

10 quick tips for making your software outlive your job

Richard Littauer^{RL*}, Greg Wilson^{GVW}, Daniel S. Katz^{DSK}, Yanina Bellini Saibene^{YBS}

RL CURIOS, SustainOSS, GNOME Foundation, and Te Herenga Waka Victoria University of Wellington, Wellington, New Zealand, 0000-0001-5428-7535

GVW Plotly Inc., Toronto, Ontario, Canada, 0000-0001-8659-8979

DSK University of Illinois Urbana-Champaign, IL, USA, 0000-0001-5934-7525

YBS rOpenSci, 0000-0002-4522-7466

- See `contributors.tex` for all contributors to date.
- Superscript initials will be replaced with numbers once the contributor list stabilizes.

Abstract

Loss of key personnel has always been a risk for research software projects. Key members of the team may have to step away due to illness or burnout, to care for a family member, from a loss of financial support, or because their career is going in a new direction. Today, though, political and financial changes are putting large numbers of researchers out of work simultaneously, potentially leaving large amounts of research software abandoned. In this article, we present ten tips to help researchers ensure that the software they have built will continue to be usable after they have left their present job.

Introduction

Academic life is often precarious, not least for those who work on research software, an asset that is essential to modern research [1] but often underfunded, undercited, and overlooked [2]. The situation has recently worsened dramatically due to changes in the political climate in many countries: researchers in all disciplines face the loss of funding or jobs, a smaller pool of incoming students, and institutional policies that prioritize commercial returns. Researchers in non-academic organizations such as government labs, non-governmental organizations (NGOs), and non-profits are also impacted [3] as national research strategies are upended and associated public funding redirected or cut entirely [4].

In this environment it is essential that research software projects be *resilient*. Code should outlast situations such as the author changing workplaces or careers, partnerships failing, a lab closing down, or tooling and digital infrastructure becoming unavailable. We therefore present ten tips to ensure your software remains accessible and usable by others even if you are no longer able to work on it. You do not need to adopt all of them to make things better; instead, as in all emergencies, you should do what you can, where you are, with what you have.

This guide focuses on software. Other guides already exist for data [5], where archives such as Zenodo, FigShare, the End Of Term Archive or the Internet Archive are now widely used to ensure long-term availability. However, we have not found any guide to making research software resilient or to *sunsetting* it (i.e., phasing it out

gracefully). Even if your job is not in danger, these tips will help others access and use your code, contribute to your project, and to reproduce and cite your work. As such, our recommendations are aligned with the global push for more equitable assessment of research contributions by initiatives like DORA, coARA, and ADORE.software.

Tip 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 have forgotten something important. As with all disasters, making those plans before you need them helps reduce the odds of them being needed, as planning may help you identify risks you can eliminate.

1. **Individual threats** 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. Especially for software under the maintenance of a single author, individual threats can also come from more mundane career or life changes. The most common way to prepare for this is to require everyone to document their work, 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.

Tip 8 explores alternatives, particularly ones that can be put in place on short notice.

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 work. Tip 3 discusses ways to plan for this.
3. **Institutional threats** affect large groups at once, such as your department being shut down or your entire field having funding cut. These events affect so many people at the same time that the rest of the community can't absorb them. Regional and national governments plans for 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. At the time of writing, universities and researchers' professional societies have not started to do the equivalent.
4. **Global threats** affect everyone, not just researchers. For example, there is no technical or legal obstacle to the US government requiring American companies to charge for video conferencing calls involving participants outside the United States. Similar levies on email, file storage, and other online services would undoubtedly prompt national governments to find alternatives, but would still result in (at least) months of disruption.

Tip 2: Stay within the law

Ensuring your work remains usable is not worth putting yourself at legal risk. Before following any of the tips below, make sure you have a legal right to do so. Most institutions and journals now have policies for licensing code and/or releasing it

publicly [6, 7]. Funding agencies at various levels also often have policies (e.g., EU¹ and NASA²) which may or may not align with those of institutions and journals, and institutional policies may place restrictions on what you are allowed to say publicly about your situation. Our first actionable tip is therefore to find out what those policies are, e.g., whether you need explicitly permission to publish your software, and if so, from whom.

If you are unsure whether there is a policy or not, ask your direct manager, the person who pays you, or offices like the research office, the library, or the tech transfer office. If your institution has a dedicated Open Source Program Office (OSPO), ask them or connect to networks like CURIOS, the OSPO Alliance, 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.

Once you know whether your institution has a policy on licensing code, get whatever sign-offs you need immediately. Reach out to colleagues who can review your code if that is necessary for publishing it, and review theirs if they ask. If it appears that there is no formal sign-off process, send an email to someone in authority (e.g., the chair of your department or your grant officer) saying explicitly that you believe this to be the case, and copy that email to an account you will be able to access after leaving your institution.

Do not assume that if you had permission before, you have it now or will have it in the future. Policies are changing rapidly, and you may find yourself locked into one that no longer allows you to do what you want. Also consider that the person you report to may be replaced with one who knows you less well or is less sympathetic, so acting now may be easier than acting later.

Finally, if you have your next job lined up, ask about their policies and make sure that your right to share your work is written into your contract. Making your code comply with policy after the fact is riskier and more time-consuming than doing so early, and sometimes not possible.

Tip 3: Decide if you are ending, pausing, or passing on

There are thousands of books in print about how to start a business, but only a handful describe how to hand one over or wind one down, and most of those focus on succession within family-owned businesses. Guides to running a laboratory such as [8, 9] say little more, so such wisdom as we have is passed on person-to-person if it is passed on at all.

The first step is to decide if work on the project is ending, being suspended temporarily (for some version of “temporarily”), or if you are handing it on to someone else. If the project is ending or going on hiatus, focus on making what you have accomplished findable and citable as discussed in the tips that follow. To help you prioritize, write up the goals you initially had for this project and ask how well they were served by what you actually did. Doing this may actually help you realize that the project doesn’t have to end, but can instead be folded into someone else’s work, even if you are no longer involved.

If instead you hope to hand the project on to someone else, you can either choose a successor (who will typically be someone already working on the project) or ask your peers for volunteers or recommendations. Writing a *succession plan* that describes what the work will actually entail will help in both cases. What will your successor have to learn and do during the transition? What will they be responsible for once they are in charge? And crucially, how much work is running the project actually going to be?

¹https://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>

Please be honest rather than optimistic: finding out after you have agreed to take something on that it is much larger than you realized can strain both professional and personal relationships.

Some projects try to ensure longevity by working with a fiscal sponsor such as NumFOCUS, Software Freedom Conservancy, the Eclipse Foundation, Open Source Collective, or other foundations and fiscal hosts. Through fiscal hosting, a project can take donations from its community to fund continued maintenance and other costs. Universities and governments are generally poorly suited for this due to high overheads and procurement costs; fiscal sponsors may take an order of magnitude less. In addition, a fiscal sponsor puts the software in a neutral home where it is no longer owned by a single research organization, which helps insulate it from shifts in policy.

However, fiscal sponsorship is only an option in some countries, and being accepted as a sponsored project is a long and difficult process. Even when it is possible, the copyright holder of the work may continue to be the institution that paid for the development of the software. Private foundations may be a more attractive alternative in some jurisdictions; as per Tip 2, check with *local* legal experts before taking action.

Tip 4: Choose an open license (if you can)

Making code publicly available does not ensure that other people can use it: you maintain all copyright unless you explicitly include a license, so they may still need explicit permission or may simply be unsure whether they do or not. The Open Source Initiative (OSI) maintains a list of open source licenses it has approved; their implications are widely understood, so choosing one of those will make your project more understandable to others. Please do *not* try to write your own license, as it will have the opposite effect.

Broadly speaking, the MIT, Apache 2.0, or BSD-3-Clause licenses place the fewest restrictions on re-use. The GPLv3 and AGPL-3.0 licenses are *copyleft* licenses that require people to share their modifications to the software with the community [10]. These licenses can prevent companies from taking advantage of your work without giving anything in return, but institutions discourage or disallow the use of copyleft licenses out of fear that they will constrain commercialization.

More broadly, Creative Commons licenses can be used for documentation or research reports, while ethical licenses or licenses based on the Blue Oak Model License may also be appropriate. For guidance, go to choosealicense.com or refer to [11, 12]; the discussion in [13] about why people *don't* share is also helpful. Whatever license you choose, place its text in a file called `LICENSE`, `LICENSE.txt`, or `LICENSE.md` in the root directory of your project, as this is where people and automated tools alike will look for it.

The Law, Again

Most discussion of open source licensing focuses on circumstances in a handful of affluent countries in the same way as discussion of fiscal sponsorship (Tip 3). For example, many Latin American countries have access to information laws that include research outcomes and artifacts. You can publish data and software under that, but the institution sets the license type, and you must use institutional repositories. As always, consult *local* legal experts before taking action.

Tip 5: Save everything in multiple places

Now that you're legally able to share your code, remember LOCKSS: Lots of Copies Keep Stuff Safe. But copies are not enough: the threats to your project are political as

well as technological, so you should ensure that loss of institutional support (or worse, that institution turning on you) does not mean loss of access.

The Road Not Taken

[14] described a file-sharing system for scientific data based on the BitTorrent protocol. It failed to find wide adoption because of the (quite reasonable) association in many institutions' collective minds between BitTorrent and illegal downloading. At the time of writing, though, many individuals and groups are turning to BitTorrent to share datasets that are at risk of disappearing precisely because of the protocol's resilience.

GitHub, GitLab, and BitBucket are social coding platforms centered around Git, a widely-used open source version control tool. However, they are all commercial entities based in the same legal jurisdiction, which means they are vulnerable to *correlated threats*: trouble with any of them may mean trouble with all of them. ([15] is a sobering account of what can happen when key infrastructure comes under threat.) Codeberg, while currently much smaller, is a non-profit based in a different jurisdiction; consider mirroring your work there or using it as your primary host.

When you create projects on hosted services, use a team account as the project owner rather than a personal account. Doing this makes it easier to give other people permission to manage the project, and as noted in Tip 6, a project with multiple owners from different institutions is harder for any one institution to lock down. For the same reason, do not rely solely on logins or email addresses associated with your institution: instead, ask yourself if you will still have access to the project if you lose that identity, and add an identity you personally control to the team that owns the project. You can also add your ORCID to the project's metadata so that the project links to a profile that you control even when your contact address changes.

What's In a Project?

When deciding what to store where, remember that your version control repository only stores part of what makes up your project. For example, GitHub issues and wikis are displayed in the browser but live in GitHub's database; you can use the GitHub API to download their content, but (a) the result is intended for consumption by machines, not people, and (b) if you are doing this on short notice, the odds are high that other people are trying to do it at the same time. GitHub's servers may not be able to manage that load when you need them to, so use tools like offline-issues to save hosted content as plain-text files on a regular basis.

Version control isn't the only way to create and save copies of your work. Software Heritage archives software from multiple forges; you can also snapshot the current state of repositories in a compressed archive file (e.g., `.zip` or `.tar.gz`) and deposit those copies with the Open Science Foundation (OSF), Zenodo, or figshare. (Zenodo even integrates with GitHub so that tagging a release on GitHub automatically triggers deposit of a new archive on Zenodo, but again, this automation is only useful as long as both ends are accessible.) Institutional, national, or international data repositories also enhance longevity.

Storing copies on someone else's computer isn't the only option: you can (and should) make copies of your projects on computers that you own. Again, this is a place where colleagues can help: ask them to make copies on their computers in exchange for you making copies of theirs on yours. Whatever you do, document it as rOpenSci has.

Similarly, if you distribute your code as a package on PyPI, Conda, or CRAN, that package can contain all of the source code. Doing this also fosters community collaboration.

Tip 6: Encourage community adoption

Publishing your code is not the same as publicizing it. The more people who know about and rely on your project, the greater the odds that someone will help keep it alive, whether by working on directly or supporting you to do so. A simple way to help people get involved is to label entry-level issues (e.g., to use the “good first issue” label on GitHub) so that people can find a place to start [16].

If you haven’t named your project yet, choose a name that doesn’t hint at direct and exclusive affiliation to a single institution. This can help if the code needs to be relocated, and also makes contributors from other institutions feel welcome.

Announcing your project on mailing lists, forums, or social media, and talking about it at conferences are necessary but not sufficient: everyone else is doing this too, so your project will almost certainly get lost in the noise. [17, 18] are good guides to what you can do beyond the obvious. A short video showing how to use the software or a slide deck that others can incorporate into their lectures will increase uptake by lowering the cost of adoption. Similarly, a one-page website that opens with an elevator pitch explaining who the software is for and how it will make their lives better is much more likely to lead to that crucial “second glance” than a list of publications.

Disseminating your code via tutorials at conferences is another effective strategy. Tutorials are an opportunity for you, as an author, to describe your work in depth and (more importantly) convince your audience to use it. Networking through such events will enable you to build relationships with the people most likely to care about your project, and is the best way to ensure that you land well after you leave your current position.

One to Many

If you can, have at least one person outside your organization commit to your code. Such community contributions lead to joint ownership of intellectual property (IP), which makes it harder for any single institution to lock down your work. You can reinforce this by adding their name to the copyright statement in your license.

Finally, remember that it isn’t all about *your* project. If you ask others to help maintain your code, be prepared to give them something in return: a letter of reference, help testing or documenting their project, or co-authorship credit on the project and associated publications.

Tip 7: Do what you’re supposed to (if you have time)

Research software is increasingly recognized as a citable object [19–21], and numerous books and papers describe how to organize and run a research software project. Rather than repeating what is in [22–33], our sixth tip is a prioritized checklist of things to do *if you have time*. If you do not, please consult the next tip.

1. Create a DOI for each release of your software to ensure citability and retrievability even if the project repository becomes unavailable, and add a Citation.CFF or codemeta.json file to your code [34] containing citation metadata. Tools such as cffinit codemeta generator, and others can help you do this.
2. Have your code peer-reviewed and published in a venue such as the Journal of Open Source Software, the Journal of Open Research Software, the Journal of Statistical Software, the R Journal, or rOpenSci.

3. Document how to *use* your software so that others can pick it up and use it if you are not available. In particular, add a `README.md` at the root of your repository that explains the purpose, setup, and use of your work, and add a tutorial showing how to use the major components of the code [35–39].
 4. Document any datasets your project depends on, and use CSV, JSON, Parquet, HDF5, or other widely-recognized formats rather than proprietary formats. Include the data in your project if possible; if that is not possible for legal reasons or because of the data’s size, include URLs with data version IDs (or the date of the last download) in your documentation.
 5. Describe your project’s dependencies in a machine-readable way using Python’s `pyproject.toml` file, an `npm package.json` file, the `DESCRIPTION` and `NAMESPACE` files of an R package, or whatever else is standard for your language.
 6. Write at least a few tests for your software so that people who want to use it (or contribute to it) can tell if it’s working in their environment. These do not have to be formal unit tests [40]: as [41] explains, even a handful of “this input should produce this output” checks can save a lot of frustration.
 7. Document how to *contribute* to your software, e.g., how to set up a development environment, run tests, and add new features. Such a guide leads to an increase in contributions [42], as does an explanation of the project’s governance, i.e., how decisions are made and who gets a voice in making them.
- Note that all of the suggestions above will also help with adoption, for the same reason that a tidy restaurant attracts more customers than a messy one.

Tip 8: Do what you can (if you have to act quickly)

A paramedic is not a doctor who cuts corners, but rather someone who has been trained to work subject to very different constraints. [43] provides some guidance for “good enough” practices in research computing; other tips in this paper are in this vein, such as creating `.zip` archives of your project as an alternative to using version control. A few others include:

1. Explanations of what the software does and how to use it take priority over documentation of its internals because (a) someone who knows the former will be better able to figure out the latter and (b) AI code assistants can do a reasonable job of helping newcomers navigate an unfamiliar code base if they are given enough context.
2. Similarly, a short screencast showing how you use your software is often faster to create than pages of documentation, and may actually be more useful provided provided closed captions are attached to make the content searchable. Again, AI tools can help create those captions, though some editing is invariably required to clean up mistranslation of technical jargon.
3. If you do not have time to bring your project in line with best practices, try instead to clean it up by removing files that are no longer used, deleting sections of your setup instructions that no longer apply, closing bug reports that are no longer relevant, and so on. Eliminating clutter in this way will make what’s useful easier to find.

4. Briefly document what happened. If you're the main developer working on a grant and you've moved on to something else, say that in the README. If the funding changed then say that the work is unfunded. This information is often the most useful for users but the hardest to find. Saying it up front will help others understand how your work is positioned.

Tip 9: Talk about what you're doing

All of the work we have described is extra labor. You won't want to do it, particularly not when so many other things demand attention as well. Make that work meaningful by helping others. Tell your colleagues what you have learned about your institution's legal requirements, about navigating its bureaucracy, and about the least hurtful way to break the news to your students, collaborators, and users. Share your own tips on social media (preferably Mastodon rather than X or Bluesky, as the latter two are single points of institutional failure), and share things that *haven't* worked as well so that other people can avoid stepping on the same landmines.

Most importantly, take the time to grieve. Having years of work come to an abrupt end will take a toll on your mental health; many of your colleagues will be suffering as well, and those who haven't been hit (yet) may be affected by survivors' guilt. Be angry, be sad, but remember that in the aftermath of a disaster—natural or man-made—people often become altruistic, resourceful, and brave.

Tip 10: Organize

There is little point surviving today's flood if you drown in tomorrow's. The best strategy is to take preventive action, and the best way to do that in a research context is to become active in professional associations and push them to take meaningful action.

Many readers of this paper will find the thought of doing this uncomfortable. As Putnam points out [44], voluntary organizations have seen a slow but steady decline in membership over several decades. While there are many contributing factors, the net result is that professionals aged 30–60 are less likely to be involved in civil society and the political process than the generation before them. This disengagement effectively hands power to extremists and special interest groups and enables them to seize control of civil institutions in order to advance their agendas [45]. To see the impact, ask yourself who is more likely to run for a seat on your local school board: someone with a doctorate in epidemiology or an anti-vax conspiracy theorist?

We have written this paper because we believe scientific inquiry is a public good and worth fighting for. The present climate requires us to focus on preserving what we can, but playing defense is just a way to lose more slowly. If we work together and accept that we and our institutions need to change, we can create a world in which research does not just survive, but thrives.

References

1. Pearson H, Ledford H, Hutson M, Van Noorden R. Exclusive: the most-cited papers of the twenty-first century. *Nature*. 2025;640(8059):588–592.
2. Carver JC, Weber N, Ram K, Gesing S, Katz DS. A survey of the state of the practice for research software in the United States. *PeerJ Computer Science*. 2022;8:e963. doi:10.7717/peerj-cs.963.

3. Woodward A, Leeder S. Making science great again. Or not. *International Journal of Epidemiology*. 2025;54(2). doi:10.1093/ije/dyaf029. 340
341
4. Nature. Trump 2.0: An assault on science anywhere is an assault on science everywhere. *Nature*. 2025;639(8053):7–8. doi:10.1038/d41586-025-00562-w. 342
343
5. Perkel JM. How to make your scientific data accessible, discoverable and useful. *Nature*. 2023;618(7967):1098–1099. doi:10.1038/d41586-023-01929-7. 344
345
6. Katz DS, Niemeyer KE, Smith AM. Publish your software: introducing the journal of open source software (JOSS). *Computing in Science & Engineering*. 2018;20(3):84–88. 346
347
348
7. Ham D, Hargreaves JC, Kerkweg A, Roche DM, Sander R. The publication of geoscientific model developments v1.2. *Geosci Model Dev*. 2019;12(6). doi:10.5194/gmd-12-2215-2019. 349
350
351
8. Barker K. *At the Helm: Leading Your Laboratory*. Cold Spring Harbor Laboratory Press; 2010. 352
353
9. Cohen CM, Cohen SL. *Lab Dynamics: Management and Leadership Skills for Scientists*. 3rd ed. Cold Spring Harbor Laboratory Press; 2018. 354
355
10. Morin A, Urban J, Sliz P. A Quick Guide to Software Licensing for the Scientist-Programmer. *PLOS Computational Biology*. 2012;8(7):e1002598. 356
357
358
doi:10.1371/journal.pcbi.1002598.
11. Fogel K. *Producing Open Source Software* (updated); 2020. Available from: <https://producingoss.com/>. 359
360
12. Fortunato L, Galassi M. The case for free and open source software in research and scholarship. *Phil Trans Royal Soc A*. 2021;379(2197). 361
362
363
doi:10.1098/rsta.2020.0079.
13. Gomes DGE, Pottier P, Crystal-Ornelas R, Hudgins EJ, Foroughirad V, Sánchez-Reyes LL, et al. Why don't we share data and code? Perceived barriers and benefits to public archiving practices. *Proceedings of the Royal Society B: Biological Sciences*. 2022;289(1987). doi:10.1098/rspb.2022.1113. 364
365
366
367
14. Langille MGI, Eisen JA. BioTorrents: A File Sharing Service for Scientific Data. *PLoS ONE*. 2010;5(4):e10071. doi:10.1371/journal.pone.0010071. 368
369
15. Tamburri DA, Blincoe K, Palomba F, Kazman R. "The Canary in the Coal Mine..." A cautionary tale from the decline of SourceForge. *Software: Practice and Experience*. 2020;50(10):1930–1951. doi:10.1002/spe.2874. 370
371
372
16. Steinmacher I, Conte T, Gerosa MA, Redmiles D. Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects. In: *Proc. CSCW'15*. ACM; 2015. Available from: <https://doi.org/10.1145/2675133.2675215>. 373
374
375
376
17. Kuchner MJ. *Marketing for Scientists: How to Shine in Tough Times*. Island Press; 2011. 377
378
18. Bellini Saibene Y, Salmon M. *Marketing Ideas For Your Package*; 2024. Available from: <https://ropensci.org/blog/2024/03/07/package-marketing/>. 379
380
19. Smith AM, Katz DS, Niemeyer KE, Group FSCW. Software citation principles. *PeerJ Computer Science*. 2016;2:e86. doi:10.7717/peerj-cs.86. 381
382

20. 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. doi:10.12688/f1000research.26932.2.
21. Garijo D, Arroyo M, Gonzalez E, Treude C, Tarocco N. Bidirectional paper-repository tracing in software engineering. In: Proceedings of the 21st International Conference on Mining Software Repositories; 2024. p. 642–646. Available from: <https://doi.org/10.1145/3643991.3644876>.
22. Sandve GK, Nekrutenko A, Taylor J, Hovig E. Ten Simple Rules for Reproducible Computational Research. PLOS Computational Biology. 2013;9(10):e1003285. doi:10.1371/journal.pcbi.1003285.
23. Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, Guy RT, et al. Best Practices for Scientific Computing. PLOS Biology. 2014;12(1):e1001745. doi:10.1371/journal.pbio.1001745.
24. Lee BD. Best practices for scientific computing. F1000Research. 2018;7:1734. doi:10.12688/f1000research.15424.2.
25. Dryden MDM, Bhamla MS, Kubicek-Sutherland JZ, St Laurent J, Bochicchio D. Ten simple rules for graduate students and researchers in the time of COVID-19. PLOS Computational Biology. 2020;16(5):e1007958. doi:10.1371/journal.pcbi.1007958.
26. Akhmerov A, Cruz M, Drost N, Hof C, Knapen T, Kuzak M, et al. Scientific Data Analysis Pipelines and Reproducibility. Netherlands eScience Center. 2020;.
27. Chue Hong NP, Druskat S, Haines R, Jay C, Katz DS, Sufi S. Software Management Plans in Research Projects. Software Sustainability Institute. 2021;.
28. Lees JP, Chue Hong NP, Cruz M, Hanwell MD, Hetherington J, Ram K, et al. Building and sustaining research software development capacity. Research Ideas and Outcomes. 2022;8:e89876. doi:10.3897/rio.8.e89876.
29. Druskat S, Katz DS, Calatrava Moreno MDC. Research software: Definition, taxonomy, and community-driven guidelines. Patterns. 2023;4(8):100797. doi:10.1016/j.patter.2023.100797.
30. Akhmerov A, Kuzak M, Martinez-Ortiz C, Turkyilmaz-van der Velden Y, Bakker T, Coen G, et al. Recommendations for sustainable research software practices. Patterns. 2023;4(11):100824. doi:10.1016/j.patter.2023.100824.
31. Kumar V, Mulder N, Psomopoulos F, Wee LE. Enhancing scientific software sustainability through automated testing practices. F1000Research. 2023;12:498. doi:10.12688/f1000research.134994.1.
32. Struck A, Kuhring M, Kutzner D, Grunert M, Renard BY. Best practices for project setup for bioinformatics software developers. Briefings in Bioinformatics. 2023;24(1):bbac546. doi:10.1093/bib/bbac546.
33. Reina G, Bouquin DR, Crosas M, Katz DS, Mayernik MS, Smith AM, et al. Ten simple rules to help ensure software citation is considered and usable. PLOS Computational Biology. 2024;20(1):e1011381. doi:10.1371/journal.pcbi.1011381.
34. Druskat S, Spaaks JH, Chue Hong N, Haines R, Baker J, Bliven S, et al.. Citation File Format; 2021.

35. Lee BD. Ten simple rules for documenting scientific software. PLOS Computational Biology. 2018;14(12):e1006561. doi:10.1371/journal.pcbi.1006561. 426
427
36. Huybrechts P, Trekels M, Abraham L, Desmet P. B-Cubed software development guide; 2024. Available from: 428
<https://docs.b-cubed.eu/guides/software-development/>. 429
430
37. Littauer R. Standard Readme; 2025. Available from: 431
<https://github.com/RichardLitt/standard-readme>. 432
38. Katz DS, Forbes M, Silen L, Curcuru S, Hucka M, Tang Y, et al.. Open-source software project documents; 2025. Available from: 433
<https://github.com/corsa-center/oss-documents>. 434
435
39. The Turing Way Community. The Turing Way: A handbook for reproducible, ethical and collaborative research; 2025. Available from: 436
<https://doi.org/10.5281/zenodo.15213042>. 437
438
40. Irving D, Hertweck K, Johnston L, Ostblom J, Wickham C, Wilson G. Research Software Engineering with Python: Building Software that Makes Research Possible. CRC Press/Taylor and Francis; 2021. 439
440
441
41. Taschuk M, Wilson G. Ten simple rules for making research software more robust. PLOS Computational Biology. 2017;13(4):e1005412. 442
doi:10.1371/journal.pcbi.1005412. 443
444
42. Sholler D, Steinmacher I, Ford D, Averick M, Hoye M, Wilson G. Ten simple rules for helping newcomers become contributors to open projects. PLOS Computational Biology. 2019;15(9):e1007296. doi:10.1371/journal.pcbi.1007296. 445
446
447
43. Wilson G, Bryan J, Cranston K, Kitzes J, Nederbragt L, Teal TK. Good Enough Practices in Scientific Computing. PLOS Computational Biology. 2017;13(6):1–20. doi:10.1371/journal.pcbi.1005510. 448
449
450
44. Putnam RD. Bowling Alone: The Collapse and Revival of American Community. 2nd ed. Simon & Schuster; 2020. 451
452
45. Bueno de Mesquita B, Smith A. The Dictator’s Handbook: Why Bad Behavior is Almost Always Good Politics. PublicAffairs; 2022. 453
454