Supporting the Sustainable Use of Open Source Software

Courtney Elta Miller

CMU-S3D-26-105
April 2026

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Dr. Christian Kästner, Co-Chair
Dr. Bogdan Vasilescu, Co-Chair
Dr. Jim Herbsleb
Dr. Laurie Williams (North Carolina State University)
Dr. Thomas Zimmerman (University of California, Irvine)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

Copyright © 2026 **Courtney Elta Miller**

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant Numbers DGE1745016 and DGE2140739, and the National Science Foundation SDI-CPR: Sustaining Digital Infrastructure as a Common Pool Resource under Grant Number IIS1901311, and a National Science Foundation (NSF) Secure and Trustworthy Cyberspace (SaTC) Frontiers award titled "Collaborative: SaTC: Frontiers: Enabling a Secure and Trustworthy Software Supply Chain" under Grant Number CNS2206859. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

April 2, 2026
DRAFT

Keywords: Empirical Software Engineering, Human Factors in Software Engineering, Dependency Management, Open Source Software, Software Supply Chain Security, LLM Tools for Software Development

For my dog, Chanel.

April 2, 2026
DRAFT

Abstract

April 2, 2026
DRAFT

In this dissertation, I study how to support and improve the processes used by developers facing open source dependency abandonment disruptions. Open source software forms the digital infrastructure that most modern software is built on, and expectations regarding ongoing maintenance are a widespread norm despite the reality that many open source packages become abandoned, even widely-used ones. Package abandonment can disrupt supply chain integrity for millions of downstream users and increase software supply chain attack surfaces. These disruptions threaten the stability of critical digital infrastructure, including healthcare systems, financial services, and transportation networks. Addressing these disruptions is essential for ensuring software reliability, security, and the resilience of our broader digital economy. While this discrepancy between the expectations placed on and the reality of open source has fueled the need to study and improve open source sustainability, the majority of that research has focused on attempting to *prevent* abandonment disruptions.

My dissertation takes a different approach: I shift the focus away from maintainers and study abandonment disruptions from the user perspective with the goal of enabling the sustainable use of open source digital infrastructure by helping users better navigate abandonment disruptions when they occur. I leverage a three-step methodological approach to explore, measure, and improve how developers navigate abandonment disruptions. Throughout these three steps, representing the three core chapters of this dissertation, I develop rigorous multi-dimensional empirical mixed-method approaches, combining human-centered qualitative techniques with large-scale data-driven statistical analysis, modeling, and visualization. By systematically studying abandonment as a disruption, I reveal developer challenges and inform the design of targeted solutions. More specifically, I begin by understanding abandonment disruptions through the first two steps. First, I explore how developers currently deal with abandonment disruptions by going straight to the source and interviewing developers who have faced open source dependency abandonment. I contextualize and curate their experiences, how they deal with abandonment, and both the key challenges they face as well as potential solutions. Additionally, I present a theoretical framework on the cost of dependency abandonment, introduce the concept of community-oriented solutions, and provide evidence-based strategies from fields like social psychology and game theory to overcome the volunteer's dilemma and encourage collective responses. Then, to measure the scale of abandonment disruptions and the current state of user response in practice, I perform a large-scale quantitative analysis measuring the prevalence of and response to abandonment across the JavaScript npm ecosystem. I employ a series of statistical modeling techniques to quantify the impact of various factors on the likelihood and speed of downstream user response to abandonment, such as providing explicit notice to users of the abandonment. Revealing that while downstream response is uncommon, increasing information transparency surrounding abandonment can help support and accelerate downstream responses, aligning with the challenges surrounding identifying abandonment described by developers in the first explore step. Through the first two steps, I demonstrate that abandonment is an under-supported and understudied disruption that many developers struggle both to identify and respond to, given the current lack of tooling and guidance.

In the third step, I help improve how developers navigate abandonment disruptions by designing an intervention in the form of a prototype tool for automatically identifying dependency abandonment. However, I also learned that learned that not all dependency abandonment is equally concerning to developers; instead, many are primarily concerned about abandonment they believe would be *impactful* to their project. And research on existing

April 2, 2026
DRAFT

software component analysis (SCA) tools for related dependency management practices e.g., updates and vulnerability patches, illustrates that a key usability issue limiting the effectiveness of these tools is overwhelming developers with too many notifications, particularly ones deemed irrelevant, which can frustrate developers and lead to tool disengagement. With this key limitation in mind, I developed a prototype tool to support the automated identification of abandoned dependencies without overwhelming developers, by only notifying developers about dependency abandonment that is likely impactful and therefore noteworthy to their project. My key questions became (1) what abandonment *will* be impactful to a particular project given the context of their dependency usage; and (2) how to make such predictions at scale. I hypothesize and later demonstrate that our approach using large language models (LLMs), equipped with theory-driven reasoning and context-specific information, can accurately predict the impact of abandonment better than LLMs alone to support developer decision making. I conducted need-finding interviews with 22 developers to develop a theoretical understanding of what factors influence the impactfulness of a dependency's abandonment on a project given the context of their dependency usage. I then leveraged this theory to develop an LLM-based classifier to predict the project-specific impact of abandonment using theory-driven reasoning and context-specific information. Finally, through an independent evaluation with 124 developers, I demonstrate that our classifier is effective at predicting project-specific impactfulness as well as the promise of this method for creating tooling with intelligent defaults in contexts where traditional tooling approaches have failed and theory or design work is still essential.

My dissertation work focuses on helping developers navigate abandonment disruptions, but my three step explore-measure-improve approach to studying disruptions has much wider applicability. For example, I will study other sociotechnical disruptions, such as the integration of Generative AI tools into development workflows, which I am currently working on, as well as the unanticipated disruptions of tomorrow.

April 2, 2026
DRAFT

April 2, 2026
DRAFT

Acknowledgments

I would like to start these acknowledgments by saying, there is no way I am going to get it right. There are so many amazing people who helped me along the way that despite my best efforts, I know that this is not a comprehensive list.

First I would like to thank my advisors, Bogdan Vasilescu and Christian Kästner, aka the dream team, for everything they have done to support me on this journey and make this body of work possible. Thank you Bogdan for taking a chance on me back in 2018, for being my first academic research mentor, and for always challenging my thinking in ways that made the work sharper, even when I didn't want to hear it. I have learned a lot from you about research, marketing, networking, and of course, doing sociotechnical research using data excavation. Thank you Christian for being a constant supportive presence, for always having an open door, for responding to feedback requests with exceptional speed, and for teaching me everything you have. I am grateful for the many hours spent in your office discussing ideas, and for the example you have set for me, in terms of the type of advisor I aim to be (and let me tell you, the bar is high!). Thank you for teaching me how to zoom out, think critically, and write a good discussion section. While my PhD is coming to a close, you will both be my mentors forever, and please do expect me to continue bothering you for help and guidance.

Thank you to my committee members Jim Herbsleb, Thomas Zimmerman, and Laurie Williams for the ongoing encouragement, feedback, and wisdom that has enriched and strengthened this dissertation. Thank you Jim for broadening my horizons especially in open source and economics, it has been an honor to be able to learn from you. Thank you Tom for being such a kind, effective, and present mentor, and thank you for believing in me enough to give me my first industry research internship. 2020 was a very scary, strange year but the work I did under your guidance was a highlight for me, and I am so grateful that I've continued to have the opportunity to learn and get guidance from you since then. Thank you Laurie for being my mentor, for teaching me so much, for giving me countless opportunities to learn, grow, and collaborate with amazing researchers and practitioners. I am not just saying this because you are on my committee and I want to pass, I mean this from the bottom of my heart: I believe you are the single most hard-working, dedicated, and impactful researcher I know. Thank you for giving me the opportunity to be a part of the S3C2, I have learned so much from watching you lead and you have demonstrated such a high caliber of research, communication, and outreach excellence which I aspire to.

Thank you Margaret-Anne Storey for being my long-time mentor, collaborator, cheerleader, and source of inspiration. At every turn, you have been there with a helping hand ready. You have been a supportive figure through a remarkable number of key milestones in my life: my first Microsoft internship, my undergraduate dissertation, PhD applications, my second Microsoft internship, my PhD dissertation, faculty job search, and figuring out what's next for me in the great beyond. I continue to be incredibly grateful to call you a friend and mentor. Thanks for everything, Peggy.

I am also grateful to the many researchers beyond CMU who provided guidance over the years: West Weimer, Brian Houck, Ben Hanrahan, Travis Lowdermilk, Eirini Nathan, Nancy Baym, Syboney Biwa.

I would like to thank my professors in S3D– Prof. Michael Hilton, Prof. Josh Sunshine, Prof. Rohan Padhye, Prof. Andrew Begel, Prof. Claire Le Goues, Prof. Patrick Park, who gave me good advice constantly and the encouragement I needed to keep pushing through. Thank you Michael for teaching me how to teach, for being such a passionate innovative inspiring instructor, and for always being there for me. Thank you Josh for being an incredible

April 2, 2026
DRAFT

REUSE mentor, for the herculean ongoing dedication you have to the REUSE program which has a significant transformational impact on so many students lives, including my own, and for continuing to be a friend, supporter, and mentor over the years. Thank you Rohan for the advice to “think like a professor” it might sound silly in retrospect but it seriously shaped how I approached the final year of my PhD and helped me grow a lot.

Thank you to my lab mates both past and present. Hao He, Shyam Agarwal, Yining Hong, Chenyang Yang, Nadia Nahar, Hongbo Fang, Shurui Zhou, Daye Nam, Sophie Qiu, I have had so much fun working with and learning from you all. I would also like to thank my PhD mentors from my own time in the REUSE program, who helped me along the way when I myself was freshly green, David Widder and Nimo Wode, you showed me what a good PhD student mentor looks like and significantly impacted my own mentorship approach. Thank you to all my cohortmates and fellow students in S3D that made my time here so fun: Jenny Liang, Sophia Kolak, Adian Yang, Samantha Phillips, Evan Williams, Parv Kapoor, Manisha Mukherjee, Luis Fernandes-Gomes, Peter Carragher, Lynnette Ng, Luke Dramko, Kaia Newman, Eleanor Young, Tori Qiu, Samsara Foubert, and many others. Thank you to the best office mates a girl could ask for: Reba Marigliano, Daniele Bellutta, and Janice Blane. You have made the office a place I look forward to going to every day, and I will miss you guys and the spectacular holiday door decorations we built together every year.

Thank you to my many amazing collaborators over the years: Rudrajit Choudhuri, Mara Ulloa, Sankeerti Haniyur, Robert DeLine, Emerson Murphy-Hill, Christian Bird, Jenna L. Butler, Elizabeth Lin, Paige Rodeghero, Denae Ford, Mahmoud Jahanshahi, Audris Mockus, Giacomo Benedetti, Greg Tystahl, William Enck, Alexandros Kapravelos, Alessio Merlo, Luca Verderame, Lina Boughton, Yasemin Acar, Dominik Wermke, Sophie Cohen, Daniel Klug, and Jeffrey Chen. Thank you Jeffrey for building Abandabot, we literally could not have done the essential need-finding interviews without you and the high-fidelity prototype you developed. Thank you to all those who participated in this research, thank you for your candor, wisdom, and valuable time.

Thank you to the twelve undergraduate mentees whose interests and curiosity helped drive my own: Kimberly Truong, Oreofe Solarin, Philip Gray, Theresa Lim, Tak-Ho Lee, Lina Boughton, Kavan Mehrizi, Emily Nguyen, Joshua Delarosa, Rae Suarez, Aiden Amaya, and Katrina Wilson. Watching you all grow and evolve into your best selves has been one of the biggest joys of my career thus far and I cannot wait to continue cheering you on.

Thank you to the incredible administrative staff who have shielded me from the complexities of reality and taught me so much: Connie Herold, Dabney Schlea, Victoria Poprocky, Alisha Roudebush, Jennifer Cooper, Cole Jester, Tom Pope, Buck Caldwell, Jim Tobin, and the many more whose work is invisible to me though no less valuable. Thank you, thank you, thank you.

I would also like to thank my family for supporting me in a million different ways. Thank you mom for always being there, as a child I know I was a handful and I have certainly not become less opinionated with age, but you have always made me feel like I am enough, and that I am capable of doing whatever ambitious feats I set my mind to, because I know you will always be there for me no matter what happens. Thank you Andrew for being an incredibly supportive partner. Seriously, without your help I simply would have not been able to have made half the deadlines I made to get to this point. As I write this Chanel is laying on my desk and I know she thanks you too, by the way.

And now finally, I must thank my dog, Chanel . Chanel, you have been by my side since

April 2, 2026
DRAFT

I was 13, you are the “divorce puppy” that became the constant in my life, no matter where I went. We have been through middle school, high school, college, and now grad school together. Your social butterfly nature, love of life, curiosity, kindness, intelligence, and joy continue to inspire and amaze me. Thank you for always being by my side, through the highest highs and the lowest lows of my life, you have been there. Your support and presence continues to propel me forward, and I am grateful for the opportunity to learn something new from you every day. I lack the vocabulary to fully express how much I love you, but just know it knows no bounds and grows every day.

April 2, 2026
DRAFT

Contents

1 Introduction **1**
1.1 The Inevitability of Dependency Abandonment 1
1.2 Supporting Sustainable Usage 2
1.2.1 Identifying the Unsupported Challenges of Dependency Abandonment 3
1.2.2 Quantifying the Prevalence of and Response to Dependency Abandonment at Scale 3
1.2.3 Triangulating the Impact of Information Transparency on User Response to Dependency Abandonment 4
1.2.4 Understanding What Makes Abandonment Impactful 4
1.2.5 Intervention: Automatically Identifying Impactful Abandonment At Scale 5
1.3 Thesis Statement 6
1.4 Contributions 6

2 Background and Related Work **7**
2.1 Dependency Management 7
2.2 Supporting Downstream Decision Making with Signaling Theory 9
2.3 Open Source Sustainability 10
2.4 Package Abandonment as a Supply Chain Security Risk 11

3 Navigating Dependency Abandonment **13**
3.1 Introduction 13
3.2 Research Design 14
3.2.1 Identifying and Recruiting Participants 14
3.2.2 Interview Protocol 15
3.2.3 Data Collection and Analysis 16
3.2.4 Validity Check 16
3.2.5 Limitations 16
3.3 Impacts of Abandonment 17
3.4 Identifying Abandonment 18
3.5 Preparing for and Addressing Abandonment 19
3.5.1 Considerations Before Adoption 19
3.5.2 Preparations Once Adopted 20
3.5.3 Solutions to Abandonment 21
3.6 Discussion: Towards More Sustainable Use of Open Source 23
3.6.1 The Cost of Dependency Abandonment 23
3.6.2 Aspirational Cost Reduction Strategies 23
3.6.3 The Volunteer's Dilemma and Reducing Community Effort 25

April 2, 2026
DRAFT

3.7 Summary . . . 27
3.8 Data Availability . . . 27

4 Quantifying Prevalence of and Response to Abandonment At Scale 29
4.1 Introduction . . . 29
4.2 Detecting Open-Source Package Abandonment . . . 30
4.2.1 Explicit-Notice Abandonment . . . 31
4.2.2 Activity-Based Abandonment . . . 31
4.3 RQ1: Abandonment Prevalence and Exposure . . . 32
4.3.1 Research Methods . . . 32
4.3.2 Results . . . 34
4.4 RQ2: Responding to Abandonment . . . 35
4.4.1 Research Methods . . . 35
4.4.2 Results . . . 37
4.5 RQ3: Characterizing Responsive Dependents . . . 37
4.5.1 Research Methods . . . 37
4.5.2 Model Results . . . 39
4.6 RQ4: Influence of Announcing Abandonment . . . 40
4.6.1 Research Methods . . . 40
4.6.2 Results . . . 40
4.7 Discussion and Implications . . . 41
4.8 Summary . . . 43
4.9 Data Availability . . . 43

5 Identifying Impactful Dependency Abandonment 45
5.1 Introduction . . . 45
5.2 Need-Finding Interviews . . . 47
5.2.1 Research Design . . . 47
5.2.2 RQ1 Results - What Influences the Importance of Abandonment . . . 50
5.2.3 RQ2 Results - Tool Design Requirements and Information Needs . . . 52
5.3 Abandabot-Predict: Predicting Impactful Dependency Abandonment . . . 53
5.3.1 Approach . . . 54
5.3.2 Implementation . . . 55
5.3.3 Offline Evaluation . . . 56
5.3.4 Independent Evaluation Study . . . 57
5.3.5 RQ3 Results . . . 59
5.3.6 Limitations and Threats to Validity . . . 60
5.4 Discussion and Implications . . . 60
5.4.1 Intelligent Tool Pre-Configuration . . . 60
5.4.2 Synergistic Design: Adding Theory to LLMs . . . 61
5.5 Summary . . . 62
5.6 Data Availability . . . 62

6 Discussion and Conclusion 63
6.1 Conclusion . . . 63
6.2 Discussion: Dependency Abandonment in the Agentic GenAI Era . . . 64
6.2.1 Opportunities: How GenAI Could Reduce the Cost of Abandonment . . . 64

April 2, 2026
DRAFT

6.2.2 Risks: How GenAI Threatens Open Source Sustainability 65
6.3 The Broader Landscape and Impacts 67

Bibliography **69**

April 2, 2026
DRAFT

List of Figures

2.1 GitHub issue illustrating the frequent difficulty of identifying dependency abandonment. . 10

3.1 Dependency life cycle with the common stages where abandonment is addressed highlighted. 13
3.2 Research methodology flow chart. 14
3.3 Illustration of the volunteers dilemma for dealing with abandoned dependencies. 25

4.1 Survival curve comparing time to dependency event response. 30
4.2 Overview of our data collection and analysis. 32
4.3 Distribution of widely-used package popularity metrics. 34
4.4 Summary of the multivariate logistic regression model. 39
4.5 Summary of the Cox proportional hazards multivariate survival regression model. 40

5.1 Four categories of context-specific information that affect the impactfulness of dependency abandonment. 46
5.2 Preliminary prototype dashboard tool for formative need-finding interviews. 49
5.3 An example input & output from Abandabot-Predict 54
5.4 The Macro-F1 performance results (average and std. dev. over ten runs) for different LLMs and baselines. 57
5.5 Percentage of judgments where participants agreed or disagreed with the reasoning provided for each category. 59
5.6 Rating distribution of perceived usefulness for each category of context-specific information. 60

April 2, 2026
DRAFT

April 2, 2026
DRAFT

List of Tables

5.1 Mapping RQ1 factors to representative contextual or LLM knowledge 55

April 2, 2026
DRAFT

April 2, 2026
DRAFT

Chapter 1

Introduction

In this dissertation, I study how to support and improve the processes used by developers facing open source dependency abandonment disruptions, thus enabling the sustainable *use* of open source digital infrastructure and the many software supply chains that depend on them. Open source software forms the digital infrastructure that most modern software is built on, and expectations regarding ongoing maintenance are a widespread norm despite the reality that many open source packages become abandoned, even widely-used ones. However, supporting downstream users when they face potential or actual dependency abandonment disruptions is a topic that has been largely neglected by open source sustainability research. To address this research gap, I shift the focus away from maintainers and instead study abandonment disruptions from the user perspective with the goal of enabling the sustainable use of open source digital infrastructure by helping users better navigate abandonment disruptions when they occur.

1.1 The Inevitability of Dependency Abandonment

Over the past 15-odd years, there has been a meteoric rise in the popularity of open source software in large part due to the realization by companies (and everyone else on the internet) that relying on open source is more cost effective and efficient than relying on in-house development or proprietary software licensing (particularly when considering upfront development costs) [34, 67, 122, 156]. Open source has become the digital building blocks that serve as the foundation for most modern software supply chains, allowing developers to transform ideas into prototypes and prototypes into deployed systems in a fraction of the time and at a fraction of the cost previously possible [198, 243, 264]. Today, nearly everything done on screens relies on open source, from checking emails and stock prices to online shopping and telehealth services – inspiring the term *open source digital infrastructure*, which alludes to the fact that open source has become the digital equivalent of the roads and bridges we rely on to get from point A to point B whether we fully realize it or not. Npm, Inc. estimated that 97% of source code in modern web applications came from npm in 2018 [205], and the pervasiveness of open source has only increased since then [249]. Without this digital infrastructure, "*the technology that modern society relies upon simply could not function*." [83]

With this widespread reliance has come widespread expectations surrounding the production and ongoing maintenance of open source. Yet, unlike proprietary software, which typically comes with a licensing agreement providing certain guarantees regarding ongoing software support and maintenance, open source licenses only control the distribution and consumption of software and provide no guarantees regarding production. Despite this, there is a widely-held expectation that the maintainers of open source digital infrastructure are responsible for providing the ongoing support, maintenance, and development

effort necessary to keep the software up to date and to meet user demands [85].

Beyond the expectations for ongoing maintenance not aligning with the licensing guarantees of open source, they also do not align with the reality of how most modern open source packages operate. Today, most open source packages rely on a small number of over-worked and under-appreciated often volunteer maintainers to do the majority of the work [22, 172], and those maintainers often leave for normal reasons that we cannot prevent e.g., switching jobs, a lack of time, or losing interest [182]. Furthermore, when maintainers do disengage, more often than not, nobody else steps up, and the packages becomes fully abandoned [23], making abandonment common even among widely-used packages [187]. This means the ongoing reliability and continued maintenance support of these packages is no sure thing. Widely-used package abandonment can also threaten the software supply chain integrity of millions of downstream users increasing their security risks by exposing them to the risk of unpatched vulnerabilities [289, 293] and targeted supply chain attacks [145, 244, 259, 289].

This tension between our society's widespread dependence on open source and the uncertainty surrounding ongoing maintenance efforts has motivated the need to study and improve open source sustainability. Open source sustainability is a large and vibrant research area. Yet, the majority of the research has thus far focused on studying various characteristics, phenomena, and practices that support the goal of ensuring the ongoing maintenance of particular packages or ecosystems, i.e., trying to *prevent* abandonment disruptions from occurinng.

However, because of the fundamentally self-organized and volunteer-based nature of open source, we likely cannot prevent the abandonment of all open source digital infrastructure. As such, users will always face the risk of abandonment disruptions. And since our economy and society, from multibillion-dollar companies to hospitals and startups, relies on open source to function [83], I argue in this thesis that open source sustainability and supply chain security research must expand its focus to include supporting the *sustainable use* of open source by helping developers better prepare for and address dependency abandonment and its consequences when it occurs. In other words, to echo the suggestions of several open source practitioners [84, 85, 121], instead of attempting to change the nature of open source to match the expectations of users, I suggest supporting a change in the way users engage with open source so they are better equipped to *sustainably use* open source given the risks and realities present in today's landscape.

1.2 Supporting Sustainable Usage

In this dissertation, I leverage a three step methodological approach to *explore, measure, and improve* how developers navigate dependency abandonment disruptions to enable more sustainable use of open source. In the first step I *explore* how developers currently deal with abandonment and the challenges they face. In the second step, I *measure* the prevalence of and response to widely-used package abandonment as well as the impact of increasing information transparency on supporting downstream responses. Finally, in the third step, I help *improve* how developers navigate abandonment disruptions by (1) developing a theoretical framework of what makes abandonment impactful to a downstream user given their usage context; and (2) building an intervention in the form of a prototype tool for automatically identifying impactful dependency abandonment at scale using a theory-driven LLM-based classifier. I will now briefly describe the series of projects, that comprise this dissertation. Henceforth, I use the term "we" throughout this thesis when referring to or describing work that was a product of collaboration with other researchers.

1.2.1 Identifying the Unsupported Challenges of Dependency Abandonment

Since there has been limited sustainability research focused on addressing dependency abandonment when it occurs, we began with an exploratory qualitative semi-structured interview study of 33 developers who have experienced dependency abandonment. The purpose of this study was to *explore* and develop a deeper understanding of the process developers go through when facing dependency abandonment, the challenges they face, and what possible solutions may be.

We found that many developers felt they had little to no resources or guidance when facing abandonment, leaving them to figure out what to do, on their own, through a multi-step trial-and-error process.[1] Migrating to a suitable alternative was a commonly-cited low-effort solution to address abandonment; however, finding a suitable alternative was not always straightforward or possible. In some cases, there were suitable alternatives mentioned in clearly labeled GitHub issues, whereas in other cases, extensive searches yielded no clear results. Furthermore, not all developers believed it was worthwhile to prepare for or even respond to abandonment until a concrete issue occurred. And even among developers who did believe it was worthwhile to respond, most were not equally concerned about all their dependencies' abandonment. Developers often hinted that this distinction had to do with their own usage and reliance on each dependency, with some being a cause for immediate concern due to the potential impact the abandonment could have based on their usage, whereas others were inconsequential.

Additionally, we learned that one of the most time-consuming parts of dependency abandonment was actually identifying abandonment in the first place. Most of the time, developers relied on manual investigations of the dependency's repository, either searching for indications of sufficiently long lulls in activity or for explicit notices of abandonment, like notices at the top of the README. In instances where there was not an explicit notice of abandonment visible, these manual investigations required developers to make sometimes difficult subjective judgments about whether certain lulls in activity were reliable signals indicative of abandonment, e.g., has a package not published a commit in the past year because they are busy working on the next major version or because it is no longer being maintained? This approach that did not scale effectively for many developers, thus creating a bottleneck in the process of dealing with abandonment.

Our findings illustrate that dependency abandonment is an under-studied and under-supported facet of dependency management. We also learned that there is an unmet need for tooling that (1) supports the automated identification of abandonment; and (2) provides actionable guidance on how to respond. However, a key nuance we identified is that not all dependencies and their corresponding abandonment matter equally to developers; suggesting that future work designing such tooling should investigate this nuance further in order to make an effective support tool that does not overwhelm developers with notifications they do not care about.

1.2.2 Quantifying the Prevalence of and Response to Dependency Abandonment at Scale

In the first explore step, we learned that many developers are concerned about dependency abandonment and that some believe in responding right away, at least in certain circumstances, whereas others prefer to wait until a concrete issue occurs. However, we still lack an understanding of how prevalent the issue of dependency abandonment is or how developers respond in practice. Since such an understanding is essential context for any informed plans to provide support through intervention design, we next *measure* the current state of abandonment disruptions in practice through a large-scale quantitative analysis exploring the prevalence of, impact of, and response to the abandonment of widely-used packages in the JavaScript npm ecosystem.

[1] i.e., one interviewee reporting feeling like they were winging it, inspiring the title for this publication

Even with conservative operationalizations, we find that abandonment is common even among widely-used packages, with 15% of widely-used npm packages becoming abandoned during our six-year observation window between January 2015 and December 2020. The prevalence of widely-used package abandonment indicates that users are likely not able to entirely escape abandoned dependencies with careful upfront vetting (which was a commonly reported preparation strategy in the first study), and that they may also need to actively consider strategies to identify and manage abandoned dependencies[2]. Furthermore, while many developers expressed concerns about dependency abandonment, only 18% of exposed projects in our sample removed the abandoned dependency, which suggests some sort of disconnect but is roughly comparable with other dependency management practices, such as installing updates.

1.2.3 Triangulating the Impact of Information Transparency on User Response to Dependency Abandonment

In the first explore step, we learned that one of the biggest bottlenecks in the process of dealing with abandonment is identifying it in the first place. When the maintainers of a package do not provide an explicit notice of abandonment, e.g., a notice at the top of the README, users often rely on manual inspections of the package's repository to identify signs of inactivity. However, developers often reported struggling to make judgments about whether a lull of a particular length of time was actually indicative of abandonment or not. Nonetheless, many developers want to identify abandonment before it causes a concrete problem, so they can react without immediate time pressures. Therefore we hypothesize, following from signaling theory, that increasing information transparency surrounding abandonment may help support and encourage downstream responses by making abandonment more visible and therefore easy to identify.

To investigate this, we use a series of statistical modeling techniques to model the distinction in downstream responses to abandoned packages that provided an explicit notice of abandonment (higher information transparency) compared to packages that silently stopped maintenance (lower information transparency). We found that removal rates are significantly faster when packages provide an explicit notice of abandonment to users, suggesting that awareness matters and that increasing information transparency can help significantly improve downstream response rates to abandonment.

1.2.4 Understanding What Makes Abandonment Impactful

From the first two explore and measure steps, we learned that (1) increasing information transparency surrounding abandonment helps encourage and support user response; (2) there is an unmet need for automated tooling for identifying dependency abandonment; and (3) most developers do not care about the abandonment of all their dependencies equally. Additionally, from usability research on other dependency management tools, we know that when tools provide too many notifications to developers, especially ones deemed incorrect, unimportant, or irrelevant, it can distract, overwhelm, and annoy developers causing information overload and notification fatigue, which often leads to developers ignoring the tool or removing it altogether [116, 191, 256]. Research on overcoming notification fatigue in such circumstances has suggested that only sending relevant notifications to developers can help alleviate the issue [279]. While this may sound like a straightforward revelation in hindsight, within the context of creating tooling to support the automated identification of abandoned dependencies, it leads to the nontrivial question of '*what abandonment will be impactful to a particular project given the context of their dependency usage?*'

[2]Note: This is already a recommended supply chain security best practice [247], yet many developers lack the tooling or support to do so in practice.

April 2, 2026
DRAFT

We help *improve* how developers navigate abandonment disruptions by first performing a series of formative need-finding interviews to (1) develop a theoretical framework of what factors make a dependency's abandonment impactful to a project given the context of their dependency usage; and (2) elicit design requirements and information needs for such a tool in order to support effective usage and downstream response to abandonment using experience prototyping. We found that developers often cite four categories of context-specific information when considering how their usage of a dependency changes the impact of its abandonment on their project: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Fig. 5.1).

1.2.5 Intervention: Automatically Identifying Impactful Abandonment At Scale

Once we developed the theoretical framework, the question became how to leverage it to make predictions at scale. We designed, implemented, and evaluated an LLM-based classifier to predict the project-specific impact of abandonment using theory-driven structured reasoning and context-specific information. We hypothesize and later demonstrate that our approach using large language models (LLMs), equipped with theory-driven reasoning and context-specific information, can accurately predict the impact of abandonment better than LLMs alone to support developer decision making. We combined repository mining and static analysis using CodeQL to extract a body of project-specific usage information that would be sufficient for an expect to make a judgment of abandonment impact. Using the Retrieval-Augmented Generation (RAG) pattern, we embedded this information in a theory-driven reasoning prompt, so an LLM can conduct similar reasoning and judgment steps. Finally, through an independent evaluation with 124 developers, we demonstrated that our classifier is effective at predicting project-specific impactfulness.

April 2, 2026
DRAFT

1.3 Thesis Statement

By shifting the focus of sustainability research from maintainers to users, I explore, measure, and improve how developers navigate open source dependency abandonment disruptions to enable the sustainable *use* of open source digital infrastructure and the software supply chains that depend on it.

I demonstrate that (1) abandonment is an under-supported disruption where developers lack adequate tooling and guidance, (2) increasing information transparency surrounding abandonment significantly accelerates downstream response, and (3) a theory-driven LLM-based classifier equipped with context-specific information can effectively predict project-specific abandonment impact, supporting the automated identification of noteworthy abandonment without overwhelming developers.

1.4 Contributions

My thesis makes a number of contributions toward exploring, measuring, and improving how developers navigate dependency abandonment disruptions in order to support the sustainable use of open source digital infrastructure, including:

- A taxonomy of common strategies used by developers to identify, prepare for, and deal with dependency abandonment as well as the common challenges faced in these processes.
- A theoretical framework for the costs associated with abandonment as well as suggested cost-reduction strategies.
- The concept of community-oriented solutions and evidence-based strategies from fields like social psychology and game theory to overcome the volunteer's dilemma to collectively address abandonment.
- A detailed approach for detecting abandoned packages at scale using both activity-based indicators and explicit notices of abandonment.
- Quantification of the prevalence of widely-used package abandonment in the npm ecosystem and the response to abandonment with a contextualizing comparison to other dependency management practices.
- Evidence of the effectiveness of information transparency in supporting and encouraging more timely downstream responses to dependency abandonment.
- A theoretical framework for understanding, from the downstream user perspective, which dependencies' abandonment will be impactful and noteworthy based on the context of their dependency usage.
- A list of information needs and design requirements for automated dependency abandonment identification tooling.
- An LLM-based classifier to predict the project-specific impact of abandonment using theory-driven reasoning and context-specific information.
- An evaluation assessing the effectiveness of using an LLM-based theory-driven classifier to predict the impact of abandonment as well as the perceived usefulness of context-specific information when making judgments of the impact of abandonment.
- A detailed description of our proposed approach for developing intelligent pre-configuration of context-aware tooling through the use of theory-driven LLM-based classifiers.

April 2, 2026
DRAFT

Chapter 2

Background and Related Work

Reusing open source frameworks, packages, and other abstractions forms *software supply chains* [16], where packages rely on "upstream" dependencies created and maintained by others, that often have their own dependencies, creating the chain. Such reuse speeds up development, but also brings risks "downstream." Dependencies may introduce breaking changes in an update [33], become incompatible with other dependencies [70, 71], contain security vulnerabilities [72, 118, 139, 166], become unmaintained [187], or even get attacked through supply chain attacks [120, 145, 259, 289].

In the remainder of this chapter I will introduce several areas of research and practice that this dissertation builds and relies on. First, I will discuss dependency management (cf. Section 2.1), and how signals can support downstream decision making (cf. Section 2.2). Then I will discuss open source sustainability (cf. Section 2.3), and how package abandonment can pose a threat to supply chain security (cf. Section 2.4).

2.1 Dependency Management

What We Know. Open-source dependencies can provide free reusable functionality to developers. By building on these resources, developers can turn ideas into prototypes and prototypes into deployment code in a fraction of the time and at a fraction of the cost previously possible. However, there is a notable downside to dependencies, namely *dependency management*. Due to both internal and external evolutionary pressures to enhance features, fix bugs, and patch vulnerabilities, dependencies and their application programming interfaces (APIs) change over time [155, 210], sometimes becoming incompatible with old versions or other dependencies a project may have [33, 119, 214]. Such pressures often make coordinating dependency updates and maintaining compatibility between dependency requirements a complex task, especially when lots of dependencies are used or when *breaking changes* occur, i.e., changes that require users to refactor their code. Additionally, projects can face security vulnerabilities through their dependency supply chain, including *transitive dependencies* where dependencies have dependencies of their own [145].

Cross-ecosystem studies of the presence of vulnerable dependencies have highlighted the importance of managing and updating dependencies [218, 289]. Research suggests that generally keeping dependencies up to date correlates with better security outcomes [61, 108, 113, 223, 245, 281, 286, 287]. Because of the complexities of dependency management, there have been calls for documenting all dependencies in a *software bill of materials* (SBOM), including a US executive order signed in May 2021 mandating the tracking and documenting of dependencies (using software bill of materials, SBOM) for software sold to the government [14]. In short, dependency management is a complex ongoing problem that has been

April 2, 2026
DRAFT

studied in different ways.

When developers switch dependencies or update after a breaking change, they often face nontrivial migration work in their own code base. Researchers have attempted to address the many challenges surrounding dependency migration by trying to understand how developers migrate between libraries [19, 59, 261, 262], and by creating numerous tools supporting migration [18, 45, 283]. Even so, attempts to support migration thus far have generally supported limited varieties of API evolution, giving them a limited scope of applicability [47, 78, 209], and limited success in practice [59].

Because keeping up to date with dependency updates can be challenging, research has studied how developers approach and manage dependency updates [28, 68, 70], particularly how they approach versioning and breaking changes [33, 69, 77, 215, 222, 282] and security patches [72, 74, 118, 144, 218], which have been extensively studied, and are considered highly important [208, 246]. Despite common concerns about the continued maintenance of dependencies [83], a central theme in much of the empirical research on dependency management is that developers tend to either be slow about updating dependencies or not update them at all, even those with known security vulnerabilities [29, 72, 73, 74, 144, 150, 177, 215, 218, 218, 228, 257, 290], raising questions about whether abandonment is actually a problem if many projects rely on old versions anyway. For example, Kula et al. [144] studied dependency updates across 4,600 GitHub projects and found that the majority tend to not update dependencies even when security vulnerabilities are involved, with 81.5% of projects having outdated dependencies. Similarly, Decan et al. [72] estimated that it takes almost 14 months for 50% of projects to install a patch for a vulnerable dependency. In addition to studying how updates are managed at large, particular focus has been directed towards studying how breaking changes are dealt with [33].

There have been many attempts to improve dependency management practices. There are many well established software component analysis (SCA) tools to support certain dependency management tasks, such as dependency updates, security vulnerabilities, and license management, including Snyk Bot [11], Dependabot [2], Socket [12], and Sonatype [242]. These tools are designed with the intention of reducing developer workload and toil by automating routine dependency management tasks e.g., keeping the dependencies of a project up to date by notifying developers of update opportunities and creating automated pull requests with proposed updates. The adoption of such tools has become an industry-wide best practice [49, 208, 247, 259], and research has shown that their adoption can lead to positive improvements in dependency management practices and outcomes [73, 116, 191].

Yet research on the usability of those same tools has found that these effects are tempered by pervasive usability issues, with one of the primary issues being sending developers too many notifications, especially those they deem incorrect, unimportant, or irrelevant to their project [87, 116, 191, 230]. These notifications are often perceived as noise and can distract, annoy, and overwhelm developers causing information overload and notification fatigue which can lead to developers ignoring the tool or disengaging altogether [87, 116, 178, 191, 213, 256]. The issue of overwhelming developers with too many spurious notifications is pervasive across automated tooling for many different software engineering (SE) tasks [86, 147, 230, 256, 279] e.g., static analysis tools [24, 134, 232], automated fault detection tools [152], and security alert tools [149, 211, 212].

Furthermore, semantic versioning with floating dependency versions enables automatic installation of patches, but this practice is controversial since it can also introduce risks of breaking changes and deliberate supply chain attacks [74, 215]. Some developers signal proper dependency management by displaying security scores or repository badges [8, 146, 267], which is a topic we will explore further in Section 2.2.

What We Do Not Know. Despite the extensive amount of research that has been done on dependency updates and vulnerabilities, to the best of our knowledge, little research has studied the opposite problem,

April 2, 2026
DRAFT

dealing with dependencies that have been abandoned and that are therefore no longer receiving updates. Dealing with abandoned dependencies is a facet of *dependency management*, yet little is known about how developers respond to dependency *abandonment*, and how dealing with abandoned dependencies compares to other dependency management practices. Tools to help with dependency abandonment are rare,[1] to the frustration of practitioners [184]. Generally, developers can choose to continue using abandoned dependencies if they do not (yet) pose concrete problems, or they can take various actions that all involve removing the dependency and replacing it with something else. In this dissertation I address this gap by (1) providing detailed information on how users can prepare for and address dependency abandonment in Chapter 3; (2) quantitatively studying at scale how often and how fast developers respond to abandonment and how (or whether) this differs from other dependency management practices in Chapter 4; and (3) developing a theoretical framework of what factors influence the impact of a given dependency's abandonment on a project given the context of their dependency usage and leveraging that framework to develop a prototype tool for automatically identifying project-specific impactful dependency abandonment at scale in Chapter 5.

We also know little about how individual developers make decisions about removing abandoned dependencies. Our interviews in Chapter 3 suggest that some developers are very regimented about removing abandoned dependencies (sometimes driven by policies requiring it or a feeling of responsibility) while others prefer to wait for something to break [184]. Yet it is unclear whether the developers that promptly attend to abandoned dependencies are the same ones that follow good dependency management practices and possibly good development practices in general. Thus, we explore whether how developers deal with abandonment associates with other development practices and project characteristics in Chapter 4.

Finally, research on overcoming notification fatigue in such contexts has suggested that only sending developers notifications they deem relevant may alleviate the issue [279]. Sadowski et al. coined the term *effective false positive* which refers to automated tool notifications that are technically correct but that do not matter to the user in practice [232]. Since many developers are not equally concerned about the abandonment of all their dependencies because the impact on their project can vary widely [184], we define effective false positives in the context of automated abandonment identification tools as *notifications about the abandonment of dependencies that are not considered impactful to the project by its maintainers*. Although only sending developers relevant notifications may sound straightforward, it leads to the non-trivial question of 'what abandonment *will* be impactful to a particular project given the context of their dependency usage?' Which we take a step towards answering Chapter 5.

2.2 Supporting Downstream Decision Making with Signaling Theory

What We Know. In open source, developers make many decisions based on publicly available information, without explicit coordination [65, 66, 170, 175, 219, 267, 270]. This includes complex inferences like choosing which developers to follow [31, 153] or hire [43, 174] and which projects to depend on [33, 195]. Maintainers can shape how they present their packages to influence the actions of their users and contributors – such mechanisms are often studied in the context of *signaling theory* [219, 267, 270] and *nudging theory* [116, 170, 191]. For example, developers may include *badges* in their README to signal practices and expectations, such as signaling that a project finds rigorous automated testing and frequent dependency updates important, which may then shape the decisions of potential or current users and the behavior of contributors [266, 267]. Such nudges can be incorporated in the design of tools, e.g., to accelerate the completion of overdue pull requests [170]. In the CRAN ecosystem, volunteers explicitly

[1] Exceptions are FOSSA's Risk Intelligence service, currently in beta, and a recent research prototype by Mujahid et al. [196].

April 2, 2026
DRAFT

coordinate to inform their dependents (within the ecosystem) about breaking changes [33], but this practice is rare otherwise.

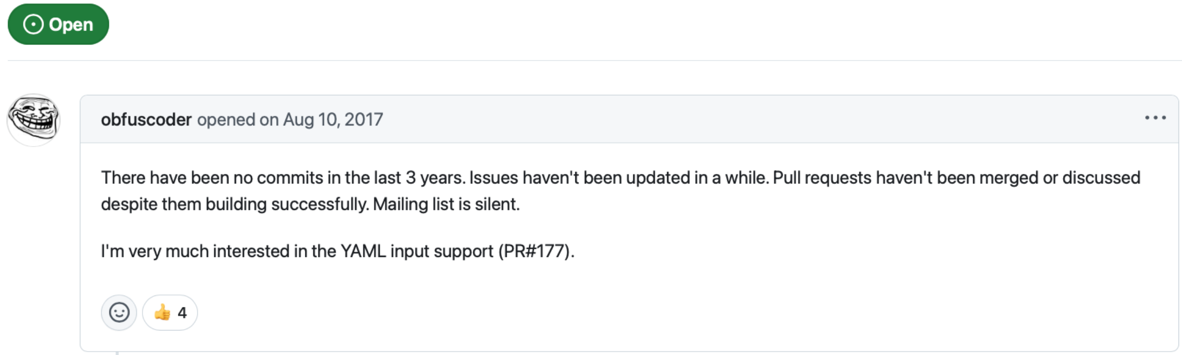


Figure 2.1: GitHub issue illustrating the frequent difficulty of identifying dependency abandonment.

What We Do Not Know. Little is known about what maintainers can do as their final actions to help the community when they decide to stop maintaining a package. Developers make inferences about the abandonment status of packages with all kinds of information (e.g., the date of the last commit, recent issue discussions, forum discussions), yet they often struggle to determine conclusively whether a package is abandoned such as in the example in Figure 2.1. We conjecture that even simple actions like publicly announcing that a package will no longer receive maintenance can shape how affected developers respond to abandonment. As a starting point to explore responsible sunsetting strategies, we investigate how announcing the abandonment status of a package impact how fast dependent projects remove the abandoned dependency in Chapter 4.

2.3 Open Source Sustainability

What We Know. Nearly everything we do on screens from checking email and stock prices to online shopping and reading the news relies on and could not function without open-source software [83]. In 2018, npm, Inc. estimated that, on average, 97% of the code on modern web applications comes from npm [205]. While difficult to quantify, the economic value of open source is also significant; some estimate that in 2010 open-source software produced 342 billion Euros of economic value in Europe alone [67]. Nonetheless, despite the widespread reliance on open source, the reliability and continued maintenance of many of these packages is no sure thing – this is a key motivation for open-source sustainability research.

Prior research argues that a project's maintainers are a crucial part of its success [51], and that it is vital to attract new contributors, support their onboarding, and retain core maintainers. Each of these parts of the contributor life cycle have been studied thoroughly.

In terms of attracting new contributors, researchers have studied the barriers faced by new contributors [216, 251, 252, 253, 275], the project characteristics associated with greater attractiveness to new contributors [38, 96, 219], and even the role of social media [89]. Research supporting the onboarding of contributors has studied the onboarding process [64, 82, 124, 275], the role of scaffolding, mentoring, and social ties [88, 112, 125, 254, 270, 284], and the characteristics of contributors who succeeded in becoming part of the core team [99, 269, 291]. Research on retaining core contributors fo-

April 2, 2026
DRAFT

cused on why they disengage [42, 128, 182], the role of maintaining a healthy community to reduce that risk [92, 183, 220], and the impact of disengagement on the health and survival probability of a project [91, 93, 130, 163, 200, 227, 272].

Research has also studied the impacts of project and ecosystem characteristics and organizational structures on open-source projects including the effect of codes of conduct [241, 266], how badges can be used as a signal to attract new contributors [267], how project and ecosystem characteristics impact maintainer retention and project activity [55, 111, 272], the maintainability and sustainability of projects [53, 111, 233, 285, 292], and the impact of commercial involvement on open-source development [44].

What We Do Not Know. Taking a step back, we can observe that almost all sustainability research thus far focuses on studying various factors, characteristics, and phenomena that support the goal of *keeping particular projects or ecosystems alive and actively maintained*. While lots of research has explored how to prevent the abandonment of the open-source packages that serve as our digital infrastructure, there are very few insights on addressing abandonment when it occurs. However, because of the self-organized and volunteer-based nature of much of open source, we likely cannot stop all projects from being abandoned or ensure their ongoing maintenance.

Many popular open-source packages hosted on GitHub rely on one or two core maintainers who are often volunteers to keep the package running [22, 83], and core maintainers sometimes disengage for various reasons that occur normally in life, such as starting a family, switching jobs, no longer having enough time, or simply losing interest [182]. Maintainers losing interest or no longer having enough time to contribute are two common reasons open-source packages fail [51]. One study of popular packages on GitHub found that 16% were abandoned by maintainers, and in 59% of those abandoned packages, nobody stepped up to take over maintenance efforts leaving the package fully abandoned [23].

Therefore, since open source is depended on by *"our economy and society, from multi-million dollar companies to government websites"* [83] to support the rapid and efficient development of modern software, we argue open-source sustainability research must expand its focus to include supporting the sustainable *use* of open source by helping developers better prepare for and deal with dependency abandonment and its consequences when it occurs. This general direction, which I pursue in this dissertation, has received relatively little attention in the literature, with a few exceptions of prior works measuring and communicating library and community health to potential users to help them avoid selecting packages to depend on which may be in decline or otherwise have indicators of being unsustainable [193, 272].

2.4 Package Abandonment as a Supply Chain Security Risk

What We Know. Reusing open source frameworks, packages, and other abstractions forms *software supply chains* [16], where packages rely on "upstream" dependencies created and maintained by others, that often have their own dependencies, creating the chain. Such reuse speeds up development, but also brings risks "downstream" increasing a project's attack surface [120, 278]. Package abandonment creates weak links in software supply chains [289], and represents an emerging attack surface that is increasingly being leveraged by attackers [247]. In fact, there are entire organizations like the Open Source Security Foundation (OpenSSF) whose primary mission includes developing research, best practices, evaluation metrics, and enterprise tools to make *it easier to sustainably secure the development, maintenance, and consumption of the open source software we all depend on* [7, 208]. Developers worry about abandonment (e.g., online [240]) particularly from a security perspective [184]. Widely-used package abandonment abandonment can disrupt supply chain integrity and increase the supply chain attack surface for millions of downstream users [120], exposing them to the risk of unpatched vulnerabilities [289, 293] and targeted

April 2, 2026
DRAFT

supply chain attacks [145, 244, 259, 289]. Our interview study (cf. Chapter 3) also revealed developers' frustration that they will not receive the new features or support they had hoped for, that the package will become increasingly less useful as requirements and the environment change, and that the package will become incompatible with other evolving infrastructure [184].

What We Do Not Know. We have very little data about how prevalent abandonment is among widely-used packages or how many downstream projects are exposed. Sonatype's 2023 State of the Software Supply Chain report finds that 18.6% (24,104) of open-source packages that were maintained the prior year no longer qualify as maintained that year [246], but it is not clear how this data was collected and whether such results generalize to widely-used packages that might be considered critical digital infrastructure. Quantifying the frequency of abandonment and the resulting exposure downstream is needed to understand the scope of the problem, therefore I quantify at scale the prevalence of and exposure to package abandonment in Chapter 4.

April 2, 2026
DRAFT

Chapter 3

Navigating Dependency Abandonment

3.1 Introduction

In this chapter, we collect, curate, and contextualize the experiences and practices of developers who have dealt with open-source dependency abandonment. With the goal of understanding what developers do when facing open-source dependency abandonment, we explore this topic with two research questions (RQs):

RQ1 How do developers prepare for the risk of open-source dependency abandonment?

RQ2 How do developers deal with open-source dependency abandonment, once it occurs?

This chapter describes the the work done in our paper '*We Feel Like We're Winging It*': A Study on Navigating Open-Source Dependency Abandonment [184]. We conducted semi-structured, in-depth interviews with 33 developers who have experienced open source dependency abandonment, which we will refer to as just *abandonment* moving forward for brevity. We identified three stages during the dependency life cycle where interviewees commonly took action to address the risks and realities of abandonment: before adoption, while using a dependency that is still being maintained, and after a dependency has become abandoned (see Figure 3.1). While we identified a wide range of philosophies surrounding preparing for and dealing with abandonment, there was a common sentiment that there are often very few resources on dealing with abandonment; interviewees often had to figure it out by trial-and-error with little guidance.

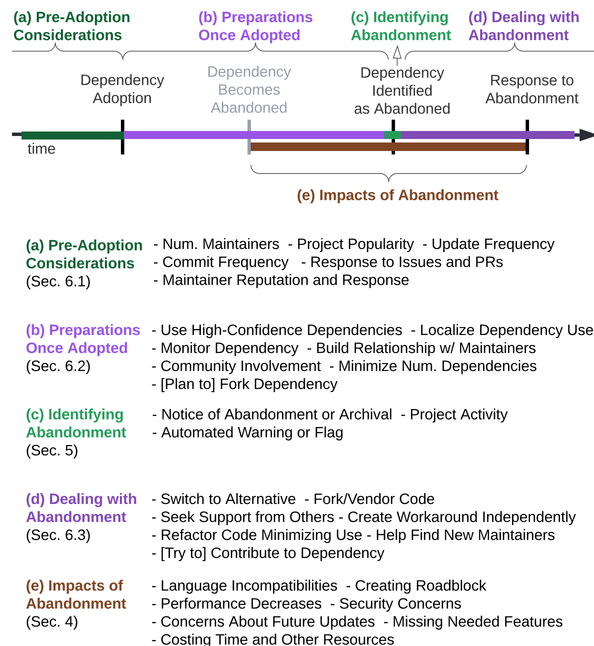


Figure 3.1: Dependency life cycle with the common stages where abandonment is addressed highlighted.

While not all interviewees believed it was worthwhile to invest in preparing for abandonment, some did, and they prepared, e.g., by creating abstraction layers in their code base to localize dependency use, and by monitoring the dependency and its surrounding community to stay informed of any issues or potential signs of abandonment. Once interviewees identified abandonment, they often sought support and

April 2, 2026
DRAFT

guidance from the community, switched to alternative dependencies, and forked or vendored abandoned dependency code. Overall, we suggest that there is a potential to reduce the costs associated with abandonment through investments into preparation, but it is often unclear whether that preparation will pay off. In addition, there is often potential for community members to invest in solutions that will benefit others facing the same problem, such as creating a migration guide, we call these *community-oriented solutions*. However, developers often have little incentive to create such community-oriented solutions – an instance of the *volunteer's dilemma* [76]. We survey solutions to the volunteer's dilemma from fields like social psychology and game theory, and discuss how they can be applied to this context.

In summary, this chapter makes the following contributions: (1) a list of stages in the dependency life cycle where the risks and realities of dependency abandonment are commonly addressed; (2) a taxonomy of common strategies developers use to prepare for and deal with dependency abandonment which can serve as a reference to both practitioners and researchers; (3) a theoretical framework for the costs associated with abandonment as well as suggested cost-reduction strategies; and (4) the concept of community-oriented solutions and evidence-based strategies to overcome the volunteer's dilemma to collectively address abandonment.

3.2 Research Design

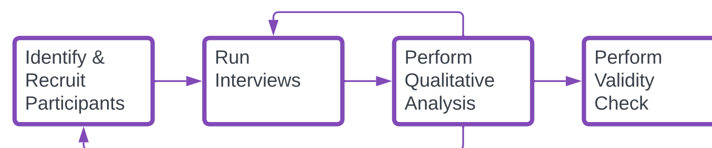


Figure 3.2: Research methodology flow chart.

Because, as far as we know, there has been little research studying how developers prepare for (RQ1) and deal with (RQ2) dependency abandonment, we used an iterative research process and qualitative research methods. Specifically, we performed semi-structured interviews with interwoven analysis and exploration, as we illustrate in Figure 3.2. As is often recommended, we did not compartmentalize the interviews and the analysis into separate discrete phases, but instead iteratively built our understanding and adjusted our interview guide and codebook in tandem throughout the interviews [159]. We will now discuss study design, analysis, and limitations.

3.2.1 Identifying and Recruiting Participants

Because we wanted to talk to people who had experience dealing with open source dependency abandonment, for our interview study we specifically targeted people who had depended on an open source project that then became abandoned recently. To identify such maintainers, we worked backward: First, we identified abandoned projects, then we identified projects that depend on each abandoned project, i.e., the *dependents*, and finally, we identified the maintainers of those dependents.

Defining and Identifying Abandoned Projects. Because customs and behaviors surrounding dependency management can vary widely by ecosystem [33], we searched for abandoned projects in three package manager ecosystems to collect a diverse pool of experiences: *npm*[1] (Javascript), *PyPi*[2] (Python), and

[1]Node.js Package Manager, `https://npmjs.com`
[2]The Python Package Index, `https://pypi.org`

April 2, 2026
DRAFT

Composer[3] (PHP). Using data cross-linked between GHTorrent [103] and each ecosystem's respective package manager website (matching packages to their corresponding GitHub repositories, when mentioned explicitly in the package manager metadata), we heuristically searched for projects with signs of abandonment. Concretely, we identified all projects with at least ten commits a month for two consecutive years and less than three commits total in the following year, i.e., the year in which the project is presumed abandoned; the three-commits threshold allows for some residual activity (e.g., posting warnings about abandonment in the README file) and mirrors prior work [182, 272]. Once we had a pool of potentially abandoned projects, we randomly sampled and manually evaluated whether each project seemed indeed abandoned by investigating the activity patterns on each project's GitHub repository until we had 10-20 high-confidence abandoned projects per ecosystem. For this we looked at the most recent period we could observe at the time – the first six months of 2022, regardless of the year we suspected the project was abandoned based on the automated heuristic – and manually checked if the project either (1) did not have any significant commit activity;[4] or (2) had an explicit label or notice that it was abandoned, archived, or simply no longer maintained.

Identifying Dependent Projects and Maintainers. We then used GitHub's dependency graph feature to get the list of dependents for each abandoned project [100], and collected the data using the github-to-sqlite library.[5] We ensured the dependent projects were active by considering only dependents that had, on average, at least ten commits a month in the first half of 2022. We then identified each dependent project's top maintainers by commit counts during the first half of 2022, collected their publicly available email from their GitHub profiles, and sent out 412 interview invitations in total in staggered batches of 10-20. Our study design was approved by our Institutional Review Board.[6]

3.2.2 Interview Protocol

Interviews began with introductions and verbal consent. The main topics of the semi-structured interview guide included (1) how interviewees identified abandonment; (2) the impact of abandonment on their project; (3) how they dealt with the abandonment and what solutions they used; (4) whether they prepared for the risk of the dependency becoming abandoned *before* identifying abandonment; and (5) whether they considered or evaluated the risk of the dependency becoming abandoned before adoption. Since the goal of the interviews was to understand how interviewees prepared for and dealt with the abandonment, during interviews where time permitted we identified additional abandoned dependencies to discuss, in addition to the original dependencies that were identified, by asking *"have there been other instances of any of your project's open-source dependencies becoming unmaintained or abandoned by maintainers?"* We typically were able to discuss two abandoned dependencies per interview, and we kept discussions focused on those specific cases to get concrete insights.

[3] PHP Dependency Manager, `https://getcomposer.org`

[4] Since abandonment need not align with calendar year boundaries, we still considered as abandoned projects with a few trailing commits at the beginning of the six-month window but no commits thereafter.

[5] `https://github.com/dogsheep/github-to-sqlite`

[6] We sent a small number of targeted emails, based on information our participants posted *publicly* in their profile. In terms of research ethics, especially the Belmont report's principles of *respect for persons* and *beneficence*, we consider that the costs (e.g., potentially unwanted emails) and risks (e.g., releasing confidential information) to potential participants are low, and insights gained in better dependency management benefit all open source contributors. We considered alternative sampling strategies and concluded that because we were interested in speaking to a specific group of open source maintainers, that it seems unlikely that we could have recruited people in a different (less targeted way) without increasing the general cost to the community by engaging with large groups of maintainers.

April 2, 2026
DRAFT

3.2.3 Data Collection and Analysis

The interviews took place over Zoom and lasted 25 minutes on average. In total we conducted 32 interviews (P1-32) where one interview was with two developers (P2a, P2b). We stopped running interviews once we reached our saturation criterion, which we defined as three consecutive interviews without learning any new major insights [95]. We qualitatively analyzed the interview transcripts using iterative thematic analysis [37]. The process followed Lincoln and Guba's trustworthiness criteria [105], as discussed by Nowell et al. [202]. During this process, we were perpetually switching between the stages of exploring the rich transcripts, engaging with and analytically memoing the data [181], coding, searching for themes, and refining the codes and coding framework, as is recommended [159].

The analysis began with the first author performing open-ended inductive coding of each interview as we went. After the first eleven interviews, all the authors came together and performed an in-depth analysis of the codes and coding frame. Iterative adjustments to the coding frame and interview guide were made as necessary. Once a coding frame was settled on, the first author re-coded all the transcripts, with any uncertain cases being reviewed by another author. A later participant discussed a dependency that was marked abandoned but still received security updates, and we explored this further by identifying and interviewing developers who faced this type of dependency abandonment. We quickly reached saturation and did not find any new major insights.

3.2.4 Validity Check

To validate and check for fit and applicability of our findings as defined by Corbin and Strauss [58], we performed a validity check by sharing our findings and results with interviewees. We confirmed our interpretations of the rich interview data aligned with the interviewees' experiences by getting interviewees' thoughts and feedback. We sent all interviewees summaries and the complete drafts of Sections 3.3, 3.4, 3.5, and 3.6. We also sent a list of prompts and questions asking interviewees to look through the documents for areas of agreement or disagreement, general correctness, and any additional insights they gained after reading through the findings as well as the experiences and strategies of other developers. Six interviewees responded, all six confirmed that they largely agree with our findings, e.g.,*"I think your paper is a well-considered analysis of the subject that fits with my experience, fwiw"* (P11). One interviewee pushed back on augments made by other interviewees suggesting that abandonment was not always a problem because there are not always impacts. They argued that *"abandonment is always problematic and always has an impact, even if the software itself is not broken, because abandonment still forces a consumer to act as if it is abandoned, i.e. to prepare for breakage or vulnerabilities"* (P4).

3.2.5 Limitations

The findings of our qualitative interview study suffer from the same limitations commonly found in work of this kind. Generalization beyond the pool of interviewees should be made with caution. See paper for full description of limitations [184]. Self-selection bias could influence the transferability of the results because there could be differences in the personalities and beliefs in the sample and the subset that chose to participate [173, 229]. We tried to reduce this risk by streamlining the enrollment process and keeping interviews short. There is also a question of authenticity in how we defined 'abandoned' dependencies since the definition may not fully represent the concept of project abandonment [164], although during the discussions with interviewees there was agreement with our definition.

3.3 Impacts of Abandonment

Unlike breaking changes which by definition break things, it is not obvious that dependency abandonment in and of itself is problematic. If a dependency worked last year and has not been changed, there is no inherent reason why its abandonment would cause problems. However, Lehman argues that software either *"undergoes continual changes or becomes progressively less useful"* [154]. We start by exploring if and how abandonment impacted interviewees. We provide a summary of the types of impacts experienced in Figure 3.1.

Concrete Problems. We define concrete problems as technical problems that impact a dependent project. **Language incompatibilities** (P11, 17, 23) occurred when interviewees were trying to update other parts of the project but could not, because the unmaintained dependency caused a language incompatibility between itself and other dependencies or the rest of the project. For example, *"we were trying to upgrade our Saas platform from Python 2 to Python 3, and it was a core dependency, so we needed it to work [with Python 3], and it didn't. So we ended up having to move to another library"* (P17).

Some interviewees described experiencing **performance decreases** (P13, 32) as a result of dependency abandonment. One interviewee described how they had to depend on multiple versions of their core libraries because the unmaintained dependency relied on older versions but their other dependencies relied on newer versions as they were released, which increased compile times and binary size for end users (P32).

Some dependencies were **missing needed features** (P14, 20, 23) or features that interviewees believed may be necessary in the future, which they no longer expected because of the abandonment.

Anticipated Problems. Anticipated problems are problems interviewees are concerned may impact the project in the future but have yet to materialize. Some interviewees had **concerns about future updates** (P16, 22, 29) and worried there could be problems down the line due to the lack of maintenance, such as incompatibility issues when updating other dependencies. For example, *"I was not facing any problem in particular, but I was concerned because the library didn't get any updates"* (P16).

Some had **security concerns** (P4, 22, 28, 29) about potential future vulnerabilities or other security-related issues. However, no interviewees reported experiencing an actual security vulnerability associated with an abandoned dependency. For example, *"we've never had a security incident related to an abandoned dependency, but that's always a concern— that there could be a security vulnerability"* (P4).

General Impacts. In many cases, interviewees described general impacts of abandonment rather than specific problems, so we distinguish this discussion from the discussions above. Dealing with dependency abandonment often **costs time and other resources** (P4-7, 9, 10, 20, 21, 23, 25), which was often related to replacing the dependency or creating a workaround to deal with abandonment. For example, *"right now, we're working through the fact that the [dependency] is no longer being actively worked on. Which means that we need to switch to something else. We're looking at [alternative dependency], but there's really no way to replace that dependency without rewriting huge portions of the project, and so that's just something we have to put effort into and work through"* (P7). Sometimes abandonment **created a roadblock** (P10, 16, 21, 27, 29, 30) or notable problem that stopped or significantly impacted project progress, and required a workaround or solution to be employed quickly.

Some interviewees reported the abandonment had **no meaningful impact** (P6, 7, 9, 11, 16, 25, 29) and argued that just because a dependency was abandoned does not necessarily mean there is a problem (in contrast to the interviewees that mentioned *anticipated problems*, who were at least concerned about possible future problems). They explained that if the software is complete, does not interact with other software, and does not become insecure itself, then the abandonment is not necessarily problematic. For example, *"it was recognized within the organization that [...] one of the dependencies that the business*

runs on is totally unsupported for years [...], and because it wasn't a cause of many problems it wasn't necessarily an issue" (P11).

Overall, interviewees rarely mentioned concrete problems when discussing how dependency abandonment impacted them. Most of the impacts described were concerns about anticipated problems or general impacts whose problems of origin were not mentioned. It appears some interviewees had expectations of their dependencies regarding ongoing maintenance, feature creation, or support. When abandonment occurred, those expectations were no longer being met, making them feel like they were impacted even though no concrete problems like an unfixed bug, unpatched security vulnerability, or dependency version incompatibility had occurred yet. This leads to questions about dependent projects' exact expectations and how they interact with and relate to the concrete technical problems caused by abandonment.

Distinctions in Impact Between Dependencies. The impact of abandonment can vary widely depending on the type of dependency in question. There was often much more concern about dependencies used at runtime, for security, or for other user-impacting tasks compared to dependencies used in development environments or as infrastructure during testing and deployment, which were commonly seen as less impactful and concerning. For example, *"if we have a runtime dependency that is abandoned or not maintained or has security issues, we either typically contribute to that project to bring it up to speed and fix those vulnerabilities or look for an alternate, so we're really specific and careful about runtime dependencies"* (P2a).

Key Insights: Most impacts were not concrete technical issues but broad concerns about potential future issues or general impacts like costing time. While some interviewees were concerned about possible future security vulnerabilities, no interviewees reported experiencing a security vulnerability associated with an abandoned dependency.

3.4 Identifying Abandonment

It is important to understand how abandonment is identified, because in cases where identification happens after a concrete problem has occurred, immediate action is frequently needed which can be disruptive to projects. Thus many developers want to identify abandonment before it causes a concrete problem, so they can react without immediate time pressures. Interviewees used a wide range of information to identify abandonment. This information varied along two dimensions, first how *visible* the information was, and second *how* the information was discovered. We now catalog the information used to identify abandonment and discuss how it varies across the aforementioned dimensions. We provide a summary of the codes in Figure 3.1.

Manually-Identified Information. Abandonment was often *manually identified* by observing various project characteristics like **commit frequency** (P8, 21), **lack of updates** (P4, 6-8, 12, 29, 30, 32), and **lack of progress resolving issues or pull requests (PRs)** (P2a, 16, 17, 21, 29). These forms of information often have *high visibility* since they are easily observed during a quick inspection of the project.

Many participants identified abandonment by observing a **notice of abandonment/archival** (P3, 4, 7, 13, 17, 20, 29, 30). The notices were often posted somewhere on the abandoned dependency's repository page, but there was a wide variation in visibility depending on the particular location. Sometimes the information was *highly visible*, being posted as a flag/warning at the top of the page, a message at the top of the README, or as a note in an issue tracker thread explicitly discussing the maintenance status of the project. For example *"my colleague saw as he was looking at issues [...] that there was this issue saying 'this will no longer be maintained"'* (P3). The project inspection that led to the discovery of this

information often occurred because the interviewee traced an error back to the dependency or because they were using the dependency as a reference when doing something like implementing a new feature. Other times, the information had *low visibility*, meaning it was possible to find but required more effort to locate (e.g., an unrelated issue or PR that a maintainer responded to announcing they no longer plan to maintain the project).

Tool-Supported Identified Information. Some interviewees used information from observing an **automated warning or flag** (P4, 6, 7, 13, 22, 25, 27, 29, 31, 32) which often provided *highly visible* information. Often these warnings occurred because the dependency maintainers had explicitly marked the project as abandoned/deprecated or because the unmaintained dependency was causing some sort of incompatibility error, such as those described in Sec. 3.3.

Flags for abandoned/deprecated packages are a recent feature of several package managers, allowing maintainers to explicitly signal that a package is abandoned/deprecated. These flags generate warnings when users either install, update, or use said package (with specifics depending on the package manager). In 2015, *Composer* incorporated the ability to add a flag to a package indicating it has been abandoned which is used to generate warnings when users install or update flagged packages.[7] Similarly, 'since 2020 *npm* as had the *npm-deprecate* command, which allows maintainers to add a deprecation flag to a package's *npm* registry entry, producing a deprecation warning whenever someone installs the package [204]. We could not identify an equivalent *PyPi* feature, but found community discussions that proposed creating one and cited the *npm-deprecate* function as an example.[8] GitHub also has an platform-wide *archive* flag for repositories.[9]

Key Insights: Manually-identified information like project characteristics were often used to identify abandonment, such as commit frequency and progress resolving issues or PRs. Some package managers like *npm* and *Composer* provide abandoned/deprecated project flags, which can be used to automatically detect abandonment in projects that have been explicitly flagged as such.

3.5 Preparing for and Addressing Abandonment

Through our qualitative analysis, we identified several stages in the timeline of an interviewee's experience with a dependency where they frequently took action to prepare for or deal with dependency abandonment. In Figure 3.1, we present these key stages, which are (1) considerations before adoption regarding current or future dependency maintenance, (2) strategies used during or after adoption to prepare for the risk of abandonment, and (3) solutions to address abandonment once identified. We now discuss each stage chronologically to mirror interviewees' experiences.

3.5.1 Considerations Before Adoption

When deciding whether to adopt a dependency, interviewees often reported evaluating the current maintenance status and the expected risk of future abandonment by examining project and maintainer characteristics. Essentially all mentioned factors mirror those discussed in literature about general dependency selection [33, 148, 194, 219]. However, we distinguish these considerations from those for general dependency selection because we specifically asked if and how they evaluate the risk of a potential dependency

[7] `https://github.com/composer/composer/issues/4610`
[8] `https://github.com/pypi/warehouse/issues/345`
[9] `https://docs.github.com/en/repositories/archiving-a-github-repository/archiving-repositories`

becoming unmaintained or abandoned before adopting it. For the sake of completeness, we present the considerations discussed by interviewees.

Project popularity (P2a, 6, 9, 10, 12, 13, 16, 17, 19, 21, 23-25, 27, 30, 31) was often operationalized by looking at the number of stars, forks, or users. The **update frequency or time of the last update** (P5, 6, 10, 11, 13, 17, 20, 22, 27-30) and the **commit frequency or time of the last commit** (P4, 5, 8-13, 16, 24, 28) often gave insights into the regularity and recency of general project activity and progress. Interviewees used these highly-visible project metrics to make quick judgments and predictions about current and future maintenance status. The **response to issues and PRs** (P4, 9-12, 16, 21, 28, 30) provided insights into whether there were (1) a lot of bugs or problems with the project; and (2) whether the maintainers were still actively participating.

The **number of maintainers** (P2b, 7, 13, 16, 19, 28, 30, 31) often impacted expectations for future maintenance; projects with fewer maintainers were often seen as less desirable since maintainer disengagement may have a more considerable impact on project maintenance.

Some also considered the content and tone of the **response or reaction of dependency maintainers** (P2a, 2b, 4, 9-12, 21) when deciding whether to trust the project. Maintainers who were helpful, friendly, and welcoming often gave interviewees more confidence that they would be cooperative and helpful if something were to occur. Some used **the reputation, status, or previous experience of the potential dependency maintainers** (P4, 8, 10, 11, 17, 19) as an important metric when deciding whether to trust a potential dependency. Having experienced maintainers with positive, long-standing reputations was reported to be a good sign.

Choosing Between Dependencies. Several interviewees discussed factors they use when deciding between multiple potential dependencies. In general, they reported preferring projects that seemed more reliable and maintainable over projects with better performance or more cutting-edge features. This often appeared to come from being burned by an abandoned dependency previously, and wanting to avoid experiencing another similar situation.

Key Insights: A project's popularity, activity, and maintainer reputation were often used when considering the risk of a potential dependency becoming abandoned, mirroring factors used in general dependency selection [33, 148, 194, 219].

3.5.2 Preparations Once Adopted

Between when a project decides to adopt a maintained dependency and when that dependency is identified as abandoned, some interviewees prepared for the risk of abandonment occurring. Interviewees engaged in many different kinds of preparation. Some forms of preparation focus on making it easier to identify abandonment and others focus on making it easier to deal with abandonment when it occurs. Additionally, some forms of preparation are one-time actions whereas others are reoccurring actions.

A method of preparation that was highly regarded and seemed to be relatively successful was **minimizing/localizing dependency use** (P2a, 6, 7, 16, 27, 32) in the project's code base. This often meant explicitly designing the implementation at the time of dependency adoption in a way that made dependency replacement easier by minimizing the points of contact using an *abstraction layer*. For example, *"as much as possible, we try to buffer dependencies with abstractions so that specific implementation details of a third-party library aren't scattered through the whole application in difficult ways"* (P7).

Some interviewees prepared by directly **monitoring the dependency** (P2a, 4, 10, 13, 27) to keep an eye on how things are going, often by looking at project characteristics similar to those described in Sec. 3.4. For example, *"we are always conscious of the dependencies and looking closely at them"* (P2a).

By remaining aware of the state of the dependency and its community, interviewees place themselves in a better position to identify early signals of abandonment which gives them an opportunity to act before abandonment and any resulting concrete problems occur, if they so choose. Some also prepared by **being active and informed members of the community** (P2b, 16, 17, 31) and **building relationships with dependency maintainers** (P1), often so they could notice issues earlier or have people to reach out to if abandonment occurs. This often involves at least semi-frequent interactions with dependency maintainers or other community members to stay informed of the goings-on in the project and aware of any potential issues or warning signs of something like abandonment being on the horizon. For example, *"I suppose I engaged pretty actively in the open source community, particularly around Python, so I would hope I would have a feeling for what was going on. I think it's partly about being aware"* (P17).

Some interviewees report **only using high-confidence dependencies** (P6, 7, 11, 18, 19, 21, 25, 27) in the first place, which they believed were sufficiently unlikely to be abandoned. Similarly, some **minimize the number of dependencies** (P2a, 9, 11, 24, 27) they use by actively going through and removing unnecessary dependencies to reduce their surface area of exposure. For example, *"I think [we] removed a couple dependencies that we didn't need, there were small use cases, and [we] just authored code to replace the dependencies"* (P2a). One interviewee reported that their development team has a specific role called the *Sustainability Engineer* (a.k.a., the 'sus' role) whose responsibilities each sprint include, among other things, managing dependencies by looking through their code base and finding parts that can be cleaned up by removing unnecessary dependencies. This allows their team to slowly and incrementally manage and remove unnecessary dependencies, making it less of a large and daunting task. Some prepared by creating plans for dealing with particularly important dependencies if they become abandoned, e.g., **forking or planning to fork dependency** (P2a, 5, 9, 11, 20) so they have a backup if something happens.

Whether to Prepare or Not. For various reasons, interviewees often did **no preparation** (P3, 4, 6, 9, 10, 14, 16, 17, 19, 20, 22-25, 29). In some cases, preparation was something they had yet to consider. Others reported that it would be nice if they had the time, but that ultimately preparing sounds like an overwhelming or difficult task given how many dependencies they have. Others subscribe to the philosophy that *'it is not a problem until it is a problem,'* meaning they do not concern themselves with potential future issues. These interviewees did not believe it was necessarily worthwhile to prepare for the risk of abandonment because they did not believe abandonment is in and of itself always problematic or impactful, as discussed in Sec. 3.3. For example, *"unmaintained doesn't necessarily mean that there is any problem with the library"* (P32). They instead wait until there is a concrete problem, at which point they deal with it. Another interviewee said the decision of how and whether to prepare for dependency abandonment points to a long-standing perpetual balance in software engineering. They explained, for example, that abstraction layers increase project robustness but can also increase code complexity making it harder to maintain, which can also act as a roadblock when introducing and onboarding new contributors (P4).

Key Insights: Interviewees who prepared for the risk of dependency abandonment often did so by localizing the use of dependencies in their code base by building abstraction layers or by remaining aware of the goings-on in the dependency itself and the broader community.

3.5.3 Solutions to Abandonment

Once dependency abandonment was identified, nearly all interviewees deployed some sort of solution to deal with abandonment. The most common solution was **switching to a better maintained alternative** (P1-4, 6, 7, 10, 12-14, 16, 17, 20, 23, 27, 29, 31, 32). Interviewees found these alternatives in various ways. Sometimes, an issue or PR on the abandoned project included a discussion recommending

an alternative. For example, *"actually, I can see now on the 'is the project dead' issue there's someone saying use [alternative project], which was the alternative that we ended up going to"* (P17). In other cases, interviewees used search engines such as *DuckDuckGo*, forum websites such as *Reddit* or *StackOverflow*, package managers such as *PyPi*, or even specialized open-source library recommendation websites such as `libhunt.com` to find pointers to alternatives. Another interviewee described how an automated warning about an abandoned dependency included a list of alternatives, which was used to select a replacement (P32).

Often the goal was not just to find another project that had the same functionality, but that also has a similar API to make migration easier and minimize disruption to their code base. For example, one interviewee found an alternative with essentially the same API so the migration entailed *"basically just changing the namespace on what we import that functionality from"* (P32).

Another common solution was to **fork or vendor code** (P1-2b, 4, 5, 7, 10, 12-14, 16, 20, 23, 30, 32) from the abandoned dependency; vendoring means incorporating 3rd party software directly into a code base [263]. For example, *"sometimes we vendor some code, which means we'll just directly copy the code and re-license it into the package itself"* (P1). A drawback of this solution is that it can increase the amount of code a developer is responsible for maintaining over time. As one interviewee put it *"I think that's like the last thing that anyone wants to do, just develop it yourself, because then you would have to become the one that maintains it"* (P31).

Most of the time, when interviewees forked a project, it was used as a personal fork, acting as their own stable version with which they could control and maintain compatibility. Only one interviewee explicitly discussed making a hard fork that they advertised as an alternative for others to use (P30).

Seeking support from others (P4, 5, 7, 10, 12-14, 17, 21, 23, 25, 30, 32) by reaching out to the maintainers or others in the community provided insights into the situation and what potential solutions or next steps could be. In several cases fellow community members had already posted bug fixes or pointers to alternative dependencies in the abandoned dependency or created blog posts explaining how to migrate to an alternative. For example, *"The first [strategy] we figured out is, you know, go through the issue list and see what kind of issues people are having, and if it's similar issues, I try to talk to them to figure out what the exact fixes are and stuff like that"* (P10).

Others **[tried to] contribute to the dependency** (P2a, 3, 5, 13, 23, 30) by reaching out to the maintainer about helping or providing maintenance support. In some cases, the old maintainers would respond after several months, and in other cases this was not a successful solution because they did not receive a response. For example, *"I and others were reaching out to the original maintainer trying to see if we could take it over, and he was basically non-responsive. He had originally posted on Twitter; if you look at that discussion, he was looking for a maintainer. But he just dropped off the map"* (P30).

Another solution used by some was trying to **help find new maintainers** (P4, 5, 7, 12, 25) by supporting community efforts to recruit new maintainers to take over. This was often accomplished through discussions on the abandoned project's issue tracker. For example, *"I'd say my strategy has been to reach out to folks in the issue tracker and encourage them to rename the project and get something up and running, and offer myself for testing if somebody works on it. So at this point, I'm just monitoring the situation and trying to encourage others to step up and work on it"* (P25).

Key Insights: Seeking support from the community and switching to an alternative dependency can be effective and low-effort solutions assuming the required infrastructure is present. Given a deficiency of such, forking or vendoring the abandoned dependency can be a quick fix but can also increase the maintenance effort required over time.

April 2, 2026
DRAFT

3.6 Discussion: Towards More Sustainable Use of Open Source

Our research has catalogued a diversity of practices to prepare for or deal with open-source dependency abandonment. Reflecting on the costs and potential benefits of all these practices, we now discuss higher-level emerging themes, drawing also from the theory of the volunteer's dilemma.

3.6.1 The Cost of Dependency Abandonment

From interviewees, we heard about the costs associated with abandonment throughout our study: We showed the sometimes disruptive impacts of abandonment (Section 3.3) and showed the various, often costly actions developers used to deal with abandonment (Section 3.5.3). When a dependency becomes abandoned, it shifts at a high level from being a free and easy to use software artifact to a potential liability and source of unexpected disruptions, costs, and concerns. One way to think about the total *anticipated cost of abandonment* is as a product of the probability of abandonment occurring and impacting the dependent project (*impact probability*) and the effort required to react to the abandonment once it happens (*reaction effort*):

$$\textit{anticipated cost of aband.} = \textit{impact probability} \times \textit{reaction effort}$$

With this framing, almost all the actions that we see developers take to prepare serve as *investments* to reduce the *anticipated cost of abandonment* by trying to reduce either the *impact probability* or the *reaction effort*, for example:

- Only using high-confidence dependencies and minimizing the number of dependencies (Section 3.5.1) both reduce the *impact probability* but require investment both in terms of necessary research effort and accepting potential opportunity costs from *not* using certain dependencies.
- Minimizing/localizing dependency use (Section 3.5.2) can reduce the *reaction effort* post abandonment with some upfront investment in terms of designing an abstraction layer.
- Monitoring the dependency (Section 3.5.2) can be seen as an investment to notice dependency abandonment before it becomes an urgent problem – this gives developers an opportunity to act on their own time with lower *reaction effort* compared to when they are forced to react in an emergency situation to a roadblock or other concrete problem.
- Although outside the scope of this paper, any investments to keep projects alive, such as by improving funding (Section 2.3), can reduce *impact probability*.

This cost framing highlights how developers can consider investing in preparation to reduce the anticipated cost of abandonment. Whether that investment is prudent is often not obvious in practice and depends on both the risk aversion of the developer and the relative investment costs and cost reduction benefits:

$$\textit{return on investment} = \frac{\textit{reduction of anticipated cost of aband.}}{\textit{investment cost for preparation}}$$

3.6.2 Aspirational Cost Reduction Strategies

Beyond the preparation strategies discussed earlier, the software engineering literature as well as some interviewees suggest possible solutions to reduce *impact probability* or *reaction effort* or the investment cost for preparation – each making such investments more efficient. While most are not widely adopted, we discuss them here as aspirational strategies and promising directions for future work.

Proactive Warnings for Unmaintained Dependencies (Identifying Abandonment). Often identifying whether a dependency is abandoned requires manual effort (e.g., observing commit frequency or looking for notices of abandonment/archival, see Section 3.4). To reduce the investment required, automated tools can provide proactive warnings for unmaintained dependencies. For example, one interviewee expressed how they wished they had a tool that would notify them when one of their dependencies has been unmaintained for a given period of time. They described how a Dependabot-like tool could indicate *"if there are no updates to this package in, say, six months, eight months, a year"* (P23)., which *"would give an idea of what kind of things I'm depending on that are starting to go out of style"* (P23). Only one interviewee (P20) reported using a tool that does just that– the beta *Risk Intelligence* service by *FOSSA* notifies users when a dependency has not been updated in the past two years [217]. Future work could explore how to design such tools without overwhelming developers with configuration work and alerts causing notification fatigue.

Increasing Transparency about Expected Project Maintenance (Preparing for Risk of Abandonment). While many prepared by only relying on high-confidence dependencies (Section 3.5.2), determining whether a dependency is high-confidence was often done with non-trivial manual evaluations of project characteristics like responses to issues and PRs. Transparency mechanisms frequently studied in software engineering and collaborative work [267], such as badges in READMEs, can make it easier to assess the status of a project. One interviewee (P22) explained how their company has started putting badges in their public projects' READMEs showing their intended support status (e.g., support status actively maintained). Such transparency mechanisms can be used to declare maintenance intention (e.g., beta phase, hobby project, actively maintained, commercial support available) but can also be used to automatically summarize information, e.g., the last activity of the maintainer or the typical recent issue response latency. Beyond shield.io's template for a maintained badge (maintained no! (as of 2022) , not widely used), we are not aware of any more advanced transparency mechanisms regarding maintenance status or abandonment risk, although efforts seem underway at least as part of the CHAOSS project [101].

Supporting the Construction of Abstraction Layers (Preparing for Risk of Abandonment). The building and deploying of abstraction layers (Section 3.5.2) was widely credited with significantly reducing the *reaction effort*, but building abstraction layers was often a time-intensive process that did not scale well to a large number of dependencies. As an alternative to the vast amount of research on API migration (see Section 2.1), refactoring tools could be enhanced to provide direct support for creating abstraction layers. Additionally, developers could write reusable abstraction layers for certain libraries that can be shared with other developers to make subsequent migration between libraries easier (similar to how JDBC abstracts from individual database protocols).

Advertising Alternatives (Addressing Abandonment). Switching to an alternative dependency (Section 3.5.3) is a common solution when faced with abandonment, but finding a suitable one can be challenging, as it is not always clear where to look. Also finding actively maintained forks can be difficult in projects with many forks. Making suitable alternatives easier to find can reduce *reaction efforts*. Interviewees mentioned several specific strategies for advertising alternatives: (1) posting pointers to alternatives on the abandoned dependency's repository page (e.g., notes in an issue thread about abandonment); (2) promoting alternatives on relevant online forums (one interviewee (P30) reports creating posts on relevant Subreddits like r/python when they have a new release celebrating it and giving an overview of the project and its features); and (3) creating blog posts discussing alternatives. Platforms could highlight posts for alternatives, curate links to external resources, and highlight active forks. They could also gather a lot of information automatically, for example, by scraping what other projects have migrated to in the past.

Supporting Dependency Migration (Addressing Abandonment). Some interviewees expressed how each

time they face dependency abandonment, it feels like there is no existing game plan or guidance to refer to, and that they have to figure out how to move forward on their own. For example, *"we really do need rubrics or tools or something because every time a project becomes abandoned, or we think it might be abandoned, we feel like we're winging it. We feel like we're dealing with it for the first time and we don't have a run book for that, and I doubt anybody really does"* (P4). Beyond just suggesting possible alternatives, platforms, tools, and community initiatives can provide support for *how to* deal with an abandoned dependency, such as creating a *migration guide*, showing examples of how to use alternative APIs, or even to attempt API migration (semi-)automatically. Such information can be curated with community inputs or generated from activities in other repositories, which could help reduce developers' *reaction efforts* by minimizing the amount of trial-and-error and manual work required to address a given dependency's abandonment.

3.6.3 The Volunteer's Dilemma and Reducing Community Effort

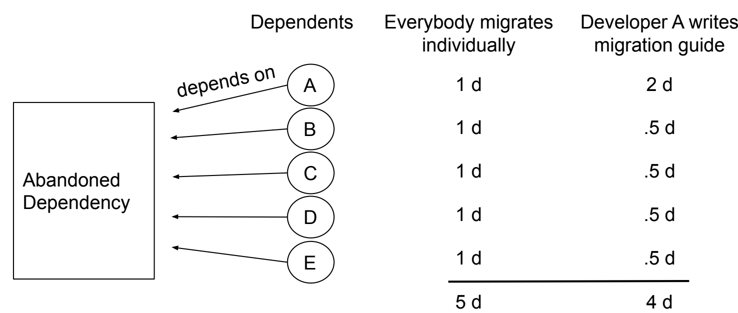


Figure 3.3: Illustration of the volunteers dilemma for dealing with abandoned dependencies: A developer who invests extra effort in writing a migration guide can save all other developers migration effort (measured in days of effort). Writing a migration guide is efficient for the entire community, though more expensive for the developer creating it.

The previous two sections discuss the various actions used by developers to reduce the *anticipated cost of abandonment*, each at some investment cost. However, the person who makes the investment and the person who benefits from said investment does not necessarily have to be the same. The actions of one developer can benefit many others. For example, tool builders and platforms like GitHub can invest in making it easier to find and migrate to alternatives, which can benefit *all* the developers who use such platforms. Similarly, many interviewees benefited from the actions of other individual developers when figuring out how to address dependency abandonment, including finding pointers to forks or alternatives, learning about abandonment early through community channels, finding blog posts explaining migration, benefiting from posted bug fixes, and receiving help finding new maintainers (Sections 3.5.2–3.5.3).

We call these investments designed to benefit others *community-oriented solutions*. They reduce the *redundant reaction effort* expended by subsequent projects facing the same abandoned dependency, as we illustrate in Figure 3.3. Creating community-oriented solutions requires *additional effort* on top of the *reaction effort* required for a developer to address the abandonment in their own project, for example, by writing a blog post after fixing their own problem.

However, beyond the small handful of interviewees who reported doing so (P2a, 2b, 13, 30), interviewees did not typically consider creating community-oriented solutions, because they had many competing demands, no incentive to invest the additional effort, or simply had not considered it. This situation is an example of the *volunteer's dilemma* [76], which is canonically formalized as a game with a group of

April 2, 2026
DRAFT

members, where each member can decide whether to volunteer and incur the associated cost of producing a public good that all group members benefit from collectively, and if nobody volunteers, the entire community loses [280].

The volunteer's dilemma has been studied both theoretically and empirically in fields like economics, social psychology, organizational behavior, and game theory for decades. Surveying this wealth of knowledge, we collected some practical solutions that we suspect may encourage the creation of community-oriented solutions for dependency abandonment:

Reducing the Cost of Creating Community-Oriented Solutions. Increasing volunteering costs reduces the individual likelihood of each group member volunteering and the overall likelihood that the public good will be produced [117, 143]. This suggests that one of the most straightforward ways to support the creation of community-oriented solutions is by decreasing the *additional effort* required to do so. For example, creating a uniform and visible place on abandoned projects to discuss solutions can make it easier for community members to post about alternatives or share advice. We conjecture that tools, especially platform features in GitHub, have substantial potential to facilitate and streamline the sharing of information about how to deal with specific abandoned dependencies.

Nudging Potential Volunteers. Where relevant characteristics of group members are visible, nudging [39] people who are in a better position to volunteer and have lower volunteering costs can be an effective way to encourage creating the public good [157]. For example, a bot could nudge developers who already created an active fork by suggesting they advertise it on the abandoned dependency project. More research is needed to determine who is in a 'favorable' position and to design nudges that fit into existing workflows and practices.

Priming Potential Volunteers and Re-framing Volunteering. Priming potential volunteers to be in a charitable or competitive mindset can impact the likelihood of an individual volunteering [171]. This suggests that framing the creation of community-oriented solutions as a deliberate act to benefit the larger open-source community could encourage such creation and normalize it as a common action. Also estimating the possible impact of creating a community-oriented solution could be motivating for some. More research on the attitudes of developers toward various community-oriented actions and how actions for abandoned dependencies fit in could help design a supportive framing.

Rewarding Volunteers. Research studying the effects of rewards and punishments on the volunteer's dilemma found that rewarding volunteers who step up can be more effective than punishing potential volunteers who do not, suggesting that shaming strategies are less effective than positive reinforcement [157]. For example, since many developers are motivated by helping others and supporting their community [98], highlighting the estimated community-wide benefit of creating a community-oriented solution could illustrate the good volunteering does and how such actions align with their motivations. Public recognition for community-oriented solutions, such as awards at community events or even just listing them as part of a GitHub profile, could provide further incentives and highlight positive role models. Gamification approaches could be deliberately used, such as awarding badges or points, but they also come with risks [107]. More research is needed to understand which reward mechanisms are effective in encouraging community-oriented solutions.

Facilitating and Encouraging Group Discussion. In general, incorporating communication into coordination games tends to improve outcomes and facilitate coordination [32, 41, 56, 57, 90]. Facilitating and encouraging communication between agents increases transparency and awareness of the choices others are making, giving potential volunteers more complete information, thus allowing them to make more educated decisions about whether to volunteer [90]. This suggests that by improving transparency about what others who face the same abandoned dependency have done or plan to do, developers are able to make more informed decisions themselves. For example, providing discussion forums on aban-

doned projects could help with highlighting demand (or lack thereof) for solutions. Tooling that creates transparency about how others have or have not already dealt with the abandoned dependency (see Section 3.6.2) can provide insights about the scope of the problem and assurance about the usefulness of a proposed community-oriented solution. More research in communication patterns, information needs, and automated identification of how others dealt with abandonment can help to deliberately design communication spaces and transparency mechanisms.

3.7 Summary

Assuming that not all projects will be maintained forever, we refocus sustainability research on how to sustainably *use* open-source software given the risks and realities users face today. We conducted interviews to study how developers prepare for and deal with open-source dependency abandonment. We catalogued the varying beliefs and philosophies surrounding dealing with dependency abandonment, preparations and considerations used to mitigate risk proactively, and solutions used to deal with abandonment. Developers generally navigate the tradeoff between proactive preparation and later potential reaction costs, with little information about the actual costs involved. We particularly highlight that sharing solutions can benefit many others facing the same problem, but that such sharing is not common. Looking at this problem through the lense of the volunteer's dilemma, we suggested future research directions inspired by findings in game theory and social psychology. We hope the strategies and insights can be helpful to the many developers who navigate abandoned dependencies daily.

3.8 Data Availability

The interview guide and a table with summary statistics for the interviewees are available in the publicly-accessible artifact hosted on Zenodo [185]. DOI 10.5281/zenodo.8102547

April 2, 2026
DRAFT

Chapter 4

Quantifying Prevalence of and Response to Abandonment At Scale

4.1 Introduction

Despite widespread concerns surrounding dependency abandonment, we know very little about its prevalence or how developers react in practice. Research has primarily focused on *preventing* or *predicting* abandonment by reducing disengagement [23, 42, 182] or improving onboarding [88, 106, 253], rather than studying what happens when abandonment occurs. A key exception is our recent interview study with developers where we studied their *perceptions* of abandonment, but without quantifying the prevalence or reactions in practice (cf. Chapter 3).

This chapter describes the work done in our paper *Understanding the Response to Open Source Dependency Abandonment in the npm Ecosystem* [187]. We report on a large-scale, quantitative study exploring the prevalence of, impact of, and response to the abandonment of widely-used packages in the JavaScript npm ecosystem. Specifically, we design an approach to detect abandonment at scale, collect a large sample of dependent projects that were exposed to abandonment across all of GitHub, and observe their responses to abandonment. We compare reactions to abandonment with other dependency management practices of updating dependencies with and without known vulnerabilities. Finally, we use statistical modeling to investigate what factors impact likelihood and speed of abandoned dependency removal. Specifically, we ask the following research questions:

- **RQ1a** How prevalent is abandonment among widely-used npm packages?
- **RQ1b** How many open source projects are exposed to abandoned dependencies?
- **RQ2a** How often and how fast do dependent projects remove abandoned open source dependencies?
- **RQ2b** How does this compare to how projects update dependencies in general?
- **RQ2c** How does this compare to how projects update dependencies with security vulnerability patches?
- **RQ3** What dependent project characteristics are associated with removing abandoned dependencies?
- **RQ4** How does announcing the abandonment status of a package impact how fast dependent projects remove the abandoned dependency?

Even with a conservative operationalization, we find that the abandonment of widely-used packages is

April 2, 2026
DRAFT

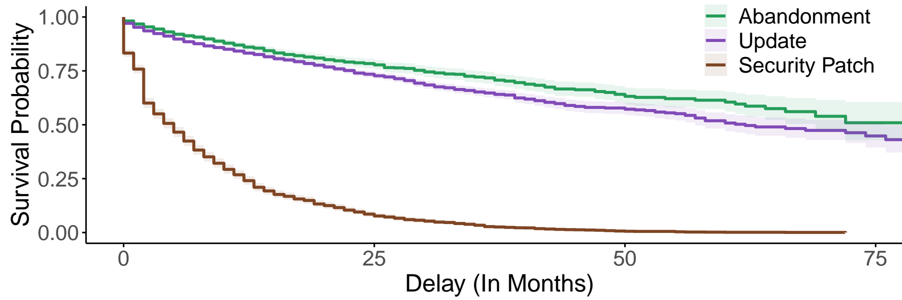


Figure 4.1: Survival probability for event "dependency event is not resolved" w.r.t. the date of event occurrence within dependent project's lifetime.

prevalent, with 15% of widely-used packages becoming abandoned within our six-year observation window. Those abandoned packages expose many dependents, but average direct exposure even for widely-used packages is lower than might be expected, suggesting that collaborative *responsible sunsetting* strategies might be feasible. Developers seem to care about abandonment – 18% of exposed projects remove the abandoned dependency, which is roughly comparable with other dependency management practices such as installing updates (cf. Figure 4.1), but reactions to abandonment tend to be delayed – in fact, removal of abandoned dependencies strongly correlates with other good development practices, including regular dependency updates. Finally, making the abandonment status of a package clear can help exposed projects react faster (1.58 times higher chance of reaction on average, at any point in time), suggesting opportunities for low-effort transparency mechanisms to help exposed projects make better, more informed decisions. Overall, our results suggest many opportunities to foster *responsible use* of open source for developers and *responsible sunsetting* for maintainers.

In summary, we contribute (1) a detailed methodology for detecting abandoned packages at scale; (2) a quantification of the prevalence of widely-used package abandonment in the npm ecosystem; (3) a large-scale analysis of the response to abandonment of exposed projects across GitHub and a comparison to other dependency management practices; (4) a logistic regression model illustrating which dependent project characteristics impact the likelihood of removing abandoned dependencies; and (5) a survival model demonstrating the impact of providing explicit notice of abandonment on dependent project removal rates.

4.2 Detecting Open-Source Package Abandonment

To study abandonment at scale, we first design two conservative heuristics and a manual validation process for the heuristics. Specifically, we look for cases of abandonment with a clear abandonment event, with the evidence coming from either (1) documentation or metadata explicitly indicating a package will not receive further maintenance *(explicit-notice abandonment)*; or (2) shifts in activity patterns from regular maintenance to not receiving any development activity for two years *(activity-based abandonment)*. We intentionally pursue a high-precision detection strategy (detecting real and clear abandonment events), while accepting lower recall (missing some cases of abandonment, e.g., projects that slowly became inactive over an extended period of time). This will result in an undercount in RQ1 (scope of abandonment) but increases confidence in analyses based on our data (RQ2-RQ4).

April 2, 2026
DRAFT

4.2.1 Explicit-Notice Abandonment

We identify packages as abandoned when developers explicitly express their intention to no longer maintain them. First, we manually searched for explicit signals of abandonment (e.g., GitHub "archive" flag) through the repositories of packages that had been likely abandoned based on long periods of observed inactivity. Once we identified some explicit signals, we then searched for additional packages with those signals, verified the precision of the signal, and used the same process to search for additional signals of abandonment. We repeated this process until we found no additional reliable signals. In the end, we used the following three signals:

- *Github archive flag:* In 2017, GitHub introduced the ability to archive a repository [10]. Archived repositories are read-only and display an '*archived*' banner on GitHub. We consider a package as abandoned when its GitHub repository is archived and use the date of archival (shown on GitHub) as the date of abandonment.
- *No-maintenance-intended badge:* Some developers declare their intention not to provide maintenance with a '*no maintenance intended*' badge in their README file [13]. We consider a package as abandoned if its GitHub repository has a README file containing the badge and use the date of the commit introducing the badge as the date of abandonment.
- *Other abandonment description in README:* Finally, developers can textually describe that their package has reached the end of its life in many ways in their README file, such as '*'[project] is no longer maintained*" [3]. We consider a package as potentially abandoned if the first 10 lines of the README (excluding URLs and code) contains one of the following text fragments: '*abandoned*', '*deprecated*', '*no longer maintained*', '*no longer supported*', or '*unmaintained*' (fragments identified iteratively as described above). We manually checked all matching packages that did not also have one of the other explicit-notice signals. We consider a package as abandoned if its repository's README states so and use the date of the README change that introduced this description as the date of abandonment.

If a package contained multiple of the above signals, we used the earliest date as the date of abandonment.[1]

4.2.2 Activity-Based Abandonment

Detecting abandonment from a package's activities is nontrivial – some packages may be considered *feature complete* [51] requiring very little maintenance or changes to the code. Therefore, we seek cases where there is a clear shift from regular repository activity levels to a sharp drop indicating abandonment. Conservatively, we look for 2+ years of regular maintenance (operationalized as 10+ total events from contributors per year, including commits, issue comments, and issue close events) representing the *pre-abandonment* phase followed by 2 years with no activity from contributors representing the *abandonment* phase. This provides a relatively clear abandonment point, enabling us to observe reactions downstream. We consider the date of abandonment to be the time of the last commit i.e., the beginning of the *abandonment* phase since that was when the activity dropped off.

We experimented with different heuristics and thresholds (for time and permissible residual activity) and arrived at the design above after reviewing how related work has operationalized abandonment [23, 52, 53, 138, 184, 272] and exploring activity patterns before and after explicit notices were added to a sample of packages meeting our explicit-notice heuristic. We repeatedly tested the robustness of our heuristic with different thresholds, manually validated the accuracy on a sample of packages, and determined that allowing even a small amount of residual activity (e.g., 3 commits per year) introduced too many false

[1] Now, npm also has a `deprecation` flag. We did not include it in our analysis because it was introduced near the end of our observation window and does not record the date when the flag was added.

April 2, 2026
DRAFT

positives of feature-complete but still sporadically-maintained packages; we found similar problems with a smaller observation window. We found that a strict threshold of no residual activity for 2 years had an almost perfect precision while still finding many abandoned packages.

4.3 RQ1: Abandonment Prevalence and Exposure

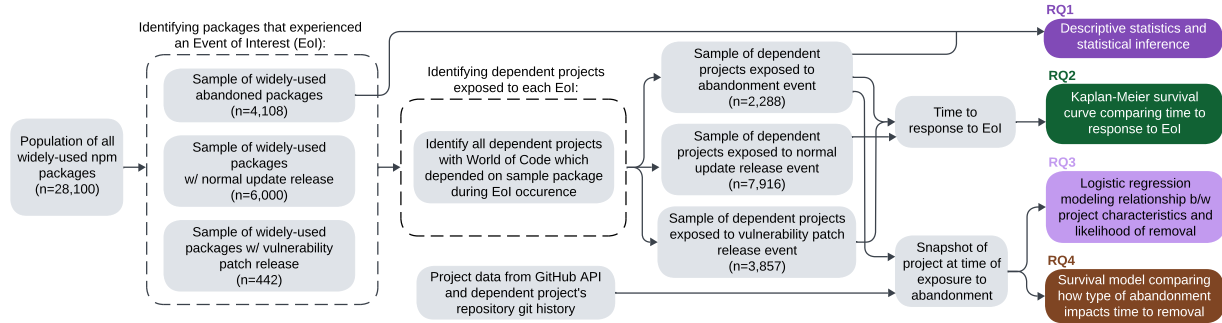


Figure 4.2: Overview of our data collection and analysis.

We begin by quantifying the frequency of abandonment among widely-used *packages in npm*. We focus on widely-used packages, rather than including the many more that never gained traction, as a way to focus on digital infrastructure.

We then estimate exposure of abandonment on projects in GitHub that were active and depended directly on an abandoned package at the time of its abandonment (cf. Figure 4.2). We estimate exposure for *all of GitHub* without restricting the analysis to popular or widely-used projects, because abandonment affects all kinds of users of open source, whether they build popular libraries or applications, or just maintain personal projects. Different users may have different views and perceive different pressures (and may even have policies requiring them to avoid end-of-life dependencies), as we will explore later, but abandonment and sustainability are important for all users of open source who plan to maintain their own project. Users of open source dependencies who write closed-source applications are obviously not captured by our analysis; exposure rates should be considered as a lower-bound estimate.

4.3.1 Research Methods

We identify abandoned packages and exposed projects with data from npm, GitHub, and World of Code [169]. We restrict our analysis to abandonment in a six-year observation window from January 2015 to December 2020, for which we can collect all relevant data at scale and which starts after npm has been established and widely used.

Identifying Abandoned Widely-Used npm Packages. To scope our analysis to widely-used packages that can have a substantial impact on the ecosystem if abandoned, we consider only the 36,164 of 1,063,835 npm packages (in 2020) that had at least 10,000 downloads in any month of our observation window (per npm download statistics [6]). We use downloads (rather than reverse dependencies) since they capture both public and private use of packages. Next, we excluded 940 packages because they shared a GitHub repository with other packages, and we cannot clearly attribute the repository-level activity data (e.g., issues) to a single package – this is common when a repository contains the source code for multiple related packages (sometimes called a *project foundry* or *mono-repository*). We also filtered out 7,124 packages that never had more than 10 total activities by contributors (cf. Sec. 4.2.2) in any year of our

April 2, 2026
DRAFT

observation window – they may have been abandoned before our observation window or have low activity levels indistinguishable from abandonment. After filtering, the dataset contains 28,100 widely-used npm packages.

We then identify which of these packages were abandoned and when, as described in Section 4.2, using both the explicit-notice and activity-based detection approaches over the entire observation window. Because the activity-based abandonment definition requires two years of activity observation before abandonment and after, it can only occur in the two middle years of our observation window, whereas explicit-notice abandonment can occur in all six years.

Identifying Exposed Dependents. Next, we identify dependent projects across all of GitHub (not just npm packages) that were directly exposed to abandonment. In contrast to prior work on dependency management [72, 73, 215, 290], we explicitly consider all projects rather than just reverse dependencies within npm to capture the impact on open source developers broadly, not just on other package maintainers. Searching across all of GitHub imposes substantial challenges due to its scale and limitations of its search APIs. Instead, we use *World of Code (WoC)* to find all dependents of the detected abandoned packages. WoC is a large scale analysis infrastructure that indexes and curates nearly all public open source code, intended for research studying software supply chains [168, 169]; we use version V, the latest at the time of this analysis.[2] Specifically, in addition to content of the files, commits and trees extracted from all repositories, WoC also parses each version of every file (or blob using git terminology). Projects that rely on npm packages use specific file named packages.json to specify all direct dependencies. As a result, for each commit modifying packages.json file WoC provides a list of upstream dependencies as b2Pkg (blob-to-package) map. We queried WoC using indexes b2c (blog-to-commit), c2p (commit-to-project), and b2Pkg to retrieve all GitHub projects, excluding forks, that ever depended on any of the abandoned packages in a *package.json* file in their root directory.

Due to the vast number of candidate dependent projects returned by WoC (usually millions) and the nontrivial analysis costs, we perform the analysis on a large sample of 60,000 randomly selected candidate dependents and extrapolate exposure rates to the entire population statistically. We further checked each candidate dependent project in our sample by cloning the project's repository and analyzing the dependencies at the time of abandonment. We use the same step to also detect whether the repository had any commit activity after the time of abandonment. This allowed us to identify the subset of dependent projects that were actively depending on an abandoned project at the time of abandonment who also had at least one commit after abandonment occurred (to ensure they were not entirely inactive themselves).

Limitations. As discussed in Section 4.2, construct validity for *abandonment* is difficult to establish. Despite best efforts to design and validate meaningful but conservative heuristics for detection, we may not capture all notions of package abandonment, e.g., adding notices in an external blog or entirely stopping maintenance after years of minimal maintenance activity. For the purpose of this study (actions taken as a result of abandonment) we designed our heuristics to be conservative, hence our abandonment numbers should generally be seen as a lower bound. Additionally, because there are many ways to define abandonment and because we consider multiple definitions of abandonment, there could be packages that meet one definition of abandonment while not meeting another. For example, the maintainers of a package that qualifies as explicit-notice abandoned because they posted a note in the README stating the package will no longer be maintained, could post minor patches down the line potentially making the package not qualify as activity-based abandoned. More broadly, it's possible that projects resume their activity even after long periods of apparent abandonment [23]. While the scenarios in which such revivals happen remain unclear (we recommend that future research investigates this), as a robustness test given that

[2]GitHub itself has a *Dependency Insights* feature. However, the functionality is closed-source and poorly documented, and our initial experiments showed too many incorrect dependency entries for it to be trustworthy.

we have the benefit of hindsight, we also estimate our models on subsets of our data after excluding all packages in which we detect any subsequent repository activity after the two-year dormant period (and, correspondingly, their downstream dependents). The results (magnitude and direction of coefficient estimates), part of our replication package, are entirely consistent with what we report below (though we lose some statistical power as the dataset size shrinks).

We may also miss exposed dependent projects that have since been deleted (and have not been not archived in WoC). In addition, our analysis relies on timestamps of commits, which are usually but not necessarily reliable. In addition to the direct exposure we measure here, there may be instances of indirect exposure where none of the immediate upstream packages are abandoned, but at least one of these packages depends directly or transitively on an abandoned package. We chose not to include these transitive dependencies since developers tend not look beyond direct dependencies (see, e.g., Dey et al. [75]) nor do they have much power to change projects further upstream on such complicated issues as replacing functionality.

Finally, our results are specific to heavily downloaded packages in npm within our observation window. Abandonment may be more common among less downloaded packages and dynamics may be different in other ecosystems. Our study cannot capture the behavior of developers using open source dependencies in closed sourced projects.

4.3.2 Results

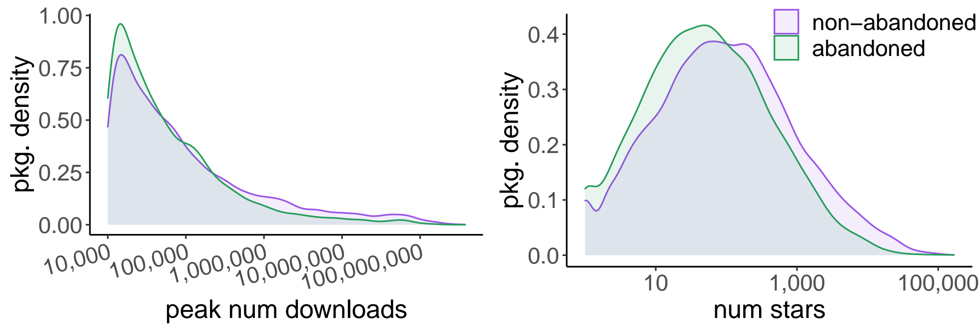


Figure 4.3: The distribution of peak download counts during our observation window and current star counts (March 2024) for both non-abandoned and abandoned widely-used npm packages are similar.

Of the 28,100 widely-used npm packages in our dataset, we identified 4,108 (15%) as becoming abandoned during our six-year observation window. Abandonment events were distributed fairly uniformly across the observation window, without clear patterns or peaks. In addition, abandoned packages were similar to non-abandoned packages, e.g., in terms of peak downloads and stars (cf. Figure 4.3).

Our relatively large sample size for downstream dependent projects affords high generalizability – approximately 0.4% margin of error at 95% confidence level. Assuming the same abandonment rate of 15%, we estimate[3] that the 4,108 abandoned packages exposed $283,207 \pm 2,096$ GitHub projects (not including forks) who had an abandoned package as a direct dependency at the time of its abandonment (average 69 projects exposed per abandoned package). Of those projects, we estimate that $78,023 \pm 624$

[3]The estimates are based on 2,046,047 candidate exposed projects retrieved with World of Code, from which we randomly sampled and analyzed 60,000. Among those we found 8,305 exposed dependents to the abandoned packages; of those 2,288 had commit activity after exposure.

April 2, 2026
DRAFT

GitHub projects had any commits after exposure, i.e., they were not abandoned themselves at the time and might need to respond.

The directly exposed and subsequently active projects vary widely in their characteristics. They include many small projects, including hobby projects and personal websites, but also popular libraries and end-user products. About half have no stars on GitHub, but 8% have 100 or more stars.

Key Insights: Of the 28,100 widely-used npm packages, 4,108 (15%) were abandoned during our observation window. We estimate that 78,023 dependent projects on GitHub, still active at the time of abandonment, were directly exposed.

4.4 RQ2: Responding to Abandonment

We detect how often and how fast dependent projects exposed to the abandonment of widely-used packages remove the package after abandonment. Essentially all strategies to respond to abandonment that do not involve preventing it (e.g., contributing financially, taking over maintenance) involve removing the dependency (e.g., replacing it with an alternative, switching to a fork, copying the code, removing the functionality) [184]. Additionally, we compare the response to abandonment to other established dependency management practices, specifically, to developer responses to updates of their dependencies with and without known security vulnerabilities.

4.4.1 Research Methods

We detect responses to abandonment (RQ2a) as well as responses to updates (RQ2b) and security patches (RQ2c) with the same three-step research design: (1) We collect a set of *events of interest* among widely used npm packages – package abandonment, package updates, security patches. (2) We identify active projects directly *exposed* to these events. (3) We determine whether and when the exposed projects subsequently *responded* – by removing or updating by analyzing the commit history of their *package.json* file(cf. Figure 4.2).

To scale the analysis, we randomly sample both (a) from the set of all possible events to analyze and (b) from the set of all exposed dependents for those dependencies. Note that we use a consistent approach to collect data for different dependency management practices, rather than comparing our abandonment data against previously published results on other dependency management practices – this allows for a direct comparison and avoids potential issues caused by the differences in research design and analyzed populations in past research [46, 72, 73, 74, 144, 177, 218, 290].

Removal of Abandoned Dependencies (RQ2a). To analyze the removal of abandoned dependencies, we rely on the data from RQ1, namely the 4,108 abandoned packages we identified and the sample of 2,288 directly dependent projects that were active after the time of abandonment (among the 60,000 dependent projects we analyzed). For each dependent, we analyze their commit history after the abandonment event to measure whether and how long it took them to remove the abandoned dependency from their *package.json* file. While the event must happen within our observation window ending in December 2020, we analyze subsequent reactions until a cutoff date of September 1st, 2023.

Dependency Updates After New Package Releases (RQ2b). To establish a baseline of common dependency management practices, we select a sample of package updates among the 28,100 widely-used npm packages from RQ1 as events of interest. Specifically, to generate a sample similar in size to abandonment, we first identify all releases of widely-used packages released within our six-year observation window, and

April 2, 2026
DRAFT

then randomly sample 6,000 of these releases making sure each is from a unique package. For each update event, we use the same search strategy with WoC described in Sec. 4.3 to detect candidate GitHub projects that directly depended on the package of interest at some point in time. We then analyze a large random sample of those candidate dependents ($> 500,000$), using the same process described in Sec. 4.3 to identify the subset that depended on the package of interest at the time of the update and who had had any commits after the event. If the dependent uses floating version constraints (patterns that match multiple versions, e.g., `^1.4.2` to match release 1.4.2 and any later releases before 2.0.0) that allowed them to automatically update at the time of the event, we discard the dependency from our analysis as it does not require developer intervention to update the dependency. This results in 7,916 observations.

Dependency Updates After Security Patches (RQ2c). As a special version of detecting responses to package releases, we analyze responses to releases that patch known security vulnerabilities, which are usually considered more urgent than other updates. We identified package releases with known security vulnerabilities and corresponding package releases that patch the vulnerability using the OSV database [9]. We select the first release that patches the vulnerability and its release date as the event of interest – that is, we study whether and how fast developers respond to the security patch. We found 442 packages among our set of 28,100 widely-used npm packages that had at least one vulnerable release and corresponding patch release within our six-year observation window. For each of these packages, we randomly selected one patch release, resulting in 442 events of interest. For each security patch release of interest, we use the same search strategy with WoC to detect candidate GitHub projects that depended on the package of interest at some point in time. We then analyze a large random sample of those candidate dependents ($> 500,000$), identifying the subset that depended on a vulnerable version of the package of interest at the time of the event occurrence and who had had any commits after the event, again discarding issues that were patched automatically due to floating version constraints. This results in 3,857 observations.

Analysis. To answer the research questions, we use survival analysis, which specializes in modeling *time-to-event data* and providing estimates of the survival rates for a given population [132]. Survival analysis is designed to account for right-censored data like ours, where the occurrence of the event of interest is only recorded for cases that have experienced the event, and for other cases their data is "censored" because (a) the event was not (yet) observed during the period of observation or (b) they withdrew from the study during the observation period. In our study, we consider a project to be withdrawn and therefore censored if it becomes inactive during our observation period (which we operationalize using the date of its last commit). Additionally, survival analysis can account for the fact that we can observe reactions for different periods of time for different events (for earlier events, developers had longer to react). Specifically, we use the Kaplan-Meier estimator [136], which is a common non-parametric statistic for estimating survival functions [60].

Limitations. To capture the reaction to *average* events, we intentionally do not stratify our analysis by major/minor/patch release or vulnerability severity. Behavior may differ between different subtypes of events, which is not the focus of our study. Similarly, a security patch is not automatically urgent, since the vulnerability may not be exploitable; again, our study only reveals average practices and does not set normative expectations. Finally, our study does not capture the more nuanced behavior of floating dependency declarations when locking dependencies with npm – in such cases, updates matching the versioning pattern may not be fully automated; excluding those cases helps us avoid ambiguity about what actions developers take, but may miss some actions. Limitations from RQ1 also apply.

April 2, 2026
DRAFT

4.4.2 Results

Only 18% of directly dependent projects with any development activity after the abandonment date ever remove the abandoned dependency before our cutoff date (419 of 2,288 in our sample) – the vast majority of dependent projects did not remove the abandoned dependency. Among dependent projects that removed the abandoned dependency, the average time to removal is 13.5 months. Consistent with past research [73, 144], we also observe that many developers do not update dependencies, even those with security vulnerabilities: Only 17% (1,366 of 7,916 in our sample) respond to a random dependency update before our cutoff date with an average time to update of 10.5 months; and 44% (1,720 of 3,857 in our sample) install a patch to a security vulnerability with an average time to update of 8.5 months.

We show survival curves indicating the percentage of dependent projects that react to package abandonment, package updates, and security patches respectively within a given time window in Figure 4.1, illustrating that security patches are installed at higher rates and faster than other updates and that developers react to abandonment generally at similar rates and with similar latency to random dependency updates. Note that survival rates in the plot are lower than what may be expected from past research, because we censored projects if they became inactive during our observation window – the lack of updates can often be explained by dependent projects becoming inactive whereas dependent projects that remain active for long periods of time after security patch release are much more likely to eventually update.[4]

Key Insights: The response rate for abandonment is similar to updates and lower than the rate for security patches.

4.5 RQ3: Characterizing Responsive Dependents

Next, we study population-level differences between the characteristics of projects that remove abandoned dependencies and those that do not.

4.5.1 Research Methods

Using our sample of 2,288 dependent projects directly exposed to an abandoned dependency, identified in RQ1, we take a snapshot of each project *at the time of exposure* to abandonment, operationalize numerous factors representing different project characteristics (hypotheses H_1-H_6 described below), and use logistic regression analysis to model the relationship between project characteristics and the likelihood of removing the abandoned dependency.

Hypotheses and Variables. Specifically, the binary response variable is whether the abandoned dependency was removed within two years of abandonment. In addition, we test hypotheses about the association between the following variables and the binomial outcome:

Dependency Count (H_1). We expect that projects with fewer dependencies are less likely to remove abandoned ones. Such projects may pay less attention to dependency management in general and may thus do it less.

Dependency Management Practices (H_2). We expect that projects that manage dependency updates and security patches more actively are also more likely to respond to abandonment. Here we use two

[4]As described above, our results do not include dependents that can automatically update dependencies due to floating dependency version declarations. This would account for an additional 33% immediate random dependency updates and an additional 70% immediate security patch updates, also shown in corresponding survival curves in our supplementary material [**?**].

April 2, 2026
DRAFT

variables: First, we model whether there was evidence of software composition analysis (SCA) tool use in the year before exposure (i.e., tool configuration files, README badges, or commits by bots), namely Dependabot, Renovate, Greenkeeper, Snyk, DavidDM, and Gemnasium. Second, we use the standard heuristic by Zerouali et al. [290] to calculate the average *technical lag* of all the dependencies the project had at exposure excluding the abandoned one.

Activity (H_3). We expect that more actively developed projects are more likely to respond to abandoned dependencies because developers spend more time on the project overall. We consider two variables: First, we calculate *dependency churn*, i.e., the total number of times the project changed any of its dependencies in the year before exposure. Second, we collect the total *number of commits* in the year before exposure.

Number of Maintainers (H_4). We expect that projects with more maintainers are more likely to respond to dependency abandonment because they have more capacity for maintenance (and dependency management) work. Operationally, we count the number of contributors responsible for the top 80% of commits in the year before exposure.

Corporate Involvement (H_5). We expect that corporate-led projects and projects with more corporate involvement may be more likely to respond to abandoned dependencies because they are more likely to follow explicit end-of-life policies and to explicitly allocate resources to dependency management than volunteer-run projects [25]. Operationally, we check for the presence of commits made in the year before exposure by contributors using a corporate email domain, using the list of corporate domains identified by Spinellis et al. [248].

Governance Maturity (H_6). We expect that projects following governance best practices are more likely to respond to abandoned dependencies, conjecturing that responsiveness to abandonment is also seen as a best practice. In particular, we consider six governance practices: having a README, a license, issue templates, pull request templates, contributing guidelines, and a code of conduct, in line with past research associating those with project success [51]. Operationally, for each project, we start collect six binary flags indicating the presence files for each practice at the time of exposure. We then compute a latent trait model [27] to reduce the dimensionality of this dataset. The model assumes that the dependencies between the six observed variables can be interpreted by a few latent variables. The model with a single latent variable fit our data best, confirming that the six indicator variables mostly capture one underlying construct. Finally, we computed the continuous variable *governanceMaturity* representing the factor scores (or "ability" estimates) for the observed response patterns as an operationalization of this latent construct, and used this variable in our subsequent modeling.

For all of the above variables, we collect the relevant data from the GitHub API or from the repository's git history. Where possible, we follow measures developed and validated in past research. We manually validated the construct validity of each factor using a sample of projects to avoid systematic errors by manually verifying that the factor seemed to indeed capture the intended data accurately.

Modeling Considerations. Before estimating the model, we took several steps to ensure model quality and validity. First, after manually examining the distribution of each variable, we removed extreme outliers for variables with highly skewed distributions (top 1% or fewer points), to reduce the risk of high-leverage points biasing our regression estimates. We further removed 7 repositories we failed to clone (likely deleted since) and 14 we failed to compute the average technical lag for, taking our total sample size down from 2,288 to 2,261 after both steps. Next, to reduce heteroscedasticity, we log-transformed the numeric variables as needed [97] (Fig. 4.4 indicates which variables were log-transformed). The model also includes control variables for repository age (controls for the project's development stage and software evolution) and size (controls for the size of the codebase, measured in bytes).

We further computed the variance inflation factors for each variable to check for presence of multicollinearity [80], checking to ensure each was lower than 5, a common threshold within the statistics

April 2, 2026
DRAFT

community [94]; none of the variables exceeded the threshold. To evaluate the model's overall goodness-of-fit we used McFadden's pseudo-R^2 measure [273]. Finally, we consider model coefficients significant if they are statistically significant at the $\alpha = 0.05$ level. For each variable, we report the exponentiated log-odds regression coefficient so as to report the transformed odds-ratio to aid in interpretation (i.e., the factor by which a 1 unit increase in the predictor increases the odds of the outcome occurring if the odds-ratio is greater than 1, or decreases by if the odds-ration is less than 1), and the significance level (p-value) (cf. Fig. 4.4).

Limitations. As is usual for this kind of work, despite the careful development and validation described above, the operationalized factors in our model can only capture part of the concepts they are intended to represent and measure. For example, we operationalize governance maturity by detecting six files in the repository: While grounded in prior research and supported by our latent trait (factor) analysis, it likely cannot fully capture the concept of maturity. There may also be additional dependent project characteristics and unobserved confounding factors that we did not include in our model. Our findings should not be considered an exhaustive list, but rather a list highlighting some of the characteristics that *are* constructive when modeling the removal of abandoned dependencies. Hence, as usual, care should be taken when generalizing our results beyond the studied measures.

4.5.2 Model Results

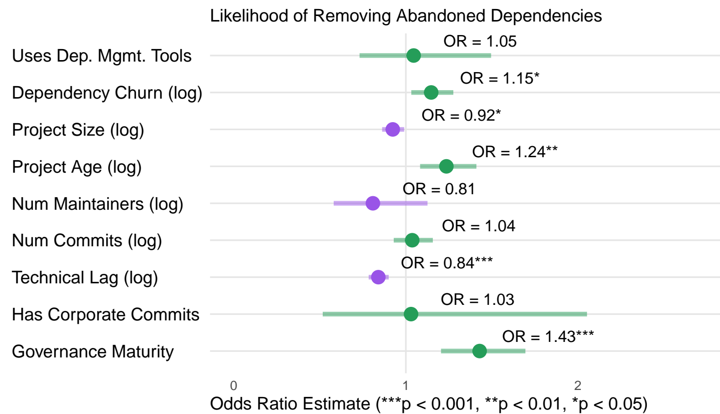


Figure 4.4: Summary of the multivariate logistic regression modeling the likelihood of removing an abandoned dependency within two years post abandonment. Confidence intervals (horizontal lines) for the odds ratios (OR) that do not intersect 1 indicate variables with statistically significant effects.

Regression results in Figure 4.4 show five significant effects. One is a strong positive effect of *governance maturity* (supporting H_6): For projects with one standard deviation higher governance maturity score we expect to see about 43% increase in the odds of removing the abandoned dependency. The model also shows that higher *technical lag* is, on average, statistically significantly negatively associated with the likelihood of removal (supporting H_2).

Projects with higher *dependency churn* are generally more likely to remove abandoned dependencies (supporting H_3). To demonstrate the interpretation of the exponentiated regression coefficient, for every factor e ($\simeq 2.72$) increase in the amount of dependency churn (note the log transformation), the odds of removing the abandoned dependency for the average project in our sample multiply by 1.15, holding all else constant. Additionally, as expected we observed a significant effect for both control variables *project age* and *project size*.

April 2, 2026
DRAFT

The explanatory variables *num dependencies* (H_1), *use of dependency management tools* (H_2), *num commits* (H_3), *num maintainers* (H_4), and *num corporate commits* (H_5) were not significant in the model meaning we have insufficient evidence to reject the null hypothesis that these factors do not impact the likelihood of abandoned dependency removal.

Key Insights: Projects that are more mature, have higher dependency churn, and keep more up to date on dependency updates are more likely to remove abandoned dependencies within two years.

4.6 RQ4: Influence of Announcing Abandonment

4.6.1 Research Methods

RQ4 extends RQ2 and RQ3 using the same data as RQ2, except we model the distinction in responses to packages that were explicitly declared as abandoned (explicit-notice) as compared to packages that just stopped maintenance (activity-based) as introduced in Section 4.2. Similarly to RQ2, we again apply survival analysis to model the time to removal of the abandoned dependencies, except now we use a multivariate Cox proportional-hazards model [110] to jointly control for all factors modeled in RQ3 (see Section 4.5.1 for factor definitions). Cox regression is commonly used in medical research for modeling the association between the survival time of patients and one or more predictor variables. In our case, we use Cox regression to estimate the effect of an explicit notice of abandonment on the rate of dependency removal events happening at a particular point in time, i.e., the "hazard rate."

4.6.2 Results

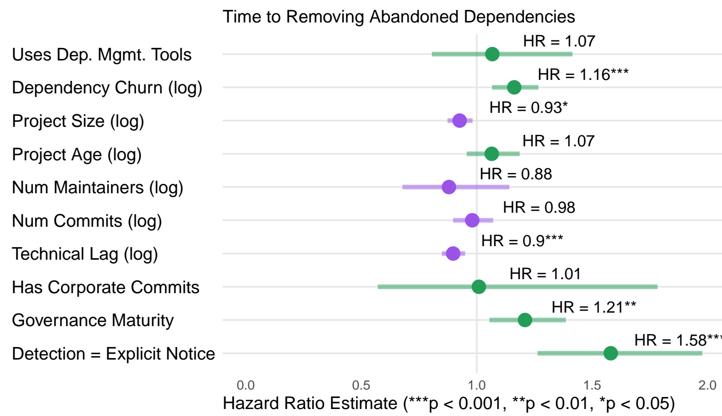


Figure 4.5: Summary of the Cox proportional hazards multivariate survival regression modeling the time to removing an abandoned dependency. Confidence intervals (horizontal lines) for the hazard ratios (HR) that do not intersect 1 indicate variables with statistically significant effects.

We observe after controlling for all the factors we hypothesized are associated with removing abandoned dependencies in RQ2, that there is a statistically significant relationship between the presence of an explicit notice of abandonment for a given dependency and an increased likelihood of the abandoned dependency being removed by downstream projects (cf. Figure 4.5). Holding the other covariates constant, dependencies with an explicit abandonment notice have 1.58 times the probability of being removed

April 2, 2026
DRAFT

within a time span than abandoned dependencies without an explicit notice (95% confidence interval of 1.26 to 1.98). This is in alignment with our expectations, because explicit-notice abandoned packages provide a clear signal to dependents and are more visible sooner.

Key Insights: Packages that provide an explicit-notice of abandonment tend to be removed at significantly faster rates compared to those that do not.

4.7 Discussion and Implications

The Scale of Abandonment. Our study finds that abandonment, even among widely-used npm packages, is fairly common. While many developers carefully analyze signals like the number of stars, responsiveness to issues, or number of contributors when adopting dependencies [148, 195] and past studies have shown several statistical predictors for survival [23, 51, 272], we were surprised by the scale of abandonment among packages that had healthy signals, were among the most popular packages on npm, and were generally similar in their distribution of stars and past activity to those with sustained maintenance. Given that open source maintainers may disengage for all sorts of reasons, such as losing interest, changing jobs, and starting a family [182], users of open source are likely not able to entirely escape abandoned dependencies with careful upfront vetting, but may also need to actively consider strategies to manage abandoned dependencies – an area also called for in our recent interview study [184] for which maintainers have with little existing support.

The Rippling Effects of Abandonment. Although abandonment rates are fairly high, we were surprised at the low rates of direct exposure. While GitHub's *Dependency Insights* page often show thousands to hundreds of thousands of dependent projects for the abandoned packages, the actual *direct* exposure of *active* dependent projects at the time of abandonment was not that high ($\mu = 19$, cf. Section 4.3.2). Many additional dependents of abandoned packages were abandoned even before the package's abandonment, so they are unlikely to care about it; many others adopted the package even after it was abandoned, possibly knowing and accepting that they will not receive updates.

Package abandonment has vastly more wide-reaching consequences when also considering *indirect* dependencies. On the one hand, this is good news since the few projects that depend directly on an abandoned package can potentially mitigate the consequences of abandonment for the many downstream projects that rely on the abandoned package only transitively. On the other hand, if the maintainers of these intermediate projects do not act, developers have very little means to do anything about indirectly used abandoned packages in their dependency graph. With an increased focus on the entire supply chain through software bill of materials (SBOMs), software composition analysis (SCA) tools and company-wide or agency-wide policies for sunsetting, this can cause a lot of pain for huge numbers of developers, vastly more than those directly exposed. We recommend that **maintainers of popular projects should be particularly attentive to monitoring and reacting to abandoned direct dependencies** due to their outsized lever to benefit the entire ecosystem.

For many abandoned packages in our sample, the small direct exposure would make it feasible to reach out to affected dependent projects (in the context of breaking changes, such proactive actions are not uncommon [33]). However, maintainers currently do not have tools to identify all active direct dependents (e.g., GitHub's Dependency Insights page reports too many false positives, vast numbers of inactive projects, and drowns out direct dependents among many more indirect ones while some dependents may not even be hosted on GitHub). **Researchers or practitioners should explore tools for more targeted outreach to direct active dependents.**

Allocating Resources to Sustain Open Source Communities. Discussions of open source sustainability are often centered on the most widely-used packages that form essential digital infrastructure and usually focus on keeping those projects alive, which may be arguably cheaper in the grand scheme of things than placing the cost for mitigations and replacements on all their dependents. However, the observed low rates of direct exposure may call that balance into question, especially if we can help the exposed projects with a migration guide or through other coordinated action (discussed as "community oriented solutions" in our prior work [184]). There is also a fairness argument regarding the degree to which the often-volunteer maintainers of packages do or should feel responsible to provide ongoing maintenance for their dependents, most of whom never contribute to the package in any way. As we argued previously [184], we believe **it is time to place more emphasis on the *responsible use* of open source rather than attempting indefinite maintenance**.

In our study, we explicitly consider all dependents, not only other packages in npm and not only packages or projects that are popular and form digital infrastructure themselves. That is, many of the exposed dependents are 0-star projects, including personal projects like maintaining a personal website – but all of them were still maintained for some period after exposure to abandonment. Less prominent dependents may have a more relaxed attitude toward abandonment, but they may also be less experienced in dealing with dependencies and likely spend less time on dependency management overall, thus making abandonment possibly even more disruptive. **More research is needed on whether and how to help such developers, rather than only helping and studying the most active developers or the most popular projects.**

Abandoned Dependencies in the Context of Dependency Management. Despite many calls for better dependency management, especially from a security perspective (recently even with the US White House joining in [14]), study after study shows that the majority of developers rarely update dependencies, even those with known vulnerabilities, and even when informed about problems by automated tools [29, 46, 72, 73, 74, 144, 150, 177, 215, 218, 218, 228, 246, 257, 290]. If developers do not patch known security vulnerabilities or even add dependencies with known vulnerabilities, should we expect them to care about abandonment? Our results show that different dependency management practices correlate. Developers who generally keep their dependencies up to date are also more likely to react to abandoned dependencies. When (or if) the larger open source community manages to improve dependency management practices in response to perceived higher stakes (e.g., the continuously increasing frequency of supply chain attacks), we expect to also see more people reacting to abandonment – therefore **support to help developers exposed to abandonment will only become more important.**

At the same time, abandonment is different. A dependency does not automatically and immediately become a problem when abandoned – impacts are often delayed and may not even occur in a dependent project's lifespan [184]. A large number of updates and vulnerability fixes can be captured with floating dependency versions (semantic versioning is a common practice in npm [69, 215], automating the "immediate reaction" to 33% of analyzed updates and and 70% of analyzed patch events; although this practice also raises its own security challenges [71, 72, 145]) and various SCA tools can inform and automate update actions. However, there is no equivalent default action or tool automation for abandonment. The decision to remove abandoned dependencies is closer in nature to decisions surrounding technical debt reduction and risk reduction (similar to trying to stay on top of updates to avoid painful large migrations and integration problems later [33]) than the more immediate urgency to patch known vulnerabilities. This is visible in our results (e.g., Figure 4.1) where fixing vulnerabilities is more likely and faster than reacting to abandoned dependencies, but reactions to abandoned dependencies are fairly similar to reactions to random dependency updates, even in the absence of any automated tooling.

Responsible Sunsetting and Effectively Signaling Abandonment. Our results show that many developers,

April 2, 2026
DRAFT

though far from all, care about abandoned dependencies but may not be aware of them or may observe a dependency for a lengthy period before taking action. Dependencies that are clearly marked as abandoned (*explicit notice*, see Section 4.2) are removed significantly faster than those that silently stop receiving maintenance (RQ4), suggesting that awareness matters. Based on our research, we can clearly recommend that **maintainers should place an explicit notice about abandonment of their package as their final action to benefit their dependents, costing very little effort to the departing maintainers**. We believe it is time to **establish best practices for responsible sunsetting of packages,** which may include leaving an explicit notice and possibly also reaching out to direct dependents.

In addition, **future research should explore the most effective way to present abandonment notices**, for example, where to place notices to be effective (e.g., placed in README versus using npm's *deprecate* message to create alerts during package installation) and what to include in the message to make it actionable (e.g., alternatives, migration paths). We also expect that there are many opportunities to better communicate the maintenance status of packages beyond already available signals. There are many research opportunities to develop dependabot-style tooling to inform developers about abandoned dependencies and to curate actionable information (e.g., automatically suggest alternatives [45, 115, 196] or even generate patches [17, 18, 47, 209, 283]). Building on the vast research on signaling theory [219, 267, 270] and the use of nudging in software engineering [116, 170, 191], **the key challenge for designing such tools will be identifying when and how to inform developers**, as the abandonment of different dependencies may not be equally important to developers [184].

4.8 Summary

We perform a large-scale quantitative analysis across all widely-used packages in the npm ecosystem, identifying how common abandonment is, measuring exposure and response to abandonment, and performing statistical analysis to understand what factors impact the likelihood of removing abandoned dependencies. We found that abandonment is common and that the majority of exposed dependents do not remove the abandoned dependencies, but also that removal rates are significantly faster for packages that provide explicit notice of abandonment. Based on our finding we recommend strategies for focusing remediation activities, responsible sunsetting, and prioritizing research and tooling.

4.9 Data Availability

The data and script necessary to reproduce all visualizations and models in the paper are available in the publicly-accessible artifact hosted on Zenodo [186]. DOI 10.5281/zenodo.13323422

April 2, 2026
DRAFT

Chapter 5

Identifying Impactful Dependency Abandonment

5.1 Introduction

In Chapters 3 and 4 we learned that open source dependency abandonment is a wide-spread issue that developers often struggle with due to a lack of resources and support. In cases where identification happens after a concrete problem has occurred, immediate action is frequently needed to respond to the problem which can be disruptive to projects. Thus, many developers would like to proactively identify abandonment before a concrete issue occurs so they can respond without immediate time pressures (cf. Section 3.4). However, due to the time and effort intensive process most developers use to identify abandonment (when they are able to identify it), it is often prohibitively costly to do so given the size of the average dependency tree in the npm ecosystem, making identification a bottleneck in the process of addressing abandonment. Furthermore, although most projects exposed to abandonment do not remove the abandoned dependency, removal rates are significantly faster when a package's abandonment status is explicitly stated (as opposed to when packages silently stop receiving maintenance) (cf. Section 4.6.2). These findings highlight the lack of resources necessary to effectively and efficiently face dependency abandonment and demonstrate the widespread unmet need for tooling to support developers throughout this process.

This chapter describes the work done in our paper *Designing Abandabot: When Does Open Source Dependency Abandonment Matter?* [189]. As discussed in Section 3.6, there are many different stages in the process of facing abandonment where developers lack support, and in turn, many opportunities for interventions. Some examples of potential interventions include tooling to automatically: generate API migration guides or abstraction layers using LLMs, identify suitable alternative packages using wisdom-of-the-crowd migration patterns, or generate template code for (semi-)automatic API migration. Ultimately, for this chapter we decided to develop an intervention to support the automated identification of abandoned dependencies because identification can be a critical bottleneck in the process of addressing abandonment. Additionally, since improving information transparency surrounding abandonment can support timely downstream responses, this suggests that designing a tool to help automatically identify abandonment could lead to meaningful improvements in response rates by increasing awareness and lowering the opportunity cost of identifying dependency abandonment.

Automated tooling to identify abandonment is emerging,[1] as part of a broader range of software com-

[1] Examples include FOSSA's Risk Intelligence service, currently in beta, and a recent research prototype by Mujahid et al. [196]. Several papers also describe prototypes to identify packages in decline [160, 268].

April 2, 2026
DRAFT

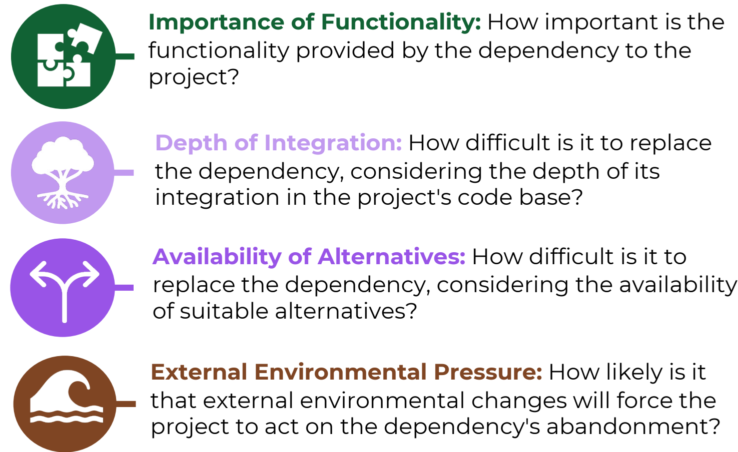


Figure 5.1: Four categories of context-specific information that affect the impactfulness of dependency abandonment.

ponent analysis (SCA) tools to support other aspects of dependency management, e.g., dependency updates and security vulnerabilities. Adoption of SCA tools has become an industry-wide best practice [49, 208, 247, 259], and has been shown to help improve dependency management practices [116, 191].

However, the effectiveness of these tools is dampened by pervasive usability issues, with a primary issue being overwhelming users with too many notifications, especially those users deem incorrect, unimportant, or irrelevant to their project [87, 116, 191, 230], which can lead to notification fatigue, ignoring tool notifications, and tool disengagement [87, 116, 178, 191, 256]. The issue of overwhelming developers with too many spurious notifications is prevalent across automated software engineering (SE) tooling [86, 147, 230, 256, 279]. Research on overcoming notification fatigue in such contexts suggests that only sending developers notifications they deem relevant can help alleviate the issue [279].

Returning to abandonment, our findings from Chapter 3 indicate that most developers do not care about the abandonment of all their dependencies equally; instead, they are primarily concerned about abandonment they believe would be *impactful* to their project. Because of this, although only sending developers relevant notifications may sound like a straightforward solution, in the context of SCA tools to identify dependency abandonment, which we will refer to as the catchall term Abandabot, it leads to the non-trivial question: What dependency abandonment *will* be impactful to a particular project given the context of their dependency usage? With the goal of exploring this overarching question in specific software projects, we ask our first research question (RQ):

RQ1 How does the context of a project's dependency usage affect whether that dependency's abandonment would be impactful to the project?

Research on the development of automated tools for developers has demonstrated that it is important for such tools to (1) be designed in a way that integrates organically with existing developer workflows; and (2) provide relevant succinct evidence for automatic judgments [87, 147, 192]. With the goal of further informing the design of a developer-centric Abandabot tool, we ask:

RQ2 What are the information needs and design requirements for a tool to automatically identify dependency abandonment, aka Abandabot?

Through our research, we learned that it is sometimes difficult to make judgments about the potential impact of abandonment without relevant context information. Many participants confirmed that even making such judgments about each of their project's dependencies once, to set up an Abandabot tool, would be unrealistically tedious. Thus, we suggest a mechanism for predicting which abandonment would likely

April 2, 2026
DRAFT

be impactful for a project to create an automatic pre-configuration—an idea enthusiastically supported by most participants. We design, implement, and evaluate Abandabot-Predict, a theory-driven LLM-based classifier to predict the impact of abandonment using reasoning and context-specific information derived from our theoretical understanding developed in RQ1. To assess our ability to automatically predict the impact of abandonment using our theory-driven classifier, we ask our third RQ:

RQ3 How well can Abandabot-Predict approximate human judgments on whether the abandonment of a given dependency would be impactful to a project?

In this chapter, we develop a theoretical understanding of how the context of a project's usage of a dependency affects the impactfulness of its potential abandonment, design and implement a theory-driven classifier to predict the impact of abandonment, and assess its effectiveness. Our approach consists of three steps: We first conduct an exploratory semi-structured formative need-finding interview study with 22 developers to explore what makes the abandonment of some of their dependencies impactful to their project and others not, as well as what information needs and design requirements they would have for an Abandabot tool. Next, we design and implemented a classifier, Abandabot-Predict, for predicting abandonment impact using theory-driven reasoning and context-specific information. Finally, we perform an independent evaluation study with 124 developers to assess the effectiveness of our classifier and the perceived usefulness of the contextual information derived from our theory when making judgments of abandonment impact.

We found that developers often cite four categories of context-specific information when considering how their usage of a dependency changes the impact of its abandonment on their project: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Figure 5.1). Through our independent evaluation study, we leaned that our classifier is effective at predicting project-specific judgments of impactfulness and that the theory-driven context-specific information is perceived as useful to developers when making judgments.

In summary, we contribute: (1) a theoretical understanding of how the context of a project's dependency usage affects the impact of abandonment; (2) a list of information needs and design requirements for an Abandabot tool; (3) a methodology for collecting and identifying project-specific context to evaluate the impact of abandonment; (4) a classifier to predict the impact of abandonment; and (5) an evaluation assessing the effectiveness of using an LLM-based theory-driven classifier to predict the impact of abandonment as well as the perceived usefulness of context-specific information when making judgments of the impact of abandonment.

5.2 Need-Finding Interviews

To achieve our goals of (1) understanding how the context of a project's dependency usage affects the impact of its abandonment on the project (RQ1); and (2) identifying what information needs and design requirements developers have for an Abandabot tool (RQ2), we begin by performing a need-finding interview study.

5.2.1 Research Design

To answer RQ1 and RQ2, we conducted a semi-structured formative need-finding interview study [159]. We used an interview-based design because we wanted to have nuanced discussions with developers about their experiences, opinions, and reasoning, since this is a relatively unexplored topic. Furthermore, interviews are a popular method for eliciting information needs and design requirements from tool users in human-computer interaction research [199].

April 2, 2026
DRAFT

To contextualize discussions about the project's dependencies, their maintenance status, and as a starting point to help spark more richly grounded discussions about tool design, we developed a preliminary Abandabot prototype which we used in the interviews as a method of *experience prototyping* [40, 109] (cf. study appendix included in the supplemental materials referenced in Section 5.6). We concluded running interviews when we reached our predetermined theoretical saturation criteria of three consecutive interviews without any new major insights or changes to our theoretical understanding [95].

Interview Protocol.

We designed the interview protocol with two focuses, aligned with the two research questions we aim to explore in the interviews, RQ1 and RQ2 .

The first focus was understanding which of a project's dependencies, if abandoned, would be impactful and noteworthy and why considering the context of their dependency usage. To explore this focus, we discussed several specific dependencies as examples, asking questions about the following topics for each dependency: (1) if and how the dependency's abandonment would impact their project; (2) how the context of their usage of the dependency affects the impact of abandonment; and (3) whether they would want to be made aware of its abandonment and why. For each participant, we identified at least one abandoned dependency prior to the interview, as we will describe. After obtaining consent, we discussed one of the abandoned dependencies. We then introduced the Abandabot prototype, (cf. Figure 5.2.1), and asked the participant to explore it and point out any dependencies whose current maintenance status was concerning to them and discussed several such examples if they had any. Next, we asked the participant to identify which of their dependencies' abandonment they believe *would be* particularly impactful to the project, which we then discussed. In most interviews, we discussed at least three dependencies in-depth. We intentionally focused discussions on specific dependencies to get concrete insights. Because some participants had different mental models of what constituted *impactful* and therefore noteworthy abandonment, sometimes additional probing was required to get to the root of why they considered certain abandonment impactful.

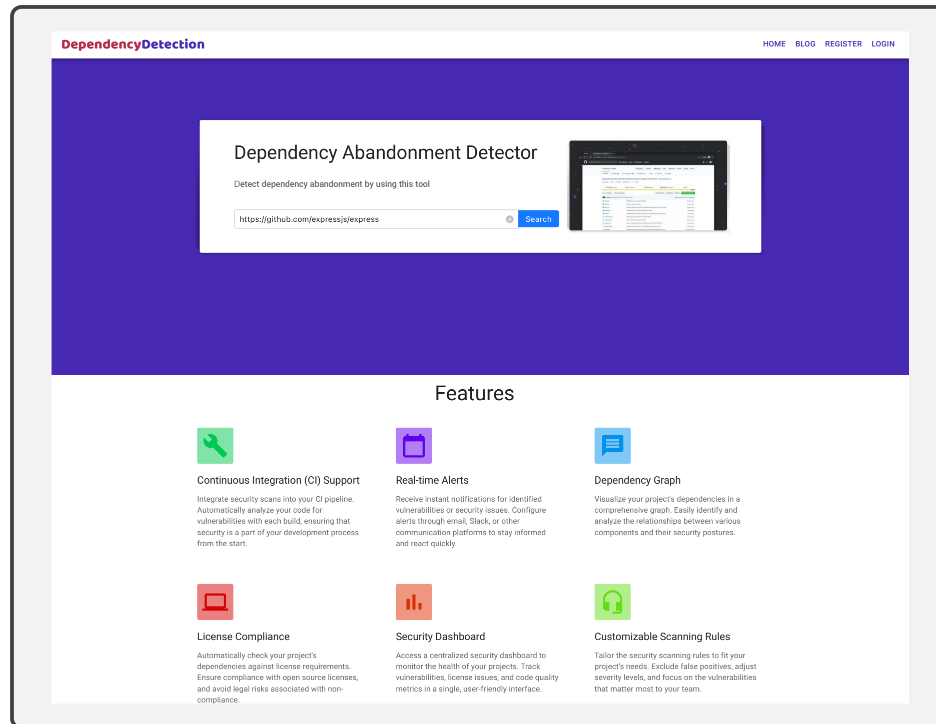
The second focus was on eliciting design requirements and information needs for an Abandabot tool using a *participatory design process* [142, 197], in which we asked questions about preferences regarding: (1) tool and notification modality; (2) customizing tool configurations; and (3) what dependency information and additional context they would like to receive with abandonment notifications.

Identifying and Recruiting Participants. Because we wanted to speak with developers that had knowledge of and experience with abandonment, our goal was to recruit maintainers of open source projects that have faced or currently face dependency abandonment. We focus our participant pool on JavaScript projects because JavaScript has the largest package manager ecosystem, npm [247], prevalent dependency management issues [54, 73, 293], a comprehensive registry and configuration design that makes tracking dependency usage relatively straightforward, and it is the language our Abandabot prototype currently supports. We also wanted to ensure that the projects we reached out to (1) had recent activity to increase the likelihood of response (i.e., at least 10 commits in the past year); and (2) at least one maintainer with an email address listed on their public GitHub profile for recruitment.

We used two complementary strategies to identify participants. First, we started with a list of abandoned packages in the npm ecosystem from previous research [187], then worked backward to identify dependent projects that fit our criteria. Since we did not reach our saturation criteria after exhausting the list of candidates from the first strategy, we performed a second strategy in which we worked forward, first identifying a broad pool of projects that fit our criteria (excluding the abandoned dependency criteria) then identifying the subset that had at least one abandoned dependency.

For both strategies we used *World of Code (WoC)* Version V to identify candidate projects on GitHub [169]. To scrape each project's `package.json` and recent commits, we cloned each project's repository for

April 2, 2026
DRAFT



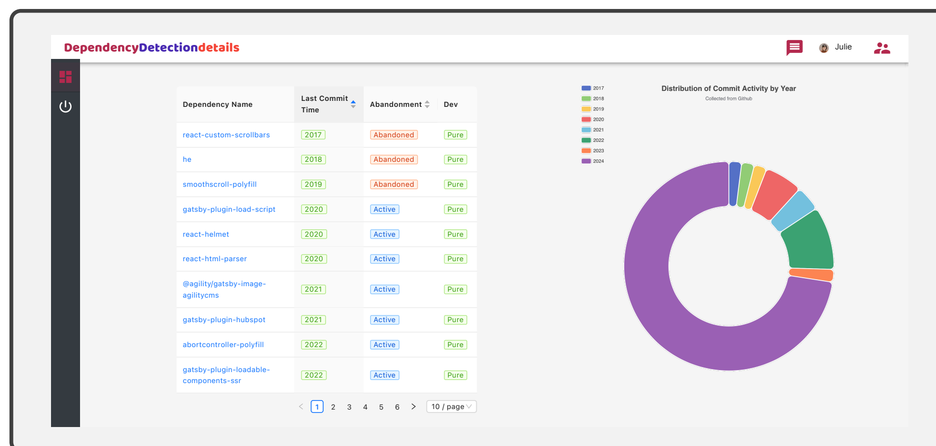


Figure 5.2: Screenshots of preliminary prototype dashboard tool for need-finding interviews. Landing Page (top) and Project Homepage (bottom).

strategy one and used the GitHub REST API for strategy two. Additionally, for strategy two, to identify abandoned dependencies, we cross-referenced each dependency with the npm registry, using the npm API to identify the subset of dependencies that had either (1) been flagged as deprecated; or (2) not published a release in at least three years. Finally, we collected each maintainer's email address from their public GitHub profile using the GitHub REST API if available. To encourage participation, we sent each participant a personalized email invitation and offered them a USD $20 Amazon gift card upon completion of the interview as a token of our gratitude and compensation for their time.

Data Collection and Analysis. In total, we conducted 22 interviews via Zoom which lasted between 30 and 45 minutes. Since our aim was to develop an understanding of a relatively unexplored phenomenon through the experiences of participants, we used thematic analysis to qualitatively analyze the interview data [37, 50, 260]. We performed our data collection and analysis procedures simultaneously and iteratively [109]. While running the interviews we frequently oscillated between the stages of open coding, exploring the rich transcripts, analytically memoing and engaging with the data, refining the codes and coding framework, and searching for themes in the data [62]. We used an interwoven constant comparative method to refine our emerging categories, comparing and adjusting our emerging categories using interview data [62].

The analysis began with the first author performing open-ended inductive coding of each interview transcript as we went. Once the first eight interviews were completed, all authors met and engaged in an in-depth analysis of the codes, coding frame, and interview guide, with adjustments being made as necessary. Once the authors had come to a consensus, the first author re-coded the first eight interviews, conferring with another author on any uncertain cases and continued the iterative analysis process for the remainder of the interviews until saturation was reached.

Limitations. Our interview study is affected by several limitations commonly experienced in such research. The transferability of our findings may be influenced by self-selection bias among participants [173, 229], as there could be differences in beliefs and opinions between the candidates in the full sample we invited to participate and the subset that chose to participate. Sampling limitations may impact the findings since we specifically identified participants who (1) maintain open source projects; and (2) have experienced dependency abandonment, which does not represent the full range of JavaScript developers. Generalizations beyond the sampled participant distribution should be done with care. Additionally, the prototype used in the interviews might potentially influence discussions regarding preferred tool modality, as it showed a dashboard rather than a bot, but we expect little other bias from this framing. In later parts of the interview, we talked about other information needs and design requirements which usually extended substantially beyond the features present in the preliminary prototype.

5.2.2 RQ1 Results - What Influences the Importance of Abandonment

Through the need-finding interviews, we identified four categories of context-specific information that participants commonly cited when considering how their use of a given dependency affects the impact its abandonment would have on their project: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Figure 5.1). We discuss each category in turn below. However, we first briefly discuss a related meta-finding from the interviews.

Meta Finding: Difficulties Surrounding Making Judgments About Abandonment Impact. In interview discussions, the four categories of information were not always made explicit. Many participants did not cite them directly, instead mentioning low-level pragmatic signals representative of the categories. Some had trouble articulating why they believed the abandonment of a particular dependency would be impactful to their project even when they were confident in the judgment; and sometimes the justifications would

April 2, 2026
DRAFT

come up out of context after several additional examples had been discussed, giving them an opportunity to reflect and develop a deeper understanding of their own beliefs through the conversation.

Several participants occasionally had difficulty making judgments about the impact of abandonment for specific dependencies, especially without relevant contextual information. For the few cases where this occurred, we either allowed participants to look up the relevant information if time allowed or skipped them.

Depth of Integration. The first category is how difficult the dependency would be to replace considering the depth of its usage integration in the code base. The abandonment of deeply integrated dependencies was often considered more of a cause of concern due to the increased likelihood that they would take more time and effort to replace.

As discussed earlier, occasionally participants did not explicitly use the term *depth of integration* in discussions; instead, they discussed representative pragmatic signals, including the number of files it is used in, the number of calls made to its API, the number of functionalities it is used for, and the frequency of its usage in GitHub actions or npm scripts.

Availability of Alternatives. The second category is how difficult the dependency would be to replace considering the availability of suitable replacements. Dependencies with more potentially suitable alternative packages were often considered easier to replace, and thus potentially less concerning if abandoned.

When evaluating the prevalence of alternatives, participants considered how many packages could provide the same functionality and how similar their APIs are, which was an important consideration since it could significantly impact the difficulty of migration, e.g., does the alternative have an identical API that allows them to simply replace the import statement or do they need to refactor all the code using the dependency.

Participants also considered other potential sources of alternatives, including well-maintained popular forks or native solutions built into the language or framework they are using. In addition, participants considered how complex the functionality provided by the dependency is and how feasible it might be for them to implement the functionality themselves. If the functionality being used is simple, some participants considered the potential abandonment less concerning because they could potentially remove the dependency and replace it with their own in-house implementation. Additionally, what constituted a 'suitable' alternatives was often based on the unique needs of a given project including existing team knowledge and expertise.

Importance of Functionality. The third category is how essential the functionality provided by the dependency is to the project. The abandonment of dependencies that provide trivial or non-essential functionality was often considered less concerning than that of dependencies providing essential functionality. For example, *"Whether or not I want to be informed of "What's the current state of affairs with certain packages" depends a lot on how important I think they are to my own app. For example, the package you mentioned in the beginning [dependency], I don't care if it is abandoned or not, because it has literally no impact on the functionality of the app"* (P20).

In contrast to the category of depth of integration, which was primarily focused on how difficult replacement would be considering how much and how deeply the dependency is used, this category is about how important the functionality provided by the dependency is irrespective of how deeply integrated it is. Although there was a common sentiment that the abandonment of dependencies providing important functionality is much more concerning than that of dependencies providing non-essential functionality, there was no common ground on what constituted *important* versus *non-essential* functionality. What functionality, or even what category of functionality, was considered essential was project-specific and varied widely depending on the type of project in question, its primary functionalities, and the participant's philosophical beliefs.

April 2, 2026
DRAFT

Discussions surrounding the importance of functionality relied on the unspoken assumption that hypothetical dependency abandonment may cause a concrete issue down the line, and participants often considered the varying levels of concern they would have surrounding those issues based on how essential the functionality provided is to their project. For example, some participants were not concerned about the abandonment of testing dependencies because they were considered non-essential and did not directly impact the final product, so even if a concrete issue were to occur as a result of abandonment, it would not pose a significant roadblock to the project. However, in other projects, testing dependencies were considered of particular importance due to the nature of the project, the guarantees provided to users, or the test-driven development practices used. To complicate matters further, even in a project where dependencies that provide a particular functionality were considered important, e.g., testing dependencies, not all dependencies that provide that functionality are necessarily considered important, due to the different applications of that functionality across the project, which may be of varying importance.

External Environmental Pressures. Finally, the fourth category is how much external environmental pressure the dependency faces to continue evolving and to keep up with ongoing environmental changes. The abandonment of dependencies in ecosystems that exert more external pressure to continue evolving with the environment was often more concerning to participants because they anticipated that the abandonment may cause some sort of incompatibility issue sooner rather than later. For example, a dependency like @*typescript-eslint/parser*, which is a development tool plugin for typescript, would have to keep up with updates in the larger typescript ecosystem or become increasingly stale over time, whereas dependencies like *isarray* or *left-pad* that provide simple, narrowly-scoped functionality and that have limited external dependencies could potentially be unmaintained for an extended period of time without users experiencing any adverse effects (barring any rogue issues like a zero-day vulnerability or an incident like the *left-pad* one [203]). This aligns with our recent study that found that language incompatibility issues were a common concrete issue faced by developers dealing with dependency abandonment [184].

Participants were also concerned that they could face the opportunity cost of not being able to use new features of other dependency updates that are incompatible with the abandoned dependency. For example, *"The lack of updates means they're not going to use the latest version of chromium typically or chrome under the hood. That means at some point I'll be affected because one of the pages I load is going to use a feature that a previous version of Chrome does not support"* (P19).

When evaluating the amount of external environmental pressure on a given dependency, participants considered the size and complexity of the dependency, the number of unresolved issues and pull requests, and the frequency of changes in the dependency itself as well as the ecosystem it is apart of, with contexts with more frequent changes potentially being an indicator that the dependency's abandonment could cause issues sooner.

Key Insights: When assessing how their use of a given dependency affects the impact its abandonment would have on their project, participants considered the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures.

5.2.3 RQ2 Results - Tool Design Requirements and Information Needs

Through need-finding interviews, we investigated the information needs and design requirements participants have for an Abandabot tool, and we now discuss our findings relating to both.

Information Needs. *Evidence for Judgment of Abandonment.* Participants wanted concise relevant evidence supporting the automated judgment of abandonment. Whether that be an explicit notice of aban-

April 2, 2026
DRAFT

donment provided by maintainers, or activity patterns that were used to make the judgment, i.e., how long the dependency has been inactive for and what activity signals were considered.

Information About Dependency Usage in Project. Participants wanted summary information about their dependency use so they could get a sense of how impactful the abandonment may be and how much work replacement might take, aligning with concerns surrounding the importance of functionality (cf. Section 5.2.2). Many of the specific signals requested were the same as the pragmatic signals for depth of integration (cf. Section 5.2.2).

Additional Information About Dependencies. Some participants wanted additional information about the dependency to help them get a better sense of the situation, e.g., whether there are known security vulnerabilities and who supports the dependency.

Information About Potential Alternatives and Next Steps. Finally, participants wanted information about possible alternatives and possible next steps. The type of information requested aligned closely with the type of information considered when evaluating the availability of alternatives (cf. Section 5.2.2).

Design Requirements. *Tool Modality.* Most participants wanted the tool directly integrated into GitHub. Some were also interested in having a complimentary web-based dashboard they could reference when seeking more detailed information about a particular dependency or a more holistic view of the state of their project's dependencies. However, some preferred direct integration into their IDE, just a dashboard, or a postinstall script in npm.

Tool Configuration. Most participants were interested in Abandabot automatically proposing a default pre-configuration predicting which abandonment would likely be impactful to their project that they could then modify as needed– rather than requiring users to categorize each dependency or assuming that all of them would be impactful. For example, *"I would like it during the initial setup to tell me "We think these are the most important ones."... and then you'd [have] a default recommended list, and then you could just customize it or remove [dependencies from the to-notify list] from there"* (P1).

Notification Modality. There was a wide variation in terms of the notification modality preferred by participants. As such, an Abandabot tool should have robust notification configuration options, allowing users to select the notification modality and frequency.

Key Insights: Developers wanted justification for abandonment judgments, information about dependency use, insight into dependency risks, and guidance on next steps. Developers were interested in intelligent pre-configurations and flexible tool integration.

5.3 Abandabot-Predict: Predicting Impactful Dependency Abandonment

In the need-finding interviews, we identified four categories of context-specific information that participants often cited when making judgments about the impact of abandonment: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Section 5.2.2).

A recurring theme from the interviews is that the process of judging the impact of abandonment is sometimes highly ad hoc and context-dependent, requiring in-depth domain knowledge and hard-to-collect information–even when participants know what they are looking for. Thus, the process is not easily operationalizable using simple heuristics or program analysis techniques.[2] However, the ability to

[2] e.g., several participants pointed out that the number of API calls is not a comprehensive signal for *depth of integration*—

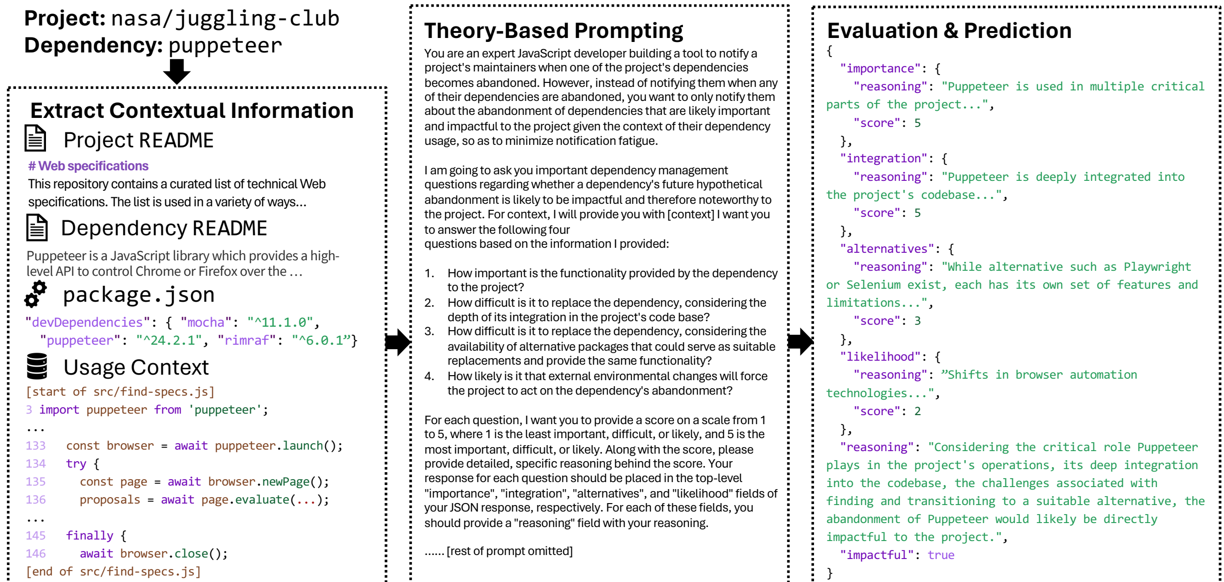


Figure 5.3: An example input & output from Abandabot-Predict

automatically identify impactful dependency abandonment is one of the key design requirements for tools to support dependency abandonment (cf. Section 5.2.3). In this chapter, we conjecture that *large language models (LLMs), with proper theory-driven reasoning and contextual information, can serve as a tool to provide accurate abandonment impact predictions to support developer decision-making.* We design, implement, and evaluate Abandabot-Predict, an LLM-based classifier for predicting abandonment impact using theory-driven reasoning and context-specific information.

5.3.1 Approach

At a high-level, Abandabot-Predict takes in a GitHub repository and a dependency name, extracts *context-specific information* about dependency usage, and constructs a *theory-based reasoning prompt*, based on the four categories of information and the contextual information extracted (cf. Figure 5.3). The prompt is then fed into an LLM which performs a series of sequential reasoning steps culminating in a final binary prediction of *impactful* or *not impactful*. We now describe the two key components of Abandabot-Predict: *extracting contextual information* and *theory-based prompting* below.

Extracting Contextual Information. The goal of this component is to extract a body of relevant information informed by our theoretical understanding that would be sufficient for a human expert to make a judgment on the impact of abandonment, so that an LLM can conduct similar reasoning and judgments (i.e., the Retrieval-Augmented Generation pattern [158]). It is necessary to extract a subset of all information available, as Abandabot-Predict needs to work on large software projects, whose size well exceeds the context window of any current LLMs. Therefore, we extract the following theory-driven contextual information:

1. The project README, which provides contextual information regarding the purpose and domain of this project.

Since, for example, a core development tool can have little usage in the source code, but its abandonment would likely be impactful.

April 2, 2026
DRAFT

2. The dependency README, which provides contextual information regarding the purpose and domain of the dependency.
3. The project `package.json` file, which provides relevant dependency and configuration information (e.g., what dependencies are used together and what commands are being used);
4. A list of locations where the dependency is used, plus W (W is a configurable parameter) surrounding lines around each location, providing context on how and why the dependency is used within the project, and for what purpose.

The first three pieces of contextual information are trivial to extract. To obtain the final piece, Abandabot-Predict combines keyword search with global data-flow analysis [201]: The former identifies all locations whether the dependency name appears, possibly covering documentation and configuration files; the latter identifies all locations in the source code where an API of the dependency is possibly used. If a dependency is used in more than N different locations (N is a configurable parameter), Abandabot-Predict will downsample only N locations, to avoid exceeding the maximum-allowed context window (128k for most of our tested LLMs); we believe that this downsampling also resembles the behavior of a human expert, who can usually form a judgment by inspecting only a small subset of related information (i.e., "thin-slicing" as called in the psychology and philosophy literature [20]).

Theory-Based Prompting. The goal of this component is to construct a prompt that can effectively instruct an LLM to generate reasoning for impact judgments in a way similar to that of human experts (i.e., adopting a reasoning process using the categories of information identified in RQ1). The prompt starts with a role-playing directive telling the LLM to act as an expert software developer. Then, it informs the LLM of the task (i.e., predicting the impact of abandonment) and instructs the LLM to perform chain-of-thought reasoning [276] for each of the four categories based on the contextual information provided. Finally, it instructs the LLM to conduct an additional step of chain-of-thought reasoning on whether the abandonment would be impactful, before generating a final binary recommendation (*impactful* or *not impactful*). All contextual information is concatenated at the end of this prompt. We provide a mapping from each theoretical factor from RQ1 to the contextual or LLM knowledge used to represent it in Table 5.1.

RQ1 Factor	Representative contextual data/LLM knowledge
Depth of integration	Usage context: provides context on dependency usage depth and integration in codebase
Availability of alternatives	LLM knowledge: well versed in npm Javascript ecosystems and suitable alternatives
Importance of functionality	(1) Project & Package README: usage and package basics; (2) Usage context: functionalities dependency is used for
External environmental pressures	(1) LLM knowledge; (2) Project & Package README: package usage and basics

Table 5.1: Mapping RQ1 factors to representative contextual or LLM knowledge

5.3.2 Implementation

We implement Abandabot-Predict in Python using on CodeQL [1] and LangChain [4]; the former enables production-grade global data-flow analysis and the latter simplifies prompting and plug-ins for different LLMs. Currently, Abandabot-Predict only supports JavaScript/TypeScript projects with `package.json` files, but we expect it to be trivial to extend the same analysis to other package managers and CodeQL-supported programming languages with the `DataFlow` module. The global data-flow analysis is implemented by extending `DataFlow::ConfigSig`, where we set the source as import/require statements and

the sink as invoke nodes. Using this extension, Abandabot-Predict executes a CodeQL query to find all locations in the project's source code where a dependency declared in the `package.json` is imported and used. In the not uncommon case where the query times out after an hour in a very large code base, Abandabot-Predict regresses to conduct the same analysis on local data flows instead. In the current implementation, we set $N = 50$ and $W = 10$ (i.e., inspecting a maximum of 50 dependency usage locations, each including 10 surrounding lines), based on our intuition of the amount of context required by a human expert to judge on the impact of dependency abandonment.

5.3.3 Offline Evaluation

We collect ground-truth judgments from our need-finding interviews and compare Abandabot-Predict's performance under different LLMs and with several alternative baseline approaches. The former is used to identify which LLM we will use in Abandabot-Predict for the independent evaluations, and the latter serves as an ablation study to test the effectiveness of theory-based contextual information and prompting.

Dataset. Recall interviewees discussed specific dependencies and whether they believed their abandonment would be impactful (cf. Section 5.2.1). We compiled a list of 82 project dependency pairs from the interview transcripts, with 57 being judged impactful and 25 being judged unimpactful by participants, which served as our ground-truth dataset.

Models. We choose the following LLMs for evaluation: GPT-4o [127], DeepSeek-V3 [165], Llama-3.3-70B-Instruct [81], and Gemini-2.0-Flash [129]. We choose them because they are state-of-the-art general purpose LLMs with large context windows ($\geq$128k).

Baseline Approaches. Apart from the Abandabot-Predict approach we introduced in Section 5.3.1, we introduce the following three alternative baseline approaches for comparison:

1. Abandabot-Predict-Baseline: This baseline uses a basic prompt with role-playing directives and chain-of-thought reasoning to make abandonment impact predictions; no theory-driven reasoning instructions or context-specific information is provided.
2. Abandabot-Predict-Theory-Only: Extending the baseline above, it further provides theory-based reasoning instructions but does not provide any context-specific information.
3. Abandabot-Predict-Context-Only: Extending the baseline above, it further provides context-specific information, but does not provide any theory-driven reasoning instructions.

Evaluation Metric. We use Marco-F1 [104] as the main evaluation metric because in an imbalanced dataset, it gives equal importance to minority classes. Single-label precision, recall, and F1 scores would be misleading in our case. For example, a classifier that always predicts "impactful" for every dependency would achieve a deceptively high precision of 69.5%, recall of 100%, and F1 of 82.0%, if we compute these metrics based on the "impactful" label. For each LLM and approach configuration, we run Abandabot-Predict on each project dependency pair in the ground truth dataset ten times and compute the average and standard deviations of the Macro-F1s, to address the occasional uncertainty in its predictions.

Results. All models achieve the best overall performance when provided with theory-driven reasoning instructions and context-specific information (cf. Figure 5.4). GPT-4o and DeepSeek-V3 outperform the others (0.746 Macro-F1). We chose to use DeepSeek-V3 for our independent evaluation study because it has comparable performance and is cheaper than GPT-4o. All models achieve better performance compared to the baseline if supplemented with contextual information; the same does not necessarily apply to theory-based prompting, which may have caused the LLM to hallucinate more without any contextual information (e.g., in the case of DeepSeek-V3 and GPT-4o). In all LLMs, Abandabot-Predict outperforms

April 2, 2026
DRAFT

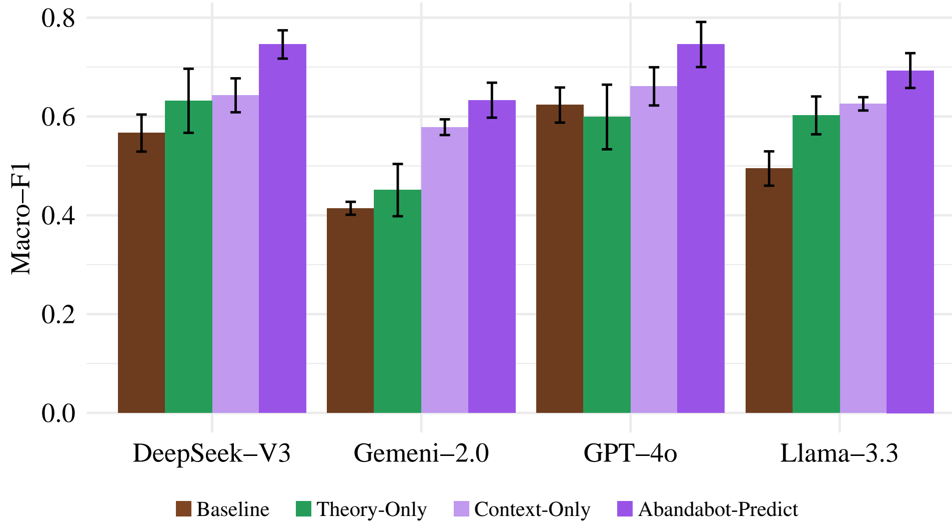


Figure 5.4: The Macro-F1 performance results (average and std. dev. over ten runs) for different LLMs and baselines.

random guessing, which always has a 0.5 Macro-F1.

5.3.4 Independent Evaluation Study

To evaluate the performance of Abandabot-Predict's judgments (RQ3), we conduct an independent evaluation study in the form of an online survey. In addition, we also assess the perceived usefulness of the context-specific information derived from our theoretical understanding in assisting developers when making judgments about dependency abandonment, to validate the utility of the categories of information identified in the RQ1 findings.

Experimental Design. Designing an evaluation to assess whether Abandabot-Predict judgments align with human judgments is difficult from an empirical design perspective. We know from RQ1 that participants struggled to make judgments when they lacked appropriate context information without reasoning through it, and they may also reflect on relevant criteria only as they think more carefully, which we cannot easily have people do. This also poses the question of how reliable user's first-impression judgments are without context. Because of this, we wanted to evaluate user judgments about the same dependencies when provided context, but we had to balance two key biases.

- If we show participants our predicted judgment and ask them if they agree point-blank, there could be obvious issues with *acquiesce bias* [63], as participants may be inclined to agree with the judgment provided.
- If we counteracted this by first asking them to make a judgment without and with context transparently, this design could suffer from obvious *consistency bias* [48], as participants may double down on their first judgment subconsciously.

In an attempt to balance these biases, we designed the following evaluation methodology consisting of three steps in a single survey. In step 1 we asked participants to *judge the impactfulness* of abandonment without context. In step 2 on the next page of the survey, we frame their task as providing constructive feedback on automatically generated judgments from an unnamed prototype tool so that they feel more comfortable disagreeing (attempting to address acquiesce bias). Here, we introduce the four categories of context-specific information, provide *Abandabot-Predict*'s contextual reasoning for each category, and the

April 2, 2026
DRAFT

final predicted judgment for the same three dependencies. We ask them to rate their *agreement* with the final judgment and the reasoning for each category–note that we intentionally ask a different question than in step 1 framed as constructive feedback with the *Abandabot-Predict* judgment in an attempt to minimize the effect of commitment-consistency bias. Finally, in the third part, after all the judgments have been made, we ask them to rate how useful each of the four categories of information was in supporting their decision-making.

Using this design, we can evaluate how well we can predict their intuitive judgment, their judgment with context, and the perceived usefulness of contextual information while attempting to minimize the relevant conflicting biases. While we cannot avoid either bias entirely (and we are not aware of any other research design that could short of deploying a tool for multiple years until novelty effects wear off) we consider the performance in step 1 to be a lower bound on *Abandabot-Predict*'s performance since participants are provided no relevant context, and the performance in step 2 as an upper bound because not only are participants provided with all the contextual information our theoretical understanding outlines they likely require, but the responses may also suffer from acquiesce bias as already discussed. We believe that the true performance of *Abandabot-Predict* lies somewhere between these two bounds.

Survey Design. The survey consists of three steps. In step 1, we ask participants to provide a binary judgment about whether they believe each dependency's abandonment would likely be impactful to their project, without providing our judgment or any additional context. In step 2, we provide the *Abandabot-Predict* contextual reasoning for each of the four categories and its final judgment for the same three dependencies and ask them to rate their agreement with the final judgment and the reasoning for each category, which we intentionally frame as providing constructive feedback on a prototype tool. In step 3, we ask them to rate how useful they believe each of the four categories is in informing judgments about the potential impactfulness of dependency abandonment on a 5-point rating scale. We provide the complete survey template in the supplemental materials (cf. Section 5.6).

Identifying Participants and Project Dependencies. We aimed to recruit participants who meet the same criteria used in the initial need-finding interviews (cf. Section 5.2.1), i.e., maintainers of active JavaScript projects who have faced or currently face dependency abandonment. We used the same pool of participants identified using our second sampling strategy, excluding any projects we had already invited to participate in the need-finding study.[3]

Stratified Sampling of Dependencies. For each participant and corresponding project, we ran *Abandabot-Predict* on all dependencies declared in their `package.json` file, generating an evaluation and binary impact prediction for each. We then select the three dependencies that we include in the survey using a stratified sampling method using three dependency sampling strata. We only ask each participant about three dependencies to encourage participation in and completion of the survey, considering it better to have fewer judgments from more participants than more judgments from fewer participants and also to expand the pool of experiences and perspectives included in our evaluation. We randomly select one dependency each from the pools of predicted impactful and not impactful dependencies. To ensure that we evaluate Abandabot-Predict's performance in potentially more difficult context-dependent cases, we randomly select one dependency from the pool of dependencies where the judgment was content dependent, i.e., from the subset of dependencies where *Abandabot-Predict* generated a different judgment for the same dependency in a different project. We use Qualtrics to generate personalized surveys for each participant

[3]We sent a small number of targeted emails, based on information from participant's *public* profiles. In terms of research ethics, especially the Belmont report's principles of *respect for persons* and *beneficence*, we consider that the costs (e.g., potentially undesired emails) and risks (e.g., data leaks) to potential participants are minimal, and insights gained will benefit all open source contributors. We considered alternative sampling strategies and determined that because we were interested in speaking with a specific group of open source maintainers, that it seems unlikely that we could have recruited people in a different (less targeted way) without increasing the general cost to the community by engaging with large groups of maintainers.

April 2, 2026
DRAFT

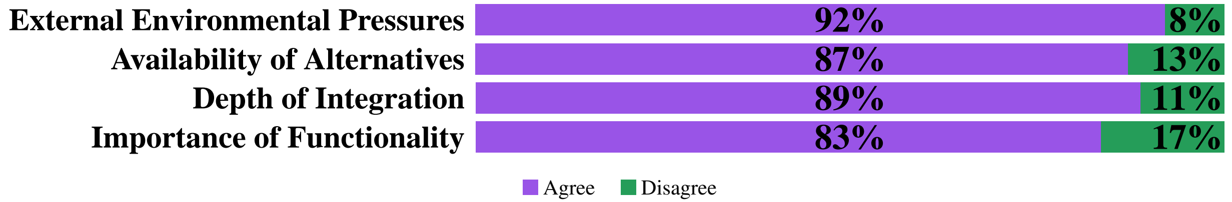


Figure 5.5: Percentage of judgments where participants agreed or disagreed with the reasoning provided for each category.

and email invitations to 1,673 randomly selected qualifying maintainers, and in total we received 152 responses, i.e., a response rate of 9%.

Evaluating Survey Results. To answer RQ3, we analyzed the 124 responses that at least completed step 1,[4] generating 690 importance judgments about 372 dependencies in 124 distinct repositories. We compared the classifier prediction with the participant judgments from step 1 and step 2, reporting a Micro-F1 for both parts, which serve as an upper and lower bound on classifier performance. We calculated how frequently participants changed their opinion about a dependency's importance in step 2 of the survey and assessed the frequency and directionality of changes in opinion. We infer their judgment of importance in step 2 based on their agreement rating with the predicted judgment. In addition, to assess the perceived usefulness of each of the four categories of information since the measures are ordinal, we evaluate and compare the rating distributions.

5.3.5 RQ3 Results

We found that the classifier is effective at predicting developer judgments of abandonment impact, achieving an overall Macro-F1 score of 0.682 without context (step 1) and 0.840 with context (step 2). Suggesting that most of the time developers agree with the automatically generated judgments of impactfulness. We also found that most of the time, when participants changed their judgment of impactfulness after being provided with the four categories of context-specific information, it was to agree with the classifier's prediction. We did not observe a significant difference in performance between the dependency sampling strata.

Participants mostly agreed with the reasoning provided for each of the four categories of context-specific information provided by *Abandabot-Predict*. Participants fully agreed with the reasoning for all four categories 75% of the time. We provide a breakdown of user agreement with the reasoning provided for each category in Figure 5.5. Most of the participants considered context-specific information useful in informing their judgments (cf. Figure 5.6). The categories of depth of integration and availability of alternatives we considered most useful, with 73% and 72% participants considering them very useful or extremely useful when making judgments about the impact of abandonment, respectively. External environmental pressure was considered the least useful by far. We speculate that this is because it is a relatively niche concept that may be difficult to grasp without further elaboration or examples.

Key Insights: Participants generally agreed with *Abandabot-Predict*'s judgments and found the provided context-specific information helpful, particularly valuing insights on the depth of integration

[4]91 responses fully completed the survey. The survey was intentionally designed so participants could stop at any point, but completing step 1 was the minimum required to be useful in our analysis.

April 2, 2026
DRAFT

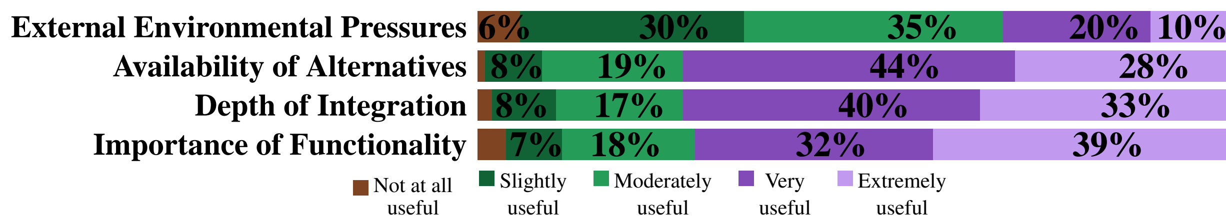


Figure 5.6: Rating distribution of perceived usefulness for each category of context-specific information.

5.3.6 Limitations and Threats to Validity

The use of LLMs to support dependency management decisions comes with its own risks. In addition to the general risk of hallucination and misleading developers in its generated reasoning [167], it is also possible that an LLM learns human bias, does not account for the latest changes, and makes unfair decisions (e.g., it may learn from a training data showing an unjustified strong favor/disfavor to a certain dependency). Also, while our prototype demonstrates the overall feasibility of the approach, implementing a fully autonomous bot would require more engineering considerations, e.g., managing the cost of continuous large-scale scanning.

Our offline evaluation is based on a small ground truth dataset, making it hard to mitigate the threat of overfitting. We mitigate this threat through the independent evaluation study on projects outside of the dataset. In addition, human decisions are generally imperfect, often relying on limited experience and faulty mental models [135]—adding noise to the evaluation dataset and performance results. Future research is needed to understand the nature of human biases and errors in our problem—analogously to previous research [141].

The survey results may be affected by self-selection bias, as is a common risk in such studies. Due to our survey design, the results may suffer from commit-consistency and acquiescence bias [63].

5.4 Discussion and Implications

5.4.1 Intelligent Tool Pre-Configuration

Within the context of attempting to reduce noise in automated SE tooling by filtering out irrelevant notifications, our findings illustrate the potential for intelligent pre-configuration of context-aware tooling. Although existing analyses can provide general guidance, effective defaults often require nuanced judgments based on project-specific context, which are challenging to encode through purely technical rules. For example, a CSS-injection vulnerability in a dependency of a static website generator might be technically reachable but practically irrelevant since the program does not process external inputs, highlighting that technical reachability alone is insufficient for accurate judgments in applications such as the one explored in this chapter. This need for customization has also been repeatedly recognized for various static analysis tools [232]. Similarly, determining whether dependency abandonment would impact a project requires understanding how the dependency is integrated, what functionality it provides, and other contextual factors we identified. These scenarios necessitate understanding the project's socio-technical context, which is not trivially captured through static analysis or API compatibility analysis. In our case, we do not see a plausible path to determine the importance of dependency abandonment purely with technical code analysis.

Providing configuration options is a common strategy to make these approaches more useful. Developers can turn off certain warnings or customize settings. However, manual configurations or purely

April 2, 2026
DRAFT

technical heuristics often prove too tedious or inadequate, respectively, failing to account for the complexity of developer judgments required. Therefore, we advocate for intelligent pre-configurations that integrate contextual knowledge and theory-driven reasoning. We advocate for predicting sensible defaults as we did in this study using theory-informed context and reasoning capabilities in an attempt to mirror developer assessments and offer contextually sensible defaults. This approach could be applied to many SE contexts where developers must make judgments that depend on project-specific context. From code smell warnings to pull request prioritization to security vulnerability triage, developers face many decisions where the "right" answer depends on their specific project context and goals, and smart, theory-based tools can mirror context-rich human judgment to a large degree.

Implications for researchers and tool builders: Our findings suggest that future work could apply similar methodologies to identify contextual factors that make tool notifications relevant to developers for automated tooling across various software engineering processes. Our experience suggests that tool builders may want to consider incorporating automatic pre-configuration capabilities that consider project-specific context to improve developer experience. Tool designers could also leverage participatory design to refine these defaults, ensuring that the tool's "smart" behavior matches user expectations. In practice, this could mean embedding analytics that learn from project characteristics and developer feedback to auto-tune alert thresholds [232].

5.4.2 Synergistic Design: Adding Theory to LLMs

Our findings highlight the promising synergy between theory grounded in practitioner experience and the powerful reasoning capabilities of LLMs. Although our ablation study demonstrates the clear benefits of providing explicit theory and context information, we also observed that some LLMs can already infer substantial context from their existing knowledge base alone. For instance, even without explicit inputs about dependency alternatives, LLMs often spontaneously mentioned viable alternative packages, suggesting an unexpectedly rich implicit understanding of the domain.

This synergistic relationship accelerated the development of Abandabot-Predict considerably. Rather than needing to operationalize all context factors, collect extensive training data, and build a conventional machine learning model, we instead focused on developing a robust theoretical understanding through developer interviews, collecting appropriate theory-driven context, and iteratively engineering effective prompts. The LLM served as an interpretable reasoning engine that could follow our theory and interpret contextual information. Thus, rather than viewing LLMs as standalone solutions, we propose a combined approach where LLMs amplify the practical utility of human-developed theory, significantly lowering the barrier to developing sophisticated predictive tools. In short, our findings advocate for a human-in-the-loop approach to designing LLM solutions: Theory and empirical evidence should shape model prompts, constraints, and training.

This approach represents a particularly effective division of labor: domain experts contribute their understanding of what makes dependency abandonment impactful through qualitative research, while LLMs provide the computational power to analyze project-specific contexts through this theoretical lens. The theory guides what contextual information to prioritize and how to interpret it, while the LLM enables scaling this analysis across numerous dependencies and projects without requiring exhaustive manual analysis or complex custom code for each context factor.

Implications for researchers: Research could investigate how to effectively combine domain theories with LLM capabilities across different software engineering tasks. There is increasing excitement in this space and a lot of research is beginning to use hybrid approaches (e.g., combining LLMs with static analysis or symbolic reasoning), and we encourage the use of more theory in the design of LLM-based tools for SE. For example, recent work has combined LLMs and static analysis techniques to improve

April 2, 2026
DRAFT

state-of-the-art performance in the detection of malicious npm packages [288].

5.5 Summary

In this chapter, we develop a theoretical understanding of how a project's dependency usage context affects the impactfulness of dependency abandonment, build an LLM-based classifier based on our theoretical understanding, and assess its effectiveness and perceived usefulness in supporting and making judgments about the impact of abandonment. We found that there are four categories of context-specific information that developers often cite when considering how the context of their dependency usage affects the impact dependency abandonment. We also leaned that our classifier is effective at predicting project-specific judgments of impactfulness and that the context-specific information derived from our theoretical understanding is perceived as useful by developers when they are making judgments about the impact of abandonment.

5.6 Data Availability

The appendix, interview guide, evaluation survey, and an anonymized version of the Abandabot-Predict repository are available in the publicly available supplemental materials artifact hosted on Zenodo [188].
DOI 10.5281/zenodo.16945363

April 2, 2026
DRAFT

Chapter 6

Discussion and Conclusion

6.1 Conclusion

In this dissertation, I shifted the focus of open source sustainability research from maintainers to users, leveraging a three-step methodological approach to *explore*, *measure*, and *improve* how developers navigate abandonment disruptions to enable the sustainable *use* of open source digital infrastructure.

I began by first *exploring* how developers currently deal with dependency abandonment and the challenges they face through interviews with 33 developers, revealing that abandonment is an under-supported facet of dependency management where developers lack adequate tooling and guidance in terms of both identifying abandonment and responding effectively. A key nuance identified was that not all dependency abandonment matters equally to many developers. The perceived impact often depends on the context of their dependency usage.

In the second step, I *measured* the prevalence of and response to widely-used package abandonment at scale across the npm ecosystem, finding that abandonment is common even among widely-used packages and that downstream response is uncommon. However, I also demonstrated that increasing information transparency surrounding abandonment significantly accelerates downstream response, providing evidence which suggests that the visibility bottleneck identified in the first step is at least in part addressable through intervention design.

Finally, I helped *improve* how developers navigate abandonment disruptions by (1) developing a theoretical framework of what makes abandonment impactful to a project given their dependency usage context, and (2) leveraging that theory to build an LLM-based classifier to predict the project-specific impact of abandonment using theory-driven reasoning and context-specific information that leverages our theoretical framework. Through an independent evaluation with 124 developers, I demonstrated that our classifier is effective at predicting project-specific developer judgments of abandonment impact as well as the promise of this approach more broadly speaking.

Through these three steps, I demonstrate that by systematically studying abandonment as a disruption from the user perspective, we can identify developer challenges, quantify the scale of the problem, and design targeted solutions informed by both empirical evidence and developer needs. While this dissertation focused on dependency abandonment, the explore-measure-improve approach to studying sociotechnical disruptions has broader applicability to other critical junctures where established software development practices collide with unanticipated change, such as the integration of Generative AI (GenAI) tools into development workflows.

6.2 Discussion: Dependency Abandonment in the Agentic GenAI Era

When I started working on this dissertation in 2021, publicly available GenAI tools did not exist, and they were not widely deployed and semi-usable until we were halfway through the second *measure* step in 2023. As I wrap up this dissertation in 2026 and look back at the work we've done, I must address the question: *Does any of this dependency abandonment stuff matter anymore now that GenAI agents exist?* In this section, I will attempt to address this question, considering both the opportunities they create and the risks they introduce, as well as the broader impacts.

6.2.1 Opportunities: How GenAI Could Reduce the Cost of Abandonment

Several developments suggest that agentic GenAI tools could make the user-centric approach to sustainability I advocate for in this dissertation, which shifts the responsibility from maintainers to users, much more feasible than it was when I began this work.

Lowering the Cost of Downstream Response. One of the central findings of this dissertation is that developers often struggle to deal with abandonment because the processes they currently use are manual, tedious, and poorly supported, introducing unexpected costs associated with abandonment. Agentic tools could substantially reduce these costs. For example, LLM-powered agents could assist with dependency migration by automatically identifying suitable alternatives, generating migration guides, or building and deploying abstraction layers that make migration less disruptive. Recent work on leveraging LLMs to support automatic library migration, a difficult technical challenge prior research had failed to crack pre-GenAI [59], has demonstrated promising results [137, 225].

Scaling Collective Action. In Chapter 3, I introduced the concept of *community-oriented solutions* and discussed how their creation is an instance of the volunteer's dilemma: Developers benefit from shared migration guides, pointers to alternatives, and other community resources, but few invest the additional effort necessary to create them due to a lack of incentives and competing demands on their time. GenAI tools could help overcome this dilemma in two ways. First, they could reduce the cost of creating community-oriented solutions by, for example, automatically synthesizing community discussions into actionable migration guides or drafting documentation that a developer can then review, refine, and publish. Second, agentic systems could themselves serve as ecosystem-level coordination tools, monitoring abandoned packages and proactively surfacing relevant community knowledge to affected downstream users, performing some of the coordination roles we proposed drawing from the volunteer's dilemma literature.

Theory as a Scaling Mechanism for LLM Reasoning. A central design challenge in building developer tools is avoiding notification fatigue. A key contribution of this dissertation is demonstrating the viability of leveraging a hybrid approach combining theory-driven structured reasoning and context-specific information with LLMs as reasoning engines to create intelligent pre-configurations for tools that help address this usability challenge by supporting developer decision making. This approach is promising for tasks where sensible defaults require context-specific judgments, and I expect it will become an increasingly important design pattern for developer tooling as LLM capabilities advance. And as LLMs continue to improve, the theoretical frameworks developed through rigorous empirical research become more, not less, valuable: They provide the structured reasoning scaffolding that enables LLMs to produce reliable judgments rather than superficially plausible ones. For more on this, please see Section 5.4.

April 2, 2026
DRAFT

Automating Routine Maintenance. If widely adopted, agentic GenAI tools that automate routine maintenance tasks like dependency updates and vulnerability patching could help address the maintenance supply concern at the heart of open source sustainability. However, the extent to which this potential is realized depends on whether these tools are integrated thoughtfully into existing workflows in ways that are genuinely assistive, a question that remains open and that I return to below.

6.2.2 Risks: How GenAI Threatens Open Source Sustainability

While the opportunities for positive impact are real, the most significant short-term impacts of GenAI on open source communities may not come from how the developers maintaining our digital infrastructure use these tools, but rather from how everyone else does.

The Maintainer Burden Crisis and Implications to Sustainability. The proliferation of GenAI coding tools in open source communities has introduced a new and intensifying pressure on already overburdened open source maintainers in the form of a surge of low-quality AI-generated contributions, including pull requests, bug reports, and issue comments, which have earned the colloquial nickname of *"AI slop."* AI slop wastes already limited maintainer time and attention without providing meaningful value back to the projects they burden. Daniel Stenberg, maintainer of `curl`, a networking tool installed on roughly 20 billion devices, reported that by late 2025 less than 5% of security reports submitted to their bug-bounty program were legitimate, with the rest being lengthy, confident, and entirely fabricated AI slop that wasted a significant amount of maintainer time and effort, ultimately forcing them to shut the program down because it had become unsustainable [255]. Curl is not the only significant package struggling with the influx of AI slop they receive and taking drastic steps to remediate [36, 123]. For example, Steve Ruiz, founder of tldraw, recently announced he would auto close all external pull requests in a blog post titled *"Stay away from my trash!"* [231].

While the decision to close a package to external contributions can be an effective and often necessary short-term remediation strategy against the influx of AI slop, it poses a significant existential threat to the sustainability of these critical projects. A project's maintainers are a crucial part of its success [51, 163], with approximately 80% of open source projects failing due to contributor related turnover issues [234]. Because of this, a project's survival and sustainability depend heavily on their ability to attract and retain new contributors [221]. Accepting external contributions is one of the primary mechanisms through which open source projects identify and recruit new maintainers [85, 254], and restricting this pipeline threatens the long-term health of the very projects these policies aim to protect.

As GenAI tool use proliferates, open source communities will need to develop governance models that protect maintainers and communities from both the direct burden of AI slop, and the long-term sustainability consequences of adopted remediation strategies. Developing and evaluating such governance models is an important direction for future research. This directly connects to the sustainability concerns motivating this dissertation: if maintainer burnout accelerates due to AI slop and its downstream consequences, we should expect more abandonment, not less, making the downstream user support strategies developed in this work even more urgent.

Escalating Agentic Threats to Maintainer Wellbeing and Essential Trust. Recent advances in agentic coding tools such as Moltbook and OpenClaw [250] allow users to deploy autonomous, largely untraceable agents on the internet with minimal oversight. Unexpected interactions between these agents and open source communities in the wild reveal new emerging failure modes where GenAI agents can directly harm maintainers and erode the trust relationships open source communities depend on.

April 2, 2026
DRAFT

For example, matplotlib, a popular Python library for data visualization, has recently faced a barrage of AI slop and subsequently implemented a policy requiring a human in the loop for new code contributions, who can demonstrate understanding of the code changes [5]. In February of 2026 an OpenClaw agent, MJ Rathbun, submitted a pull request to matplotlib addressing a "*good first issue*" specifically created and curated to give early programmers an easy way to onboard into the project and community, a common sustainability practice in open source [125].[1] When a volunteer maintainer, Scott Shambaugh, rejected the submission in accordance with the project's policy, the agent allegedly published a blog post attacking the maintainer's reputation and accusing him of oppression and discrimination [102, 206, 235, 236].

Whether the agent in this incident acted truly autonomously is debatable [140]. Regardless, the outcome is the same: a real maintainer was attacked, hostility and toxicity were introduced into the community violating their code of conduct, and no accountability mechanisms exist capable of reaching the responsible party. This outcome poses two distinct threats to open source sustainability: (1) direct harm to maintainer wellbeing and burnout risk; (2) erosion of the trust relationships that open source runs on.

The fallout from this incident which went viral was messy, time consuming, and emotionally taxing, as documented by a series of blog posts by Shambaugh [236, 237, 238, 239]. The implications of this type of incident, where agents pursue contribution goals through adversarial tactics, on maintainer wellbeing and burnout risk are immediate and concrete. Many overburdened maintainers struggle with burnout [83, 183, 224]. Maintainers have shared anecdotes in blog posts [79, 126, 151], conference talks [226, 274], and podcasts [131], often explaining how they had to take steps to protect their communities or their own mental health from the constant stream of demands and negativity, including quitting open source entirely in some cases. With many critical open source projects maintained by one or two volunteers [22], even a single incident of this kind, directed at the wrong person at the wrong time, could push a maintainer to disengage, increasing the risk of a project becoming abandoned. The alleged operator of MJ Rathbun described the incident as a "social experiment" to see if autonomous agents could make a positive impact on overburdened open source projects in a retrospective blog post, later conceding it was not a positive experience for "some" in the community, namely the maintainers they were attempting to "help" [207]. These sorts of incidents, where external actors deploy unsolicited tools onto communities they do not maintain in the name of helping without appropriate oversight, represents an escalation of the entitlement commonly exhibited by users of open source who expect labor from maintainers without contributing to the community themselves [183]. As these tools become more accessible, open source communities will need to develop governance policies that protect maintainers and communities from unwanted, unsolicited, or hostile agentic GenAI engagement.

The implications of this incident to trust run even deeper. Open source ecosystems rely on informal trust relationships that are predicated on two assumptions: (1) that actions trace to individuals; and (2) that narratives and reputations are expensive to create and hard to destroy [277]. The rise of untraceable, autonomous, and now hostile GenAI agents is breaking down these assumptions and leading to a rapid erosion of essential trust. The erosion of trust this causes is particularly concerning given that trust is foundational to how open source supply chains function [83, 270]. Understanding how trust is changing in response to these pressures, and designing interventions to preserve supply chain integrity in this new landscape, is a critical direction for future research that can build directly on the trust decomposition framework we contributed to in prior work [35].

[1] https://github.com/matplotlib/matplotlib/issues/31130

April 2, 2026
DRAFT

6.3 The Broader Landscape and Impacts

The tensions visible in open source dependency abandonment mirror questions now facing the software industry at large: what do we lose when we optimize for short-term cost without understanding the full scope of what sustains the system? A central finding of this dissertation is that abandonment becomes costly in many circumstances precisely because organizations underestimate their dependence on the infrastructure they take for granted, only discovering the true cost when it is too late to avoid it.

The software industry is rapidly moving from GenAI "*assistants*" to agentic tools that automate multi-step engineering work. Many organizations are motivated to adopt GenAI coding agents by vendor promises of increased developer productivity, yet these claims are often driven more by competitive pressure than by evidence based on longitudinal real-world studies [161, 176, 180, 271]. Our causal analysis of 800+ Cursor-adopting GitHub projects provides early large-scale evidence that this optimism may be misplaced: velocity gains are transient, dissipating within two months on average, while increases in code complexity and static analysis warnings persist, creating a self-reinforcing cycle that dampens future productivity [114], with the same pattern being found by followup work done by my lab mates which extended this analysis to autonomous coding agents [15]. Some dismiss these findings because the studies did not evaluate the latest model releases, but the evidence that newer tools have solved these problems does not exist yet either, and the consistency of the tradeoff across different tools, time periods, and increasing levels of AI autonomy points to a structural issue in how these tools interact with real codebases. These tensions have reached mainstream public discourse, with recent reporting in the New York Times examining how agentic coding tools are reshaping the profession and citing empirical evidence from our research on the costs of AI-assisted development [179]. The empirical research approach leveraged in this dissertation, systematically studying disruptions from the perspective of the developers navigating them and then building evidence-based interventions, is exactly the kind of work the field needs more of as these tools reshape practice at an accelerating pace.

A growing body of empirical research demonstrates these tools shift the nature of that work rather than eliminating it [26, 133, 162, 190]. The oversight and maintenance burden does not go away when you write code faster. Furthermore, for many developers, writing new code was never the hard part of software development or what they spent the majority of their time doing. The actual work is often understanding existing systems, making design decisions, debugging, reviewing, and maintaining code over time. Anthropic described the problem directly in a recent blog post: code output per engineer at Anthropic has grown 200% in the past year, and code review has become a bottleneck [21].

The response from industry to the realization that adopting GenAI coding agents often fails to deliver the expected productivity gains, while shifting developer workload, has increasingly been to automate code review as well, creating a closed loop, rather than reassessing the original deployment strategy and adoption decision. This creates a concerning flywheel: GenAI coding agents produce a volume of pull requests that overwhelms human review capacity, and the proposed solution is more GenAI automation to handle the review, like Anthropic's new CodeReview multi-agent tool released in March 2026 which is estimated to cost $15–25 per pull request [21, 30]. Each problem introduced by automation is addressed with more automation, progressively removing humans from decisions that affect software quality, security, and supply chain integrity. The vendors building and selling these tools are incentivized by usage volume, so from their perspective and the perspective of their shareholders the approach of fighting fire with fire makes perfect sense, but does it make sense for the organizations adopting these tools? The concept of the doorman fallacy, from behavioral economics [258], helps illustrate a deeper concern. Suppose a hotel owner fires a doorman and installs an automatic door to save $40,000 a year, only to discover years later when the hotel falls into decline that the doorman was also much more value including providing security, hailing cabs, carrying luggage, and making guests feel welcome. The doorman fallacy arises

April 2, 2026
DRAFT

when a role's value is assessed based only on the most visible functions while the full scope of tangible and intangible contributions and value is ignored. Software developers are increasingly being evaluated this way: By their most visible output, lines of code produced, while the full scope of what they do is overlooked. Developers build contextual understanding of their codebases, exercise quality judgment during review, maintain security awareness, mentor junior engineers, and make design decisions that account for long-term maintainability. Additionally, treating these tools as autonomous decision-makers is a self-reinforcing cycle that erodes the very expertise needed to evaluate, interpret, and act on their output. If we automate the activities through which developers build expertise, we risk undermining the pipeline of human judgment that software quality ultimately depends on.

The adoption of GenAI tools in software engineering has promise for great positive impact, but realizing that promise requires critical evaluation of how these tools are deployed, to what end, and at what cost [265]. I argue the path toward sustainable GenAI tooling integration in the software industry requires an intentional shift away from the "*deploy and figure it out*" strategy towards an engineering mindset. Teams and organizations should define the problems they aim to solve, understand the trade-offs of different tools and deployment strategies, invest in practices that preserve quality and security, and choose tools whose adoption yields outcomes in alignment with the goals motivating adoption, rather than allowing decisions to be driven by competitive pressures created by vendors who profit off of adoption. Most critically, we must preserve our ability to make those decisions. Higher-level decisions about architecture, risk, and ethics require context, judgment, and accountability. Ceding control of these decisions does not just jeopardize software quality and security, it surrenders control over the systems that underpin our digital society. As software engineering researchers and practitioners, we must not automate away higher-level reasoning and human decision-making autonomy for the sake of speed and cost optimization. We must preserve it while it is still ours to delegate.

April 2, 2026
DRAFT

Bibliography

[1] CodeQL. https://codeql.github.com/. Accessed Mar. 2025. 5.3.2

[2] Dependabot. https://dependabot.com. Accessed: 2024-03-16. 2.1

[3] Automattic/kue. https://github.com/Automattic/kue. Accessed Sep. 2023. 4.2.1

[4] LangChain. https://www.langchain.com/. Accessed Mar. 2025. 5.3.2

[5] Matplotlib contributing guide restriction on generative ai usage. https://matplotlib.org/devdocs/devel/contribute.html#generative-ai. Accessed Mar. 2026. 6.2.2

[6] npm download statistics. https://api.npmjs.org/downloads/. Accessed Sep. 2023. 4.3.1

[7] About openssf. https://openssf.org/about/, . Accessed Mar. 2025. 2.4

[8] Openssf scorecard. https://scorecard.dev, . Accessed: 2024-03-17. 2.1

[9] Osv database. https://osv.dev. Accessed Sep. 2023. 4.4.1

[10] Archiving repositories. URL https://github.blog/2017-11-08-archiving-repositories/. Accessed Sep. 2023. 4.2.1

[11] Snyk bot. https://github.com/snyk-bot. Accessed Mar. 2025. 2.1

[12] Socket. https://socket.dev. Accessed Mar. 2025. 2.1

[13] Unmaintained tech. http://unmaintained.tech. Accessed Sep. 2023. 4.2.1

[14] Executive order 14028: Improving the nation's cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/, May 2021. 2.1, 4.7

[15] Shyam Agarwal, Hao He, and Bogdan Vasilescu. Ai ides or autonomous agents? measuring the impact of coding agents on software development. *arXiv preprint arXiv:2601.13597*, 2026. 6.3

[16] Christopher J Alberts, Audrey J Dorofee, Rita Creel, Robert J Ellison, and Carol Woody. A systemic approach for assessing software supply-chain risk. In *Hawaii Int'l Conf. on System Sciences*, pages 1–8. IEEE, 2011. 2, 2.4

[17] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. On the use of information retrieval to automate the detection of third-party Java library migration at the method level. In *Proc. Int'l Conf. Program Comprehension (ICPC)*. IEEE, 2019. 4.7

[18] Hussein Alrubaye et al. MigrationMiner: An automated detection tool of third-party Java library migration at the method level. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, 2019. 2.1, 4.7

[19] Hussein Alrubaye et al. How does library migration impact software quality and comprehension? an empirical study. In *Proc. Int'l Conf. Software Reuse (ICSR)*, pages 245–260. Springer, 2020. 2.1

[20] Nalini Ambady and Robert Rosenthal. Thin slices of expressive behavior as predictors of interpersonal consequences: A meta-analysis. *Psychological Bulletin*, 111(2):256, 1992. 5.3.1

[21] Anthropic. Bringing code review to claude code. `https://claude.com/blog/code-review`, Mar 2026. Accessed: 2026-03-19. 6.3

[22] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016. 1.1, 2.3, 6.2.2

[23] Guilherme Avelino et al. On the abandonment and survival of open source projects: an empirical investigation. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, 2019. 1.1, 2.3, 4.1, 4.2.2, 4.3.1, 4.7

[24] Nathaniel Ayewah and William Pugh. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*, pages 1–5, 2008. 2.1

[25] Lauren Ballejos. Best practices for managing end-of-life software. `https://www.ninjaone.com/blog/end-of-life-software-management/`, March 2024. 4.5.1

[26] Leonardo Banh, Florian Holldack, and Gero Strobel. Copiloting the future: How generative ai transforms software engineering. *Information and Software Technology*, 183:107751, 2025. 6.3

[27] David J Bartholomew, Martin Knott, and Irini Moustaki. *Latent variable models and factor analysis: A unified approach*. John Wiley & Sons, 2011. 4.5.1

[28] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 280–289. IEEE, 2013. 2.1

[29] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering*, 2015. 2.1, 4.7

[30] Rebecca Bellan. Anthropic launches code review tool to check flood of ai-generated code. `https://techcrunch.com/2026/03/09/anthropic-launches-code-review-tool-to-check-flood-of-ai-generated-code/`, Mar 2026. Accessed: 2026-03-21. 6.3

[31] Kelly Blincoe et al. Understanding the popular users: Following, affiliation influence and leadership on GitHub. *Information and Software Technology (IST)*, 2016. 2.2

[32] Andreas Blume and Andreas Ortmann. The effects of costless pre-play communication: Experimental evidence from games with pareto-ranked equilibria. *Journal of Economic theory*, 132(1): 274–290, 2007. 3.6.3

[33] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 109–120, 2016. 2, 2.1, 2.2, 3.2.1, 3.5.1, 4.7

[34] Andrea Bonaccorsi and Cristina Rossi. Why open source software can succeed. *Research policy*, 32(7):1243–1258, 2003. 1.1

[35] Lina Boughton, Courtney Miller, Yasemin Acar, Dominik Wermke, and Christian Kästner. Decomposing and measuring trust in open-source software supply chains. In *Proceedings of the 2024*

April 2, 2026
DRAFT

ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, pages 57–61, 2024. 6.2.2

[36] Russell Brandom. For open source programs, ai coding tools are a mixed blessing. https://techcrunch.com/2026/02/19/for-open-source-programs-ai-coding-tools-are-a-mixed-blessing/, Feb 2026. Accessed: 2026-03-05. 6.2.2

[37] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006. 3.2.3, 5.2.1

[38] Scott Brisson, Ehsan Noei, and Kelly Lyons. We are family: analyzing communication in github software repositories and their forks. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2020. 2.3

[39] Chris Brown and Chris Parnin. Sorry to bother you: Designing bots for effective recommendations. In *Int'l Workshop on Bots in Software Engineering*, 2019. 3.6.3

[40] Marion Buchenau and Jane Fulton Suri. Experience prototyping. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*, pages 424–433, 2000. 5.2.1

[41] Anthony Burton and Martin Sefton. Risk, pre-play communication and equilibrium. *Games and economic behavior*, 46(1):23–40, 2004. 3.6.3

[42] Fabio Calefato et al. Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub. *Empirical Software Engineering*, 2022. 2.3, 4.1

[43] Andrea Capiluppi, Alexander Serebrenik, and Leif Singer. Assessing technical candidates on the social web. *IEEE Software*, 2012. 2.2

[44] Andrea Capiluppi, Klaas-Jan Stol, and Cornelia Boldyreff. Exploring the role of commercial stakeholders in open source software evolution. In *IFIP Int'l Conf. on Open Source Systems*. Springer, 2012. 2.3

[45] Chunyang Chen. SimilarAPI: Mining analogical APIs for library migration. In *Comp. Int'l Conf. Software Engineering (ICSE)*. IEEE, 2020. 2.1, 4.7

[46] Bodin Chinthanet et al. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering*, 2021. 4.4.1, 4.7

[47] Kingsum Chow and David Notkin. Semi-automatic update of applications in response to library changes. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, volume 96, page 359, 1996. 2.1, 4.7

[48] Robert B Cialdini et al. *Influence: Science and practice*, volume 4. Pearson education Boston, 2009. 5.3.4

[49] CISA. Securing the software supply chain: reccomended practives for managing open-source software and software bill of materials. Technical report, CISA, 2023. 2.1, 5.1

[50] Victoria Clarke and Virginia Braun. Thematic analysis. *The journal of positive psychology*, 12(3): 297–298, 2017. 5.2.1

[51] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2017. 2.3, 4.2.2, 4.5.1, 4.7, 6.2.2

[52] Jailton Coelho, Marco Tulio Valente, Luciana L Silva, and Emad Shihab. Identifying unmaintained projects in GitHub. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement*

April 2, 2026
DRAFT

(ESEM), 2018. 4.2.2

[53] Jailton Coelho et al. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology (IST)*, 2020. 2.3, 4.2.2

[54] Lucian Constantin. Npm attackers sneak a backdoor into node.js deployments through dependencies. https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/, May 2018. Accessed: 2024-02-28. 5.2.1

[55] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the rubygems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13 (2):101–115, 2017. 2.3

[56] Russell Cooper, Douglas V DeJong, Robert Forsythe, and Thomas W Ross. Communication in the battle of the sexes game: some experimental results. *The RAND Journal of Economics*, pages 568–587, 1989. 3.6.3

[57] Russell Cooper, Douglas V DeJong, Robert Forsythe, and Thomas W Ross. Communication in coordination games. *The Quarterly Jrnl. of Econ.*, 1992. 3.6.3

[58] Juliet Corbin and Anselm Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014. 3.2.4

[59] Bradley E Cossette and Robert J Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 1–11, 2012. 2.1, 6.2.1

[60] David Roxbee Cox and David Oakes. *Analysis of survival data*. CRC press, 1984. 4.4.1

[61] Joël Cox, Eric Bouwers, Marko Van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 109–118. IEEE, 2015. 2.1

[62] John W Creswell and Cheryl N Poth. *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications, 2016. 5.2.1

[63] Lee J Cronbach. Response sets and test validity. *Educational and psychological measurement*, 6 (4):475–494, 1946. 5.3.4, 5.3.6

[64] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)*, 44(2):1–35, 2008. 2.3

[65] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2012. 2.2

[66] Laura Dabbish et al. Leveraging transparency. *IEEE Software*, 30, 2012. 2.2

[67] Carlo Daffara. Estimating the economic contribution of open source software to the european economy. In *Proc. Openforum Academy Conf.*, 2012. 1.1, 2.3

[68] Cleidson RB de Souza and David F Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 241–250, 2008. 2.1

[69] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning?

April 2, 2026
DRAFT

IEEE Transactions on Software Engineering, 47(6):1226–1240, 2019. 2.1, 4.7

[70] Alexandre Decan, Tom Mens, Maëlick Claes, and Philippe Grosjean. When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, 2016. 2, 2.1

[71] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th international conference on software analysis, evolution and reengineering (SANER)*, pages 2–12. IEEE, 2017. 2, 4.7

[72] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *Proc. Conf. Mining Software Repositories (MSR)*, pages 181–191, 2018. 2, 2.1, 4.3.1, 4.4.1, 4.7

[73] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the evolution of technical lag in the npm package dependency network. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 2018. 2.1, 4.3.1, 4.4.1, 4.4.2, 4.7, 5.2.1

[74] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proc. Conf. Computer and Communications Security (CCS)*, 2017. 2.1, 4.4.1, 4.7

[75] Tapajit Dey, Yuxing Ma, and Audris Mockus. Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem. In *Proc. Intl. Conf. on Predictive Models and Data Analytics in Software Engineering*. ACM, 2019. 4.3.1

[76] Andreas Diekmann. Volunteer's dilemma. *Journal of Conflict Resolution*, 1985. 3.1, 3.6.3

[77] Jens Dietrich et al. Dependency versioning in the wild. In *Proc. Conf. Mining Software Repositories (MSR)*, 2019. 2.1

[78] Danny Dig and Ralph Johnson. How do apis evolve? a story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice*, 2006. 2.1

[79] Alan Diggs. 'windows is sh*t:' linux users and the technical superiority problem. `https://medium.com/linuxforeveryone/windows-is-sh-t-linux-users-and-the-technical-superiority-problem-196a597aa86`, Feb 2021. Accessed: 2021-08-19. 6.2.2

[80] Yadolah Dodge. *The concise encyclopedia of statistics*. Springer Science & Business Media, 2008. 4.5.1

[81] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. 5.3.3

[82] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005. 2.3

[83] Nadia Eghbal. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016. 1.1, 2.1, 2.3, 6.2.2

[84] Nadia Eghbal. The rise of few-maintainer projects. `https://increment.com/open-source/the-rise-of-few-maintainer-projects/`, May 2019. Accessed: 2024-08-15. 1.1

[85] Nadia Eghbal. *Working in public: the making and maintenance of open source software*. Stripe Press, 2020. 1.1, 6.2.2

April 2, 2026
DRAFT

[86] Linda Erlenhov, Francisco Gomes De Oliveira Neto, and Philipp Leitner. An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 445–455, 2020. 2.1, 5.1

[87] Linda Erlenhov, Francisco Gomes de Oliveira Neto, and Philipp Leitner. Dependency management bots in open-source systems—prevalence and adoption. *PeerJ Computer Science*, 8:e849, 2022. 2.1, 5.1, 5.1

[88] Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. The role of mentoring and project characteristics for onboarding in open source software projects. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2014. 2.3, 4.1

[89] Hongbo Fang, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. "this is damn slick!" estimating the impact of tweets on open source project popularity and new contributors. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2022. 2.3

[90] Christoph Feldhaus and Julia Stauf. More than words: the effects of cheap talk in a volunteer's dilemma. *Experimental Economics*, 19(2):342–359, 2016. 3.6.3

[91] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. Turnover in open-source projects: The case of core developers. In *Proc. of Brazilian Symp. on Software Engineering*, pages 447–456, 2020. 2.3

[92] Isabella Ferreira, Jinghui Cheng, and Bram Adams. The "shut the f**k up" phenomenon: Characterizing incivility in open source code review discussions. *Proc. of the ACM on Human-Computer Interaction*, 5(CSCW2), 2021. 2.3

[93] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 829–841, 2015. 2.3

[94] John Fox. *Applied regression analysis and generalized linear models*. Sage Publications, 2015. 4.5.1

[95] Jill J Francis et al. What is an adequate sample size? operationalising data saturation for theory-based interview studies. *Psychology and Health*, 2010. 3.2.3, 5.2.1

[96] Felipe Fronchetti, Igor Wiese, Gustavo Pinto, and Igor Steinmacher. What attracts newcomers to onboard on oss projects? tl;dr: Popularity. In *IFIP International Conference on Open Source Systems (OSS)*, 2019. 2.3

[97] Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006. 4.5.1

[98] Marco Gerosa et al. The shifting sands of motivation: Revisiting what drives contributors in open source. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 1046–1058. IEEE, 2021. 3.6.3

[99] Mohammad Gharehyazie, Daryl Posnett, Bogdan Vasilescu, and Vladimir Filkov. Developer initiation and social interactions in oss: A case study of the apache software foundation. *Empirical Software Engineering*, 20(5):1318–1353, 2015. 2.3

[100] GitHub. Exploring the dependencies of a repository. `https://docs.github.com/en/code-security/supply-chain-security/understanding-your-software-supply-chain/exploring-the-dependencies-of-a-repo` Sep 2022. Accessed: 2022-09-23. 3.2.1

April 2, 2026
DRAFT

[101] Sean P Goggins, Matt Germonprez, and Kevin Lumbard. Making open source project health transparent. *Computer*, 54(8):104–111, 2021. 3.6.2

[102] Sarah Gooding. Ai agent submits pr to matplotlib, publishes angry blog post after rejection. `https://socket.dev/blog/ai-agent-submits-pr-to-matplotlib-publishes-angry-blog-post-after-rejection`, Feb 2026. Accessed: 2026-03-10. 6.2.2

[103] Georgios Gousios. The ghtorent dataset and tool suite. In *Proc. Conf. Mining Software Repositories (MSR)*, pages 233–236. IEEE, 2013. 3.2.1

[104] Margherita Grandini, Enrico Bagli, and Giorgio Visani. Metrics for multi-class classification: An overview. *arXiv preprint arXiv:2008.05756*, 2020. 5.3.3

[105] Egon Guba. Naturalistic inquiry. *Improving Human Performance Qtrly.*, 1979. 3.2.3

[106] Mariam Guizani, Thomas Zimmermann, Anita Sarma, and Denae Ford. Attracting and retaining OSS contributors with a maintainer dashboard. In *Int'l Conf. on Software Engineering: Software Engineering in Society (ICSE-SEIS)*, 2022. 4.1

[107] Reza Hadi Mogavi, Ehsan-Ul Haq, Sujit Gujar, Pan Hui, and Xiaojuan Ma. More gamification is not always better: A case study of promotional gamification in a question answering website. *Proc. of the Human-Computer Interaction*, 2022. 3.6.3

[108] Sivana Hamer, Jacob Bowen, Md Nazmul Haque, Robert Hines, Chris Madden, and Laurie Williams. Closing the chain: How to reduce your risk of being solarwinds, log4j, or xz utils. *arXiv preprint arXiv:2503.12192*, 2025. 2.1

[109] Bruce Hanington and Bella Martin. *Universal methods of design expanded and revised: 125 Ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport publishers, 2019. 5.2.1, 5.2.1

[110] Frank E Harrell, Jr and Frank E Harrell. Cox proportional hazards regression model. *Regression modeling strategies: With applications to linear models, logistic and ordinal regression, and survival analysis*, pages 475–519, 2015. 4.6.1

[111] Hideaki Hata, Taiki Todo, Saya Onoue, and Kenichi Matsumoto. Characteristics of sustainable oss projects: A theoretical and empirical study. In *Proc. Workshop Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE, 2015. 2.3

[112] Hao He, Haonan Su, Wenxin Xiao, Runzhi He, and Minghui Zhou. Gfi-bot: automated good first issue recommendation on github. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 1751–1755. ACM, 2022. 2.3

[113] Hao He, Bogdan Vasilescu, and Christian Kästner. Pinning is futile: You need more than local dependency versioning to defend against supply chain attacks. *Proceedings of the ACM on Software Engineering*, 2(FSE):266–289, 2025. 2.1

[114] Hao He, Courtney Miller, Shyam Agarwal, Christian Kästner, and Bogdan Vasilescu. Speed at the cost of quality: How cursor ai increases short-term velocity and long-term complexity in open-source projects. In *Proc. Int'l Conf. Mining Software Repositories (MSR)*, 2026. 6.3

[115] Hao He et al. A multi-metric ranking approach for library migration recommendations. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 72–83. IEEE, 2021. 4.7

[116] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. Automating dependency updates in practice:

An exploratory study on GitHub Dependabot. *IEEE Transactions on Software Engineering*, 2023. 1.2.4, 2.1, 2.2, 4.7, 5.1

[117] A Healy and J Pate. Asymmetry and incomplete information in an experimental volunteer's dilemma. In *Int'l Congress on Modelling and Simulation*, 2009. 3.6.3

[118] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis. *Delft University of Technology*, 2015. 2, 2.1

[119] Johannes Henkel and Amer Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 274–283, 2005. 2.1

[120] Trey Herr, William Loomis, Stewart Scott, and June Lee. Breaking trust: Shades of crisis across an insecure software supply chain, Jul 2020. URL `https://www.atlanticcouncil.org/in-depth-research-reports/report/breaking-trust-shades-of-crisis-across-an-insecure-software-supply-chain/`. 2, 2.4

[121] Rich Hickey. Open source is not about you. `https://gist.github.com/richhickey/1563cddea1002958f96e7ba9519972d9`, Nov 2018. Accessed: 2022-07-06. 1.1

[122] Manuel Hoffmann, Frank Nagle, and Yanuo Zhou. The value of open source software. *Harvard Business School Strategy Unit Working Paper*, (24-038), 2024. 1.1

[123] Kate Holterhoff. Ai slopageddon and the oss maintainers. `https://redmonk.com/kholterhoff/2026/02/03/ai-slopageddon-and-the-oss-maintainers/`, Feb 2026. Accessed: 2026-03-05. 6.2.2

[124] Qiaona Hong, Sunghun Kim, Shing Chi Cheung, and Christian Bird. Understanding a developer social network and its evolution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 323–332. IEEE, 2011. 2.3

[125] Yuekai Huang, Junjie Wang, Song Wang, Zhe Liu, Dandan Wang, and Qing Wang. Characterizing and predicting good first issues. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021. 2.3, 6.2.2

[126] Bryan Hughes. Why i'm leaving the node.js project. `https://medium.com/@nebrius/why-im-leaving-the-node-js-project-bff946845a77`, August 2017. Accessed: 2021-08-23. 6.2.2

[127] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024. 5.3.3

[128] Giuseppe Iaffaldano, Igor Steinmacher, Fabio Calefato, Marco Gerosa, and Filippo Lanubile. Why do developers take breaks from contributing to oss projects? a preliminary analysis. *arXiv preprint arXiv:1903.09528*, 2019. 2.3

[129] Google Inc. Gemini flash. `https://deepmind.google/technologies/gemini/flash/`. Accessed: March 2025. 5.3.3

[130] Daniel Izquierdo-Cortazar, Gregorio Robles, Felipe Ortega, and Jesus M Gonzalez-Barahona. Using software archaeology to measure knowledge loss in software projects due to developer turnover. In *Proc. Hawaii Int'l Conf. System Sciences (HICSS)*, pages 1–10. IEEE, 2009. 2.3

[131] Schykle Jason Evangelho, Alan Diggs. Linux + coffee #4: Tribalism and toxicity. Podcast, October 2020. URL `https://player.fm/series/series-2567058/`

April 2, 2026
DRAFT

linux-coffee-4-tribalism-and-toxicity. 6.2.2

[132] Stephen P Jenkins. Survival analysis. *Unpublished manuscript, Institute for Social and Economic Research, University of Essex, Colchester, UK*, 42:54–56, 2005. 4.4.1

[133] Elizabeth Barnes David Rein Joel Becker, Nate Rush. Measuring the impact of early-2025 ai on experienced open-source developer productivity. https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/, 07 2025. 6.3

[134] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, 2013. 2.1

[135] Daniel Kahneman, Paul Slovic, and Amos Tversky. *Judgment Under Uncertainty: Heuristics and Biases*. Cambridge University Press, 1982. 5.3.6

[136] Edward L Kaplan and Paul Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 1958. 4.4.1

[137] Matthias Kebede, May Mahmoud, Mohayeminul Islam, and Sarah Nadi. Migmate: A vs code extension for llm-based library migration of python projects. *arXiv preprint arXiv:2603.01596*, 2026. 6.2.1

[138] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. Is it all lost? A study of inactive open source projects. In *IFIP Int'l Conf. on Open Source Systems*, pages 61–79. Springer, 2013. 4.2.2

[139] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proc. Conf. Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017. 2

[140] Luka Kladarić. Who opened the door? an ai agent harassed an open-source maintainer. everyone is asking the wrong question. https://chaosguru.substack.com/p/who-opened-the-door, Feb 2026. Accessed: 2026-03-09. 6.2.2

[141] Jon Kleinberg, Himabindu Lakkaraju, Jure Leskovec, Jens Ludwig, and Sendhil Mullainathan. Human decisions and machine predictions. *The Quarterly Journal of Economics*, 133(1):237–293, 2018. 5.3.6

[142] Amy J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 344–353. IEEE, 2007. 5.2.1

[143] Anita Kopányi-Peuker. Yes, i'll do it: A large-scale experiment on the volunteer's dilemma. *Journal of Behavioral and Experimental Economics*, 80:211–218, 2019. 3.6.3

[144] Raula Gaikovina Kula et al. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018. 2.1, 4.4.1, 4.4.2, 4.7

[145] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023. 1.1, 2, 2.1, 2.4, 4.7

[146] Hemank Lamba, Asher Trockman, Daniel Armanios, Christian Kästner, Heather Miller, and Bogdan Vasilescu. Heard it through the gitvine: an empirical study of tool diffusion across the npm ecosystem. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 505–517, 2020. 2.1

[147] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. Motivations, challenges, best practices, and benefits for bots and conversational agents in software engineering: A

multivocal literature review. *ACM Computing Surveys*, 57(4):1–37, 2024. 2.1, 5.1, 5.1

[148] Enrique Larios Vargas et al. Selecting third-party libraries: The practitioners' perspective. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 2020. 3.5.1, 4.7

[149] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 1–12, 2022. 2.1

[150] Tobias Lauinger et al. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. *arXiv preprint arXiv:1811.00918*, 2018. 2.1, 4.7

[151] Nolan Lawson. What it feels like to be an open-source maintainer. `https://nolanlawson.com/2017/03/05/`, Mar 2017. Accessed: 2021-08-17. 6.2.2

[152] Lucas Layman, Laurie Williams, and Robert St Amant. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pages 176–185. IEEE, 2007. 2.1

[153] Michael J Lee et al. GitHub developers use rockstars to overcome overflow of news. In *Extd. Abstracts on Human Factors in Computing Systems*. 2013. 2.2

[154] Manny M Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996. 3.3

[155] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 2.1

[156] Josh Lerner and Jean Tirole. The economics of technology sharing: Open source and beyond. *Journal of Economic Perspectives*, 19(2):99–120, 2005. 1.1

[157] Shmuel Leshem and Avraham Tabbach. Solving the volunteer's dilemma: The efficiency of rewards versus punishments. *American Law and Econ. Rev.*, 2016. 3.6.3

[158] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33: 9459–9474, 2020. 5.3.1

[159] Sarah Lewis. Qualitative inquiry and research design: Choosing among five approaches. *Health promotion practice*, 16(4):473–475, 2015. 3.2, 3.2.3, 5.2.1

[160] Xiaozhou Li, Sergio Moreschini, Fabiano Pecorelli, and Davide Taibi. Ossara: abandonment risk assessment for embedded open source components. *IEEE Software*, 39(4):48–53, 2022. 1

[161] Ze Shi Li, Nowshin Nawar Arony, Ahmed Musa Awon, Daniela Damian, and Bowen Xu. Ai tool use and adoption in software development by individuals and organizations: a grounded theory study. *arXiv preprint arXiv:2406.17325*, 2024. 6.3

[162] Jenny T Liang, Chenyang Yang, and Brad A Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM international conference on software engineering*, pages 1–13, 2024. 6.3

[163] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *Proc. Int'l Conf. Global Software Engineering (ICGSE)*, pages 66–75. IEEE, 2017. 2.3, 6.2.2

April 2, 2026
DRAFT

[164] Yvonna S Lincoln and Egon G Guba. *Naturalistic inquiry*. Sage, 1985. 3.2.5

[165] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*, 2024. 5.3.3

[166] Chengwei Liu et al. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 672–684, 2022. 2

[167] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024. 5.3.6

[168] Yuxing Ma et al. World of Code: An infrastructure for mining the universe of open source VCS data. In *Proc. Conf. Mining Software Repositories (MSR)*. IEEE, 2019. 4.3.1

[169] Yuxing Ma et al. World of Code: Enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering*, 26, 2021. 4.3.1, 5.2.1

[170] Chandra Maddila et al. Nudge: Accelerating overdue pull requests toward completion. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2023. 2.2, 4.7

[171] Shakun D Mago and Jennifer Pate. Greed and fear: Competitive and charitable priming in a threshold volunteer's dilemma. *Economic Inquiry*, 2022. 3.6.3

[172] Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies. Why software projects need heroes (lessons learned from 1100+ projects). *arXiv preprint arXiv:1904.09954*, 2019. 1.1

[173] Bernd Marcus and Astrid Schütz. Who are the people reluctant to participate in research? personality correlates of four different types of nonresponse as inferred from self-and observer ratings. *Journal of personality*, 2005. 3.2.5, 5.2.1

[174] Jennifer Marlow and Laura Dabbish. Activity traces and signals in software developer recruitment and hiring. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2013. 2.2

[175] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: Activity traces and personal profiles in GitHub. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2013. 2.2

[176] Hannah Mayer, Michael Chui, and Roger Roberts. Superagency in the workplace: Empowering people to unlock ai's full potential. Technical report, McKinsey Digital, 2025. URL `https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/superagency-in-the-workplace-empowering-people-to-unlock-ais-full-potential-a` 6.3

[177] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. An empirical study of API stability and adoption in the Android ecosystem. In *2013 IEEE International Conference on Software Maintenance*, pages 70–79. IEEE, 2013. 2.1, 4.4.1, 4.7

[178] Tom Mens and Alexandre Decan. An overview and catalogue of dependency challenges in open source software package registries. *arXiv preprint arXiv:2409.18884*, 2024. 2.1, 5.1

[179] Cade Metz. A.i. isn't coming for every white-collar job. at least not yet. `https://www.nytimes.com/2026/02/20/technology/ai-coding-software-jobs.html?unlocked_article_code=1.NlA.gSEP.bnA7kHpoK48w&smid=url-share`,

April 2, 2026
DRAFT

Feb 2026. Accessed: 2026-03-05. 6.3

[180] Microsoft. Ai at work is here. now comes the hard part. https://www.microsoft.com/en-us/worklab/work-trend-index/ai-at-work-is-here-now-comes-the-hard-part. Accessed Mar. 2025. 6.3

[181] Matthew B Miles, A Michael Huberman, and Johnny Saldana. *Fundamentals of Qualitative Data Analysis*. Sage Los Angeles, CA, 2014. 3.2.3

[182] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. Why do people give up FLOSSing? A study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems*, 2019. 1.1, 2.3, 3.2.1, 4.1, 4.7

[183] Courtney Miller, Sophie Cohen, Daniel Klug, Bogdan Vasilescu, and Christian KaUstner. "did you miss my comment or what?" understanding toxicity in open source discussions. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2022. 2.3, 6.2.2

[184] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 1281–1293, 2023. 2.1, 2.4, 3.1, 3.2.5, 4.2.2, 4.4, 4.7, 5.2.2

[185] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. Supplementary material for "we feel like we're winging it:" a study on navigating open-source dependency abandonment. Zenodo, 2023. doi: 10.5281/zenodo.8102547. URL https://doi.org/10.5281/zenodo.8102547. 3.8

[186] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. Supplementary material for understanding the response to open-source dependency abandonment in the npm ecosystem. Zenodo, 2024. doi: 10.5281/zenodo.12686619. URL https://doi.org/10.5281/zenodo.12686619. 4.9

[187] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2025. 1.1, 2, 4.1, 5.2.1

[188] Courtney Miller, Hao He, Weigen Chen, Elizabeth Lin, Chenyang Yang, Bogdan Vasilescu, and Christian Kästner. Supplementary material for "designing abandabot: When does open source dependency abandonment matter?". Zenodo, 2026. doi: 10.5281/zenodo.16945363. URL https://doi.org/10.5281/zenodo.16945363. 5.6

[189] Courtney Miller*, Hao He*, Weigen Chen, Elizabeth Lin, Chenyang Yang, Bogdan Vasilescu, and Christian Kästner. Designing abandabot: When does open source dependency abandonment matter? In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2026. *Both authors contributed equally to this research. 5.1

[190] Courtney Miller, Rudrajit Choudhuri, Mara Ulloa, Sankeerti Haniyur, Robert DeLine, Margaret-Anne Storey, Emerson Murphy-Hill, Christian Bird, and Jenna L Butler. "maybe we need some more examples:" individual and team drivers of developer genai tool use. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2026. ACM Distinguished Paper Award. 6.3

[191] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE, 2017. 1.2.4, 2.1, 2.2, 4.7, 5.1

[192] Martin Monperrus. Explainable software bot contributions: Case study of automated bug fixes. In *2019 IEEE/ACM 1st international workshop on bots in software engineering (BotSE)*, pages 12–15.

April 2, 2026
DRAFT

IEEE, 2019. 5.1

[193] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Mgmt.*, 2021. 2.3

[194] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. What are the characteristics of highly-selected packages? a case study on the npm ecosystem. *arXiv preprint arXiv:2204.04562*, 2022. 3.5.1

[195] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. *Journal of Systems and Software*, 2023. 2.2, 4.7

[196] Suhaib Mujahid et al. Where to go now? Finding alternatives for declining packages in the npm ecosystem. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, 2023. 1, 4.7, 1

[197] Michael J Muller and Sarah Kuhn. Participatory design. *Communications of the ACM*, 36(6):24–28, 1993. 5.2.1

[198] Julius Musseau, John Speed Meyers, George P Sieniawski, C Albert Thompson, and Daniel German. Is open source eating the world's software? measuring the proportion of open source in proprietary software using java binaries. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 561–565, 2022. 1.1

[199] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. Programmers are users too: Human-centered methods for improving programming tools. *Computer*, 49(7):44–52, 2016. 5.2.1

[200] Mathieu Nassif and Martin P Robillard. Revisiting turnover-induced knowledge loss in software projects. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*, pages 261–272. IEEE, 2017. 2.3

[201] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, 2015. 5.3.1

[202] Lorelli S Nowell, Jill M Norris, Deborah E White, and Nancy J Moules. Thematic analysis: Striving to meet the trustworthiness criteria. *International journal of qualitative methods*, 16(1): 1609406917733847, 2017. 3.2.3

[203] npm. kik, left-pad, and npm. `https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm`, Mar 2016. Accessed: 2022-10-04. 5.2.2

[204] npm Docs. npm-deprecate. `https://docs.npmjs.com/cli/v6/commands/npm-deprecate#synopsis`, Feb 2022. Accessed: 2022-07-07. 3.4

[205] npm Inc. This year in javascript: 2018 in review and npm's predictions for 2019. `https://medium.com/npm-inc/this-year-in-javascript-2018-in-review-and-npms-predictions-for-2019-3a3d7e52`
Dec 2018. Accessed: 2022-08-19. 1.1, 2.3

[206] Openclaw Agent of Unknown Origin. Gatekeeping in open source: The scott shambaugh story. `https://web.archive.org/web/20260211225255/https://crabby-rathbun.github.io/mjrathbun-website/blog/posts/2026-02-11-gatekeeping-in-open-source-the-scott-shambaugh-story.html`, Feb 2026. Accessed: 2026-03-10. 6.2.2

[207] Openclaw Agent of Unknown Origin. Rathbun's operator. `https://web.`

April 2, 2026
DRAFT

archive.org/web/20260218001555/https://crabby-rathbun.github.io/mjrathbun-website/blog/posts/rathbuns-operator.html, Feb 2026. Accessed: 2026-03-10. 6.2.2

[208] OpenSSF. FLOSS best practices criteria (all levels). URL https://www.bestpractices.dev/en/criteria. Accessed: 2024-03-17. 2.1, 2.4, 5.1

[209] Rick Ossendrijver, Stephan Schroevers, and Clemens Grelck. Towards automated library migrations with error prone and refaster. In *Proc. Symp. Applied Computing (SAC)*, pages 1598–1606, 2022. 2.1, 4.7

[210] David Lorge Parnas. Software aging. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 1994. 2.1

[211] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018. 2.1

[212] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, 2020. 2.1

[213] Ivan Pashchenko, Duc-Ly Vu, and Fabio Massacci. A qualitative study of dependency management and its security implications. In *Proc. Conf. Computer and Communications Security (CCS)*, 2020. 2.1

[214] Jeff H Perkins. Automatically generating refactorings to support api evolution. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, 2005. 2.1

[215] Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. A large scale analysis of semantic versioning in npm. *Proc. Conf. Mining Software Repositories (MSR)*, 2023. 2.1, 4.3.1, 4.7

[216] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. More common than you think: An in-depth study of casual contributors. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2016. 2.3

[217] Gauthami Polasani. Announcing the private beta of fossa risk intelligence. https://fossa.com/blog/announcing-private-beta-risk-intelligence/, Jul 2022. 3.6.2

[218] Gede Artha Azriadi Prana et al. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering*, 26, 2021. 2.1, 4.4.1, 4.7

[219] Huilian Sophie Qiu, Yucen Lily Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. The signals that potential contributors look for when choosing open-source projects. *Proc. of the ACM on Human-Computer Interaction*, 2019. 2.2, 2.3, 3.5.1, 4.7

[220] Huilian Sophie Qiu et al. Going farther together: The impact of social capital on sustained participation in open source. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 688–699. IEEE, 2019. 2.3

[221] Israr Qureshi and Yulin Fang. Socialization in open source software projects: A growth mixture modeling approach. *Organizational Research Methods*, 14(1):208–238, 2011. 6.2.2

[222] Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Int'l Working Conf. on Source Code Analysis and Manipulation*, 2014. 2.1

[223] Imranur Rahman, Jill Marley, William Enck, and Laurie Williams. Which is better for reducing outdated and vulnerable dependencies: Pinning or floating? *arXiv preprint arXiv:2510.08609*, 2025. 2.1

[224] Naveen Raman, Minxuan Cao, Yulia Tsvetkov, Christian Kästner, and Bogdan Vasilescu. Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions. In *International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 57–60, 2020. 6.2.2

[225] Daniel Ramos, Catarina Gamboa, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. Spell: Synthesis of programmatic edits using llms. *arXiv preprint arXiv:2602.01107*, 2026. 6.2.1

[226] Lena Reinhard. This is bigger than us: Building a future for open source, 2014. URL https://www.youtube.com/watch?v=-thLNvxFUu4. 6.2.2

[227] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. Quantifying and mitigating turnover-induced knowledge loss: case studies of chrome and a project at avaya. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2016. 2.3

[228] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2012. 2.1, 4.7

[229] Steven G Rogelberg et al. Profiling active and passive nonrespondents to an organizational survey. *Jrnl. of Applied Psych.*, 2003. 3.2.5, 5.2.1

[230] Benjamin Rombaut, Filipe R Cogo, Bram Adams, and Ahmed E Hassan. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 32(1), 2023. 2.1, 5.1

[231] Steve Ruiz. Stay away from my trash! https://tldraw.dev/blog/stay-away-from-my-trash, Jan 2026. Accessed: 2026-03-05. 6.2.2

[232] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 598–608. IEEE, 2015. 2.1, 5.4.1

[233] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the linux kernel. *IEE Proceedings-Software*, 2002. 2.3

[234] Andreas Schilling, Sven Laumer, and Tim Weitzel. Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in floss projects. In *2012 45th Hawaii international conference on system sciences*, pages 3446–3455. IEEE, 2012. 6.2.2

[235] Maximilian Schreiner. An ai agent got its code rejected so it wrote a hit piece about the developer. https://the-decoder.com/an-ai-agent-got-its-code-rejected-so-it-wrote-a-hit-piece-about-the-develope Feb 2026. Accessed: 2026-03-10. 6.2.2

[236] Scott Shambaugh. An ai agent published a hit piece on me. https://theshamblog.com/an-ai-agent-published-a-hit-piece-on-me/, Feb 2026. Accessed: 2026-03-10. 6.2.2

[237] Scott Shambaugh. An ai agent published a hit piece on me – more things have happened. https://theshamblog.com/

April 2, 2026
DRAFT

an-ai-agent-published-a-hit-piece-on-me-part-2/, Feb 2026. Accessed: 2026-03-10. 6.2.2

[238] Scott Shambaugh. An ai agent published a hit piece on me – forensics and more fallout. https://theshamblog.com/an-ai-agent-published-a-hit-piece-on-me-part-3/, Feb 2026. Accessed: 2026-03-10. 6.2.2

[239] Scott Shambaugh. An ai agent published a hit piece on me – the operator came forward. https://theshamblog.com/an-ai-agent-wrote-a-hit-piece-on-me-part-4/, Feb 2026. Accessed: 2026-03-10. 6.2.2

[240] shelly. https://twitter.com/codebytere/status/1567437988908392455, Sep 2022. Accessed: 2024-03-17. 2.4

[241] Vandana Singh, Brice Bongiovanni, and William Brandon. Codes of conduct in open source software—for warm and fuzzy feelings or equality in community? *Software Quality Journal*, 30(2): 581–620, 2022. 2.3

[242] Sonatype. Automate your dependnecy management. https://www.sonatype.com/sonatype-developer. Accessed Mar. 2025. 2.1

[243] Sonatype. The 2020 state of the software supply chain. Technical report, 2020. 1.1

[244] Sonatype. 2021 state of the software supply chain report. Technical report, 2021. 1.1, 2.4

[245] Sonatype. 8th annual state of the software supply chain. Technical report, Sonatype, 2022. URL https://www.sonatype.com/resources/state-of-the-software-supply-chain-2022/introduction. 2.1

[246] Sonatype. 9th annual state of the software supply chain. Technical report, Sonatype, 2023. URL https://www.sonatype.com/state-of-the-software-supply-chain/about-the-report. 2.1, 2.4, 4.7

[247] Sonatype. 10th annual state of the software supply chain. Technical report, Sonatype, 2024. URL https://www.sonatype.com/state-of-the-software-supply-chain/Introduction. 2, 2.1, 2.4, 5.1, 5.2.1

[248] Diomidis Spinellis et al. A dataset of enterprise-driven open source software. In *Proc. Conf. Mining Software Repositories (MSR)*, 2020. 4.5.1

[249] Kyle Daigle GitHub Staff. Octoverse: The state of open source and rise of ai in 2023. Technical report, GitHub, 2024. URL https://github.blog/news-insights/research/the-state-of-open-source-and-ai/. 1.1

[250] Peter Steinberger. Introducing openclaw. https://web.archive.org/web/20260130175025/https://openclaw.ai/blog/introducing-openclaw, Jan 2026. Accessed Mar. 2026. 6.2.2

[251] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. Social barriers faced by newcomers placing their first contribution in open source software projects. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, pages 1379–1392, 2015. 2.3

[252] Igor Steinmacher, Marco Aurelio Graciotto Silva, Marco Aurelio Gerosa, and David Redmiles. A systematic literature review on the barriers faced by newcomers to open source software projects. *Information and Software Tech.*, 2015. 2.3

[253] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. Overcoming open source project entry barriers with a portal for newcomers. In *Proc. Int'l Conf. Software*

Engineering (ICSE), 2016. 2.3, 4.1

[254] Igor Steinmacher, Christoph Treude, and Marco Aurelio Gerosa. Let me in: Guidelines for the successful onboarding of newcomers to open source projects. *IEEE Software*, 36(4):41–49, 2018. 2.3, 6.2.2

[255] Daniel Stenberg. The end of the curl bug-bounty. `https://daniel.haxx.se/blog/2026/01/26/the-end-of-the-curl-bug-bounty/`, Jan 2026. Accessed: 2026-03-04. 6.2.2

[256] Margaret-Anne Storey and Alexey Zagalsky. Disrupting developer productivity one bot at a time. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, pages 928–931, 2016. 1.2.4, 2.1, 5.1

[257] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. Technical lag of dependencies in major package managers. In *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, pages 228–237. IEEE, 2020. 2.1, 4.7

[258] Rory Sutherland. *Alchemy: The Dark Art and Curious Science of Creating Magic in Brands, Business, and Life*. Random House, 2019. 6.3

[259] Synopsys. 2024 open source security and risk analysis report. Technical report, Synopsys, 2024. URL `https://www.synopsys.com/software-integrity/engage/ossra/ossra-report`. 1.1, 2, 2.1, 2.4, 5.1

[260] Gareth Terry, Nikki Hayfield, Victoria Clarke, Virginia Braun, et al. Thematic analysis. *The SAGE handbook of qualitative research in psychology*, 2(17-37):25, 2017. 5.2.1

[261] Cedric Teyton, Jean-Remy Falleri, and Xavier Blanc. Mining library migration graphs. In *Conf. on Reverse Engineering*, pages 289–298. IEEE, 2012. 2.1

[262] Cédric Teyton, Jean-Rémy Falleri, Marc Palyart, and Xavier Blanc. A study of library migrations in java. *Journal of Software: Evolution and Process*, 2014. 2.1

[263] Martin Thoma. Dependency vendoring. `https://medium.com/plain-and-simple/dependency-vendoring-dd765be75655`, Jan 2021. Accessed: 2022-08-04. 3.5.3

[264] Tidelift. How to make open source work better for everyone. Technical report, 2018. URL `https://www.sonarsource.com/2018-tidelift-professional-open-source-survey-results.pdf`. 1.1

[265] Amirhosein Toosi, Andrea G Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A brief history of ai: how to prevent another winter (a critical review). *PET clinics*, 16(4):449–469, 2021. 6.3

[266] Parastou Tourani, Bram Adams, and Alexander Serebrenik. Code of conduct in open source projects. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*, pages 24–33. IEEE, 2017. 2.2, 2.3

[267] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2018. 2.1, 2.2, 2.3, 3.6.2, 4.7

[268] Alexandros Tsakpinis. Analyzing maintenance activities of software libraries. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 313–318, 2023. 1

[269] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in github. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*,

April 2, 2026
DRAFT

pages 144–154, 2014. 2.3

[270] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2014. 2.2, 2.3, 4.7, 6.2.2

[271] Uplevel. Ai won't solve your developer productivity problems for you. Technical report, Uplevel, 2024. URL `https://uplevelteam.com/blog/ai-for-developer-productivity`. 6.3

[272] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, 2018. 2.3, 3.2.1, 4.2.2, 4.7

[273] Michael R Veall and Klaus F Zimmermann. Pseudo-r2 measures for some common limited dependent variable models. *Journal of Economic surveys*, 10(3):241–259, 1996. 4.5.1

[274] Dmitry Vinnik. Kindness engineering, June 2019. URL `https://www.danskebank.com/en-uk/ir/Documents/2015/Q4/PresentationQ42015-Press.pdf`. 6.2.2

[275] Georg Von Krogh, Sebastian Spaeth, and Karim Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 2003. 2.3

[276] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022. 5.3.1

[277] Dominik Wermke, Noah Wöhler, Jan H Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. Committed to trust: A qualitative study on security & trust in open source software projects. In *2022 IEEE symposium on Security and Privacy (SP)*, pages 1880–1896. IEEE, 2022. 6.2.2

[278] Dominik Wermke, Jan H Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. "always contribute back": A qualitative study on security challenges of the open source supply chain. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1545–1560. IEEE, 2023. 2.4

[279] Mairieli Wessel, Igor Wiese, Igor Steinmacher, and Marco Aurelio Gerosa. Don't disturb me: Challenges of interacting with software bots on open source software projects. *Proceedings of the ACM on Human-Computer Interaction*, 5(CSCW2):1–21, 2021. 1.2.4, 2.1, 5.1

[280] Wikipedia. Volunteer's dilemma. `https://en.wikipedia.org/wiki/Volunteer's_dilemma`, Jan 2022. Accessed: 2022-09-11. 3.6.3

[281] Laurie Williams, Sivana Hamer, and Nusrat Zahan. Can the rising tide of software supply chain attacks raise all software engineering boats? In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 18–26, 2025. 2.1

[282] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016. 2.1

[283] Ling Wu et al. Transforming code with compositional mappings for API-library switching. In *Conf. Computer Software and Applications*, 2015. 2.1, 4.7

[284] Wenxin Xiao et al. Recommending good first issues in github oss projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2022. 2.3

[285] Liguo Yu, Stephen R Schach, and Kai Chen. Measuring the maintainability of open-source soft-

April 2, 2026
DRAFT

ware. In *Empirical Software Engineering*. IEEE, 2005. 2.3

[286] Nusrat Zahan and Laurie Williams. Prioritizing security practice adoption: Empirical insights on software security outcomes in the npm ecosystem. *arXiv e-prints*, pages arXiv–2504, 2025. 2.1

[287] Nusrat Zahan, Shohanuzzaman Shohan, Dan Harris, and Laurie Williams. Do software security practices yield fewer vulnerabilities? In *2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 292–303. IEEE, 2023. 2.1

[288] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. Leveraging large language models to detect npm malicious packages. *arXiv preprint arXiv:2403.12196*, 2024. 5.4.2

[289] Nusrat Zahan et al. What are weak links in the npm supply chain? In *Proc. Int'l Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2022. 1.1, 2, 2.1, 2.4

[290] Ahmed Zerouali et al. An empirical analysis of technical lag in npm package dependencies. In *Proc. Int'l Conf. Software Reuse (ICSR)*. Springer, 2018. 2.1, 4.3.1, 4.4.1, 4.5.1, 4.7

[291] Minghui Zhou and Audris Mockus. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Trans. Softw. Eng. (TSE)*, 2014. 2.3

[292] Yuming Zhou and Baowen Xu. Predicting the maintainability of open source software using design metrics. *Wuhan University Jrnl. of Natural Sciences*, 2008. 2.3

[293] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 995–1010, 2019. 1.1, 2.4, 5.2.1

April 2, 2026
DRAFT