hours *not* spent doing research, so people will always short-change the former to focus on the latter.
    b. People invariably fail to write down the "obvious" parts of their work that are anything but obvious to the next person.

In practice, “automate what you can and create checklists for what you can't” seems to be a better approach, particularly if you check those checklists by having someone else try to use them while their authors are still available to take notes and update them.

2. **Leadership threats** are individual threats that affect the project's leader, such as the leader being doxxed or targeted personally in the media because of their position. One way to prepare is to have a designated successor (who can also stand in for you if you ever want to take a holiday); another is to talk with peers about who will inherit what if your project is shut down.

3. **Institutional threats** are those that affect large groups at once, such as your university or professional association shutting down or downsizing, for example, due to a natural or human-made disaster. In such events so many people may be affected at the same time that the rest of the community can't absorb them. Regional and national governments handle disasters like these by having evacuation plans to get victims to safe(r) places, and corollary plans for putting beds in high school gyms and flying in food and emergency medical personnel to help people when they arrive. The equivalents in research are to have professional associations lobby governments for changes in visa rules for scientific refugees and for contingency funding to attract and support them.

4. **Global threats** are ones that affect everyone, not just researchers. For example, there is no technical or legal obstacle to the US government requiring American companies to charge a dollar a minute for video conferencing calls involving participants outside the United States. Similar levies on email services, file storage, and other online services would paradoxically have *less* long-term impact on research than the targeted threats described above, as they would force national governments to find effective remedies quickly. There are calls for digital sovereignty coming from governments and lobby groups (for instance, the Netherlands, Germany, New Zealand (Duckles et al. 2025), the proposal for a EuroStack, and this open letter from the European Tech Industry) which may also influence where code can and should be stored and shared.

Figuring out what you’re planning *for* is the first item in our recommendations because everything else depends on it. As is the case in all disasters, making those plans before you need them helps reduce the odds of them being needed, as the process may help you identify risks you can eliminate.

2. Get sign-off on releasing it publicly

Some institutions have specific policies for releasing code publicly or licensing it appropriately. The first actionable step is to find out what policies your institution follows, and whether or not you need sign-off from someone before releasing your code. Some institutions will demand that code be released publicly; others will prohibit it. Demands might also originate from funding

agencies (e.g., EU[1], NASA[2]) or from research journals where you plan to publish the research (e.g., Katz *et al.* 2018, Ham et al. 2019). Find out where your code sits.

If your institution has a policy regarding open source code, get sign-off for your code now. Do not wait. Contact code reviewers who can review your code if that is necessary for publishing your work. If there is no formal sign-off process, that means you may be in charge of your code, which may make it easier for you to make it publicly available.

If you are unsure, you can often find information by asking your direct line manager, the person who pays you, or offices like the research office, your school office, the library, the tech transfer office, or other similar locations. If your institution has a dedicated Open Source Program Office (OSPO), ask them, or connect to networks like CURIOSS or the TODO Group that advocate for OSPOs. If you work with multiple institutions, look at your contract and the bilateral agreements around products or deliverables.

Don't assume that if you had permission before, you have it now or will have it in the future. Policies may change, and you may be left locked into a policy that doesn't benefit the openness of your code. Also consider that the person you report to may change or be removed, so it's important to act fast if you're interested in the longevity of your code.

If you have your next job lined up, prepare to ask about their policies. Work licensing into your contracts, and know your rights for any potential outputs. Making your code comply with policy at the end of the project is more time-consuming than making these decisions early, and sometimes may not be possible..

3. Choose an open license

Making your code publicly available does not ensure that other people can use it legally. That's what a license does. Without a license, most people cannot use your code, as you maintain all copyright.

Use the MIT, Apache 2.0, or BSD-3-Clause license to maximize usability. If copyleft is preferred, use GPLv3 or AGPL-3.0, but be aware that copyleft licenses may limit commercial adoption. Copyleft comes with advantages and disadvantages: it can protect your open source project and the community around it from exploitation, but it can also limit adoption. If you need help, go to choosealicense.com, refer to Furtonato & Galassi (2021), or look at institutional policies from institutions similar to your own.

There are licenses that are beyond the scope of open source licenses approved by the Open Source Initiative (OSI) that may be useful for your code. For instance, Creative Commons licenses can be used for your documentation or assets that are attached to your code. Ethical

[1] https://commission.europa.eu/about/departments-and-executive-agencies/digital-services/open-source-software-strategy_en

[2] https://science.nasa.gov/open-science/nasa-open-science-funding-opportunities

licenses or licenses based on the Blue Oak Model License may also be helpful. Choose a license that fits your use case.

Add the `LICENSE` file to your code, most often at the root of your repository. This should have the contents of the license. This may be more legally binding than just referencing a license in your documentation. You may need to consider copyright laws for countries besides your own if your code is hosted online, too.

If you have doubts, check with your legal team at your institution. We are not lawyers, and thus we are not your lawyers.

4. Put your code somewhere else

Always remember LOCKSS: Lots of Copies Keep Stuff Safe. This is one of the easiest ways to ensure that your code remains accessible and discoverable. Beyond risks to your code relating to technology, some recent threats are more political and targeted: you also want to ensure that institutional resources you rely on are not able to be blocked or turned against you.

If your code only exists locally, put it online somewhere. GitHub, GitLab, and Codeberg are all social coding platforms (forges) that provide easy ways to store code. Software Heritage archives software from multiple forges, such as GitHub and Gitlab. You can also snapshot the current state of repositories in a compressed archive file (e.g., `.zip` or `.tar.gz`) and put those copies on any archival research repository like Open Science Foundation (OSF), Zenodo, and figshare. Zenodo also provides an integration with GitHub by which creating a tagged release on GitHub triggers a new deposit of the repository on Zenodo, complete with a DOI. You can do this even if you do not have a publication that needs a DOI for your code.

If your code is already in an online repository, mirror that repository on multiple platforms. You can also make sure you have current local clones of all your projects, on multiple computers if possible. Also, it is a good idea to fork and download projects that you value or your software depends on, so if they are taken down, you are not starting from scratch. Online forks of projects are not enough, as they may not be persistent if the source code is removed. Don't assume a single storage solution is going to make it accessible; multiple storage places are always better. If you do this work, you can document it, like rOpenSci has.

If possible, host your code under a community-maintained organization, and not solely on your own account or under an institutional organization. Instead, clone it to your personal account.

Remember that the choice of platform for organizing your code will influence where your documentation is stored and whether you maintain easy control of it. For instance, GitHub issues and wikis are not easily exportable. You may be able to export the text into a local, machine-readable archive that you could mine using a localized LLM (for instance, through this tool). Your social network, on the other hand, is not exportable, and social lock-in to a single platform makes coding platforms non-fungible.

Distributing your code as an independent software package – such as a Python package on PyPI or Conda, or R package on CRAN – integrates it into a broader ecosystem and fosters community collaboration. It also ensures that your code is stored on yet another platform.

Consider institutional, national, or international data repositories for added longevity. This makes it more likely that your code will be available or accessible later. It is not a good idea to rely solely on employer-associated repositories (like Google Drive) unless you retain control. This also applies to email addresses used as logins for platforms, as if you lose access to your email, you may lose access to your code. Attaching the software with a secondary email which is not tied to a specific institution adds also an additional point of contact for others. Furthermore, adding your ORCID as metadata allows a persistent link to a profile that you can control, and change in the future in case your point of contact changes.

You can also talk to an international colleague who is in your field, and ask them if you can share your code with them, and vice versa. This has the added benefit of making someone else aware of your code.

But, as always, if your institution has a policy for sharing code publicly, follow it. If they have a policy that talks about what you do with your code if you are let go or forced to leave your job, follow it. If there is no policy, ensure that you are able to clone everything locally and share widely.

5. Document your code

Your code is only as useful as the use others can make of it. Document it as much as is necessary for others to be able to pick it up and use it for its intended use. Don't go overboard; if you're reading this, you don't have time to document every method.

In particular, add a `README.md` at the root of your repository that explains the purpose, setup, and usage of your work. If you can, add a tutorial or an explanation for how to use the major functions of the code. Adding a minimal example that shows how to use it makes it easier for others to get started. If it is a component or a library, try showing how to integrate it into existing systems. Consider sharing a screen recording of you using your code. Instructions on how to build, launch, test, or deploy it are also valuable - but a README is the first step. There are guides on writing documentation that may help, such as Huybrechts *et al.* (2024), Littauer (2025), Katz *et al.* (2025), and The Turing Way Community (2025).

If you can, make the link to documentation easy to find, or available at a canonical URL. Bundling and archiving documentation together with the code would also help.

A contributor's guide that explains how to set up a development environment, how to add tests, and how to become part of the project's team can be extremely helpful. Having one of these also leads to an increase in contributions - see the papers by Steinmacher cited in these ten simple rules for onboarding contributors. Your project is more than just your code - it is your

governance, your issue tracker and changelogs, and your shared values. If you can, write these down.

If you haven't named your code yet, consider naming it in an inclusive way, without hinting at direct and exclusive affiliation to a single institution. This can help if the code needs to be relocated, but also in herding developers from other institutions to contribute. Naming is, of course, hard.

There are many other steps that are possible for documenting your code: see this guide. It is worth repeating that you do not need to go overboard. Documentation is never finished, and documentation is an art, not a checklist to be filled out.

6. Make your code reproducible

Reproducibility is an essential part of making your code last. We touched on this above with the call to document your code. Documenting the code includes describing dependencies in a machine-readable way, e.g., Python's pyproject.toml file, an npm package.json file, or equivalent manifests. When referencing code in your docs or manifests, ensure that a version label is included, so that future users have a way to identify the proper version.

Making your code independently executable with consistent outcomes is more than just documenting what it needs to run, however. It involves abstracting the code, thinking about how it is used, and then refactoring it to ensure that it is easier for the next person. It also involves considering all of the steps necessary for your code to run. That may also mean adding instructions on how to make code run on other machines under different environments. If you have access to other machines, you can attempt to run your code on them, and document the process as you go. Providing instructions on how to reproduce your results "in the cloud" enables reviewers and followers of your work to skip time-consuming environment setup and dive straight into running the code. If you use Jupyter notebooks for your work (see Perkel 2018), and host them in public repositories, enabling their execution in the cloud can be reduced to a single URL-click with platforms such as mybinder.org or analogous proprietary solutions from tech giants (e.g., Google Colab, Github Codespaces).

If you can, version your runtime environment. Use tools like Docker, Podman, renv (Ushey K & Wickham H, 2025), conda, or rix to make clear whatever anyone else would need to set up to make sure the code is being reproduced in the same computational environment.

Try and prevent avoidable hardware or software vendor lock-in that might limit usability of code by another institution. If your code or the process you use to work on it depends upon access to platforms that you may not keep, see if you can find alternatives (for instance, look into simulation solutions on EaaSI). Some proprietary software solutions do have open source alternatives, and it is worth documenting if and how one may leverage them to run your code. When developing code for specific hardware (e.g., GPUs), it's worth exploring frameworks that enable the same code to execute—albeit at reduced performance—on standard commodity

hardware. Finally, preferentially work with common tools if you can. They'll make adoption easier, as they're generally more portable than obscure projects.

7. Ensure your code maximizes accessibility of your data

Data and code live within a shared context. As previously mentioned, there are other resources for making your data available. Data that is necessary to run or reproduce your code should be made publicly available when possible. In order for accessible data to be useful, you may need to do some work with your code to ensure that others can use it.

Document your datasets and include metadata describing the data. Even if you need to exclude sensitive data, you may be able to provide a safe synthetic dataset, or include metadata describing the data so that someone else could reasonably do so.

If your code references specific datasets or formats, try to link to them in your documentation. If your code was used in any publications, you can list those publications in your documentation, too.

Avoiding proprietary formats for your data will help make your code more relevant. Use CSV, JSON or other types of machine-readable data. PDFs lock in the data and are difficult to mine. Other formats are preferable for most data. Adjust your code accordingly.

If you can, document analysis pipelines, using Jupyter Notebooks, R Markdown, or any standard documentation. Provide Makefiles or other workflows to automate data processing. Ultimately, others may have to make new data to work with your code if yours is not used for a significant period of time. Ensuring that those who follow you - including your future self - can do so will help extend the reach and impact of your code.

7. Make your code citable

Your code is your work. If you would like to track how the code is used and the impact it has had, you can ensure that properly citing it is easy to do. Code is increasingly cited by researchers (see Smith *et al.* 2016, Katz *et al.* 2021, Garijo *et al.* 2024) and recognized as a research object in its own right. Cited work may have a longer lifespan.

Create a DOI for Code Releases. You can do this directly through a Zenodo integration if you're on GitHub. This ensures long-term citation and retrieval even if the repository disappears.

Add a Citation.CFF or codemeta.json file to your code. The format (Druskat *et al*. 2021) captures the metadata associated with the software in the repo, which in turn helps to ensure that it is citable by making it easy for others to do so. This will enable you to get credit down the road, and it should be a low lift, especially as some generators and tools (cffinit, codemeta generator, and others) for these files exist.

If you have the time, consider having your code peer-reviewed and published. There are now many places where you can do this; for example, in the Journal of Open Source Software, the Journal of Open Research Software, Journal of Statistical Software, R Journal or in rOpenSci. Make sure to reference any code, including your own, in any papers that use it.

Another option is publishing your normal work, but including your code with that work. Submitting your software as a part of the supplementary material of a journal article that uses it not only improves the reproducibility of your analyses, but also acts as a snapshot backup of your software on the publisher's website.

There is work underway to create types of persistent identifiers for research software (for instance, Software Heritage's work on SWHIDs or this proposal), but it is still nascent.

8. Encourage community adoption

Publishing your code is not the same as publicizing it. The steps so far have focused on things you can do with the code itself, but the social aspect of your work is also important.

A short video showing how to use the software, a tutorial made for beginners, or a short slide deck that other people can incorporate into their classroom lectures or lab sessions can make a big difference. Think of this work as marketing. There are guides on how to do this for science. For instance, Kuchner's "Marketing for Scientists" for project leaders has good guidance for approaches, like coming up with an elevator pitch for your project, or making a good poster about it.

Announce your project on relevant mailing lists, forums, or social media. Talk about it in your department (you've already got sign-off from your immediate line manager, right?). Ask for other maintainers of your code, and be prepared to give them something in return - compliments, a hand up on their projects, or the power to merge commits or be a coauthor on your project's publications.

Disseminating your code through tutorials and workshops at conferences is an effective strategy to reach and engage potential adopters within your community. Tutorials and workshops can offer an opportunity for you, as an author, to talk about your work and motivate your audience to use it. It is also an opportunity for participants to interact directly with your work and appreciate its potential. The choice of the conferences, where your potential community gathers, is important. Go where your users are.

Additionally, you can think of this work as networking, which will enable you to build relationships with attendees who share a keen interest in your research. Effective networking is the best way to ensure that you land well, and it may be possible for you to land a position where you can continue working on your code.

If you can, get one other person to commit to your code. Having community contributions leads to joint IP ownership, which makes your work harder to take down by single actors. This is

especially true for government and industry. If you can enable a collaborator from another industry to work on your code, it becomes more complicated for any one organization to remove it. You can also work their name into the license by adding them as one of the holders of the license. A simple way to help people get involved is to label and create "Good First Issues" to assist people in finding easy access as contributors.

As above, we recommend that you seek advice from both your and your collaborator's organization's legal teams.

9. Write a succession plan

All of this work is in the expectation that the code may be used later. Even if it isn't, you want to leave it in such a state that it could be. A succession plan can be used to suss out how others can take over; a sunsetting plan can be used as a framework for how to mothball the code permanently. Writing either plan will help you know what you're going to do to ensure that the code, data, and project are bundled up and put away nicely.

This is also useful because this process can have a deliverable in itself: a statement of how the project was used and what you accomplished with it. Write up the story of the project. Share it widely. This will make it easier to wind your project down, while also recognizing that all projects eventually end.

Write up the goals you initially had for this project, and ask if they were served by the project and whether you succeeded. Sometimes, it may make sense to fold a project into another one, or to ask someone else to take it over. The process of writing up a sunsetting plan can involve realizing that the project doesn't need to end, but your continued involvement may not serve the project. Think about whether someone else could be trained to use it, or take up leadership for it, or fund work on it.

For some projects, it may be worth working with a fiscal sponsor such as NumFOCUS, Software Freedom Conservancy, the Eclipse Foundation, or Open Source Collective. There are many foundations and fiscal hosts. Through fiscal hosting, your project can take donations from its community of users to fund continued maintenance or other project costs. Universities or governments are generally not ideal fiscal hosts for software projects, due to high overhead costs and procurement processes. Smaller hosts like Open Source Collective take a minimal overhead in comparison, often an order of magnitude less. Some fiscal hosts are more hands on, and may help offload much of the work of keeping your code going. A key part of using a fiscal host is that they put the software in a neutral home where it is no longer owned by a research performing organization. This is based on the sponsor owning the IP. This also lets people become "owners" of the project over time without having to move to a single institution.

Before you go, remember to empower at least one other person to act on behalf of the project. If you use GitHub, consider designating a successor account that can administer your repositories

if you can no longer access or control them. If you don't have a successor designated, it still helps to write up a document explaining where the keys are, and how to use them.

11. Talk about what you're doing

All of this work is labor. You'll learn that not all of these steps apply to you, and you'll find other steps that could also be useful. Further, you may not want to do any of this, and it may feel like pulling teeth.

Make that work meaningful. Talk about how hard it is to archive working code. Talk about how necessary the code you've written is. Talk about how others should protect and harden their systems.

You can also use this process to find more contributors (Steps 7 and 8), or get others to hold it for you (Step 2), or to get sign off (Step 1). This may help your project live longer, and help others use it.

Use Mastodon or Bluesky to find other archives that will be able to host your code in the future. These steps will help ensure your work remains accessible and usable, regardless of your employment situation.

Your institution may be able to help you, with advance notice. If your institution has an OSPO, ask them to help with this process. If not, the library or the research office may also have helpful guides.

Conclusion

If you learn things, bring them back to this taskforce. Don't underestimate the power of screaming into the void.

Main conclusion waiting on Greg. Celebrate and grieve.

References

- Carver JC, Weber N, Ram K, Gesing S, Katz DS. 2022. A survey of the state of the practice for research software in the United States. PeerJ Computer Science 8:e963 https://doi.org/10.7717/peerj-cs.963
- Duckles, J., Littauer, R., Black, M., Ellerm, A., Eyers, D., Gee, M., & Harang, A. (2025). eResearch NZ / eRangahau Aotearoa – The Case for Open Source in the Science System of Aotearoa. eResearch NZ / eRangahau Aotearoa 2025, Ōtautahi Christchurch, Aotearoa NZ. Zenodo. https://doi.org/10.5281/zenodo.15080979
- Druskat S., Spaaks J.H., Chue Hong N., Haines R., Baker J., Bliven S., Willighagen E., Pérez-Suárez D., Konovalov O. (2021). Citation File Format (version 1.2.0). DOI: 10.5281/zenodo.1003149

- Fortunato, L, Galassi, M. (2021). The case for free and open source software in research and scholarship. Phil. Trans. Royal Soc. A 379(2197). https://doi.org/10.1098/rsta.2020.0079
- Garijo, D., Arroyo, M., Gonzalez, E., Treude, C., & Tarocco, N. (2024, April). Bidirectional paper-repository tracing in software engineering. In *Proceedings of the 21st International Conference on Mining Software Repositories* (pp. 642-646). https://doi.org/10.1145/3643991.3644876
- Ham, D., Hargreaves, J.C., Kerkweg, A., Roche, D.M., Sander, R. (2019). The publication of geoscientific model developments v1.2. Geosci. Model. Dev. 12(6) https://doi.org/10.5194/gmd-12-2215-2019
- Huybrechts P, Trekels M, Abraham L, Desmet P (2024). B-Cubed software development guide. https://docs.b-cubed.eu/guides/software-development/
- Katz D.S., Forbes M., Silen L., Curcuru S., Hucka M., Tang Y., Branan B., Richard L. (2025). Open-source software project documents URL: https://github.com/corsa-center/oss-documents
- Katz, D. S., Niemeyer, K. E., & Smith, A. M. (2018). Publish your software: introducing the journal of open source software (JOSS). *Computing in Science & Engineering*, *20*(3), 84-88.
- Katz DS, Chue Hong NP, Clark T et al. Recognizing the value of software: a software citation guide [version 2; peer review: 2 approved]. F1000Research 2021, 9:1257 https://doi.org/10.12688/f1000research.26932.2
- Knoth, P., Romary, L., Lopez, P., Di Cosmo, R., Smrz, P., Umerle, T., ... & Pride, D. (2025). Making Software FAIR: A machine-assisted workflow for the research software lifecycle. *arXiv preprint arXiv:2501.10415*.
- Lee BD (2018) Ten simple rules for documenting scientific software. PLOS Computational Biology 14(12): e1006561. https://doi.org/10.1371/journal.pcbi.1006561
- Littauer R. (2025). Standard Readme (version 1.2.2). DOI: 10.5281/zenodo.11164868 URL: https://github.com/RichardLitt/standard-readme
- Morin A, Urban J, Sliz P (2012) A Quick Guide to Software Licensing for the Scientist-Programmer. PLOS Computational Biology 8(7): e1002598. https://doi.org/10.1371/journal.pcbi.1002598
- Páll-Gergely, B., Krell, FT., Ábrahám, L. *et al.* Identification crisis: a fauna-wide estimate of biodiversity expertise shows massive decline in a Central European country. *Biodivers Conserv* **33**, 3871–3903 (2024). https://doi.org/10.1007/s10531-024-02934-6
- Pearson, H., Ledford, H., Hutson, M., & Van Noorden, R. (2025). Exclusive: the most-cited papers of the twenty-first century. *Nature*, *640*(8059), 588-592. https://www.nature.com/articles/d41586-025-01125-9
- Perkel, J. M. (2018). Why Jupyter is data scientists' computational notebook of choice. Nature, 563, 145–146. *https://doi.org/10.1038/d41586-018-07196-1*
- Perkel, J. M. (2023). How to make your scientific data accessible, discoverable and useful. *Nature*, *618*(7967), 1098-1099. https://doi.org/10.1038/d41586-023-01929-7
- Smith AM, Katz DS, Niemeyer KE, FORCE11 Software Citation Working Group. 2016. Software citation principles. PeerJ Computer Science 2:e86 https://doi.org/10.7717/peerj-cs.86

- Ushey K, Wickham H (2025). *renv: Project Environments*. R package, https://CRAN.R-project.org/package=renv.
- Alistair Woodward, Stephen Leeder, Making science great again. Or not, International Journal of Epidemiology, Volume 54, Issue 2, April 2025, dyaf029, https://doi.org/10.1093/ije/dyaf029
- Yurkanin, A. (2025, April 15). Two months after Trump's funding cuts, a nonprofit struggles to support refugees and itself. *ProPublica*. https://www.propublica.org/article/refugees-funding-cuts-nashville
- The Turing Way Community. (2025). The Turing Way: A handbook for reproducible, ethical and collaborative research. Zenodo. https://doi.org/10.5281/zenodo.15213042
- Nature (2025) Trump 2.0: An assault on science anywhere is an assault on science everywhere. *Nature*, *639*(8053), 7–8. https://doi.org/10.1038/d41586-025-00562-w

---

Publication in PLoS?

- See https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1003858

Final review steps listed by commenters that should be done when this is in better shape:

- "even as someone who has already done many of these things for most of their work, I find the frequent use of imperative mood a bit much; I can imagine readers who are already panicked getting overwhelmed very quickly. If there's time near the end for a pass that softens the language a little ("***If possible,*** ensure data too is versioned and citeable") I think it would be worth doing.
- "Is the intention of this document to also cover work done by a team or lab (as well as a single individual)? If so, there may need to be a review with this use case in mind."
- Ask PLOS about anonymous authorship.