A Theory of Scientific Programming Efficacy

Elizaveta Pertseva
epertsev@ucsd.edu
Stanford University
Stanford, California, USA

Melinda Chang
melindachang.hy@gmail.com
Canyon Crest Academy
San Diego, California, USA

Ulia Zaman
uzaman@uci.edu
University of California Irvine
Irvine, California, USA

Michael Coblenz
mcoblenz@ucsd.edu
University of California San Diego
La Jolla, California, USA

ABSTRACT

Scientists write and maintain software artifacts to construct, validate, and apply scientific theories. Despite the centrality of software in their work, their practices differ significantly from those of professional software engineers. We sought to understand what makes scientists effective at their work and how software engineering practices and tools can be adapted to fit their workflows. We interviewed 25 scientists and support staff to understand their work. Then, we constructed a theory that relates six factors that contribute to their efficacy in creating and maintaining software systems. We present the theory in the form of a *cycle of scientific computing efficacy* and identify opportunities for improvement based on the six contributing factors.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; • **Human-centered computing → Empirical studies in HCI**; • **Applied computing → Physical sciences and engineering**.

KEYWORDS

Scientific programming, qualitative study of programmers

ACM Reference Format:
Elizaveta Pertseva, Melinda Chang, Ulia Zaman, and Michael Coblenz. 2024. A Theory of Scientific Programming Efficacy. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639139

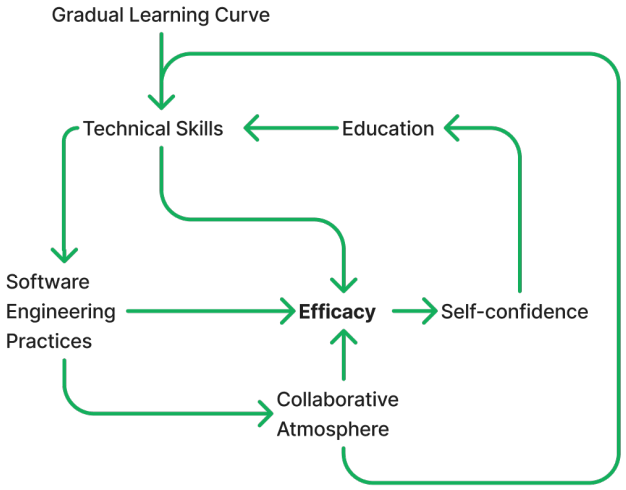

Figure 1: A Theory of Scientific Programming Efficacy. We present our theory as a cycle, with six factors contributing to increased efficacy. A green arrow between two factors indicates that an advancement in the first factor leads to an advancement in the next one. For example, we found that an increased feeling of self confidence made scientists feel that they could attain additional education.

1 INTRODUCTION

Many scientists, such as oceanographers, physicists, climate scientists, and economists, create and maintain significant software systems to develop, assess, and leverage scientific theories. Although they do not generally have software engineering training, scientists create large software systems that model physical systems to predict future conditions; analyze satellite data; control mobile remote sensing systems; and visualize data to communicate their results with others. To produce correct and timely results, scientists must be able to effectively write software that meets their objectives.

Our primary research question is: **What factors relate to scientists' success in writing software to meet their needs?** Prior work [11, 16, 34, 36, 43] has examined what techniques scientists use when developing software; we focus on the factors that affect scientists' efficacy. In this paper, *efficacy* refers to a scientist's ability to create software that meets their needs at minimal development cost. Despite the central nature of software development in their work, computational scientists differ from software engineers. Many scientists do not view software as their primary deliverable, making them end-user software engineers [25]; they often lack experienced SE colleagues; and it often suffices for their software to work for the specific input data they are interested in analyzing.

Despite software being central to their success, scientists in many cases do not adopt tools and practices used by professional software

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639139

engineers [11]. Rather than assuming the solution is additional training, we were interested in a second research question: **To what extent do tools and processes from the software engineering community need to be replaced, modified, or augmented to be as effective as possible for computational scientists?** The long-term aim is to identify opportunities for interventions that could make scientists more effective at software development.

We interviewed 20 scientists and five computing support staff. Motivated by the climate crisis, we started with a focus on scientists working on climate-related topics. We then broadened our scope to include other kinds of computational scientists as well as staff at high-performance computing centers who support scientists in writing and running software.

We used techniques from grounded theory [7] in our analysis of the interview transcripts, which enabled us to identify relationships among processes the scientists used. Our resulting theory identifies factors that make computational scientists effective at creating and maintaining software that meets their requirements. Key components of the theory concern:

- Alignment of engineering practices with scientists' needs
- Impact of collaborative environments
- Role of self-confidence in knowledge advancement
- Accessibility of targeted educational experiences
- Relevance of domain-specific technical skills
- Presence of gradual learning curves in development tools

Figure 1 shows the theory. *Software engineering practices*, when well-matched to scientists' needs, *technical skills*, and *atmospheres that support collaboration among scientists* all contribute directly to scientists' *efficacy*. Efficacy results in feelings of *self-confidence*, which makes scientists feel that they can attain additional *education*. Obtaining new knowledge, particularly when the education concerns skills that can be learned *gradually*, results in stronger technical skills. Gradual learning curves are characterized by technologies in which small amounts of effort are met by small amounts of reward, rather than requiring a large amount of up-front study.

Our diagram does not show every possible relationship; surely, for example, education could promote self-confidence directly. Our focus is on the most prominent relationships that we observed.

In summary, this paper's key contributions include:

- A theory describing a *cycle of scientific computing efficacy* based on 25 interviews of scientists and support staff
- A discussion of opportunities to make scientists more effective at engineering scientific software

2 LIMITATIONS

Our interpretivist analytic approach, which underlies the constructive grounded theory method [7], does not attempt to derive a singular true theory. Instead, we sought to describe software development as *we understand* our participants to experience it. We inferred category relationships from a combination of participants' statements and our insights about their work. For each category, we describe specific pieces of evidence from interviews that led us to posit the relationships we report, but space constraints preclude us from describing all evidence in detail. Future work may identify relationships that we did not find and could lead to important theoretical refinements. As with Box's observation [4] that "all models are wrong, but some are useful," the possibility of future refinements to our theory does not make it less useful.

Our perspectives as researchers in software engineering, programming languages, and human-computer interaction may have influenced our theoretical development. Researchers with other backgrounds may have identified relationships that we did not, leading to a different theory. Also, an early emphasis on iterative analysis, as would be typical in grounded theory, might have lent additional depth to the analysis.

Because of the difficulty of recruiting expert participants, our participants represent a convenience sample. However, practices and challenges may differ across organizations or fields. We recruited broadly because a diversity of perspectives strengthens our theory, but future work that includes scientists from different backgrounds, organizations, or fields may identify additional factors.

Finally, because we did not directly observe the participants working, our results are limited according to participants' recall and the questions we asked.

3 RELATED WORK

Our work is the first in this area, of which we are aware, that centers on *theoretical development*. As a result, we were able to elicit a theory that describes the process of achieving efficacy, as opposed to focusing solely on coding practices. Prior studies have either prioritized a focus on industry [11] or employed a more rigidly structured approach, which limits depth and is less conducive to theory-building [36].

Software Engineering Practices. Prior work has shown that some computational scientists, working in large teams, use traditional software engineering practices that reflect agile and open-source development, including code review and continuous integration. For example, Easterbrook et al. [11] found that teams of scientists working on climate models at a national weather service in the U.K. employed practices similar to those of professional software engineers. Some large research groups even include dedicated research software engineers [9]. Our study primarily focused on academic scientists who worked individually or in smaller teams and lacked funding for software engineers.

Nonetheless, congruent with our findings, prior work showed that traditional software engineering practices can have significant shortcomings for these kinds of scientists [10]. Because many scientists' work is exploratory, they have difficulty validating their requirements. Then, because in many cases the software simulates experiments that are impossible or too expensive to run, it is often unclear how to verify that their implementations are correct [11, 20, 34]. As Prabhu et al. [36] found in interviews with 114 scientists, most scientists do not rigorously test their code. This could have significant implications for the correctness of published scientific results. Additionally, smaller scientific teams often gather requirements informally; track issues using informal means, as opposed to dedicated issue trackers; and adopt techniques from methods such as agile development piecemeal rather than engage in well-understood processes. [19, 41]

Many scientists view software artifacts as a means to an end and software development as secondary to scientific work [19, 20, 36]. This results in a different outlook on software development than

might be common among software engineers, who value software directly. This difference in perspective contributes to a decreased emphasis on artifact quality—more important is to obtain scientific results efficiently.

HPC. Scientists' struggles with high-performance computer systems (e.g., supercomputers), have been well studied in prior work. Leveraging these resources requires expertise in parallel computing; the techniques needed to achieve high performance are not easily learned [3] and supercomputers can be hard to use in general [50]. Indeed, scientists worry about the motivational challenges involved in learning computing, which requires persisting through repeated failures [31]. It is not obvious what tools to use: C++ is in common use but is unsafe; Rust can be hard to learn [14]; object-oriented languages are not a good fit for the kinds of high-performance parallel computation that is needed for scientific computing [3]. This is particularly unfortunate given the origin of object-oriented programming in physical simulation [33].

Education. Wiese et al. [45] found that the most common issues that scientists struggle with are technical problems (requirements, management, testing, and debugging). Thus, scientists need education to become effective at writing software. Some limitations of scientific computing education opportunities have been identified in prior work. Hannay et al. [16] surveyed 1972 participants regarding how scientists develop and use software. They found that resources for learning about software engineering were limited to peers and self-study rather than formal education. The Software Carpentry curriculum [46] provides a brief formal educational opportunity and has helped more than 34,000 researchers since 2012 [42]. Although it includes one lesson on Git, it does not cover other critical software engineering concepts, such as design or code review. Programming-related instruction for scientists does not generally include software engineering techniques; for example, of 79 bioinformatics degree programs, only two required a software engineering course [44].

High-performance computing (HPC) can be used as an opportunity to teach broader computational skills. Lathrop et al. [27] described lessons from teaching HPC skills, including the need for lifelong learning. Collaboration, even with AI systems, is another approach to education: Kuttal et al. [26] proposed using intelligent agents to simulate partners in pair programming. Non-technical skills may also be helpful: Prickett et al. [37] found that *resilience* correlated with performance in first-year computing curricula.

Wilson et al. [49, 50] and the Computational Infrastructure for Geodynamics [1, 21] developed sets of recommended practices for scientific computing. However, studies indicate that these practices often fail to become mainstream in computational science [11, 50]. Consistent with our findings, Killcoyne et al. [24] identified developer isolation as a key barrier to the adoption of better software engineering practices in computational life sciences and Kellogg et al. [21] suggest that the challenges faced by the scientific computing community are as much social as they are technical.

4 METHOD

Participants. We recruited $N = 25$ participants, shown in Table 1 with anonymous identifiers P1, P2, etc, in the order in which they were interviewed. Twenty participants worked as scientists, and five worked as computing support staff. Five of the participants identified as female and twenty as male. Fifteen participants held Ph.D. degrees. All twenty scientists, except for the three professors, directly wrote research software. To further understand our participants' demographics we asked our participants to complete an *optional* post survey.**14/20** scientists completed the survey. We did not record the results of the supporting staff as their coding skills were not the focus of our study. The results were not used to build the theory and are provided for demographic purposes. We report the results of the survey in tables 2 and 3. Based on our conversations, we estimate that typical projects ranged from 200 to 10,000 lines.

Recruitment. We identified possible participants by studying our institutional directory and snowball sampling (referrals from other participants). While initially, we reached out primarily to climate scientists, our participants referred us to economists and mathematicians. After conducting these initial interviews we wanted to assess to what extent our observations might generalize to other kinds of scientists, so we recruited more broadly from economics, physics, linguistics, and bioinformatics. Following a *theoretical sampling* [7] approach, in which we sought data in areas in which we had incomplete evidence, we also recruited computing support staff with technical expertise who could offer another perspective on the scientists' experiences. Nonetheless, eleven participants came from oceanography and climate science, reflecting our initial focus. Some oceanography and climate science interviews also occurred later in the study due to scheduling constraints.

Interview Process. Our interview guide [35] had 24 questions in three categories: "Background/General", "SE Specific", "More Broad". Our questions include broad questions, such as "Which parts of your coding workflow are most challenging?" and questions about SE practices such as version control, testing, and collaboration. All interviews were semi-structured. The first author was present for all of the interviews while the other authors attended a subset. The interviews were conducted in person and over Zoom and lasted up to an hour and a half. We recorded the interviews and transcribed them with the Otter.ai transcription service.

Analysis. We adapted techniques from constructivist grounded theory [7] to analyze the interviews. We open-coded the transcripts, resulting in 1530 coded sections with 1201 different codes [35], and then grouped the codes to form seven most-salient categories. All of the final categories had codes from at at least three or more participants. We then compared categories and cases to each other to identify relationships, forming a theory of how the categories related to each other. After Charmaz [7], we inferred relationships by synthesizing quotes, participants' behavior, and our background knowledge from our interviews. Figure 2 shows an example of how we inferred a relationship from a quote. Finally, we expressed each of the relationships as an edge in a graph (fig. 1).

5 RESULTS

In the sections below, we explain each component of our theory in more detail. For each component, we explain how its presence affects other components and overall development efficacy.

Table 1: Participants' backgrounds. *indicates computing support staff. In columns 4 and 6, some of the participants were involved in additional coding tasks or education opportunities that did not contribute as much to their current work/knowledge and were omitted for space. In column 5, some of the participants did not tell us how much coding experience they had.

#	Field	Position	Main Role of S.E. in Research	Yrs. Coding	Main S.E Education
P1	Oceanography	Post Doc	Modeling Physical Systems	8	College Classes
P2	Oceanography	Proj. Scientist	Data Pipelines	20	Workshops
P3	Oceanography	Professor	Server management	-	E.E degree
P4	Oceanography	PhD Student	Data Analysis	5	Workshops
P5	Economics	Researcher	Data Analysis	22	Self-taught
P6	Oceanography	Researcher	ML Applications	-	Self-taught
P7	Mathematics	Researcher	Predicative Modeling	10	Workshops
P8*	Geosciences	Engineer	Support	25	DSE Degree
P9	Physics	Professor	Advising Students	30	College Classes
P10	Linguistics	PhD Student	Web Scraping	-	Workshops
P11	Climate Science	Post Doc	Predictive Modeling.	-	College Classes
P12*	HPC	User Support	Support	-	C.S Degree
P13*	HPC, Climate Sci.	Research Staff	Support	45	Self-taught
P14	Bioinformatics	Professor	Advising Students	20	C.S Degree
P15	Climate Science	PhD Student	Data Analysis	5	Workshops
P16	Oceanography	Lab. Director	Advising Students	-	Self-taught
P17	Oceanography	PhD Student	Modeling Physical Systems	4	College Classes
P18	Oceanography	MS Student	Data Analysis	6	Workshops
P19	Economics	Pre Doc	Statistical Analysis	3	College Classes
P20	Economics	PhD Candidate	Predictive Modeling	-	College Classes
P21	Bioinformatics	PhD Student	Model Comparison	3	Workshops
P22	Global Policy	Post Doc	Data Analysis	-	College Classes
P23*	Research IT	SI Engineer	Support	20	College Classes
P24	Oceanography	Post Doc	Data Ingestion/Analysis	11	College Classes
P25*	Data Curation	Librarian	Support	7	Work training

Quote: "One large set of bugs is from mistakes in configuration [...] We're dealing with errors around 25 or 30% [of the time...] I was looking over one of my student's shoulders and was—this tells you something about the fact that I'm not doing code reviews—horrified to see that the file that we're working on was 8000 lines long and was basically two functions [...] We took that apart [and] rewrote it as a package [...] It is much better given the situation." (P9)
Inference: Lack of supervision leads to messy code leads to errors.
Codes: *Puts off refactoring*, *messy analysis*
Category: Collaborative Atmosphere
Relationships Lack of collaboration lowers efficacy
Explanation: Grad students often have little supervision writing code and little incentive to write clean code, leading to errors and advisor mistrust.

Figure 2: An example of how relationships between categories were inferred from data

5.1 Software Engineering Practices

We asked participants about practices from prior literature, including language use, testing, bug tracking, version control, and collaboration, that we hypothesized would relate to efficacy. We also asked participants to describe their programming workflow to elicit general information about practices.

As scientific computing practices have evolved, participants have gradually adopted software engineering practices from the software industry that they felt led to improved collaboration and efficacy. P7 explained, "We're starting to adopt [...] version control [...] test-driven development, much more emphasis on modular code design, standard APIs. And these processes are really maturing the field."

Some practices and tools were inappropriate for their contexts. Testing frameworks do not align with scientists' workflows; version control systems have unfamiliar interfaces. Applying version control systems and leveraging design principles require up-front investment but offer delayed, nebulous returns. Some scientists invented practices, such as makeshift version control systems and testing through manual analysis of visualizations, to fill the gaps.

Scientists also reported using programming languages that were not designed for their needs. Python is a general-purpose language not focused on efficiency; R's design focuses on statistical analysis; C provides high performance but requires low-level programming skills. Julia is designed for high-performance scientific computing but may not facilitate reasoning about the correctness of high-level data analyses.

Table 2: S.E tasks. Participants were asked to select one or more of the below options in response to the question "Which of the following tasks do you regularly perform for your research?"

Task	% of Responses	Num. Responses
Data Analysis	92.9%	13
Graphs/Plots	92.9%	13
Array operations	85.7%	12
Matrix operations	71.4%	10
Data wrangling/ organization	64.3%	9
Optimizing code performance	64.3%	9
Converting formulas to code	64.3%	9
Building models/ predicting the future	42.9%	6
Monte Carlo Simulations	42.9%	6
None of the above	0.0 %	0

Table 3: Programming Languages. Participants were asked to fill in the blank in response to the question "What programming languages do you code in for research?"

Language	% of Responses	Num. Responses
Python	92.8%	13
Matlab	50.0%	7
Fortran	35.7%	5
Bash	28.6%	4
R	21.4%	3
C++	21.4%	3
Julia	14.3%	2
Rust	7.1%	1
Java	7.1%	1
Mathematica	7.1%	1
None	0.0%	0

Reuse and Maintenance. Scientists reported re-using code from libraries and published papers but mentioned that they were concerned about its impact on the accuracy of their results. P6 explained: "One class of bug [that] most scares me [is] I see a lot of academic people using [...] code as a black box [...] You don't know what were the constraints and assumptions of the person who wrote the code [...] There is no way to validate to check if it's going as expected." P17 also said that due to "ambiguity in science" and noise in experiments he cannot use data to check the accuracy of legacy code: "I'm trusting that it can get interactions correctly [...] Maybe it is overestimating things a bit."

Other impediments to reuse include software that no longer works: Scientists rely on a lot of libraries in their code [19], but these libraries can introduce bugs with version updates or become discontinued. P12 explained: "Often you'll see the codes that they developed five years ago don't work — no one's maintaining them."

Of those who reported publishing their code for reuse in GitHub or Zenodo [12], only two mentioned maintaining their code. Similarly, while libraries sometimes provide institutional repositories to publish data and associated code, the staff did not have the training to test artifacts and maintain dependencies (P25).

Testing and Risk. Nine of the 20 scientists we interviewed reported attempting to write unit tests, but the majority found the process futile and inapplicable to their work: P20 elaborated: "It isn't super common to organize your code in such a way that a unit test makes sense." Scientists reported that there were often too many cases to test (P19); too many small functions (P17); not enough functions, in the case of exploratory code (21); or too many frameworks (P9). Instead, to verify the correctness of their code, scientists generated visualizations and confirmed that they visually corresponded with their expectations. As P2 explained: "There will be so many instances [when] we know what the theoretical expectations should look like." This method often allowed them to quickly spot errors. For example, P11 found an indexing bug when the error in her forecast did not increase with time, and P21 identified mislabeled data due to a strange cluster in an evolutionary distance tree. Scientists would also sample individual data points and use logical criteria to ensure points were within appropriate ranges (P15). While P2 mentioned that bugs that look right on plots are "very rare," manual testing can still be error-prone; P15, for example, explained how she could not find a copy-and-paste error that resulted in multiple copies of the same file being imported in a graph. The majority of bugs this method did not catch tended to be minor, e.g., those about details of figures, but our participants still worried about the risks. P4 said: "There's probably a good chance I have some bug in my code that I haven't found yet [...] I try my best to clear everything out and make sure everything's working. But I don't know [...] there's so much code, there's so many lines, I feel like you're bound to do something [wrong]."

The difficulty in verifying code changes can result in scientists avoiding making changes that could benefit science. For example, scientists who deployed expensive hardware systems worried about the risk of a bug causing a mobile remote device to be lost. P3 explained that only one lab in the world has the testing infrastructure to validate changes to the software for a particular sensing system. As a result, his group avoided making changes.

Version control. In large and long-term projects, some scientists leveraged version control. For example, P16 used version control for a MATLAB program used to control remote sensors. A more popular approach was using version control systems only for collaborative projects (P5, P21, P15). Instead, scientists developed individualized, local workflows. P21, for example, said: "I probably should be [uploading to GitHub]. But I haven't yet [...] because it's not collaborative." P19 explained: "It's still not standard practice in economics at all to use version control, which quite frankly scares me. And I think [...] most labs just kind of manage everything via Dropbox." P19 explained his path to using version control: "Before starting to use version control [...] six months pass, and you realize that some analysis that you have done before [...] [is] actually sort

of useful, right? And then if you have to [...] rewrite it every time, it can get quite annoying, and laborious, and time-consuming." P22 wished for a record of edits in Jupyter Notebooks and a way to keep related notes. P22 was aware of GitHub but found it hard to use.

Some participants replicated features of version control systems with bespoke procedures. P18 described keeping a folder called "bad code" in which she stored previously-written code in case she might need it again. P18 used GitHub for backup purposes rather than version control: '[E]very few days, I'll just move everything [...] into my GitHub folder, and then push it and [have it] just in case." When asked why she doesn't use GitHub more, P18 explained "I'm just not super comfortable with the interface."

P15 described using GitHub to publish code at the end of a project: "Certain grants [...] require you to push your things to GitHub." This is consistent with Cadwallader's findings [5]: linking to repositories was a common method to share code.

Process and Design. Some participants explicated their view of coding as a primarily exploratory process. P14 explained: "You have to do a lot of exploration before you have something that will make it into the paper." As a result, few participants reported developing plans or using high-level design techniques. P17 gave an example of his process: "Every time that I code, I think 'Okay, what is the equation and then what's the easiest way I can turn that equation into a code [...] That might not be [...] the fastest or most efficient way to do things. Every time that I write something, it starts out very, very clumpy." Similarly, other participants reported starting with equations and results from papers, attempting to replicate them, and then attempting to build upon the results.

When building upon previous work, scientists struggled to differentiate between bugs and novel findings. P17 explained, "There hasn't been any work done on it. So if anything I find [...] I don't know how to check that." Discrepancies from expected outputs could be caused by gaps in the theory, bugs in the implementation, or issues with the data. As a result, debugging was often a very time-intensive, unpleasant process for scientists. Nonetheless, only two participants reported using bug-tracking systems. The majority of participants reported using visualizations to debug, similar to our observations about testing. After obtaining the final results, participants decided whether or not to share their code—if they chose to do so, they needed to invest considerable time into code refactoring. P14 added that this exploratory process can lead to lost code: "Occasionally [...] you never quite document or release little scripts that you did if they're too trivial, or sometimes even when they are not trivial. You're not careful. It's just kind of lost."

Collaboration. Traditional software engineering contexts result in software artifacts that are shared among collaborators, promoting a collaborative atmosphere of code reuse and co-design. In contrast, we found that only 2 of 20 scientists were working on an ongoing project with multiple programmers. Some participants worked together writing code but did not report structured methods such as pair programming: "I don't know how we're going to divide up the work. But I imagine it'll be mostly on his computer, and then I'll be sort of stepping in (P17)."

Participants felt a lot of code in their field was not designed to be shared, further impeding collaboration. P5 explained: 'People will request the code and [...] I have to go through and clean it all [...] It would be a lot of extra work to go back and refactor everything as I go along, [but] it would be so worth it." Recipients of code agreed. P3 said, "The code you get is not very clean, and is meant to run on someone's computer, but has never been tested anywhere else. So the difficulty will result in first getting it to run on your computer."

5.2 Collaborative Atmosphere

Participants' access to a *collaborative atmosphere* — the ability to get a second opinion on a project from an advisor, peer, co-author, or technical support — played a crucial factor in efficacy.

Mentors. Due to the specialization of academic research, six of the 20 scientists we interviewed worked on solo projects, with only their mentors as a source of collaboration. While collaboration with mentors was helpful, advisors' technical expertise was in some cases insufficient. Advisors' experience sometimes constrained tool options, since students often chose approaches that would enable them to get advice from their advisors.

P10, one of the only Ph.D. students in her department working on a computational project, discussed her worries about having a non-technical advisor: "I have very little oversight from any faculty [...] I'm always figuring things out, maybe not in the smartest way [...] I know the whole thing and have a lot of problems."

P18 recounted her desire to use Python: "And then when I started my project [...] I had asked [my advisor] 'Can I do Python?' and he was like, 'You can do whatever, but I can only help you in MATLAB' I'm gonna need help. So yeah, I decided to do it in MATLAB."

Scientists' willingness to use potentially less efficient or more complex tools in favor of collaboration suggests how crucial a mentor's feedback can be for a project's success. P4, another oceanography Ph.D. student, recalled a time when he spent hours trying to fix a copy and paste error – "we went on tangents trying to figure out like, what's going wrong?" The error was only discovered by his advisor: "with a fresh pair of eyes [...] my advisor found it, I eventually sent it to her and then she found it."

However, there were also cases when the students' needs pushed advisors to learn more effective scientific computing practices, emphasizing the mutual benefit of collaboration. P11 helped her advisor set up GitHub: "She [advisor] is not very well versed in version control [...] But [she] is trying to be very helpful with my research and trying to look over my shoulder (P11)."

Peers. In bigger lab groups, *peers* provided another source of collaboration. Even those working on different projects were often more aware of novel technical tools and practices and thus were the driving force behind the technical advancements of their labs.

P1 described how, despite his advisor knowing only MATLAB, the lab atmosphere propelled him to learn Python, which he found a better fit for his projects due to its syntax and libraries: "There were a couple of graduate students that I really respected. And I was moaning at them about how much I hated MATLAB. And then they were like, 'No, dude, you got to use Python.'" P19 recounted that while the rest of his lab used Dropbox to share code, a past research assistant helped transition his project to GitHub.

Peer mentorship inspires the mentees to become mentors themselves and help less technically advanced students. P4 discussed how he learned many programming techniques and practices from

a graduate student who has left the lab: "I would definitely not be where I am today if I didn't have someone as patient and dedicated working with me [...] She spent hours and hours with me, working on my code or helping me understand concepts." Now, P4 wants to "pay it forward" and promote efficacy in his lab. In turn, P17 benefited from P4's help: "He wrote a bash and Git workshop that he gave, last year [...] He's good, very helpful."

Students who do not have a collaborative peer group must search for help elsewhere, but these sources are not always as helpful as someone with domain-specific knowledge. P10 lacked peers with computational linguistics expertise, so she often found help in the cognitive science department.

Co-Authors. Only two of our participants were working on an ongoing programming project with more than one contributor, although others agreed that team programming practices would promote efficacy. P2 stated, for example: "And so you do your own thing in your computer, and you don't [share it] sometimes."

Participants agreed that collaboration would promote better practices such as version control, improving the readability of code, and making code easier to package. P17 explained that collaborating on GitHub creates "an incentive [...] to make that [code] pretty good," as opposed to the comparatively lackluster code he writes for himself.

Technical Support. The final avenue for collaboration was reaching out to *technical support* — staff hired to support scientists with a variety of technical problems, ranging from high-performance computing (HPC) to data curation. While such support can be extremely helpful for scientists to complete their projects, its impact can be limited by resource constraints and communication barriers.

P8 talked about how he can onboard new scientists and help them understand the complicated code base and techniques: "We'll save you months of trying to figure out what to do, and put you on the right track, and then guide you along." P13, research staff who works at the same center, helps scientists set up packages and compile climate models on supercomputers: "Some just really struggle with this. And then they come to me, my nickname was flag lady: which flags do I need to get this to compile?"

Support staff were often overwhelmed by the demand. P12, who is a user group support lead at a supercomputer center, stated: "We have a small group [...] we can't cover everything. So then the gaps come up even from [...] [the] support side." For example, staff reported not being able to check the code itself for excessive I/O access, sudo commands, or inefficient parallelization, all of which can cause slowdowns or even crashes on the supercomputer. Participants agreed that the limited support staff sometimes could not address their needs; P21 reported struggling to install dependencies on a supercomputer using containers: "I tried talking to the administrator in that case, and they didn't help me out that much."

There also exists a language disconnect between the scientists and staff due to their different technical backgrounds. P23, a Systems Integration Engineer who focuses on helping scientists build data pipelines, said, "The communication barriers can make it way more difficult to ensure that you're not frustrating them [by] giving them a recommendation that isn't going to help."

When giving advice, staff worried that scientists were ill-prepared to use the best techniques, leading them to suggest suboptimal alternatives. P23 explained why he seldom urges the scientists to use the cloud or personal workstations: "You don't want to give someone a resource that suits them in terms of their combined computational needs, but they don't have the organizational or other resources to manage." P23 tried to spend more time fully understanding the problems and capabilities of each of the scientists he works with. Unfortunately, the time-intensive nature of this approach sometimes renders it impractical.

ChatGPT. For scientists who work mainly alone, ChatGPT may act as a substitute for missing collaboration opportunities. Seven out of the 20 scientists reported using ChatGPT in their development process.

P22, a postdoc who mainly works alone on her projects, finds ChatGPT useful when interfacing with unfamiliar APIs such as Google Earth Engine: "I just asked ChatGPT to provide me [with] a very basic framework to see how I can link my personal drive to [it]." Scientists tended to prefer using ChatGPT over code off GitHub due to its modularity and fast response time. When asked if he uses other people's code, P5 responded, "No, but lately, GPT has been giving me good code that I can just pull in quickly."

In many cases, scientists relied on ChatGPT as a guide and thus tolerated its errors. P5 stated: "I mean, it often gives me code that's wrong [...] But usually, I'll just go back to it and say 'Well, that didn't work.' And it'd be like, 'My apologies, I was incorrect [...] Try this' [...] and that doesn't work either. But it usually gives me enough of an idea of which direction to go." P19 agreed: "Sometimes [ChatGPT] is not right. But it's always a start."

P23 hypothesized that ChatGPT is so popular because scientists are already predisposed to an iterative mode of coding: "Iterative dialogue, where you write something, you test it, you write something, you test it, and you go back and forth." P14, who earned a Ph.D. in Computer Science before becoming a professor in bioinformatics, corroborated this notion: "For someone with a software engineering background [...] I don't think of [bioinformatics work] as programming. I think of it as talking to the computer."

5.3 Self-confidence

Learning to write software requires assuming the risk that studying a technique will not be effective. When taking risks was rewarded with success, participants were more likely to seek out new techniques and technologies to improve their ability to develop software. This led to a feeling of *self-confidence*—the belief in oneself to effectively use and learn new tools. *Motivation theory* argues that if a learner's primary motivation is performance and not intellectual fulfillment, then self-confidence plays a crucial role in motivation to learn new things [8]. Thus, for scientists, feelings of confidence when programming may motivate learning new programming tools or skills, enabling future successes. In other cases, failure or setbacks can lead to scientists feeling that work learning new techniques is unjustified. Then, they may focus their efforts on using familiar tools and processes, missing opportunities to advance.

Although P19 lacked formal training in programming, he said: "It would be fairly easy to get another data job in econ now, just because I do have some of the applicable skills." P19 was considering

embarking on a project with "security and authentication concerns" that he "need[s] to figure out how to deal with" even though they are "obviously not" his "area of expertise." P19's feelings of confidence led him to feel prepared to seek out informal education in relevant areas.

Similarly, P6, who viewed himself as "not a conventional academic" because he "always work[s] a lot with software" was willing to work on a communication project, undeterred by not knowing "anything about radio or satellite communication" and never having completed a project in Rust. Despite concerns about the "learning curve" and finding some concepts such as ownership and variable scope "hard to grasp at first", P6 ended up loving Rust more than Python, which he used previously.

In contrast, participants who found coding more daunting or were less secure in their background tended to shy away from learning new coding tools and techniques. For example, P21 reported that he found bash to be a "little bit stressful sometimes [...] I accidentally used a recursive chmod on my personal computer, and I locked myself out of it [...] It has so much power over your actual system." He currently waits for his job to finish rather than learning parallelization: "It's on my to-do list [...] At first, I thought it was going to be really easy [...] I haven't used it yet, but I know it exists and will be helpful." He did not give any reasons for not learning, such as lack of time or online resources.

Similarly, P10 reported that when she first learned R she would "cry every time she had to use it." Furthermore, she experienced "a lot of issues" using supercomputers because she uses scripts that she did not write herself. Nonetheless, she did not find this sufficient motivation to learn bash.

Seventeen of the scientists that we interviewed had some form of apprehension or anxiety about coding despite their experience. P5, with 22 years of coding experience, described his experiences with statistical analysis: "I have written the low-level code [...] in the past, but mainly just to explore and to make sure that I understood the concepts, not to [...] use as [...] production code. I don't think I would trust myself to write good enough code for that." In a 2023 Stack Overflow Survey in which 73% of the respondents were professional developers, over 50% of the respondents had less than 10 years of experience [2]. Thus, most people who write production code may have less experience than P5.

Similarly, P1, with 8 years of coding experience, states: "I am pretty sloppy in the way I document my code [...] I am sure that there's all kinds of stuff that I don't even know about." P2, with 20 years of coding experience, said of his process for transforming satellite data for future analysis, "I think there's probably a better way to kind of go about it but it wasn't obvious."

While many scientists had clear sets of additional skills they wanted to learn that would improve their coding, (section 5.4) few sought out these extra skills without formal education unless required by an advisor. P4 summarized scientists' general sentiments: "I kind of stayed within those things I learned and the libraries I've learned and worked with [...] I didn't really branch out."

This not only results in decreased efficacy but also foreshadows possible future employment difficulties. When describing the hiring process, P6 emphasized the importance of self-learning: "I will be looking for someone that's able to learn by themselves [...] because you cannot expect that person to know everything."

P9, a physics professor, felt that scientific computing is "behind" the software industry. He recalled discussions with a software engineer: "He likes to joke that when I [...] propose or write some code. He looks at it [and] he says, 'In industry, you'd be fired for that."'

Similarly, P8, a software engineer by training and currently support staff for climate scientists, called climate science a "dragon—way behind the other sciences" since from his perspective Jupyter Notebooks only became popular two years ago and "GPU computing and stuff like that, they're just non-existent, virtually." However, according to support staff who support all scientific disciplines (P23 and P25), climate science is one of the more technologically advanced sciences due to the large amounts of data they process.

Eighteen scientists experienced feelings of guilt over SE guidelines they felt they should follow, regardless of whether or not it confers them material benefit. P3 felt guilty about not testing like a software engineer despite being unable to identify a case where unit testing would have caught a bug: "The figure tells us a lot [...] so we know if things are wrong [...] but what you are supposed to do is write a test to confirm." P21, when asked about GitHub, responded: "Oh, I probably should be doing that [...] but I haven't had a reason to." P1 even expressed feelings of guilt over his IDE choice of Sublime: "I've heard that there are better ones [...] I should switch to PyCharm, but I haven't done it yet."

5.4 Education

Fifteen of the 20 scientists we interviewed indicated that they saw a lack of education as one of the primary barriers to efficacy in scientific computing, with a focus on the opportunity for education to foster technical skills. In this section, we explore the limitations of the main education opportunities identified by our participants: workshops, online resources, and university classes.

Workshops. Seven of our 20 scientists indicated that workshops were their primary source of education. The majority of the workshops were hosted by *Software Carpentry* [47, 48], a volunteer initiative committed to providing researchers with the basic computing skills they need for computational science. Software Carpentry's curriculum focuses on techniques such as Python, R, and GitHub.

Scientists argued that practical training was key to their success. P25, a librarian who has hosted several carpentry workshops, discussed the feedback from participants: "A lot of learners like [...] the hands-on live coding [and] the peer-to-peer feedback during the workshops." These workshops were particularly efficient ways of learning skills that they might not otherwise learn. P7, a math researcher at a weather center, said the workshops "helped me [learn] ideas around version control, [and] around scripting."

Despite the popularity of these workshops, their focus on practical skills may not provide sufficient background on abstractions. Participants 3, 10, 15, and 19 felt they could be more effective with a deeper understanding of the technologies they were using. P10 explained, "[I] want someone to [...] explain to me, fundamentally, what Python is and how it works [...] in terms of what an environment is, and what it's doing on the back end." Likewise, P19 said, "I don't think I have the fundamental conceptual understanding of what these different objects are and how they interact."

Although workshops aim to be domain-specific, instructors can only teach with small examples with small data sets and restricted

sets of dependencies due to time and resource constraints. This limits scientists' abilities to apply learned skills to their research. According to P4, these workshops could "only [...] skim the surface [...] and it's [their] job to [...] dig deeper." The workshop included domain-specific data sets, but P4 still had difficulty transitioning what he learned to the bigger data sets he used for research.

Workshop teachers often felt that, due to their limited expertise, they could not teach participants more advanced skills. P25, for example, led several workshops but did not consider himself an expert in the topics he taught. He said that workshop participants frequently follow up with requests for "'intermediate or advanced lessons"' that they do not offer due to lack of expertise.

Online resources. Corresponding with our observations about self-confidence (section 5.3), only three participants mentioned that they like to self-learn using online sources such as tutorials and documentation. P13 reported that tutorials often suffer from the same problems as workshops in focusing on "toy problems" that can be hard to generalize to "actual science." She further explained the gap that she sees between available resources and the educational needs of the scientists she helps: "You can go and read a Python book [...] But when it comes to using Python for your problem, that's when it gets hard. Because [mapping] what you want to do to the right Python function [is] where things fall apart."

University Classes. Ten of the 15 scientists who wanted more education had taken at least one coding class. Yet they did not view these as their primary source of knowledge, since the classes tended to align less with their needs than workshops or online resources. P17 summed up the problem: "I never had a class [that was] like, 'Here is how you code as a scientist.'" Courses offered by the CS department tended to focus on techniques that scientists felt were not applicable to their workflows. For example, P17's teacher emphasized unit tests, but he felt more confident testing via visualization since he had many very small functions (section 5.1). Introductory courses offered by the CS department also sometimes assumed prerequisites that scientists did not have (P2) and were hard to access as a non-CS student (P15). In contrast, classes offered outside of the CS department tended to focus more on the mathematics behind the concepts as opposed to writing efficient code (P17).

5.5 Technical Skills

Technical skills clearly influence scientists' efficacy. Many skills were domain-specific, such as using the Xarray [51] library for oceanography or the Stata statistics package [28] for economics. However, we identified three technical skills that were crucial to success in many scientific disciplines: managing and manipulating data; optimizing and predicting memory and storage usage; and generating complex, high-quality visualizations.

Data Management. P6 observed that he wasted a lot of time accessing data and fixing its format. Data could be contaminated or mislabeled (P21); omit observations (P7); be accessible only with software that cannot be scripted (P5); or stored in an inconvenient format (P21). Some disciplines used standardized formats, such as NetCDF [39]; others used proprietary packages, such as MATLAB. Data files could be so large that institutional repositories were unable to archive them; instead, they archived subsets (P25). When asked about common bugs, six of 20 scientists mentioned faulty assumptions they held about data. For example, P5 did not realize that a column was an average and not an actual value; P14 used a data set that had a different precision than he expected.

Even once the above issues were addressed, P8 observed that many scientists had difficulty writing I/O code that was suitable for supercomputers. Programs were often first written for desktop computers; when run in parallel, it generated too many I/O operations. The code needed to be modified to do I/O in larger batches.

Memory and Storage Consumption. Scientists often encountered problems with memory consumption when analyzing large data sets. Because of the opacity of garbage collection, Python programs that run out of memory can be difficult to fix. P8 explained: "Python garbage collection can be a little lazy occasionally. So learning how to force garbage collection on large memory usage [...] what are the tricks you can do to not run out of memory when you're doing this analysis?" P9 talked about needing to monitor memory requirements for programs that run on supercomputers because they restrict which computers the programs can run on. P7 reported doing "back-of-the-envelope" calculations to figure out how much storage a program would consume on a supercomputer when deciding which type of memory node to use.

As inputs get larger, memory challenges get harder. P1 worked with 8 GB matrices: "Python [...] has some kind of a limit where [...] even [on] huge computers, it just will crash." It can also be hard to predict memory requirements, especially when using libraries; even when inputs seem small, programs can inexplicably use large amounts of memory. P2 reports struggling with code interfacing with Xarray: "For very small requests, it will actually consume way more memory than I was expecting it did."

Visualizations. Scientists use visualizations to evaluate their code's correctness; to understand their models' behavior; and to report their results. Visualization libraries are thus key to many scientists' efficacy. P21, for example, described how he used visualizations to identify spurious data in a data set. P9 a physicist, makes heavy use of histograms to understand data: "Of all the various sciences I've had experience with, we are the heaviest users of histograms [...] our histogram libraries are lovingly created [...] a little bit too much religion is involved in creating histogram libraries."

Generating visualizations is difficult enough that it affects which languages scientists choose. Scientists often mentioned visualization libraries when discussing programming language features. P10 and P14 both liked R: "For R it's just basically availability of good packages, so if you want to do visualizations it's really hard to find anything better than ggplot" (P14). P7 preferred Matplotlib in Python despite also being familiar with R. Generating good visualizations requires significant expertise; P8 mentioned a local expert on generation images and charts. P8 also mentioned that sometimes charts are sent to artists, who generate the final images: "We give them a chart, and they make it pretty."

5.6 Gradual Learning Curve

Many scientists struggled to transition from beginner-friendly tools to more sophisticated ones. As a result, we conjecture that *gradual*

learning curves — contexts in which small amounts of study are rewarded with efficacy improvements throughout the learning process — are essential for technical skills acquisition. In this section, we focus on three examples that demonstrate how scientists have difficulty transitioning across gaps between systems.

GUI to CLI. While our participants found graphical user interfaces (GUI) to be helpful, many eventually found that they needed to learn command-line interfaces (CLIs).

P10 relied on the Anaconda GUI to manage environments. However, P10 also reported that the GUI could not help her solve package version discrepancies between her local and HPC environments. P5 and P7 reported that they could not find a reliable GUI to connect to multiple HPC instances and were forced to use the CLI. P3 explained that "most of" the GUIs for data visualization "are glitchy, and I don't really trust them." P3, P7, P13, and P14 reported that the CLI was much faster for their work than using an IDE.

However, as scientists tried to convert to CLI, they experienced a steep learning curve. P7, who mentored many students in his research center, elaborated: "There is a little bit of a learning curve just getting familiar with the idea of working from command line [...] For a lot of folks who are used to working [...] with very graphical interfaces, it can be [...] a leap." Similarly, P4, who mainly used an IDE to connect to his lab's workstation but had to use the CLI to transfer files, found that he was "not totally understanding" what he was writing and was mainly "trying to replicate" what someone else wrote. He blamed his "newness to working in [bash shell]." Only five scientists reported being comfortable with a CLI. P25, who taught CS introduction workshops for scientists, claimed that a large portion of his students are unaware of Unix shell capabilities: "We get a lot of people that say, 'Oh, I didn't know that you can basically control your [...] desktop from command line [...] I didn't know you could create a directory [...] This is neat.'"

Notebooks to Traditional Programming Environments. Scientists also struggled to transition from notebooks to writing scripts. P13, who supports scientists at a university-affiliated research center, reported: "We see people who are kind of stuck in Jupyter notebooks, which are awesome. But then they can't run outside the notebook. And when you get to these kinds of large, computationally heavy Fortran codes, you [...] [have to] run them on a big high-performance machine." P10, who transitioned from notebooks to scripts to run on an HPC system, elaborated: "I normally write in Jupyter notebooks and not Python scripts. So the format of the code needs to change substantially [...] I have to [...] test all the code [...] [but] it's very hard to [...] when you're just running it as a script."

Participants also reported that notebooks impeded packaging, as they were prone to creating "messy code" that took a long time to refactor to a script: "You're working with some code for a long time, you keep adding things to it. And at some point, you've created a monster." P11 added: "I think the packaging of the code would probably be better if we just don't think about the Jupyter notebook [...] And think more about packaging, just the functions plus [...] one notebook that tells the story." However, other participants (P1, P4, P17, P21, P22) often felt as though even pulling the functions out of a Jupyter notebook was difficult.

Local to HPC. Participants struggled to transition from working locally to working on HPC, not only because of the required transitions such as GUI to CLI and notebooks to traditional scripts but also due inability to test out and understand the differences between HPC and local such as IO access and batch jobs.

P7 explained: "I mean, there's definitely a steep learning curve from going from 'I'm just trying to write a script' to [...] high-performance computing. And I think that learning curve is prohibitive for many folks." P12, a user group lead who helps scientists with HPC, claims that the main problems come from the fact that "[Scientists] don't realize [...] it's scaling because [...] we have more of these computers, not because the single computer itself is a lot faster." As a result, scientists were often surprised when code that works locally does not work on a supercomputer.

6 DISCUSSION AND FUTURE WORK

Practices. Mismatches between existing software engineering practices and the practices used by scientists suggest that there is significant room for improved methods that are more applicable to scientists. Arguably, those we observed constitute *programming* but perhaps not *engineering*, missing opportunities to consider the long-term implications of their process and design choices. No participants reported following structured software processes, such as agile development—perhaps because they approach programming in an exploratory way. One approach could be a refinement of agile methods with a focus on practices for individual developers and helping scientists construct components whose properties they can reason about. Process-based interventions might also help scientists consider the *quality attributes* of the components they create earlier in their development process. For example, they could consider the extent to which they are reusable by others.

Property-based testing [30] has shown promise in general software engineering but has not been adapted to a scientific programming context. However, scientists' code often analyzes data; perhaps an adaptation of property-based testing could allow scientists to express ways of varying the input data to generate more test cases, and then express relationships between the inputs and outputs to enable some automated test generation and execution.

A discipline of scientific software engineering might be accompanied by more appropriate tools. Rather than assuming that tools designed for large-scale software are best, future research could focus on developing tools for small-scale or goal-oriented scientific software engineering. Some research has focused on bringing version control to data scientists [22] in Jupyter Notebook, but it may be possible to build support for other kinds of scientific workflows, such as a usable long-term history for bash commands.

Since much scientific code reflects data analysis pipelines rather than complex mutation of data structures, functional programming techniques may be more appropriate. A functional style may reflect the mathematical nature of scientific code and facilitate more effective reasoning about correctness [18]. A gap, however, may reside in performance; functional languages are often garbage-collected. Using an affine type system that adapts ideas from Rust, it may be possible to create a high-performance functional language that is suitable for high-performance computing.

Collaboration. Online platforms can be valuable for knowledge sharing inside organizations [40]. Ma et al. [29] found that a "sense of belonging to the learning community" is an essential factor in the successful engagement of participants on the platform. Since scientists do not identify as software engineers [19] and there are not enough opportunities for in-person collaboration, scientists may benefit from online platforms where they can discuss domain-specific code tips and tricks. Our participants echoed similar sentiments. P23 argued that GitHub Pages was not enough to build a community and P2 wished there were more forums to discuss code. In-person seminars or technical "office hours" might also aid in constructing communities within universities, lifting the burden of the technical staff and facilitating more peer collaboration.

The popularity of ChatGPT suggests conversational agents may be good interfaces for code help for scientists. The "interactive" aspect seems to mediate the lack of correctness and helps scientists start projects and find useful libraries. As previous work found that using ChatGPT requires "strong debugging skills" [32], extending chatbots to provide debugging help could be helpful. P23 proposed a chat interface to interact with supercomputers, which might be more effective than the spreadsheet-based interface employed by a large number of centers. Verifying the correctness of AI-generated code [23] and providing documentation of libraries used by AI-generated code may make the systems even more useful.

Self-confidence. Freedman et al. [13] claim that, by reducing self-esteem, guilt motivates for change. However, when the source of guilt is not under one's control, guilt may be maladaptive [6] (does not prompt change).

The prominence and success of the software industry weighed heavily on our participants, leading them to feel guilty for not adopting SE practices. In many cases, these practices were not appropriate for adoption off the shelf causing. Guilt lowered their self-confidence and may have impacted their motivation to learn.

While there has been work attempting to set more reasonable [50] expectations for scientists' practice of software engineering, they are limited and these approaches still view traditional SE practices as the standard. Developing more suitable practices for scientists may empower scientists to seek out more opportunities to boost their efficacy. Scientists often seek out and acquire new *scientific* knowledge on their own, so further research is needed to help them employ similar techniques to learn engineering tools that could boost their productivity.

Education. Currently available education may be failing to meet the demands of scientists as they struggle to adapt the material from lessons to their work. As Goble et al. found [15], scientific software engineering necessitates training that diverges from traditional computer science education. Such education may emerge from classes co-taught by computer scientists and scientists: while computer scientists have more technical knowledge, only scientists fully understand their needs. Assisting with Software Carpentry may be a good avenue for computer scientists to form collaborative relationships with scientists at their institutions.

Technical Skills. New tools could enhance scientists' efficacy. One opportunity could be tools that predict resource consumption and advise users as to which platform (local, supercomputer, cloud), is most cost- and time-effective, or a portable mechanism for re-targeting a single codebase to the appropriate computational platform. Tools for debugging plots may also assist scientists in validating their results.

Gradual Learning Curve. Resnick emphasized the importance of designing tools with low floors, high ceilings, and wide walls [38], meaning that there should be a low barrier to entry, high potential for complexity, and room for exploration. We propose that a *staircase*, could make it easier for users to gradually transition from the floor to the ceiling. One option may be developing a high-quality IDE for Bash: by making Bash more user-friendly, scientists might feel less intimidated by it. Another is constructing more refactoring tools for Jupyter notebooks to make it easier to transition to standalone scripts [17]. Building a testing framework for HPC could also limit anxiety over running code on supercomputers and could allow users to learn more about HPC.

Efficacy. Our theory focuses on opportunities for researchers and tool developers to improve processes and methods of software development for scientists. However, the theory could also be helpful for scientists for self-diagnosis to help themselves or each other. For example, an advisor might promote collaboration in their lab, facilitating peer learning and thus efficacy. A scientist who struggles with SE practices might focus on learning particular SE methods and becoming more self-confident in the process.

7 CONCLUSION

Our theory of scientific programming efficacy concerns six components: self-confidence, education, technical skills, gradual learning curve, software engineering practices, and collaborative atmosphere. In this paper, we argue based on an analysis of 25 interviews how these factors form a cyclic relationship. These factors and relationships offer opportunities to develop new tools and processes that could help scientists become more effective at writing and maintaining software, smoothing the path to scientific discoveries.

We hope that our theory and identified opportunities will help more scientists progress in their programming efficacy and see programming not as a barrier but as a helpful tool. Perhaps software can bring even more joy to science, as P5 declared: "I love coding [...] If I could do that all day and not have to write grants and like actually write papers, I would just do the coding."

REFERENCES

[1] 2023. Best Practices. https://geodynamics.org/software/software-bp.
[2] 2023. Stack Overflow Developer Survey 2023. https://survey.stackoverflow.co/2023/
[3] Victor R. Basili, Jeffrey C. Carver, Daniela Cruzes, Lorin M. Hochstein, Jeffrey K. Hollingsworth, Forrest Shull, and Marvin V. Zelkowitz. 2008. Understanding the High-Performance-Computing Community: A Software Engineer's Perspective. *IEEE Software* 25, 4 (2008), 29–36. https://doi.org/10.1109/MS.2008.103
[4] George E. P. Box. 1976. Science and Statistics. *J. Amer. Statist. Assoc.* 71, 356 (1976), 791–799. https://doi.org/10.1080/01621459.1976.10480949 arXiv:https://www.tandfonline.com/doi/pdf/10.1080/01621459.1976.10480949
[5] Lauren Cadwallader and Iain Hrynaszkiewicz. 2022. A survey of researchers' code sharing and code reuse practices, and assessment of interactive notebook prototype, Vol. 10. PeerJ. https://doi.org/10.7717/peerj.13933
[6] Diana-Mirela Cândea and Aurora Szentagotai-Tătar. 2018. Shame-proneness, guilt-proneness and anxiety symptoms: A meta-analysis. *Journal of anxiety disorders* 58 (2018), 78–106.
[7] Kathy Charmaz. 2014. *Constructing grounded theory.* Sage.

[8] David A. Cook and Anthony R. Artino. 2016. Motivation to learn: an overview of contemporary theories. *Medical Education* 50 (2016), 997 – 1014.

[9] Ian A. Cosden, Kenton McHenry, and Daniel S. Katz. 2022. Research Software Engineers: Career Entry Points and Training Gaps. *Computing in Science & Engineering* 24, 6 (2022), 14–21. https://doi.org/10.1109/MCSE.2023.3258630

[10] Anshu Dubey. 2022. Good Practices for High-Quality Scientific Computing. *Computing in Science & Engineering* 24, 6 (2022), 72–76. https://doi.org/10.1109/MCSE.2023.3259259

[11] Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Computing in Science and Engineering* 11, 6, 65–74. https://doi.org/10.1109/MCSE.2009.193

[12] European Organization For Nuclear Research and OpenAIRE. 2013. Zenodo. https://doi.org/10.25495/7GXK-RD71

[13] Jonathan L Freedman, Sue A Wallington, and Evelyn Bless. 1967. Compliance without pressure: The effect of guilt. *Journal of Personality and Social Psychology* 7, 2p1 (1967), 117.

[14] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. 2021. Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*. USENIX Association. https://www.usenix.org/conference/soups2021/presentation/fulton

[15] Carole Goble. 2014. Better Software, Better Research. *IEEE Internet Computing* 18, 5 (2014), 4–8. https://doi.org/10.1109/MIC.2014.88

[16] Jo Erskine Hannay, Carolyn MacLeod, Janice Singer, Hans Petter Langtangen, Dietmar Pfahl, and Greg Wilson. 2009. How do scientists develop and use scientific software?. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. 1–8. https://doi.org/10.1109/SECSE.2009.5069155

[17] Andrew Head, Fred Hohman, Titus Barik, Steven M Drucker, and Robert DeLine. 2019. Managing messes in computational notebooks. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*. 1–12.

[18] J. Hughes. 1989. Why Functional Programming Matters. *Comput. J.* 32, 2 (01 1989), 98–107. https://doi.org/10.1093/comjnl/32.2.98 arXiv:https://academic.oup.com/comjnl/article-pdf/32/2/98/1445644/320098.pdf

[19] Arne Johanson and Wilhelm Hasselbring. 2018. Software Engineering for Computational Science: Past, Present, Future. *Computing in Science and Engineering* 20, 2 (2018), 90–109. https://doi.org/10.1109/MCSE.2018.021651343

[20] Upulee Kanewala and James M. Bieman. 2014. Testing scientific software: A systematic literature review. *Information and Software Technology* 56, 10 (2014), 1219–1232. https://doi.org/10.1016/j.infsof.2014.05.006

[21] Louise H. Kellogg, Lorraine J. Hwang, Rene Gassmöller, Wolfgang Bangerth, and Timo Heister. 2019. The Role of Scientific Communities in Creating Reusable Software: Lessons From Geophysics. *Computing in Science & Engineering* 21, 2 (2019), 25–35. https://doi.org/10.1109/MCSE.2018.2883326

[22] Mary Beth Kery, Amber Horvath, and Brad Myers. 2017. Variolite: Supporting Exploratory Programming by Data Scientists. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 1265–1276. https://doi.org/10.1145/3025453.3025626

[23] Darren Key, Wen-Ding Li, and Kevin Ellis. 2022. I Speak, You Verify: Toward Trustworthy Neural Program Synthesis. arXiv:2210.00848 [cs.SE]

[24] Sarah Killcoyne and John Boyle. 2009. Managing chaos: lessons learned developing software in the life sciences. *Computing in science & engineering* 11, 6 (2009), 20–29.

[25] Amy J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-User Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (apr 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[26] Sandeep Kaur Kuttal, Bali Ong, Kate Kwasny, and Peter Robe. 2021. Trade-Offs for Substituting a Human with an Agent in a Pair Programming Context: The Good, the Bad, and the Ugly. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems* (Yokohama, Japan) *(CHI '21)*. Association for Computing Machinery, New York, NY, USA, Article 243, 20 pages. https://doi.org/10.1145/3411764.3445659

[27] Scott A. Lathrop, Katharine Cahill, Steven I. Gordon, Jennifer Houchins, Robert M. Panoff, and Aaron Weeden. 2020. Preparing a Computationally Literate Workforce. *Computing in Science & Engineering* 22, 4 (2020), 7–16. https://doi.org/10.1109/MCSE.2020.2994763

[28] StataCorp LLC. 2023. *Stata*. https://www.stata.com/

[29] Will W.K. Ma and Allan H.K. Yuen. 2011. Understanding online knowledge sharing: An interpersonal relationship perspective. *Computers & Education* (2011).

[30] David R. MacIver, Zac Hatfield-Dodds, and Many Other Contributors. 2019. Hypothesis: A new approach to property-based testing. *Journal of Open Source Software* 4, 43 (2019), 1891. https://doi.org/10.21105/joss.01891

[31] Lauren E. Margulieux, James Prather, Masoumeh Rahimi, and Gozde Cetin Uzun. 2023. Leverage Biology to Learn Rapidly From Mistakes Without Feeling Like a Failure. *Computing in Science & Engineering* 25, 2 (2023), 44–49. https://doi.org/10.1109/MCSE.2023.3297750

[32] Cory Merow, Josep M Serra-Diaz, Brian J Enquist, and Adam M Wilson. 2023. AI chatbots can boost scientific coding. *Nature Ecology & Evolution* (2023), 1–3.

[33] Richard E. Nance. 1993. A History of Discrete Event Simulation Programming Languages. *SIGPLAN Not.* 28, 3 (mar 1993), 149–175. https://doi.org/10.1145/155360.155368

[34] Luke Nguyen-Hoan, Shayne Flint, and Ramesh Sankaranarayana. 2010. A Survey of Scientific Software Development. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement* (Bolzano-Bozen, Italy) *(ESEM '10)*. Association for Computing Machinery, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1852786.1852802

[35] Elizaveta Pertseva, Melinda Chang, Ulia Zaman, and Michael Coblenz. 2024. A Theory of Scientific Programming Efficacy Artifact. https://doi.org/10.5281/zenodo.10462606

[36] Prakash Prabhu, Thomas B. Jablin, Arun Raman, Yun Zhang, Jialu Huang, Hanjun Kim, Nick P. Johnson, Feng Liu, Soumyadeep Ghosh, Stephen Beard, Taewook Oh, Matthew Zoufaly, David Walker, and David I. August. 2011. A Survey of the Practice of Computational Science. In *State of the Practice Reports* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 19, 12 pages. https://doi.org/10.1145/2063348.2063374

[37] Tom Prickett, Julie Walters, Longzhi Yang, Morgan Harvey, and Tom Crick. 2020. Resilience and Effective Learning in First-Year Undergraduate Computer Science. In *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education* (Trondheim, Norway) *(ITiCSE '20)*. Association for Computing Machinery, New York, NY, USA, 19–25. https://doi.org/10.1145/3341525.3387372

[38] Michael Resnick. 2020. *Designing for Wide Walls*. https://mres.medium.com/designing-for-wide-walls-323bdb4e7277

[39] Russ Rew and Glenn Davis. 1990. NetCDF: an interface for scientific data access. *IEEE computer graphics and applications* 10, 4 (1990), 76–82.

[40] Mark Sharratt and Abel Usoro. 2003. Understanding Knowledge-Sharing in Online Communities of Practice. *Electronic Journal of Knowledge Management* 1, 2 (2003), pp18–27.

[41] Tim Storer. 2017. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *ACM Comput. Surv.* 50, 4, Article 47 (aug 2017), 32 pages. https://doi.org/10.1145/3084225

[42] The Software Carpentry Foundation. 2023. *Software Carpentry: About Us*. https://software-carpentry.org/about/

[43] Huy Tu, Rishabh Agrawal, and Tim Menzies. 2020. The changing nature of computational science software. *arXiv preprint arXiv:2003.05922* (2020).

[44] Medha Umarji, Carolyn Seaman, A. Gunes Koru, and Hongfang Liu. 2009. Software Engineering Education for Bioinformatics. In *2009 22nd Conference on Software Engineering Education and Training*. 216–223. https://doi.org/10.1109/CSEET.2009.44

[45] Igor Wiese, Ivanilton Polato, and Gustavo Pinto. 2020. Naming the Pain in Developing Scientific Software. *IEEE Software* 37, 4 (2020), 75–82. https://doi.org/10.1109/MS.2019.2899838

[46] G. Wilson. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science & Engineering* 8, 6 (November–December 2006), 66–69. https://doi.org/10.1109/MCSE.2006.122 Summarizes the what and why of Version 3 of the course..

[47] Greg Wilson. 2014. Software Carpentry: lessons learned. *F1000Research* 3 (2014).

[48] Greg Wilson. 2023. Software Carpentry web site. http://software-carpentry.org. Main web site for Software Carpentry, replacing http://swc.scipy.org..

[49] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. 2014. Best practices for scientific computing. *PLoS biology* 12, 1 (2014), e1001745.

[50] Greg Wilson, Jennifer Bryan, Karen Cranston, Justin Kitzes, Lex Nederbragt, and Tracy K Teal. 2017. Good enough practices in scientific computing. *PLoS computational biology* 13, 6 (2017), e1005510.

[51] xarray Developers. 2023. *Xarray documentation*. https://docs.xarray.dev/en/stable/