Supporting the Sustainable Use of Open Source Software

Courtney Elta Miller

CMU-S3D-25-XXX
October 2024

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
Dr. Thomas Zimmerman

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

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant Numbers DGE1745016 and DGE2140739. 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.

Keywords: Open Source Software, Software Supply Chains, Empirical Software Engineering

For my dog, Chanel.

Abstract

In this dissertation, I study how to support and improve the processes used by developers facing open source dependency abandonment, thus promoting and enabling the sustainable use of open source digital infrastructure. 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 is a topic that has been largely neglected by open source sustainability research.

To address this, I redirect the focus from maintainers to users and design a research protocol using a wide variety of empirical methods to address the risks and realities of dependency abandonment that users face on a daily basis. I begin 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 to abandonment, and provide evidence-based strategies from fields like social psychology and game theory to overcome the volunteer's dilemma and encourage collective responses to abandonment. To identify the scale of the dependency abandonment issue 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.

Through these first two steps, I demonstrate the widespread unmet need for tooling to increase information transparency regarding the abandonment of open source dependencies. We also learned that not all dependency abandonment is equally concerning to developers. However, from research on the effectiveness of existing software component analysis (SCA) tools for related dependency management practices e.g., updates and vulnerabilities, we know a key issue 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 of existing dependency management tools in mind, I set out in the third step to develop 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 noteworthy to their project. I use an iterative participatory design approach centered on the downstream user perspective (1) to establish an understanding of the distinctions between dependencies whose abandonment is relevant and noteworthy to users versus those whose abandonment is not based on downstream user usage; and (2) to inform the design of a prototype tool to increase information transparency and support the automated identification of abandoned dependencies without overwhelming users.

Acknowledgments

My advisors are cool. So are my committee members, collaborators, family, and friends.

Contents

List of Figures

Chapter 1

Introduction

In this dissertation, I study how to support and improve the processes used by developers facing open source dependency abandonment, 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 is a topic that has been largely neglected by open source sustainability research. To address this research gap, I redirect the focus from maintainers to users and design a research protocol using a wide variety of empirical methods to address the risks and realities of open source dependency abandonment that downstream users face on a daily basis.

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). 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. 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 [122], and the pervasiveness of open source has only increased since then [146]. Without this digital infrastructure, "*the technology that modern society relies upon simply could not function*." [52]

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 [54].

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 projects operate. Today, most open source projects rely on a small number of over-worked and under-appreciated often volunteer maintainers to do the majority of the work [9, 106], and those maintainers often leave for normal reasons that we cannot prevent e.g., switching jobs, a lack of time, or losing interest [111]. Furthermore, when maintainers do disengage, more often than not, nobody else steps up, and the project becomes fully abandoned [10], making abandonment common even among widely-used projects [115]. This means the ongoing reliability and continued maintenance support of these projects is no sure thing.

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 projects or ecosystems.

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, the users of open source digital infrastructure will always face the risk of dependency abandonment. And since our economy and society, from multibillion-dollar companies to hospitals and startups, relies on open source to function[52], I argue in this thesis that sustainability 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 [53, 54, 79], 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

As a first step at supporting the sustainable use of open source, in this thesis, I use an empirical mixed-methods approach combining user-focused qualitative methods with large-scale data mining and statistical modeling techniques to identify the challenges experienced by developers facing open source dependency abandonment and to demonstrate the effectiveness of information transparency at encouraging user response. Additionally, I leverage an iterative participatory approach to better understand the distinction between dependencies whose abandonment is impactful to downstream users versus those whose abandonment is not based on the context of their usage. I then use this knowledge to inform the design of a tool supporting the automated identification of abandoned dependencies to encourage and support their replacement without overwhelming users. I will now briefly describe the series of projects, including the proposed work, 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 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 to abandonment, 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, whereas others were inconsequential.

Additionally, we learned that one of the most time-consuming parts of dependency abandonment was actually identifying the abandonment itself. 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.

Our findings illustrate that dependency abandonment is an under-supported facet of dependency management. We also learned that there is an unmet need for tooling that supports the automated identification of abandonment and provides resources 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 on designing such tooling must 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 study, we learned that many developers are concerned about dependency abandonment and that some believe in responding to abandonment 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, we performed 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.

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. 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 study, we learned that one of the biggest bottlenecks in the process of dealing with dependency abandonment is identifying the abandonment in the first place. When explicit notices of abandonment

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

are not apparent, users often rely on manual inspections of the dependency's repository for signs of inactivity; however, developers often reported struggling to determine 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.

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 Proposed Work: Identifying the Abandonment that Matters

From the first two studies, we learned (1) that increasing information transparency surrounding abandonment helps support and encourage user response; (2) that there is an unmet need for tooling to support the automated identification of dependency abandonment; and (3) that most developers do not care about the abandonment of all dependencies equally. Additionally, from 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 [73, 116, 151]. Research on overcoming notification fatigue in such circumstances has suggested that only sending relevant notifications to developers can help alleviate the issue [164]. 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 dependencies' abandonment will a developer care about in the context of a particular code base?*'

In this proposed study, we will employ an iterative human-centered participatory approach to design an intelligent prototype tool for supporting automated dependency identification without overwhelming users with notifications they do not care about. More specifically, we will perform a series of formative need-finding interviews (1) to develop a theoretical understanding of which dependencies' abandonment will be impactful to a project based on the context of their usage; and (2) to elicit design requirements and information needs for such a tool in order to support effective usage and downstream response to abandonment using experience prototyping. Then we will develop and operationalize a heuristic based on our theoretical understanding, assess its effectiveness in proactive through user evaluation interviews, and utilize a co-design protocol to develop a user-centric prototype tool design.

1.3 Thesis Statement

By shifting the focus of sustainability research from maintainers to users through the identification of both the challenges developers face when dealing with open source dependency abandonment as well as how they currently react at scale, I will improve the resiliency and adaptability of open source digital infrastructure and the many software supply chains that depend on them.

I highlight the unmet need for tooling to automate the identification of dependency abandonment, demonstrate the efficacy of information transparency in encouraging timely downstream responses, and design user-centric tooling for identifying abandonment by developing a heuristic to identify impactful abandonment based on downstream usage context, thus facilitating the sustainable *use* of open source.

1.4 Contributions

My thesis makes a number of contributions toward supporting 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.

1.4.1 Proposed Contributions

The contributions of the proposed work of this thesis are expected to include the following:

- 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.
- An operationalized heuristic based on the theoretical framework for predicting which of a projects dependencies would be impactful based on the context of their usage allowing for automatic customized abandonment notifications.
- A prototype design for tooling for the automated identification of impactful abandoned dependencies, developed using an iterative participatory design process incorporating developer design requirements and information needs.

Chapter 2

Background and Related Work

Reusing open source frameworks, packages, and other abstractions forms *software supply chains* [5], 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 [15], become incompatible with other dependencies [42, 43], contain security vulnerabilities [44, 75, 86, 100], or even get attacked through supply chain attacks [78, 89, 153, 170].

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 [96, 126], sometimes becoming incompatible with old versions or other dependencies a project may have [15, 77, 128]. 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 [89].

Cross-ecosystem studies of the presence of vulnerable dependencies have highlighted the importance of managing and updating dependencies [132, 170]. Research suggests that generally keeping dependencies up to date correlates with better security outcomes [35]. 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 [4] In short, dependency management is a complex ongoing problem that has been 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 [8, 33, 154, 155], and by creating numerous tools supporting migration [7, 24, 167]. Even so, attempts to support migration thus far have generally supported limited varieties of API evolution, giving them a

limited scope of applicability [26, 49, 124], and limited success in practice [33].

Because keeping up to date with dependency updates can be challenging, research has studied how developers approach and manage dependency updates [11, 40, 42], particularly how they approach versioning and breaking changes [15, 41, 48, 129, 135, 166] and security patches [44, 46, 75, 88, 132], which have been extensively studied, and are considered highly important [123, 144]. Despite common concerns about the continued maintenance of dependencies [52], 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 [12, 44, 45, 46, 88, 93, 109, 129, 132, 132, 137, 152, 171], raising questions about whether abandonment is actually a problem if many projects rely on old versions anyway. For example, Kula et al. [88] 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. [44] 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 [15].

There have been many attempts to improve dependency management practices. Software composition analysis (SCA) tools, such as *dependabot, Sonatype,* and *Snyk*, track dependencies and their updates and alert developers of known vulnerabilities. Studies show that using such SCA tools can improve dependency management outcomes [45, 73, 116]. The adoption of SCA tools is widely seen as a best practice [123], but these tools suffer from many problems, especially high false positive rates and resulting notification fatigue [73, 116, 127, 138]. 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 [46, 129]. Some developers signal proper dependency management by displaying security scores or repository badges [2, 90, 158], which is a topic we will explore further in Section 2.1.1.

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, 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 [113]. 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 both providing detailed information on how users can prepare for and address dependency abandonment in Chapter 3 and 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.

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 [113]. 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.

[1] Exceptions are FOSSA's Risk Intelligence service, currently in beta, and a recent research prototype by Mujahid et al. [119].

2.1.1 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 [37, 38, 104, 108, 133, 158, 160]. This includes complex inferences like choosing which developers to follow [13, 94] or hire [22, 107] and which projects to depend on [15, 118]. 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* [133, 158, 160] and *nudging theory* [73, 104, 116]. 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 [157, 158]. Such nudges can be incorporated in the design of tools, e.g., to accelerate the completion of overdue pull requests [104]. In the CRAN ecosystem, volunteers explicitly coordinate to inform their dependents (within the ecosystem) about breaking changes [15], but this practice is rare otherwise.

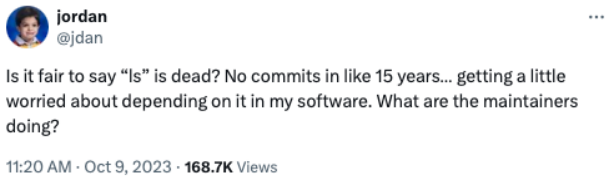


Figure 2.1: Tweet illustrating the frequent difficulty of identifying dependency abandonment.

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.

2.2 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 [52]. In 2018, npm, Inc. estimated that, on average, 97% of the code on modern web applications comes from npm [122]. 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 [39]. 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 [27], 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 [130, 147, 148, 149, 163], the project characteristics associated with greater attractiveness to new

contributors [17, 62, 133], and even the role of social media [56]. Research supporting the onboarding of contributors has studied the onboarding process [36, 51, 80, 163], the role of scaffolding, mentoring, and social ties [55, 71, 81, 150, 160, 168], and the characteristics of contributors who succeeded in becoming part of the core team [64, 159, 172]. Research on retaining core contributors focused on why they disengage [21, 82, 111], the role of maintaining a healthy community to reduce that risk [59, 112, 134], and the impact of disengagement on the health and survival probability of a project [58, 60, 83, 99, 120, 136, 161].

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 [142, 157], how badges can be used as a signal to attract new contributors [158], how project and ecosystem characteristics impact maintainer retention and project activity [29, 70, 161], the maintainability and sustainability of projects [28, 70, 140, 169, 173], and the impact of commercial involvement on open-source development [23].

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 [9, 52], 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 [111]. Maintainers losing interest or no longer having enough time to contribute are two common reasons open-source packages fail [27]. 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 [10].

Therefore, since open source is depended on by *"our economy and society, from multi-million dollar companies to government websites"* [52] 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 [117, 161].

2.2.1 Understanding the Prevalence of Dependency Abandonment

What We Know. Package abandonment is an understudied risk of open source dependencies. Developers worry about abandonment (e.g., online [141]) particularly from a security perspective, since an abandoned package may no longer receive security patches [113, 170, 174]. 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 [113].

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 [144], 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.

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 summarises the work done in our paper '*We Feel Like We're Winging It*': A Study on Navigating Open-Source Dependency Abandonment [113]. 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.

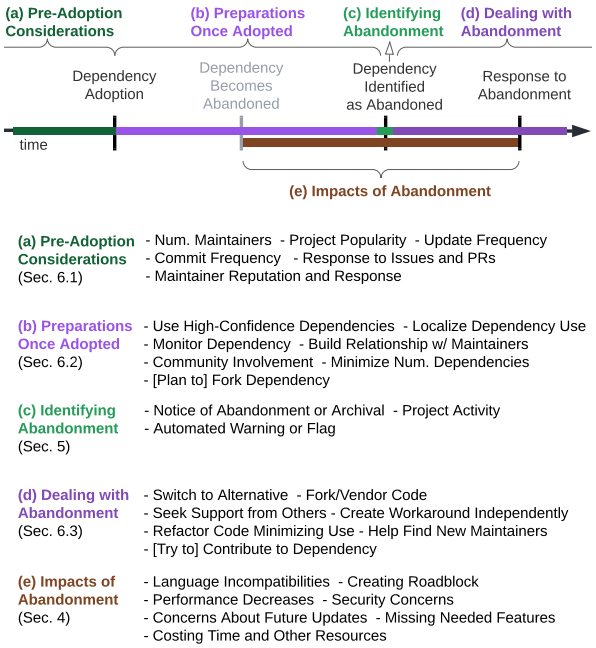


Figure 3.1: Dependency life cycle with the common stages where abandonment is addressed highlighted.

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

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* [47]. 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.

3.2 Research Design

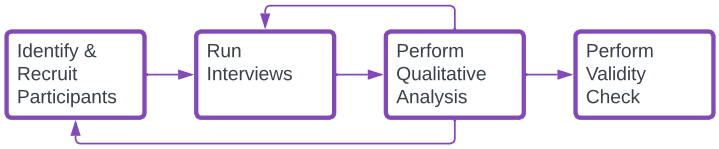


Figure 3.2: Research methodology flow chart.

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 [98]. 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. See paper for details on this participant identification process [113].

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.

3.2.3 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 [61]. We qualitatively analyzed the interview transcripts using iterative thematic analysis [16]. During this process, we were perpetually switching between the stages of exploring the rich transcripts, engaging with and analytically memoing the data [110], coding, searching for themes, and refining the codes and coding framework, as is recommended [98]. See paper for more details on this analysis process.

3.2.4 Validity Check

To validate and check for fit and applicability of our findings as defined by Corbin and Strauss [32], we performed a validity check by sharing our findings and results with interviewees. 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).

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 [113].

3.3 Results

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. For the sake of brevity, we now discuss the solutions used by interviewees to address abandonment and briefly mention key findings for the other stages. For further details on all stages, please refer to the paper [113].

3.3.1 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 align with those discussed in literature about general dependency selection [15, 91, 118, 133]. 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.

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 [15, 91, 118, 133].

3.3.2 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.

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.

3.3.3 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.

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.

3.3.4 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"* [95]. We provide a summary of the types of impacts experienced in Figure 3.1.

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.

3.3.5 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 [156]. 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.

3.4 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.

3.4.1 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 3.3.4) and showed the various, often costly actions developers used to deal with abandonment (Section 3.3.5). 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 3.3.1) 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 3.3.2) can reduce the *reaction effort* post abandonment with some upfront investment in terms of designing an abstraction layer.
- Monitoring the dependency (Section 3.3.2) 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 2.2), 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:

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

3.4.2 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 3.3.3). 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 [131]. 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 3.3.2), 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 [158], 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., 

). 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 (

, 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 [65].

Supporting the Construction of Abstraction Layers (Preparing for Risk of Abandonment). The building and deploying of abstraction layers (Section 3.3.2) 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 3.3.5) 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.

3.4.3 The Volunteer's Dilemma and Reducing Community Effort

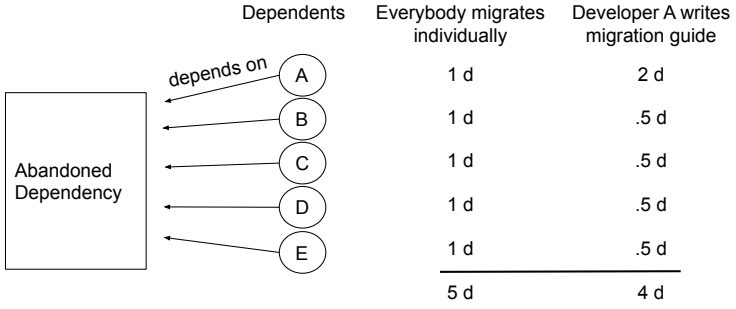


Figure 3.3: Illustration of the volunteers dilemma for dealing with abandoned dependencies: A developer who invests extra effort in writing a migration guide can save all other developers migration effort (measured in days of effort). Writing a migration guide is efficient for the entire community, though 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 3.3.2–3.3.5).

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* [47], which is canonically formalized as a game with a group of 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 [165].

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 [74, 87]. 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 [18] 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 [97]. 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 [105]. 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 vol-

unteers who do not, suggesting that shaming strategies are less effective than positive reinforcement [97]. For example, since many developers are motivated by helping others and supporting their community [63], 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 [67]. 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 [14, 20, 30, 31, 57]. 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 [57]. 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 abandoned 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 3.4.2) 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.

3.5 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.

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 [10, 21, 111] or improving onboarding [55, 66, 149], 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 [113].

In this paper, 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 prevalent, with 15% of widely-used packages becoming abandoned within our six-year observation win-

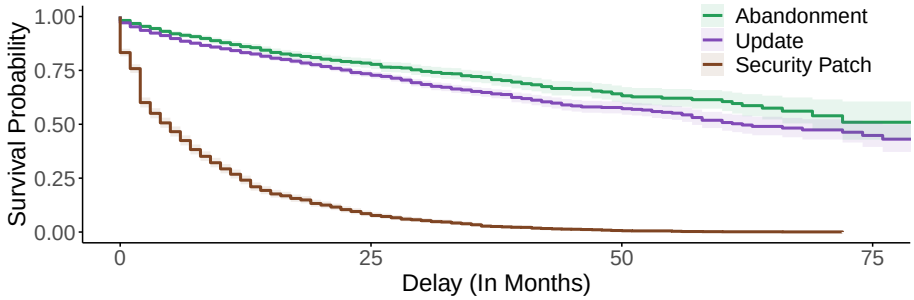


Figure 4.1: Survival probability for event "dependency event is not resolved" w.r.t. the date of event occurrence within dependent project's lifetime.

dow. 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.

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). For details on the operationalization of both abandonment detection heuristics, please see the paper [115].

4.3 RQ1: Abandonment Prevalence and Exposure

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

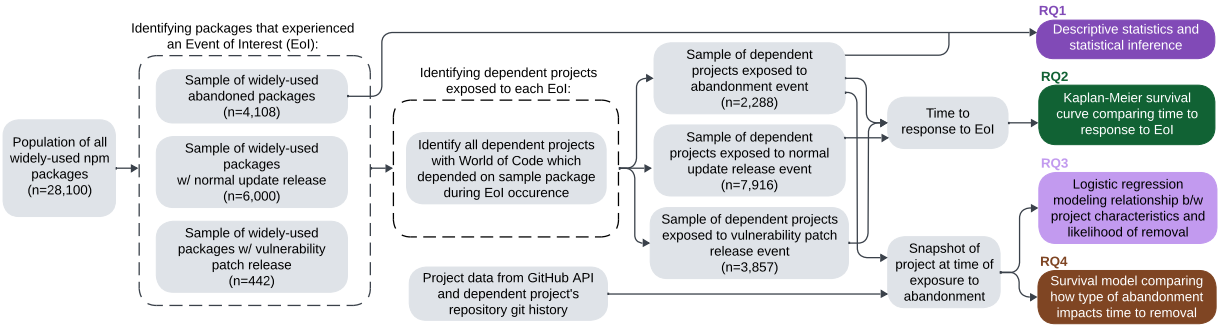


Figure 4.2: Overview of our data collection and analysis.

affects all kinds of users of open source, whether they build popular libraries or applications, or just maintain personal projects. 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 [103]. 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 [1]). We use downloads (rather than reverse dependencies) since they capture both public and private use of packages. After filtering out 940 packages from mono-repositories (i.e., repositories hosting multiple packages) and 7,124 packages that did not have at least 10 total activities by contributors in any year of our observation window, 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 [44, 45, 129, 171], 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. 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 [102, 103]; we use version V, the latest at the time of this analysis.[1] We queried WoC to retrieve all GitHub projects, excluding forks, that ever depended on any of

[1]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.

the abandoned packages in a *package.json* file in their root directory. For more details about WoC and the aforementioned queries, please refer to the paper [115].

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 further details on the limitations of these research methods, please refer to the paper [115].

4.3.2 Results

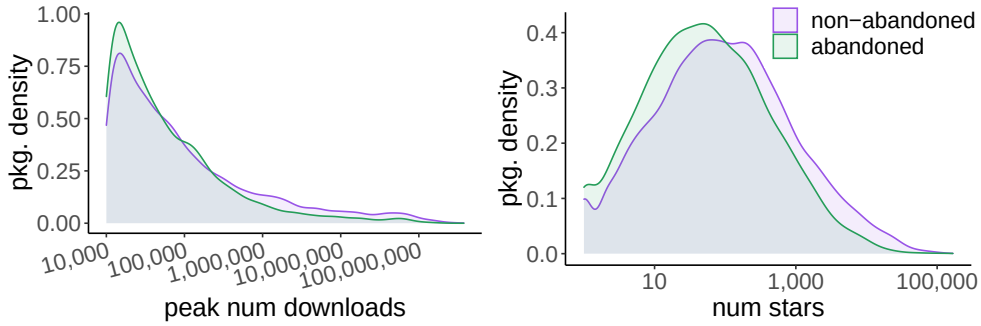


Figure 4.3: The distribution of peak download counts during our observation window and current star counts (March 2024) for both non-abandoned and abandoned widely-used npm packages are similar.

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[2] 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

[2]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.

(average 69 projects exposed per abandoned package). Of those projects, we estimate that $78,023 \pm 624$ GitHub projects had any commits after exposure, i.e., they were not abandoned themselves at the time and might need to respond.

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) [113]. 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 [25, 44, 45, 46, 88, 109, 132, 171].

Removal of Abandoned Dependencies (RQ2a). 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 abandonment event. 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 abandonment 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) and Security Patches (RQ2c). 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 then randomly sample 6,000 of these releases making sure each is from a unique package. As a special version of detecting responses to package updates, we analyze responses to releases that patch known security vulnerabilities using vulnerability data from the OSV database [3]. 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 both these groups of packages of interest, we use the same search strategy with WoC described in Section 4.3 to detect candidate GitHub projects that directly depended on the package of interest ever. We then analyze a large random sample of those candidate dependents for each group ($> 500,000$), using the same process described in Section 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. For further details please refer to the paper [115].

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 [84]. Specifically, we use the Kaplan-Meier estimator [85], which is a common non-parametric statistic for estimating survival functions [34]. For more details on survival modeling and why it was an appropriate choice in this context, please refer to the paper [115].

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.

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 [45, 88], 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.[3]

[3] 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 [114].

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:

- H_1 *Dependency Count*
- H_2 *Dependency Management Practices*
- H_3 *Activity*
- H_4 *Number of Maintainers*
- H_5 *Corporate Involvement*
- H_6 *Governance Maturity*

For details on the theory behind and operationalization of each variable, please refer to the paper [115]. 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. Including checking for influential outliers and highly skewed distributions, heteroscedasticity, and multicollinearity, including relevant control variables, evaluating the model's overall goodness-of-fit we used McFadden's pseudo-R^2 measure [162], and evaluating the statistical significance of the model coefficients at the $\alpha = 0.05$ level. For further details on these steps, please refer to the paper [115].

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. There may also be additional dependent project characteristics and unobserved confounding factors that we did not include in our model meaning our findings should not be considered an exhaustive list. Hence, as usual, care should be taken when generalizing our results beyond the studied measures. For more details, please refer to the paper [115].

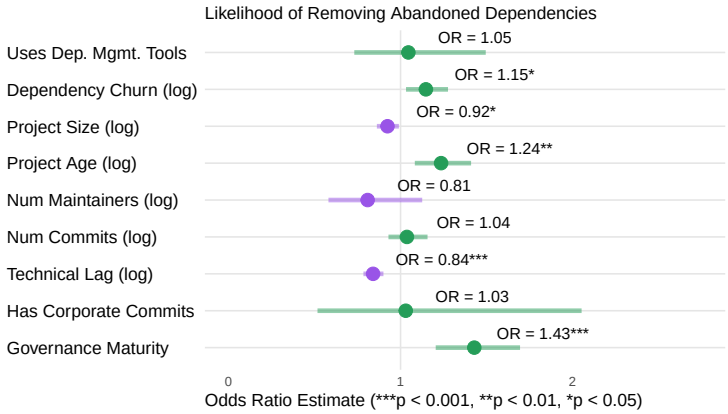


Figure 4.4: Summary of the multivariate logistic regression modeling the likelihood of removing an abandoned dependency within two years post abandonment. Confidence intervals (horizontal lines) for the odds ratios (OR) that do not intersect 1 indicate variables with statistically significant effects.

4.5.2 Model Results

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*.

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 [69] 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

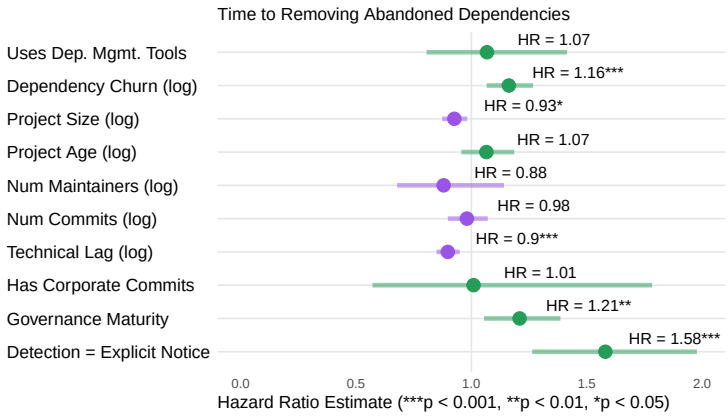


Figure 4.5: Summary of the Cox proportional hazards multivariate survival regression modeling the time to removing an abandoned dependency. Confidence intervals (horizontal lines) for the hazard ratios (HR) that do not intersect 1 indicate variables with statistically significant effects.

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 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 [91, 118] and past studies have shown several statistical predictors for survival [10, 27, 161], 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 [111], 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 [113] 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 [15]). 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 Section 3.4.3).

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 [113], 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 [4]), 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 [12, 25, 44, 45, 46, 88, 93, 109, 129, 132, 132, 137, 144, 152, 171]. 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 [113]. A large number of updates and vulnerability fixes can be captured with floating dependency versions (semantic versioning is a common practice in npm [41, 129], 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 [43, 44, 89]) 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 [15]) 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, 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 [24, 72, 119] or even

generate patches [6, 7, 26, 124, 167]). Building on the vast research on signaling theory [133, 158, 160] and the use of nudging in software engineering [73, 104, 116], **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 [113].

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 [114].

Chapter 5

Proposed Intervention: 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 3.3.3). 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 the identification of abandonment 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 package 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.

In this proposed study, we aim to design an intervention to help improve and streamline [a part of] the process used by developers facing abandonment. As discussed in Section 3.4, 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 proposed study 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.

5.2 Research Design

While there is an unmet need for tooling to support the automated identification of abandoned dependencies, there are many well established software component analysis (SCA) tools for supporting other dependency management tasks, such as dependency updates and security vulnerabilities, including Snyk, Dependabot, and Sonatype. These tools serve as mechanisms to help automate routine dependency management tasks in order to alleviate developer workloads (in theory) e.g., keeping a project's dependencies up-to-date by notifying developers of update opportunities and creating automated pull requests with proposed updates. While these tools can have a positive impact on dependency management practices [73, 116], those effects are tempered by pervasive usability issues [101, 151]. A common issue with such tooling is providing too many notifications to developers, especially ones deemed incorrect, unimportant, or irrelevant. These notifications are often perceived as noise and can distract and annoy developers causing information overload and notification fatigue which can lead to developers ignoring the tool or disengaging altogether [73, 116, 151].

Using this wealth of knowledge on the unexpected collateral effects of existing dependency management SCA tools, in this study, we specifically aim to design an intervention to support the automated identification of abandoned dependencies in a way that will reduce the amount of *effective false positives*. In the context of software analysis tools, traditionally speaking from the tool builder perspective, the term "false positive" refers to incorrect reports produced by the tool. However, since from the user perspective a false positive is any report the user did not want to see, to capture the user's perspective when discussing false positives, Sawdowski et al. coined the term *effective false positives* which they define as "any report from the tool where a user chooses not to take action to resolve the report" [139].

In the context of this study, effective false positives can be considered notifications about the abandonment of dependencies whose abandonment is not noteworthy or impactful to a project given the context of their usage. Since we know from Section 3.3.4 that not all dependency abandonment is equally concerning to developers, instead of creating a "*Abandabot*" tool that opens an issue every time a dependency is abandoned, in this study we will design 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 noteworthy to their project. However, to do so, we first have to understand what differentiates dependencies whose abandonment is impactful, as such we first ask:

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

RQ1b How well can we approximate whether a given dependency's abandonment would be impactful to a project using an operationalized heuristic based on the context of the project's dependency usage?

Furthermore, in the context of notifications for dependency abandonment, in addition to the question of *which* dependence's abandonment will be impactful there is also the question of *when* a given dependency's inactivity becomes noteworthy. I.e., after how long of a period of inactivity does a particular dependency's lack of maintenance become a noteworthy signal to the project, assuming there is no explicit-notice of abandonment the judgement must be made solely based on activity patterns. Although we are unsure whether an attempt to design a singular internally-consistent heuristic for when a project will want to be made aware of a particular dependency's inactivity will be successful given the deeply personal and highly subjective nature of defining activity-based abandonment, we engage with this question as an optimistic exercise that we hope will at least yield some general insights. As such, we as:

RQ2 What factors influence when a project would want to be made aware of a dependency's activity-based abandonment, specifically focused on the context of their dependency usage

Finally, since several studies on designing effective automated software analysis tooling have demonstrated that such tooling should be (1) transparent and provide evidence and data justifying why a judgement-based decision was made; and (2) designed to integrate into existing developer workflows [73, 116, 139, 151], as our final research question we ask:

RQ3 What are the information and design requirements for a prototype tool to automatically identify dependency abandonment?

We plan to explore all three research questions in parallel and we outline the research design we plan to use to answer these questions in the remainder of this section.

5.2.1 Phase 0: Reanalyzing Existing Context

In Chapter 3 we found that not all dependency abandonment is equally concerning to developers, and although we gained some general hypothesis surrounding what types dependencies' abandonment may be particularly concerning (e.g., security-related packages), we lack a sufficiently nuanced understanding of what differentiates packages whose abandonment will be impactful from those whose abandonment is a trivial concern motivating the need to explore this further in Phase 1. Nonetheless, the bespoke examples and general rules-of-thumb shared by some participants in the Chapter 3 interviews revealed some potential ideas for differentiating factors. As a first step to develop preliminary hypothesis for RQ1 and RQ1b, we will revisit these interviews and reanalyze them with the goal of identifying any potential characteristics of dependencies whose abandonment would be impactful discussed.

5.2.2 Phase 1: Need-Finding Interviews

Because we know little about how dependency usage effects whether a dependency's abandonment will be impactful and when, we then perform semi-structured formative need-finding interviews [98], whose protocol has two primary focuses. The first focus of the protocol is attempting to understand which dependencies participants will care about becoming abandoned (and when their inactivity becomes concerning). We plan to explore this specifically though the lens of understanding the impacts and consequences of a particular dependency's abandonment on a project, given the context of their dependency usage in their code base. We will use these insights to inform the design of theoretical heuristics for RQ1 and RQ2 encapsulating these findings in Phase 2.

The second focus of the protocol will be eliciting design requirements and information needs for the prototype tool to support the *automated identification of abandoned dependencies*. For RQ3 we plan to use a *user-centered participatory design process* to iteratively and collaboratively develop a prototype design that is driven by the needs and existing workflows of developers. We chose to use a participatory design process for RQ3 because we wanted to ensure the prototype design was created in a way that would make it an effective and genuinely helpful to developers. Participatory design is a well regarded approach which prioritizes stakeholder engagement throughout the research process and respects participant insights to inspire and help guide the design process [68]. We will work collaboratively with developers at multiple phases of the process to not only understand what their needs and ideas are, but to also iterate on the prototypes together. This second focus of the interview protocol will be the first step in this participatory design process for RQ3, which will inform the prototype tool sketch developed in Phase 2 that we will then elicit another round of user feedback on in Phase 4.

Additionally, we will use a preliminary prototype dashboard tool in the protocol to (1) help contextualize discussions about all their project's dependencies and their current maintenance status, and (2) as a starting point example to help spark more rich discussions about tool design rich and information needs (cf. Figure 5.3).

Interview Participants and Recruiting. The interview participants will be active maintainers of open source projects, who have previously faced or currently face open source dependency abandonment and all interviews will be remote. For the sake of efficiency, we will use the rich data collected during Chapter 4 as the primary data source for identifying abandoned widely-used packages and downstream projects that depend on them. We will generate a large random sample of downstream projects from the population of all dependent projects found across GitHub using World of Code (WoC). Similarly to in Chapter 3, we want to ensure the developers we interview maintain projects with recent activity, so we will employ additional large scale data and repository scraping to collect more recent data to allow for this exclusion criteria.

For recruiting, we will send emails to the maintainers of the projects in our sample that have a publicly-listed email on their GitHub profile and we will offer participants a $20 Amazon gift card as compensation for their time and to incentive participation. We anticipating running between 10-20 interviews, and plan to use a saturation criteria of three consecutive interviews with no major novel insights as grounds for stopping.

5.2.3 Phase 2: Data Analysis and Deriving Heuristic

Preliminary Data Analysis. As a starting point for our analysis of the rich interview data, we will begin by qualitatively analyzing the interview transcripts using iterative thematic analysis [16]. Using iterative thematic analysis in this first step will allow us to distill the hundreds of pages of interview transcripts into a digestible format (codes and a coding framework) that we can subsequently use to develop higher-level theoretical interpretations to answer RQ1, RQ2, and RQ3. During this process, we will switch between the stages of exploring the rich transcripts, engaging with and analytically memoing the data [110], coding, searching for common themes, and refining the codes and coding framework, as is standard recommended practice when engaging in iterative thematic analysis [98].

Deriving Heuristic. Once the initial interview data is analyzed, we will use affinity diagramming [68] to compile a set of rules, derived from the interview data, that we hypothesize can characterize and identify dependencies whose abandonment would be noteworthy and impactful to a project based on the context of their usage. Affinity diagramming will allow us to externalize and meaningfully cluster observations from the iterative thematic analysis into overarching themes to answer RQ1 which are grounded in the experiences of participants [68]. We will also use an analogous process using affinity diagramming to compile a set of factors, derived from the interview data, that we hypothesize influence when a project would want to be made aware of a dependency's activity-based abandonment, thus answering RQ2.

Next, we will use concept mapping [121] to combine the rules together into an internally-consistent theoretical heuristic encapsulating our findings from RQ1 which we will operationalize in Phase 3 and then evaluate in Phase 4 in order to answer RQ1b. Concept mapping is widespread sense-making activity that allows researchers to create a unified visual framework synthesizing and connecting together a large set of rules as they relate to an overarching focus question [68], in our particular context, it serves as an effective strategy to combine together the complex and likely interacting set of rules into a holistic heuristic.

Designing Prototype. As the next step in the participatory design process for RQ2, we will iteratively develop a sketch for a prototype tool based on the design requirements and information needs identified during the interviews in Phase 1 using two different types of prototyping. First we will start by using paper prototyping [143] to create a preliminary prototype design. We use paper prototyping because it allows for the rapid development and testing of ideas through iterative design review, which we will engage in throughout the prototyping process as well as in Phase 4. We will then collaboratively engage

in experience prototyping [19], which is where a group of people (in this case, researchers) engage with the prototype in a simulated realistic context of use. We will use experience prototyping because it will allow us to try things out and gain critical feedback on realistic scenarios, which we can then use to further improve the prototype by identifying any obvious or low-hanging issues before eliciting user feedback in Phase 4.

5.2.4 Phase 3: Operationalizing Heuristic

Once we develop a theoretical heuristic to predict whether a given dependency's abandonment will be noteworthy to a project based on the context of their usage for RQ1b, we will operationalize the heuristic. The goal of this operationalization is that we will eventually be able to input the *package.json* file for an arbitrary npm project into the operationalized heuristic, which will then output a list representing the subset of dependencies whose abandonment we predict will be noteworthy and impactful to the project based on an analysis of the project's code base.

Since the heuristic itself will depend on the findings from Phase 1, we do not yet know exactly how we will operationalize the heuristic. However, based on what we learned in Chapter 3 and related work on identifying impactful dependency vulnerabilities [92] we have some ideas. Those ideas include analyzing the dependency use in the code base– specifically what types of functionalities the API calls utilize [76], the type of dependency usage (e.g., development vs user-facing) [145], and whether the dependency is used in production [92]. Additionally, we predict some more code analysis [50, 125] and LLM usage could also end up being incorporated as well. We predict that the final classification of each dependency using the operationalized heuristic will potentially use either a linear model (with the predicted response being whether this dependency's abandonment would be noteworthy) or some sort of point-based threshold system (e.g., we predict the abandonment of dependencies with X points or higher will be noteworthy, with points being attributed based to various operationalized rules of the heuristic).

5.2.5 Phase 4: Evaluate Heuristic and Elicit Prototype Design Feedback

Finally we plan to (1) evaluate the effectiveness and perceived usefulness of the operationalized heuristic for RQ1b; and (2) elicit prototype design feedback which we will use to engage in a final round of revisions to the prototype design which we then present as the results for RQ3, using user evaluations. Note the results for RQ3 will also be accompanied by a qualitative summary and interpretation of the design requirements and information needs identified through the participatory design process. We intentionally leave the user evaluation plans vague because the actual outputs of RQ1b and RQ3 that we will be evaluating and eliciting feedback on will vary depending on the findings from Phase 1, which will in turn affect what an effective strategy for evaluation will be. We expect to use either interviews or surveys for these user evaluations. In terms of participants we plan to use the same type of participants used in Phase 1 (i.e., open source maintainers who have experience with dependency abandonment), however we do not plan to speak to the same developers again due to both concerns that repeat participants could be biased toward giving positive answers and because we want to expand the pool of developers we gauge feedback from.

We consider interviews for the evaluation of the heuristic instead of something like a user study for two key reasons. First, because the subject matter and question at hand does not lend themselves naturally to a user evaluation design. User studies typically require a discrete task to be completed, a control group, and a metric to be evaluated. The goal of the heuristic evaluation is to determine how accurately the heuristic could predict for a given project which of its dependencies' abandonment would be impactful and noteworthy. Since what we are trying to evaluate is how well our predictions align with the hypothetical, subjective decisions of users, there is no discrete task for users to perform that we could then calculate

a performance metric for and compare to a control group. There is also no experimental condition to stratify on to get a control group. The second reason is because the user evaluations will also serve as an opportunity to elicit design feedback for the prototype tool for RQ3, and within the context of that goal, interviews would be an effective methodology because they provide opportunity for more in-depth feedback and collaborative brainstorming.

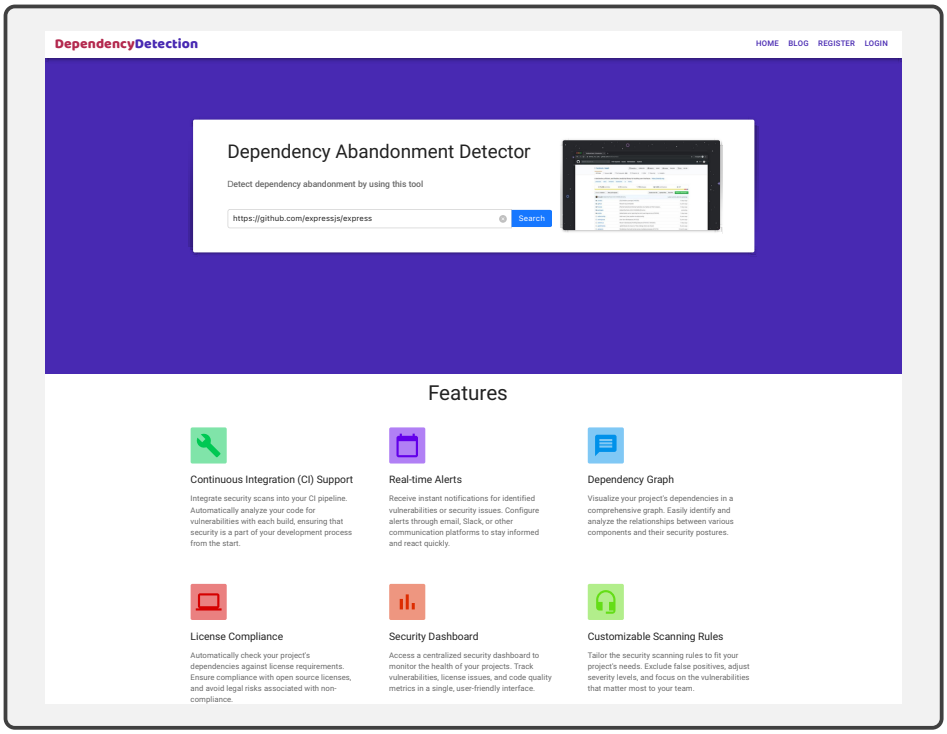
5.3 Preliminary Work

Now, we will briefly summarise the preliminary work that has been completed thus far.

Phase 0: Reanalyzing Existing Context. After reanalyzing all 33 interviews from Chapter 3, we identified 13 where participants discussed at least one characteristic of dependencies whose abandonment they are [or are not] particularly concerned about. Anecdotally, we learned that some developers care about the abandonment of dependencies that are runtime requirements or used in production environments and that some developers do not care about the abandonment of dependencies used for testing and development.

Phase 1: Need Finding Interviews. We have designed the interview guide, recruitment materials, and received IRB approval for the need-finding interviews as well as the Phase 4 evaluation interviews or surveys. We have designed and built the preliminary prototype dashboard tool for the interview study. In Figure 5.3 we show the dashboard landing page and what the project homepage for an arbitrary project. We have also completed the data collection necessary to identify the pool of potential interview participants, and we have started sending out invitations in batches of 10-20, with around 100 being sent so far. The interviews are underway, with eight being completed thus far.

Although we are not done running interviews and analyzing the interview data, we have begun the process of qualitatively analyzing the interview transcripts using iterative thematic analysis. One early insight is that several interviewees considered the abandonment of dependencies involved with data processing or data access to be particularly noteworthy. Interestingly, several interviewees also considered the abandonment of dependencies that are critical components of their development pipeline to be noteworthy and impactful to their project, which contradicts one of the hypotheses developed in Phase 0. In terms of prototype tool information needs, one request made by almost every interviewee thus far has been including resources about potential viable alternative packages when a dependency is identified as abandoned. Several interviewees also suggested including email notifications about dependencies, but only for noteworthy dependencies. However, one interviewee expressed an interest in email notifications for all abandoned dependencies, suggesting a potential opportunity to combine intelligent predictions with lightweight flexible customizations to better meet unique developer needs and preferences.



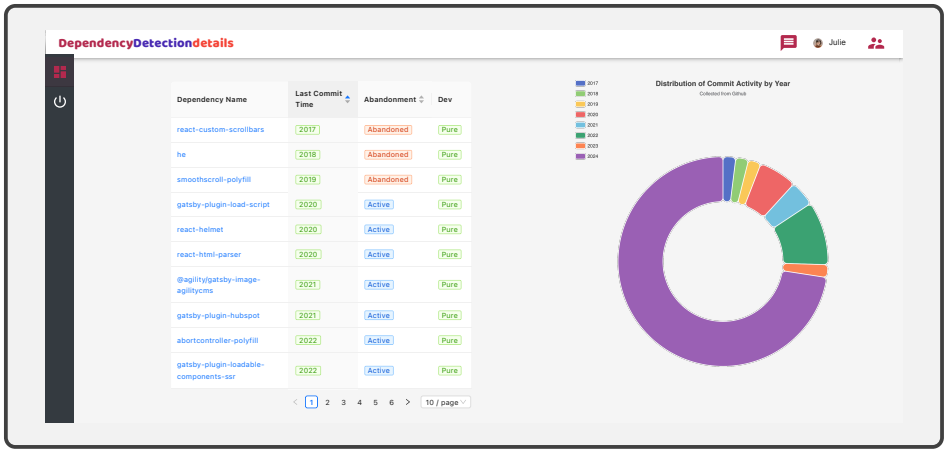


Figure 5.1: Screenshots of preliminary prototype dashboard tool for need-finding interviews. Landing Page (top) and Project Homepage (bottom).

Chapter 6

Proposed Timeline

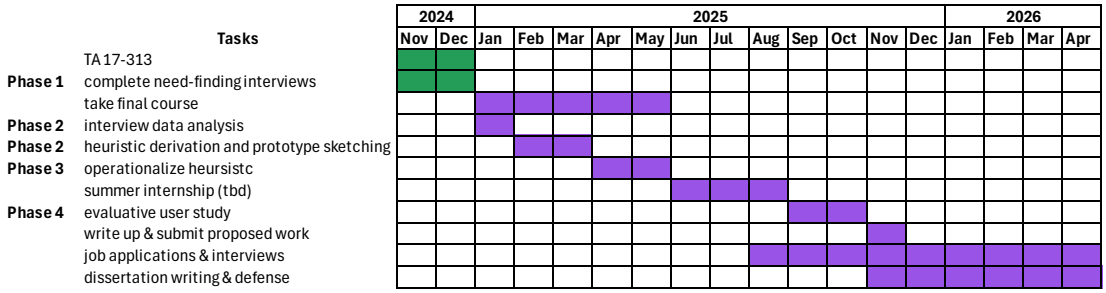


Figure 6.1: Proposed timeline. Green indicates tasks in progress and purple indicates tasks that are planned.

There is one final study left to be completed in this dissertation: Intelligent Dependency Abandonment Notification Tool Development (Chapter 5). The final study consists of four phases: Need-Finding Interviews, Data Analysis and Heuristic Development, Heuristic Operationalization, and Evaluative User Study. Phase 1 Need-Finding Interviews is already underway and I estimate will take two months to complete and I will be completing my final TA requirement at the same time. Phase 2 I broke up into two key tasks, the first being preliminary interview data analysis which will occur in-part simultaneously with Phase 1, but I expect will take an additional month to complete. Then the second task of Phase 2 deriving the heuristic and designing prototype sketch I estimate will take two months to complete. Phase 3 operationalizing the heuristic I conservatively estimate will take two months, although depending on the complexity of the operationalization process and the amount of trial and error necessary, I predict worst-case it might take more like three months. I may do an internship during Summer 2025 (I'm still in the application process). Then Phase 4 the evaluation I estimate will take two months including recruitment and protocol finalization. I then reserve a month for writing up and publishing the proposed work at a conference. Finally in Fall/Winter 2025 I plan to be on the job market. Then I give myself a reasonable buffer for paper revisions, dissertation writing, and presentation preparation which will finally culminate in a thesis defense in April 2026.

Bibliography

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

[2] Openssf scorecard. `https://scorecard.dev`. Accessed: 2024-03-17. 2.1

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

[4] 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

[5] 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

[6] 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

[7] 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

[8] 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

[9] 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, 2.2

[10] 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, 2.2, 4.1, 4.7

[11] 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

[12] 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

[13] Kelly Blincoe et al. Understanding the popular users: Following, affiliation influence and leadership on GitHub. *Information and Software Technology (IST)*, 2016. 2.1.1

[14] 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. 3.4.3

[15] 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, 2.1.1, 3.3.1, 4.7

[16] Virginia Braun and Victoria Clarke. Using thematic analysis in psychology. *Qualitative research in psychology*, 3(2):77–101, 2006. 3.2.3, 5.2.3

[17] 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. 2.2

[18] Chris Brown and Chris Parnin. Sorry to bother you: Designing bots for effective recommendations. In *Int'l Workshop on Bots in Software Engineering*, 2019. 3.4.3

[19] 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. 5.2.3

[20] Anthony Burton and Martin Sefton. Risk, pre-play communication and equilibrium. *Games and economic behavior*, 46(1):23–40, 2004. 3.4.3

[21] Fabio Calefato et al. Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub. *Empirical Software Engineering*, 2022. 2.2, 4.1

[22] Andrea Capiluppi, Alexander Serebrenik, and Leif Singer. Assessing technical candidates on the social web. *IEEE Software*, 2012. 2.1.1

[23] 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. 2.2

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

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

[26] 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

[27] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*, 2017. 2.2, 4.7

[28] 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. 2.2

[29] 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. 2.2

[30] 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. 3.4.3

[31] Russell Cooper, Douglas V DeJong, Robert Forsythe, and Thomas W Ross. Communication in coordination games. *The Quarterly Jrnl. of Econ.*, 1992. 3.4.3

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

[33] 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

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

[35] 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

[36] 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. 2.2

[37] 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. 2.1.1

[38] Laura Dabbish et al. Leveraging transparency. *IEEE Software*, 30, 2012. 2.1.1

[39] Carlo Daffara. Estimating the economic contribution of open source software to the european economy. In *Proc. Openforum Academy Conf.*, 2012. 2.2

[40] 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

[41] Alexandre Decan and Tom Mens. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering*, 47(6):1226–1240, 2019. 2.1, 4.7

[42] 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

[43] 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

[44] 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

[45] 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

[46] 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

[47] Andreas Diekmann. Volunteer's dilemma. *Journal of Conflict Resolution*, 1985. 3.1, 3.4.3

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

[49] 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

[50] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. Bloat beneath python's scales: A fine-grained inter-project dependency analysis. *Proceedings of the ACM on Software Engineering*, 1(FSE):2584–2607, 2024. 5.2.4

[51] 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. 2.2

[52] Nadia Eghbal. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016. 1.1, 2.1, 2.2

[53] 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

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

[55] 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. 2.2, 4.1

[56] 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. 2.2

[57] 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. 3.4.3

[58] 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. 2.2

[59] 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. 2.2

[60] 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. 2.2

[61] 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

[62] 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. 2.2

[63] 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. 3.4.3

[64] 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. 2.2

[65] Sean P Goggins, Matt Germonprez, and Kevin Lumbard. Making open source project health transparent. *Computer*, 54(8):104–111, 2021. 3.4.2

[66] 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

[67] 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. 3.4.3

[68] 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. 5.2.2, 5.2.3

[69] 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

[70] 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. 2.2

[71] 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. 2.2

[72] 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

[73] 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, 2.1.1, 4.7, 5.2

[74] A Healy and J Pate. Asymmetry and incomplete information in an experimental volunteer's dilemma. In *Int'l Congress on Modelling and Simulation*, 2009. 3.4.3

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

[76] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering*, 27(5):102, 2022. 5.2.4

[77] 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

[78] 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

[79] Rich Hickey. Open source is not about you. `https://gist.github.com/richhickey/`

1563cddea1002958f96e7ba9519972d9, Nov 2018. Accessed: 2022-07-06. 1.1

[80] 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. 2.2

[81] 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. 2.2

[82] 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. 2.2

[83] 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. 2.2

[84] 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

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

[86] 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

[87] 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. 3.4.3

[88] 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

[89] 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. 2, 2.1, 4.7

[90] 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

[91] Enrique Larios Vargas et al. Selecting third-party libraries: The practitioners' perspective. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 2020. 3.3.1, 4.7

[92] 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. 5.2.4

[93] 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

[94] Michael J Lee et al. GitHub developers use rockstars to overcome overflow of news. In *Extd. Abstracts on Human Factors in Computing Systems*. 2013. 2.1.1

[95] Manny M Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124. Springer, 1996. 3.3.4

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

[97] Shmuel Leshem and Avraham Tabbach. Solving the volunteer's dilemma: The efficiency of rewards versus punishments. *American Law and Econ. Rev.*, 2016. 3.4.3

[98] Sarah Lewis. Qualitative inquiry and research design: Choosing among five approaches. *Health promotion practice*, 16(4):473–475, 2015. 3.2, 3.2.3, 5.2.2, 5.2.3

[99] 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. 2.2

[100] 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

[101] Dongyu Liu, Micah J Smith, and Kalyan Veeramachaneni. Understanding user-bot interactions for small-scale automation in open-source development. In *Extended abstracts of the 2020 CHI conference on human factors in computing systems*, pages 1–8, 2020. 5.2

[102] 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

[103] 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

[104] Chandra Maddila et al. Nudge: Accelerating overdue pull requests toward completion. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2023. 2.1.1, 4.7

[105] Shakun D Mago and Jennifer Pate. Greed and fear: Competitive and charitable priming in a threshold volunteer's dilemma. *Economic Inquiry*, 2022. 3.4.3

[106] 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

[107] Jennifer Marlow and Laura Dabbish. Activity traces and signals in software developer recruitment and hiring. In *Proc. Conf. Computer Supported Cooperative Work (CSCW)*, 2013. 2.1.1

[108] 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. 2.1.1

[109] 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

[110] Matthew B Miles, A Michael Huberman, and Johnny Saldana. *Fundamentals of Qualitative Data Analysis*. Sage Los Angeles, CA, 2014. 3.2.3, 5.2.3

[111] 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, 2.2, 4.1, 4.7

[112] 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. 2.2

[113] 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, 2.2.1, 3.1, 3.2.1, 3.2.5, 3.3, 4.1, 4.4, 4.7

[114] 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`. 3, 4.9

[115] 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, 4.2, 4.3.1, 4.4.1, 4.5.1

[116] 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, 2.1.1, 4.7, 5.2

[117] 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. 2.2

[118] 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. 2.1.1, 3.3.1, 4.7

[119] 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

[120] 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. 2.2

[121] Joseph D Novak and Alberto J Cañas. The theory underlying concept maps and how to construct and use them. 2008. 5.2.3

[122] 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, 2.2

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

[124] 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

[125] Ranindya Paramitha and Fabio Massacci. Technical leverage analysis in the python ecosystem. *Empirical Software Engineering*, 28(6):139, 2023. 5.2.4

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

[127] 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

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

[129] 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

[130] 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. 2.2

[131] Gauthami Polasani. Announcing the private beta of fossa risk intelligence. `https://fossa.com/blog/announcing-private-beta-risk-intelligence/`, Jul 2022. 3.4.2

[132] 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

[133] 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. 2.1.1, 2.2, 3.3.1, 4.7

[134] 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. 2.2

[135] 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

[136] 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. 2.2

[137] 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

[138] 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

[139] 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. 5.2

[140] Stephen R. Schach, Bo Jin, David R. Wright, Gillian Z. Heller, and A. Jefferson Offutt. Maintainability of the linux kernel. *IEE Proceedings-Software*, 2002. 2.2

[141] shelly. `https://twitter.com/codebytere/status/1567437988908392455`, Sep 2022. Accessed: 2024-03-17. 2.2.1

[142] 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. 2.2

[143] Carolyn Snyder. *Paper prototyping: The fast and easy way to design and refine user interfaces*. Morgan Kaufmann, 2003. 5.2.3

[144] 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, 2.2.1, 4.7

[145] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. Automatic specialization of third-party java dependencies. *IEEE Trans. Softw. Eng. (TSE)*, 2023. 5.2.4

[146] 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

[147] 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. 2.2

[148] 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. 2.2

[149] 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. 2.2, 4.1

[150] 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. 2.2

[151] 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, 5.2

[152] 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

[153] Synopsys. 2024 open source security and risk analysis report. Technical report, Synopsys, 2024. URL https://www.synopsys.com/software-integrity/engage/ossra/ossra-report. 2

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

[155] 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

[156] Martin Thoma. Dependency vendoring. https://medium.com/plain-and-simple/dependency-vendoring-dd765be75655, Jan 2021. Accessed: 2022-08-04. 3.3.5

[157] 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. 2.1.1, 2.2

[158] 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, 2.1.1, 2.2, 3.4.2, 4.7

[159] 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)*, pages 144–154, 2014. 2.2

[160] 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. 2.1.1, 2.2, 4.7

[161] 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. 2.2, 4.7

[162] 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

[163] Georg Von Krogh, Sebastian Spaeth, and Karim Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 2003. 2.2

[164] 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

[165] Wikipedia. Volunteer's dilemma. `https://en.wikipedia.org/wiki/Volunteer's_dilemma`, Jan 2022. Accessed: 2022-09-11. 3.4.3

[166] 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

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

[168] Wenxin Xiao et al. Recommending good first issues in github oss projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*, 2022. 2.2

[169] Liguo Yu, Stephen R Schach, and Kai Chen. Measuring the maintainability of open-source software. In *Empirical Software Engineering*. IEEE, 2005. 2.2

[170] 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. 2, 2.1, 2.2.1

[171] 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, 4.7

[172] Minghui Zhou and Audris Mockus. Who will stay in the floss community? modeling participant's initial behavior. *IEEE Trans. Softw. Eng. (TSE)*, 2014. 2.2

[173] Yuming Zhou and Baowen Xu. Predicting the maintainability of open source software using design metrics. *Wuhan University Jrnl. of Natural Sciences*, 2008. 2.2

[174] 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. 2.2.1