“We Feel Like We’re Winging It:”
A Study on Navigating Open-Source Dependency Abandonment

Courtney Miller
courtneymiller@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

Christian Kästner
Carnegie Mellon University
Pittsburgh, PA, USA

Bogdan Vasilescu
vasilescu@cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

ABSTRACT

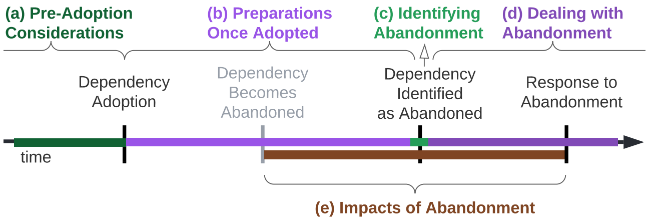
While lots of research has explored how to prevent maintainers from abandoning the open-source projects that serve as our digital infrastructure, there are very few insights on addressing abandonment when it occurs. We argue open-source sustainability research must expand its focus beyond trying to keep particular projects alive, to also cover the sustainable *use* of open source by supporting users when they face potential or actual abandonment. We interviewed 33 developers who have experienced open-source dependency abandonment. Often, they used multiple strategies to cope with abandonment, for example, first reaching out to the community to find potential alternatives, then switching to a community-accepted alternative if one exists. We found many developers felt they had little to no support or guidance when facing abandonment, leaving them to figure out what to do through a trial-and-error process on their own. Abandonment introduces cost for otherwise seemingly free dependencies, but users can decide whether and how to prepare for abandonment through a number of different strategies, such as dependency monitoring, building abstraction layers, and community involvement. In many cases, community members can invest in resources that help others facing the same abandoned dependency, but often do not because of the many other competing demands on their time – a form of the *volunteer’s dilemma*. We discuss cost reduction strategies and ideas to overcome this volunteer’s dilemma. Our findings can be used directly by open-source users seeking resources on dealing with dependency abandonment, or by researchers to motivate future work supporting the sustainable *use* of open source.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software**; **Open source model**; **Software evolution**.

KEYWORDS

Open Source Sustainability, Dependency Management, Human Factors in Software Engineering


(a) Pre-Adoption Considerations
(b) Preparations Once Adopted
(c) Identifying Abandonment
(d) Dealing with Abandonment
Dependency Adoption
Dependency Becomes Abandoned
Dependency Identified as Abandoned
Response to Abandonment
time
(e) Impacts of Abandonment

(a) Pre-Adoption Considerations (Sec. 6.1)	- Num. Maintainers - Project Popularity - Update Frequency - Commit Frequency - Response to Issues and PRs - Maintainer Reputation and Response
(b) Preparations Once Adopted (Sec. 6.2)	- Use High-Confidence Dependencies - Localize Dependency Use - Monitor Dependency - Build Relationship w/ Maintainers - Community Involvement - Minimize Num. Dependencies - [Plan to] Fork Dependency
(c) Identifying Abandonment (Sec. 5)	- Notice of Abandonment or Archival - Project Activity - Automated Warning or Flag
(d) Dealing with Abandonment (Sec. 6.3)	- Switch to Alternative - Fork/Vendor Code - Seek Support from Others - Create Workaround Independently - Refactor Code Minimizing Use - Help Find New Maintainers - [Try to] Contribute to Dependency
(e) Impacts of Abandonment (Sec. 4)	- Language Incompatibilities - Creating Roadblock - Performance Decreases - Security Concerns - Concerns About Future Updates - Missing Needed Features - Costing Time and Other Resources

Figure 1: Dependency life cycle with the common stages where dependency abandonment is addressed highlighted.

ACM Reference Format:
Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. “We Feel Like We’re Winging It:” A Study on Navigating Open-Source Dependency Abandonment. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE ’23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616293

1 INTRODUCTION

Open-source digital infrastructure is heavily relied upon by billion-dollar corporations, governments, startups, hobbyists, and pretty much everyone else who builds software [33]. However, despite the broad reliance on open source, the reliability and continued maintenance of many of these projects is no sure thing, especially since much of the creation and maintenance effort comes from volunteer maintainers who may stop contributing and disengage from the project at any point [33, 39]. When open source maintainers disengage, more often than not, nobody else steps up and the project becomes abandoned [4]. This tension between the reliance on open source and the uncertainty of future maintenance has fueled the need to study, and improve, open-source sustainability.

In general, open-source sustainability research has so far focused on keeping particular projects and ecosystems alive, i.e., maintained,

ESEC/FSE ’23, December 3–9, 2023, San Francisco, CA, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0327-0/23/12.
https://doi.org/10.1145/3611643.3616293

e.g., by improving onboarding processes [34, 49, 95], finding sustainable funding models [92, 112], or preventing maintainer disengagement [4, 13, 73]. Yet, maintainers often leave projects for arguably sensible reasons [73], such as changing jobs, losing interest, or starting a family. As such, there will *always* be a risk for users of open-source infrastructure that some direct or indirect dependencies become abandoned. Therefore, we argue that more time, attention, and effort should be invested into supporting the users of open source who face dependency abandonment. For example, as we will show, there are many ways that developers can prepare for abandonment either individually or collectively, and many strategies that can help reduce reaction costs when abandonment occurs. In this paper, 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?

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 dependency abandonment: before adoption, while using a dependency that is still being maintained, and after a dependency has become abandoned (see Figure 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.

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* [30]. We survey solutions to the volunteer's dilemma from fields like social psychology and game theory, and discuss how they can be applied to this context.

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

2 RELATED WORK

Dependency Management. 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 [63, 81], sometimes becoming incompatible with old versions or other dependencies a project may have [8, 54, 82]. 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 [61]. Cross-ecosystem studies of the presence of vulnerable dependencies have highlighted the importance of managing and updating dependencies [85, 111]. 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.[1] 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 [2, 23, 98, 99], and by creating numerous tools supporting migration [3, 15, 108]. Even so, attempts to support migration thus far have generally supported limited varieties of API evolution, giving them a limited scope of applicability [16, 31, 80], and limited success in practice [23].

Because keeping up to date with dependency updates can be challenging, research has studied how developers approach and manage dependency updates [6, 26, 27]. Despite common concerns about the continued maintenance of dependencies [33], developers tend to either be slow about updating dependencies or not update them at all [28, 29, 89], raising questions about whether abandonment is actually a problem if many projects rely on old versions anyway. A study of 4,600 GitHub projects found that developers tend not to update dependencies even when security vulnerabilities are involved, with 81.5% of projects having outdated dependencies [60]. In addition to studying how updates are managed at large,

[1] https://whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/

particular focus has been directed towards studying how breaking changes are dealt with [8]. However, 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. Because dependency abandonment can be a costly and complex issue for dependents [33], this study provides detailed information on how dependents can prepare for and address dependency abandonment.

Open-Source Sustainability. 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 [33]. In 2018, npm, Inc. estimated that, on average, 97% of the code on modern web applications comes from npm [79]. 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 [25]. Nonetheless, despite the widespread reliance on open source, the reliability and continued maintenance of many of these projects 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 [17], 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 [83, 94–96, 106], the project characteristics associated with greater attractiveness to new contributors [10, 42, 87], and even the role of social media [35]. Research supporting the onboarding of contributors has studied the onboarding process [24, 32, 55, 106], the role of scaffolding, mentoring, and social ties [34, 52, 56, 97, 103, 109], and the characteristics of contributors who succeeded in becoming part of the core team [44, 104, 113]. Research on retaining core contributors focused on why they disengage [13, 57, 73], the role of maintaining a healthy community to reduce that risk [38, 72, 86], and the impact of disengagement on the health and survival probability of a project [37, 40, 58, 67, 76, 88, 105].

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 [93, 101], how badges can be used as a signal to attract new contributors [102], how project and ecosystem characteristics impact maintainer retention and project activity [19, 51, 105], the maintainability and sustainability of projects [18, 51, 91, 110, 114], and the impact of commercial involvement on open-source development [14].

Taking a step back, we can observe that almost all sustainability research focuses on studying various factors, characteristics, and phenomena that support the goal of *keeping particular projects or ecosystems alive and actively maintained.* 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 projects hosted on GitHub rely on one or two core maintainers who are often volunteers to keep the project running [5, 33], 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 [73]. Maintainers losing interest or no longer having enough time to contribute are two common reasons open-source projects fail [17]. One study of popular projects on GitHub found that 16% were abandoned by maintainers, and in 59% of those abandoned projects, nobody stepped up to take over maintenance efforts leaving the project fully abandoned [4]. Therefore, since open source is depended on by *"our economy and society, from multi-million dollar companies to government websites"* [33] 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 we pursue in this paper, 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 [75, 105].

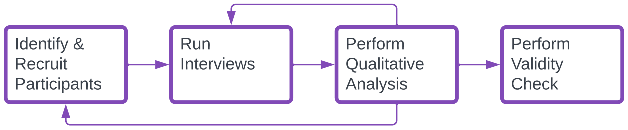


Figure 2: Research Methodology Flow Chart

3 RESEARCH DESIGN

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 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 [66]. We will now discuss study design, analysis, and limitations.

3.1 Identifying and Recruiting Participants

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

Defining and Identifying Abandoned Projects. Because customs and behaviors surrounding dependency management can vary widely by ecosystem [8], we searched for abandoned projects in three package manager ecosystems to collect a diverse pool of experiences: *npm*[2] (Javascript), *PyPi*[3] (Python), and *Composer*[4] (PHP). Using data cross-linked between GHTorrent [47] and each ecosystem's respective package manager website (matching packages to their corresponding GitHub repositories, when mentioned explicitly in the package manager metadata), we heuristically searched for projects with signs of abandonment. Concretely, we identified all projects with at least ten commits a month for two consecutive years and less than three commits total in the following year, i.e., the year in which the project is presumed abandoned; the three-commits threshold allows for some residual activity (e.g., posting warnings about abandonment in the README file) and mirrors prior work [73, 105]. Once we had a pool of potentially abandoned projects, we randomly sampled and manually evaluated whether each project seemed indeed abandoned by investigating the activity patterns on each project's GitHub repository until we had 10-20 high-confidence abandoned projects per ecosystem. For this we looked at the most recent period we could observe at the time – the first six months of 2022, regardless of the year we suspected the project was abandoned based on the automated heuristic – and manually checked if the project either (1) did not have any significant commit activity;[5] or (2) had an explicit label or notice that it was abandoned, archived, or simply no longer maintained.

Identifying Dependent Projects and Maintainers. We then used GitHub's dependency graph feature to get the list of dependents for each abandoned project [45], and collected the data using the github-to-sqlite library.[6] We ensured the dependent projects were active by considering only dependents that had, on average, at least ten commits a month in the first half of 2022. We then identified each dependent project's top maintainers by commit counts during the first half of 2022, collected their publicly available email from their GitHub profiles, and sent out 412 interview invitations in total in staggered batches of 10-20. Our study design was approved by our Institutional Review Board.[7]

3.2 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.

3.3 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 qualitatively analyzed the interview transcripts using iterative thematic analysis [9]. The process followed Lincoln and Guba's trustworthiness criteria [48], as discussed by Nowell et al. [77]. During this process, we were perpetually switching between the stages of exploring the rich transcripts, engaging with and analytically memoing the data [71], coding, searching for themes, and refining the codes and coding framework, as is recommended [66].

The analysis began with the first author performing open-ended inductive coding of each interview as we went. After the first eleven interviews, all the authors came together and performed an in-depth analysis of the codes and coding frame. Iterative adjustments to the coding frame and interview guide were made as necessary. Once a coding frame was settled on, the first author re-coded all the transcripts, with any uncertain cases being reviewed by another author. We stopped running interviews once we reached our saturation criterion, which we defined as three consecutive interviews without learning any new major insights [41]. A later participant discussed a dependency that was marked abandoned but still received security updates, and we explored this further by identifying and interviewing developers who faced this type of dependency abandonment. We quickly reached saturation and did not find any new major insights.

3.4 Validity Check

To validate and check for fit and applicability of our findings as defined by Corbin and Strauss [22], we performed a validity check by sharing our findings and results with interviewees. We confirmed our interpretations of the rich interview data aligned with the interviewees' experiences by getting interviewees' thoughts and feedback. We sent all interviewees summaries and the complete drafts of Secs. 4, 5, 6, and 7. We also sent a list of prompts and questions asking interviewees to look through the documents for areas of agreement or disagreement, general correctness, and any additional insights they gained after reading through the findings as well as the experiences and strategies of other developers.

Six interviewees responded, all six confirmed that they largely agree with our findings, e.g.,*"I think your paper is a well-considered analysis of the subject that fits with my experience, fwiw"* (P11). One interviewee pushed back on augments made by other interviewees suggesting that abandonment was not always a problem because

[2] Node.js Package Manager, https://npmjs.com
[3] The Python Package Index, https://pypi.org
[4] PHP Dependency Manager, https://getcomposer.org
[5] Since abandonment need not align with calendar year boundaries, we still considered as abandoned projects with a few trailing commits at the beginning of the six-month window but no commits thereafter.
[6] https://github.com/dogsheep/github-to-sqlite
[7] We sent a small number of targeted emails, based on information our participants posted *publicly* in their profile. In terms of research ethics, especially the Belmont report's principles of *respect for persons* and *beneficence*, we consider that the costs (e.g., potentially unwanted emails) and risks (e.g., releasing confidential information) to potential participants are low, and insights gained in better dependency management benefit all open source contributors. We considered alternative sampling strategies and concluded that because we were interested in speaking to a specific group of open source maintainers, that it seems unlikely that we could have recruited people in a different (less targeted way) without increasing the general cost to the community by engaging with large groups of maintainers.

there are not always impacts. They argued that *"abandonment is always problematic and always has an impact, even if the software itself is not broken, because abandonment still forces a consumer to act as if it is abandoned, i.e. to prepare for breakage or vulnerabilities"* (P4).

3.5 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. Self-selection bias could influence the transferability of the results because there could be differences in the personalities and beliefs in the sample and the subset that chose to participate [70, 90]. We tried to reduce this risk by streamlining the enrollment process and keeping interviews short. There is also a question of authenticity in how we defined 'abandoned' dependencies since the definition may not fully represent the concept of project abandonment [68], although during the discussions with interviewees there was agreement with our definition.

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"* [64]. We start by exploring if and how abandonment impacted interviewees. We provide a summary of the types of impacts experienced in Figure 1.

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

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

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

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

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

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

Some interviewees reported the abandonment had **no meaningful impact** (P6, 7, 9, 11, 16, 25, 29) and argued that just because a dependency was abandoned does not necessarily mean there is a problem (in contrast to the interviewees that mentioned *anticipated problems*, who were at least concerned about possible future problems). They explained that if the software is complete, does not interact with other software, and does not become insecure itself, then the abandonment is not necessarily problematic. For example, *"it was recognized within the organization that [...] one of the dependencies that the business runs on is totally unsupported for years [...], and because it wasn't a cause of many problems it wasn't necessarily an issue"* (P11).

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

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

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

5 IDENTIFYING ABANDONMENT

It is important to understand how abandonment is identified, because in cases where identification happens after a concrete problem has occurred, immediate action is frequently needed which can be disruptive to projects. Thus many developers want to identify abandonment before it causes a concrete problem, so they can react without immediate time pressures. Interviewees used a wide range of information to identify abandonment. This information varied along two dimensions, first how *visible* the information was, and second *how* the information was discovered. We now catalog the information used to identify abandonment and discuss how it varies across the aforementioned dimensions. We provide a summary of the codes in Figure 1.

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

Many participants identified abandonment by observing a **notice of abandonment/archival** (P3, 4, 7, 13, 17, 20, 29, 30). The notices were often posted somewhere on the abandoned dependency's repository page, but there was a wide variation in visibility depending on the particular location. Sometimes the information was *highly visible*, being posted as a flag/warning at the top of the page, a message at the top of the README, or as a note in an issue tracker thread explicitly discussing the maintenance status of the project. For example *"my colleague saw as he was looking at issues [...] that there was this issue saying 'this will no longer be maintained"'* (P3). The project inspection that led to the discovery of this information often occurred because the interviewee traced an error back to the dependency or because they were using the dependency as a reference when doing something like implementing a new feature. Other times, the information had *low visibility*, meaning it was possible to find but required more effort to locate (e.g., an unrelated issue or PR that a maintainer responded to announcing they no longer plan to maintain the project).

Tool-Supported Identified Information. Some interviewees used information from observing an **automated warning or flag** (P4, 6, 7, 13, 22, 25, 27, 29, 31, 32) which often provided *highly visible* information. Often these warnings occurred because the dependency maintainers had explicitly marked the project as abandoned/deprecated or because the unmaintained dependency was causing some sort of incompatibility error, such as those described in Sec. 4.

Flags for abandoned/deprecated packages are a recent feature of several package managers, allowing maintainers to explicitly signal that a package is abandoned/deprecated. These flags generate warnings when users either install, update, or use said package (with specifics depending on the package manager). In 2015, *Composer* incorporated the ability to add a flag to a package indicating it has been abandoned which is used to generate warnings when users install or update flagged packages.[8] Similarly, 'since 2020 *npm* as had the *npm-deprecate* command, which allows maintainers to add a deprecation flag to a package's *npm* registry entry, producing a deprecation warning whenever someone installs the package [78]. We could not identify an equivalent *PyPi* feature, but found community discussions that proposed creating one and cited the *npm-deprecate* function as an example.[9] GitHub also has an platform-wide *archive* flag for repositories.[10]

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.

6 PREPARING FOR AND ADDRESSING ABANDONMENT

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

6.1 Considerations Before Adoption

When deciding whether to adopt a dependency, interviewees often reported evaluating the current maintenance status and the expected risk of future abandonment by examining project and maintainer characteristics. Essentially all mentioned factors mirror those discussed in literature about general dependency selection [8, 62, 74, 87]. However, we distinguish these considerations from those for general dependency selection because we specifically asked if and how they evaluate the risk of a potential dependency becoming unmaintained or abandoned before adopting it. For the sake of completeness, we present the considerations discussed by interviewees.

Project popularity (P2a, 6, 9, 10, 12, 13, 16, 17, 19, 21, 23-25, 27, 30, 31) was often operationalized by looking at the number of stars, forks, or users. The **update frequency or time of the**

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

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

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

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

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

Key Insights: A project's popularity, activity, and maintainer reputation were often used when considering the risk of a potential dependency becoming abandoned, mirroring factors used in general dependency selection [8, 62, 74, 87].

6.2 Preparations Once Adopted

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

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

Some interviewees prepared by directly **monitoring the dependency** (P2a, 4, 10, 13, 27) to keep an eye on how things are going, often by looking at project characteristics similar to those described in Sec. 5. For example, *"we are always conscious of the dependencies and looking closely at them"* (P2a). By remaining aware of the state of the dependency and its community, interviewees place themselves in a better position to identify early signals of abandonment which gives them an opportunity to act before abandonment and any resulting concrete problems occur, if they so choose. Some also prepared by **being active and informed members of the community** (P2b, 16, 17, 31) and **building relationships with dependency maintainers** (P1), often so they could notice issues earlier or have people to reach out to if abandonment occurs. This often involves at least semi-frequent interactions with dependency maintainers or other community members to stay informed of the goings-on in the project and aware of any potential issues or warning signs of something like abandonment being on the horizon. For example, *"I suppose I engaged pretty actively in the open source community, particularly around Python, so I would hope I would have a feeling for what was going on. I think it's partly about being aware"* (P17).

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

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

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

6.3 Solutions to Abandonment

Once dependency abandonment was identified, nearly all interviewees deployed some sort of solution to deal with abandonment. The most common solution was **switching to a better maintained alternative** (P1-4, 6, 7, 10, 12-14, 16, 17, 20, 23, 27, 29, 31, 32). Interviewees found these alternatives in various ways. Sometimes, an issue or PR on the abandoned project included a discussion recommending an alternative. For example, *"actually, I can see now on the 'is the project dead' issue there's someone saying use [alternative project], which was the alternative that we ended up going to"* (P17). In other cases, interviewees used search engines such as *DuckDuckGo*, forum websites such as *Reddit* or *StackOverflow*, package managers such as *PyPi*, or even specialized open-source library recommendation websites such as *libhunt.com* to find pointers to alternatives. Another interviewee described how an automated warning about an abandoned dependency included a list of alternatives, which was used to select a replacement (P32).

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

Another common solution was to **fork or vendor code** (P1-2b, 4, 5, 7, 10, 12-14, 16, 20, 23, 30, 32) from the abandoned dependency; vendoring means incorporating 3rd party software directly into a code base [100]. 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.

7 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.

7.1 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 (Sec. 4) and showed the various, often costly actions developers used to deal with abandonment (Sec. 6.3). When a dependency becomes abandoned, it shifts at a high level from being a free and easy to use software artifact to a potential liability and source of unexpected disruptions, costs, and concerns. One way to think about the total *anticipated cost of abandonment* is as a product of the probability of abandonment occurring and impacting the dependent project (*impact probability*) and the effort required to react to the abandonment once it happens (*reaction *effort*):

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

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

- Only using high-confidence dependencies and minimizing the number of dependencies (Sec. 6.1) 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 (Sec. 6.2) can reduce the *reaction effort* post abandonment with some upfront investment in terms of designing an abstraction layer.
- Monitoring the dependency (Sec. 6.2) 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 (Sec. 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:

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

7.2 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 Sec. 5). 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 [84]. 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 (Sec. 6.2), 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 [102], such as badges in READMEs, can make it easier to assess the status of a project. One interviewee (P22) explained how their company has started putting badges in their public projects' READMEs showing their intended support status (e.g., support status actively maintained). Such transparency mechanisms can be used to declare maintenance intention (e.g., beta phase, hobby project, actively maintained, commercial support available) but can also be used to automatically summarize information, e.g., the last activity of the maintainer or the typical recent issue response latency. Beyond shield.io's template for a maintained badge (maintained no! (as of 2022)), not widely used), we are not aware of any more advanced transparency mechanisms regarding maintenance status or abandonment risk, although efforts seem underway at least as part of the CHAOSS project [46].

Supporting the Construction of Abstraction Layers (Preparing for Risk of Abandonment). The building and deploying of abstraction layers (Sec. 6.2) 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 Sec. 2), 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 (Sec. 6.3) is a common solution when faced with abandonment, but finding a suitable one can be challenging, as it is not always clear where to look. Also finding actively maintained forks can be difficult in projects with many forks. Making suitable alternatives easier to find can reduce *reaction efforts*. Interviewees mentioned several specific strategies for advertising alternatives: (1) posting pointers to alternatives on the abandoned dependency's repository page (e.g., notes in an issue thread about abandonment); (2) promoting alternatives on relevant online forums (one interviewee (P30) reports creating posts on relevant Subreddits like r/python when they have a new release celebrating it and giving an overview of the project and its features); and (3) creating blog posts discussing alternatives. Platforms could highlight posts for alternatives, curate links to external resources, and highlight active forks. They could also gather a lot of information automatically, for example, by scraping what other projects have migrated to in the past.

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

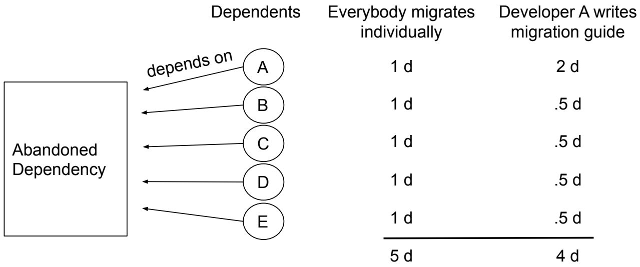
7.3 The Volunteer's Dilemma and Reducing Community Effort

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 (Sec. 6.2–6.3).

We call these investments designed to benefit others *community-oriented solutions*. They reduce the *redundant reaction effort* expended by subsequent projects facing the same abandoned dependency, as we illustrate in Figure 3. 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* [30], 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 [107].

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:



Dependents	Everybody migrates individually	Developer A writes migration guide
A	1 d	2 d
B	1 d	.5 d
C	1 d	.5 d
D	1 d	.5 d
E	1 d	.5 d
	5 d	4 d

Figure 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.

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 [53, 59]. 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 [11] 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 [65]. 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 [69]. This suggests that framing the creation of community-oriented solutions as a deliberate act to benefit the larger open-source community could encourage such creation and normalize it as a common action. Also estimating the possible impact of creating a community-oriented solution could be motivating for some. More research on the attitudes of developers toward various community-oriented actions and how actions for abandoned dependencies fit in could help design a supportive framing.

Rewarding Volunteers. Research studying the effects of rewards and punishments on the volunteer's dilemma found that rewarding volunteers who step up can be more effective than punishing potential volunteers who do not, suggesting that shaming strategies are less effective than positive reinforcement [65]. For example, since many developers are motivated by helping others and supporting their community [43], 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 [50]. 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 [7, 12, 20, 21, 36]. 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 [36]. 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 Sec. 7.2) 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.

8 CONCLUSION

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.

9 DATA AVAILABILITY

The complete interview guide along with a table with anonymized summary statistics for the 33 interview participants are available on Zenodo [1]. DOI 10.5281/zenodo.8102547

ACKNOWLEDGMENTS

Firstly, special thanks and deep gratitude are given to Chanel for continuing her integral work as a brilliant world-class canine researcher and for always being there to support, encourage, and inspire the team. We would like to thank all our interview participants for sharing their time, expertise, and wisdom with us, without them this project would not have been possible. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant Number 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. Kaestner and Vasilescu's work was supported in part by the National Science Foundation (awards 2106853, 2131477 and 2206859) and the Sloan Foundation. This work was also supported in part by a Google Faculty Research Award and a Google Award for Inclusion Research.

REFERENCES

[1] 2023. *Supplementary Material for "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment.* Zenodo. https://doi.org/10.5281/zenodo.8102547
[2] Hussein Alrubaye et al. 2020. How does library migration impact software quality and comprehension? an empirical study. In *Proc. Int'l Conf. Software Reuse (ICSR).* Springer, 245–260.
[3] Hussein Alrubaye, Mohamed Wiem Mkaouer, and Ali Ouni. 2019. 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).*
[4] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. 2019. On the abandonment and survival of open source projects: an empirical investigation. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM).* ACM Press, 1–12.
[5] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *Proc. Int'l Conf. Program Comprehension (ICPC).* IEEE, 1–10.
[6] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The evolution of project inter-dependencies in a software ecosystem: The case of apache. In *Proc. Int'l Conf. Software Maintenance (ICSM).* IEEE, 280–289.
[7] Andreas Blume and Andreas Ortmann. 2007. The effects of costless pre-play communication: Experimental evidence from games with Pareto-ranked equilibria. *Journal of Economic theory* 132, 1 (2007), 274–290.
[8] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE).* 109–120.
[9] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
[10] Scott Brisson, Ehsan Noei, and Kelly Lyons. 2020. We are family: analyzing communication in GitHub software repositories and their forks. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER).*
[11] Chris Brown and Chris Parnin. 2019. Sorry to bother you: Designing bots for effective recommendations. In *Int'l Workshop on Bots in Software Engineering.*
[12] Anthony Burton and Martin Sefton. 2004. Risk, pre-play communication and equilibrium. *Games and economic behavior* 46, 1 (2004), 23–40.
[13] Fabio Calefato, Marco Aurelio Gerosa, Giuseppe Iaffaldano, Filippo Lanubile, and Igor Steinmacher. 2022. Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub. *Empirical Software Engineering* (2022).
[14] Andrea Capiluppi, Klaas-Jan Stol, and Cornelia Boldyreff. 2012. Exploring the role of commercial stakeholders in open source software evolution. In *IFIP Int'l Conf. on Open Source Systems.* Springer, 178–200.
[15] Chunyang Chen. 2020. Similarapi: mining analogical apis for library migration. In *Comp. Int'l Conf. Software Engineering (ICSE).* IEEE, 37–40.
[16] Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *Proc. Int'l Conf. Software Maintenance (ICSM),* Vol. 96. 359.
[17] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE).*

[18] Jailton Coelho, Marco Tulio Valente, Luciano Milen, and Luciana L Silva. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology (IST)* (2020).

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

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

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

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

[23] Bradley E Cossette and Robert J Walker. 2012. 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)*. 1–11.

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

[25] Carlo Daffara. 2012. Estimating the economic contribution of open source software to the European economy. In *Proc. Openforum Academy Conf.*

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

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

[28] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *Proc. Int'l Conf. Software Maintenance and Evolution (ICSME)*. IEEE, 404–414.

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

[30] Andreas Diekmann. 1985. Volunteer's dilemma. *Jrnl of conflict resolution* (1985).

[31] Danny Dig and Ralph Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution: Research and Practice* (2006).

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

[33] Nadia Eghbal. 2016. *Roads and bridges: The unseen labor behind our digital infrastructure.* Ford Foundation.

[34] Fabian Fagerholm, Alejandro S Guinea, Jürgen Münch, and Jay Borenstein. 2014. 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)*. 1–10.

[35] Hongbo Fang, Hemank Lamba, James Herbsleb, and Bogdan Vasilescu. 2022. "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)*.

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

[37] Fabio Ferreira, Luciana Lourdes Silva, and Marco Tulio Valente. 2020. Turnover in Open-Source Projects: The Case of Core Developers. In *Proc. of Brazilian Symp. on Software Engineering*. 447–456.

[38] Isabella Ferreira, Jinghui Cheng, and Bram Adams. 2021. 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).

[39] Nicole Forsgren et al. 2021. 2020 State of the Octoverse: Securing the World's Software. *arXiv preprint arXiv:2110.10246* (2021).

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

[41] Jill J Francis et al. 2010. What is an adequate sample size? Operationalising data saturation for theory-based interview studies. *Psychology and Health* (2010).

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

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

[44] Mohammad Gharehyazie, Daryl Posnett, Bogdan Vasilescu, and Vladimir Filkov. 2015. Developer initiation and social interactions in OSS: A case study of the Apache Software Foundation. *Empirical Software Engineering* 20, 5 (2015), 1318–1353.

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

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

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

[48] Egon Guba. 1979. Naturalistic inquiry. *Improving Human Performance Qtrly.* (1979).

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

[50] Reza Hadi Mogavi, Ehsan-Ul Haq, Sujit Gujar, Pan Hui, and Xiaojuan Ma. 2022. More Gamification Is Not Always Better: A Case Study of Promotional Gamification in a Question Answering Website. *Proc. of the Human-Computer Interaction* (2022).

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

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

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

[54] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proc. Int'l Conf. Software Engineering (ICSE)*. 274–283.

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

[56] Yuekai Huang, Junjie Wang, Song Wang, Zhe Liu, Dandan Wang, and Qing Wang. 2021. Characterizing and Predicting Good First Issues. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[57] Giuseppe Iaffaldano, Igor Steinmacher, Fabio Calefato, Marco Gerosa, and Filippo Lanubile. 2019. Why do developers take breaks from contributing to OSS projects? A preliminary analysis. *arXiv preprint arXiv:1903.09528* (2019).

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

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

[60] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.

[61] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2022. Taxonomy of attacks on open-source software supply chains. *arXiv preprint arXiv:2204.04008* (2022).

[62] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. ACM, 245–256.

[63] Meir M Lehman. 1980. Programs, life cycles, and laws of software evolution. *Proc. IEEE* 68, 9 (1980), 1060–1076.

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

[65] Shmuel Leshem and Avraham Tabbach. 2016. Solving the Volunteer's Dilemma: The Efficiency of Rewards Versus Punishments. *American Law and Econ. Rev.* (2016).

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

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

[68] Yvonna S Lincoln and Egon G Guba. 1985. *Naturalistic inquiry.* sage.

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

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

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

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

[73] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. 2019. Why do people give up flossing? a study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems.*

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

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

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

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

[78] npm Docs. 2022. npm-deprecate. https://docs.npmjs.com/cli/v6/commands/npm-deprecate#synopsis. Accessed: 2022-07-07.

[79] npm Inc. 2018. 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-2019-3a3d7e5298ef. Accessed: 2022-08-19.

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

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

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

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

[84] Gauthami Polasani. 2022. Announcing the private beta of FOSSA Risk Intelligence. https://fossa.com/blog/announcing-private-beta-risk-intelligence/.

[85] Gede Artha Azriadi Prana et al. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* (2021).

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

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

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

[89] Romain Robbes, Mircea Lungu, and David Röthlisberger. 2012. How do developers react to API deprecation? The case of a Smalltalk ecosystem. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. 1–11.

[90] Steven G Rogelberg et al. 2003. Profiling active and passive nonrespondents to an organizational survey. *Jrnl. of Applied Psych.* (2003).

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

[92] Naomichi Shimada, Tao Xiao, Hideaki Hata, Christoph Treude, and Kenichi Matsumoto. 2022. GitHub Sponsors: exploring a new way to contribute to open source. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM, 1058–1069.

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

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

[95] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. 2016. Overcoming open source project entry barriers with a portal for newcomers. In *Proc. Int'l Conf. Software Engineering (ICSE)*.

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

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

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

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

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

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

[102] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*. 511–522.

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

[104] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*. 144–154.

[105] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. 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)*. 644–655.

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

[107] Wikipedia. 2022. Volunteer's Dilemma. https://en.wikipedia.org/wiki/Volunteer's_dilemma. Accessed: 2022-09-11.

[108] Ling Wu, Qian Wu, Guangtai Liang, Qianxiang Wang, and Zhi Jin. 2015. Transforming code with compositional mappings for API-library switching. In *Conf. Computer Software and Applications*, Vol. 2. IEEE, 316–325.

[109] Wenxin Xiao et al. 2022. Recommending good first issues in GitHub OSS projects. In *Proc. Int'l Conf. Software Engineering (ICSE)*.

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

[111] Nusrat Zahan et al. 2022. What are weak links in the NPM supply chain?. In *Proc. Int'l Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP)*.

[112] Xunhui Zhang et al. 2022. Who, what, why and how? towards the monetary incentive in crowd collaboration: A case study of GiHhub's sponsor mechanism. In *Proc. Conf. Human Factors in Computing Systems (CHI)*. ACM, 1–18.

[113] Minghui Zhou and Audris Mockus. 2014. Who will stay in the FLOSS community? Modeling participant's initial behavior. *IEEE Trans. Softw. Eng. (TSE)* (2014).

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

Received 2023-02-02; accepted 2023-07-27