Barriers to Reproducible Scientific Programming

David Gray Widder[†], Joshua Sunshine[§], Stephen Fickas[‡]
Carnegie Mellon University[†§], University of Oregon[†‡]
dwidder@cmu.edu, sunshine@cs.cmu.edu, fickas@cs.uoregon.edu

***Abstract*—Scientists often write code to support their science. To investigate the state of scientific programming on smaller teams in diverse scientific contexts, we interviewed eleven scientists about their programming practices, and the extent to which they adhere to six common best practices. We argue that these practices are essential to the core scientific value of reproducibility. Our results indicate that many of these practices are not followed because of barriers such as low self-efficacy and misaligned incentive structures. We conclude with suggested improvements to the tooling, education, and incentives of scientific programmers.**

I. Introduction

Programming is often relied on as a tool of scientific discovery. Scientists perform *replication* to "convince ourselves that we are not dealing with a mere isolated "coincidence' [1], [2] often conducted by external scientists by merely re-analyzing the original data, or painstakingly duplicating the original experiment anew [3], [4]. While professional software engineers often have formal training (*e.g.*, a computer science degree) and must make complex decisions (*e.g.*, designing a scalable architecture), many scientific programmers, including those which we study, lack this training. Past work (Section V) on scientific programming tends to focus on larger science teams which include trained software engineers, meaning that practices such as testing, version control, and documentation, which we argue are necessary for reproducibility, are often followed by default. However, we investigate the barriers that smaller research groups in university settings, comprised of domain experts who moonlight as software engineers, may face in adhering to these basic practices. We find that time constraints, tooling barriers, self efficacy barriers, training barriers, and barriers resulting from unequal experience levels in a team are the most significant barriers participants face.

To evaluate what barriers scientific programmers face in their adherence to practices which enable reproducibility, we: define six best practices and justify how they enable reproducibility, conduct interviews with 11 scientifically diverse participants, report their adherence to these best practices, and examine barriers they face in adhering to these these best practices. We identify and describe barriers which cut across different best practices, and conclude by recommending ways to reduce barriers and encourage adherence.

TABLE I
Participant Information

P#	Field	Job Title	Coding %
1	Mathematics	Asst. Prof.	25
2	Astrophysics	Prof.	15
3	Physics	Prof.	15
4	Chemistry	PhD Student	50
5	Climatology	Prof.	25
6	Geographic Info. Sci.	Asst. Prof.	5
7	Geography	PhD Student	25
8	Ecology	Postdoc	90
9	Bioinformatics	Prof.	5
10	Biology	Prof.	5
11	Organizational Science	PhD Student	50

II. Method

Participants: Our study is set in an R1 public research university in the USA. We used maximum diversity sampling (*i.e.*, [5]) to understand programming practices in as many sciences as possible. We asked departmental secretaries or other known contacts in scientific departments for "faculty and Ph.D. students in your department who may use code as part of their academic work", and emailed these candidates with a request to participate. A total of 11 participants agreed. Their field, rank, and percent of time they spend writing code is in Table I.

Materials and Procedure: Participants filled out a Pre-Interview Questionnaire to give the interviewer context[1], and early responses from this questionnaire were used to refine the interview protocol before the first interview. With the participant, we explained the study and why we were recording their responses, answered any questions, and gave participants a consent form to review and sign. Over 30 to 60 minutes, the first author administered the questions found in the "Interview Script"[1]asked followups as needed, and made detailed notes.

Analysis: The first author then reviewed their notes and refined them using the recorded audio. Using an iterative process, the first author then coded sections of the transcript and survey responses relevant to the best practices identified.

Threats to Validity: With eleven participants at one university, our results may not generalize to other participants, in other universities. Participants' processes were creative and varied, so determining whether they adhered to a best practice involved subjective judgment. We did not notice the role documentation plays in reproducibility until observing the frequency with which participants brought it up organically, so

978-1-7281-0810-0/19/$31.00 ©2019 IEEE

[1]Available in http://davidwidder.me/files/2019_VLHCC_Barriers.pdf

we lack data from four of our participants' who did not (see Table II). Our questionnaire and interview protocols are in an online appendix so others may critique or extend our study.

III. Results

A. Deliberate Design

Definition: A process to determine requirements for the system, and plan what to build and how to build it.

Rationale: A deliberate design requires involves careful thought about the interfaces of one's system so that it is more reusable, and thus easier to reproduce one's results. This reduces the cost of replication, and allows others to test science the code supports in a new context.

Results: Only six participants produce any documentation of their design (2, 3, 4, 6, 8, 10), such as paper napkin drawings (4), flowcharts (2, 10), or whiteboard sketches inserted into grant proposals (6). Two participants said that they *do not have significant design processes*: one describing his as "*minimal*" (9); another only has a deliberate design processes for "*big systems*". Seven participants' design processes are collaborative with local or external teams (1, 3, 4, 5, 6, 7, 8). Six participants' design processes involve searching for existing code which may fulfill their goals (2, 3, 5, 7, 8, 11). Five participants' design process begins by thinking through the scientific phenomena their system will study or model (1, 2, 3, 10, 11). Four participants use an iterative process to design prototypes which they later refine (1, 4, 9, 11).

Barriers: Three participants mentioned the difficulty of *long term planning* as a design barrier: one believes his process would improve if he could follow an explicit process of visualizing the design for the project from the start (8), another expressed a will improve his design process by having a concrete plan of intended functionality instead of a abstract flowchart (2). Two participants expressed *team based* design barriers: one said that "*getting an interdisciplinary team including political scientists, fire scientists, and computer scientists to agree on model parameters is difficult*" (6); another said that designing a system on a team with differing levels of technical expertise is difficult because members make different assumptions about what is possible to build (7). One participant finds it difficult to design transparent interfaces so that others can trust and build on his system, and believes that this would be easier if he had taken a programming course (4). One said that some"*kind[s] of exploration don't lend themselves well to a formal design process*" (6).

B. Documentation

Definition: Documentation is material which teaches others about the purpose of, correct usage of, or suggestions for extending or modifying systems.

Rationale: Without adequate documentation, scientists have trouble tracking correspondence between code versions, datasets, and results, as well as what scientific intentions behind their code [6]. Like paper lab notebooks, documenting code explains how to reproduce results from code. Documentation also allows others to understand how different parts of a system fit together, allowing them to more easily replicate parts of the system piecemeal.

TABLE II
Best Practice Adherence Summary.

P#	Design	Doc.	E. C.	V.C.S.	Test.	P. R.
1	~	✗	✓	✓	✓	✓
2	✓	-	~	✓	✓	✓
3	✓	✓	✓	✓	✓	✗
4	✓	✓	✓	✓	✓	✓
5	~	✗	~	✗	✗	~
6	✓	-	✗	✓	✓	✗
7	✓	✗	~	✓	✗	✓
8	~	~	~	✓	✓	✓
9	~	✓	✓	✓	✓	✓
10	✓	-	✗	✗	✓	✓
11	~	-	✗	✗	✓	✗

✓: practice followed, ✗: not followed, –: no data, ~: grey areas where we felt unable to make a binary categorization.

Results: While participants recognize the the pitfalls of insufficient documentation, some do not do it. One participant describes the "*code and data archaeology*" he must do later to understand old pieces of code he had written (5); another says he "*spends hours and hours looking at old [undocumented] code to figure out what it does*" (8). One participant said insufficient documentation makes it hard to collaborate because it is difficult to understand code written by other people when no one writes documentation (11). Two *do not document* their code aside from the associated research publication (1, 3). Another says "*research code is often 'good enough', once it works, you don't go back and clean it up and add comments … there isn't the same polish that goes into industry code*" (7). Three of the four who document use specialized tools: two use Jupyter notebooks (9, 4), one a paper lab notebook (9), another participant's group has a wiki page for "*higher level documentation*" and a review process to ensure that code is readable (3). Another comments his code (8).

Barriers: Documentation can often be achieved with lightweight tooling, so it is not surprising that barriers are not tool based: one assistant professor said that his job does not *incentivize* him to document his code besides writing its associated journal article (1), another participant *lacks time* to write "*clean, well commented and documented code*" (7), and another often *forgets* to include comments and other documentation (3).

C. Use Existing Components

Definition: The usage, when possible, of existing code, such as libraries, packages, or frameworks to build a system.

Rationale: Existing and vetted software components are less likely to have errors [7], and writing less code from scratch may allow scientists to be more productive. However, we argue this also brings replicability benefits: by using standard trusted components, replicators can more quickly understand and focus on the system's novel, potentially erroneous code.

Results: Despite how common reuse is in open source and other contexts [7], [8], only four participants reuse code from outside sources. Three write *entirely novel code* for each new

project (6, 10, 11); another minimally reuses code by copying examples from textbooks (5), and another reuses build scripts but writes the rest from scratch (7). Two participants *reuse code internally* from their past projects, from as far back as the 1980s (2, 8), and four *reuse external code*, such as Python or Sage libraries (1, 4), or from an external team in a shared repository (3, 9). Relatedly, six participants begin their design process by searching for existing code which may fulfill their goals (2, 3, 5, 7, 8, 11), see Sec III-A).

Barriers: Participants expressed three *tooling barriers*: one believes Python has poor package management, and that it is time consuming to check if a desired component exists in the Sage math library because it is poorly indexed (1); another faces difficulty using libraries which make different assumptions about the final system (7). Another starts his design searching the literature to find relevant code, but "*can't see the behind-the-scenes code from the paper*" (11).

D. Use a Version Control System

Definition: Version control systems help teams track changes to code and data, and make undoing changes easier.

Rationale: Version control systems act as a backup, and facilitate public release by easing code sharing between researchers. Additionally, version control systems can track which dataset was created by which exact code version, facilitating replication later.

Results: Despite the centrality of version control use in industrial and open source contexts, three of our participants do not use a version control system: one simply records the the date on each file he edits (5), another creates a new folder for each subsequent version of his code (11), and another uses no version control for his *15 year old* project (10). However, the majority of our participants use Git (1, 2, 3, 4, 6, 8, 9) or Subversion (7) for at least one of their projects.

Barriers: Participants largely expressed *team barriers* to version control use: one said "*people who didn't know [version control] required that we change our process. It was hard, because these tools were helpful, but it was not worth teaching and requiring others to learn them.*" (7), and faces difficulty convincing his team to use version control even though they know how (2). Another expresses a *self efficacy* barrier: "*I wish] I was more fluent with Git. It works fine but sometimes [I have to] to Google*" (1). Another does not use version control for his 15 year old project, believing it "*adds too much overhead*" (10), and another believes it is unnecessary because his code is "*write once, use once*" (5),

E. Testing

Definition: Software testing is a manual or automatic process to ensure that the system under test meets given requirements and other quality attributes.

Rationale: In addition to benefits by way of quality assurance, concrete examples of what code does, such as test cases, allow others to understand new code better, thus assisting external replicability [9].

Results: Only three participants use any specialized tools to test their code, such as JUnit or a custom build script (1, 8, 9). Six others test their system by manually feeding it fake data and comparing output with expected values (2, 3, 4, 7, 10, 11). One participant performs *sensitivity analysis* (*e.g.*, [10]) (10), and another checks that her results are "*reasonable*" and conducts a "*paper logic check*" at each step in her system (7).

Barriers: Many cite *time barriers* to testing: deadlines prevent one participant from adhering to Test Driven Development and must usually adds tests later (1), others wish they had more time to write formal tests suites (8) and automate automate their tests (9). Another finds it odious to manually verify each intermediate number at each step of his system: "*[debuggers] allow you to step though code, line by line and find whatever fills all the arrays, make sure all the darn numbers that should be there are correct.*", despite this: "*it takes months to a year to finally convince yourself that the system's answer is correct*" (2). One participant believes there are insufficient opportunities for math PhD students to learn testing practices (1).

F. Public Release

Definition: Publicly releasing code on hosting sites such as GitHub, or even as part of a formal academic review and publication process.

Rationale: Code is often a core component of the scientific process. Public release of code allows others to critique and replicate the original experiment using the original code.

Results: Public release of code is the most basic step needed for reproduction; despite this, three participants do not release their code publicly (3, 6, 11), and another does not but shares his code when asked (5). Seven release their code publicly in at least some cases (1, 2, 4, 7, 8, 9, 10); two of seven participants submit code as part of their journal or conference submissions (8, 10), and an additional participant wishes to (1).

Barriers: Barriers to public release largely concerned *self efficacy*. Many wished they were better at making their systems usable so that other less technically able domain experts can evaluate and build off it after public release (4, 6, 8); relatedly, another wishes he knew a more conventional programming language so that his system would be of greater use after public release (10). Another does not have enough *time* to package or clean up his code before hosting it on a public repository, and separately says that he is *disincentivized* from systematic public release: "*Software is ... an experiment [but] if you try to include a Python snippet in a paper, the editor will ask you to remove it, and this is irksome*", adding "*The way we evaluate research should value code contributions more.*" (1), indicating the cause of the complaint a participant above in Section III-C had, who wanted to use code from journal papers he read but couldn't because it was "*behind the scenes*".

IV. Discussion: Cross Cutting Barriers

Six participants cited **time barriers**: one wishes he had more time to learn programming practices better to play a more active role in developing his team's system (6). Another said

quality is often "good enough" for the fast paced needs of research, but given more time she would write better documented code (7). Another doesn't have time to seek the programming training he feels he needs even though he believes it would save time in the long run (8). Two participants do not write tests because they lack time (8, 9), and another desires more time to improve his tool's UI (10). Another said that given time constraints and his poor programming skills, he depriortizes research which requires more coding (1).

Six participants cited **tool based barriers**: four spoke of poor support for parallelization or computational efficiency (1, 2, 8, 11). Another finds debuggers difficult to use with his system (2), another does not move his system to the cloud because of trust issues, calling it "*opaque, a black box*" (3).

Six participants raised **self efficacy barriers**. One participant believes he is bad at using Git (1), another believes he is inefficient at writing code, often using trial and error (2), and another feels he should take a class to correct his inadequate coding practices (8). Another wishes he felt confident pursuing research which would require more complex systems, rather than making small changes to existing ones (3), similar to another who wishes his programming practices were better to allow him to be more in charge of developing his team's system (6). Two participants believe their systems are inadequate for public release (4, 6).

Three participants mentioned **training-related barriers** to best practice adherence: one believes testing practices are difficult to learn without training, so much so that his math department is adding a course to teach this and other programming practices to their PhD students (1), similarly, another believes it is difficult to write sustainable software without taking a CS course (4). Another participant believes a more formal CS training would have taught him good coding practices, as well let him solve programming problems more quickly (2).

Five participants mentioned **team related barriers**: three find it hard to design the project because each finds their team to span many fields, and not all understand the complexities involved in building the system (6, 7, 8), one adding that this makes it difficult to get feedback from those with domain expertise (8), and another stops using tools like version control to accommodate less experienced programmers (7). One uses a niche language called Igor, and notes that the small ways in which it differs from C makes it difficult to teach to new team members (10). Another participant said that it is difficult to understand one anothers code when building systems collaboratively (11).

V. Related Work

Much past work has argued for the best practices we cover in our interviews: one article argues "Publish your computer code: it is good enough" [11], aligning with our section III-F "Public Release", another article codifies many practices aligning with our study including "Use a version control system" [12] covered in section III-D "Use a Version Control System"), and another suggests "Design the project up-front". [13], covered in Sec III-A, "Deliberate Design".

There is also empirical work on similar topics: one surveyed a broad sample including teachers and researchers in government or industrial labs, asking participants to self reported their understanding and perceived importance of software engineering best practices, but did not identify barriers to following these practices nor specific variations in the manner in which they do [14]. A similar survey study has the same limitations [15]. Another multiple case study used interviews and survey to understand process barriers in four software engineering teams working on large scientific, engineering, and military projects, with upwards of 15 full time engineers and 100k customers [16]. There appears to be more literature on testing in scientific programming than other practices we study (*e.g.*, [17], and there have also been a literature review on a broader set of scientific programming practices [18], which highlights conflicting results regarding barriers to documenting scientific code, and also concludes that barriers to "infrastructure" practices, including version control and using existing code, are understudied.

Our work relates to broader research on end user programming (*i.e.*, [19]), one line of work closely related to ours studied biologists and planetary scientists but did not specifically study reproducibility barriers, and teams studied deliberately included trained software engineers [20]–[23].

VI. Implications

Many participants faced barriers which existing tools and processes could solve, without modification, but they did not know of these tools or did not know how to use them. Increased **education** is needed about best practices and tools which support them, but that which would be provided by a traditional software engineering curriculum would be excessive and not fit within the time constraints our participants face. We agree with our Participant 9, who recommended training by software-carpentry.org, who "teach basic lab skills for research computing", particularly because they host tutorials at conferences in his field, thus literally and educationally "meeting learners where they're at". A few participants said that existing tools designed to solve their problems are overly complex and add to much overhead because they are designed with the needs of professional software engineers in mind, therefore **adaption of existing tools** for those with lower self efficacy, and for teams with wildly varying levels of software engineering experience is needed. Several examples of researchers building low barrier to entry tools for this audience exist (*i.e.*, Variolite [24], Helena [25], Burrito [6]) and future tools could assist with these best practices specifically. Finally, participants discussed a **lack of incentive**. Journals could demand releasing reproducible, best practice adherent code in the same way they demand other forms of reproducibility. Tenure decisions could consider the same as "service to the community," and funding agencies could consider producing public, applicably code as an explicit criterion. These may seem like lofty goals, but we believe science's increasing reliance on code, combined with increasingly pervasive fears of a reproducibility crisis, may force the issue.

REFERENCES

[1] K. R. Popper, "The logic of scientific discovery hutchinson," *Hughes, John,(1987).La Filosofía de la Investigación Social, Breviarios, Fondo de Cultura Económica, México*, 1959.

[2] N. Juristo and O. S. Gómez, "Replication of software engineering experiments," in *Empirical software engineering and verification*. Springer, 2010, pp. 60–88.

[3] C. Laine, S. N. Goodman, M. E. Griswold, and H. C. Sox, "Reproducible research: moving toward research the public can really trust," *Annals of Internal Medicine*, vol. 146, no. 6, pp. 450–453, 2007.

[4] R. D. Peng, F. Dominici, and S. L. Zeger, "Reproducible epidemiologic research," *American journal of epidemiology*, vol. 163, no. 9, pp. 783–789, 2006.

[5] I. T. Coyne, "Sampling in qualitative research. purposeful and theoretical sampling; merging or clear boundaries?" *Journal of advanced nursing*, vol. 26, no. 3, pp. 623–630, 1997.

[6] P. J. Guo and M. I. Seltzer, "Burrito: Wrapping your lab notebook in computational infrastructure," 2012.

[7] S. Haefliger, G. Von Krogh, and S. Spaeth, "Code reuse in open source software," *Management science*, vol. 54, no. 1, pp. 180–193, 2008.

[8] M. Sojer and J. Henkel, "Code reuse in open source software development: Quantitative evidence, drivers, and impediments," *Journal of the Association for Information Systems*, vol. 11, no. 12, pp. 868–901, 2010.

[9] J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer, "Example-centric programming: integrating web search into the development environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2010, pp. 513–522.

[10] A. Saltelli, S. Tarantola, F. Campolongo, and M. Ratto, "Sensitivity analysis in practice: a guide to assessing scientific models," *Chichester, England*, 2004.

[11] N. Barnes, "Publish your computer code: it is good enough," *Nature News*, vol. 467, no. 7317, pp. 753–753, 2010.

[12] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. C. Hong, M. Davis, R. T. Guy, S. H. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley *et al.*, "Best practices for scientific computing," *PLoS biology*, vol. 12, no. 1, p. e1001745, 2014.

[13] S. M. Baxter, S. W. Day, J. S. Fetrow, and S. J. Reisinger, "Scientific software development is not an oxymoron," *PLoS Computational Biology*, vol. 2, no. 9, p. e87, 2006.

[14] J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson, "How do scientists develop and use scientific software?" in *Proceedings of the 2009 ICSE workshop on Software Engineering for Computational Science and Engineering*. IEEE Computer Society, 2009, pp. 1–8.

[15] L. Nguyen-Hoan, S. Flint, and R. Sankaranarayana, "A survey of scientific software development," in *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2010, p. 12.

[16] J. C. Carver, R. P. Kendall, S. E. Squires, and D. E. Post, "Software development environments for scientific and engineering software: A series of case studies," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 550–559.

[17] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and software technology*, vol. 56, no. 10, pp. 1219–1232, 2014.

[18] D. Heaton and J. C. Carver, "Claims about the use of software engineering practices in science: A systematic literature review," *Information and Software Technology*, vol. 67, pp. 207–219, 2015.

[19] B. A. Myers, A. J. Ko, and M. M. Burnett, "Invited research overview: end-user programming," in *CHI'06 extended abstracts on Human factors in computing systems*. ACM, 2006, pp. 75–80.

[20] J. Segal, "Some problems of professional end user developers," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2007)*. IEEE, 2007, pp. 111–118.

[21] ——, "Software development cultures and cooperation problems: A field study of the early stages of development of software for a scientific community," *Computer Supported Cooperative Work (CSCW)*, vol. 18, no. 5-6, p. 581, 2009.

[22] ——, "Scientists and software engineers: A tale of two cultures," 2008.

[23] C. Morris and J. Segal, "Some challenges facing scientific software developers: The case of molecular biology," in *2009 Fifth IEEE International Conference on e-Science*. IEEE, 2009, pp. 216–222.

[24] M. B. Kery, A. Horvath, and B. A. Myers, "Variolite: Supporting exploratory programming by data scientists." in *CHI*, 2017, pp. 1265–1276.

[25] S. Chasins and R. Bodik, "Skip blocks: reusing execution history to accelerate web scripts," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, p. 51, 2017.