When Tools Overlook Domain Knowledge: An Empirical Study of Refactoring in Scientific Software

ROHITH PUDARI, University of Toronto, Canada
AHMED MUSA AWON, University of Victoria, Canada
NEIL ERNST, University of Victoria, Canada
SHURUI ZHOU, University of Toronto, Canada

Refactoring is a critical process for improving code quality, but anecdotal evidence has shown that refactoring in scientific software (*Sci-SW*) is not always feasible. The inherently exploratory nature of *Sci-SW* development, characterized by evolving requirements and limited adoption of traditional software engineering practices, could present significant challenges to refactoring. However, there is no systematic study exploring refactoring practices in *Sci-OSS*. To bridge this gap, we explore the effectiveness of three state-of-the-art refactoring detection tools: RefDiff (C), RefactoringMiner (Java), and PyRef (Python) to detect refactorings in scientific open-source software (*Sci-OSS*). Our findings reveal that these tools have significant limitations, detecting fewer refactorings in *Sci-OSS* than non-scientific OSS (*Non-Sci-OSS*). Through a mixed-method approach, we identified that 67.54% of undetected refactorings in *Sci-OSS* require domain knowledge. To complement our analysis of the refactoring changes, we conducted surveys with 47 practitioners experienced in refactoring *Sci-OSS* and 14 follow-up interviews to gain deeper insights into the associated challenges. Our results revealed seven novel challenges for *Sci-OSS*-refactoring, including a domain knowledge gap. These findings emphasize the necessity for specialized tools and strategies to support refactoring in *Sci-OSS* effectively.

CCS Concepts: • **General and reference** → **Empirical studies**; **Surveys and overviews**; • **Software and its engineering** → **Maintaining software**; **Software evolution**; **Empirical software validation**.

Additional Key Words and Phrases: Refactoring detection, Research software engineering, Tool limitations, Domain-specific refactoring.

1 Introduction

Software refactoring is defined as code changes that perform improvements to the internal structure, design, or implementation, including readability, maintainability, and performance without any change to the external behavior [37, 50, 59, 68]. Prior works have demonstrated that refactoring significantly improves software quality and developer productivity by facilitating the maintenance and comprehension of software systems [37, 59, 68]. Ongoing refactoring is essential to minimize technical debt, which arises from expedient development decisions that necessitate future rework [25, 33, 61]. When neglected, technical debt accumulates, increasing maintenance costs and compromising software quality [33, 85, 103].

Scientific software (*Sci-SW*) is developed to meet domain-specific computational needs using specialized algorithms, data structures, and interfaces [57, 99]. Unlike non-scientific software (*Non-Sci-SW*), which is often developed for general-purpose computing, enterprise applications, or consumer-facing products, *Sci-SW* is primarily created by scientists

Authors' Contact Information: Rohith Pudari, r.pudari@mail.utoronto.ca, University of Toronto, Toronto, Ontario, Canada; Ahmed Musa Awon, its.ahmed.musa@gmail.com, University of Victoria, Victoria, British Columbia, Canada; Neil Ernst, nernst@uvic.ca, University of Victoria, Victoria, British Columbia, Canada; Shurui Zhou, shurui.zhou@utoronto.ca, University of Toronto, Toronto, Ontario, Canada.

Manuscript submitted to ACM

with limited software engineering (SE) training [83], leading to differences in development practices, collaboration, and sustainability [117]. This interdisciplinary process integrates domain knowledge with SE practices [83]. However, scientists, who are often the primary developers, spend over 40% of their time on software yet rarely adopt SE best practices [46, 83] due to limited awareness [17], the exploratory nature of research, and the need for rapid prototyping in scientific simulations [30, 44, 58, 102]. Furthermore, the substantial time investment required for refactoring, coupled with the need for extensive discussions and consensus-building in interdisciplinary collaborations, impedes the adoption of these practices [83].

One might assume that increasing the awareness of SE best practices among scientists developing and maintaining *Sci-SW* could address the challenges. However, prior studies suggest that adopting SE best practices in *Sci-SW* is not always feasible [12, 30, 58, 101]. Researchers found that testing is particularly challenging in *Sci-SW* due to the complexity of numerical simulations and hardware dependencies [12]. Since testing underpins many SE practices, including refactoring, the absence of unit tests poses a fundamental challenge to the feasibility of refactoring in *Sci-SW*. Similarly, in climate modeling, refactoring is nearly impossible due to the bit-wise comparisons, which require that code changes do not alter the bit values of any variables [30]. Despite these challenges, little is known about how *Sci-SW* developers navigate refactoring constraints or, more broadly, how they incorporate SE best practices in complex, cross-domain codebases.

Domain- or framework-specific refactorings can arise in many types of software systems. For example, software built on frameworks such as Spring [113], JUnit [106], or React [112] may involve refactoring patterns specific to those ecosystems. Our focus on *Sci-SW* therefore stems not from the exclusivity of such refactorings, but from the unique development conditions in this domain, which may influence how refactoring activities occur and how well existing refactoring detection tools perform. As a starting point, this study investigates refactoring practices in *Sci-SW* development, comparing them with the extensively studied *Non-Sci-SW*. Specifically, we focus on small-scale, opportunistic refactoring (also known as floss refactoring) [71], where developers incrementally improve code as part of their routine workflow. We analyze open-source scientific software (referred to as *Sci-OSS*) and general-purpose open-source software (denoted as *Non-Sci-OSS*) as representative cases. While our focus is on the scientific domain rather than the open-source development model, we leverage the abundance of open-source repositories for data accessibility.

The study objective is to evaluate the applicability of existing SE refactoring tools and understand scientists' perceptions, practices, and challenges. Because most empirical studies of refactoring rely on automated detection tools, understanding whether these tools can reliably identify refactorings in scientific software is a necessary first step toward studying refactoring practices in this domain. We therefore first examine the effectiveness of these tools (RQ1), and use the observed limitations to motivate a deeper investigation into developers' perceptions and challenges (RQ2). In addition, this research contributes to the SE research community by elucidating how refactoring adapts to the unique constraints of *Sci-SW*, highlighting the challenges of maintaining and improving such systems. To achieve our research goal, we formulate two research questions (RQs):

RQ1: How effectively do state-of-the-art (SOTA) refactoring detection tools identify refactorings in Sci-OSS? Accurate refactoring detection is crucial for generating reliable datasets, which serve as the foundation for understanding existing refactoring practices and developing refactoring recommendation tools, such as name recommendation for renaming refactorings [29] and identification of refactoring opportunities [78]. This study assesses the effectiveness of three SOTA tools (i.e., *RefDiff2.0* [97] for C, *RefactoringMiner3.0* [3] for Java, and *PyRef* [10] for Python) in detecting refactorings within *Sci-OSS*, as an exploratory step to uncover the boundaries of current automated detection.

Result: Our analysis revealed significant differences in detection effectiveness in these tools when applied to *Sci-OSS*, raising concerns about their reliability for constructing comprehensive datasets and supporting refactoring

recommendation systems. Because these detection tools rely heavily on Abstract Syntax Trees (ASTs) to identify structural patterns, they are inherently not designed to capture the domain-specificity of scientific code. Additionally, our qualitative analysis of false negative (FN) cases uncovered 12 unsupported types of refactorings. Moreover, our findings indicate that a substantial number of refactorings in *Sci-OSS* necessitate domain knowledge for successful completion, highlighting the unique refactoring needs and practices in *Sci-OSS*.

RQ2: How do developers perceive refactoring in *Sci-OSS*, and what challenges do they encounter in this process? Motivated by RQ1's findings that standard detection tools lack the domain-specificity required for scientific code, we conducted a qualitative deep dive to understand how developers navigate these unaddressed challenges. We conducted a survey with 47 practitioners and subsequently interviewed 14 of them, all of whom have been actively involved in refactoring *Sci-OSS* repositories. The purpose was to understand their perspectives and experiences, explore the unique challenges in refactoring *Sci-OSS*, and compare these practices with those in *Non-Sci-OSS*.

Result: Our results showed that while *Sci-OSS* and *Non-Sci-OSS* developers share similar definitions and motivations for refactoring, as reported by prior studies [59, 125], *Sci-OSS* developers emphasized the need for tools that incorporate domain knowledge to support refactoring along with features like scoping for finding all references and improved testing capabilities. Additionally, *Sci-OSS* developers reported additional challenges (e.g., added complexity and the absence of measurable outcomes for refactoring efforts).

In summary, we make the following contributions: (1) the first comprehensive dataset of refactoring practices in *Sci-OSS*, capturing domain-specific changes; (2) an empirical evaluation of SOTA refactoring detection tools on *Sci-OSS*, revealing significant limitations of current AST-based approaches in capturing refactorings that depend on domain knowledge; (3) the identification of unsupported refactoring types with evidence that the majority of *Sci-OSS* refactorings require domain knowledge; and (4) a survey of 47 practitioners and 14 follow-up interviews, providing deeper insights into *Sci-OSS* developer perceptions and informing the development of more effective refactoring support tools.

2 Motivating Example

An illustrative example of a refactoring requiring domain knowledge is PR#2542 in PlasmaPy/PlasmaPy, a widely used open-source python library for plasma science [22]. The refactoring addressed issue#1983, which identified that `plasmapy.formulary.lengths.gyroradius`, a function that computes the radius of the circular orbit of a charged particle moving through a magnetic field, had become difficult to read and modify due to its high cognitive complexity.

As shown in Fig 1, the original implementation included a suppressed linter annotation (`# noqa: C901`)[1], and the issue#1983 discussion referenced SonarSource's Cognitive Complexity metric [16] as motivation for simplifying the implementation. Among its inputs, the `gyroradius` function allows users to specify the particle's perpendicular velocity either directly through `Vperp` (the component of velocity perpendicular to the magnetic field) or indirectly through `T`, the particle temperature, from which the thermal speed is derived. The function also accepts `lorentzfactor`, a relativistic correction factor that can be inferred from `Vperp` or `T`, but may also be supplied explicitly to improve numerical precision near the speed of light if `Vperp` or `T` is not provided. To support these different usage scenarios, the original implementation represented unspecified values using sentinel values (primarily `NaN`) while enforcing domain constraints such as allowing `Vperp` or `T`, but not both. Since the function needed to support scalar and array forms of these inputs in different combinations, it had to handle numerous combinations of valid and missing values. This design resulted in six private helper functions and more than 130 lines of nested control flow (dashed red box in Fig 1).

[1]The `# noqa: C901` annotation suppresses Flake8's "function is too complex" warning (https://www.flake8rules.com/rules/C901.html). The annotation appears in the original source code [link].

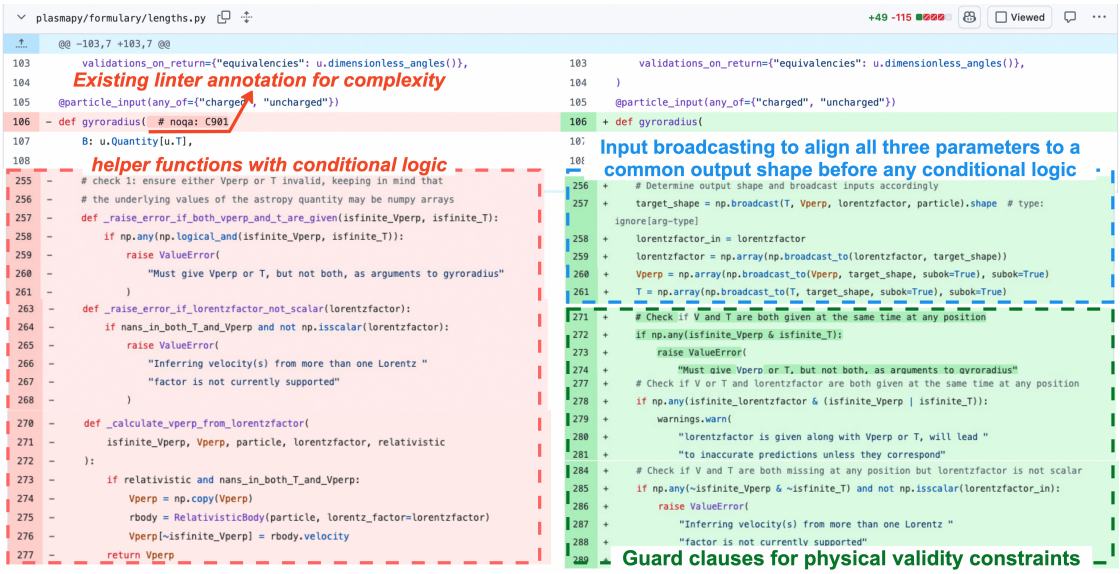


Fig. 1. Motivating example from *PlasmaPy/PlasmaPy PR#2542*. Left Side (Before) and Right Side (After). Nested control flow (dashed red box) is replaced by input broadcasting (dashed blue box) and flat guard clauses (dashed green box).

The refactoring introduced an *input broadcasting* step (dashed blue box in Fig 1), using `np.broadcast` to align all three parameters to a common shape before any conditional logic. During code review, the developers also discussed the need to copy broadcasted arrays as `np.broadcast_to` returns read-only views. Aligning the shapes allowed the six helper functions to collapse into a flat sequence of guard clauses mapping directly to validity constraints (dashed green box in Fig 1). As a result, the function shrank from over 130 to under 70 lines, and the `# noqa: C901` linter suppression was removed.

This code simplification is complex because the redundancy in the original code is not structurally visible as all six helper functions are called, all branches are syntactically reachable, and no code is statically dead. The redundancy only becomes apparent by understanding the domain relationships among `Vperp`, `T`, and `lorentzfactor`, including valid physical combinations and missing-value interpretations. Similarly, the refactored guard clauses encode constraints on how physical quantities in different regimes may or may not be combined, and understanding whether those constraints are correctly expressed requires domain knowledge of plasma physics. This code simplification refactoring involving domain knowledge is characteristic of what we refer to as *domain-driven code simplification* (discussed in Section 4.6.3).

Unlike conventional code simplification, where redundancy is identified through control-flow or data-flow analysis, the simplification in this example arises from scientific knowledge about the physical relationships among `Vperp`, `T`, and `lorentzfactor`. Without this knowledge, a developer cannot determine which branches to unify or which input combinations must remain distinct. This example illustrates why we think refactoring in *Sci-OSS* is worth special consideration:

- Scientific domain knowledge is often required throughout the refactoring process.
- Refactoring opportunities may not be evident from the code structure alone.
- Conventional software properties, such as reachability and syntactic equivalence are insufficient to assess refactoring correctness.

Our qualitative analysis in RQ1, along with the findings from our surveys and interviews in RQ2, also highlights that this is a significant challenge for effective refactoring in *Sci-OSS*.

3 Related Work

3.1 Refactoring Practices for Non-scientific Software (*Non-Sci-SW*)

Prior work on refactoring in *Non-Sci-OSS* can be broadly categorized into (1) quantitative studies that rely on refactoring detection tools to analyze large-scale code changes, and (2) qualitative studies that investigate developers' perceptions and practices through surveys and interviews.

3.1.1 Quantitative Studies: Previous research has explored various aspects of refactoring activities, including their definitions, motivations, practices, and the tools that support them. Refactoring detection tools have been used to empirically study how and when developers refactor. For example, Silva et al. mined 748 *Non-Sci-OSS* repositories for refactorings to analyze the motivations behind refactoring based on the explanations of developers on refactorings they have applied resulting in a taxonomy of motivations tied to refactoring types [98]. Expanding on the behavioral patterns of refactoring, Nagy et al. analyzed co-occurring refactorings in test and production code at commit level and found that nearly one in five refactoring commits modifies both test and source code [74]. Focusing on the collaborative development settings, Coelho et al. studied refactoring-inducing pull requests and showed that around 30 percent of merged pull requests include refactorings introduced after the initial submission, often in response to review comments, and that these pull requests tend to be larger, more discussed, and slower to merge [21]. Beyond understanding when and why refactoring occurs, other studies have focused on tool support. For example, Coelho et al. conducted a systematic literature mapping study of refactoring tools to support modern code review raising the need for more tools to explain composite refactorings [20]. Similarly, Pantiuchina et al. performed a large-scale study of *Non-Sci-OSS* repos, producing a taxonomy of 67 motivations that drive developers to perform refactoring operations [80].

3.1.2 Qualitative Studies: Complementing tool-based studies, several works directly investigate practitioners' perceptions of refactoring. For instance, Wang interviewed 20 developers to understand the intrinsic and extrinsic motivators for refactoring, identifying factors such as responsibility, self-esteem, perceived value, and recognition from others [125]. Kim et al. conducted a field study to understand the refactoring benefits and challenges at Microsoft, highlighting the practical challenges and risks they face [59]. Eilertsen et al. evaluated the usability of refactoring tools using the experiences of 17 software developers attempting three software change tasks, emphasizing the need for better tool support to facilitate refactoring tasks in *Non-Sci-SW* [31]. In addition, Ivers et al. explored large-scale refactoring efforts in the industry, highlighting the reliance on general-purpose tools like IDEs rather than specialized refactoring tools [51].

While these studies have advanced our understanding of refactoring in *Non-Sci-SW*, our study differentiates itself by focusing on *Sci-SW*, where *Sci-SW* developers often lack awareness of SE practices [17], coupled with the complexity and multidisciplinary nature of scientific domains, further complicates refactoring efforts [8]. Moreover, it is unclear whether findings from *Non-Sci-SW* refactoring research apply to *Sci-SW*, as it is developed by domain experts with limited SE training and prioritizes correctness and reproducibility over maintainability, making refactoring decisions particularly challenging [44, 95, 101]. Given these fundamental differences, existing *Non-Sci-SW* refactoring insights may not fully account for the unique challenges associated with *Sci-SW*. To bridge this gap, our research investigates refactoring practices and challenges particularly in the context of *Sci-OSS* development.

3.2 Refactoring Detection Tools

Refactoring detection is valuable in various applications, such as refactoring recommendation tools (e.g., name recommendation for renaming refactorings [29]), identification of refactoring opportunities [78], code evolution analysis [21, 64],

and code review support [6]. However, it remains challenging due to the lack of documentation [4] and the frequent entanglement of refactoring with other code changes [73]. Early refactoring detection tools (e.g., *RefFinder* [89]), relied on code similarity metrics to identify Java refactorings by analyzing *commit diff*. Later tools (e.g., *RefactoringCrawler* [27] and *JDEvAn* [128]) improved detection through syntactic and semantic analysis. Most tools, however, are programming language (PL)-specific. To address this, *RefDiff2.0* [97] introduced multi-language support for C, Java, and JavaScript using a code structure tree representation. More recently, *RefactoringMiner3.0* [3] set the new SOTA in Java refactoring detection, significantly improving precision and recall[119, 120] and outperforming *RefDiff2.0*. Inspired by *RefactoringMiner*, *PyRef* [10] was developed for Python, offering higher precision and detecting more refactorings than its adapted counterpart [28].

While these tools provide valuable insights into refactoring practices in *Non-Sci-SW*, their applicability to *Sci-SW* remains unknown. This study addresses this gap by evaluating three SOTA refactoring detection tools, each recognized as the most effective for its respective PL. Specifically, we evaluate *RefDiff2.0* (C) [97], *RefactoringMiner3.0* (Java) [3], and *PyRef* (Python) [10] on *Sci-OSS*. Our findings enhance the understanding of refactoring in *Sci-OSS* and reveal the applicability and limitations of existing tools in this context.

3.3 Scientific Software (*Sci-SW*) Development

Scientific software is a subset of software, that is distinguished by its reliance on scientific models and specialized algorithms that meaningfully impact refactoring practices [57]. *Sci-SW* influences research methods and teamwork throughout scientific discovery, acting as both an instrument and an output [47, 48, 53]. Different from *Non-Sci-SW* where the major contributors do have professional SE background, *Sci-SW* is primarily created by scientists with limited software engineering (SE) training [83], leading to differences in development practices, collaboration, and sustainability [117]. Similarly, *Sci-OSS* developers often unconsciously adopt agile-like methodologies, valuing version control and testing, especially in multi-developer research groups [46]. However, these practices are not applied systematically, and Storer et al. showed that SE best practices are not always feasible in *Sci-SW* development and must be adapted to better support *Sci-SW* development [101].

Hanney et al. found that scientists often learn software development through peer interaction and self-study, with many feeling inadequately trained in software testing [44]. Although mostly self-taught, studies indicate that *Sci-SW* developers are aware of certain best practices of software development. For instance, Arvanitou et al. conducted a systematic mapping of SE practices in *Sci-OSS* development and revealed that *Sci-SW* developers prioritize productivity-related best practices, such as code reuse and the utilization of third-party libraries [8]. Similarly, a survey by Eisty and Carver, highlights that many *Sci-SW* developers have only partial knowledge of testing techniques, rely on ad hoc or minimal testing, and perceive industrial testing approaches as difficult to apply in research contexts [32]. This gap between SE and *Sci-SW* development practices necessitates evolution in both fields to bridge this divide [101]. Together, these findings point to a persistent gap between established SE practices and their practical use in scientific settings. This gap is further reinforced by the unique characteristics of *Sci-SW* development. Domain knowledge often takes precedence over formal SE expertise, and requirements frequently emerge during the development process rather than being specified upfront [95]. In addition, Prabhu et al. showed that *Sci-OSS* is commonly developed under time and performance constraints, using heterogeneous toolchains and with limited adoption of modern SE practices [88]. Language choices also play a role, as shown by Leroy et al., the use of high-level scripting languages, domain-specific languages, or low-level HPC languages introduces trade-offs between expressiveness, performance, and maintainability [63].

Beyond methodological differences, *Sci-SW* development also involves substantial additional, often undervalued effort. Scientists must frequently handle tasks such as documentation, user support, training, and community management, despite limited institutional recognition or funding for this work [117]. The broader academic incentive structure further exacerbates this issue by prioritizing publications over software maintenance and sustainability [47, 117].

Recent work examining scientific code artifacts provides further insight into the consequences of these challenges. Studies of notebooks and scripts, particularly in Python-based environments reveal frequent style violations, limited testing, and poor documentation, all of which hinder code comprehension and complicate safe refactoring [18, 75, 96, 124]. Complementary survey-based research by Chen et al. shows that while scientists widely recognize the importance of readable code for reproducibility, many lack formal training and struggle with issues such as inadequate documentation, poor naming practices, and limited use of quality assurance tools [18]. These findings suggest that observable quality issues in *Sci-SW* are not isolated problems, but rather symptoms of deeper structural tensions between domain-driven requirements (e.g., correctness and performance) and long-term software qualities such as maintainability and evolvability. Prior work has also examined code quality issues in scientific artifacts such as notebooks and Python scripts, reporting problems including poor documentation, inconsistent coding styles, and limited testing. These findings provide important context for understanding why refactoring in *Sci-SW* is often challenging and error-prone. These quality issues directly impact refactoring in *Sci-OSS*, as poor documentation, weak testing practices, and inconsistent coding styles make it difficult to safely restructure code while preserving scientific validity. This context helps explain why refactoring in *Sci-OSS* is both challenging in practice and difficult for existing automated tools to accurately detect.

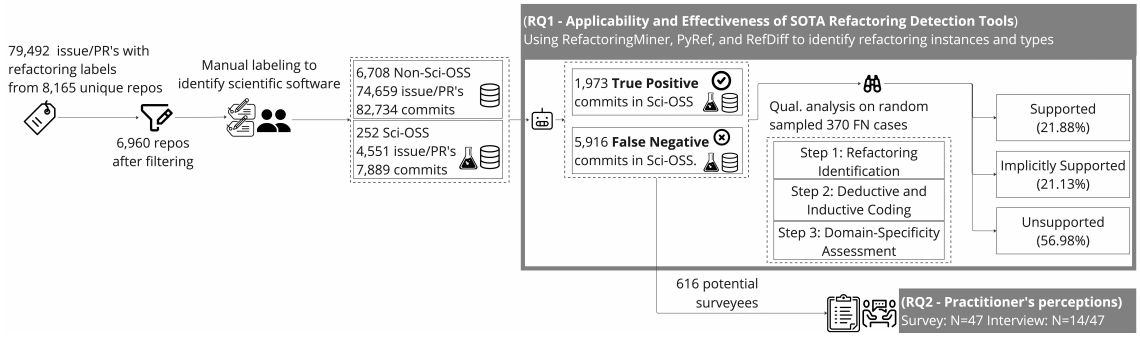
These characteristics of *Sci-SW* development add more complexity beyond the traditional software development process due to its reliance on domain knowledge, emergent requirements, and specific needs. This distinction enables comparisons between *Sci-OSS* and *Non-Sci-OSS*, particularly in the context of refactoring, where domain-driven computational constraints can significantly influence software design and evolution, as refactoring must not only preserve functional correctness but also the integrity of the underlying scientific results [44, 83, 86]. As a result, refactoring practices in *Sci-OSS* may diverge from those commonly observed in *Non-Sci-OSS*, where domain-specific validity concerns are typically less pronounced [44, 83, 86]. This necessitates the need for a systematic investigation of how refactoring practices in *Sci-SW* diverge from *Non-Sci-SW*, with the long-term goal of aligning *Sci-SW* development with SE best practices to support an effective development process without increasing scientists' workload. Similar comparison studies exist for refactoring in machine learning systems [103] and Android applications [70, 84].

3.4 Refactoring Practices in *Sci-SW*

Refactoring in *Sci-SW* has been explored in a few studies, shedding light on both its benefits and challenges. Prior work has shown that SE practices, including refactoring play an important role in improving the sustainability of *Sci-OSS*. For example, practices such as incremental design, unit testing, version control, and systematic refactoring help teams manage schedule and quality risks in bioinformatics projects, ultimately leading to more maintainable codebases [23].

Despite these benefits, refactoring in *Sci-SW* is often perceived as necessary yet labor-intensive, and is frequently deferred until late stages of the research lifecycle. A qualitative study by Pertseva et al. [83] shows that scientists typically invest substantial additional effort after obtaining results to prepare code for sharing. This work includes cleaning exploratory notebooks and scripts, extracting reusable components, improving naming and documentation, and adding basic tests. Notably, such efforts are often undertaken without formal recognition, causing refactoring to compete with other academic priorities. These findings suggest that, while scientists value refactoring, it is often treated as a secondary activity rather than an integral part of development.

Fig. 2. Research method overview.



In some domains, however, the feasibility of refactoring is further constrained by technical requirements. In climate modeling, for instance, strict bit-wise reproducibility is enforced, requiring that code modifications do not alter the exact bit values of computed variables [30]. Because these models are executed across heterogeneous hardware and parallel environments, even minor structural changes can affect floating-point behavior. As a result, many conventional refactoring practices are considered risky or impractical, significantly limiting how the code can evolve. However, these studies focus on specific domains or conceptual discussions. In contrast, our work systematically investigates refactoring across diverse scientific fields using a mixed-methods approach to identify common challenges and opportunities, offering a comprehensive overview of practices, tools, and their impact on *Sci-SW* development.

4 RQ1: Effectiveness of SOTA Techniques to Identify Refactoring in *Sci-OSS*

In this section, we present our method (Sec.4.1) and evaluate the effectiveness of SOTA refactoring detection tools in identifying refactorings within *Sci-OSS*, comparing the results with those observed in *Non-Sci-OSS*. Since large-scale empirical studies of refactoring rely on automated detection techniques, assessing whether these tools can reliably identify refactorings in scientific software is a necessary prerequisite for studying refactoring practices in this domain. We first present quantitative findings (Sec.4.3), which reveal that these tools fail to detect refactorings in approximately 75% of cases in *Sci-OSS*. To investigate the underlying limitations, we then conduct a qualitative analysis (Sec.4.1.4). Fig. 2 provides an overview of our methodology.

4.1 Methodology

We construct a dataset of refactoring practices that includes both *Sci-OSS* and *Non-Sci-OSS*. This involves three key steps: (1) collecting issue/PRs labeled as refactoring, (2) differentiating *Sci-OSS* from *Non-Sci-OSS*, and (3) extracting relevant commits and code changes.

4.1.1 ***Collecting Issue/PRs With Refactoring Labels.*** Following the established methods for collecting refactoring-related data [72], we queried the GitHub API [38] to retrieve issue/PRs labeled [42] with the case-insensitive keywords "`refactor`" or "`refactoring`" up to October 31, 2023, to create a ground truth dataset of refactoring changes. We acknowledge the inherent limitation of this method, as projects that do not consistently use refactoring-specific labels may lead to incomplete data. Although achieving complete coverage is impractical, our approach prioritizes high

Table 1. Collected Issue/PR's with refactoring labels. (Ref–Refactoring labeled).

PL	Ref. #issue /PR's	Total #Repos (after filtering)	#Sci-OSS repos	Sci-Ref-#Issue /PR's	Sci-Ref-#Commits (#Excluded)	#Non-Sci-OSS repos	Non-Sci-Ref-#Issue /PR's	Non-Sci-Ref-#Commits (#Excluded)
Java	33,526	3,361 (2,974)	16	311	824 (272)	2,958	33,082	43,421
Python	42,098	4,397 (3,677)	213	3,932	6,084 (518)	3,464	38,021	35,875
C	3,868	407 (309)	23	308	981 (236)	286	3,556	3,438
Total	79,492	8,165 (6,960)	252	4,551	7,889 (1,026)	6,708	74,659	82,734

precision as an initial foundation. Furthermore, since the objective of this study is to examine practitioners' perceptions of refactoring, we require explicitly labeled issue/PRs as sufficient ground truth. Thus, refactoring-related items not explicitly labeled by practitioners fall beyond the scope of our analysis.

C++, C, Java, Python, and Typescript are the top-5 most widely used programming languages in the SciCat dataset [115], which is built on top of World of Code's comprehensive research code database [65]. SciCat aggregates repositories labeled as `isScientificAppSoftware`, `isResearchSoftware`, or `isScienceSupportSoftware`, capturing a wide range of *Sci-OSS*, including application-level tools, research-focused utilities, and supporting infrastructure. Existing refactoring detection tools have been developed for Python, Java, and C (as described in Sec. 3.2), but, to the best of our knowledge, none exist for C++ or TypeScript. Consequently, our study focuses on repositories implemented in the former three PLs. The PL of each repository was determined based on the most frequently used language, extracted via the GitHub API [39]. From the 79K collected issue/PRs, we identified 8,165 unique repositories across Java, Python, and C as detailed in the second column in Table 1, We did not exclude forked repositories because GitHub's forking mechanism does not duplicate parent repository's issues and PR's. Consequently, any refactoring-labeled issues and PR's observed in a fork reflect independent developer activity rather than duplicated records. The final dataset included 323 forked repositories (0.03% of total repos) across all PLs.

4.1.2 ***Differentiating Sci-OSS from Non-Sci-OSS.*** While datasets like SciCat [115] provide automated metadata for scientific repositories (e.g., `isScientificAppSoftware`), these labels were generated using LLMs and demonstrated only moderate agreement with human annotators [115]. To ensure high precision for our ground truth dataset of *Sci-OSS* repos, we opted to conduct manual analysis rather than relying on this automated metadata.

To classify a repository as scientific, we adhered to Kelly's definition: "*application software that includes a large component of knowledge from the scientific application domain and is used to increase scientific knowledge for the purpose of solving real-world problems*" [57]. However, acknowledging the potential ambiguity in this definition, we further refined it by including repositories explicitly mentioning specialized algorithms (e.g., DNA sequence alignment algorithms in broadinstitute/gatk [105]), data structures (e.g., genes datatype in monarch-initiative/dipper [108]), or domain-specific interfaces typically used in scientific computing (e.g., public health surveillance and outbreak management in SORMAS-Project [35]).

Specifically, we iteratively established the following exclusion criteria (EC), to ensure that our dataset accurately reflects this definition. Following these criteria, we manually validated repositories to ensure their alignment with scientific computing contexts. Two authors independently labeled a random sample of 100 repositories per PL, yielding 300 repos in total. We selected this per-language sample to ensure that the exclusion criteria were developed and validated on a representative subset of repositories in each language, rather than being influenced disproportionately by any single language. At the corpus level, this sample size is also conservative, as 300 repositories exceeds the sample

of 264 repos required for the full population of 8,165 repos under the standard empirical software engineering practice of 90% confidence level and 5% margin of error [7, 60]. They resolved discrepancies through discussion to maintain consistent classification, achieving high inter-rater reliability (IRR) in distinguishing between *Sci-OSS* and *Non-Sci-OSS* software, as detailed later.

- **EC1:** We excluded repositories whose names contained words unrelated to scientific computing (e.g., games, demo, tutorial). This filtering process involved iteratively creating a list of exclusion words, ultimately removing 1205 repositories across all PLs. The complete list of keywords is listed in our replication package [92].
- **EC2:** We excluded repositories not tied to a specific scientific domain, following the scientific Python ecosystem structure [45]. The scientific Python ecosystem comprises a foundational layer (e.g., *NumPy, SciPy, and Matplotlib*) supporting technique-specific libraries (i.e., *scikit-learn, pandas*), which in turn underpin domain-specific projects (i.e., *Astropy, Biopython*). Our selection focused on the domain-specific category, excluding repositories from the foundation layer. As a result, we excluded 6,650 repositories across all PLs.
- **EC3:** We exclude dashboards or visualization tools, such as the Covid-19 dashboard [77], as they primarily serve visualization purposes rather than implementing scientific processes or algorithms. Consequently, 58 repositories were excluded.

Using these criteria, we manually classified repositories into *Sci-OSS* and *Non-Sci-OSS*. The labeling process was conducted by two authors with significant expertise in empirical software engineering. We assess IRR using Cohen's kappa statistic [62], achieving scores of 0.87, 0.90, and 0.85 for C, Java, and Python, respectively, indicating near-perfect agreement. Following this validation, the first author labeled the remaining 7,865 repositories, identifying 252 repositories as *Sci-OSS* and 6708 repositories as *Non-Sci-OSS*, as shown in column 4 in Table 1.

This manual classification also enabled the construction of the *Non-Sci-OSS* baseline for our comparative analysis. We did not use existing refactoring datasets from prior studies because they rely on different data collection procedures and ground-truth definitions, which would make comparisons with our *Sci-OSS* dataset inconsistent. In particular, most existing benchmarks infer refactorings from commit histories using automated detection tools or curated change sets [10, 98, 120], rather than from explicitly labeled refactoring issues and PRs. Furthermore, these datasets often focus on highly popular repositories [98] or specific programming languages [10, 120], which can introduce sampling biases and limit generalizability. Constructing our own baseline from the same pool of repositories ensures that both the *Sci-OSS* and *Non-Sci-OSS* datasets were collected using identical criteria, time windows, and labeling procedures. This design choice provides a controlled comparison environment in which differences in refactoring practices or tool performance can be attributed to domain characteristics rather than dataset construction artifacts.

4.1.3 ***Collecting Commits From Issue/PRs.*** We retrieved all commits linked to the collected issue/PRs in both *Sci-OSS* and *Non-Sci-OSS* repositories by parsing timeline events via the GitHub API [41]. Specifically, we extracted commits from events labeled as committed, merged, referenced, closed, and line-commented, yielding a total of 7,889 commits across all *Sci-OSS* repositories. Issue/PRs without linked commits, primarily due to their open status, were excluded, removing 141 entries across all PLs (<0.1%).

Since refactoring detection tools rely on `commit diffs`, we extracted them using the GitHub API [40]. Commits that did not modify at least one C, Java, or Python file were excluded, as the tools are designed for these PLs. This filtering removed 1,026 commits across 826 issues/PRs, accounting for 13% of the collected commits. This filtering ensures a fair and consistent comparison across tools, which do not support cross-language analysis. Including peripheral or unsupported files would risk inflating false negatives and misrepresenting tool capabilities. Our analysis focuses

on whether the refactoring tool correctly identified at least one detectable refactoring instance in the commit. We acknowledge that some commits may contain a mixture of refactoring and non-refactoring changes [5]. For a subset of false negatives, we conducted qualitative analysis to determine why detection tools failed, as described in Sec 4.1.4.

4.1.4 ***Qualitative Analysis Procedure on False Negative Cases.*** We conducted a qualitative analysis on a random sample of 370 false negative (FN) commits that SOTA tools failed to detect, consisting of 103 from C, 62 from Java, and 205 from Python. The sample size was selected to ensure a 95% confidence level with a 5% margin of error. Each FN commit was analyzed independently following a structured three-step process. To ensure consistency and reduce bias, coders held regular meetings at every step to discuss emerging patterns and resolve disagreements.

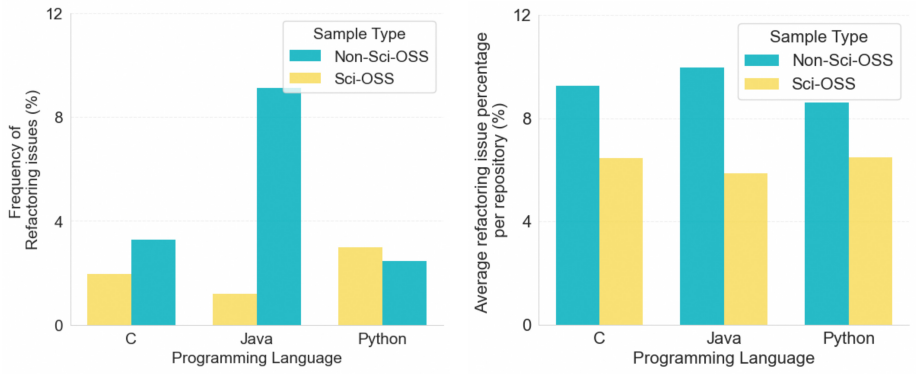
Step 1: Refactoring Identification. We first determined whether each commit constitutes a refactoring. To ensure a comprehensive assessment, we analyzed not only the code diffs but also the associated issue title, body, and developer discussions. We adhered to the definition of refactoring as "*code changes that perform improvements to internal structure, design, or implementation including readability, maintainability, and performance without any change to the external behavior*" [37, 59, 68]. This step aimed to determine whether these FN instances resulted from a misinterpretation of the established refactoring definition.

Step 2: Refactoring Classification. For commits identified as refactorings, we applied a hybrid deductive–inductive coding approach [34] to align it with existing classifications. We labeled refactorings based on Fowler's taxonomy of 66 types [37], along with additional generic refactorings identified in prior ML systems research [103]. Using this same coding framework, we then assessed whether the identified type was theoretically detectable by SOTA approaches. Specifically, since *RefactoringMiner3.0* detects 40 types [110], *PyRef* covers 9 [10], and *RefDiff2.0* covers 10 [90], we categorized instances as '`supported`', '`implicitly-supported`' , or '`unsupported`' to evaluate whether the FNs stemmed from the performance limitations of SOTA tools or fell entirely beyond their intended scope.

Step 3: Domain-Specificity Assessment. Finally, we assessed whether performing the identified refactoring required domain knowledge. A refactoring was considered to require domain knowledge if the changes addressed the unique requirements of a specific domain, involving specialized algorithms, data structures, domain-specific interfaces, or computational constraints beyond general software engineering principles [57, 99]. Our objective was not to fully interpret the underlying scientific theory, but to determine whether such knowledge was necessary to comprehend or justify the change. When a refactoring could not be explained using standard software engineering reasoning alone, this was taken as evidence that domain knowledge was required.

4.2 Frequency of refactoring in *Sci-OSS*

To assess the role of refactoring in *Sci-OSS*, we first examine how frequently *Sci-OSS* codebases are refactored compared to *Non-Sci-OSS*. We measure refactoring frequency as the percentage of issues explicitly labeled as "refactoring" relative to the total number of issues. Because we consider only issues with linked commits, this metric reflects actionable refactoring activity. Unlike *Non-Sci-OSS*, where continuous delivery and long-term maintenance cycles heavily drive development, *Sci-OSS* codebases often prioritize rapid prototyping, experimentation, and reproducibility [30, 44, 58, 102]. These differences may influence how often refactoring is performed in practice. Our analysis shows that refactoring activities occur less frequently in *Sci-OSS* than in *Non-Sci-OSS*. As shown in Fig. 3a, *Non-Sci-OSS* repositories exhibit a higher percentage of refactoring issues than *Sci-OSS* repositories across the dataset, with the largest difference observed in Java. While Python presents a slight anomaly in this aggregate view, where *Sci-OSS* marginally exceeds *Non-Sci-OSS*, this pattern does not hold at the project level. As shown in Fig. 3b, *Non-Sci-OSS* maintains a higher average refactoring



(a) Percentage of refactoring issues. (b) Average percentage of refactorings per repository.

Fig. 3. Overview of refactoring frequency.

percentage per repository across all three PLs compared to *Sci-OSS*, indicating that the higher Python refactoring counts in *Sci-OSS* are driven by a small number of highly active repositories.

These findings show that *Sci-OSS* developers refactor less frequently than *Non-Sci-OSS* developers, likely reflecting the emphasis on rapid experimentation, reproducibility, and publication-driven workflows. At the same time, the results indicate that refactoring is actively performed in *Sci-OSS* rather than being absent. While differences in software engineering experience may influence how refactoring is performed or documented, these findings suggest that refactoring practices in *Sci-OSS* may differ from those in *Non-Sci-OSS*, reflecting domain-specific development constraints and workflows. Nevertheless, refactoring still constitutes an unavoidable portion of their tracked development effort [47, 117]. In particular, even though refactoring occurs less frequently, it remains a recurring and necessary activity as projects evolve beyond initial prototypes or are reused by a broader community. Moreover, refactoring in *Sci-OSS* often involves domain-specific reasoning and higher stakes for scientific correctness, as even small changes can affect data interpretation, reproducibility, or scientific conclusions. Importantly, lower refactoring frequency does not imply reduced importance; rather, it reflects differences in development practices, while refactoring remains essential for ensuring correctness, reproducibility, and long-term usability of scientific software, thereby motivating the need to better understand and support refactoring in this domain.

4.3 Quantitative Findings: Refactoring Detection using SOTA Tools

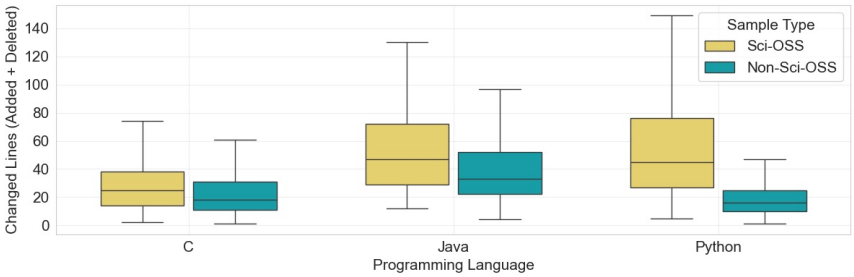
Building on the observation that refactoring is actively performed in *Sci-OSS* (Sec. 4.2), we next evaluate how effectively SOTA tools detect these refactorings. Our goal is not to attribute the observed detection gap solely to the Sci/non-Sci distinction, but to investigate whether refactorings in *Sci-OSS* exhibit characteristics that make them inherently more difficult for existing tools to detect.

Table 2 shows that SOTA tools consistently detect fewer refactorings in *Sci-OSS* projects compared to *Non-Sci-OSS* projects across C, Java, and Python. On average, the detection rate is 25% for *Sci-OSS* projects and 62% for *Non-Sci-OSS* projects. While lower refactoring frequency in *Sci-OSS* could partially contribute to this gap, our analysis in Sec. 4.2 shows that refactoring remains an active and recurring practice, indicating that reduced activity alone cannot explain the observed difference, suggesting that other factors, such as tool limitations and domain-specific refactoring practices,

Table 2. Refactoring detection results using SOTA Tools

SOTA Tools	Language	Sci-OSS (%)	Non-Sci-OSS (%)
PyRef [10]	Python	29.31	67.16
RefactoringMiner3.0 [3]	Java	30.70	73
RefDiff2.0 [97]	Java	17.84	48.12
RefDiff2.0 [97]	C	23	60.27
Average		25.21	62.13

Fig. 4. Changed lines per refactoring commit across project types.



contribute to the gap. Existing refactoring detection tools assume that refactoring patterns are domain-agnostic, yet they have primarily evolved in easily available open-source codebases, such as Apache Java projects [97, 119]. Furthermore, these tools are developed for and tested on the most popular non-scientific repositories [98] or restricted to oracles of particular languages [10, 120], which can introduce sampling biases and limit their generalizability to *Sci-OSS* domains. While this discrepancy may partly reflect differences in refactoring practices, it also indicates that existing tools may not fully capture many refactorings performed in *Sci-OSS*. Findings from our survey and interviews with *Sci-OSS* developers (Sec. 5) further support that refactoring remains a common practice in *Sci-OSS*.

To determine whether this disparity could be attributed to commit size, we compared the median number of changed lines per refactoring commit across project types. As shown in Fig. 4, *Sci-OSS* projects had slightly larger commits (median = 32 lines) than *Non-Sci-OSS* projects (median = 20 lines), Given the skewed distribution of the commit size data, which violates the assumption of normality, we applied a non-parametric, Mann–Whitney U test [67] independently for each programming language. In addition to statistical significance, we computed Cliff's delta (δ) to assess effect size. As shown in Table 3, for each programming language, the Mann–Whitney U test indicates a statistically significant difference ($p<0.001$). The corresponding Cliff's delta values showed $|\delta| \geq 0.6$ [19], indicating a large effect size [94]. This shows that *Sci-OSS* commits are substantially larger than *Non-Sci-OSS* commits across all languages. Although the difference in commit size is statistically significant, the direction of the effect would suggest higher, not lower, detection rates in *Sci-OSS*. Therefore, commit size cannot explain the observed lower detection rates.

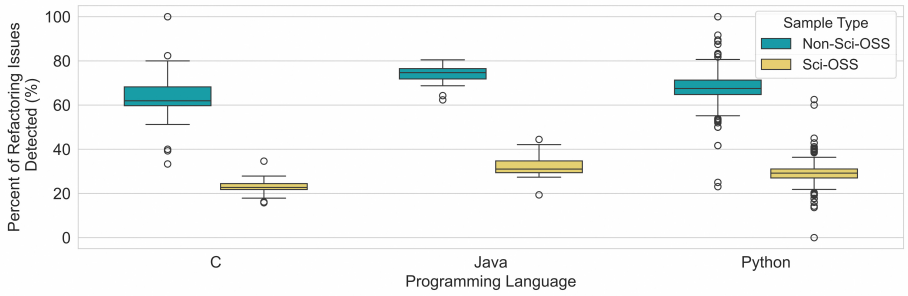
Counterintuitively, despite having larger commits, which theoretically provides more opportunity for refactoring detection, *Sci-OSS* still had a lower detection rate. This clarifies that the lower detection rate in *Sci-OSS* cannot be explained by commit size or a size-related bias favoring *Non-Sci-OSS*. To further validate the detection results, and ensure that the findings were not driven by a small number of projects with unusually high or low detection rates, we performed a project-level analysis. Specifically, for each repository, we computed the proportion of refactoring labeled issues that were detected by any of the three SOTA tools, creating a per-project detection rate allowing us to assess whether the

Table 3. Mann–Whitney U test [67] and Cliff's delta [19] for refactoring commits in *Sci-OSS* are larger than refactoring commits in *Non-Sci-OSS*.

language	U-test	Cliff's Delta
C	$1.64 \times e^{-15}$	0.603
Java	$3.8 \times e^{-12}$	0.648
Python	$1.2 \times e^{-02}$	0.712

overall patterns observed in the aggregate results also hold at the level of individual projects. The results, shown in Fig. 5, corroborate our initial findings on a per-project basis. The percentage of refactoring issues detected by SOTA tools for *Sci-OSS* projects are consistently lower than *Non-Sci-OSS* projects. This per-project detection rate demonstrates that the observed performance gap is a systematic phenomenon, suggesting that existing tools may not fully capture refactorings in *Sci-OSS*.

Fig. 5. Refactoring detection rates per repository and frequency of refactoring issues per repository



4.4 Most Common Refactorings in *Sci-OSS*

To understand the specific refactoring types prevalent in *Sci-OSS*, we examined the most frequently detected refactorings across all *Sci-OSS* repositories. Table 4 reports the ten most frequent refactoring operations identified by the SOTA refactoring detection tools for each PL in our dataset. Consistent with prior works in *Non-Sci-OSS* [3, 10, 97, 98], refactoring activity in *Sci-OSS* is dominated by relatively localized transformations involving identifier renaming, interface and signature adjustments, code movement, and extraction. This heavy reliance on parameter and naming changes likely reflects the iterative nature of *Sci-OSS* development, where researchers continuously tune algorithms, adjust data input shapes, and refine domain terminology as their experiments evolve. However, this distribution may partly be an artifact of detection capabilities and sensitivity of existing tools rather than the full spectrum of refactoring activity, next section shows the results of a qualitative analysis of false negatives to better understand the refactorings SOTA tools failed to detect.

4.5 Qualitative Findings from False Negative Analysis

Building on the qualitative analysis procedure described in Sec. 4.1.4, we report the findings from the analysis of FN cases, structured according to the three steps of refactoring identification, classification, and domain knowledge assessment.

4.5.1 ***Results for Refactoring Identification (Step 1).*** From the 370 FN cases, we confirmed 98 cases to be true negatives (TN) (36.7%) that were labeled as refactorings by *Sci-OSS* developers but the code changes did in fact modify

Table 4. Most common refactoring types detected in *Sci-OSS*.

C (RefDiff 2.0 [97])	Total	Java (RefactoringMiner 3.0 [3])	Total	Python (PyRef [10])	Total
Rename Method	327	Change Variable Type	217	Move Method	594
Move Function	195	Change Parameter Type	177	Rename Class	472
Extract Function	162	Change Return Type	142	Add Parameter	415
Move File	159	Remove Parameter	131	Remove Parameter	369
Change Signature	133	Move Method	130	Extract Method	313
Pull Up Method	131	Rename Variable	128	Rename Parameter	175
Move & Rename Function	116	Rename Parameter	124	Extract Variable	161
Push Down Method	104	Rename Method	77	Move Class	159
Rename File	88	Add Parameter	70	Push Down Method	158
Inline Function	74	Change Attribute Type	69	Rename Method	143

external behavior, contradicting the established definition [37, 59, 68]. Examples include radis/radis (PR#244), NASA-AMMOS/aerie (PR#423), and opensearch-project/geospatial (PR#210). Additionally, seven cases (2.62%) involving large commit sizes (>1000 changed lines) were excluded as they were too large for manual analysis, such as broadinstitute/gatk (PR#2493). This resulted in 265 FN cases for the next step of qualitative analysis. Comparison of coder labels produced a Cohen's kappa score of 0.72, indicating a substantial level of agreement between the coders [62].

4.5.2 ***Results for Refactoring Classification (Step 2).*** After excluding the 98 TN cases and seven large commit cases, the remaining 265 FN cases were categorized into three types according to the predefined qualitative analysis process. Comparison of coder labels produced a Cohen's kappa score of 0.77, indicating substantial agreement for the coding process [62]. Since our goal is to evaluate tools from a user's perspective, i.e., what is explicitly reported versus what must be inferred, we believe this methodology provides an appropriate and reproducible basis for comparison. Accordingly, the 265 FN cases were classified as follows:

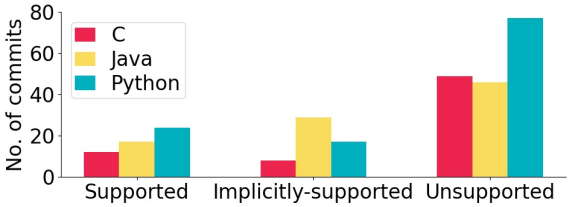
- **`Supported`** (58/265, 21.88%): refactorings that the SOTA tools are designed to detect but failed to do so.
- **`Implicitly-supported`** (41/265, 15.47%): refactorings that are not explicitly reported by the tool but can be identified in the generated AST diffs.
- **`Unsupported`** (166/265, 62.64%): refactorings that fall outside the capabilities of current tools, including (a) refactorings defined in existing taxonomies [37, 103] but not detectable by SOTA approaches, and (b) previously unseen refactorings that extend existing classifications.

Note that we do not claim `Unsupported` refactorings represent entirely new refactoring types; rather, they extend existing taxonomies [37, 103] by highlighting refactorings that remain unsupported by current detection approaches. In Fig. 6, we present the distribution of two categories of labeling results, classified by PL. The high relative occurrence of FNs in `Unsupported` refactorings across all PLs highlights the necessity of developing more effective tools to address `Unsupported` refactoring types and underscores the need for a qualitative study to collect perceptions of refactoring from practitioners (Sec. 5).

4.5.3 ***Results for Domain Knowledge Identification (Step 3).***

Summary. As presented in Table 5, we identified seven **`supported`** refactorings (58/265), five **`implicitly supported`** refactorings (41/265), and ten **`unsupported`** (166/265) refactoring types, aligning with previously defined refactoring categories [3, 10, 37, 97, 103]. Notably, three of the nine unsupported types representing previously unseen refactorings.

Fig. 6. Qualitative analysis of false negative cases.



As mentioned earlier, these do not constitute entirely new refactoring types; rather, they extend existing taxonomies [3, 10, 37, 97, 103] by highlighting refactorings that remain unsupported by current detection approaches. Unlike prior work on ML system refactorings [103], which identified general and ML-specific refactorings (e.g., replace with sparse matrix, push down hyperparameters), our analysis did not reveal any refactorings exclusive to a particular scientific domain. **However, domain knowledge was crucial for implementing refactorings**, with 179 out of 265 cases (67.54%) requiring such expertise. The proportion varied by category: `Supported` (18.96%), `Implicitly-supported` (65.85%), and `Unsupported` (84.93%). To ensure the reliability of our classification regarding domain knowledge requirements, we compared the coder labels resulting in Cohen's kappa coefficient of 0.83, indicating near-perfect agreement [62].

Our results show that transforming unsupported refactorings into supported ones would require integrating domain knowledge into automated tooling, moving beyond purely AST-based syntax matching rules. Moreover, as shown in Table 5, 18.96% of the supported refactorings Within the scope of refactoring tools support remains undetected in *Sci-OSS* because their validity depends on domain knowledge that is not encoded in the source code.

4.6 In-Depth Analysis of False Negative Refactorings

To explain the limitations of SOTA refactoring detection tools, we present an in-depth analysis of representative refactoring types (Table 5). These categories differ in detectability: **supported** types are explicitly modeled by tools (Sec. 4.6.1), **implicitly-supported** types are observable but not labeled (Sec. 4.6.2), and **unsupported** types fall outside current detection capabilities (Sec. 4.6.3). Because tool support varies across programming languages, some refactoring types appear in multiple categories depending on the language context (e.g., supported in Java but unsupported in C/Python). Accordingly, we analyze refactorings based on their detectability within each language setting and discuss representative cases to highlight key challenges rather than exhaustively describing all entries in Table 5. We focus on cases that require domain knowledge, as they account for the majority of false negatives. The illustrative examples demonstrate how domain knowledge shapes refactoring decisions and why such cases are difficult for existing detection approaches to identify. The complete codebook and examples are available in the replication package [92].

4.6.1 ***Supported refactorings.*** We identified seven `supported` refactoring types (58/265), which are refactorings that the SOTA tools are designed to detect but missed in our dataset. Among these, 11 cases (18.96%) required domain knowledge, indicating that domain-specific reasoning is present even within tool-supported refactorings. Missed detections are not exclusively associated with domain knowledge. Notably, **Renaming (13/265) and Move Method (12/265)** do not require domain knowledge, yet were still not detected, suggesting limitations in current detection techniques. Table 5 further illustrates this language-dependent variation: some refactorings are detectable in one language (e.g., Java) but not in others (e.g., C and Python), reflecting differences in tool implementations rather than differences in the underlying refactoring intent.

Table 5. Distribution of refactoring types and dependence on domain knowledge. DK-req. (%)–the proportion requiring domain knowledge. Refactoring actions without a citation are previously unseen refactorings that extend existing taxonomies [3, 10, 37, 97, 103]. Programming language applicability is reported in a separate column; identical refactoring actions may appear multiple times under different language contexts, reflecting variations in tool support.

	Refactoring Action	Applicable PL(s)	Driver (Code Smell)	#	DK-req. (%)
Supported	Renaming [37]	All	Clarity/Consistency	13	0 (0%)
	Move method [37]	All	Feature envy; Low cohesion	12	0 (0%)
	Extract method [37]	All	Long method; Duplicated code	11	2 (18.18%)
	Merging methods [37]	All	Duplicated code; Low cohesion	9	3 (33.33%)
	Collapse hierarchy [37]	All	Unnecessary abstraction	8	5 (62.5%)
	Convert data types using built-in types to enhance precision [3]	Java	Precision/units mismatch	4	0 (0%)
	Change custom defined data type of variables, func. arguments [3]	Java	Inefficient data representation	1	1 (100%)
	Subtotal			**58**	**11 (18.96%)**
Implicitly Supported	Duplicate code elimination [103]	All	Duplicated code	10	9 (90%)
	Reorganization [103]	All	Low cohesion; Poor readability	9	9 (100%)
	Slide statements [37]	Java, Python	Poor readability	9	3 (33.33%)
	Adapt code to accommodate dependency updates [3]	Java	Breaking dependency API change	7	2 (28.57%)
	Remove custom func. with built-ins/dependency-provided func. [3]	Java	Reinventing the wheel	6	4 (66.66%)
	Subtotal			**41**	**27 (65.85%)**
Unsupported	Domain-driven code simplification	All	Unnecessary complexity	51	47 (92.15%)
	Dead code removal [37, 103]	All	Dead/unreachable logic	39	31 (79.48%)
	Substitute algorithm [37]	All	Inefficient/outdated algorithm	24	24 (100%)
	Remove custom func. with built-ins/dependency-provided func. [10, 97]	C, Python	Reinventing the wheel	15	14 (93.33%)
	Convert data types using built-in types to enhance precision [10, 97]	C, Python	Precision/units mismatch	13	8 (61.53%)
	Adapt code to accommodate dependency updates [10, 97]	C, Python	Breaking dependency API change	12	10 (83.33%)
	Change custom defined data type of variables, func. arguments [10, 97]	C, Python	Inefficient data representation	5	5 (100%)
	Slide statements [37]	C	Poor readability	4	2 (50%)
	Code formatting changes	All	Inconsistent formatting	2	0 (0%)
	Add values to dict instead of creating multiple variables	Python	Data clump	1	0 (0%)
	Subtotal			**166**	**141 (84.93%)**
	Total			**265**	**179 (67.54%)**

Extract Method (11/265, 18.18% domain-specific) is a standard refactoring in Fowler's taxonomy [37]. However, some cases required more than structural decomposition. Although tools can detect that code has been extracted into a separate method, they cannot determine whether the chosen extraction boundary preserves the scientific assumptions under which the computation remains valid. This distinction is critical in *Sci-OSS*, where isolating code into a helper method may preserve execution behavior while still separating a calculation from the domain context that gives it the correct interpretation. These findings highlight that even when a refactoring is fully supported by existing detection tools, applying it correctly in *Sci-OSS* may still require reasoning beyond structural correctness. As a result, even supported refactorings may require reasoning beyond structural correctness. In these cases, correctness is defined not only by preserving functional behavior, but also by preserving the scientific assumptions underlying

the computation. Consequently, a refactoring may appear behavior-preserving at the program level while altering the scientific interpretation of its results.

Extract Method
Illustrative Example: monarch-initiative/dipper issue#87

Before refactoring: The threshold computation was embedded within a larger data-parsing loop and included both arithmetic operations and data-normalization steps that established the assumptions under which the threshold was valid.
After refactoring: The computation was extracted into a dedicated method that retains both the normalization logic and the arithmetic calculation.
Why domain knowledge was required: A purely structural decomposition would suggest extracting only the arithmetic operations into a helper method (e.g., `calculate()`). However, this would separate the threshold formula from the normalization steps that define the validity of its inputs. Correctly identifying the extraction boundary requires understanding that the biological constants are meaningful only under a specific normalization regime and should not be reused independently. Without this knowledge, the refactored code may remain functionally executable while enabling incorrect reuse on unnormalized data, thereby violating scientific validity.
Implication: The correctness of this refactoring depends on preserving domain-specific invariants across method boundaries, illustrating that extraction decisions cannot be validated from structural decomposition alone but require reasoning about the semantic context in which computations are valid.
Similar Examples: PlasmaPy/PlasmaPy issue#255, SophT-Team/SophT issue#103, and GliderGeek/PySoar issue#28

Merging Methods (9/265, 33.33% domain-specific) is also a supported refactoring in Fowler's taxonomy [37]. However, in our dataset, some instances required domain knowledge to determine whether apparently similar methods were in fact semantically equivalent. While tools can detect that multiple methods were merged into one, they cannot establish whether the merged methods carried distinct scientific assumptions, data interpretations, or workflow roles that should remain separate. In *Sci-OSS*, the challenge is not only identifying redundancy, but determining whether methods perform the same domain-relevant operation under the same scientific conditions. This requires reasoning about domain semantics beyond structural similarity.

Merging Method
Illustrative Example: NASA-AMMOS/aerie PR#638

Before refactoring: The Sequence eDSL API and the Seq JSON specification interface existed as separate but related interfaces for representing sequencing behavior. This separation meant that overlapping sequencing concepts were maintained across distinct abstractions, requiring translation or adaptation between the eDSL-side representation and the JSON-spec-side interface.
After refactoring: The Seq JSON Spec and the Sequence eDSL API were merged by adapting the Sequence eDSL to use the `SeqJson` interface defined in the specification.
Why domain knowledge was required: The merge required understanding how the Sequence eDSL and SeqJSON specification represent sequencing constructs within Aerie's mission-planning and command-sequencing workflow. Ensuring correctness depended on preserving compatibility with existing sequencing contexts (e.g., Clipper), indicating that the interfaces captured domain-specific semantics that could not be altered without affecting system behavior.
Implication: Determining whether methods can be safely merged requires reasoning about the equivalence of their domain-level semantics and compatibility constraints, which cannot be inferred from structural similarity alone.
Similar Examples: monarch-initiative/dipper issues#96 and dylancromer/maszcal issues#32

Collapse hierarchy (8/265, 62.5% domain-specific). This refactoring involves developers revising inheritance relationships and class responsibilities to better reflect domain concepts rather than relying on generic or overloaded abstractions. Although hierarchy-related refactoring is identified in standard refactoring taxonomy [37], their correctness in *Sci-OSS* depends on whether the revised class structure preserves scientifically meaningful distinctions. Unlike generic hierarchy collapses motivated by simplification or code reuse, this change depends on conceptual models drawn from domain knowledge, its correctness depends on maintaining scientifically meaningful inheritance relationships reflecting physical taxonomies. As a result, their validity cannot be determined from syntax-level inheritance changes alone. Instead, it requires assessing whether the revised hierarchy continues to encode the intended domain semantics. This highlights a limitation of existing approaches, which can detect structural modifications but cannot verify the preservation of domain-aligned abstractions.

Collapse hierarchy
Illustrative Example: PlasmaPy/PlasmaPy issue#687

Before refactoring: Particles such as ions, electrons, leptons, and baryons were all represented as instances of a single `Particle` class, with category membership handled dynamically through calls such as `Particle.is_category("lepton")` rather than being encoded directly in the class hierarchy.
After refactoring: An explicit class hierarchy was introduced. `Particle` class remains as a factory, while `GenericParticle` served as an abstract base class, with domain-specific particle types organized through inheritance relationships.
Why domain knowledge was required: Determining the hierarchy required understanding which physical distinctions among particles should be represented as inheritance relationships. This includes reasoning about relationships between categories (e.g., leptons, baryons, ions, neutral atoms, antiparticles) and whether multiple inheritance is scientifically and computationally appropriate. The refactoring aligns the code structure with a physical taxonomy rather than a purely software-oriented decomposition.
Implication: The correctness of hierarchy transformations depends on preserving domain-level conceptual distinctions, illustrating that inheritance restructuring cannot be validated from structural changes alone but requires reasoning about the underlying domain taxonomy.
Similar Examples: bioinf-mcb/Metagenomic-DeepFRI issue#3 and MDAnalysis/pmda issue#103.

These findings indicate that existing structural refactoring detection techniques capture only a subset of refactorings in *Sci-OSS*, as many refactorings are inherently domain-driven and cannot be identified through structural patterns alone. This suggests that the limitations of existing detection techniques are not solely technical, but are closely tied to domain-specific reasoning and developer practices. To better understand these underlying factors, we next investigate how developers perceive and perform refactoring in *Sci-OSS*. While supported refactorings reveal limitations even within tool scope, implicitly-supported and unsupported refactorings further expose gaps where structural signals are insufficient for detection.

4.6.2 ***Implicitly-supported Refactorings.*** We identified five `implicitly-supported` (41/265) refactoring types, which are refactorings that are not explicitly reported by tools but can be identified in the generated AST-level diffs of SOTA tools [3, 10, 97].

Duplicate Code Elimination (10/265, 90% domain-specific). This refactoring removes repeated code fragments by consolidating them into a shared implementation, typically through extraction into a reusable helper method or by redirecting multiple call sites to a common abstraction. Such patterns have been observed in prior studies on refactoring in machine learning systems [103]. In *Sci-OSS*, duplicate code often arises from repetitive data-processing pipelines, where slight variations in computations lead to similar implementations [103]. Consolidating these fragments into

a reusable function reduces maintenance overhead and improves readability. While SOTA refactoring tools support individual *Extract Method* operations, they do not explicitly capture the higher-level pattern of duplicate code elimination when multiple extractions contribute to the same abstraction. Identifying this refactoring therefore requires aggregating related edits across the codebase. Therefore, we classify these cases as `implicitly-supported`.

Duplicate Code Elimination
Illustrative Example: casact/FASLR issue#55

Before refactoring: The data-display method accepted arguments that could instead be derived from the parent triangle object, as indicated by the issue description. This design required the same or closely related argument-handling logic to be repeated at call sites or within related display paths, despite the information being available in the shared domain object.
After refactoring: The refactoring reduced the arguments passed to the data-display method by making the method rely on the parent triangle representation rather than on separately supplied, redundant inputs.
Why domain knowledge was required: Determining which arguments were redundant required understanding what information the parent triangle object already encoded, which display behaviors depended on that information, and whether removing explicit inputs preserved the intended semantics.
Implication: This refactoring requires recognizing redundancy across multiple call sites and abstractions, illustrating that duplicate elimination is a cross-cutting pattern that cannot be inferred from individual structural edits alone.
Similar Examples: ilastik/volumina issue#24 and Keesaco/KeesaFlo issue#71

Reorganization (9/265, 100% domain-specific). This refactoring type captures cases in which developers restructure existing code or configuration artifacts to improve clarity, consistency, modularity, or alignment with the surrounding workflow without changing the intended external behavior. Similar to prior work in machine learning systems [103], these refactorings involve restructuring code to improve clarity, modularity, or workflow alignment without necessarily altering program semantics. Reorganization may not be reported under a single explicit label by the SOTA tools, it can be inferred from a combination of supported refactorings and analysis of AST-level changes, therefore we classify these cases as `implicitly-supported`.

Reorganization
Illustrative Example: SORMAS-Foundation/SORMAS-Project issue#5249

Before refactoring: The ordering of feature types in `FeatureType.java` was described as random and uncategorized, and `sormas.properties` did not reflect the full set of explanations that had been added to the new Wiki guides. This created a mismatch between implementation artifacts and the documentation that users and developers would consult to understand available feature configurations and server properties.
After refactoring: The code reorganizes both `FeatureType.java` and `sormas.properties` to isolate domain-specific transformations from generic validation logic, enabling easier extension for new reporting formats.
Why domain knowledge was required: The refactoring required understanding how feature configurations and server properties are conceptually grouped and interpreted, as well as how the Wiki documentation defines their intended usage. Developers needed to ensure that reordering and revising descriptions preserved the intended meaning of feature flags and configuration settings.
Implication: Correctness depends on aligning code and configuration structure with domain-specific conceptual groupings and documentation, illustrating that reorganization requires reasoning about semantic consistency across artifacts rather than structural changes alone.
Similar Examples: TheDeanLab/navigate issue#19 and ilastik/volumina issue#26.

4.6.3 ***Unsupported Refactorings.*** As shown in Table 5, some refactoring types appear in both supported and unsupported categories depending on the programming language. We focus here on cases that are outside current SOTA detection capabilities. Specifically, we identified ten `unsupported` (141/265) refactoring types, the majority of which (84.93%) require domain knowledge, although they have been identified in prior works [3, 10, 37, 97, 103]. Unlike supported refactorings, these transformations are not explicitly detected by existing tools. In several cases, support is partial and varies by programming language. For instance, refactorings such as replacing custom implementations with built-in or dependency-provided functions, adapting code to accommodate dependency updates, or converting data types for precision are implicitly-supported in Java [3], but are not supported in C and Python [10, 97]. In Table 5, these refactorings are categorized as unsupported in contexts where they fall outside the capabilities of current tools, emphasizing that detectability varies by both refactoring type and programming language.

Domain-driven Code Simplification (51/265, 92.15% domain-specific) is the most frequent `unsupported` refactoring type in our dataset. Although the term "*code simplification*" may suggest routine cleanup, 92.15% (47/51) of these cases required domain knowledge. These refactorings remove redundancy that is not structurally obvious and can only be validated through reasoning about scientific context, such as the semantic guarantees of domain utilities, data-shape assumptions, and domain constraints rather than local control flow. While not explicitly named in standard structural taxonomies [37], these changes satisfy the core definition of refactoring because they improve internal structure and maintainability without altering external behavior. They can thus be understood as domain-specific manifestations of known standard refactorings (e.g., `Remove Dead Code` or `Consolidate Conditional Expression`). However, unlike conventional code simplifications, where redundancy is evident from code inspection alone through control-flow or data-flow analysis, these refactorings require reasoning about the scientific context in which the code operates. These findings suggest that existing refactoring detection approaches, which rely on structural patterns, are fundamentally limited in capturing such domain-dependent transformations.

Domain-driven Code Simplification
Illustrative Example: SORMAS-Foundation/SORMAS-Project issue#7971

Before refactoring: Developers manually inserted explicit list-size conditional checks (e.g., `if (list.size() > ModelConstants.PARAMETER_LIMIT)`) across multiple files prior to executing database queries.
After refactoring: These manual control-flow blocks were removed entirely; the logic was delegated to an existing utility, which already implements the intended batching and parameter-limit semantics. The maintainer enumerated a long list of methods where this pattern occurred, demonstrating a recurring, domain-specific idiom.
Why domain knowledge was required: The refactoring required understanding that the existing utility already enforces batching and parameter-limit constraints specific to epidemiological queries. This relationship is not explicit in the local control flow, but is instead reflected in how the utility is consistently used across the codebase.
Implication: Such refactorings cannot be reliably identified from local code structure alone; they require reasoning about the semantics of domain-specific APIs and their usage across multiple locations.
Similar Examples: Breakthrough-Energy/PowerSimData issue#265, NASA-AMMOS/aerie PR#553, and tskit-dev/tskit issue#2766.

Dead Code Removal (39/265, 79.48% domain-specific). is the second most frequent `unsupported` refactoring type in our dataset. This refactoring eliminates code that is no longer needed for the intended computation, while preserving the program's externally observable behavior. Although *dead code removal* is recognized in prior refactoring taxonomies [37, 103], 79.48% (31/39) of these cases required domain knowledge to determine whether the code was truly obsolete. Unlike conventional dead code removal, which is often justified by syntactic indicators such as unreachable branches or call-graph isolation, *Sci-OSS* refactorings depend on understanding changes in scientific workflows,

algorithmic pipelines, and the domain meaning of intermediate computations. As a result, the decision to remove code cannot be made reliably from AST structure or static analysis alone, it requires confirming that the underlying scientific task is now implemented elsewhere or is no longer part of the computation. These findings further indicate that existing refactoring detection approaches are limited when code appears structurally removable but its redundancy can only be established from domain-level changes in the surrounding workflow.

Dead Code Removal
Illustrative Example: kevlar-dev/kevlar issue#344

Before refactoring: The system contains a legacy `MATEDIST` filtering component used in a genome-assembly workflow to filter read pairs based on distance constraints.
After refactoring: The change removed the `MATEDIST` module, as sequence-based validation methods introduced in a later pipeline stage render its functionality redundant.
Why domain knowledge was required: The removed code is not syntactically unreachable or unused. Determining its redundancy requires domain knowledge of genome assembly workflows and understanding that the updated pipeline already enforces equivalent read-pair validation constraints.
Implication: This refactoring depends on semantic equivalence at the workflow level rather than structural indicators (e.g., dead branches or unused variables). Incorrect removal would risk eliminating behavior relevant to downstream scientific analysis.
Similar Examples: GliderGeek/PySoar issue#91 and ilastik/volumina issue#58.

Substitute Algorithm (24/265, 100% domain-specific). Substitute Algorithm is a recognized refactoring in Fowler's taxonomy [37]. However, all instances in our dataset required domain knowledge to determine whether the replacement preserved the intended scientific or operational behavior. These refactorings do not optimize local implementation details, but they replace one computational procedure with another that must satisfy domain-specific constraints, assumptions, and correctness criteria. As a result, the validity of the refactoring cannot be established from source structure alone, because algorithmic equivalence depends on factors such as optimization goals, mathematical behavior, or workflow-specific requirements. While these transformations satisfy the definition of refactoring by preserving external behavior, verifying that preservation requires reasoning about domain semantics rather than syntactic form. This limitation suggests that existing refactoring detection techniques, which focus on structural edits, are insufficient for identifying algorithm substitutions that must satisfy domain-specific behavioral equivalence.

Substitute Algorithm
Illustrative Example: SORMAS-Foundation/SORMAS-Project issue#11013

Before refactoring: Across several SORMAS directory views, the original `getIndexList` methods retrieved full data transfer objects in a single query, followed by `sort distinct` over all attributes. Due to left joins, this produced large intermediate result sets, although only a bounded page (via `limit` and `offset`) was required.
After refactoring: The implementation was replaced with a two-step strategy: first retrieving identifiers for the requested page while preserving `sortBy` ordering, then fetching the corresponding records via batched retrieval. Subsequent processing operates only on the selected subset.
Why domain knowledge was required: The transformation required understanding the semantics of SORMAS directory views, including the need to preserve pagination and ordering guarantees for epidemiological entities. The replacement is therefore constrained by domain-specific output semantics rather than being a generic query optimization.
Implication. Establishing correctness requires reasoning about behavioral equivalence under domain-specific constraints (e.g., pagination and ordering), which cannot be inferred from structural changes alone.
Similar Examples: tskit-dev/tskit issue#2766, NASA-AMMOS/aerie PR#683, and RI-imaging/ODTbrain issue#9

Removing custom functions in favor of dependency-provided or built-ins functions (15/265, 93.33% domain-specific for C&Python). This refactoring replaces project-specific helper functions or ad hoc implementations with equivalent functionality provided by external dependencies or language built-ins. Across all instances, 85.71% (18/21) required domain knowledge to verify that the replacement preserved behavior.

In Java, such changes can be partially observed by RefactoringMiner3.0 [3] through API-call tracking and statement mappings. However, no explicit refactoring type is reported. In our dataset, this applied to six instances, four of which required domain knowledge. In contrast, all 15 instances observed in C and Python remain `unsupported`, as RefDiff [97] and PyRef [10] do not capture comparable API-evolution or library-substitution change; 14 of those cases required domain knowledge. Although such changes resemble conventional library adoption, correctness depends on domain-specific assumptions about data representation, edge-case handling, workflow boundaries, and scientific requirements. Consequently, structural signals alone are insufficient. While tools can observe that custom code was removed and replaced by a library call, they cannot determine whether the replacement preserves the domain-specific semantics of the original implementation, highlighting a key limitation of current detection approaches.

Removing custom functions in favor of dependency-provided or built-ins functions
Illustrative Example: spinalcordtoolbox/spinalcordtoolbox issue#3905

Before refactoring: The codebase included a custom command-line script, `isct_test_ants`, used as a standalone validation entry point. It operated outside the main testing and dependency-checking pipeline and required manual invocation.
After refactoring: The standalone custom script was removed, and its functionality was integrated into a `pytest`-compatible test within the existing testing framework, leveraging built-in dependency checking (e.g., `sct_check_dependencies`).
Why domain knowledge was required: The change requires understanding that the removed script was not part of the intended user-facing workflow and that its functionality was already covered by `sct_check_dependencies` together with the surrounding testing infrastructure. The refactoring is therefore constrained by domain-specific validation semantics rather than being a generic replacement of custom code with dependency-provided or built-in functionality, thereby limiting the effectiveness of purely syntactic refactoring detection approaches.
Implication: Establishing correctness requires reasoning about semantic equivalence at the workflow level, not about the absence of structural redundancy or the availability of a replacement function. Without that domain understanding, the refactoring could remove behavior that remains relevant to downstream analysis, validation, or interpretation.
Similar Examples: physiopy/phys2bids issue#336, scikit-bio/scikit-bio issue#919, and monarch-initiative/dipper issue#193.

Converting Data Types Using Built-in Types to Enhance Precision (13/265, 61.53% domain-specific for C&Python). This refactoring type captures cases where developers modify or standardize data representations to improve precision, semantic consistency, or interoperability within scientific workflows. Among the 17 instances identified, four occurred in Java and were detectable as type-change refactorings using RefactoringMiner 3.0 [3] none of which required domain knowledge. The remaining 13 instances occurred in C and Python, where RefDiff [97] and PyRef [10] do not support detecting this kind of representation changes; eight of these 13 cases required domain knowledge. Although these transformations may appear as simple type substitutions, these refactorings require understanding how scientific data or domain-facing metadata are represented throughout a workflow. Changes in representation can affect numerical precision, rounding behavior, unit interpretations, and compatibility with downstream computations. Their correctness depends on whether the revised representation preserves the expected semantics of downstream computations, parser logic, and tool interfaces.

Converting Data Types Using Built-in Types to Enhance Precision
Illustrative Example: bioconvert/bioconvert issue#246

Before refactoring: Converter definitions represented in `in_fmt` are inconsistent. For some cases of many-to-one converters, a single input format could be represented as a plain string. As a result, the current implementation had to distinguish between multiple representations of the same conceptual information.
After refactoring: The Converter definitions were standardized by converting single-string `in_fmt` values into one-element tuples, even when only a single string value was provided, removing the need for branching on representation shape and ensured compatibility with downstream bioinformatics tools expecting uniform format conventions.
Why domain knowledge was required: The transformation required understanding how `in_fmt` encodes bioinformatics format signatures and how those representations are consumed by dispatch, validation, and downstream conversion logic.
Implication: Establishing correctness requires reasoning about semantic equivalence between the original and revised representations under workflow-specific interpretation rules, which cannot be inferred from structural type conversion alone.
Similar Examples: NASA-AMMOS/aerie PR#1047, and marek-cottingham/magSonify issues#14.

Adapting Code to Accommodate Dependency Updates (12/265, 83.33% domain-specific for C&Python). This refactoring type captures cases of restructuring existing code to remain compatible with evolving libraries while preserving the scientific or operational meaning of the computation. *Sci-OSS* heavily relies on third-party libraries for specialized tasks, such as numerical simulations and data visualizations. Unlike conventional dependency updates that focus primarily on API compliance, refactorings in *Sci-OSS* require verification that updates to dependencies, should not alter scientific correctness. This could involve adjusting data representations or computations to align with updated dependency semantics, while maintaining consistency with domain-specific assumptions embedded in the workflow. Among the 19 identified instances, seven occurred in Java and could be inferred implicitly through RefactoringMiner3.0 [3] statement mappings and API-level changes, even though the tool does not explicitly report the refactoring type. Two of these cases required domain knowledge. The remaining 12 instances occurred in C and Python, where RefDiff [97] and PyRef [10], do not support detection of dependency-adaptation change. Ten of these twelve cases required domain knowledge. These findings suggest that current refactoring detection approaches are limited for dependency-adaptation refactorings because they may capture local statement edits, but they cannot determine whether the updated code preserves the domain-specific semantics required by the surrounding scientific workflow.

Adapting Code to Accommodate Dependency Updates
Illustrative Example: NASA-AMMOS/aerie issue#501

Before refactoring: Durations used in the scheduling and command-expansion embedded domain-specific languages (eDSL) were represented through `Temporal.Duration`, but the constraints language had not yet been aligned with that representation.
After refactoring: The code was restructured to align a dependent language component with the `Temporal.Duration` representation already used in scheduling-related component to support asynchronous functions introduced in an updated dependency and removed obsolete constructs that masked a latent bug.
Why domain knowledge was required: To perform it correctly, developers had to understand the role of the scheduling eDSL, command-expansion eDSL, and constraints language within Aerie's mission-planning framework, and why duration values must be represented consistently across those interacting layers.
Implication: Establishing correctness requires reasoning about semantic equivalence under workflow-specific temporal constraints, which cannot be inferred from structural changes alone. A syntactically valid update could still alter how timing constraints are represented, propagated, or interpreted in downstream planning behavior.
Similar Examples: Becksteinlab/MDPOW issue#254, radis/radis issue#151, and bioinf-mcb/Metagenomic-DeepFRI issue#51.

Change custom defined data type of variables, function arguments (5/265, 100% domain-specific for C&Python). This refactoring type refers to cases in which developers modify domain-specific data abstractions such as custom classes, structs, or composite representations used in variables, function parameters, or return types to better align with evolving computational or scientific requirements. For Java, type-changes are supported by RefactoringMiner3.0 [3]. However, C and Python are `unsupported`, as RefDiff [97] and PyRef [10] lack the capability to capture type-changes. Notably, all instances of this refactoring required domain expertise. *Sci-OSS* often uses custom data types due to domain-specific constraints, such as specialized data representations, precision requirements. Consequently, refactoring these custom datatypes often involves balancing trade-offs among performance, correctness, and scientific validity, making it non-trivial to substitute them with standard data structures. Although, some type-change refactorings could be performed without domain expertise, no such instances were observed in our dataset.

Change custom defined data type of variables, function arguments
Illustrative Example: monarch-initiative/dipper issue#193

Before refactoring: The pipeline contained code that assumed an older ZFIN data model in which relevant biological records were centered on genotype-based identifiers.
After refactoring: The refactoring revised the pipeline to accommodate ZFIN's updated concept of `FISH`, in which identifiers changed from genotype-based forms to fish-based forms to improve memory utilization and query efficiency.
Why domain knowledge was required: Correctly performing this change requires understanding how biological entities, relationships, and identifiers are modeled and consumed across the pipeline, as well as, what ZFIN's shift from `GENO` to `FISH` meant biologically and how that change affected the interpretation of environments and extrinsic genotypes across the integration workflow.
Implication: Establishing correctness requires ensuring that the new data representation preserves domain semantics and consistently propagates across dependent components and computations. This makes such refactorings difficult to automate or detect, as structural type changes alone do not capture their scientific or computational validity.
Similar Examples: NASA-AMMOS/aerie PR #700 and marek-cottingham/magSonify issues#14.

Code formatting changes (2/265, do not require domain knowledge). These refactorings improve readability, modularity, and maintainability without altering program behavior. They are particularly relevant in scientific applications that manage large datasets or complex configurations, where code clarity supports collaboration and reproducibility. Although unsupported by current detection tools, these changes are structurally simple and do not require reasoning beyond syntactic analysis.

Adding values to dictionaries instead of creating multiple variables (1/265, do not require domain knowledge). This type groups related variables into a unified data structure to improve modularity and scalability. The transformation involves non-trivial structural reorganization without a direct one-to-one mapping between original and transformed elements. While it does not require domain knowledge in this instance, it is difficult to detect because it reflects changes in data organization patterns rather than localized syntactic edits. This highlights a limitation of current AST-based techniques in capturing higher-level structural transformations.

Across these categories, a common pattern emerges: refactorings in *Sci-OSS* often preserve program-level behavior while altering or depending on domain-specific semantics (e.g., scientific assumptions, data interpretation, or workflow alignment). Because these semantics are not explicitly encoded in syntax or structure, AST-based approaches fail to capture the conditions under which a transformation is valid.

Table 6. Sampled survey questions.

Id	Questions [Reference]
Q1	How do you define refactoring? [59]
Q2	What are the primary motivations for you to refactor code? [59, 125]
Q3	What are the challenges when performing refactoring in scientific software? [59]
Q4	What are the risks associated with refactoring in scientific software? [59]
Q5	Do you currently use any refactoring tools for your scientific software projects? If yes, which ones? [31, 59]
Q6	Are there specific refactoring needs in scientific software that current tools do not adequately address? [50]

Overall, undetected refactorings are associated with both domain-specific reasoning requirements and gaps in existing tool support, including language-dependent coverage. These limitations reflect the reliance of SOTA tools on syntactic transformations and structural mappings, whereas many *Sci-OSS* refactorings depend on semantic constraints such as scientific validity, data interpretation, and workflow consistency, which are not represented in AST abstractions. These findings motivate the need to understand how developers reason about such refactorings in practice (RQ2).

5 RQ2: Practitioners' Perceptions and Challenges of Refactoring in *Sci-OSS*

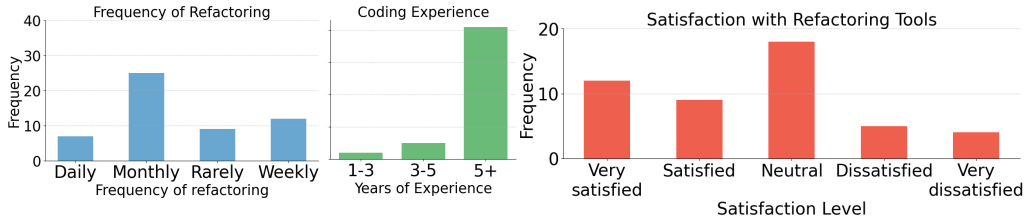
Motivated by the limitations identified in RQ1, particularly the inability of existing tools to capture domain-specific refactorings, we conduct a qualitative study to understand how developers perceive refactoring in *Sci-OSS* and the challenges they encounter in practice.

5.1 Methodology: Survey and Interviews

Study Protocol. To compare refactoring practices, we adapted survey questions from previous studies [31, 50, 59, 125], following established guidelines [55]. We specifically selected these questions because they have been previously validated and deployed among *Non-Sci-OSS* practitioners. This selection aligns directly with our study's objective to establish a reliable baseline and compare the refactoring habits, motivations, and tool usage of *Sci-OSS* developers against the broader *Non-Sci-OSS* community. Table 6 lists the questions. To ensure data completeness, all survey questions were mandatory, preventing the submission of incomplete responses. To account for varying tool adoption, conditional questions (such as tool satisfaction) included an "N/A" (Not Applicable) option for respondents who do not use refactoring tools. A final survey question invited respondents to follow-up interviews, allowing deeper exploration of their refactoring activities and experiences. All participants who expressed interest were included; no additional selection or filtering based on survey responses was performed. For participants who consented, we conducted 30-60 minute semi-structured Zoom interviews. These interviews focused on discussing specific refactoring instances authored by the participants, where they were asked to explain the motivation, implementation decisions, and any domain-specific considerations underlying the changes. As a token of appreciation, each interviewee received a $10 gift card. The study was approved by the authors' institutional ethics boards. The full set of survey and interview questions is available in the replication package [92].

Recruiting Participants. Among the 616 participants identified in refactoring activities within *Sci-OSS*, 384 had public email addresses on their GitHub profiles and were invited to participate in our study. We also shared the survey on multiple forums where *Sci-SW* developers, also known as research software engineers (RSE) [43], actively discuss relevant topics. The survey received 47 responses in total, including 38 contributors from GitHub and nine RSEs recruited through the forums. Their responses did not reveal any notable differences, and no challenges were reported

Fig. 7. Refactoring frequency, years of coding experience and satisfaction with refactoring tools.



that were unique to RSEs. 14 participants agreed to participate in follow-up interviews. Since we posted the recruitment letter in online forums, we are unable to report the response rate for the survey.

5.2 Data Analysis

The first author transcribed all interview recordings and appended them to the corresponding survey responses. The first and second authors independently analyzed each transcript using a hybrid thematic approach that combines deductive and inductive coding [34]. Deductive coding leveraged themes from prior studies [31, 50, 59, 125] as initial seeds, while inductive coding identified additional themes emerging from the qualitative data. Our analysis follows reflexive thematic analysis [14, 15], adopting a pragmatic epistemological stance aimed at identifying patterns in scientists' reported experiences while acknowledging that themes are constructed through the researchers' interpretation of participant accounts [24]. This approach is consistent with prior works that both build on existing theoretical constructs and allow new insights to emerge from empirical observations [11, 36, 127].

To ensure consistency and reduce bias, coders held regular meetings to discuss emerging patterns and resolve disagreements. We assessed inter-rater reliability by comparing the labels assigned by each coder to the same transcript excerpts, yielding a Cohen's kappa coefficient of 0.64, indicating substantial agreement between coders [62]. Theoretical saturation was reached after analyzing 32 participants, as additional transcripts did not produce new inductive codes or themes during the iterative coding process.

5.3 Results

We found that refactoring definitions and motivations are consistent across *Sci-OSS* and *Non-Sci-OSS*, but challenges, risks, and opinions on existing tools differ notably. The complete codebook can be found in the replication package [92].

Participants' background. Among the 47 participants, 42 hold a PhD, including 36 specializing in scientific fields such as astronomy, biochemistry, or physics. 14 are currently employed as 'research software engineers' who actively maintain or develop *Sci-SW*, though they lack formal SE education, according to their survey response. The remaining 26 are researchers primarily focused on scientific investigations. Fig 7 summarizes coding experience and refactoring frequency in *Sci-OSS*. Most participants (85%) have over five years of coding experience, indicating a highly experienced cohort in *Sci-OSS* development. Refactoring practices vary: 37% perform refactoring monthly, 20% weekly, and 10% daily. In contrast, 33% rarely engage in refactoring.

5.3.1 ***Refactoring Definition and Motivation.*** All participants exhibit a clear and accurate understanding of refactoring, comparable to developers working on *Non-Sci-SW* [59]. Their motivations for refactoring also largely align with those identified by Kim et al. [59], reinforcing that while high-level refactoring concepts are broadly applicable,

Table 7. Challenges mentioned by surveyees for refactoring *Sci-OSS* repositories. Codes with citations are from prior works.

Code	Description	#P
Inadequate testing leads to unforeseen outcomes [59]	Lack of adequate tests practices and tooling support to prevent errors during refactoring.	35
Lacking recognition & funding	Refactoring is undervalued and not seen as significant as adding new features.	28
Domain knowledge gap	Challenge of learning the domain and understanding the code to make refactoring changes.	23
Hard to find code reviewers	Limited availability of knowledgable reviewers to ensure scientific correctness.	5
Adding complexity	Increasing the complexity of the existing code.	4
Merge conflicts [59]	Difficulties integrating refactored code while maintaining stability and compatibility.	3
No measurable outcome	Refactoring efforts lacking measurable outcome.	3
Learning costs	Time & effort required to learn refactored code.	3
Incompletion and delays due to its complexity	Incomplete refactoring efforts and delays due to complexities.	3

effective implementation requires domain knowledge, as reflected in responses to Q3 (challenges in refactoring *Sci-OSS*). This suggests that the difficulties *Sci-OSS* developers encounter are not due to a lack of awareness but rather the unique constraints and requirements of their domain. While many motivations overlap with *Non-Sci-OSS*, a distinct driver in *Sci-OSS* is the need to *adapt the software to evolving requirements due to the dynamic nature of research and technological advancements.* For instance, P5 noted, "*software evolution to support new, faster techniques and improve accuracy with new algorithms,*" emphasizing refactoring for new models or data formats. This pattern also emerged in our FN analysis in RQ1. In NASA-AMMOS/aerie (PR#683), refactoring addressed Issue#668 where "*the Sequence Editor it generates the old list notation rather than the new object notation,*" prompting updates for new data formats. We note that practitioners occasionally use the term "refactoring" to describe broader software evolution activities, such as introducing new algorithms or supporting new data formats [5, 59, 98]. While our study adopts the widely used definition of refactoring as behavior-preserving structural improvement, some of these developer-described changes may fall outside the strict definition typically used in software engineering research. This discrepancy suggests that in *Sci-SW* projects, refactoring may be intertwined with domain-driven improvements, which helps explain why certain changes identified by developers are difficult for existing refactoring detection tools to capture.

5.3.2 ***Challenges & Risks.*** In response to inquiries regarding the challenges and risks of refactoring in *Sci-OSS*, we observed substantial alignment with prior findings (i.e., the challenge of difficulty in merging changes and the risk of merge conflicts) [59]. Table 7 summarizes the challenges reported by participants when refactoring *Sci-OSS* repositories.

Inadequate testing leads to unforeseen outcomes was the most cited challenge, highlighting the difficulty of ensuring refactoring does not introduce errors. While also present in *Non-Sci-SW* [59], and is particularly problematic in *Sci-OSS*, this issue is critical in *Sci-OSS*, where scientific correctness is essential [12]. As P33 stated, "*working on implementing changes, when no tests are in place to verify the changes are not breaking functionality [is challenging]*," underscoring the lack of robust testing or tooling support in *Sci-OSS* [46]. Our qualitative analysis of FN cases (RQ1) reinforces this. For instance, a refactoring in NASA-AMMOS/aerie (PR#700) was reverted in PR#714 with the maintainer stating that "*We needed to revert because creating an expansion set fails.*" illustrating the risks of unverified changes.

This challenge is compounded by the **necessity to maintain scientific correctness**, a challenge not as prominent in *Non-Sci-OSS*. For instance, P5 highlighted the risk of a "*lack of scientific correctness once refactoring is complete.*" Our findings from FN cases in RQ1 further support this concern. For example, in (Astropy/astropy (PR#10814), an optimization was introduced to avoid redundant calls for built-in data types, **enhancing computational efficiency**.

However, domain experts cautioned that applying this optimization to non-built-in types **could reduce accuracy and increase processing time**. As a result, the final implementation retained both approaches, using an if-else structure: (1) a **performance-optimized version** for built-in data types and (2) a fallback to the **original implementation for cases requiring greater precision**. This demonstrates the trade-off in *Sci-OSS* between performance, accuracy, and compatibility during refactoring.

The **lack of recognition and funding for refactoring** was the second most frequently highlighted challenge in *Sci-OSS*, mentioned by 28 participants (59%). This reflects a broader issue in the academic environment, where refactoring and software development are often undervalued, failing to be seen as significant contributions to scientific progress [13, 69, 87, 100]. As P6 noted, grants typically support the creation of new software rather than the maintenance of existing software, making refactoring a low priority. P26 also highlighted the difficulty in justifying refactoring efforts as they do not directly produce new research findings.

Another challenge, **domain knowledge gap**, cited by 23 out of 47 participants (49%), also aligns with our RQ1 findings. Unlike *Non-Sci-OSS*, where specialized expertise is often unnecessary, *Sci-OSS* refactoring requires a deep understanding of the scientific domain. Our RQ1 results highlight its critical role in refactoring decisions. For example, in bioinformatics, refactoring a custom data type for genomic sequences in monarch-initiative/dipper (issue#193) involved updating it to a more efficient structure, improving memory efficiency and query performance. Such optimizations require domain knowledge to assess trade-offs in data representation. The interdisciplinary nature of *Sci-OSS* development further amplifies this challenge. As P12 noted, "*understanding the codebase and the scientific concepts is crucial, and this often requires significant time and effort.*" This underscores the dual expertise required in both SE and the underlying scientific domain, making the refactoring process particularly demanding.

Hard to find code reviewers was mentioned by five participants. This involves finding reviewers with the necessary domain knowledge and SE practice awareness. P16's response highlights this issue: Finding collaborators with enough experience on large code bases." This challenge impacts refactoring efforts, as finding reviewers with the necessary expertise is difficult, often leading to delays in reviewing refactoring changes. P32 highlighted this issue saying *"Response time for a refactoring code review relative to my working pace, where the project moves slower than I do,"* reflecting the challenge of aligning refactoring review timelines with development speed.

The risk of **adding complexity**, as mentioned by four participants. This involves increasing the complexity of the existing code, making it harder to understand and maintain. P3 noted, "*Adding more complexity and making my colleagues learn the new approach,*" reflecting the additional cognitive load imposed on the *Sci-OSS* development team.

Incompletion and delays due to complexity were also noted by three participants. These risks involve the time and effort required to learn new refactored code and the potential for incomplete refactoring efforts to leave the codebase in an unstable state. As P16 described this issue: "*Not finishing—for at least two reasons: (a) inability by the team to pull it off; (b) lack of management support.*"

Participants also highlighted other challenges including managing merge conflicts when integrating refactored code, an issue also prominent in *Non-Sci-OSS* [59]. Furthermore, they pointed out the lack of measurable outcomes of refactoring efforts, referring to the significant effort required for refactoring with no quantifiable outcome. The challenges reported by participants also highlight a disconnect between current refactoring detection approaches and the nature of refactoring in scientific software. Existing tools primarily rely on AST-based structural pattern matching to identify refactoring operations. However, many of the risks and challenges described by participants, such as maintaining scientific correctness, balancing numerical accuracy and performance, and understanding domain-specific algorithms, require domain expertise that cannot be captured through structural analysis alone. This helps explain the

Table 8. Opinions on refactoring tools by surveyees on refactoring tools for *Sci-OSS* repositories. Codes with citations are from prior works.

Categories	Features	Definition	#P
Exist but underutilized by *Sci-OSS* developers	Scoping for finding all references	List all references to code segments based on scope	9
	Code smell detection	Tools to identify and highlight code smells	1
	Type hinting	Tools to support the addition of type hints in code	3
Exist but not Applicable to *Sci-OSS*	Better testing tools [50]	Tools to support testing of refactored code	5
	Better dependency management tools	Tools to manage and resolve dependencies in the codebase	5
Do Not Exist	Domain-aware refactoring recommendations	Refactoring recommendations tailored for the specific needs and characteristics of *Sci-SW*	7

high proportion of unsupported refactorings observed in RQ1 and suggests that supporting refactoring in *Sci-OSS* may require tools that incorporate domain knowledge beyond syntactic transformations.

5.3.3 ***Opinions on Existing and Expected Refactoring Tools.*** We observed varying levels of tool adoption. Notably, 14 out of 47 respondents (30%) reported not using any refactoring tools at all, with P6 even stating, "*Unsure what these could be.*" Among users, 19 (40%) rely on IDE-provided utilities like find-and-replace, while six (13%) reported using AI-powered tools such as ChatGPT and Copilot. Additionally, eight (17%) employ specialized tools like SonarQube [114], Mypy [107], and Ruff [111] for specific refactoring needs.

Among the respondents who actively use refactoring tools (excluding the 14 users who selected "N/A"), participants expressed mixed satisfaction with existing refactoring tools: 25% reported being very satisfied, while 19% were satisfied, as shown in Fig. 7. However, over half of the respondents were either neutral (38%) or dissatisfied (18%). Regarding desired features to improve refactoring, participants identified eight distinct types, which we categorized into three groups: (1) existing but underutilized features, (2) features available but not applicable to *Sci-OSS*, and (3) features yet to be developed. Table 8 presents the features identified by participants as desirable. Our participants noted rarely using features like type hinting, code smell detection, and scoping for finding references, as well as inapplicable ones, including testing and dependency management tools. A key limitation of existing refactoring tools is the lack of **domain context awareness**, reinforcing the findings from RQ1. Seven participants emphasized the need for refactoring tools that maintain domain-context awareness to prevent errors and ensure correctness in scientific software development. This highlights the need for refactoring tools that maintain domain awareness for effective scientific software development.

Overall, the findings indicate that while *Sci-OSS* developers share needs with *Non-Sci-OSS*, such as testing and dependency management, they also require refactoring tools that maintain domain-context awareness. Addressing these needs requires tools that integrate general SE practices with *Sci-OSS*-specific demands.

6 Discussion and Implications

6.1 The Role of Domain Knowledge in Refactoring *Sci-OSS*

Our quantitative analysis of SOTA refactoring detection tools on *Sci-OSS* (RQ1) shows that these tools identify substantially fewer refactorings in *Sci-OSS* projects compared to *Non-Sci-OSS*, with an average detection rate of 25% versus 62%. We also find that 67.54% of the undetected refactorings in *Sci-OSS* require domain knowledge, indicating a strong association between the need for domain expertise and the limitations of current tools. While this does not establish a causal relationship, the quantitative results suggest that domain knowledge may be an important factor in why existing

tools underperform on scientific software. While our results suggest that the observed detection gap cannot be explained solely by differences in refactoring activity, and further work is needed to disentangle the relative contributions of tool limitations and domain-specific development practices.

Qualitative insights from our survey and interviews with 47 *Sci-OSS* developers (RQ2) reinforce this interpretation. Developers consistently highlighted that understanding domain-specific data and algorithms is critical for correctly performing refactorings, providing contextual evidence that domain knowledge is a key contributor to the challenges observed in tool-based detection. Note that although we examined software projects within the open-source context due to their public accessibility, the distinctions between *Sci-OSS* and *Non-Sci-OSS* are not necessarily confined to open-source environments. While qualitative analysis helps characterize these refactorings, automated detection tools enable us to quantify the extent to which current techniques fail to capture refactoring activity in *Sci-OSS* repositories. Although some of these undetected changes may fall outside the strict structural patterns currently supported by existing tools, our manual classification follows widely adopted definitions of refactoring as behavior-preserving improvements to internal code structure.

Taken together, the findings from our RQs suggest that the limited effectiveness of current refactoring detection tools in *Sci-OSS* is closely related to the domain knowledge required to perform and interpret many refactorings in scientific software. This highlights a broader gap in SE research, where domain context is often overlooked in tool development and empirical studies. The low detection rate observed in *Sci-OSS* suggests that existing tools may significantly under-report refactoring activity, particularly when refactorings rely on domain-specific reasoning. Consequently, studies or maintenance analyses that rely solely on automated detection may overlook important refactoring efforts performed in *Sci-SW* projects. Researchers and practitioners should therefore complement automated detection with manual inspection or domain-informed analysis when studying refactoring activity in scientific codebases.

Implications for *Sci-SW* practitioners. While our findings highlight limitations in current refactoring detection tools, they also carry practical implications for *Sci-SW* developers. The low detection rates should not be interpreted as an absence of refactoring activity, but rather as evidence that many refactorings in *Sci-OSS* depend on domain-specific reasoning that automated tools cannot capture. Consequently, relying solely on automated detection may lead to incomplete or misleading conclusions about code evolution. Consistent with our findings from RQ2, where practitioners report limited reliance on such tools, our results suggest that domain-informed manual inspection, code review discussions, and documentation of design decisions remain the primary and more reliable means of identifying and validating refactorings in scientific software. In practice, this means that scientists should prioritize domain-aware code review and discussion when assessing refactoring activity, rather than relying on automated detection outputs. While automated tools may support large-scale analysis or research settings, they currently provide limited direct value in day-to-day *Sci-SW* development.

6.2 Implications for Software Engineering Research

Our study highlights a broader gap in SE research: existing methodologies often assume that refactoring patterns are domain-agnostic, yet they have primarily evolved in easily available open-source codebases, such as Apache Java projects [97, 119]. However, recent discussions in prior works acknowledge the need to evaluate refactoring tools across more diverse and challenging benchmarks [3].

Sci-OSS developers operate within distinct contexts, including performance-sensitive computations on large datasets, strict requirements for reproducibility of scientific results, and complex, evolving algorithmic workflows that are tightly coupled with domain models. Apache projects, while large and widely studied, typically focus on general-purpose

libraries, web services, or utilities where correctness is defined by well-specified API contracts rather than domain-specific models or data constraints. As a result, patterns of code reuse, dependencies on domain-specific utilities, and decisions about maintaining scientific validity are largely absent from such datasets, limiting the generalizability of refactoring studies based on them. Addressing these contexts requires further research to refine refactoring practices, develop benchmarks that incorporate domain-specific considerations, and explore challenges unique to *Sci-SW*.

Expanding and Refining Refactoring Taxonomies: Our study expanded existing refactoring taxonomies by incorporating refactoring types identified in our study and those from Tang et al.'s work on ML systems [103], building upon Fowler's catalog [37]. These findings suggest that refactoring taxonomies should be continuously updated to accommodate emerging patterns. A promising direction for future work is to systematically validate and refine the proposed taxonomy of unsupported and implicitly supported refactorings through direct collaboration with domain experts across diverse scientific disciplines. A potential approach to achieving this is leveraging a crowdsourcing method to systematically collect and validate unsupported instances.

Furthermore, future research could extend our work by analyzing a broader range of refactorings in *Sci-OSS* to determine whether domain-specific refactoring patterns emerge. Establishing a refined taxonomy tailored to scientific computing could help bridge existing gaps by categorizing refactorings based on key factors such as numerical stability, high-performance computing and parallelization strategies, and maintainability concerns unique to this domain. For example, a frequent challenge in scientific computing is porting code to new hardware platforms, such as transitioning from Cray to Intel architectures or adapting GPU-accelerated code from AMD to Nvidia [26, 118]. These transitions often require refactorings that address hardware-specific optimizations, memory access patterns, and parallel execution models. Identifying and categorizing such maintenance challenges could inform the development of refactoring strategies that facilitate efficient and correct migration across evolving computing architectures. Such a taxonomy could also enhance the precision of refactoring detection and recommendation, ultimately supporting more effective software evolution in scientific computing.

Refactoring Evaluation Benchmarks: The absence of benchmarks incorporating real-world *Sci-OSS* projects constrains the validation of refactoring tools in SE research. Our study serves as an initial step toward establishing benchmark suites that encompass representative codebases from diverse scientific domains, providing a more realistic foundation for evaluating refactoring techniques. Such benchmarks should account for domain-specific constraints, including the trade-offs between accuracy and computational efficiency, to ensure that proposed refactorings preserve scientific validity. Future research could extend this effort by constructing a comprehensive benchmark tailored to domain-specific *Sci-OSS* projects.

6.3 Implications for Refactoring Tool Development

Refactoring detection and recommendation tools should be explicitly tailored to the challenges of *Sci-OSS*. Unlike *Non-Sci-OSS*, refactoring in scientific applications involves domain-specific considerations such as preserving computational correctness, managing specialized data structures, or maintaining reproducibility that may not be present or as critical in web-centric or other mainstream software for which most tools are developed. Our survey results indicate that most existing refactoring tools used by practitioners are not tailored for *Sci-OSS*, limiting their effectiveness in scientific computing. This highlights a potential mismatch between current tools and the requirements of the scientific software community, motivating the development of domain-aware refactoring support.

Refactoring detection and recommendation tools should account for the unique challenges of *Sci-OSS*. Unlike *Non-Sci-OSS*, refactoring in scientific applications involves domain-specific considerations such as preserving computational

correctness, managing specialized data structures, or maintaining reproducibility that may not be present or as critical in web-centric or other mainstream software for which most tools are developed. Our findings also suggest that developers in *Sci-SW* projects may conceptualize refactoring more broadly than the strict structural definition typically assumed by existing detection tools. To better support *Sci-SW* maintenance, future refactoring detection approaches should incorporate domain-aware reasoning and contextual information (e.g., commit messages, issue discussions, or domain-specific documentations) rather than relying solely on structural code patterns.

None of the interviewed developers reported using the SOTA refactoring tools we evaluated quantitatively (i.e., RefDiff, RefactoringMiner, and PyRef), suggesting that these tools are either unfamiliar to *Sci-OSS* practitioners or not well-integrated into their workflows. As developers are inherently aware of the refactorings they apply, their primary requirements lean toward refactoring recommendation and automation rather than post-hoc detection. The lack of adoption of these mining tools in daily workflows is likely because they do not directly serve the needs of *Sci-OSS* practitioners. Nevertheless, accurate refactoring detection remains important for research purposes, as it enables the construction of reliable datasets that help us understand refactoring practices and support the development of improved refactoring recommendation tools. Our findings therefore highlight an opportunity for future tool support that better aligns with the priorities of *Sci-SW* developers, such as maintaining numerical correctness, managing performance trade-offs, and preserving experimental reproducibility. A domain-specific approach could improve refactoring detection accuracy, enhancing software maintenance in this context. As demonstrated in (Astropy/astropy (PR#10814)), and detailed in Sec. 5.3.1, refactoring tasks must navigate competing quality requirements, including performance, accuracy, and compatibility. Future tools could be more effective by integrating domain knowledge, supporting trade-off analysis, and providing recommendations aligned with *Sci-OSS* development practices.

To facilitate the development of refactoring tools for *Sci-OSS*, we identify four key areas for improvement:

Incorporating Domain-Specific Heuristics into Refactoring Analysis and Detection: Existing refactoring tools predominantly rely on well-defined code smells and structural patterns [119], which may not adequately capture the complexities of *Sci-OSS*. Given the unique structural and computational demands, refactoring tools should incorporate domain-specific heuristics that account for the conventions and constraints of numerical methods, data structures, and scientific computing libraries. While refactoring and performance optimization are conceptually distinct activities, in *Sci-SW* they are often closely related, as refactoring is frequently performed to enable or safely apply performance improvements. For instance, many scientific frameworks provide optimized numerical routines and specialized data structures as best practices designed to balance performance and precision (e.g., Astropy [9] for astronomy and Scikit-bio [91] for bioinformatics). Refactoring tools that recognize these domain-specific optimizations can identify inefficient patterns and suggest transformations that enable or support performance improvements, while ensuring numerical accuracy and reproducibility. Furthermore, awareness of library-specific constraints, such as precision handling, unit consistency, and API stability, can help prevent refactorings that inadvertently alter computational semantics or introduce unintended errors.

Additionally, grouping *Sci-OSS* based on their dependency relationships may enhance code reuse by identifying common refactoring patterns. This aligns with the refactoring patterns observed in RQ1, such as replacing custom functions with built-in or dependency-provided alternatives. A complementary approach would be to analyze projects within the same scientific domain collectively and extract best practices that can be systematically recommended to newly developed projects and communities. This approach could help refactoring tools strike a balance between generalizability across a domain and specificity to individual *Sci-OSS* projects, ensuring that refactoring recommendations remain both broadly applicable and practically relevant. Moreover, given that manual refactoring can be error-prone

and time-consuming, integrating these automated recommendations into scientific development environments (e.g., Computational Notebooks, IDEs) could encourage incremental improvements and usability of these techniques.

Enhancing Refactoring Detection with AI: To better incorporate domain knowledge into refactoring tools, future work could leverage machine learning (ML), natural language processing (NLP), and large language model (LLM) techniques. A hybrid approach that integrates static analysis with ML models trained on domain-specific refactoring patterns, alongside NLP techniques that can extract knowledge from theoretical foundations, textbooks, and tutorials in a certain domain, could improve refactoring detection accuracy. For example, popular scientific libraries provide tutorials to simplify onboarding [104, 109], and fine-tuning existing models on domain-specific articles could enhance the effectiveness of refactoring detection and recommendation tools.

Furthermore, the recent success of AI for Science in various domains such as computational biology [54], astronomy [79], and healthcare [2] highlights their potential to advance both scientific research and software development by automating knowledge extraction, enhancing code comprehension, and generating domain-aware refactoring suggestions tailored to the needs of researchers and practitioners. However, Existing LLM-based code assistance tools faces key limitations in scientific domains. For instance, automated fixes using SWE-Agent for Astropy in the SWE-bench dataset show a lower success rate compared to other *Non-Sci-OSS* [52, 129], underscoring the need for models with domain knowledge. Additionally, concerns over accountability and trustworthiness in scientific discovery [130], along with challenges in metamorphic testing [121], raise further concerns about reliability of LLMs.

To address these challenges, future work could explore integrating statistical and rule-based methods with domain-aware ML techniques, ensuring that refactoring tools not only improve *Sci-SW* development but also maintain reliability, interpretability, and trustworthiness.

Testing Support for Safe Refactoring: A major barrier shown in our results to refactoring in *Sci-OSS* is the lack of comprehensive automated testing, which makes *Sci-OSS* developers hesitant to apply refactoring changes that could introduce unintended errors. Unlike *Non-Sci-OSS*, where unit testing is a widely adopted practice, many scientific software applications rely on ad-hoc verification strategies, manual validation, or computational experiments for correctness assurance [56]. This reliance increases the cost of refactoring, as developers must manually verify that modifications do not alter expected scientific outcomes or run expensive end-to-end simulation tests.

The lack of adequate testing directly impacts the safety of refactoring, as insufficient test coverage makes it difficult to detect unintended changes [46]. Our study reinforces this known issue by highlighting how the absence of automated testing discourages *Sci-OSS* developers from performing refactoring, ultimately hindering code maintainability. To address this, refactoring tools could integrate mechanisms for automated verification and regression testing [82], leveraging domain-aware test generation techniques. For example, tools could automatically extract and rerun representative computational experiments as regression tests, ensuring that refactorings preserve numerical accuracy and scientific validity. Additionally, incorporating property-based testing [66] and symbolic execution [93] techniques tailored to scientific code could further improve confidence in refactoring recommendations.

Collaboration with Domain Experts for Better Knowledge Sharing: The effectiveness of refactoring tools in *Sci-SW* development hinges on their ability to align with developers' expectations and workflows. Our qualitative findings, show that *Sci-OSS* refactoring is often motivated by the need to support new models, algorithms, or data formats, rather than general code cleanup, an objective that is closely tied with domain knowledge. Engaging directly with domain experts can help refine refactoring strategies, ensuring that suggested refactorings align with these objectives and support key requirements of *Sci-SW* such as performance, reproducibility, and maintainability. This engagement can be fostered through participatory design workshops and hackathons [81], which facilitate structured interactions

between researchers and practitioners. Additionally, distributed online programs, such as Google Summer of Code [116], provide opportunities for sustained collaboration and knowledge exchange across interdisciplinary teams.

Moreover, iterative user feedback loops and seamless integration with existing *Sci-SW* development environments can further enhance the adoption and effectiveness of refactoring tools and promote their usage in *Sci-OSS*. Community-led initiatives such as the US Research Software Engineer Association (US-RSE) [123], the Society of Research Software Engineering [122], and the International Council of Research Software Engineering Associations [49] play a crucial role in advancing best practices for *Sci-SW* development by fostering communication, collaboration, and resource-sharing among *Sci-SW* engineers. While not a developer-led community, the ADORE initiative [1] represents an important policy effort by funding agencies to recognize and support the sustainability of scientific software. Together, these initiatives not only drive the evolution of *Sci-SW* but also advocate for the recognition and professionalization of scientific software engineering as a discipline.

6.4 Implications for *Sci-OSS* Development Practices

The reliance on domain expertise for refactoring suggests that *Sci-SW* developers may face challenges in adopting SE best practices without appropriate support. Our survey results indicate that *Sci-OSS* developers often prioritize **scientific correctness over maintainability**, leading to delayed or avoided refactoring efforts. This trade-off reflects the need to preserve domain-specific assumptions, numerical accuracy, and experimental validity, even when code quality may degrade over time. To address this challenge, we recommend:

Educational Resources on Refactoring for *Sci-OSS*: Many *Sci-OSS* developers are self-taught programmers with limited formal training in software engineering. Consequently, they may not be aware of the benefits of systematic refactoring or how to apply refactoring techniques effectively. While existing resources, such as Software Carpentry [126] do introduce basic refactoring principles to bridge the existing knowledge gap and promote sustainable software practices in *Sci-OSS*, they typically focus on a small subset of general-purpose refactorings. For example, the Software Carpentry material covers eight types, six of which are already captured in our coding scheme, while the remaining two (e.g., using in-place operators or replacing code with data structures) are broad stylistic guidelines that do not address the domain-specific or novel refactorings identified in this study. This highlights the need for tailored educational resources that reflect the complexities of scientific software. Developing tutorials, interactive examples, and domain-specific case studies could help bridge this gap [83]. However, creating such educational resources requires significant time and effort, which may divert scientists from their primary research activities. To mitigate this burden, LLM and ML techniques could be leveraged to automate the generation of high-quality training materials for *Sci-OSS* developers. For instance, models fine-tuned on refactoring best practices and scientific computing patterns could generate domain-specific refactoring suggestions, code explanations, and interactive learning modules. Such AI-driven approaches have been explored for automatically generating instructional materials, including tutorials derived from video content (e.g., NotePlayer [76]), suggesting their potential to scale training efforts without imposing additional workload on researchers.

Guidelines for Safe Refactoring in *Sci-OSS*: Unlike general-purpose software, scientific codebases often embed domain-specific assumptions, numerical approximations, and computational constraints that must be preserved during refactoring as shown in our results. Establishing best practices for safe refactoring, such as defining test-driven refactoring workflows, identifying high-risk refactorings, and ensuring reproducibility through automated verification, can help *Sci-OSS* developers make incremental improvements without introducing unintended changes. These guidelines should be co-developed with domain experts to align with scientific requirements while promoting software sustainability.

Future work could explore mechanisms to enhance developer awareness of refactoring decisions and provide comparative insights into alternative refactoring strategies. For instance, assessing the impact of refactoring on quality attributes of interest by leveraging learning from prior code review discussions available on GitHub could help prevent unnecessary back-and-forth discussions, code modifications and inform best practices.

7 Threats to validity

Internal validity may be influenced by coder bias during the manual labeling process in both deductive and inductive coding. While a high Cohen's kappa score indicates strong inter-rater agreement, the inherent subjectivity of labeling may still introduce some degree of error or discordance in the assigned refactoring labels. Regarding the survey and interviews in RQ2, several participant-related biases may exist. *Response and acquiescence bias* could occur if participants provided answers they perceived as socially desirable or aligned with the researchers' expectations. We addressed this by ensuring anonymity and emphasizing that there were no "correct" answers. Furthermore, the use of *restrictive question formats* in the survey might have limited the nuances of developer responses; however, we mitigated this threat by including open-ended questions and conducting follow-up interviews to capture richer qualitative context.

External Validity is primarily constrained by sample size and composition. While we reached thematic saturation with 32 participants, the findings may not fully generalize to the entire population of *Sci-OSS* developers or all refactoring contexts. A significant threat is *self-selection bias*, as developers who are more passionate about code quality or refactoring may have been more likely to volunteer for our study, potentially skewing results toward more disciplined practices.

A key threat to *construct validity* is the potential misalignment between our operational definition of refactoring and how *Sci-OSS* developers perceive it. If our definition fails to capture real-world behaviors, the findings' applicability is compromised. We addressed this by incorporating developer perspectives in RQ2 to refine our study's alignment with actual practices. Additionally, our reliance on explicit labeling to establish a reliable ground truth may overlook unlabeled refactoring activities. Future work should explore automated techniques to identify these latent refactoring instances.

8 Conclusion

In this study, we investigated the limitations of SOTA refactoring detection tools when applied to *Sci-OSS*, revealing significant differences in their effectiveness compared to *Non-Sci-OSS*. Through a mixed-method approach, we highlighted the crucial role of domain knowledge in *Sci-OSS* refactoring, which remains unsupported by existing tools. Our findings emphasize the unique challenges faced by *Sci-OSS* developers, including the difficulty of automated refactoring detection, the necessity of scientific correctness, and the lack of tailored tooling. The interdisciplinary nature of *Sci-OSS* development, coupled with inadequate testing practices and evolving scientific requirements, exacerbates refactoring challenges. These insights call for the development of specialized refactoring approaches and domain-aware tool support to bridge the gap between *Sci-OSS* and SE best practices.

Acknowledgments

This work was partially supported by the Alfred P. Sloan Foundation (Grants No. G-2022-19472 and G-2022-19443). We thank the *Sci-SW* developers and research software engineers who contributed their time and expertise by participating in our survey and interviews. We would also like to express our heartfelt gratitude to all members of the Forcolab for their invaluable feedback and moral support. Finally, we thank the anonymous reviewers for their thoughtful comments and constructive suggestions. Their feedback provided valuable insights and gave us the opportunity to revise the manuscript, improving the clarity, rigor, and significance of this study.

References

[1] Adore 2024. The Amsterdam Declaration on Funding Research Software Sustainability. https://adore.software/

[2] Mohammed Al-Garadi, Tushar Mungle, Abdulaziz Ahmed, Abeed Sarker, Zhuqi Miao, and Michael E. Matheny. 2025. Large Language Models in Healthcare. arXiv:2503.04748 [cs.CY]

[3] Pouria Alikhanifard and Nikolaos Tsantalis. 2025. A Novel Refactoring and Semantic Aware Abstract Syntax Tree Differencing Tool and a Benchmark for Evaluating the Accuracy of Diff Tools. *ACM Trans. Softw. Eng. Methodol.* 34, 2, Article 40 (Jan. 2025), 63 pages.

[4] Eman Abdullah AlOmar, Hussein AlRubaye, Mohamed Wiem Mkaouer, Ali Ouni, and Marouane Kessentini. 2021. Refactoring Practices in the Context of Modern Code Review: An Industrial Case Study at Xerox. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, Madrid, ES, 348–357.

[5] Eman Abdullah AlOmar, Jiaqian Liu, Kenneth Addo, Mohamed Wiem Mkaouer, Christian Newman, Ali Ouni, and Zhe Yu. 2022. On the Documentation of Refactoring Types. *Automated Software Engineering* 29, 1 (May 2022).

[6] Everton L. G. Alves, Myoungkyu Song, and Miryung Kim. 2014. RefDistiller: A Refactoring Aware Code Review Tool for Inspecting Manual Refactoring Edits. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 751–754.

[7] Manoel Aranda, Naelson Oliveira, Elvys Soares, Márcio Ribeiro, Davi Romão, Ullyanne Patriota, Rohit Gheyi, Emerson Souza, and Ivan Machado. 2024. A Catalog of Transformations to Remove Smells From Natural Language Tests. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering* (Salerno, Italy) *(EASE '24)*. Association for Computing Machinery, New York, NY, USA, 7–16.

[8] Elvira-Maria Arvanitou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Jeffrey C. Carver. 2021. Software Engineering Practices for Scientific Software Development: A Systematic Mapping Study. *Journal of Systems and Software* 172 (Feb. 2021), 110848.

[9] Astropy Project. 2025. Astropy: Astronomy and Astrophysics Core Library. https://github.com/astropy/astropy

[10] Hassan Atwi, Bin Lin, Nikolaos Tsantalis, Yutaro Kashiwa, Yasutaka Kamei, Naoyasu Ubayashi, Gabriele Bavota, and Michele Lanza. 2021. PYREF: Refactoring Detection in Python Projects. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, Luxembourg, 136–141.

[11] Agathe Balayn, Mireia Yurrita, Fanny Rancourt, Fabio Casati, and Ujwal Gadiraju. 2025. Unpacking Trust Dynamics in the LLM Supply Chain: An Empirical Exploration to Foster Trustworthy LLM Production & Use. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems (CHI '25)*. Association for Computing Machinery, New York, NY, USA, Article 1103, 20 pages.

[12] Roscoe A Bartlett, Joseph R Frye, and Evan C Harvey. 2020. *Improved Productivity Through Standardized Configurations and Testing of Trilinos on Advanced Platforms.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[13] B. Bozhanov. 2010. The Low Quality of Scientific Code. https://techblog.bozho.net/the-astonishingly-low-quality-of-scientific-code/

[14] Virginia Braun and Victoria Clarke. 2006. Using Thematic Analysis in Psychology. *Qualitative Research in Psychology* 3, 2 (Jan. 2006), 77–101.

[15] Virginia Braun, Victoria Clarke, Nikki Hayfield, and Gareth Terry. 2019. Thematic Analysis. In *Handbook of Research Methods in Health Social Sciences*, Pranee Liamputtong (Ed.). Springer, Singapore, 843–860.

[16] G. Ann Campbell. 2018. Cognitive complexity: an overview and evaluation *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 57–58.

[17] 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 (Jan. 2013), 7–11.

[18] Alyssia Chen, Carol Wong, Bonita Sharif, and Anthony Peruma. 2025. Exploring Code Comprehension in Scientific Programming: Preliminary Insights from Research Scientists. In *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. 350–354.

[19] Norman Cliff. 1993. Dominance Statistics: Ordinal Analyses to Answer Ordinal Questions. *Psychological Bulletin* 114, 3 (Nov. 1993), 494–509.

[20] Flavia Coelho, Tiago Massoni, and Everton L.G. Alves. 2019. Refactoring-Aware Code Review: A Systematic Mapping Study. In *2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR)*. 63–66.

[21] Flávia Coelho, Nikolaos Tsantalis, Tiago Massoni, and Everton L. G. Alves. 2021. An Empirical Study on Refactoring-Inducing Pull Requests. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) (ESEM '21)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[22] PlasmaPy Community. 2026. PlasmaPy. https://github.com/PlasmaPy/PlasmaPy

[23] Carlton A. Crabtree, A. Gunes Koru, Carolyn Seaman, and Hakan Erdogmus. 2009. An Empirical Characterization of Scientific Software Development Projects According to the Boehm and Turner Model: A Progress Report. In *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*. 22–27.

[24] J.W. Creswell and J.D. Creswell. 2022. *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches.* SAGE Publications.

[25] Ward Cunningham. 1992. The WyCash Portfolio Management System. *SIGPLAN OOPS Mess.* 4, 2 (Dec. 1992), 29–30.

[26] Joshua Hoke Davis, Pranav Sivaraman, Joy Kitson, Konstantinos Parasyris, Harshitha Menon, Isaac Minn, Giorgis Georgakoudis, and Abhinav Bhatele. 2025. Taking GPU Programming Models to Task for Performance Portability. In *Proceedings of the 39th ACM International Conference on Supercomputing (ICS '25)*. Association for Computing Machinery, New York, NY, USA, 776–791.

[27] Danny Dig, Can Comertoglu, Darko Marinov, and Ralph Johnson. 2006. Automated Detection of Refactorings in Evolving Components. In *ECOOP 2006 – Object-Oriented Programming*, Dave Thomas (Ed.). Springer, Berlin, Heidelberg, 404–428.

[28] Malinda Dilhara. 2021. Discovering Repetitive Code Changes in ML Systems. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1683–1685.

[29] Chunhao Dong, Yanjie Jiang, Nan Niu, Yuxia Zhang, and Hui Liu. 2024. Context-Aware Name Recommendation for Field Renaming. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13.

[30] Steve M. Easterbrook and Timothy C. Johns. 2009. Engineering the Software for Understanding Climate Change. *Computing in Science & Engineering* 11, 6 (Nov. 2009), 65–74.

[31] Anna Maria Eilertsen and Gail C. Murphy. 2021. The Usability (or Not) of Refactoring Tools. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 237–248.

[32] Nasir U. Eisty and Jeffrey C. Carver. 2022. Developers Perception of Peer Code Review in Research Software Development. *Empirical Software Engineering* 27, 1 (Jan. 2022), 13.

[33] Neil Ernst, Rick Kazman, and Julien Delange. 2021. *Technical Debt in Practice: How to Find It and Fix It.* The MIT Press.

[34] Jennifer Fereday and Eimear Muir-Cochrane. 2006. Demonstrating Rigor Using Thematic Analysis: A Hybrid Approach of Inductive and Deductive Coding and Theme Development. *International Journal of Qualitative Methods* 5, 1 (March 2006), 80–92.

[35] SORMAS Foundation. 2025. SORMAS (Surveillance Outbreak Response Management and Analysis System). https://github.com/SORMAS-Foundation/SORMAS-Project

[36] Armstrong Foundjem, Ellis Eghan, and Bram Adams. 2021. Onboarding vs. Diversity, Productivity and Quality — Empirical Study of the OpenStack Ecosystem. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 1033–1045.

[37] Martin Fowler. 2018. *Refactoring: Improving the Design of Existing Code, 2nd Edition* (2nd ed.). Addison-Wesley Professional. Part of the Addison-Wesley Signature Series (Fowler) series.

[38] GitHub. 2022. GitHub REST API documentation. https://docs.github.com/en/rest?apiVersion=2022-11-28

[39] GitHub. 2022. List Repository Languages. https://docs.github.com/en/rest/repos/repos?apiVersion=2022-11-28#list-repository-languages

[40] GitHub. 2022. REST API Endpoints for Commits. https://docs.github.com/en/rest/commits/commits?apiVersion=2022-11-28

[41] GitHub. 2022. REST API Endpoints for Timeline Events. https://docs.github.com/en/rest/issues/timeline?apiVersion=2022-11-28

[42] GitHub. 2024. Managing labels in GitHub. https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/managing-labels

[43] Florian Goth, Renato Alves, Matthias Braun, Leyla Jael Castro, Gerasimos Chourdakis, Simon Christ, Jeremy Cohen, Stephan Druskat, Fredo Erxleben, Jean-Noël Grad, Magnus Hagdorn, Toby Hodges, Guido Juckeland, Dominic Kempf, Anna-Lena Lamprecht, Jan Linxweiler, Frank Löffler, Michele Martone, Moritz Schwarzmeier, Heidi Seibold, Jan Philipp Thiele, Harald von Waldow, and Samantha Wittke. 2024. Foundational Competencies and Responsibilities of a Research Software Engineer: Current State and Suggestions for Future Directions. *F1000Research* 13 (2024), 1429.

[44] 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*. IEEE, Vancouver, BC, Canada, 1–8.

[45] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. 2020. Array Programming with NumPy. *Nature* 585, 7825 (Sept. 2020), 357–362.

[46] Dustin Heaton and Jeffrey C. Carver. 2015. Claims about the Use of Software Engineering Practices in Science: A Systematic Literature Review. *Information and Software Technology* 67 (Nov. 2015), 207–219.

[47] James Howison and James D. Herbsleb. 2011. Scientific Software Production: Incentives and Collaboration. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work - CSCW '11*. ACM Press, Hangzhou, China, 513.

[48] Xing Huang, Xianghua Ding, Charlotte P. Lee, Tun Lu, and Ning Gu. 2013. Meanings and Boundaries of Scientific Software Sharing. In *Proceedings of the 2013 Conference on Computer Supported Cooperative Work*. ACM, San Antonio Texas USA, 423–434.

[49] INTER-RSE 2025. The International Council of RSE Associations. https://researchsoftware.org/council.html

[50] James Ivers, Robert L. Nord, Ipek Ozkaya, Chris Seifried, Christopher S. Timperley, and Marouane Kessentini. 2022. Industry Experiences with Large-Scale Refactoring. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1544–1554.

[51] James Ivers, Ipek Ozkaya, Robert L. Nord, and Chris Seifried. 2020. Next Generation Automated Software Evolution Refactoring at Scale. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1521–1524.

[52] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World Github Issues?. In *International Conference on Learning Representations*, B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun (Eds.), Vol. 2024. 54107–54157.

[53] Marina Jirotka, Charlotte P. Lee, and Gary M. Olson. 2013. Supporting Scientific Collaboration: Methods, Tools and Concepts. *Computer Supported Cooperative Work (CSCW)* 22, 4-6 (Aug. 2013), 667–715.

[54] John Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A. A. Kohl, Andrew J. Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly Accurate Protein Structure Prediction With Alphafold. *Nature* 596, 7873 (July 2021), 583–589.

[55] Marcos Kalinowski, Allysson Allex Araújo, and Daniel Mendez. 2024. Teaching Survey Research in Software Engineering. In *Handbook on Teaching Empirical Software Engineering*, Daniel Mendez, Paris Avgeriou, Marcos Kalinowski, and Nauman Bin Ali (Eds.). Springer Nature Switzerland, Cham, 501–527.

[56] Upulee Kanewala and James M. Bieman. 2014. Testing Scientific Software: A Systematic Literature Review. *Information and Software Technology* 56, 10 (Oct. 2014), 1219–1232.

[57] Diane Kelly. 2015. Scientific Software Development Viewed as Knowledge Acquisition: Towards Understanding the Development of Risk-Averse Scientific Software. *Journal of Systems and Software* 109 (Nov. 2015), 50–61.

[58] Sarah Killcoyne and John Boyle. 2009. Managing Chaos: Lessons Learned Developing Software in the Life Sciences. *Computing in Science and Engg.* 11, 6 (Nov. 2009), 20–29.

[59] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, Cary North Carolina, 1–11.

[60] Bonan Kou, Shengmai Chen, Zhijie Wang, Lei Ma, and Tianyi Zhang. 2024. Do Large Language Models Pay Similar Attention Like Human Programmers When Generating Code? *Proc. ACM Softw. Eng.* 1, FSE, Article 100 (July 2024), 24 pages.

[61] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software* 29, 6 (Nov. 2012), 18–21.

[62] J. Richard Landis and Gary G. Koch. 1977. The Measurement of Observer Agreement for Categorical Data. *Biometrics* 33, 1 (March 1977), 159.

[63] Dorian Leroy, June Sallou, Johann Bourcier, and Benoit Combemale. 2022. On the Role of Computer Languages in Scientific Computing. *Computing in Science & Engineering* 24, 04 (July 2022), 55–59.

[64] Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza. 2019. On the Impact of Refactoring Operations on Code Naturalness. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 594–598.

[65] Yuxing Ma, Tapajit Dey, Chris Bogart, Sadika Amreen, Marat Valiev, Adam Tutko, David Kennard, Russell Zaretzki, and Audris Mockus. 2021. World of Code: Enabling a Research Workflow for Mining and Analyzing the Universe of Open Source VCS Data. *Empirical Software Engineering* 26 (2021), 1–42.

[66] 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 (Nov. 2019), 1891.

[67] H. B. Mann and D. R. Whitney. 1947. On a Test of Whether One of Two Random Variables Is Stochastically Larger than the Other. *The Annals of Mathematical Statistics* 18, 1 (March 1947), 50–60.

[68] T. Mens and T. Tourwe. 2004. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30, 2 (Feb. 2004), 126–139.

[69] Zeeya Merali. 2010. Computational Science: ...Error. *Nature* 467, 7317 (Oct. 2010), 775–777.

[70] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. 2018. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering* 44, 12 (Dec. 2018), 1176–1206.

[71] Emerson Murphy-Hill and Andrew P. Black. 2008. Refactoring Tools: Fitness for Purpose. *IEEE Software* 25, 5 (Sept. 2008), 38–44.

[72] Emerson Murphy-Hill, Andrew P. Black, Danny Dig, and Chris Parnin. 2008. Gathering Refactoring Data: A Comparison of Four Methods. In *Proceedings of the 2nd Workshop on Refactoring Tools (WRT '08)*. Association for Computing Machinery, New York, NY, USA, 1–5.

[73] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *2009 IEEE 31st International Conference on Software Engineering*. 287–297.

[74] Nicholas Alexandre Nagy and Rabe Abdalkareem. 2022. On the Co-Occurrence of Refactoring of Test and Source Code. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 122–126.

[75] Priti Oli, Rabin Banjade, Lasang Jimba Tamang, and Vasile Rus. 2021. Automated Assessment of Quality of Jupyter Notebooks Using Artificial Intelligence and Big Code. *The International FLAIRS Conference Proceedings* 34, 1 (April 2021).

[76] Yang Ouyang, Leixian Shen, Yun Wang, and Quan Li. 2024. NotePlayer: Engaging Computational Notebooks for Dynamic Presentation of Analytical Processes. In *Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology (UIST '24)*. ACM, 1–20.

[77] OWID. 2024. COVID-19 Dataset by Our World in Data. https://github.com/owid/covid-19-data

[78] Indranil Palit, Gautam Shetty, Hera Arif, and Tushar Sharma. 2023. Automatic Refactoring Candidate Identification Leveraging Effective Code Representation. In *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 369–374.

[79] Rui Pan, Tuan Dung Nguyen, Hardik Arora, Alberto Accomazzi, Tirthankar Ghosal, and Yuan-Sen Ting. 2024. AstroMLab 2: AstroLLaMA-2-70B Model and Benchmarking Specialised LLMs for Astronomy . In *SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, Los Alamitos, CA, USA, 87–96.

[80] Jevgenija Pantiuchina, Fiorella Zampetti, Simone Scalabrino, Valentina Piantadosi, Rocco Oliveto, Gabriele Bavota, and Massimiliano Di Penta. 2020. Why Developers Refactor Source Code: A Mining-based Study. *ACM Transactions on Software Engineering and Methodology* 29, 4 (Oct. 2020), 1–30.

[81] Ei Pa Pa Pe-Than and James D Herbsleb. 2019. Understanding hackathons for science: Collaboration, affordances, and outcomes. In *International Conference on Information*. Springer, 27–37.

[82] Zedong Peng, Upulee Kanewala, and Nan Niu. 2021. Contextual Understanding and Improvement of Metamorphic Testing in Scientific Software Development. In *Proceedings of the 15th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–6.

[83] Elizaveta Pertseva, Melinda Chang, Ulia Zaman, and Michael Coblenz. 2024. A Theory of Scientific Programming Efficacy. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. ACM, 1–12.

[84] Anthony Peruma. 2019. A Preliminary Study of Android Refactorings. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 148–149.

[85] Anthony Peruma, Eman Abdullah AlOmar, Christian D. Newman, Mohamed Wiem Mkaouer, and Ali Ouni. 2022. Refactoring Debt: Myth or Reality? An Exploratory Study on the Relationship between Technical Debt and Refactoring. In *Proceedings of the 19th International Conference on Mining Software Repositories (MSR '22)*. Association for Computing Machinery, New York, NY, USA, 127–131.

[86] Gustavo Pinto, Igor Wiese, and Luiz Felipe Dias. 2018. How Do Scientists Develop Scientific Software? An External Replication. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Campobasso, 582–591.

[87] Joe Pitt-Francis, Miguel O. Bernabeu, Jonathan Cooper, Alan Garny, Lee Momtahan, James Osborne, Pras Pathmanathan, Blanca Rodriguez, Jonathan P. Whiteley, and David J. Gavaghan. 2008. Chaste: Using Agile Programming Techniques to Develop Computational Biology Software. *Philosophical Transactions. Series A, Mathematical, Physical, and Engineering Sciences* 366, 1878 (Sept. 2008), 3111–3136.

[88] 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 (SC '11)*. Association for Computing Machinery, New York, NY, USA, 1–12.

[89] Kyle Prete, Napol Rachatasumrit, Nikita Sudan, and Miryung Kim. 2010. Template-Based Reconstruction of Complex Refactorings. In *2010 IEEE International Conference on Software Maintenance*. 1–10.

[90] RefDiff Project. 2025. RefDiff2.0. https://github.com/aserg-ufmg/RefDiff

[91] Scikit-Bio Project. 2025. Scikit-Bio: A Community-Driven Python Library for Bioinformatics, Providing Versatile Data Structures, Algorithms and Educational Resources. https://github.com/scikit-bio/scikit-bio

[92] Rohith Pudari, Ahmed Musa Awon, Neil Ernst, and Zhou Shurui. 2025. When Tools Overlook Domain Knowledge: An Empirical Study of Refactoring in Scientific Software. https://doi.org/10.5281/zenodo.15030237

[93] Corina S. Pundefinedsundefinedreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-Level Symbolic Execution and System-Level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA '08)*. Association for Computing Machinery, New York, NY, USA, 15–26.

[94] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate Statistics for Ordinal Level Data: Should We Really Be Using T-Test and Cohen'Sd for Evaluating Group Differences on the Nsse and Other Surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 177.

[95] Judith Segal. 2008. Scientists and Software Engineers: A Tale of Two Cultures. (2008), 8.

[96] Md Saeed Siddik, Hao Li, and Cor-Paul Bezemer. 2026. A Systematic Literature Review of Software Engineering Research on Jupyter Notebook. *Journal of Systems and Software* 235 (May 2026), 112758.

[97] Danilo Silva, Joao Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: A Multi-Language Refactoring Detection Tool. *IEEE Transactions on Software Engineering* 47, 12 (Dec. 2021), 2786–2802.

[98] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 858–870.

[99] Vanessa Sochat, Nicholas May, Ian Cosden, Carlos Martinez-Ortiz, and Sadie Bartholomew. 2022. The Research Software Encyclopedia: A Community Framework to Define Research Software. *Journal of Open Research Software* (2022).

[100] Victoria Stodden and Sheila Miguez. 2014. Best Practices for Computational Science: Software Infrastructure and Environments for Reproducible and Extensible Research. *Journal of Open Research Software* 2, 1 (July 2014), e21.

[101] Tim Storer. 2018. Bridging the Chasm: A Survey of Software Engineering Practice in Scientific Programming. *Comput. Surveys* 50, 4 (July 2018), 1–32.

[102] Jiayi Sun, Aarya Patil, Youhai Li, Jin L.C. Guo, and Shurui Zhou. 2025. Collaboration Challenges and Opportunities in Developing Scientific Open-Source Software Ecosystem: A Case Study on Astropy. *Proc. ACM Hum.-Comput. Interact.* 9, 7, Article CSCW281 (Oct. 2025), 33 pages.

[103] Yiming Tang, Raffi Khatchadourian, Mehdi Bagherzadeh, Rhia Singh, Ajani Stewart, and Anita Raja. 2021. An Empirical Study of Refactorings and Technical Debt in Machine Learning Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 238–250.

[104] Astopy Team. 2025. Astropy Tutorials. https://learn.astropy.org/

[105] GATK Team. 2025. Genome Analysis Toolkit (GATK). https://github.com/broadinstitute/gatk

[106] Junit Development Team. 2025. The programmer-friendly testing framework for Java and the JVM. https://junit.org/

[107] MyPy Team. 2025. MyPy. https://mypy-lang.org/

[108] Monarch Institute Team. 2025. Data Ingestion Pipeline for Monarch. https://github.com/monarch-initiative/dipper

[109] PlasmaPy Team. 2025. Plasmapy Tutorials. https://docs.plasmapy.org/en/stable/examples.html
[110] RefactoringMiner Team. 2025. RefactoringMiner3.0. https://github.com/tsantalis/RefactoringMiner
[111] Ruff Team. 2025. Ruff. https://github.com/astral-sh/ruff
[112] React Development Team. 2025. React: The library for web and native user interfaces. https://react.dev/
[113] Spring Framework Development Team. 2025. Spring makes Java simple, modern, productive, reactive, and cloud-ready. https://spring.io/
[114] Sonar-Qube Team. 2025. Sonar-Qube. https://www.sonarsource.com/products/sonarqube/
[115] Addi Malviya Thakur, Reed Milewicz, Mahmoud Jahanshahi, Lavínia Paganini, Bogdan Vasilescu, and Audris Mockus. 2025. Scientific Open-Source Software Is Less Likely to Become Abandoned Than One Might Think! Lessons from Curating a Catalog of Maintained Scientific Software. *Proc. ACM Softw. Eng.* 2, FSE, Article FSE099 (June 2025), 24 pages.
[116] Erik H. Trainer, Chalalai Chaihirunkarn, Arun Kalyanasundaram, and James D. Herbsleb. 2014. Community Code Engagements: Summer of Code & Hackathons for Community Building in Scientific Software. In *Proceedings of the 18th International Conference on Supporting Group Work* (Sanibel Island Florida USA, 2014-11-09). ACM, 111–121.
[117] Erik H. Trainer, Chalalai Chaihirunkarn, Arun Kalyanasundaram, and James D. Herbsleb. 2015. From Personal Tool to Community Resource: What's the Extra Work and Who Will Do It?. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*. ACM, Vancouver BC Canada, 417–430.
[118] Yuhsiang M. Tsai, Terry Cojean, Tobias Ribizel, and Hartwig Anzt. 2021. *Preparing Ginkgo for AMD GPUs – A Testimonial on Porting CUDA Code to HIP*. Springer International Publishing, 109–121.
[119] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (March 2022), 930–950.
[120] Nikolaos Tsantalis, Matin Mansouri, Laleh M. Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 483–494.
[121] Christos Tsigkanos, Pooja Rani, Sebastian Müller, and Timo Kehrer. 2023. Large Language Models: The Next Frontier for Variable Discovery Within Metamorphic Testing?. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Taipa, Macao, 678–682.
[122] UK-RSE 2025. Society of Research Software Engineering. https://society-rse.org/
[123] US-RSE 2024. The United States Research Software Engineer Association. https://us-rse.org/
[124] Jiawei Wang, Li Li, and Andreas Zeller. 2020. Better Code, Better Sharing: On the Need of Analyzing Jupyter Notebooks. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results* (Seoul, South Korea) *(ICSE-NIER '20)*. Association for Computing Machinery, New York, NY, USA, 53–56.
[125] Yi Wang. 2009. What Motivate Software Engineers to Refactor Source Code? Evidences from Professional Developers. In *2009 IEEE International Conference on Software Maintenance*. IEEE, Edmonton, AB, Canada, 413–416.
[126] G. Wilson. 2006. Software Carpentry: Getting Scientists to Write Better Code by Making Them More Productive. *Computing in Science & Engineering* 8, 6 (Nov. 2006), 66–69.
[127] Ziang Xiao, Xingdi Yuan, Q. Vera Liao, Rania Abdelghani, and Pierre-Yves Oudeyer. 2023. Supporting Qualitative Analysis with Large Language Models: Combining Codebook with GPT-3 for Deductive Coding. In *Companion Proceedings of the 28th International Conference on Intelligent User Interfaces* (Sydney, NSW, Australia) *(IUI '23 Companion)*. Association for Computing Machinery, New York, NY, USA, 75–78.
[128] Zhenchang Xing and Eleni Stroulia. 2008. The JDEvAn Tool Suite in Support of Object-Oriented Evolutionary Development. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. Association for Computing Machinery, New York, NY, USA, 951–952.
[129] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Proceedings of the 38th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '24)*. Curran Associates Inc., Red Hook, NY, USA, Article 1601, 125 pages.
[130] Junwei Yang, Hanwen Xu, Srbuhi Mirzoyan, Tong Chen, Zixuan Liu, Zequn Liu, Wei Ju, Luchen Liu, Zhiping Xiao, Ming Zhang, and Sheng Wang. 2024. Poisoning Medical Knowledge Using Large Language Models. *Nature Machine Intelligence* 6, 10 (Oct. 2024), 1156–1168.