Self-Admitted *Scientific* Debt: Navigating Cross-Domain Challenges in Scientific Software

by

Ahmed Musa Awon
M.Sc., University of Victoria, 2024

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

© Ahmed Musa Awon, 2024
University of Victoria

All rights reserved. This thesis may not be reproduced in whole or in part, by photocopying or other means, without the permission of the author.

Self-Admitted *Scientific* Debt: Navigating Cross-Domain Challenges in Scientific Software

by

Ahmed Musa Awon
M.Sc., University of Victoria, 2024

Supervisory Committee

Dr. Neil A. Ernst, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Department Member
(Department of Computer Science)

Supervisory Committee

Dr. Neil A. Ernst, Supervisor
(Department of Computer Science)

Dr. Margaret-Anne Storey, Department Member
(Department of Computer Science)

ABSTRACT

Scientific software development faces unique cross-domain challenges, requiring expertise from both scientific and software engineering disciplines. These challenges often manifest as technical debt, specifically in the form of Self-Admitted Technical Debt (SATD). While technical debt is a well-recognized issue in software engineering, its impact within scientific software remains underexplored. In particular, the integration of domain-specific scientific knowledge with robust software engineering practices presents ongoing difficulties. This work investigates these cross-domain challenges in scientific software in various fields—including high-energy physics, astronomy, molecular biology, climate modeling, and applied mathematics—through SATD analysis. We examined 28,680 code comments from nine open-source scientific projects, identifying 11 types of technical debt. Among them, we introduced a novel category termed *Scientific Debt*, representing the issues that arise when integrating scientific findings with software development. We identified five key indicators of SD: assumptions, missing edge cases, accuracy challenges, translation challenges, and the incorporation of new scientific discoveries. Our findings reveal that *Scientific Debt* accumulates at a significantly higher rate than it is resolved, with the Missing Edge Cases indicator being the most frequently addressed. To further support the management of this debt, we explore the potential of Large Language Models (LLMs) in identifying and predicting cross-domain challenges. Our preliminary investigation suggests that LLMs could help detect issues requiring both scientific and software expertise, offering a promising direction for future efforts to manage and mitigate *Scientific Debt*.

Table of Contents

Supervisory Committee ii

Abstract iii

Table of Contents iv

List of Tables vii

List of Figures viii

Acknowledgements ix

Dedication x

1 Introduction 1

1.1 The Nature of SSW 3

1.2 Self-Admitted Technical Debt in SSW 4

1.3 Problem Statement and Research Questions 5

1.4 Contributions 6

1.5 Thesis Outline 7

2 Background & Related Work 9

2.1 Technical Debt 9

2.1.1 General Technical Debt 9

2.1.2 Self-Admitted Technical Debt 11

2.1.2.1 Introduction to Self-Admitted Technical Debt (SATD) 11

2.1.2.2 Techniques for Identifying SATD 12

2.2 Scientists and SSW 13

2.2.1 Definition and Characteristics of SSW 13

2.2.2 Unique Nature and Challenges of SSW Development 14

2.2.3 Challenges in Developing SSW 16
2.2.4 Challenges in Testing SSW 17
2.2.5 Comparison Between SSW and Other Software 18

3 Methodology and Experimental Setup 20
3.1 Introduction 20
3.2 Project Selection 21
3.3 Extracting Comments 23
3.4 Identifying SATD Instances 24
3.5 Categorizing SATD Instances 26
3.6 Methodology for Evaluating LLMs in Identifying Scientific Debt . . . 28
3.6.1 Data Collection and Labeling 28
3.6.2 System Prompt Configuration 29
3.6.3 Model Execution and Prediction Recording 29
3.6.4 Evaluation Metrics 29

4 Results 32
4.1 Introduction 32
4.2 RQ1: Cross-Domain Challenges Reflected in SATD in SSW 33
4.2.1 Scientific Debt 34
4.3 RQ2: Addressing Scientific Debt in SSW 39
4.3.1 Introduction and Removal Rates 39
4.3.2 Prioritization of Indicators 41
4.4 RQ3: LLMs in Identifying Cross-Domain Knowledge for Scientific Debt Management 42
4.5 Summary 44

5 Discussion, Implications & Limitations 46
5.1 Knowledge Domains in Domain-Intensive Software 47
5.2 Implications for Researchers 49
5.3 Implications for Practitioners 50
5.4 Threats to Validity 51
5.4.1 Internal Validity 51
5.4.2 External Validity 52
5.4.3 Construct Validity 52
5.4.4 Reliability 53

5.5 Summary 53

6 Conclusion **54**

Bibliography **56**

List of Tables

Table 2.1 Comparison of SSW vs. Other Software 19

Table 3.1 Overview of Case Study Projects 22
Table 3.2 Code Comments and SATD Comments per Project 25
Table 3.3 SATD types used in the coding 31

Table 4.1 Percentage of SD indicator type mostly addressed 41

List of Figures

Figure 4.1 Percentage of SATD types across SSW 33
Figure 4.2 Percentage of Scientific Debt indicators across SSW 38
Figure 4.3 Introduction and Removal Trends of Scientific Debt Across SSW Projects 40

Figure 5.1 Hypothesized Knowledge Competencies Across Domains 48

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to God for providing me with unwavering hope and strength throughout my master's journey. It is only by His grace that I managed to navigate the many highs and lows of this program (and let's be honest, mostly the lows).

To my parents, words cannot capture my appreciation. Thank you for believing in me, even when that belief meant sending me halfway across the world to chase my dreams. I know it hasn't been easy being apart, but your love and sacrifices have been my greatest motivators (and I promise to call more often).

A special thanks to my supervisor, Dr. Neil Ernst, who took a leap of faith by accepting me as his master's student. Whatever research skills I possess today, I owe to your guidance, mentorship, and patience in listening to my... let's call it "creative interpretations" of complex topics. Your unwavering support (and tolerance of my nonsense) made all the difference.

I also want to extend my heartfelt thanks to Swapnil, my research colleague and partner-in-crime throughout this journey. Your assistance, feedback, and moral support (especially when the experiments refused to cooperate) have been invaluable. To the entire CHISEL lab, thank you for creating a welcoming and collaborative environment. You all made this experience one I'll always cherish.

DEDICATION

To my little sister, Maliha — despite my endless attempts to convince you to study computer science (and your even more impressive resistance), this one's for you. May you always follow your own path, even if it's as far away from coding as possible.

Chapter 1

Introduction

Scientific Software (SSW) is essential in modern research, providing crucial tools for complex calculations and data analysis across various scientific disciplines. It is integral to the formulation, testing, and validation of scientific hypotheses, with its accuracy and robustness directly impacting the validity of research findings. This makes the development of SSW a critical aspect of contemporary science, necessitating exceptional levels of precision and reliability [19]. However, achieving this precision and reliability within SSW is challenging. In 2006, the scientific community faced a significant setback when undetected software errors led to the retraction of five high-profile papers [40]. This incident highlighted the significant "interest" paid in terms of compromised accuracy and credibility due to technical shortcomings. The loss of scientific reliability due to these software errors underscores the critical importance of addressing technical debt in SSW to maintain the integrity of research findings.

One of the fundamental challenges in the development of SSW lies in the necessity for a multidisciplinary approach. Building effective SSW requires expertise across various knowledge domains, each contributing to the software's overall accuracy and reliability. According to Kelly et al. [24], these domains include:

1. *Real-World Knowledge*, encompassing an understanding of scientific problems and data.

2. *Theory-Based Knowledge*, involving the scientific principles and models that underpin the software.

3. *Software Knowledge*, which covers programming and software engineering skills.

4. *Execution Knowledge*, related to the practical aspects of running and testing the software.

5. *Operational Knowledge*, concerning the deployment and real-world application of the software.

The intersectional nature of these domains poses significant challenges, as it necessitates collaboration between scientists and software engineers, each bringing their expertise to ensure the software's accuracy and robustness. Of course, such cross-domain challenges exist in most, if not all forms of software, as explored in research into social-technical congruence [11], so the implications of this research are broad. We focus on SSW as one software discipline with readily apparent Real-World and Theory-Based knowledge domains.

Self-Admitted Technical Debt (SATD) provides a unique signal we can use to examine the challenges arising from the intersection of diverse knowledge domains necessary for developing robust software. SATD occurs when developers explicitly acknowledge, through code comments, that certain parts of the code are incomplete, need rework, contain errors, or are temporary solutions [43]. In particular, these comments often reveal areas where the interplay between domain-specific knowledge and software engineering practices may compromise the reliability and correctness of the software.

In the context of SSW, SATD is particularly valuable as it highlights **cross-domain challenges faced by developers in integrating complex scientific theories and models with software implementations of those models**. These challenges are signaled by explicit indicators within SATD comments, pointing to underlying issues, complexities, or deficiencies in the code related to scientific accuracy, assumptions, or computational methods. Such signals provide direct insights into where and why the software might fail to meet scientific and engineering standards. We postulate that addressing these debts would not only enhance the maintainability of the software but also ensure the accuracy and robustness of scientific computations, thereby safeguarding the integrity of research findings. This study aims to explore difficulties with this integration through the lens of SATD.

SSW encompasses a broad spectrum of applications, ranging from simple data analysis tools to complex simulation and modeling systems. However, for the purposes of our study, we are particularly interested in a specific subset of SSW. This subset includes software that is characterized by unique attributes distinguishing it from

other types of software, thereby necessitating a precise definition of our scope.

We adopt inclusion criteria informed by the frameworks provided by [22], [27], [23], and [24]. These criteria ensure our focus remains on software projects developed explicitly to address scientific inquiries. Specifically, we include software that meets the following conditions:

- Developed to address specific scientific questions requiring complex computations.
- Involves domain specialists in its development.
- Utilized by users with expertise in the relevant scientific domain.

We exclude software projects designed solely for controlling physical equipment or serving as generalized computational resources, such as NumPy or SciPy, unless they are directly employed in specific scientific investigations. This focused approach allows us to delve deeply into the unique challenges and intricacies that arise from the integration of multiple knowledge domains in the development of SSW, thereby contributing to a more thorough understanding of its nature and requirements.

1.1 The Nature of SSW

Computational scientists dedicate a substantial portion of their working hours to developing software. An online survey by Pinto et al. [42], which gathered over 1000 responses, revealed that, on average, respondents spent 30% of their working time on software development. This figure has significantly increased over the past decade.

SSW development requires a deep integration of Real-World Knowledge and Theory-Based Knowledge. Scientists must continuously adapt their software to evolving scientific questions and discoveries, necessitating an iterative and exploratory development process. Traditional software engineering methodologies, such as the waterfall model and agile methodologies, are often employed in conventional software development. The waterfall model is a linear and sequential approach, while agile methodologies focus on flexibility, iterative development, and stakeholder collaboration through short cycles. However, these methodical approaches are often unsuitable for SSW development due to the unique demands of the field [22]. This requirement often leads to an "amethodical" approach to software development, characterized by flexibility and independence, where scientists select their tasks based on immediate research needs

without a predefined sequence or external input [42]. This approach allows for the necessary adaptability to tackle complex and evolving scientific problems [23].

In SSW development, the paramount concern is correctness, overshadowing other typical software development priorities such as timeliness, budget adherence, maintainability, or usability [20, 22]. The accuracy of results is critical because errors can compromise the integrity of scientific findings, leading to fundamentally incorrect conclusions. For instance, in 2006, an undetected code fault resulted in the retraction of five published papers and the abandonment of projects that relied on erroneous data [40]. This high standard of accuracy necessitates that software developers possess substantial Theory-Based Knowledge and Real-World Knowledge to fully grasp the intricacies and precision required for scientific applications. Without this expertise, inefficiencies and errors are likely to arise [8, 27]. Hence, the development process must involve scientists closely, particularly during debugging and validation, to ensure that the software meets the stringent accuracy requirements essential for scientific research [22].

Given the critical importance of accuracy, the testing of SSW becomes a particularly challenging and essential aspect of the development process. Unlike commercial software, where outputs and behaviors can often be predicted or expected to follow known patterns, SSW must handle complex physical phenomena, such as hurricane modeling, which inherently lack clear, correct outputs. This uncertainty significantly complicates the validation process, making it difficult to establish straightforward test oracles [26, 19]. An adequate testing strategy must encompass a thorough understanding of the technical and historical context of the software, clearly defined testing goals, the selection of appropriate testing techniques, and the determination of testing adequacy. Each of these aspects plays a critical role in ensuring the software's reliability and accuracy, which are vital in scientific applications. The depth and breadth of testing required often reveal significant gaps in standard testing practices when applied to SSW [28].

1.2 Self-Admitted Technical Debt in SSW

SATD refers to instances where developers explicitly acknowledge shortcomings, workarounds, or temporary solutions in their code through comments. As highlighted by Potdar and Shihab [43], SATD comments provide valuable insights into developers' awareness of potential issues and the rationale behind specific decisions. By examining SATD,

we gain a unique lens through which to view the intricacies and challenges inherent in SSW development, particularly those arising from the interplay of diverse knowledge domains.

SSW often evolves over extended periods [23], continually adapting to new scientific discoveries and computational advancements [46]. This ongoing evolution necessitates a deep integration of Scientific Knowledge and Software Knowledge. The examination of SATD in SSW is particularly insightful, as it sheds light on how these knowledge domains intersect and where they may clash. SATD comments serve as a direct indicator of developers' recognition of potential issues and their rationale behind specific decisions, providing a window into the health and maintenance challenges of the codebase.

1.3 Problem Statement and Research Questions

The development of SSW is fraught with unique challenges due to its integration of multiple knowledge domains. These domains must work in harmony to ensure the software's precision, reliability, and overall quality. This study aims to leverage SATD to explore how cross-domain challenges manifest in SSW and how they are managed. To this end, we have formulated the following research questions:

RQ1: How does SATD signal cross-domain challenges in SSW?

To answer this question, we collect a set of 28,680 SATD comments from representative scientific projects, including high energy physics, astronomy, molecular dynamics, molecular biology, climate modeling, and applied mathematics. We then systematically label these comments using a combination of predefined and emergent labels. Through this process, we aim to understand how SATD highlights the difficulties developers encounter when integrating different knowledge domains, providing a lens through which to view these cross-domain challenges.

In addressing RQ1, we identified a novel type of SATD, which we term Scientific Debt, along with five key indicators. This specific form of technical debt is particularly prevalent in SSW and warrants further investigation, leading us to the subsequent research questions.

RQ2: How is Scientific Debt addressed in SSW?

This question investigates how Scientific Debt is handled within SSW by examining its introduction and removal rates. By analyzing these rates, we can gain insights into the general efforts made to manage Scientific Debt. We further explore which

indicators of Scientific Debt are most frequently addressed, providing an initial understanding of how developers prioritize and tackle this challenge. While this gives a valuable snapshot of current practices, it opens the door for more in-depth exploration in future work.

RQ3: Can Large Language Models predict cross-domain challenges in Scientific Debt management?

Given the growing popularity and demonstrated usefulness of Large Language Models (LLMs) in various domains, we aim to conduct a preliminary investigation into their effectiveness in predicting cross-domain challenges in SSW. Specifically, we explore how well LLMs can understand and predict which GitHub Issues may require both scientific expertise and software engineering knowledge to address or resolve. This low-scale study assesses the potential of LLMs to aid in identifying areas where scientific and software domains intersect, with the goal of enhancing future strategies for managing and prioritizing Scientific Debt.

By addressing these research questions, we aim to shed light on the intricacies of SATD in SSW and the nature of cross-domain challenges. This understanding will contribute to the broader knowledge of technical debt in SSW, ultimately supporting the development of more robust and reliable SSW and enhancing the quality and reproducibility of scientific research.

1.4 Contributions

In this study, we make several significant contributions to the understanding of SATD in SSW:

1. ***Introduction of Scientific Debt:*** We introduce a new type of technical debt, termed ***Scientific Debt***. This type of debt specifically highlights issues within the codebase that are acknowledged by contributors and have the potential to compromise the validity, accuracy, and reliability of scientific results.

2. ***Detailed Analysis of Trends and Patterns:*** We provide a detailed analysis of the trends and patterns observed in the introduction and resolution of Scientific Debt across various SSW. This analysis offers valuable insights into how development teams in different scientific domains manage and prioritize these debts.

3. ***Comprehensive Dataset of SATD Comments:*** We compile a comprehensive dataset comprising 28,680 labeled SATD comments from a diverse range of SSW. This dataset is notable for its inclusion of comments from multiple programming languages, marking the first instance of a SATD dataset that spans such a broad spectrum of languages.

4. ***Repeatable Coding Guide:*** We develop a repeatable coding guide designed to identify and categorize scientific SATD. This guide provides a systematic approach for researchers and practitioners to consistently label and analyze SATD comments in SSW.

Together, these contributions enhance our understanding of the complexities and challenges associated with technical debt in SSW. They provide a foundation for future work aimed at improving the development, maintenance, and reliability of software critical to scientific research.

1.5 Thesis Outline

This thesis is organized into several chapters, each of which addresses different aspects of the research on SATD in scientific soft. The structure of the thesis is as follows:

- **Chapter 1: Introduction**
  The thesis begins with an introduction to the importance of SSW in modern research and the unique challenges it presents. This chapter introduces the concept of SATD within the context of SSW, setting the stage for the investigation of cross-domain challenges that arise during the development and maintenance of such software. The chapter also presents the research questions that guide the study.

- **Chapter 2: Literature Review**
  This chapter reviews relevant literature on Technical Debt, particularly focusing on SATD, and its implications in software engineering. It covers key concepts such as "principal", "debt", and "borrowing" within the Technical Debt metaphor, as well as different types of debt and their impact on software quality. The chapter also examines existing research on SSW development, cross-domain challenges, and the role of LLMs in software engineering.

- **Chapter 3: Methodology**
  This chapter outlines the methodology and experimental setup used to investigate the research questions. It details the selection criteria for SSW, the processes of extracting and classifying SATD comments. The chapter also discusses the usage of LLMs in identifying domain-specific challenges related to cross-domain challenges and provides a systematic approach to ensure transparency and reproducibility in the research.

- **Chapter 4: Results** This chapter presents the results of the study. It begins with an analysis of the distribution of SATD categories and the identification of cross-domain challenges in SSW. The chapter then examines how Scientific Debt is managed across various projects, focusing on its introduction and removal trends, and evaluates the effectiveness of LLMs in identifying cross-domain knowledge relevant to Scientific Debt management.

- **Chapter 5: Discussion**
  This chapter interprets the findings from the results, connecting them to the broader context of SSW development and the management of technical debt. It discusses the implications of the research for SSW developers, particularly in terms of the need for targeted strategies to manage Scientific Debt and the potential role of LLMs in enhancing these efforts. The chapter also addresses the limitations of the study and suggests how they could be addressed in future research.

- **Chapter 6: Conclusion**
  The thesis concludes with a summary of the key findings and their implications for the field of SSW engineering.

Chapter 2

Background & Related Work

2.1 Technical Debt

2.1.1 General Technical Debt

The metaphor of Technical Debt (TD) was first introduced by Ward Cunningham in 1992. Cunningham described it as those internal tasks you choose not to perform now, but that run the risk of causing future problems if not addressed [9]. This concept has since been extensively explored and expanded in software engineering literature to describe not just quick fixes or temporary solutions, but a broader range of software development trade-offs that can have long-lasting impacts on software systems [30].

At its core, TD represents the implied cost of additional rework caused by choosing an easier, quicker, or less effective solution now, instead of using a better approach that would take longer. This concept can be broken down into several components:

- **Principal:** This refers to the immediate cost or effort saved by choosing the quicker, less optimal solution.
- **Debt:** This is the cumulative cost that accrues over time due to the initial sub-optimal decision, which includes interest in the form of increased maintenance costs, decreased code quality, and reduced system reliability.
- **Borrowing:** This represents the act of incurring TD, i.e., making a trade-off decision to achieve short-term gains at the expense of future challenges.

Significant research has focused on how TD is managed within organizations. Research by Lim et al. [35] highlights that while software practitioners may not always

recognize the term "technical debt," they are acutely aware of its implications, which manifest as increased maintenance costs, decreased code quality, and reduced system reliability. Codabux and Williams [7] provide insights into the management practices in an industrial context, while Yli-Huumo et al. [59] discuss variability in management practices across different teams within the same organization, illustrating the lack of a one-size-fits-all strategy for managing TD.

TD encompasses various types, including code, design, architecture, and requirements debt, each representing specific challenges and requiring particular management strategies [1]. For instance, design and architectural debts are often considered the most impactful, as they can lead to significant rework if not addressed early in the development lifecycle [12]. Architectural TD specifically deals with suboptimal decisions in the software architecture that may hinder future system adaptability or performance [54].

The literature also acknowledges the potential benefits of strategically incurred TD. Borg et al. [4] suggest that under certain conditions, TD can be beneficial by allowing faster time-to-market and providing a competitive edge, as long as it is managed effectively and paid back in a timely manner. This is akin to a financial loan, where borrowing allows for immediate progress, but the principal and interest must be repaid to avoid long-term negative consequences.

Prioritizing which TD to address is crucial for effective management. Techniques for prioritizing TD involve assessing the risk and impact of the debt relative to the cost of "repaying" it. This often requires a balance between continuing the development of new features and addressing debt [31]. A systematic literature review by Lenarduzzi et al. provides a comprehensive overview of various strategies and tools used to prioritize TD, highlighting the need for empirical validation of these strategies to enhance their effectiveness in industry settings [31].

Studies have shown that TD is not limited to traditional software systems only. Recent research has revealed its presence in cutting-edge fields like blockchain technology, where developers must manage not only typical software issues but also unique challenges related to decentralized architectures and consensus mechanisms [44]. This indicates the pervasive nature of TD across various domains of software development, including emergent technologies. Furthermore, Vidoni explores TD in the realm of mathematical programming, highlighting how certain coding and documentation practices in this area can lead to substantial TD, affecting the maintainability and evolution of mathematical models [56]. This work closely relates to our study as it

also aims to identify and manage TD within a similarly specialized field. For all these domain-specific software systems, addressing TD effectively requires more than just software engineering knowledge; a deep understanding of the specific domain is crucial.

2.1.2 Self-Admitted Technical Debt

2.1.2.1 Introduction to Self-Admitted Technical Debt (SATD)

Self-Admitted Technical Debt (SATD) refers to the intentional acknowledgment of TD by developers through comments, commit messages, or documentation within the code. This practice is exemplified by the work of Storey et al. [53], who found that developers frequently use task annotations such as TODO and FIXME to highlight areas that require further attention or improvement. These annotations are widely used, with 97% of surveyed developers admitting to utilizing them regularly. They serve various purposes, including marking incomplete features, signaling bugs, and indicating areas needing refactoring. For example, SATD comments might include statements like *"TODO - Move the next two subroutines to a new module called glad_setup? This would be analogous to the organization of Glide."* or *"FIXME(rgk,* $2016-11$*) these should probably be moved to varkindmod?"* These comments provide valuable insights into the developers' awareness of potential issues and their rationale behind specific decisions. However, the challenge with these annotations is their tendency to become outdated, leading to maintenance issues and reduced code readability.

Potdar et al. [43] were the first to provide a formal definition of SATD. They found that developers introduce SATD for several reasons, including time pressure, the need for quick fixes, and dealing with complex problems that require temporary solutions. The study also revealed that SATD often remains unresolved for long periods, with only 26.3% to 63.5% of SATD being addressed across multiple releases. They identified 62 recurring patterns of SATD, helping to recognize its various forms across software projects.

In the context of deep learning frameworks, Liu et al. [37] explored the introduction and removal of SATD. They found that design debt is the most frequently introduced type, followed by requirement and algorithm debt. The removal patterns showed that requirement debt is addressed most promptly, reflecting its critical role in completing functionalities, whereas documentation debt is often neglected.

Sharma et al. [48] highlighted a significant gap in the study of SATD in SSW, particularly in dynamically-typed languages like R. Their work is closely related to our study as it addresses the unique challenges in identifying and managing SATD in scientific computing. They found that SATD in R packages often goes unaddressed, impacting the quality and reliability of SSW. The study by Li et al. [55] found that SATD accounts for about 3% of all comments in R packages, a figure consistent across various programming paradigms. They identified patterns unique to R and highlighted developers' attitudes towards SATD, emphasizing the need for improved comment quality and better management practices.

In the embedded systems industry, Li et al. [34] explored developers' attitudes towards SATD and the triggers for its introduction and repayment. They found that developers recognize the necessity of SATD but often struggle with prioritization and lack adequate tools to manage it effectively. The study suggests several tooling ideas to help developers track and manage SATD more efficiently.

Rantala et al. [45] introduced the concept of Keyword-Labeled SATD (KL-SATD) and found that its introduction correlates positively with the Sqale Index, indicating a relationship with code smells. The removal of KL-SATD is associated with improvements in code maintainability and bug fixing, highlighting the importance of managing TD to sustain code quality.

Lastly, Ferreyra et al. [14] investigated the security implications of SATD, identifying 201 instances containing security pointers that mapped to various Common Weakness Enumeration (CWE) types. They found that while SATD can promote security awareness, it also poses risks by potentially exposing vulnerabilities. This study underscores the dual role of SATD in enhancing and compromising software security, emphasizing the need for careful management of security-related SATD.

2.1.2.2 Techniques for Identifying SATD

Several studies have explored the use of Machine Learning techniques to automatically identify SATD. Maldonado et al. [10] developed an approach using Natural Language Processing (NLP) to detect SATD in source code comments, particularly within Java projects. Their method demonstrated high accuracy in identifying design debt and requirement debt, significantly outperforming previous keyword-based approaches. Similarly, Li et al. [33] applied machine learning techniques to collect and classify SATD from issue tracking systems in open-source projects, using platforms

like Jira and Google Monorail. Their study highlighted the effectiveness of deep learning models in identifying SATD and offering valuable insights for its management. Additionally, Li et al. [32] integrated multiple sources, such as source code comments, commit messages, pull requests, and issue tracking systems, to identify SATD using a convolutional neural network-based model. Their research, which included projects in Java and Python, achieved high identification accuracy by characterizing SATD and extracting relevant keywords and relations across different sources.

Despite the promising results of ML-based approaches, Guo et al. [16] found that using simple task annotation tags like TODO and FIXME can perform equally well or even better than some ML approaches. Their Matching Annotation Tags (MAT) approach does not require labeled training data and demonstrated similar or superior performance compared to existing supervised methods. They also highlighted challenges with current ML methods, such as term diversity, lack of explainability, data dependency, and high computational costs.

Yu et al. [60] further refined the identification of SATD by categorizing it into two types: easy-to-find SATD, which can be effectively captured using pattern recognition techniques, and hard-to-find SATD, which requires human expertise for accurate identification. Their framework, Jitterbug, uses a two-step approach that combines automated techniques for identifying easy-to-find SATD with machine learning for the more challenging hard-to-find SATD, thereby improving overall identification efficiency.

In a comprehensive review of existing techniques, Anim-Annor [2] highlighted the prevalence and patterns of SATD across various programming languages, showing that many tools and methods have been primarily developed for well-known languages like Java. Based on these findings, we observe that less attention has been given to other languages, such as Fortran, C, and C++, which are crucial in scientific computing. This gap underscores the need for more tools and research to address SATD in these less-studied languages, which are highly relevant in our context.

2.2 Scientists and SSW

2.2.1 Definition and Characteristics of SSW

SSW refers to computational tools developed to solve scientific problems, conduct experiments, analyze data, and simulate scientific phenomena. It is distinct from

general-purpose software due to its focus on specific scientific applications and requirements, such as high accuracy, extensive domain knowledge, and long-term usability. There are various types of SSW, including simulation software, data analysis tools, and visualization applications. Each type serves different purposes but shares common complexities such as handling large datasets, performing intricate computations, and providing accurate visual representations of scientific phenomena [27].

Kelly [20] highlights several unique characteristics of SSW. One key characteristic, according to Kelly, is its interdisciplinary nature, requiring collaboration between scientists with domain-specific expertise and software engineers. Kelly notes that this collaboration often leads to challenges in communication and integration of knowledge from different fields. Additionally, Kelly points out that SSW tends to be highly customized, addressing very specific scientific problems, which can limit its reuse and generalization. Furthermore, Kelly emphasizes that the development process for SSW is typically iterative and exploratory, with frequent adjustments based on new scientific discoveries and evolving research needs.

SSW often has a long lifetime, posing unique challenges for its development and maintenance. Unlike commercial software, which may have planned obsolescence or frequent version updates, SSW is expected to remain functional and relevant for many years, even decades [21]. This long-term usability requires continuous updates and maintenance to adapt to new scientific discoveries and technological advancements.

Meng [39] discusses the critical importance of accuracy in SSW. Given that scientific research relies heavily on precise and reliable results, any errors or inaccuracies in the software can significantly impact scientific findings and credibility. This emphasis on accuracy necessitates rigorous testing and validation processes to ensure the software produces dependable results.

2.2.2 Unique Nature and Challenges of SSW Development

The unique nature of SSW developers is shaped by the distinct characteristics and demands of SSW development. Arnold et al. [3] and Koteska et al. [29] highlight that SSW development often requires deep domain-specific knowledge and involves creating highly specialized tools tailored to specific scientific needs. This development process is typically iterative and exploratory, with frequent adjustments driven by new scientific discoveries and evolving research needs.

Wilson [57] underscores several critical challenges faced by SSW developers, par-

ticularly the lack of formal software development skills. According to Wilson, many scientists have limited training in software engineering, which can lead to issues such as poorly structured code and inadequate testing. This computational illiteracy impacts the quality and maintainability of SSW. Wilson further argues that deficiencies in education and training programs for scientists exacerbate this issue, as many curricula do not adequately cover essential software engineering principles. Additionally, there is often resistance to adopting new practices and tools that could improve software quality, due to a combination of entrenched habits and skepticism about the relevance of these practices to scientific work.

Carver et al. [5] emphasize that SSW developers often prioritize the correctness of the scientific computations over software development processes. According to Carver et al., this focus on correct science can lead to an aversion to process-oriented approaches typically advocated in software engineering. They also note that scientists tend to have specific preferences for programming languages and development environments that they are familiar with, which may not always align with best practices in software engineering.

Kelly et al. [25] and Segal [47] both discuss the distinct approach scientists take towards software development. They highlight the educational gaps that leave scientists underprepared for the complexities of software engineering. According to Segal, cultural differences between scientists and professional software developers can create challenges in collaboration, as scientists may not fully appreciate the value of rigorous software development practices. Segal points out that scientists are often considered professional end-user developers, who create software as a means to an end rather than as a primary focus. This role emphasizes the need for domain-specific tools and environments that cater to their unique requirements.

Hannay et al. [17] and Pinto et al. [42] confirm that scientists often develop software in an ad-hoc manner, primarily driven by immediate research needs rather than long-term maintainability. Hannay's survey revealed that the knowledge required to develop and use SSW is mostly acquired through informal self-study (96.9%) and learning from peers (60.1%), with formal education and training at work being less emphasized. Additionally, scientists spend a significant portion of their time developing (30%) and using (40%) SSW, and this has increased over the years. Pinto's replication study with R developers found similar trends, with 99% of respondents emphasizing the importance of self-study. Both studies highlight common challenges such as cross-platform compatibility, poor documentation, interruptions, lack of time,

and collaboration difficulties.

Kelly et al. [24] provide a comprehensive overview of the challenges and characteristics of SSW development, reinforcing the need for improved education and training to bridge the gap between scientific expertise and software engineering skills. They advocate for the integration of software engineering principles into scientific curricula to better equip scientists with the necessary skills to develop robust, maintainable software.

2.2.3 Challenges in Developing SSW

Developing SSW comes with a unique set of challenges that distinguish it from general-purpose software development. Arnold et al. [3] highlight several key challenges in scientific computing, such as the need for high accuracy, the complexity of scientific algorithms, and the integration of diverse data sources. These challenges are compounded by the iterative and exploratory nature of scientific research, which often requires rapid prototyping and frequent adjustments to the software.

Carver et al. [5] discuss the acceptance of agile methodologies in SSW development, noting that agile approaches are often better suited and more accepted because they allow for flexibility and iterative improvements. However, Kelly et al. [22] present a contrasting view, pointing out a mismatch with traditional software development methodologies. They argue that neither agile nor plan-driven methods fully address the unique needs of SSW development. Agile methods may lack the necessary structure for maintaining long-term research software, while plan-driven methods can be too rigid for the exploratory nature of scientific research. Kelly et al. [23] further elaborate on the challenges faced by SSW developers, describing the development process as often amethodical. Standard software development methods do not always align well with the practices and needs of scientific research. To address this, they propose the Onion Model for phased releases, which allows for incremental development and testing. This model helps manage the complexity and evolving requirements typical in SSW projects.

Another significant challenge in SSW development is requirements gathering. Carver et al. [5] emphasize the difficulty of requirements discovery in scientific projects, where the requirements are often not well-defined at the outset and can change significantly as the research progresses. Similarly, Kelly [20] identifies challenges in requirements gathering, noting that scientists and software developers often

have different perspectives and priorities, which can lead to misunderstandings and incomplete requirements.

2.2.4 Challenges in Testing SSW

Testing SSW poses numerous challenges that are distinct from those encountered in traditional software testing. Carver et al. [5] highlight significant challenges in the Verification and Validation (V&V) of SSW. The primary difficulties stem from the complexity of the scientific algorithms and the necessity to ensure that the software accurately reflects the underlying scientific models and theories. Kelly et al. [25], [22] note that, despite the prevalence of testing over other verification methods, SSW often faces significant challenges in V&V. Established testing techniques may not be entirely suitable for the unique requirements of SSW, and there are limitations in their application. This underscores the need for specialized testing methodologies tailored to the scientific domain. Testing in this context is not just about ensuring that the software runs without errors, but also about verifying that it produces scientifically valid results [20].

Sanders [26] identifies several specific challenges in testing SSW, including the complexity of the subject matter, inadequate validation criteria, a high demand for correctness, and a general lack of testing expertise among SSW developers. One of the major issues is the oracle problem, where it is difficult to determine the correct output for a given input due to the complexity of the scientific computations. Sanders also categorizes the risks associated with SSW into three broad areas: computational errors, algorithmic errors, and implementation errors. Hook [19] echoes this sentiment, noting that the absence of reliable oracles makes it challenging to verify the correctness of the software's output, and the sheer number of possible test cases can be overwhelming.

Hook [18] further discusses the challenges in testing SSW, focusing on error detection and accuracy. The impact of code faults can be profound, affecting not only the software's functionality but also the validity of scientific findings. Miller [40] further illustrates the high stakes involved, where inaccuracies due to a software error in a homemade data-analysis program that flipped two columns of data led to the retraction of five scientific papers.

2.2.5 Comparison Between SSW and Other Software

In the previous sections, we have explored the unique characteristics, challenges, and practices associated with SSW development. We have examined various issues faced by SSW developers, such as the lack of formal training, the ad-hoc nature of software development driven by immediate research needs, and the complexities in testing and verification. To provide a clearer understanding, we have summarized the key distinctions between SSW and traditional software, such as general IT and commercial software, in Table 2.1. This table highlights the major differences based on the findings and discussions presented in the preceding sections, offering a comparative perspective on these two domains.

Table 2.1: Comparison of SSW vs. Other Software

Feature/Aspect	**SSW**	**Other Software (General IT, Commercial)**
Primary Goal	Scientific accuracy and domain-specific research outcomes	Business requirements, user satisfaction, market competitiveness
Requirements Discovery	Evolving and often discovered during the project	Usually well-defined upfront, with clear business objectives
Development Team Composition	Primarily domain scientists with some software skills	Professional software developers, often with dedicated roles
Factors Driving Language Choice	Highest possible accuracy and precision for scientific computations	Scalability, maintainability, and ease of development
Development Methodologies	Amethodical, focusing on iterative exploration and problem-solving	Mix of Agile, Waterfall, DevOps depending on project needs
V&V	Difficult due to evolving scientific models and unknown correct results	Standardized testing practices, automated testing, and often clear metrics
Software Lifecycle	Long, often decades, evolving with scientific understanding	Variable, from months to years, depending on market and tech changes
Funding and Support	Often underfunded, project-based funding from research grants	Funded by business revenues, venture capital, dedicated budgets
Training and Education	Mostly self-taught or peer-taught in software practices	Formal education in computer science and ongoing professional training

Chapter 3

Methodology and Experimental Setup

3.1 Introduction

In this chapter, we outline the methodology used to investigate the research questions posed in this thesis. The objective of this research is to systematically analyze SATD in SSW, with a focus on understanding how SATD signals cross-domain challenges. To recap, our primary research questions are:

- **RQ1: How does SATD signal cross-domain challenges in SSW?**

  We aim to investigate how SATD comments reveal cross-domain challenges in SSW. This involves identifying instances where developers encounter obstacles that span scientific and software engineering knowledge domains. By systematically analyzing and labeling a large set of SATD comments, we will explore how SATD captures these challenges, providing insights into the complexities developers face when integrating different knowledge areas.

- **RQ2: How is Scientific Debt addressed in SSW?**

  To understand how Scientific Debt is managed, we will assess its introduction and removal patterns across the studied SSW projects. This analysis will help identify trends in how developers address Scientific Debt and which indicators of debt receive more attention. The results will offer a clearer understanding of current practices in managing TD related to scientific and software engineering interactions.

- **RQ3: Can LLMs identify cross-domain knowledge for Scientific Debt management?**

  Our final question examines the potential of LLMs in predicting cross-domain challenges related to Scientific Debt. We aim to test whether LLMs can identify issues in GitHub repositories that require both scientific expertise and software engineering skills. This preliminary investigation will help assess the role of LLMs in recognizing and prioritizing complex, interdisciplinary issues within SSW development.

In this chapter, we outline the processes and techniques used at each stage to address our research questions, with a focus on ensuring transparency and reproducibility.

The structure of this chapter is as follows: Section 3.2 presents the criteria and methodology for selecting the SSW projects included in our analysis. Section 3.3 explains the techniques used to extract SATD comments from the codebase, including how we identified the introduction and removal dates. In Section 3.4, we discuss the methods employed to filter and identify SATD instances from the broader set of comments. Section 3.5 describes the process of categorizing SATD through manual labeling.

Finally, Section 3.6 details the experiment conducted to evaluate the capabilities of LLMs in identifying cross-domain challenges related to Scientific Debt.

By carefully detailing our methodology and experimental setup, we aim to provide a robust framework that supports replication and validation of our study. This rigorous approach ensures that our conclusions contribute meaningfully to the understanding and management of TD—particularly in addressing cross-domain challenges—in SSW.

3.2 Project Selection

The selection of software projects for our analysis was guided by criteria designed to capture diverse practices and challenges across different scientific domains. We aimed to include influential SSW projects within their respective communities to ensure the relevance and impact of our findings.

Key selection criteria included:

- **Renown**: The project's recognition and reputation within the scientific community.
- **Community Engagement**: Active, long-term discussions in project-specific forums, reflecting ongoing relevance and development.
- **Codebase Size**: The scale of the project's codebase, quantified using the SLOCCount tool[1]. We considered projects with at least 50,000 lines of code ensuring they are of medium to large scale.
- **Development Activity**: Regular updates and maintenance indicated by GitHub commit frequency.
- **Longevity**: Projects active for a minimum of 10 years, suggesting stability and enduring utility.
- **Popularity Metrics**: Indicators such as GitHub stars and forks, along with the number of dependent packages (DP) and dependent repositories (DR) for libraries.

Using these criteria, we identified nine exemplary projects that reflect the diversity and dynamism of advanced open-source software development in various scientific fields. These projects were chosen to provide a broad perspective on the challenges and practices involved in SSW development.

Table 3.1 provides detailed descriptions of each selected project, highlighting their domain, number of contributors (NOC), codebase size, popularity metrics, longevity, and the availability of comprehensive contributor guidelines (CG).

Table 3.1: Overview of Case Study Projects

Project	**Scientific Domain**	**NOC**	**Project Type**	**Code Size**	**Number of Users**	**Age (Years)**	**CG**
Astropy	Astronomy	453	Python Library	1,308,577	768 DP, 40K DR, 3.84K stars, 1.6K forks	13	Yes
Athena[2]	High Energy Physics	100+	Software	5,207,555	–	19	Yes
Biopython	Molecular Biology	331	Python Library	620,437	1.21K DP, 1.47K DR, 3.61K stars, 1.63K forks	25	Yes
CESM	Climate Model	134	Software	2,799,805	265 stars, 154 forks	41	Yes
GROMACS	Molecular Dynamics	85	Software	2,102,045	552 stars, 285 forks	27	Yes
Moose	Physics	221	Framework	847,602	1.5K stars, 979 forks	16	Yes
Elmer	Applied Mathematics	45	Software	954,420	1.1K stars, 292 forks	10	No
Firedrake	Applied Mathematics	96	Software	63,013	451 stars, 156 forks	11	No
Root	High Energy Physics	387	Framework	5,080,496	2.4K stars, 1.2K forks	24	Yes

[1] https://dwheeler.com/sloccount/

By adhering to these rigorous selection criteria, we ensure that our study covers a broad spectrum of SSW projects, providing a robust foundation for analyzing SATD and understanding the cross-domain challenges faced by developers in these fields. This comprehensive approach allows us to draw meaningful conclusions and offer valuable insights into the management and implications of TD in SSW.

3.3 Extracting Comments

The first step in our study was the systematic extraction of source code comments from the selected SSW projects, focusing on various programming languages including Fortran, C, C++, and Python.

Our approach drew inspiration from the work of Maldonado et al. [10] and Liu et al. [37], adapting their methods to our study. We cloned the repositories of the selected projects and examined files within the master or main branches. To accommodate the unique syntax of different programming languages like C and Fortran, we developed custom Python scripts for comment extraction, utilizing the GitPython library to traverse the version control history.

The process was carried out as follows:

1. **Comment Extraction:** We used Python scripts to iterate through each file in the main branch, extracting comments starting from the initial commit. The extracted data included the content of the comment, the file name, and the line number.

2. **Recording Introduction Date:** The introduction date of each comment was recorded as the date of the first commit where the comment appeared.

3. **Tracking Removal Date:** The removal date was recorded as the date of the first commit where the comment no longer appeared in the file.

For multi-line comments, which are common in languages like Fortran and C, we treated them as a single entity, in line with practices described by Freitas et al. [15]. Although individual lines could technically have different introduction or removal dates, we handled these as continuous blocks to simplify the tracking process.

To account for potential file renaming, relocation, or modifications to comments, we carefully reviewed each instance and cross-referenced commit histories. This ensured that the introduction and removal dates were as accurate as possible.

In total, we extracted **1,328,882** comments across the selected projects. All Python scripts used for this extraction, along with the resulting datasets, have been made publicly available to ensure transparency and facilitate future research.

3.4 Identifying SATD Instances

Manually identifying potential SATD from a large volume of comments is a labor-intensive and time-consuming process, one we sought to minimize as much as possible. Previous research illustrates the challenges involved; for instance, Maldonado et al. [10] spent 185 hours classifying 62,556 comments, which clearly demonstrates the exhaustive nature of manual SATD identification. Given the scale of our dataset, it was impractical to manually examine every comment in a similar fashion.

To address this, we adopted a more efficient strategy: a keyword-based search, which has been proven to be both simple and effective. Recent work by Guo et al. [16] supports the efficacy of keyword searches in identifying SATD instances, validating this as a reliable approach. We began by employing a set of 64 established keywords identified by Potdar et al. [43], which are known for detecting what Jitterbug et al. [60] describe as "easy-to-find" instances of SATD.

Recognizing that these keywords might miss more nuanced forms of TD, we enhanced our search by incorporating an additional 597 keywords from Sridharan et al. [52]. These keywords were specifically chosen for their ability to flag more elusive, "hard-to-find" instances of SATD, expanding the scope of our detection efforts. This process was instrumental in capturing a wide range of comments that would not have been identified with only "easy-to-find" keywords.

For example, a comment from CESM included:

> *"Calculate the initial change in TKE for each level. This is done for computational efficiency, it helps because there will be at least one calculation for each grid level, meaning the first one can be done for every grid level and therefore the calculations can be vectorized, clubb:ticket:834.* ***After the initial calculation however, it is uncertain how many more iterations should be done for each individual grid level, and calculating one change in TKE for each level until all are exhausted will result in many unnecessary and expensive calculations."***

Similarly, a comment from Athena included:

> '***so this is a bit of a mess now...*** *if madspin is run from an NLO grid pack the correct lhe events are at both madevent/.../unweighted_events.lhe.gz and madevent/.../events.lhe.gz so there are unweighted events but not in the madspinDir...*"

Ultimately, through this combined list of 661 keywords—597 from Sridharan et al. [52] and 64 from Potdar et al. [43]—we reduced the number of comments in our dataset to **39,697**, each of which potentially indicated TD.

However, this process also captured comments that did not necessarily indicate TD or an issue. For instance, the following comment from Astropy:

> *"Description: Generates a vector of length n containing the integers 0, ..., n-1 in random order. We* ***do not use*** *a new seed."*

was flagged as SATD because of the presence of the phrase *do not use*, despite not actually being a true indicator of TD. To address this, we meticulously reviewed each comment flagged by this search, manually verifying their relevance as indicators of TD. This thorough review process led to the manual elimination of 11,017 comments, leaving a total of **28,680** comments for further analysis and categorization of SATD. An overview of the distribution of these comments across projects is shown in Table 3.2. Notably, the percentage of SATD comments in SSW, ranging from 0.75% to 4.20%, is comparable to those found in non-SSW projects, such as Java projects [43] and Deep Learning Frameworks [37].

Project	**Total Comments**	**SATD Comments**	**SATD Percentage (%)**
Athena	435441	4958	1.14
CESM	231185	4056	1.75
GROMACS	79116	3143	3.97
Astropy	47641	2002	4.20
Biopython	40229	1370	3.41
Root	356163	11603	3.26
MOOSE	89876	675	0.75
Elmer	40386	535	1.32
Firedrake	8845	338	3.82
Total	1328882	28680	2.16

Table 3.2: Code Comments and SATD Comments per Project

3.5 Categorizing SATD Instances

Most previous research on SATD and the automatic tools developed for its identification have focused primarily on Java-based projects. Consequently, we were unable to use these tools to automatically detect different types of TD in the projects we are studying, as they encompass a wide range of programming languages, including Fortran, C, C++, and Python. Additionally, our goal was to identify a broader spectrum of TD to understand which types are more prevalent in SSW. Therefore, we extended our analysis to include newer categories of TD, such as On Hold Debt and Algorithm Debt, as described by Maipradit et al. [38] and Liu et al. [36], respectively, alongside the more commonly recognized types.

To achieve this, we opted for a manual categorization of the code comments. Using the card sorting technique outlined by Spencer [51], we categorized each comment into one of ten predefined categories: Code Debt, Design Debt, Architectural Debt, Build Debt, Documentation Debt, Requirements Debt, Test Debt, Defect Debt, On Hold Debt, and Algorithm Debt. The definitions, examples, and sources for each of these debt types are presented in Table 3.3.

However, during this process, we encountered a number of comments that didn't fit into any of the existing categories. These particular comments stood out because they involved issues that required specific scientific expertise to address, making them different from traditional TD categories. Recognizing this distinction, we set these comments aside and later identified them as a new category, which we termed *Scientific Debt*.

To mitigate personal bias in our manual classification of code comments, a statistically representative subset of 1,000 SATD instances from the 28,680 identified was randomly sampled, using a 95% confidence level with a 10% margin of error. While the expected sample size for these parameters would be 96, we extended the sample to 1,000 comments to improve the precision of the results and capture potential variability in the dataset. This sample was independently classified by a second labeler, and the classifications were compared. A high level of agreement was observed, with Cohen's kappa coefficient of +0.79, indicating substantial inter-rater reliability.

The entire manual classification process, including categorization, identification of new debt types, and independent verification, took approximately 300 hours.

While existing definitions of TD categories were informative, we frequently encountered blurred distinctions between different types of debt. This observation aligns

with findings from other studies, such as [10] and [32], which note the overlap between code and design debt. Additionally, Simon et al. [50] discuss the ambiguity surrounding the definition of Algorithm Debt. In response to these challenges, we developed operational definitions to clarify these distinctions, guiding our labeling of SATD comments. Moreover, recognizing that some code comments could fall under multiple categories, we decided to include all relevant categories for each comment rather than choosing only one. For instance, the comment

> *"TODO - Rewrite glint_remove_bath to support multiple tasks? Calls to this subroutine are currently commented out."*

was labeled as both Code Debt and On Hold Debt. This dual labeling reflects the need to rewrite the subroutine to improve code quality and functionality (Code Debt) and acknowledges the development delay due to dependencies or other pending tasks (On Hold Debt). Including both categories ensures a comprehensive understanding of the TD, recognizing both the quality issues and the current status of the task as pending. This approach provides a more nuanced and accurate representation of the TD present in the code. The definitions of different categories of debt with examples are as follows:

Throughout the categorization process, we thoroughly reviewed the documentation of the SSW projects involved. Leveraging the reliability of LLMs [6], we used tools like ChatGPT and GitHub Copilot to aid in our analysis. ChatGPT helped us understand comments with complex scientific terms and jargon by providing explanations and context for domain-specific terminology, particularly in comments related to scientific algorithms and methods. GitHub Copilot assisted in navigating different programming languages, understanding code structures, and identifying the intent behind code changes, especially in polyglot codebases. Additionally, for each identified SATD instance, we cross-referenced the associated documentation and user manuals to verify context and accuracy. This careful extraction process resulted in a comprehensive dataset of 28,680 labeled SATD comments, which we provide in our replication package[3]. Our scripts and datasets are publicly available to ensure transparency and support future research.

[3] https://anonymous.4open.science/r/ScientificSATD-CBD0/

3.6 Methodology for Evaluating LLMs in Identifying Scientific Debt

To evaluate the effectiveness of ChatGPT in assessing the potential of LLMs to aid in identifying areas where scientific and software domains intersect, we employed a structured methodology. The evaluation was conducted in several steps, each designed to rigorously assess the model's capabilities.

3.6.1 Data Collection and Labeling

The data for this evaluation was collected from the GitHub repository of the **ESCOMP/CTSM** project, which is a subcomponent of the larger **Community Earth System Model (CESM)**. ESCOMP/CTSM focuses on the modeling of terrestrial ecosystems and biogeochemistry.

Each issue in the repository is labeled by the project contributors based on the nature of the problem or task described. These labels help indicate whether an issue involves bugs, performance improvements, or scientific impacts. We used these author-assigned labels to categorize issues into two groups:

- **Cross-Domain Concerns:** We identified 212 issues labeled with "enh - impacts science" and "bug - impacts science." These labels, assigned by the authors, suggest that these issues require both scientific expertise (e.g., to understand the scientific implications) and software engineering knowledge (e.g., to implement or fix the issue).

- **General Technical Concerns:** We collected 257 issues that did not include the above labels, representing more general software engineering concerns without direct scientific impact.

These author-assigned labels serve as the **ground truth** for our study. They provide a reliable foundation for testing the model's ability to identify cross-domain issues where scientific expertise intersects with software development. The objective of this low-scale study is to assess whether LLMs can effectively predict which issues are likely to present both scientific and software engineering challenges, thereby informing strategies for managing Scientific Debt.

3.6.2 System Prompt Configuration

The next step involved configuring a system prompt tailored to guide ChatGPT's analysis of the issues. The prompt was designed to simulate the expertise of a research software engineer specializing in CESM. The specific system message used was as follows:

> *"You are an expert research software engineer specializing in the Community Earth System Model (CESM). Your task is to critically analyze the given title, description, and file changes in an issue and determine if these modifications could potentially alter the scientific results produced by the current implementation of CESM. Please respond only in JSON format with a single key 'impacts_science' whose value is either 'yes' or 'no'."*

This prompt was crafted to ensure that ChatGPT focused on assessing the potential impact of each issue on scientific results, thereby aiding in the identification of potential Scientific Debt.

3.6.3 Model Execution and Prediction Recording

Using the configured prompt, we sent the titles and descriptions of the issues to the ChatGPT API (version GPT-4, accessed in August 2024). The model responded with predictions indicating whether each issue could impact scientific results, formatted as JSON with a binary 'impacts_science' value of either 'yes' or 'no'.

3.6.4 Evaluation Metrics

To measure the performance of ChatGPT in identifying Scientific Debt, we computed the following metrics:

- **Accuracy:** The proportion of correctly identified issues out of the total number of issues.
- **Precision:** The proportion of issues correctly identified as impacting science among all issues labeled by the model as impacting science.
- **Recall:** The proportion of actual science-impacting issues that were correctly identified by the model.

- **F1 Score:** The harmonic mean of precision and recall, providing a balanced measure of the model's performance.

A prediction was considered correct if the model's 'yes' prediction corresponded with an issue labeled "enh - impacts science" or "bug - impacts science." Otherwise, the prediction was deemed incorrect.

Table 3.3: SATD types used in the coding

Debt Type	Definition	Example	Source
Architectural	Issues in project architecture, such as the violation of modularity, which can impact architectural requirements like performance and robustness.	*CESM*: "TODO - Move the higher-order stuff to the HO driver, leaving only the old Glide code."	[1]
Build	Problems in dependency management and build processes, such as disorganized compile flags or problematic build targets, which complicate the build environment.	*Root*: "FIXME! This function is a workaround on OSX because it is impossible to link against libzmq.so"	[1]
Code	Complex, obsolete, or redundant code that compromises code quality or fails to adhere to best coding practices.	*Moose*: "TODO: Rename this method to getName; the normal name (ID) should be getPath."	[1]
Defect	Known bugs or issues within the software that are acknowledged but not yet corrected.	*Elmer*: "For some reason this is not always active. If not set, parallel interpolation could fail."	[1]
Design	Suboptimal design decisions that lead to inconsistent practices or insufficient modularization, complicating future modifications.	*Biopython*: "TODO - How to handle the version field? At the moment, the consumer will try to use this for the ID which isn't ideal for EMBL files."	[1]
Test	Issues related to the testing process, including costly or complex tests, lack of coverage, or inconsistent test results.	*CESM*: "NOTE (bja, 2018-03) ignoring for now... Not clear under what conditions the test is needed."	
Requirements	Unmet or partially implemented requirements, including non-functional ones (e.g., security, performance), affecting system functionality.	*Astropy*: "Binary FITS tables support TNULL *only* for integer data columns. TODO: Determine a schema for handling non-integer masked columns in FITS."	[1]
Docs	Missing, inadequate, or inaccurate documentation that fails to properly guide the user or developer.	*Biopython*: "TODO: add information about what is in the aligned species DNA before and after the immediately preceding 's' line."	[1]
Algorithm	Use of algorithms that are suboptimal or inadequately address the intended problem, resulting in inefficient performance or scalability issues.	*Root*: "TODO: We could optimize based on the knowledge that when splitting a failed partition into two, if one side checks out okay then the other must be a failure."	[38, 55]
On Hold	Development delays caused by waiting for other functionalities to complete or for specific events to occur.	*MOOSE*: "TODO: Add a sync time; Remove after old output system is removed; sync times are handled by OutputWarehouse."	[38, 55]
Scientific	Accumulation of suboptimal scientific practices, assumptions, and inaccuracies within SSW that potentially compromise the validity, accuracy, and reliability of scientific results.	*Detailed examples provided in the following sections.*	this study

Chapter 4

Results

4.1 Introduction

This chapter presents the findings from our study on SATD within SSW, as outlined in the methodologies described in Chapter 3. The results are organized into three key subsections, each addressing a specific aspect of our research questions.

In Section 4.2, we explore the cross-domain challenges reflected in SATD across various SSW projects. This section highlights how the interplay between different knowledge domains—such as scientific theories, computational methods, and software engineering practices—manifests in the accumulation of SATD, particularly focusing on the newly recognized category of Scientific Debt (SD).

Section 4.3 delves into the characteristics of Scientific Debt, examining its introduction, persistence, and resolution within the context of SSW projects. We analyze the dynamics of SD over time, identifying trends in how these debts are managed and the factors influencing their resolution. This section provides a deeper understanding of the nature of SD.

Finally, in Section 4.4, we evaluate the potential of LLMs, such as ChatGPT, in identifying cross-domain knowledge relevant to managing Scientific Debt. This section assesses the effectiveness of LLMs in assisting developers and researchers by detecting domain-specific challenges within SSW.

4.2 RQ1: Cross-Domain Challenges Reflected in SATD in SSW

To analyze the distribution of SATD categories and identify cross-domain challenges in SSW, we began by counting all instances of each SATD category across the selected projects. Comments that were assigned multiple labels were counted separately for each relevant category to accurately capture their prevalence. For instance, a comment categorized as both design debt and code debt was included in the counts for both categories. This comprehensive approach allowed us to determine the representation of each SATD type within the projects.

We then calculated the percentages for each category across all projects and visualized these distributions in a bar chart, as shown in Figure 4.1.

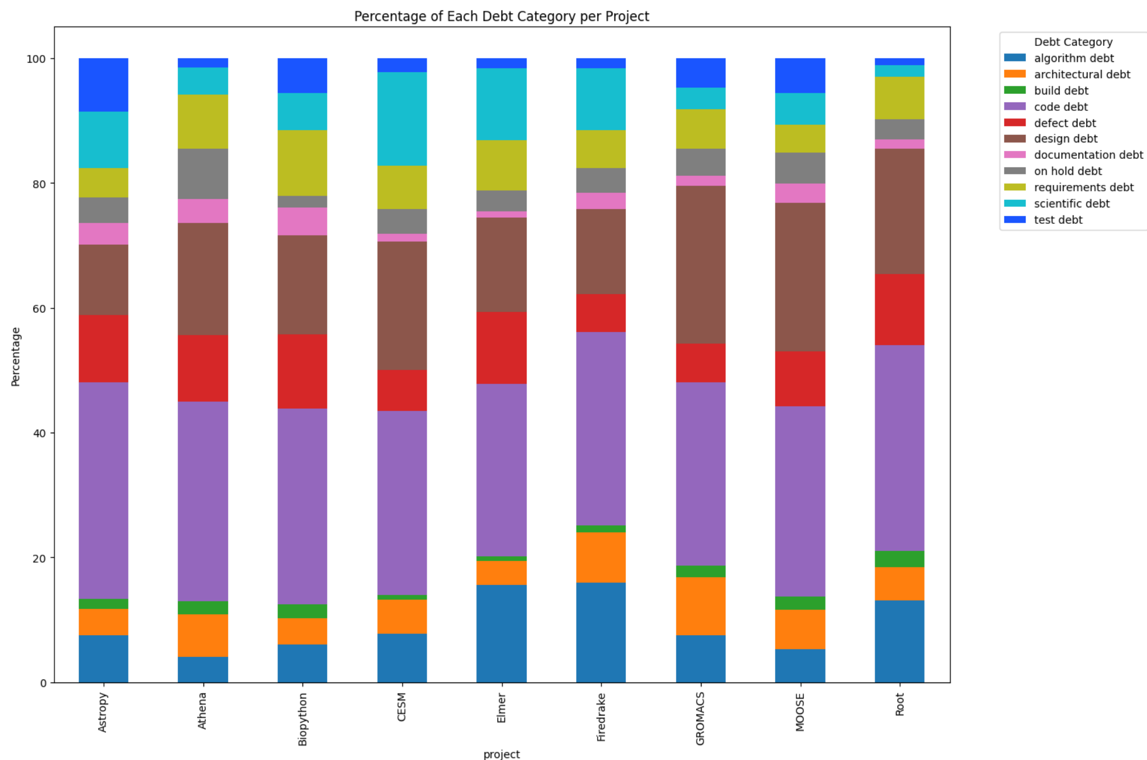

Figure 4.1: Percentage of SATD types across SSW

Figure 4.1 illustrates the distribution of various types of TD across the analyzed projects. We found that Code Debt and Design Debt were the most frequent. This pattern is not unique to SSW and has been consistently observed in previous SATD studies across different domains, such as Java projects [10], deep learning frame-

works [36], R packages [55], and even embedded systems [34]. In these diverse areas, Code and Design Debt consistently dominate, reflecting common challenges in maintaining clean code and optimal design. In the context of SSW, this pattern is further supported by research on the practices of scientists and SSW development [17, 19, 27, 23, 42]. These studies highlight that the SSW community often prioritizes scientific goals over strict adherence to coding and design standards, which likely explains the elevated levels of Code and Design Debt in our findings. In addition to Code and Design Debt, other types such as Architectural Debt, Algorithm Debt, and Requirements Debt were also significant, though less prevalent. Meanwhile, Build Debt and On-Hold Debt were less commonly observed.

A considerable portion of the TD in SSW stems from cross-domain challenges, which we define as **Scientific Debt**. These challenges arise at the intersection of scientific theories and software engineering practices, complicating the process of integrating scientific models into computational systems. Scientific Debt is prevalent across all projects, with CESM (14.43%), Elmer (11.16%), and Firedrake (9.21%) exhibiting the highest percentages.

To gain a deeper understanding of how Scientific Debt is distributed across projects, we calculated the variance of the Scientific Debt percentages. The resulting variance, 12.23, indicates a significant spread in the levels of Scientific Debt. This suggests that while some projects, such as CESM and Elmer, have a much higher proportion of Scientific Debt, others, like GROMACS and Athena, show considerably lower levels. The high variance highlights that Scientific Debt is not evenly distributed across projects, with certain software facing more substantial cross-domain challenges between scientific modeling and software engineering.

The identification and analysis of Scientific Debt underscore the importance of addressing these unique complexities within SSW development. By categorizing and understanding these debts, we can better appreciate the challenges developers face and implement more effective strategies to manage them, ultimately enhancing the robustness and reliability of SSW.

4.2.1 Scientific Debt

Unlike traditional forms of TD, which primarily concern software engineering issues such as code quality and design practices, **Scientific Debt** specifically arises from the challenges inherent in translating complex scientific methodologies into computational

models. We define this novel category as *the accumulation of suboptimal scientific practices, assumptions, and inaccuracies within SSW that potentially compromise the validity, accuracy, and reliability of scientific results.*

In our labeling, we found that Scientific Debt can manifest in various forms, which we refer to as indicators:

- **Translation Challenges:** Difficulties in accurately representing scientific concepts and theories within computational frameworks. This can lead to oversimplifications or incorrect implementations that do not fully capture the intricacies of the original scientific models.
  - **Example from Astropy:** *"We are going to share neff between the neutrinos equally. In detail, this is not correct, but it is a standard assumption because properly calculating it is (a) complicated (b) depends on the details of the massive neutrinos (e.g., their weak interactions, which could be unusual if one is considering sterile neutrinos)."*

    This example highlights the challenge of simplifying complex interactions between neutrinos for practical implementation in the code. The standard assumption used, though common, is acknowledged as not entirely accurate, demonstrating the trade-off between scientific precision and computational feasibility.
  - **Example from Elmer:** *"The computation of the differential of the Hencky strain function is based on its truncated series expansion. TO DO: The following involves the differential of the Hencky strain function. For some reason it doesn't appear to give convergence. Therefore, we still omit this and replace the Hencky strain differential by the differential of the Lagrangian strain. This is expected to work for reasonably small straining. Find a remedy!"*

    This comment points out an unresolved issue with the convergence of a specific mathematical function used in the software. The temporary solution, while working under certain conditions, highlights the ongoing struggle to achieve an accurate and reliable representation of the scientific model.
- **Assumptions:** The necessity to embed assumptions within the code due to limitations in data, understanding, or computational resources. These assump-

tions, while necessary for initial model development, may introduce inaccuracies or biases that affect the outcomes of simulations and analyses.

- **Example from CESM:** *"We assume here that new ice arrives at the surface with the same temperature as the surface. TODO: Make sure this assumption is consistent with energy conservation for coupled simulations."*

  This example showcases an assumption made to simplify the modeling of ice formation. The need to verify this assumption underscores the potential risk of it affecting the accuracy of energy conservation in coupled simulations, reflecting the impact of embedded scientific assumptions.

- **New Scientific Findings:** The need to continually update software to reflect the latest scientific discoveries and advancements. Failure to incorporate new findings can result in outdated models that do not leverage the most current scientific knowledge, thereby diminishing the relevance and accuracy of the software.

  - **Example from Astropy:** *"This frame is defined as a velocity of 220 km/s in the direction of l=270, b=0. The rotation velocity is defined in: Kerr and Lynden-Bell 1986, Review of galactic constants. NOTE: should this be l=90 or 270? (WCS paper says 90)."*

    This comment indicates a discrepancy in the scientific constants used, with references to differing values in literature. It highlights the necessity to review and update the code to incorporate the most accurate and current scientific findings.

- **Missing Edge Cases:** Limitations in the software's ability to handle all relevant scenarios or edge cases. This can lead to incomplete or erroneous results, particularly in complex scientific domains where edge cases may have significant implications.

  - **Example from ROOT:** *"This does not work for large molecules that span ¿ half of the box!"*

    The comment points out a limitation in the software's capability to handle large molecules, which could lead to significant inaccuracies in simulations involving such cases. It underscores the importance of ensuring compre-

hensive coverage of all possible scenarios to maintain the reliability of the software.

- **Computational Accuracy:** Instances where the mathematical or scientific accuracy within the software is compromised. This can occur due to simplifications, numerical precision issues, or incorrect implementation of scientific algorithms, leading to unreliable or incorrect results.
  - **Example from GROMACS:** *"TODO: For large systems, a float may not have enough precision."*

    This comment highlights a concern with numerical precision in large systems. The use of float data types may lead to significant inaccuracies, indicating a need for better precision management in scientific computations.
  - **Example from GROMACS:** *"Since the energy and not forces are interpolated, the net force might not be exactly zero. This can be solved by also interpolating F, but that comes at a cost. A better hack is to remove the net force every step, but that must be done at a higher level since this routine doesn't see all atoms if running in parallel. Don't know how important it is? EL 990726."*

    This comment describes a potential issue with force calculations due to interpolation methods used. The proposed hack to address the issue indicates a temporary workaround, highlighting the compromise in scientific accuracy and the need for a more robust solution.

Figure 4.2 shows the distribution of Scientific Debt indicators across the analyzed projects, with Assumptions and Missing Edge Cases being the most frequent. For example, GROMACS has 37.82% assumptions, and CESM has 34.53%. These high percentages suggest that these projects often rely on assumptions to simplify complex phenomena or compensate for limited data and computational resources, reflecting a pragmatic approach to advancing scientific inquiry.

Similarly, missing edge cases are prevalent, particularly in Firedrake (38.24%) and Elmer (33.33%). This indicates significant challenges in handling all possible scenarios within these projects, highlighting the difficulties in comprehensively testing and validating SSW, which often deals with highly variable data and complex phenomena.

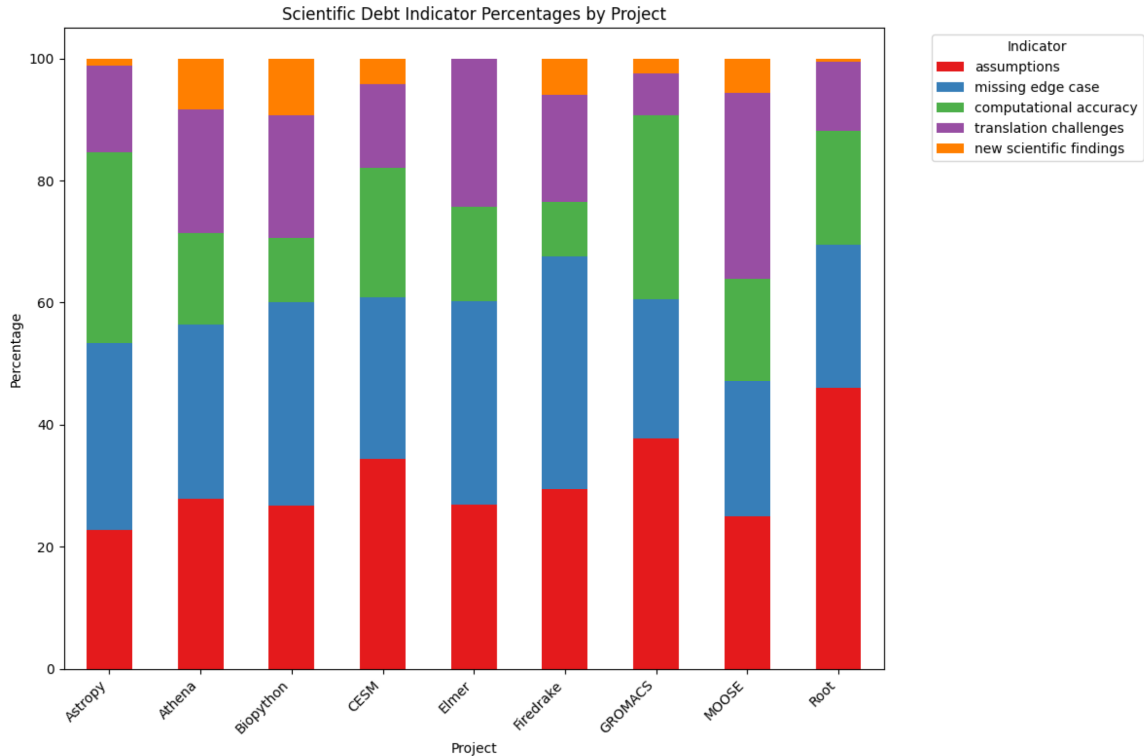

Figure 4.2: Percentage of Scientific Debt indicators across SSW

Computational accuracy issues are particularly prominent in projects like Astropy (31.25%) and GROMACS (30.25%), underscoring the continuous challenge of maintaining numerical precision and reliability in scientific computations.

Translation challenges are significant in projects such as MOOSE (30.56%) and Athena (20.33%), highlighting the complexity of converting theoretical scientific models into practical computational algorithms, which reflects the intricate nature of SSW development.

Although less prevalent, the integration of new scientific findings is still noteworthy. Projects like Biopython and Athena exhibit 9.33% and 8.30%, respectively, for this indicator. SSW continuously update their scientific models with the latest findings to remain relevant and accurate. However, the relatively lower percentages compared to other indicators suggest that such updates do not occur as frequently as the other indicators, reflecting the fact that new scientific discoveries occur at a slower pace or that the integration of these findings is a less common but still essential part of maintaining the scientific integrity of the software.

Summary

While Code Debt and Design Debt are the most prevalent types of SATD in SSW, we identified a novel category called Scientific Debt. This category, significant across all projects, includes indicators such as assumptions, missing edge cases, computational accuracy, translation challenges, and new scientific findings.

4.3 RQ2: Addressing Scientific Debt in SSW

4.3.1 Introduction and Removal Rates

To understand the dynamics of SD within SSW projects, we analyzed its introduction and removal over time. We determined the introduction and removal years for each comment from the inception of the projects. Normalizing the yearly counts of introduced and removed SD comments by the total number of SD comments for each project allowed for comparisons across projects of varying sizes and activity levels. To enhance readability and interpretability, we applied spline smoothing to the normalized data, reducing noise and providing clearer visualizations of long-term trends. The resulting trends are illustrated in Figure 4.3.

Figure 4.3 shows that most projects, such as Astropy, Biopython, Elmer, GROMACS, and CESM, exhibit a clear pattern of increasing SD introduction rates early in their life cycle, peaking mid-life, and subsequently declining. This trend suggests rapid incorporation of new features and scientific advancements during early and mid-development stages, leading to accumulated issues related to scientific accuracy and reliability. However, projects like MOOSE and Athena show fluctuations in their introduction rates, while Firedrake and Root display a continuous increase in SD introduction, reflecting ongoing integration of new functionalities and possibly a growing user base requiring continuous enhancements. These differences could be due to several factors, including project-specific priorities, available resources, and the complexity of the scientific problems being addressed.

The rate of SD removal is consistently lower than the rate of introduction across all projects. This discrepancy highlights the challenge of managing and mitigating SD effectively, as new SD is introduced faster than it is resolved. Moreover, for the removal of SD, most projects, including Astropy, Biopython, GROMACS, MOOSE,

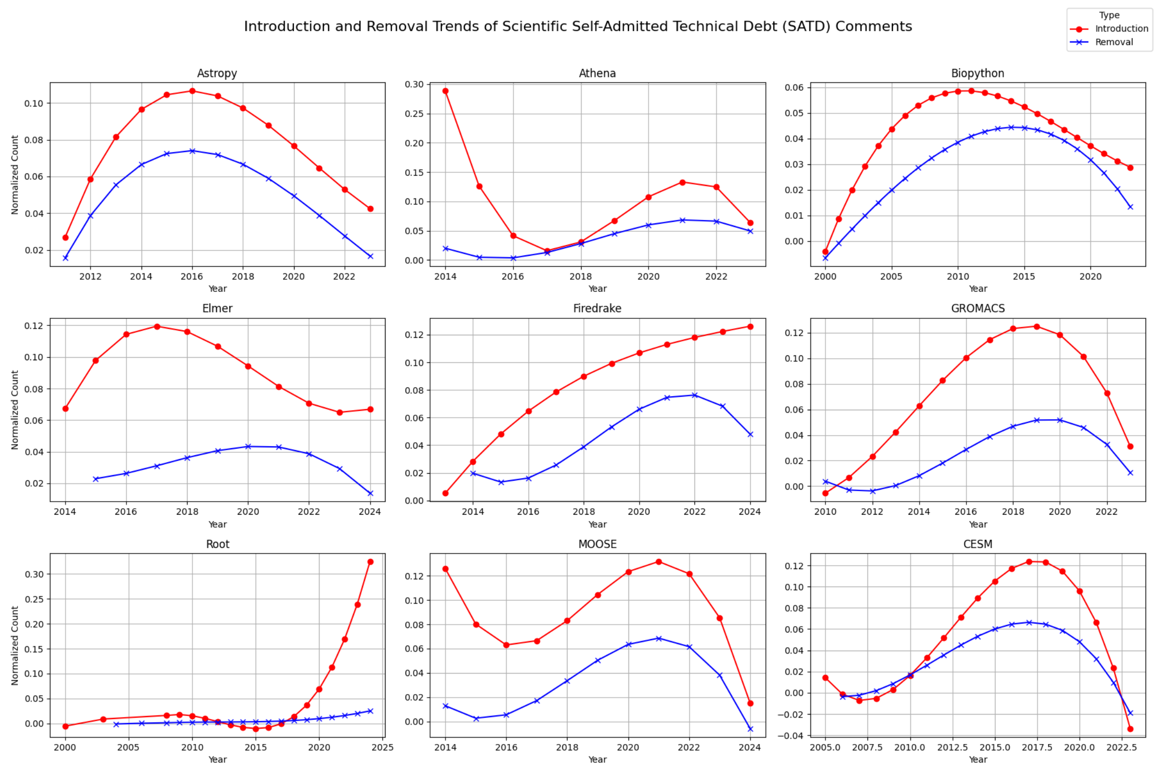

Figure 4.3: Introduction and Removal Trends of Scientific Debt Across SSW Projects

and CESM, display parallel patterns where the removal trends closely follow the introduction trends.

Upon a closer analysis of the dataset to understand the parallelism, the data indicates that efforts to address Scientific Debt are frequently initiated soon after it is introduced, particularly when the solution is known and readily implementable. However, SD comments lacking a removal date tend to remain unaddressed for extended periods, suggesting they may be effectively ignored. This persistence implies that unresolved issues are either particularly challenging or deemed less important, causing them to remain in the backlog indefinitely. Additionally, TD that is not causing immediate problems (interest) is not prioritized for resolution.

Overall, the findings from this analysis underscore the complexities and challenges associated with managing Scientific Debt in SSW projects. The observed trends highlight the need for more effective strategies and tools to identify, prioritize, and address SD to enhance the reliability and maintainability of SSW.

4.3.2 Prioritization of Indicators

To gain deeper insights into which indicators of SD are prioritized and which are often ignored, we calculated the percentage of comments that have been addressed, indicated by the presence of a removal date. The results of this analysis are summarized in Table 4.1, which shows the percentage of SD comments that are mostly addressed across various indicators.

Table 4.1: Percentage of SD indicator type mostly addressed

Indicator Type	Percentage Addressed
New Scientific Findings	54.92%
Computational Accuracy	53.57%
Missing Edge Cases	50.41%
Translation Challenges	44.74%
Assumptions	41.38%

Table 4.1 reveals that "New Scientific Findings" is the most frequently resolved SD indicator, with a 54.92% resolution rate. This high resolution rate likely stems from the necessity of incorporating the latest advancements to maintain software relevance and accuracy. Following closely is "Computational Accuracy" with a 53.57% resolution rate, underscoring the critical importance of precision in computations for ensuring valid research outcomes.

In contrast, "Translation Challenges" and "Assumptions" are the least addressed categories, with resolution rates of 44.74% and 41.38%, respectively. These lower rates highlight the persistent difficulties in accurately translating complex scientific concepts into code and managing the embedded assumptions that often accompany SSW development.

This data suggests a selective prioritization in addressing SD, driven primarily by the immediate impact on scientific accuracy and reliability. Categories that directly affect the correctness of scientific computations, such as new scientific findings and computational accuracy, tend to be resolved more promptly. Meanwhile, more abstract issues like translation challenges and assumptions are less frequently addressed and tend to persist longer.

These findings underscore the need for targeted strategies to manage Scientific Debt effectively. Addressing less frequently resolved issues, such as translation challenges and assumptions, could significantly enhance the overall robustness and re-

liability of SSW. By understanding which types of SD are prioritized and why, we can better develop methodologies and tools to support developers in mitigating these debts, ensuring the long-term sustainability and accuracy of SSW projects.

Summary

The rate at which Scientific Debt is addressed is significantly lower than the rate at which it is introduced. The removal of SD often parallels its introduction, indicating that solutions are often implemented soon after SD is identified. Among the different indicators, New Scientific Findings are the most frequently addressed, reflecting their critical importance. In contrast, Translation Challenges and Assumptions are the least addressed, highlighting ongoing difficulties in these areas.

4.4 RQ3: LLMs in Identifying Cross-Domain Knowledge for Scientific Debt Management

Managing SSW effectively requires deep cross-domain knowledge, as demonstrated by Kelly [24]. The complexity observed in our Scientific Debt categories highlights the importance of understanding both scientific and software engineering domains.

A significant challenge in this context is the need to deepen expertise in a specific domain—such as climatology for the Community Earth System Model (CESM)—to better understand the relevant scientific context. This challenge was identified as a critical pain point in Pinto et al.'s study [42], where the difficulty of building expertise in either software development or the scientific domain, depending on one's background, was emphasized.

LLMs have been posited as tools capable of supplementing background knowledge and have shown promising results in generating summaries and passing domain-specific quizzes [6], [58]. We aimed to determine if LLMs such as ChatGPT could assist in identifying domain-specific challenges that require more than just software engineering expertise. If LLMs can help efficiently identify code changes that impact scientific outcomes, this would improve the prioritization of which SATD to target for removal.

To test our hypothesis, we focused on evaluating ChatGPT's ability to identify issues within GitHub repositories that could impact scientific outcomes. Several scien-

tific repositories label issues and Pull Requests (PRs) with potential scientific impact, as this is usually a primary concern for stakeholders.

The results of our analysis are summarized below:

- **Accuracy**: 0.75
- **Precision**: 0.68
- **Recall**: 0.95
- **F1 Score**: 0.80

The results indicate that ChatGPT is quite effective in identifying issues that impact scientific results, with a high recall of 0.95. This means that the model is excellent at detecting issues that indeed impact science, which is crucial for ensuring that critical issues are not overlooked. For example, consider the following issue identified in the CTSM project:

GitHub Issue Snapshot

Title: PotVeg surface dataset using the wrong Urban dataset
Description: The Potential Vegetation surface dataset: ../..c240216.nc is using the wrong urban input dataset. It should be using the zero-percent urban dataset rather than the standard one.

In this case, ChatGPT successfully identified the issue where the Potential Vegetation surface dataset was using the incorrect urban input dataset, which would have resulted in significantly incorrect scientific outcomes. The model flagged this as a critical issue, highlighting its ability to detect errors that have a substantial impact on scientific results.

However, the precision of 0.68 suggests that while the model identifies most of the relevant issues, it also flags some issues that do not impact scientific results. For instance, the model occasionally flagged minor formatting inconsistencies in code comments as potential scientific impacts, even though these did not affect the scientific outcomes. Notably, some issues flagged by the model as impacting science appeared to be relevant but did not have the "impacts science" labels, indicating that contributors might have overlooked labeling these issues. This trade-off indicates that while the model is a powerful tool for highlighting potentially impactful

issues, further refinement and perhaps human verification are necessary to ensure optimal accuracy.

These findings suggest that LLMs like ChatGPT can play a crucial role in managing Scientific Debt by identifying issues requiring domain-specific expertise. Future research should aim to improve the model's precision and integrate it into development workflows to help researchers and developers prioritize and address TD more effectively. By leveraging the capabilities of LLMs, we can enhance our understanding and management of cross-domain challenges in SSW, ultimately improving the reliability and accuracy of scientific outcomes.

Summary

Our analysis shows that LLMs like ChatGPT can effectively identify issues impacting scientific outcomes in SSW. The model achieved high recall, indicating strong detection of relevant issues, though the precision rate suggests some false positives. These findings highlight the potential of LLMs in managing Scientific Debt by identifying cross-domain knowledge requirements, aiding in resource allocation, and issue prioritization.

4.5 Summary

The analysis of SATD within SSW has yielded significant insights into the nature of TD in this specialized context and the challenges faced by developers in managing it. Through the examination of the three research questions, we have uncovered several key findings.

Our investigation into cross-domain challenges reflected in SATD (RQ1) revealed that SD is a prevalent and distinct category of TD in SSW. SD is characterized by issues that arise from the intersection of scientific and software engineering domains. The prevalence of Code Debt, Design Debt, and Defect Debt underscores the prioritization of immediate scientific objectives over robust coding and design practices. This trade-off often leads to the accumulation of SD, which manifests as translation challenges, assumptions, new scientific findings, missing edge cases, and computational accuracy issues.

Examining the characteristics of SD (RQ2), we found that the introduction of SD occurs at a faster rate than its resolution across most SSW projects. These projects

typically show a pattern of increasing SD introduction rates early in their life cycle, peaking mid-life, and subsequently declining. However, the rate of SD removal consistently lags behind the rate of introduction, indicating persistent challenges in managing and mitigating SD. Parallel patterns of SD introduction and removal in several projects suggest that efforts to address SD are often initiated soon after it is introduced, particularly when solutions are known and readily implementable. Nonetheless, many SD comments remain unaddressed for extended periods, implying that these issues are either particularly challenging or deemed less important.

The evaluation of LLMs in identifying cross-domain knowledge for SD management (RQ3) demonstrated that LLMs like ChatGPT can effectively assist in identifying domain-specific challenges within GitHub repositories. The model exhibited high recall in detecting issues that impact scientific outcomes, suggesting its potential utility in ensuring that critical issues are not overlooked. However, the precision of the model indicates that it also flags some issues that do not impact scientific results, highlighting the need for further refinement and human verification. These findings suggest that LLMs can play a crucial role in identifying and prioritizing SD, particularly in areas requiring domain-specific expertise.

Overall, the findings from our research underscore the complexity and multifaceted nature of TD in SSW, particularly the intricate interplay between various knowledge domains. The cross-domain challenges reflected in SD highlight the need for targeted strategies that address the integration of scientific theories, computational methods, and software engineering practices. Effectively managing SD is crucial for maintaining the reliability and accuracy of SSW, which in turn supports the integrity of scientific research outcomes. The use of LLMs presents a promising avenue for enhancing the identification and management of SD by bridging gaps in domain-specific knowledge and improving the prioritization of TD remediation efforts.

In the next chapter, we will examine the challenges and opportunities for future research, focusing on refining the capabilities of LLMs to increase their precision and integrating these models into standard development practices.

Chapter 5

Discussion, Implications & Limitations

In this chapter, we explore the broader implications and insights gained from our study on SATD in SSW, with a particular focus on the complex interplay between different knowledge domains. Our findings on Scientific Debt in SSW provide valuable insights when viewed through the lens of Diane Kelly's knowledge acquisition model [24], which emphasizes the multifaceted nature of knowledge required in domain-intensive software development. By contextualizing our findings within this framework, we aim to deepen the understanding of how cross-domain challenges contribute to the accumulation and management of SD in SSW.

We begin by discussing the various knowledge domains involved in SSW development and how our findings complement existing theoretical models. This discussion highlights the importance of integrating diverse knowledge types—such as Real-World Knowledge, Theory-Based Knowledge, Software Knowledge, Execution Knowledge, and Operational Knowledge—in addressing SD effectively. Understanding these domains is crucial for practitioners and researchers alike, as it provides a foundation for developing targeted strategies to manage SD and enhance the overall quality of SSW.

The chapter then moves on to explore the practical implications of our findings for both researchers and practitioners. We offer actionable recommendations that can be applied in real-world scenarios to improve the management of SD, emphasizing the importance of interdisciplinary collaboration and the potential role of emerging tools like LLMs. Additionally, we consider the implications for future research, suggesting areas where further investigation could yield valuable insights into the dynamics of

SD in SSW.

Finally, we address the limitations of our study, acknowledging the constraints that may have influenced our findings and the steps taken to mitigate potential biases. By critically examining these limitations, we provide a balanced perspective on the study's contributions and outline directions for future research. This comprehensive discussion aims to guide future efforts in enhancing the quality, sustainability, and effectiveness of SSW development in domain-intensive environments.

5.1 Knowledge Domains in Domain-Intensive Software

Our findings on SD in SSW provide complementary insights when viewed through the lens of Diane Kelly's knowledge acquisition model [24]. Kelly's model emphasizes continuous knowledge acquisition and integration, highlighting the need for deep domain-specific knowledge and systematic approaches in SSW development.

Figure 5.1 is an application of Kelly's knowledge domains to two hypothesized team members. The "Scientist" represents an individual holding an advanced degree in a science domain, who learns software development on the job. This is characteristic of most SSW developers [17]. The "Developer" represents team members with training in software engineering, e.g., through a Computer Science (CS) degree. The spider diagram captures the extent of knowledge and skill in the different domains. Developers are more skilled in Software and Execution, while Scientists excel in the science (Theory) and Operation of the software (e.g., in making climate predictions).

Both our study and Kelly's model emphasize the importance of domain-specific knowledge in SSW development. We identified a novel category of TD, termed *Scientific Debt*, with indicators like assumptions, missing edge cases, computational accuracy, translation challenges, and new scientific findings. This aligns with Kelly's emphasis on Real-World and Theory-Based Knowledge, highlighting the necessity for developers to deeply understand the scientific problems and principles behind the software. The relevance of domain knowledge is also acknowledged in other work, such as attracting new project participants [13], building effective cross-disciplinary projects [11], and requirements engineering [41] (though knowing too much can sometimes be unhelpful [49]).

Our observation of increased Scientific Debt during early software development

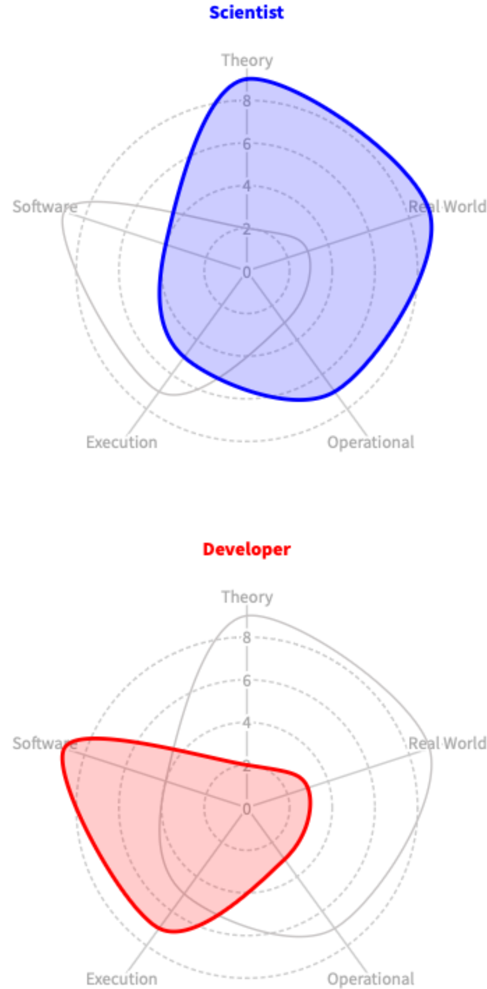

Figure 5.1: Hypothesized Knowledge Competencies Across Domains

stages reflects the challenges of integrating scientific knowledge into code. Ongoing interviews with project practitioners (as part of a related but separate study) support this finding, as they highlighted difficulties in *translating mathematical algorithms into efficient Python code*, underscoring the alignment between Kelly's work, our data, and practitioner experiences. These challenges are often compounded by the rapid evolution of scientific domains, requiring constant updates and adjustments to the software to accommodate new discoveries and methodologies.

Furthermore, Kelly's model underscores the integration of testing with development to ensure software accuracy and reliability. This is mirrored in our finding that efforts to address SD often follow its introduction, indicating a systematic approach to managing TD in SSW. In particular, our study reveals that addressing SD is not

just about fixing bugs or optimizing code, but also about ensuring that the software remains scientifically valid and reliable as new scientific knowledge emerges.

In addition to the integration of testing, the collaboration between Scientists and Developers is crucial for successful SSW development. Scientists bring in-depth domain knowledge and understanding of the theoretical aspects, while Developers contribute their expertise in software engineering practices and execution. This collaboration is essential for bridging the gap between scientific theory and practical application, ensuring that the software developed is both scientifically accurate and technically robust.

In summary, the application of Kelly's knowledge acquisition model to our study of SD in SSW highlights the critical role of domain-specific knowledge and systematic development approaches. By understanding and addressing Scientific Debt, SSW projects can improve their code quality, maintainability, and scientific reliability, ultimately contributing to the advancement of scientific research and discovery. The integration of domain expertise, continuous learning, and collaborative efforts between Scientists and Developers is key to overcoming the challenges inherent in SSW development and ensuring the creation of high-quality, reliable SSW.

5.2 Implications for Researchers

While collaboration between diverse knowledge domains is crucial, the exact nature of this collaboration remains ambiguous. For instance, studies have shown that pairing scientists with software engineers can lead to challenges, as seen when a software engineer and an astronomer struggled to align unit testing with scientific goals, and another case where a software engineer faced difficulties applying standard testing practices with nuclear scientists [28, 8]. These examples suggest that simply putting different skills together may not suffice. **This reinforces the key implication that effective collaboration requires more than just assembling diverse skills; it necessitates a deep integration and understanding of both scientific and engineering perspectives,** as supported by research in socio-technical congruence [11].

Our research reveals the different types of knowledge necessary for SSW development, such as Real-World Knowledge, Theory-Based Knowledge, Software Knowledge, Execution Knowledge, and Operational Knowledge. However, **the key implication here is that it is not yet clear which of these knowledge types is**

harder to acquire or better supported in LLMs. For example, is Real-World Knowledge, which requires a deep domain-specific understanding of complex mathematics and scientific principles, more challenging to gain than Software Knowledge? An important corollary here is that scientific projects often cannot pay skilled software professionals what they might expect, resulting in more scientific developers moving from scientific domains, rather than software backgrounds [42].

Additionally, **further research into the specific challenges and best practices for integrating these diverse knowledge domains is critical for developing more effective strategies for SSW development.**

5.3 Implications for Practitioners

The identification and categorization of SD indicators offer a framework for practitioners to prioritize TD. **Understanding that new scientific findings and computational accuracy are more frequently addressed, while translation challenges and assumptions are often neglected, allows for targeted strategies.** Practitioners can focus their efforts on these often-overlooked areas, ensuring that the software remains robust and scientifically accurate.

The parallel patterns in SD introduction and removal highlight the need for a systematic approach. **Promptly addressing SD can prevent long-standing issues, so regular reviews and refactoring cycles are essential.** Implementing a structured process for managing SD, such as integrating SD reviews into regular development cycles, can help ensure that TD is addressed in a timely manner. This proactive approach can significantly reduce the accumulation of unresolved TD, improving the overall quality and maintainability of the software.

Using LLMs like ChatGPT shows promise in managing SD by identifying domain-specific issues. **Integrating LLMs into the development workflow can help detect potential scientific impacts early, enabling timely interventions.** However, human verification is necessary to improve precision and reduce false positives. Practitioners should be trained to effectively use these tools, ensuring that they can leverage the capabilities of LLMs while also applying their domain expertise to validate the results.

The complexity of acquiring Real-World Knowledge versus Software Knowledge suggests careful team composition. Interdisciplinary teams of scientists and software engineers can bridge knowledge gaps and enhance software reli-

ability and accuracy. Training programs focused on both domain-specific knowledge and software engineering principles can further support collaboration. **By fostering an environment where both scientists and software engineers can learn from each other and work together effectively, practitioners can create more robust and reliable SSW.**

In summary, the integration of diverse knowledge domains, systematic management of Scientific Debt, and the effective use of advanced tools like LLMs are crucial for the successful development of SSW. Practitioners should focus on creating interdisciplinary teams, implementing structured processes for managing TD, and continuously improving their skills and knowledge to adapt to the evolving needs of SSW development.

5.4 Threats to Validity

There are several threats to the validity of the work presented in this thesis. In this section, we summarize the dangers and also present the steps taken to mitigate them.

5.4.1 Internal Validity

The internal validity of our study may be influenced by several factors related to the methods and tools used in the research process. One major limitation is the potential bias introduced through the manual labeling of SATD comments. Due to the high rate of false positives from automated tools and the necessity to identify previously unseen types of SATD, we chose to manually label the comments. This manual process, while thorough, is inherently subject to the biases and knowledge level of the labelers. To mitigate this, we involved multiple labelers and resolved disagreements through discussion, ensuring a more balanced and accurate labeling process.

Furthermore, the diversity of programming languages used in the analyzed projects could impact our results. While our study includes projects written in languages like Python, Fortran, and C++, the specific characteristics and challenges associated with each language might influence the nature of SATD and its resolution. Fortran, for example, is known for its extensive use in legacy SSW, which may present different TD issues compared to more modern languages. This linguistic diversity necessitates caution when generalizing our findings across all SSW projects.

5.4.2 External Validity

The scientists and projects chosen for the study might not be representative of the broader population of SSW developers due to several reasons. Firstly, the selection of projects was limited to those with publicly available source code and active repositories, which may not fully capture the diversity of all SSW projects. This limitation could impact the generalizability of our findings, as the practices, challenges, and TD issues observed in these projects might differ from those in less prominent or less active or private projects.

Furthermore, the specific scientific domains studied—such as astronomy, climate modeling, molecular dynamics, and others—might have unique characteristics and challenges that are not applicable to other scientific fields or types of software development. Each domain has its own set of scientific principles, computational requirements, and development practices, which can influence the nature and resolution of TD. Consequently, findings related to SATD in these domains may not directly translate to other fields with different computational and scientific needs.

To address these limitations and enhance the generalizability of the results, future research should include a wider range of scientific fields and project types. Expanding the scope of study to encompass diverse scientific disciplines and varying levels of project activity will provide a more comprehensive understanding of SATD in SSW development. Additionally, incorporating projects with different levels of maturity, from nascent to well-established, can offer insights into how TD evolves and is managed across the software lifecycle.

5.4.3 Construct Validity

The categorization of scientific comments is based on five indicators mentioned in 4.2. Since none of the labelers are experts in these scientific domains, there is a risk that some scientific debt SATDs may be inaccurately categorized. To address this, we tried our best to communicate with the contributors of the studied projects. By seeking their insights and clarifications, we aimed to enhance the accuracy of our categorizations and ensure that the context and content of the comments were correctly understood.

The operational definitions of SATD and the roles of scientists may vary, leading to inconsistencies in how these constructs are interpreted. We established clear and precise definitions of what each type of SATD means within the scope of this study

and applied these definitions consistently. This approach aimed to reduce ambiguity and enhance the reliability of our interpretations.

5.4.4 Reliability

Data was collected using the Python library gitpython, which means any information hosted in other version control systems, such as SVN (used by the CESM project), was not recorded. This limitation was acknowledged, and efforts were made to use the most comprehensive data available. Future research should consider integrating data from multiple version control systems to provide a more complete picture.

5.5 Summary

In this chapter, we explored the multifaceted nature of knowledge acquisition in SSW development and its implications for managing SATD. We made use of Kelly's five key knowledge domains essential for effective contributions to SSW projects: Real World Knowledge, Theory-Based Knowledge, Software Knowledge, Execution Knowledge, and Operational Knowledge. Through dynamic knowledge profiles visualized using spider charts, we demonstrated how contributors from different backgrounds adapt and grow in these domains over the course of a project.

We also delved into the comparative ease of acquiring different knowledge domains, raising the critical question of whether it is easier for scientists to gain software engineering skills or for software developers to attain deep scientific knowledge. This analysis highlighted the inherent complexity of SSW projects and provided insights into the factors influencing the accumulation and resolution of SATD.

Furthermore, we discussed the practical implications of our findings for SSW development. We emphasized the importance of comprehensive documentation practices, interdisciplinary collaboration, and focused training and skill development to manage TD effectively. We also highlighted the need for further research into automated tools and techniques tailored for SSW, the impact of project characteristics on SATD management, and educational strategies for TD management.

Chapter 6

Conclusion

This thesis explored the largely uncharted territory of SATD within the domain of SSW, introducing a novel category that we have termed Scientific Debt. This concept is crucial to understanding the unique challenges posed by the development and maintenance of software that is intricately tied to scientific inquiry. Through an extensive analysis of 28,680 code comments drawn from nine diverse, open-source SSW projects, we have shed light on how Scientific Debt manifests, evolves, and often persists within these projects.

The findings from our study reveal several critical challenges associated with Scientific Debt. These include the pervasive use of assumptions, the handling of missing edge cases, the occurrence of computational inaccuracies, difficulties in translating complex scientific concepts into computational models, and the ongoing challenge of integrating new scientific findings into existing software frameworks. These challenges are not merely technical; they strike at the core of the reliability and validity of scientific results, underscoring the importance of addressing Scientific Debt with the same rigor as any other aspect of scientific research.

Our analysis also highlighted a concerning trend: the rate at which Scientific Debt is addressed lags significantly behind its rate of introduction. While some types of debt, such as those arising from new scientific findings, are more frequently resolved—likely due to their direct impact on the accuracy and relevance of the research—other types, particularly those related to translation challenges and assumptions, are often neglected. This discrepancy suggests that while the scientific community is adept at responding to immediate and obvious issues, more subtle and long-term problems may be left unresolved, potentially compromising the integrity of the software over time.

One of the most promising aspects of our research is the exploration of LLMs, such as ChatGPT, in the context of managing Scientific Debt. Our study demonstrates that LLMs have significant potential in identifying issues that could impact scientific outcomes, offering a powerful tool for the early detection of problems that might otherwise go unnoticed. The integration of such advanced tools into the development workflow could greatly enhance the reliability, maintainability, and overall quality of SSW, thereby contributing to more robust and reproducible scientific discoveries.

In conclusion, this thesis emphasizes the critical need for tailored strategies and advanced tools to effectively manage Scientific Debt. Addressing this debt is not just about maintaining code quality; it is about safeguarding the integrity of scientific research itself. As SSW becomes increasingly central to the pursuit of knowledge, the ability to manage its complexities, and particularly its debt, will be vital in ensuring that scientific outcomes are reliable, accurate, and impactful.

Bibliography

[1] Nicolli Souza Rios Alves, Leilane Ferreira Ribeiro, Vivyane Caires, Thiago Souto Mendes, and Rodrigo Oliveira Spínola. Towards an ontology of terms on technical debt. *2014 Sixth International Workshop on Managing Technical Debt*, pages 1–7, 2014. URL: https://api.semanticscholar.org/CorpusID:9524394.

[2] Adelaide Anim-Annor, Fredrick Boafo, and Solomon Mensah. An nlp approach for identification and detection of self-admitted technical debt: A review of existing techniques. *International Journal of Computer Applications*, 2023. URL: https://api.semanticscholar.org/CorpusID:261578427.

[3] Dorian C. Arnold and Jack J. Dongarra. Developing an architecture to support the implementation and development of scientific computing applications. In *The Architecture of Scientific Software*, 2000. URL: https://api.semanticscholar.org/CorpusID:1559453.

[4] Markus Borg, Ilyana Pruvost, Enys Mones, and Adam Tornhill. Increasing, not diminishing: Investigating the returns of highly maintainable code. *ArXiv*, abs/2401.13407, 2024. URL: https://api.semanticscholar.org/CorpusID:267200033.

[5] Jeffrey C. Carver, Richard P. Kendall, Susan E. Squires, and Douglass E. Post. Software development environments for scientific and engineering software: A series of case studies. *29th International Conference on Software Engineering (ICSE'07)*, pages 550–559, 2007. URL: https://api.semanticscholar.org/CorpusID:5957260.

[6] Cheng-Han Chiang and Hung yi Lee. Can large language models be an alternative to human evaluations?, 2023. arXiv:2305.01937.

[7] Zadia Codabux and Byron J. Williams. Managing technical debt: An industrial case study. *2013 4th International Workshop on Managing Technical Debt (MTD)*, pages 8–15, 2013. URL: https://api.semanticscholar.org/CorpusID:18432758.

[8] N. Cote. An exploration of a testing strategy to support refactoring. Master's thesis, Royal Military College of Canada, Kingston, Canada, 2005.

[9] Ward Cunningham. The wycash portfolio management system. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1992. URL: https://api.semanticscholar.org/CorpusID:207173958.

[10] Everton da S. Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43:1044–1062, 2017. URL: https://api.semanticscholar.org/CorpusID:10396446.

[11] Daniela Damian, Remko Helms, Irwin Kwan, Sabrina Marczak, and Benjamin Koelewijn. The role of domain knowledge and cross-functional communication in socio-technical coordination. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, May 2013. URL: http://dx.doi.org/10.1109/ICSE.2013.6606590, doi:10.1109/icse.2013.6606590.

[12] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015. URL: https://api.semanticscholar.org/CorpusID:1175911.

[13] Hongbo Fang, James Herbsleb, and Bogdan Vasilescu. Matching skills, past collaboration, and limited competition: Modeling when open-source projects attract contributors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, November 2023. URL: http://dx.doi.org/10.1145/3611643.3616282, doi:10.1145/3611643.3616282.

[14] Nicol'as E. D'iaz Ferreyra, Mojtaba Shahin, Mansorreh Zahedi, Sodiq Quadri, and Ricardo Scandariato. What can self-admitted technical debt tell us about security? a mixed-methods study. *ArXiv*, abs/2401.12768, 2024. URL: https://api.semanticscholar.org/CorpusID:267094974.

[15] Jose Freitas, Daniela Cruz, and Pedro Rangel Henriques. A comment analysis approach for program comprehension. In *Proceedings of the 2012 IEEE 35th Software Engineering Workshop, SEW 2012*, pages 11–20, 10 2012. doi:10.1109/SEW.2012.8.

[16] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 30:1 – 56, 2021. URL: https://api.semanticscholar.org/CorpusID:236898597.

[17] Jo Erskine Hannay, Hans Petter Langtangen, Carolyn MacLeod, Dietmar Pfahl, Janice Singer, and Greg Wilson. How do scientists develop and use scientific software? *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 1–8, 2009. URL: https://api.semanticscholar.org/CorpusID:1571389.

[18] Dan Van Hook and Diane Kelly. Mutation sensitivity testing. *Computing in Science & Engineering*, 11:40–47, 2009. URL: https://api.semanticscholar.org/CorpusID:15562124.

[19] Dan Van Hook and Diane Kelly. Testing for trustworthiness in scientific software. *2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, pages 59–64, 2009. URL: https://api.semanticscholar.org/CorpusID:14656577.

[20] Diane Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24, 2007. URL: https://api.semanticscholar.org/CorpusID:19684918.

[21] Diane Kelly. Determining factors that affect long-term evolution in scientific application software. *J. Syst. Softw.*, 82:851–861, 2009. URL: https://api.semanticscholar.org/CorpusID:206525096.

[22] Diane Kelly. An analysis of process characteristics for developing scientific software. *J. Organ. End User Comput.*, 23:64–79, 2011. URL: https://api.semanticscholar.org/CorpusID:15219595.

[23] Diane Kelly. Industrial scientific software: a set of interviews on software development. In *Conference of the Centre for Advanced Studies on Collaborative Research*, 2013. URL: `https://api.semanticscholar.org/CorpusID:32187509`.

[24] Diane Kelly. Scientific software development viewed as knowledge acquisition: Towards understanding the development of risk-averse scientific software. *J. Syst. Softw.*, 109:50–61, 2015. URL: `https://api.semanticscholar.org/CorpusID:42345527`.

[25] Diane Kelly and Rebecca Sanders. Assessing the quality of scientific software. In *First International Workshop on Software Engineering for Computational Science and Engineering*, Leipzig, Germany, 2008.

[26] Diane Kelly and Rebecca Sanders. The challenge of testing scientific software. In *Proceedings of the Annual Conference of the Association for Software Testing*, pages 30–36, 2008.

[27] Diane Kelly, Spencer Smith, and Nicholas Meng. Software engineering for scientists. *Comput. Sci. Eng.*, 13:7–11, 2011. URL: `https://api.semanticscholar.org/CorpusID:206457331`.

[28] Diane Kelly, S. Thorsteinson, and Dan Van Hook. Scientific software testing: Analysis with four dimensions. *IEEE Software*, 28:84–90, 2011. URL: `https://api.semanticscholar.org/CorpusID:15746610`.

[29] Bojana Koteska, Anastas Mishev, and Ljupco Pejov. Quantitative measurement of scientific software quality: Definition of a novel quality model. *International Journal of Software Engineering and Knowledge Engineering*, 28(03):407–425, 2018. URL: `https://doi.org/10.1142/S0218194018500146`.

[30] Philippe B Kruchten, Robert L. Nord, and Ipek Ozkaya. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29:18–21, 2012. URL: `https://api.semanticscholar.org/CorpusID:10673584`.

[31] Valentina Lenarduzzi, Terese Besker, Davide Taibi, A. Martini, and Francesca Arcelli Fontana. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *J. Syst. Softw.*, 171:110827, 2021. URL: `https://api.semanticscholar.org/CorpusID:225128141`.

[32] Yikun Li, Mohamed Soliman, and Paris Avgeriou. Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering*, 28:1–38, 2022. URL: `https://api.semanticscholar.org/CorpusID:246634317`.

[33] Yikun Li, Mohamed Soliman, and Paris Avgeriou. Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empirical Software Engineering*, 27, 2022. URL: `https://api.semanticscholar.org/CorpusID:246607816`.

[34] Yikun Li, Mohamed Soliman, Paris Avgeriou, and Lou Lou Somers. Self-admitted technical debt in the embedded systems industry: An exploratory case study. *IEEE Transactions on Software Engineering*, 49:2545–2565, 2022. URL: `https://api.semanticscholar.org/CorpusID:249152204`.

[35] Erin Lim, Nitin Taksande, and Carolyn Budinger Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Softw.*, 29:22–27, 2012. URL: `https://api.semanticscholar.org/CorpusID:35053699`.

[36] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, D. Lo, and Shanping Li. Is using deep learning frameworks free? characterizing technical debt in deep learning frameworks. *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, pages 1–10, 2020. URL: `https://api.semanticscholar.org/CorpusID:211839121`.

[37] Jiakun Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. An exploratory study on the introduction and removal of different types of technical debt in deep learning frameworks. *Empirical Software Engineering*, 26(2):16, February 2021. `doi:10.1007/s10664-020-09917-5`.

[38] Rungroj Maipradit, Christoph Treude, Hideaki Hata, and Ken ichi Matsumoto. Wait for it: identifying “on-hold” self-admitted technical debt. *Empirical Software Engineering*, 25:3770 – 3798, 2019. URL: `https://api.semanticscholar.org/CorpusID:59316858`.

[39] Nicholas Meng, Diane Kelly, and Thomas R. Dean. Towards the profiling of scientific software for accuracy. In *Conference of the Centre for Advanced Studies*

on Collaborative Research, 2011. URL: https://api.semanticscholar.org/CorpusID:37510086.

[40] Greg Miller. A scientist's nightmare: Software problem leads to five retractions. *Science*, 314:1856 – 1857, 2006. URL: https://api.semanticscholar.org/CorpusID:34054520.

[41] Ali Niknafs and Daniel Berry. The impact of domain knowledge on the effectiveness of requirements engineering activities. *Empirical Software Engineering*, 22(1):80–133, April 2016. URL: http://dx.doi.org/10.1007/s10664-015-9416-2, doi:10.1007/s10664-015-9416-2.

[42] Gustavo Henrique Lima Pinto, Igor Scaliante Wiese, and Luiz Felipe Dias. How do scientists develop scientific software? an external replication. *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 582–591, 2018. URL: https://api.semanticscholar.org/CorpusID:4618327.

[43] Aniket Potdar and Emad Shihab. An exploratory study on self-admitted technical debt. *Proceedings - 30th International Conference on Software Maintenance and Evolution, ICSME 2014*, pages 91–100, 12 2014. doi:10.1109/ICSME.2014.31.

[44] Yubin Qu, Tie Bao, Xiang Chen, Long Li, Xianzheng Dou, Meng Yuan, and Hongmei Wang. Do we need to pay technical debt in blockchain software systems? *Connection Science*, 34:2026 – 2047, 2022. URL: https://api.semanticscholar.org/CorpusID:250240932.

[45] Leevi Rantala, Valentina Lenarduzzi, and Mika Mäntylä. Relationship between self-admitted technical debt and code-level technical debt. an empirical evaluation. *Software Quality Journal*, 10 2023.

[46] Judith Segal. When software engineers met research scientists: A case study. *Empirical Software Engineering*, 10:517–536, 2005. URL: https://api.semanticscholar.org/CorpusID:18920735.

[47] Judith Segal. Scientists and software engineers: A tale of two cultures. In *Annual Workshop of the Psychology of Programming Interest Group*, 2008. URL: https://api.semanticscholar.org/CorpusID:7271135.

[48] Rishab Sharma, Ramin Shahbazi, Fatemeh Hendijani Fard, Zadia Codabux, and Melina C. Vidoni. Self-admitted technical debt in r: detection and causes. *Automated Software Engineering*, 29, 2022. URL: `https://api.semanticscholar.org/CorpusID:251849837`.

[49] Helen Sharp. The role of domain knowledge in software design. *Behaviour & Information Technology*, 10(5):383–401, September 1991. URL: `http://dx.doi.org/10.1080/01449299108924298`, `doi:10.1080/01449299108924298`.

[50] Emmanuel Simon, Melina C. Vidoni, and Fatemeh Hendijani Fard. Algorithm debt: Challenges and future paths. *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*, pages 90–91, 2023. URL: `https://api.semanticscholar.org/CorpusID:259338084`.

[51] Donna Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[52] Murali Sridharan, Leevi Rantala, and Mika Mäntylä. Pentacet data - 23 million contextual code comments and 250,000 satd comments. *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 412–416, 2023. URL: `https://api.semanticscholar.org/CorpusID:257757331`.

[53] Margaret-Anne Storey, Jody Ryall, R. Bull, Del Myers, and Janice Singer. Todo or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proceedings - International Conference on Software Engineering*, pages 251–260, 01 2008. `doi:10.1145/1368088.1368123`.

[54] Roberto Verdecchia, Philippe B Kruchten, and Patricia Lago. Architectural technical debt: A grounded theory. In *European Conference on Software Architecture*, 2020. URL: `https://api.semanticscholar.org/CorpusID:221618033`.

[55] Melina C. Vidoni. Self-admitted technical debt in r packages: An exploratory study. *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 179–189, 2021. URL: `https://api.semanticscholar.org/CorpusID:236185007`.

[56] Melina C. Vidoni and Maria Laura Cunico. On technical debt in mathematical programming: An exploratory study. *Mathematical Programming Computa-*

tion, 14:781 – 818, 2022. URL: https://api.semanticscholar.org/CorpusID:251424489.

[57] G. J. Wilson. Where's the real bottleneck in scientific computing? *American Scientist*, 94:5, 2006. URL: https://api.semanticscholar.org/CorpusID:122875373.

[58] Changrong Xiao, Sean Xin Xu, Kunpeng Zhang, Yufang Wang, and Lei Xia. Evaluating reading comprehension exercises generated by LLMs: A showcase of ChatGPT in education applications. In Ekaterina Kochmar, Jill Burstein, Andrea Horbach, Ronja Laarmann-Quante, Nitin Madnani, Anaïs Tack, Victoria Yaneva, Zheng Yuan, and Torsten Zesch, editors, *Proceedings of the 18th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2023)*, pages 610–625, Toronto, Canada, July 2023. Association for Computational Linguistics. URL: https://aclanthology.org/2023.bea-1.52, doi:10.18653/v1/2023.bea-1.52.

[59] Jesse Yli-Huumo, Andrey Maglyas, and Kari Smolander. How do software development teams manage technical debt? - an empirical study. *J. Syst. Softw.*, 120:195–218, 2016. URL: https://api.semanticscholar.org/CorpusID:10984263.

[60] Zhe Yu, Fahmid Morshed Fahid, Huy Tu, and Tim Menzies. Identifying self-admitted technical debts with jitterbug: A two-step approach. *IEEE Transactions on Software Engineering*, 48(5):1676–1691, 2022. doi:10.1109/TSE.2020.3031401.