Designing Tools to Enhance Best Practices in Research Software Engineering

Minhyuk Ko
Virginia Tech
Blacksburg, VA, USA
minhyukko@vt.edu

Chris Brown
Virginia Tech
Blacksburg, VA, USA
dcbrown@vt.edu

Abstract

Background: Research software engineers (RSEs) develop software to advance research across disciplines. However, RSEs rarely adopt the best software engineering practices—activities to enhance the development and maintenance of software. This can lead to negative consequences, such as increased development effort and inaccurate study results. **Method:** We conducted two participatory design workshops with RSEs to understand development challenges and explore possible design affordances to overcome those challenges. **Results:** Our findings reveal that RSEs struggle in various aspects of programming, such as debugging and understanding codebases, and face unique challenges, including selecting programming languages that best suit their needs and adopting developers' mental models. Our findings also reveal that RSEs desire novel tools that support research development tasks—prioritizing code translation, code understanding, and communication—leveraging the power of large language models (LLMs). **Conclusion:** Our paper offers valuable insights and future research directions for designing tools to assist RSEs in adopting beneficial software engineering practices.

CCS Concepts

• **Software and its engineering** → **Software prototyping**.

Keywords

Research Software Engineering, Participatory Design, Development Tools

ACM Reference Format:
Minhyuk Ko and Chris Brown. 2026. Designing Tools to Enhance Best Practices in Research Software Engineering. In *Cooperative and Human Aspects of Software Engineering (CHASE '26), April 13–14, 2026, Rio de Janeiro, Brazil.*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3794860.3794914

1 Introduction

Software engineering (SE) encompasses the processes, methods, and tools to support software development and maintenance, producing high-quality applications [41]. However, SE is complex as it spans tasks to design, implement, test, maintain, and innovate software programs [22]. Best practices informed by practitioner insights and empirical research improve development processes [18], yet research shows developers often avoid these practices (e.g., [28, 45])—leading to lower software quality [20].

Software is also used to advance discovery across science and engineering disciplines. Researchers use software to store and analyze large datasets, perform complex calculations, and develop models and simulations across domains for scientific investigation [47]. More than 90% of researchers rely on software for their work [25], with many stating that their research would be impractical or far more time-consuming without it. SE practices also benefit scientific software developed by researchers without computing backgrounds, or research software engineers (RSEs) [16]. In this paper, we adopt a broad definition of RSEs that spans the spectrum of full-time software engineers building scientific software to researchers programming code for research. [1]

The adoption of SE practices in research contexts is hindered by several factors, including a lack of understanding of SE principles [15], insufficient time and opportunities for skill development [14], and limited access to SE resources and research [20]. Because of these difficulties, scientists frequently spend too much time battling software problems instead of concentrating on their research goals [52]. Avoiding SE practices can also have serious repercussions, such as creating security flaws [38] or yielding inaccurate study findings [46]. Prior work explores designs for automated recommendation systems to promote better development practices [11, 13]. To provide initial insights on methods to promote best practices in research SE, we sought to answer the following research questions:

RQ1: What are the development challenges that RSEs face when developing research software?

RQ2: What design affordances are needed to overcome development challenges from the perspective of RSEs?

We conducted two participatory design (PD) workshops to identify challenges and co-design solutions to support developing and maintaining research software. This provided us the opportunity to engage with RSEs, who develop and maintain research software without prior knowledge or formal SE training, to investigate human-centric aspects of research software development within a specific community. The PD workshop approach helped foster creative thinking and deeper insights by enabling participants to critically engage with one another and refine their understanding of RSE challenges [35]. Our findings reveal RSEs face common development challenges implementing and debugging code, yet also face unique issues navigating different programming languages and adapting developers' mental models [34], such as typical development tools, activities, and practices. To overcome development challenges, participants proposed a variety of tools to increase awareness and adoption of SE practices. Based on our results, we

[1] https://us-rse.org/about/what-is-an-rse/

ACM ISBN 978-1-4503-XXXX-X/18/06.
https://doi.org/10.1145/3794860.3794914

motivate future work to innovate automated solutions for supporting the development and maintenance of research software by providing hands-on learning opportunities and enhancing the utilization of generative artificial intelligence (AI).

2 Related Work

Prior work investigates challenges developing and maintaining research software. Weise et al. [51] describe three main categories of challenges in the research software community: technical, social, and scientific. They point out that more than 70% of issues are technical. Several studies survey RSEs to investigate challenges they encounter when developing research software [14, 15]. Prior work also explores challenges RSEs face during particular stages of software development, including requirements [33], design [42], and testing [29, 30]. According to Segal [44], RSEs frequently use unsuitable development models, making it difficult to create requirements, test software, and write high-quality code. To better understand the challenges faced by RSEs and develop ways to address these issues, we build on these works by conducting PD workshops to understand challenges and design tools to promote beneficial behaviors and overcome challenges in research software development.

3 Method

3.1 PD Workshops

Participants. To elicit RSEs' insights on challenges and solutions for research software engineering tools, we recruited participants from diverse disciplines with at least one year of research-based programming experience and without formal training in Computer Science or SE. Our study participants represented a diverse sample of RSEs. Participants also had diverse occupations, including lab assistant, behavioral data scientist, and instructor. Participants reported having between 1 and 10 years of programming experience ($mdn = 4$) and were familiar with 1 to 6 different programming languages ($mdn = 3$). Participants had various methods of implementing software for their research, such as "conducting data analysis" (P16, P17), "building models" (P4), and "dataset processing" (P7). Further information regarding participants' demographics is available in the supplemental materials.[2]

Workshop Design. The PD workshop activities were based on the three-stage PD methodology [48]: *(1) Initial Exploration:* leveraging the 1-2-4-ALL liberating structure [36] to outline RSE challenges; *(2) Discovery Process:* consisting of collaborative brainstorming in groups of 3~4 to devise solutions; and *(3) Prototyping:* where groups used methods physical or virtual whiteboards to design brainstormed solutions. We iterated the discovery process and prototyping stages to facilitate an iterative co-exploration and prototype refinement. The in-person session (Groups 1-3, P1-11[3]) was held at the authors' primary institution, and the virtual session (Groups 4-6, P12-P20) was held over Zoom.[4] The first author moderated both sessions. After completing the workshop, participants were compensated \$100 USD. The study was approved by Virginia Tech IRB 24-539.

[2]https://figshare.com/s/8efb30318ca23fa79b9c
[3]The P- prefix indicates a workshop participant.
[4]https://www.zoom.com/

3.2 Data Analysis

We recorded the ALL session of the 1-2-4-ALL activity and the demo presentations by using the built-in Zoom recording services. The meeting room for the in-person session had technology embedded to accommodate Zoom meetings and projection, such as cameras and microphones installed throughout the ceiling of the room. We transcribed the recordings by using the default Zoom transcription service. To ensure accuracy of the transcription, the first author manually reviewed the transcription and corrected any mistakes. We also collected the notes participants took throughout the session, prototype demo slides, and took photographs of the devised sketches and prototypes. After reviewing the collected artifacts and recordings, we conducted thematic analysis [10] to identify recurring themes throughout the sessions.

To analyze the transcripts, we leveraged the intentional AI coding feature of the Atlas.ti,[5] a computer-assisted qualitative data analysis software. Research suggests AI tools demonstrate comparable performance to humans in qualitative annotation tasks [2], and Atlas.ti has been used in prior work to support data coding and analysis [3]. We provided our RQs as intentions and the recurring themes that we identified as codes to achieve higher accuracy. The coding was performed on the ALL discussion (RQ1) and final demo presentation (RQ2) recordings. After Atlas.ti analyzed the data, the first author went through the codes to review the generated output and correct mistakes caused by the tool. We report the groundedness, the frequencies of codes detected [21], in the results using the multiplication symbol (×). We do not report IRR because IRR is a statistical measure used to determine the degree of agreement among multiple human annotators, not AI tools [23]. To facilitate replication of this study, we include the codebook in our supplemental materials.[2]

4 Results

4.1 Development Challenges

Implementation (51×) RSEs noted difficulties in thinking like a programmer, struggling to code programs into fruition due to a lack of skills and mental models to implement the software. Even when they were able to implement the code and make the code run, they faced difficulties knowing if the code was running as intended.

❝ We were not able to interpret what we have in our minds, about the coding or whatever the structure that we are thinking to be interpreted directly into the [code]. ❞ (P12)

Debugging (25×) RSEs expressed challenges with debugging, or finding and fixing coding errors, which is one of the most expensive and time-consuming processes in software development [31]. Participants expressed frustration in debugging code written by themselves, other developers or AI, noting difficulties understanding the logic behind the code.

❝ Specifically, syntax, semantics, logic errors, and formulation errors. Oh, so different types of errors that have different strategies to solve those. ❞ (P8)

Understanding (20×) Understanding the codebase that programmers are working on is essential to perform various SE tasks,

[5]https://atlasti.com/

such as testing or debugging [4]. However, RSEs expressed difficulty in understanding the code written by other programmers or AI. Prior work also suggests that few scientists believe they have sufficient understanding of their code [24].

❝ [We] often read other people's code and then they write our own code, right? But if you don't understand other people's code, it's hard. And now, even if ChatGPT that generates code. And if you don't understand what ChatGPT's code is, [you can't confidently modify, debug, or trust it to work as intended]. ❞ (P5)

Syntax (19×) Participants expressed frustration in knowing minor syntax differences across various programming languages. For example, a print statement would be `print()` in Python and `printf()` in C. These minor errors can cause errors at compile time. Thus, users would not be able to receive immediate feedback on syntax errors.

❝ [It's difficult to know] the different syntax between different programming languages or different softwares that we use. ❞ (P5)

Selection (16×) As there are hundreds of programming languages available, RSEs reported difficulties selecting the best languages and tools to suit their needs. Oftentimes, their limited exposure to programming languages restricted their choices. Participants desired the ability to describe programs in natural language or pseudocode, then have AI select the programming language and implement the code.

❝ [I would like a tool where I can] get a list of things that I can do into that particular platform or related to that platform. ❞ (P12)

Verification (12×) Participants expressed frustration with verifying code, and desired mechanisms to assess implemented code works as intended. This can be achieved through various testing methods, such as unit testing, but prior work suggests RSEs lack expertise to build unit tests for software they are working on [19].

❝ I might get an output that works and that it is running, but knowing if it's running necessarily properly like for my application or for my use [is different]. ❞ (P19)

➔ **Key Findings:** RSEs often struggle with implementing, debugging, and understanding programs. They also lack developer mental models, limiting their ability to make informed decisions for suitable programming languages, tools, and testing approaches to support their development.

4.2 Design Affordances to Overcome Challenges

Code Translation (122×) Software engineers often work with multiple programming languages to ensure performance, maintainability, and compatibility [32]. To satisfy the varying requirements of research software, supporting code translation is becoming more and more necessary [9]. Yet, programming languages tend to differ widely in terms of their respective programming idioms, syntax, and structures [5]. To address this, participants desired characteristics of tools to support code translation—where users could input code snippets and have them automatically and accurately translated across different programming languages.

❝ [We desire a tool where] each of the same code are actually translated into a desired language such as like Python, C++, Java. ❞ (P13)

Code Comprehension (74×) To assist with understanding the functionality of code, RSEs suggested various features to promote code understanding, such as generating code explanations using AI and translating code to pseudocode or simplified descriptions of algorithms without strict notations.

❝ So it will be converting everything into a plain English converter. So the second stage is the conversion stage in which the explained code. So in the final stage, that is the output stage in which the front-end displays the step-by-step breakdown of the code. ❞ (P13)

Auto-Correction (43×) Automated program repair is an active research area in SE [39], offering tools and techniques to automatically fix errors in software. Participants desired functionality where simple errors, such as missing semicolons in Java, are automatically fixed. A proposed design was a live-editing feature that could quickly fix common mistakes while users are programming, increasing coding efficiency. This feature could also detect non-logical issues, such as coding and commenting style, and fix it to increase the readability of the code.

❝ So, like MS Word, my metacode should also autocorrect the syntaxes, at least the syntax, it should autocorrect. ❞ (P1)

Communication (36×) Communication is a critical part of software development [37]. Participants reported collaborating with researchers to leverage expertise from other disciplines. However, as different researchers have different specializations, they are often proficient in different programming languages. This raises communication challenges in coding as they experience a hard time understanding the code that the other researcher has written. To mitigate communication issues, participants proposed using standardized comments, integrating group chat, and efficiently utilizing version control systems.

❝ [We could] produce a more standardized description of every user-defined function based on a contextual understanding that could help a person who's never seen the code and who just downloaded the code. ❞ (P13)

AI for Development (35×) Generative AI is increasingly used to support software development [43]. As the field of AI is rapidly developing, software engineers use various LLMs, like ChatGPT and GitHub CoPilot to assist with writing code [27, 40]. Workshop participants also desired LLM assistance in research development tasks, such as correcting mistakes and debugging. RSEs also suggested enhancing access to LLMs by embedding them into programming environments to avoid switching between apps.

❝ We basically wanted to have one app which has all the programmings skills and also ChatGPT inside so that we don't have to go back and forth. ❞ (P5)

Help Using AI (32×) Participants also desired tooling to help them use AI more efficiently. For example, the output of LLMs heavily depends on the prompt users provide [53]—which also impacts LLMs in SE contexts [50]. Participants desired tools to have characteristics that would assist RSEs in devising better prompts for LLMs to generate more helpful and accurate responses. Another example proposed by RSEs involves functionality to help them choose the appropriate LLM models or fine-tuning methods for customized results.

“ So I could select those pieces of code and I would expect a prompt to show up. It's pretty much inspired by the idea like when you hover your cursor over, let's say an icon. ” (P13)

File Type (12×) RSEs often work with different types of files. For instance, research shows data provenance and management is a challenge for scientific software, as data is often stored in varied and specialized file types [17]. PD workshop participants proposed a dynamic programming tool where they can import different file types to integrate in programs. For example, while a data mining algorithm written in Python will be stored in `.py` extension, the dataset would be stored in `.xlsx` extension. Participants also desired a tool to automatically integrate code in different programming languages to run as if they were a single file.

“ And the second thing is like in research, for example, when I look for certain functions, maybe somebody's implemented a piece of it in MATLAB or maybe Python. ” (P13)

Best Practices (10×) RSEs are often unaware of SE practices [15], leading to less efficient programming [6]. To overcome this, RSEs suggested a tool to guide them to adopt SE best practices. Examples of practices mentioned by participants include implementing reusable code, communicating with teammates to coordinate tasks, and thoroughly testing code. This can be achieved through an AI-powered bot which suggests best practices to RSEs depending on the SE task they are performing—proposed by Group 4. For example, when RSEs design a code where cyclomatic complexity exceeds 10, the bot could flag the code and suggest breaking it down into smaller functions.

“ For example, um it does take a few commented options, but that's completely up to the user and not every single person has the same commenting style. ” (P13)

Education (10×) Prior work suggests RSEs lack time to educate themselves about current practices and tools to support software development [14]. Although AI is capable of generating commonly implemented functions, sometimes it is better for programmers to implement the function so that they can gain a deeper understanding of the underlying logic and algorithms. For example, in mathematics, they would be able to reason the mathematical concept and proof by going through this process [8]. Participants noted it would be helpful to have an educational website where they can learn about various aspects of SE, such as how to code solutions or use AI to generate code, depending on users' educational needs.

“ Theorem, lemma, and proof, theorem, lemma, and proof, and stuff like that so just being able to do something like this like hand-on would be better for understanding than letting like AI do it for yourself. It's kind of like not about the result. It's about the process. ” (P20)

Diversity (7×) To support the adoption of SE practices for researchers from diverse backgrounds, participants suggested the need for inclusive tool designs. For example, one prototyped solution proposed incorporating customized interfaces and recommendations for SE practices based on users' identities and field.

“ We talked about customization related to like gender and also having like a neurodivergent mode so like for people with ADHD and autism, like how to coding like more accessible. ” (P17)

Key Findings: Participants desired interactive tooling to support translating, understanding, correcting, and communicating about code. They also viewed AI as beneficial, proposing solutions to maximize AI usage for enhancing development and SE practices.

5 Discussion and Future Work

5.1 RSEs as Software Engineers

Several themes we identified overlapped with the broader SE community. For instance, we observed RSEs desire affordances such as customizability, workflow integration, and AI-powered development tools—aligning with preferences for traditional software engineers who develop software for non-scientific commercial or consumer needs [12]. However, we also observed differences. For example, while both groups desired tools powered by AI, workshop participants were primarily interested in AI tools for code translation and understanding. On the other hand, traditional software engineers are mostly interested in AI for testing and reviewing code [1]. Moreover, we discovered RSEs found value in using tools to for ***hands-on learning opportunities***; while prior work suggests software engineers are disinterested in learning about new development tools and concepts [11]. This motivates the need for tools to increase awareness and hands-on exploration of SE tools and practices in research settings. For example, Group 4 proposed a tool that interrupts programmers in IDEs during SE activities to recommend effective practices. Recent work investigates interactive explanations to help novice programmers understand SQL queries via visualization and step-by-step explanation [49]. Future efforts can extend this approach to support learning development concepts in research SE contexts.

5.2 Utilizing Generative AI

RSEs are increasingly utilizing generative AI for programming tasks [40]. Across our PD workshops, participants expressed a desire to utilize generative AI for various development tasks. Prior work suggests research advancement and establishing scientific foundations are top priorities for AI [26], thus future work can explore leveraging AI to support research SE. We observed two opportunities. First, participants desired functionality to ***integrate research contexts in LLM reasoning*** for SE tasks—providing specific output based on users' background and expertise. For instance, Group 6 proposed a tool to convert math equations into optimized code snippets. Further, participants desired support for ***providing contexts to models through prompting***. For example, Group 2 designed a prototype that assists RSEs in writing optimal prompts. Tools such as ChainBuddy [54] help users devise prompts in LLM-based workflows. Future work can explore similar approaches to enhance prompts based on SE guidelines [7], user domain, and programming goals.

6 Conclusion

Software is increasingly critical for scientific research and innovation. We conducted two PD workshops to understand challenges and design affordances to support research SE. Our findings reveal RSEs are not familiar with state-of-the-art programming tools, and would like to see tools that assist in supporting various SE tasks and promoting the adoption of SE practices. Based on our findings, we provide implications and future directions to mitigate challenges in research software development.

References

[1] 2024. *Stack Overflow Developer Survey 2024*. Technical Report. Stack Overflow. https://survey.stackoverflow.co/2024/ai#developer-tools-ai-tool-interested

[2] Toufique Ahmed, Premkumar Devanbu, et al. 2024. Can LLMs replace manual annotation of software engineering artifacts? *arXiv preprint arXiv:2408.05534* (2024).

[3] Gökçe Akçayır and Murat Akçayır. 2018. The flipped classroom: A review of its advantages and challenges. *Computers & Education* 126 (2018), 334–345.

[4] Nedhal A Al-Saiyd. 2017. Source code comprehension analysis in software maintenance. In *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*. IEEE, 1–5.

[5] Miltiadis Allamanis and Charles Sutton. 2014. Mining idioms from source code. In *International symposium on foundations of software engineering (FSE)*. 472–483.

[6] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, et al. 2021. Software engineering practices for scientific software development: A systematic mapping study. *Journal of Systems and Software* 172 (2021), 110848.

[7] Simon Arvidsson and Johan Axell. 2023. Prompt engineering guidelines for LLMs in Requirements Engineering. (2023).

[8] Laura Benton, Piers Saunders, Ivan Kalas, Celia Hoyles, and Richard Noss. 2018. Designing for learning mathematics through programming: A case study of pupils engaging with place value. *International journal of child-computer interaction* 16 (2018), 68–76.

[9] Ainhoa Berciano, Astrid Cuida, et al. 2024. The importance of coding and translation between programming languages in sequential activities of pre-service teachers: an approach. *Education and Information Technologies* (2024), 1–25.

[10] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.

[11] Chris Brown and Chris Parnin. 2019. Sorry to bother you: Designing bots for effective recommendations. In *International Workshop on Bots in Software Engineering (BotSE)*. IEEE, 54–58.

[12] Chris Brown and Chris Parnin. 2020. Comparing different developer behavior recommendation styles. In *Cooperative and Human Aspects of Software Engineering (CHASE)*. 78–85.

[13] Chris Brown and Chris Parnin. 2020. Sorry to bother you again: Developer recommendation choice architectures for designing effective bots. In *International Workshop on Bots in Software Engineering (BotSE)*. 56–60.

[14] J Carver. 2019. URSSI conceptualization survey results.

[15] Jeffrey Carver, Dustin Heaton, Lorin Hochstein, and Roscoe Bartlett. 2013. Self-perceptions about software engineering: A survey of scientists and engineers. *Computing in Science & Engineering* 15, 1 (2013), 7–11.

[16] Ian A Cosden, Kenton McHenry, and Daniel S Katz. 2023. Research software engineers: Career entry points and training gaps. *Computing in Science & Engineering* 24, 6 (2023), 14–21.

[17] Susan B Davidson and Juliana Freire. 2008. Provenance and scientific workflows: challenges and opportunities. In *International conference on Management of data*. 1345–1350.

[18] Prem Devanbu, Thomas Zimmermann, and Christian Bird. 2016. Belief & evidence in empirical software engineering. In *International conference on software engineering (ICSE)*. 108–119.

[19] Nasir U Eisty, Upulee Kanewala, and Jeffrey C Carver. 2025. Testing research software: an in-depth survey of practices, methods, and tools. *Empirical Software Engineering* 30, 3 (2025), 81.

[20] Robert L Glass. 1994. The software-research crisis. *IEEE Software* 11, 6 (1994), 42–47.

[21] ATLAS.ti Scientific Software Development GmbH. 2024. *ATLAS.ti Mac User Manual*. https://doc.atlasti.com/ManualMac/Codes/CodingDataBasicConcepts.html Accessed: 2024-11-11.

[22] Gagan Gurung, Rahul Shah, and Dhiraj P Jaiswal. 2020. Software development life cycle models-A comparative study. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology* 6, 4 (2020), 30–37.

[23] Kilem L Gwet. 2014. *Handbook of inter-rater reliability: The definitive guide to measuring the extent of agreement among raters.* Advanced Analytics, LLC.

[24] Jo Erskine Hannay, Carolyn MacLeod, et al. 2009. How do scientists develop and use scientific software?. In *ICSE workshop on software engineering for computational science and engineering*. Ieee, 1–8.

[25] Simon Hettrick. 2018. softwaresaved/software_in_research_survey_2014: Software in research survey. https://zenodo.org/record/1183562

[26] Eric Horvitz, Vincent Conitzer, et al. [n. d.]. Now, Later, and Lasting: 10 Priorities for AI Research, Policy, and Practice. *Commun. ACM* ([n. d.]).

[27] Juyong Jiang, Fan Wang, et al. 2024. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515* (2024).

[28] Brittany Johnson, Yoonki Song, et al. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.

[29] Upulee Kanewala and James M Bieman. 2014. Testing scientific software: A systematic literature review. *Information and software technology* 56, 10 (2014), 1219–1232.

[30] Diane Kelly and Rebecca Sanders. 2008. The challenge of testing scientific software. In *Conference of the Association for Software Testing (CAST)*. Citeseer, 30–36.

[31] Minhyuk Ko, Dibyendu Brinto Bose, et al. 2023. Exploring the Barriers and Factors that Influence Debugger Usage for Students. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 168–172.

[32] Pavneet Singh Kochhar, Dinusha Wijedasa, and David Lo. 2016. A large scale study of multiple programming languages and code quality. In *International conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 563–573.

[33] Konstantin Kreyman and David Lorge Parnas. 2002. On documenting the requirements for computer programs based on models of physical phenomena. *SQRL Report* 1 (2002).

[34] Thomas D LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining mental models: a study of developer work habits. In *International conference on Software engineering (ICSE)*. 492–501.

[35] Rachael Luck. 2018. What is it that makes participation in design participatory design? *Design studies* 59 (2018), 1–8.

[36] K McCandless and H Lipmanowicz. [n. d.]. Liberating structures: 1–2–4–all.

[37] Ian R McChesney and Seamus Gallagher. 2004. Communication and co-ordination practices in software engineering projects. *Information and Software Technology* 46, 7 (2004), 473–489.

[38] Reed Milewicz, Jeffrey Carver, et al. 2022. A Secure Future for Open-Source Computational Science and Engineering. *Computing in Science & Engineering* 24, 4 (2022), 65–69.

[39] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[40] Alan Peslak and Lisa Kovalchick. 2024. AI for coders: An analysis of the usage of ChatGPT and GitHub CoPilot. *Issues in Information Systems* 25, 4 (2024), 252–260.

[41] Roger S Pressman. 2005. Software Engineering: a practitioner's approach. *Pressman and Associates* (2005).

[42] Francisco Queiroz and Rejane Spitz. 2016. The lens of the lab: Design challenges in scientific software. *International Journal of Design Management and Professional Practice* 10, 3 (2016), 17–45.

[43] Daniel Russo. 2024. Navigating the complexity of generative ai adoption in software engineering. *ACM Transactions on Software Engineering and Methodology* 33, 5 (2024), 1–50.

[44] Judith Segal. 2009. Some challenges facing software engineers developing software for scientists. In *ICSE workshop on software engineering for computational science and engineering*. IEEE, 9–14.

[45] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. 2020. Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security. In *Symposium on Usable Privacy and Security (SOUPS 2020)*. 221–238.

[46] David AW Soergel. 2015. Rampant software errors may undermine scientific results. *F1000Research* 3 (2015), 303.

[47] Damien Soulé. 2025. Programming Languages in Scientific Research. https://www.linkedin.com/pulse/programming-languages-scientific-research-damien-soul%C3%A9/ Last accessed: January 18, 2025.

[48] Clay Spinuzzi. 2005. The methodology of participatory design. *Technical communication* 52, 2 (2005), 163–174.

[49] Yuan Tian, Jonathan K. Kummerfeld, et al. 2024. SQLucid: Grounding Natural Language Database Queries with Interactive Explanations. In *Symposium on User Interface Software and Technology (UIST)* (Pittsburgh, PA, USA) *(UIST '24)*. Association for Computing Machinery, New York, NY, USA, Article 12, 20 pages. https://doi.org/10.1145/3654777.3676368

[50] Jules White, Sam Hays, et al. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative ai for effective software development*. Springer, 71–108.

[51] Igor Wiese, Ivanilton Polato, and Gustavo Pinto. 2019. Naming the pain in developing scientific software. *IEEE Software* 37, 4 (2019), 75–82.

[52] Greg Wilson. 2006. Software carpentry: getting scientists to write better code by making them more productive. *Computing in science & eng.* 8, 6 (2006), 66–69.

[53] J Diego Zamfirescu-Pereira, Richmond Y Wong, et al. 2023. Why Johnny can't prompt: how non-AI experts try (and fail) to design LLM prompts. In *CHI conference on human factors in computing systems*. 1–21.

[54] Jingyue Zhang and Ian Arawjo. 2024. ChainBuddy: An AI-assisted Agent System for Helping Users Set up LLM Pipelines. In *Symposium on User Interface Software and Technology (UIST)*. Association for Computing Machinery, New York, NY, USA, Article 48, 3 pages. https://doi.org/10.1145/3672539.3686763