Designing Abandabot: When Does Open Source Dependency Abandonment Matter?

Courtney Miller,[●●] Hao He,[●●] Weigen Chen,[●] Elizabeth Lin,[●] Chenyang Yang,[●] Bogdan Vasilescu,[●] Christian Kästner[●]
[●]Carnegie Mellon University [●]North Carolina State University
[●]Both authors contributed equally to this research.
{courtneymiller, vasilescu}@cmu.edu, haohe@andrew.cmu.edu, rnchen0218@gmail.com, etlin@ncsu.edu

Abstract

Despite the inevitable risk that depending on abandoned open source dependencies poses, many developers feel a lack of resources and guidance on how to deal with this. Automated detection of abandonment is feasible, but not all abandoned dependencies impact a downstream project equally. In this paper, we perform a need-finding interview study with 22 open source maintainers to explore what makes the abandonment of certain dependencies impactful to their project, as well as their information needs and design requirements for such an automated notification tool. We find four main factors, the depth of integration, the availability of alternatives, the importance of the functionality, and external environmental pressures. Using this emerging theory, we then build an LLM-based classifier to predict the impact of a dependency's abandonment in a given context, and evaluate it with an independent user study with 124 open source maintainers. Our results show that the classifier is effective at predicting whether a dependency's abandonment would be impactful to a project, and that theory-based explanations given by the LLM are useful to developers when making judgments about the potential impactfulness of a given dependency's abandonment.

1 Introduction

Open source software is used by nearly everyone and everything on the Internet [75, 98]. With this widespread reliance has come widespread expectations surrounding ongoing maintenance and support for these projects [30, 32]. However, these expectations are ill-fated, as recent work has shown that open source dependency abandonment, hereafter *abandonment* for short, is a prevalent issue [20, 22], even among widely used packages [9, 64].

Developers are concerned about abandonment (e.g., online [85]), particularly from a security perspective, as abandoned packages usually do not receive critical security patches [65, 100, 104]. In theory, developers could manually identify dependency abandonment and proactively respond before it potentially causes a concrete problem, but doing so is often infeasible at scale in practice [65], as the process many developers rely on is time- and effort-intensive. Automated tooling to identify abandonment is emerging,[1] as part of a broader range of software component analysis (SCA) tools to support other aspects of dependency management, e.g., dependency updates and security vulnerabilities. Adoption of SCA tools has become an industry-wide best practice [18, 76, 88, 91], and has been shown to help improve dependency management practices [40, 67].

However, the effectiveness of these tools is dampened by pervasive usability issues, with a primary issue being overwhelming developers with too many notifications, especially those users deem incorrect, unimportant, or irrelevant to their project [33, 40, 67, 83], which can lead to notification fatigue, ignoring tool notifications, and tool disengagement [33, 40, 62, 67, 89]. The issue of overwhelming developers with too many spurious notifications is prevalent across automated software engineering (SE) tooling [34, 48, 83, 89, 97]. Research on overcoming notification fatigue in such contexts suggests that only sending developers notifications they deem relevant can help alleviate the issue [97].

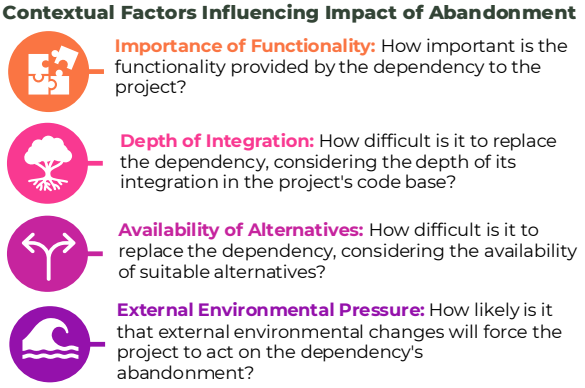

Figure 1: Four categories of context-specific information that affect the impactfulness of dependency abandonment.

Returning to abandonment, prior research similarly indicates that most developers do not care about the abandonment of all their dependencies equally; instead, they are primarily concerned about abandonment they believe would be *impactful* to their project [65]. Because of this, although only sending developers relevant notifications may sound like a straightforward solution, in the context of SCA tools to identify dependency abandonment, which we will refer to as the catchall term Abandabot, it leads to the non-trivial question: What dependency abandonment *will* be impactful to a particular project given the context of their dependency usage? With the goal of exploring this overarching question in specific software projects, we ask our first research question (RQ):

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

Research on the development of automated tools for developers has demonstrated that it is important for such tools to (1) be designed in a way that integrates organically with existing developer workflows; and (2) provide relevant succinct evidence for automatic judgments [33, 48, 68]. With the goal of further informing the design of a developer-centric Abandabot tool, we ask:

[1]Examples include FOSSA's Risk Intelligence service, currently in beta, and a recent research prototype by Mujahid et al. [69]. Several papers also describe prototypes to identify packages in decline [54, 94].

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

Through our research, we learned that it is sometimes difficult to make judgments of abandonment impact without relevant context information, and many participants confirmed that even making such judgments once, to set up an Abandabot tool, would be unrealistically tedious. Thus, we suggest a mechanism for predicting which abandonment will be impactful for a project to create an automatic pre-configuration—an idea enthusiastically supported by most participants. We design, implement, and evaluate Abandabot-Predict, a theory-driven LLM-based classifier to predict the impact of abandonment using context-specific reasoning and information derived from our theoretical understanding developed in RQ1. To assess our ability to automatically predict the impact of abandonment using our theory-driven classifier, we ask our third RQ:

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

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

We found that developers often cite four categories of context-specific information when considering how their usage of a dependency changes the impact of its abandonment on their project: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Fig. 1). Through our independent evaluation study, we leaned that our classifier is effective at predicting project-specific judgments of impactfulness and that the theory-driven context-specific information is perceived as useful to developers when making judgments.

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

2 Background and Related Work

2.1 Open Source Sustainability & Abandonment

Open source software serves as the ubiquitous building blocks of our digital lives [75], acting as the foundation for over 90% of the world's software [98]. With this widespread reliance has come widely held expectations that the maintainers of these projects are responsible for providing the ongoing maintenance and support effort necessary to keep the software up-to-date and meet user demands [30–32]. Despite these expectations, in reality, the continued maintenance of many open source projects is no sure thing.

Most open source maintainers are still unpaid volunteers to this day [93], and most projects rely on a small number of overworked and underappreciated maintainers to do most of the work [10, 58]. Open source maintainers may disengage and stop contributing at any point for normal reasons that we cannot prevent e.g., switching jobs, a lack of time, or losing interest [15, 21, 35, 59, 66, 99]. When maintainers do disengage, more often than not, no one else steps up and the project becomes fully abandoned [9], making abandonment an inevitable risk even among widely used projects [64].

This disconnect between the expectations placed on and the reality of open source has motivated the need to study and improve open source sustainability, which is an active and vibrant research area. However, most sustainability research thus far has primarily focused on studying and improving various processes, phenomena, and characteristics to support the goal of keeping specific open source projects alive and maintained [65].

A notable exception is a recent line of work by Miller et al. which argues that due to our collective widespread reliance on open source and the inevitable risk abandonment, that sustainability research should 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" [64, 65].[2] Many developers feel they have limited support when facing abandonment [65] adn would like to identify abandonment before it results in a concrete issue so they can respond without immediate time pressures (at least for certain dependencies), yet most rely on time- and effort-intensive manual investigation of various project characteristics, e.g., commit frequency, lack of updates, and unresolved issues and pull requests, making proactive identification at scale infeasible in practice [65].

There is also quantitative evidence suggesting that an Abandabot intervention that makes abandonment more visible is a promising direction: projects react significantly faster when explicit notice of abandonment[3] is provided rather than when maintenance silently ceases [64], demonstrating that increasing information transparency surrounding abandonment can support more timely downstream responses. In this paper, we support the sustainable use of open source by informing the design of a developer-centric Abandabot tool to assist developers facing abandonment.[4]

[2]Other exceptions include work exploring package-level deprecation in the Python ecosystem [103], identifying alternatives for packages in decline [69], and measuring and presenting community and library health metrics to potential adopters [70, 95].
[3]i.e., when maintainers explicitly express their intention to no longer maintain the package e.g., by flagging the repository as archived or adding a note to the README [64]
[4]It is important to note that there is no widely agreed upon definition of when inactivity crosses the threshold into abandonment or what signals of activity should be used to make that determination, with many different operationalizations being used in previous work on the subject [9, 21, 44, 63–65, 95]. In this paper, we avoid this issue by

2.2 Dependency Management Tooling

Although there are significant benefits to using open source dependencies [30], there are also downsides, namely *dependency management*. Numerous calls for improved dependency management practices have been made, including an executive order from the US White House [7], yet research consistently demonstrates that most developers neglect updating dependencies, including updates with known vulnerabilities, even when notified by automated tools [12, 16, 26–28, 47, 50, 61, 79–81, 87, 90, 102]. In fact, there are entire organizations like the Open Source Security Foundation (OpenSSF) whose primary mission includes developing research, best practices, evaluation metrics, and enterprise tools to make *it easier to sustainably secure the development, maintenance, and consumption of the open source software (OSS) we all depend on* [1, 76].

Although, as discussed, there is an unmet need for Abandabot tooling, there are many well established software component analysis (SCA) tools to support other dependency management tasks, such as dependency updates, security vulnerabilities, and license management, including Snyk Bot [5], Dependabot [3], Socket [6], and Sonatype [86]. These tools are designed with the intention of reducing the developer workload by automating routine dependency management tasks e.g., keeping the dependencies of a project up to date by notifying developers of update opportunities and creating automated pull requests with proposed updates. Adoption of such tools has become an industry-wide best practice [18, 76, 88, 91], and research has shown that their adoption can lead to positive improvements in dependency management practices [40, 67].

Yet research on the usability of those same tools has found that these effects are tempered by pervasive usability issues, with one of the primary issues being sending developers too many notifications, especially those they deem incorrect, unimportant or irrelevant to their project [33, 40, 67, 83]. These notifications are often perceived as noise and can distract, annoy, and overwhelm developers causing information overload and notification fatigue which can lead to developers ignoring the tool or disengaging altogether [33, 40, 62, 67, 89]. The issue of overwhelming developers with too many spurious notifications is pervasive across automated tooling for many different software engineering (SE) tasks [34, 48, 83, 89, 97] e.g., static analysis tools [11, 42, 84], automated fault detection tools [51], and security alert tools [49, 77, 78].

Research on overcoming notification fatigue in such contexts has suggested that only sending developers notifications they deem relevant may alleviating the issue [97]. Sawdowski et al. coined the term *effective false positive* which refers to automated tool notifications that are technically correct but that do not matter to the user in practice. Since many developers are not equally concerned about dependency abandonment because the impact on their project can vary widely [65], we define effective false positives in the context of Abandabot tools as *notifications about the abandonment of dependencies that are not considered impactful to the project by its maintainers*. Although only sending developers relevant notifications may sound straightforward, it leads to the non-trivial question of 'what abandonment *will* be impactful to a particular project given the context of their dependency usage?' Which we we take a step towards answering in this paper.

discussing either hypothetical abandonment, cases where explicit notice is provided, or cases where participants independently judge that abandonment occurred, allowing us to avoid the question of what constitutes abandonment and instead engage in discussions within the context of assuming abandonment has already been identified.

3 Need-Finding Interviews

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

3.1 Research Design

To answer RQ1 and RQ2, we conducted a semi-structured formative need-finding interview study [53]. We used an interview-based design because we wanted to have nuanced discussions with developers about their experiences, opinions, and reasoning since this is a relatively unexplored topic. Furthermore, interviews are a popular method for eliciting information needs and design requirements from tool users in human-computer interaction research [72].

To contextualize discussions about the project's dependencies, their maintenance status, and as a starting point to help spark more rich grounded discussions about tool design, we developed a preliminary Abandabot prototype which we used in the interviews as a method of *experience prototyping* [14, 39] (cf. paper appendix included in the supplemental materials on HotCRP). We concluded running interviews when we reached our predetermined theoretical saturation criteria of three consecutive interviews without any new major insights or changes to our theoretical understanding [36].

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

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

The second focus was on eliciting design requirements and information needs for an Abandabot tool using a *participatory design*

process [46, 71], in which asked questions about: (1) tool and notification modality; (2) preferences regarding customizing tool configurations; and (3) what dependency information and additional context they would like to receive with abandonment notifications.

Identifying and Recruiting Participants. Because we wanted to speak with developers that had knowledge of and experience with abandonment, our goal was to recruit maintainers open source projects that have faced or currently face dependency abandonment. We focus our participant pool on JavaScript projects because JavaScript has the largest package manager ecosystem, npm [88], prevalent dependency management issues [23, 26, 104], a comprehensive registry and configuration design that makes tracking dependency usage relatively straightforward, and it is the language our Abandabot prototype currently supports. We also wanted to ensure that the projects we reached out to (1) had recent activity to increase the likelihood of response (i.e., at least 10 commits in the past year); and that (2) at least one maintainers with an email address listed on their public GitHub profile for recruitment.

We used two complementary strategies to identify participants. First, we started with a list of abandoned packages in the npm ecosystem from previous research [64], then worked backward to identify dependent projects that fit our criteria. Since we did not reach our saturation criteria after exhausting the list of candidates from the first strategy, we performed a second strategy in which we worked forward, first identifying a broad pool of projects that fit our criteria (excluding the abandoned dependency criteria) then identifying the subset that had at least one abandoned dependency.

For both strategies we used *World of Code (WoC)* Version V to identify candidate projects on GitHub [57]. To scrape each project's `package.json` and recent commits, we cloned each project's repository for strategy one and used the GitHub REST API for strategy two. Additionally, for strategy two, to identify abandoned dependencies, we cross-referenced each dependency with the npm registry, using the npm API to identify the subset of dependencies that had either (1) been flagged as deprecated; or (2) not published a release in at least three years. Finally, we collected each maintainer's email address from their public GitHub profile using the GitHub REST API if available. To encourage participation, we sent each participant a personalized email invitation and offered them a USD $20 Amazon gift card upon completion of the interview as a token of our gratitude and compensation for their time.

Data Collection and Analysis. In total, we conducted 22 interviews via Zoom which lasted between 30 and 45 minutes. Since our aim was to develop an understanding of a relatively unexplored phenomenon through the experiences of participants, we used thematic analysis to qualitatively analyze the interview data [13, 19, 92]. We performed our data collection and analysis procedures simultaneously and iteratively [39]. While running the interviews we frequently oscillated between the stages of open coding, exploring the rich transcripts, analytically memoing and engaging with the data, refining the codes and coding framework, and searching for themes in the data [24]. We used an interwoven constant comparative method to refine our emerging categories, comparing and adjusting our emerging categories using interview data [24].

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

Limitations. Our interview study is affected by several limitations commonly experienced in such research. The transferability of our findings may be influenced by self-selection bias among participants [60, 82], as there could be differences in beliefs and opinions between the candidates in the full sample we invited to participate and the subset that did. Sampling limitations may impact the findings since we specifically identified participants who (1) maintain open source projects; and (2) have experienced dependency abandonment, which does not represent the full range of JavaScript developers. Generalizations beyond the sampled participant distribution should be done with care.

3.2 RQ1 Results - What Influences the Importance of Abandonment

Through the need-finding interviews, we identified four categories of context-specific information that participants commonly cited when considering how their use of a given dependency affects the impact its abandonment would have on their project: The depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Fig. 1). We discuss each category in turn below. However, we first briefly discuss a related meta-finding from the interviews.

Meta Finding: Difficulties Surrounding Making Judgments About Abandonment Impact. In interview discussions, the four categories of information were not always made explicit. Many participants did not cite them directly, instead mentioning low-level pragmatic signals representative of the categories. Some had trouble articulating why they believed the abandonment of a particular dependency would be impactful to their project even when they were confident in the judgment; and sometimes the justifications would come up out of context after several additional examples had been discussed, giving them an opportunity to develop a deeper understanding of their own beliefs through the conversation.

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

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

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

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

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

Participants also considered other potential sources of alternatives, including well-maintained popular forks or native solutions built into the language or framework they are using. In addition, participants considered how complex the functionality provided by the dependency is and how feasible it might be for them to implement the functionality themselves. If the functionality being used is simple, some participants considered the potential abandonment less concerning because they could potentially remove the dependency and replace it with their own in-house implementation.

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

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

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

External Environmental Pressures. Finally, the fourth category is how much external environmental pressure the dependency has to continue to evolve and keep up with ongoing environmental changes. The abandonment of dependencies in ecosystems that exert more external pressure to continue evolving with the environment was often more concerning to participants because they anticipated that the abandonment may cause some sort of incompatibility issue sooner rather than later. For example, a dependency like *@typescript-eslint/parser*, which is a development tool plugin for typescript, would have to keep up with updates in the larger typescript ecosystem or become increasingly stale over time, whereas dependencies like *isarray* or *left-pad* that provide simple, narrowly-scoped functionality and that have limited external dependencies could potentially be unmaintained for an extended period of time without users experiencing any adverse effects (barring any rogue issues like a zero-day vulnerability or an incident like the left-pad one [74]). This aligns with a recent study that found that language incompatibility issues were a common concrete issue faced by developers dealing with dependency abandonment [65].

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

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

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

3.3 RQ2 Results - Tool Design Requirements and Information Needs

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

Information Needs. *Evidence for Judgment of Abandonment.* Participants wanted concise relevant evidence supporting the automated judgment of abandonment. Whether that be an explicit notice of abandonment provided by maintainers, or activity patterns that

were used to make the judgment, i.e., how long the dependency has been inactive for and what activity signals were considered.

Information About Dependency Usage in Project. Participants wanted summary information about their dependency use so they could get a sense of how impactful the abandonment may be and how much work replacement might take, aligning with concerns surrounding the importance of functionality (cf. Sec. 3.2). Many of the specific signals requested were the same as the pragmatic signals for depth of integration (cf. Sec. 3.2).

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

Information About Potential Alternatives and Next Steps. Finally, participants wanted information about possible alternatives and possible next steps. The type of information requested aligned closely with the type of information considered when evaluating the availability of alternatives (cf. Sec. 3.2).

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

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

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

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

4 Abandabot-Predict: Predicting Impactful Dependency Abandonment

In the need-finding interviews, we identified four categories of context-specific information that participants often cited when making judgments about the impact of abandonment: the depth of integration, the availability of alternatives, the importance of functionality, and external environmental pressures (cf. Sec. 3.2).

A recurring theme from the interviews is that the process of judging the impact of abandonment is sometimes highly ad hoc and context-dependent, requiring in-depth domain knowledge and hard-to-collect information–even when participants know what they are looking for. Thus, the process is not easily operationalizable using simple heuristics or program analysis techniques.[5] However, the ability to automatically identify impactful dependency abandonment is one of the key design requirements for tools to support dependency abandonment (cf. Sec. 3.3). In this paper, we conjecture that *large language models (LLMs), with proper theory-driven reasoning and contextual information, can serve as a tool to provide accurate abandonment impact predictions to support developer decision-making*. We design, implement, and evaluate Abandabot-Predict, an LLM-based classifier for predicting abandonment impact using theory-driven reasoning and context-specific information.

4.1 Approach

At a high-level, Abandabot-Predict takes in a GitHub repository and a dependency name, extracts *context-specific information* about dependency usage, and constructs a *theory-based reasoning prompt*, based on the four categories of information and the contextual information extracted (cf. Fig. 2). The prompt is then fed into an LLM which performs a series of sequential reasoning steps culminating in a final binary prediction of *impactful* or *not impactful*. We now describe the two key components of Abandabot-Predict: *extracting contextual information* and *theory-based prompting* below.

Extracting Contextual Information. The goal of this component is to extract a body of relevant information informed by our theoretical understanding that would be sufficient for a human expert to make a judgment on the impact of abandonment, so that an LLM can conduct similar reasoning and judgments (i.e., the Retrieval-Augmented Generation pattern [52]). It is necessary to extract a subset of all information available, as Abandabot-Predict needs to work on large software projects, whose size well exceeds the context window of any current LLMs. Therefore, we extract the following theory-driven contextual information:

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

The first three pieces of contextual information are trivial to extract. To obtain the final piece, Abandabot-Predict combines keyword search with global data-flow analysis [73]: The former identifies all locations whether the dependency name appears, possibly covering documentation and configuration files; the latter identifies all locations in the source code where an API of the dependency is possibly used. If a dependency is used in more than N different locations (N is a configurable parameter), Abandabot-Predict will downsample only N locations, to avoid exceeding the maximum-allowed context

[5] e.g., several participants pointed out that the number of API calls is not a comprehensive signal for *depth of integration*— Since, for example, a core development tool can have little usage in the source code, but its abandonment would likely be impactful.

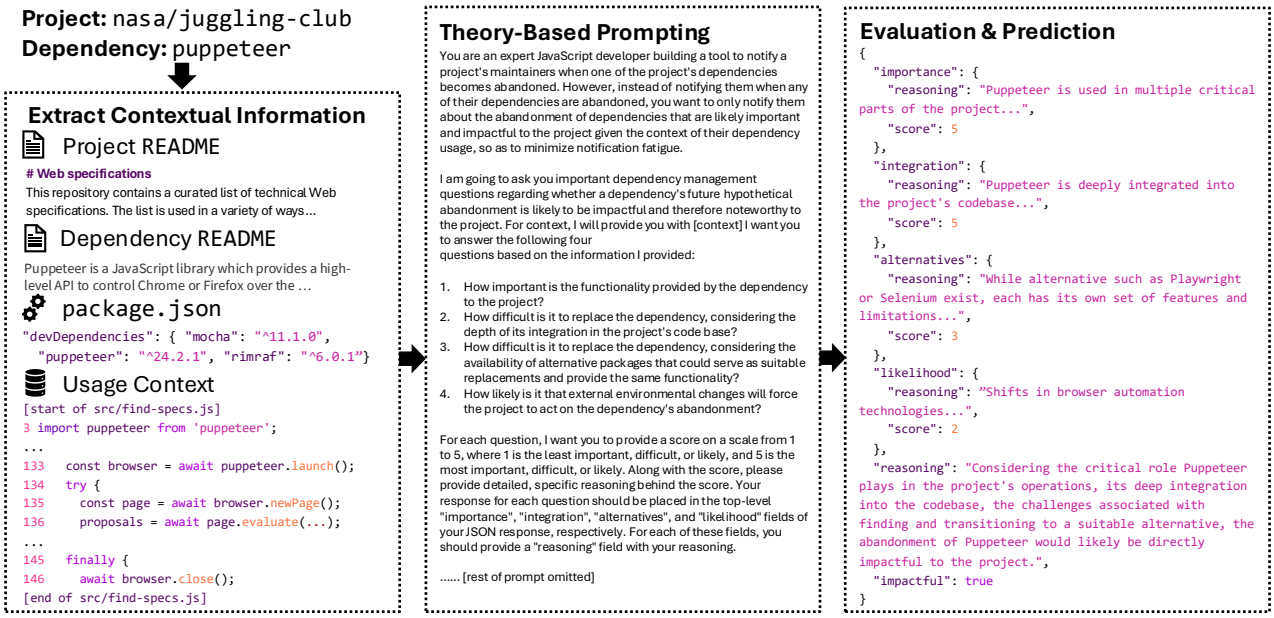

Figure 2: An example input & output from Abandabot-Predict

window (128k for most of our tested LLMs); we believe that this downsampling also resembles the behavior of a human expert, who can usually form a judgment by inspecting only a small subset of related information (i.e., "thin-slicing" as called in the psychology and philosophy literature [8]).

Theory-Based Prompting. The goal of this component is to construct a prompt that can effectively instruct an LLM to generate reasoning for impact judgments in a way similar to that of human experts (i.e., adopting a reasoning process using the categories of information identified in RQ1). The prompt starts with a role-playing directive telling the LLM to act as an expert software developer. Then, it informs the LLM of the task (i.e., predicting the impact of abandonment) and instructs the LLM to perform chain-of-thought reasoning [96] for each of the four categories based on the contextual information provided. Finally, it instructs the LLM to conduct an additional step of chain-of-thought reasoning on whether the abandonment would be impactful, before generating a final binary recommendation (*impactful* or *not impactful*). All contextual information is concatenated at the end of this prompt.

4.2 Implementation

We implement Abandabot-Predict in Python using on CodeQL [2] and LangChain [4]; the former enables production-grade global data-flow analysis and the latter simplifies prompting and plug-ins for different LLMs. Currently, Abandabot-Predict only supports JavaScript/TypeScript projects with `package.json` files, but we expect it to be trivial to extend the same analysis to other package managers and CodeQL-supported programming languages with the `DataFlow` module. The global data-flow analysis is implemented by extending `DataFlow::ConfigSig`, where we set the source as import/require statements and the sink as invoke nodes. Using this extension, Abandabot-Predict executes a CodeQL query to find all locations in the project's source code where a dependency declared in the `package.json` is imported and used. In the not uncommon case where the query times out after an hour in a very large code base, Abandabot-Predict regresses to conduct the same analysis on local data flows instead. In the current implementation, we set $N = 50$ and $W = 10$ (i.e., inspecting a maximum of 50 dependency usage locations, each including 10 surrounding lines), based on our intuition of the amount of context required by a human expert to judge on the impact of dependency abandonment.

4.3 Offline Evaluation

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

Dataset. Recall interviewees discussed specific dependencies and whether they believed their abandonment would be impactful (cf. Sec. 3.1). We compiled a list of 82 project dependency pairs from the interview transcripts, with 57 being judged impactful and 25 being judged unimpactful by participants, which served as our ground-truth dataset.

Models. We choose the following LLMs for evaluation: GPT-4o [41], DeepSeek-V3 [55], Llama-3.3-70B-Instruct [29], and Gemini-2.0-Flash [37]. We choose them because they are state-of-the-art general purpose LLMs with large context windows (≥128k).

Baseline Approaches. Apart from the Abandabot-Predict approach we introduced in Section 4.1, we introduce the following three alternative baseline approaches for comparison:

(1) Abandabot-Predict-Baseline: This baseline uses a basic prompt with role-playing directives and chain-of-thought reasoning to

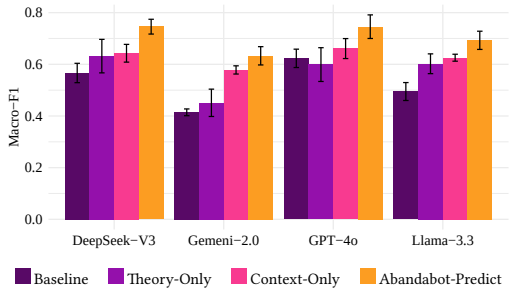

Figure 3: The Macro-F1 performance results (average and std. dev. over ten runs) for different LLMs and baselines.

make abandonment impact predictions; no theory-driven reasoning instructions or context-specific information is provided.

(2) Abandabot-Predict-Theory-Only: Extending the baseline above, it further provides theory-based reasoning instructions but does not provide any context-specific information.

(3) Abandabot-Predict-Context-Only: Extending the baseline above, it further provides context-specific information, but does not provide any theory-driven reasoning instructions.

Evaluation Metric. We use Marco-F1 [38] as the main evaluation metric because in an imbalanced dataset, it gives equal importance to minority classes. Single-label precision, recall, and F1 scores would be misleading in our case. For example, a classifier that always predicts "impactful" for every dependency would achieve a deceptively high precision of 69.5%, recall of 100%, and F1 of 82.0%, if we compute these metrics based on the "impactful" label. For each LLM and approach configuration, we run Abandabot-Predict on each project dependency pair in the ground truth dataset ten times and compute the average and standard deviations of the Macro-F1s, to address the occasional uncertainty in its predictions.

Results. All models achieve the best overall performance when provided with theory-driven reasoning instructions and context-specific information (cf. Fig. 3). GPT-4o and DeepSeek-V3 outperform the others (0.746 Macro-F1). We chose to use DeepSeek-V3 for our independent evaluation study because it has comparable performance and is cheaper than GPT-4o. All models achieve better performance compared to the baseline if supplemented with contextual information; the same does not necessarily apply to theory-based prompting, which may have caused the LLM to hallucinate more without any contextual information (e.g.. in the case of DeepSeek-V3 and GPT-4o). In all LLMs, Abandabot-Predict outperforms random guessing, which always has a 0.5 Macro-F1.

4.4 Independent Evaluation Study

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

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

- If we show participants our predicted judgment and ask them if they agree point-blank, there could be obvious issues with *acquiesce bias* [25], as participants may be inclined to agree with the judgment provided.
- If we counteracted this by first asking them to make a judgment without and with context transparently, this design could suffer from obvious *consistency bias* [17], as participants may double down on their first judgment subconsciously.

In an attempt to balance these biases, we designed the following evaluation methodology consisting of three steps in a single survey. In step 1 we asked participants to *judge the impactfulness* of abandonment without context. In step 2 on the next page of the survey, we frame their task as providing constructive feedback on automatically generated judgments from an unnamed prototype tool so that they feel more comfortable disagreeing (attempting to address acquiesce bias). Here, we introduce the four categories of context-specific information, provide *Abandabot-Predict*'s contextual reasoning for each category, and the final predicted judgment for the same three dependencies. We ask them to rate their *agreement* with the final judgment and the reasoning for each category–note that we intentionally ask a different question than in step 1 framed as constructive feedback with the *Abandabot-Predict* judgment in an attempt to minimize the effect of commitment-consistency bias. Finally, in the third part, after all the judgments have been made, we ask them to rate how useful each of the four categories of information was in supporting their decision-making.

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

Survey Design. The survey consists of three steps. In step 1, we ask participants to provide a binary judgment about whether they believe each dependency's abandonment would likely be impactful to their project, without providing our judgment or any additional context. In step 2, we provide the *Abandabot-Predict* contextual

reasoning for each of the four categories and its final judgment for the same three dependencies and ask them to rate their agreement with the final judgment and the reasoning for each category, which we intentionally frame as providing constructive feedback on a prototype tool. In step 3, we ask them to rate how useful they believe each of the four categories is in informing judgments about the potential impactfulness of dependency abandonment on a 5-point rating scale. We provide the complete survey template in the supplemental materials (cf. Sec. 7).

Identifying Participants and Project Dependencies. We aimed to recruit participants who meet the same criteria used in the initial need-finding interviews (cf. Sec. 3.1), i.e., maintainers of active JavaScript projects who have faced or currently face dependency abandonment. We used the same pool of participants identified using our second sampling strategy, excluding any projects we had already invited to participate in the need-finding study.

Stratified Sampling of Dependencies. For each participant and corresponding project, we ran *Abandabot-Predict* on all dependencies declared in their `package.json` file, generating an evaluation and binary impact prediction for each. We then select the three dependencies that we include in the survey using a stratified sampling method using three dependency sampling strata. We only ask each participant about three dependencies to encourage participation in and completion of the survey, considering it better to have fewer judgments from more participants than more judgments from fewer participants and also to expand the pool of experiences and perspectives included in our evaluation. We randomly select one dependency each from the pools of predicted impactful and not impactful dependencies. To ensure that we evaluate Abandabot-Predict's performance in potentially more difficult context-dependent cases, we randomly select one dependency from the pool of dependencies where the judgment was content dependent, i.e., from the subset of dependencies where *Abandabot-Predict* generated a different judgment for the same dependency in a different project. We use Qualtrics to generate personalized surveys for each participant and email invitations to 1,673 randomly selected qualifying maintainers, and in total we received 152 responses, i.e., a response rate of 9%.

Evaluating Survey Results. To answer RQ3, we analyzed the 124 responses that at least completed step 1,[6] generating 690 importance judgments about 372 dependencies in 124 distinct repositories. We compared the classifier prediction with the participant judgments from step 1 and step 2, reporting a Micro-F1 for both parts, which serve as an upper and lower bound on classifier performance. We calculated how frequently participants changed their opinion about a dependency's importance in step 2 of the survey and assessed the frequency and directionality of changes in opinion. We infer their judgment of importance in step 2 based on their agreement rating with the predicted judgment. In addition, to assess the perceived usefulness of each of the four categories of information since the measures are ordinal, we evaluate and compare the rating distributions.

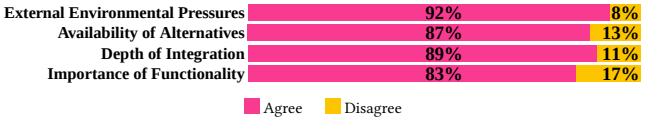

Figure 4: Percentage of judgments where participants agreed or disagreed with the reasoning provided for each category.

4.5 RQ3 Results

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

Participants mostly agreed with the reasoning provided for each of the four categories of context-specific information provided by *Abandabot-Predict*. Participants fully agreed with the reasoning for all four categories 75% of the time. We provide a breakdown of user agreement with the reasoning provided for each category in Fig. 4. Most of the participants considered context-specific information useful in informing their judgments (cf. Fig. 5). The categories of depth of integration and availability of alternatives we considered most useful, with 73% and 72% participants considering them very useful or extremely useful when making judgments about the impact of abandonment, respectively. External environmental pressure was considered the least useful by far. We speculate that this is because it is a relatively niche concept that may be difficult to grasp without further elaboration or examples.

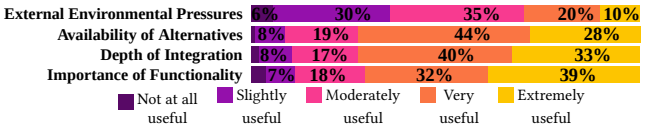

Figure 5: Rating distribution of perceived usefulness for each category of context-specific information.

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

4.6 Limitations and Threats to Validity

The use of LLMs to support dependency management decisions comes with its own risks. In addition to the general risk of hallucination and misleading developers in its generated reasoning [56], it is also possible that an LLM learns human bias, does not account for the latest changes, and makes unfair decisions (e.g., it may learn from a training data showing an unjustified strong favor/disfavor

[6] 91 responses fully completed the survey. The survey was intentionally designed so participants could stop at any point, but completing step 1 was the minimum required to be useful in our analysis.

to a certain dependency). Also, while our prototype demonstrates the overall feasibility of the approach, implementing a fully autonomous bot would require more engineering considerations, e.g., managing the cost of continuous large-scale scanning.

Our offline evaluation is based on a small ground truth dataset, making it hard to mitigate the threat of overfitting. We mitigate this threat through the independent evaluation study on projects outside of the dataset. In addition, human decisions are generally imperfect, often relying on limited experience and faulty mental models [43]—adding noise to the evaluation dataset and performance results. Future research is needed to understand the nature of human biases and errors in our problem—analogously to previous research [45].

The survey results may be affected by self-selection bias, as is a common risk in such studies. Due to our survey design, the results may suffer from commit-consistency and acquiescence bias [25].

5 Discussion and Implications

5.1 Intelligent Tool Pre-Configuration

Within the context of attempting to reduce noise in automated SE tooling by filtering out irrelevant notifications, our findings illustrate the potential for intelligent pre-configuration of context-aware tooling. Although existing analyses can provide general guidance, effective defaults often require nuanced judgments based on project-specific context, which are challenging to encode through purely technical rules. For example, a CSS-injection vulnerability in a dependency of a static website generator might be technically reachable but practically irrelevant since the program does not process external inputs, highlighting that technical reachability alone is insufficient for accurate judgments in applications such as the one explored in this paper. This need for customization has also been repeatedly recognized for various static analysis tools [84]. Similarly, determining whether dependency abandonment would impact a project requires understanding how the dependency is integrated, what functionality it provides, and other contextual factors we identified. These scenarios necessitate understanding the project's socio-technical context, which is not trivially captured through static analysis or API compatibility analysis. In our case, we do not see a plausible path to determine the importance of dependency abandonment purely with technical code analysis.

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

Implications for researchers and tool builders: Our findings suggest that future work could apply similar methodologies to identify contextual factors that make tool notifications relevant to developers for automated tooling across various software engineering processes. Our experience suggests that tool builders may want to consider incorporating automatic pre-configuration capabilities that consider project-specific context to improve developer experience. Tool designers could also leverage participatory design to refine these defaults, ensuring that the tool's "smart" behavior matches user expectations. In practice, this could mean embedding analytics that learn from project characteristics and developer feedback to auto-tune alert thresholds [84].

5.2 Synergistic Design: Adding Theory to LLMs

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

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

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

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

LLMs and static analysis techniques to improve state-of-the-art performance in the detection of malicious npm packages [101].

6 Conclusion

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

7 Data Availability

The appendix, interview guide, evaluation survey, and an anonymized version of the Abandabot-Predict repository are available in the supplemental materials on HotCRP. We will post the final supplementary materials publicly on Zenodo with the camera ready.

Acknowledgments

We thank the reviewers for their consideration.

References

[1] [n. d.]. About OpenSSF. https://openssf.org/about/. Accessed Mar. 2025.
[2] [n. d.]. CodeQL. https://codeql.github.com/. Accessed Mar. 2025.
[3] [n. d.]. Dependabot. https://dependabot.com. Accessed: 2024-03-16.
[4] [n. d.]. LangChain. https://www.langchain.com/. Accessed Mar. 2025.
[5] [n. d.]. Snyk Bot. https://github.com/snyk-bot. Accessed Mar. 2025.
[6] [n. d.]. Socket. https://socket.dev. Accessed Mar. 2025.
[7] 2021. Executive Order 14028: Improving the Nation's Cybersecurity. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.
[8] Nalini Ambady and Robert Rosenthal. 1992. Thin slices of expressive behavior as predictors of interpersonal consequences: A meta-analysis. *Psychological Bulletin* 111, 2 (1992), 256.
[9] Guilherme Avelino et al. 2019. On the abandonment and survival of open source projects: an empirical investigation. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*.
[10] 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.
[11] Nathaniel Ayewah and William Pugh. 2008. A report on a survey and study of static analysis users. In *Proceedings of the 2008 workshop on Defects in large software systems*. 1–5.
[12] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the Apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering* (2015).
[13] Virginia Braun and Victoria Clarke. 2006. Using thematic analysis in psychology. *Qualitative research in psychology* 3, 2 (2006), 77–101.
[14] Marion Buchenau and Jane Fulton Suri. 2000. Experience prototyping. In *Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques*. 424–433.
[15] Fabio Calefato et al. 2022. Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub. *Empirical Software Engineering* (2022).
[16] Bodin Chinthanet et al. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* (2021).
[17] Robert B Cialdini et al. 2009. *Influence: Science and practice*. Vol. 4. Pearson education Boston.
[18] CISA. 2023. *Securing the software supply chain: reccomended practives for managing open-source software and software bill of materials*. Technical Report. CISA. https://www.cisa.gov/sites/default/files/2023-12/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN%20RECOMMENDED%20PRACTICES%20FOR%20MANAGING%20OPEN%20SOURCE%20SOFTWARE%20AND%20SOFTWARE%20BILL%20OF%20MATERIALS.pdf
[19] Victoria Clarke and Virginia Braun. 2017. Thematic analysis. *The journal of positive psychology* 12, 3 (2017), 297–298.
[20] Jailton Coelho et al. 2020. Is this GitHub project maintained? Measuring the level of maintenance activity of open-source projects. *Information and Software Technology (IST)* (2020).
[21] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proc. Int'l Symposium Foundations of Software Engineering (FSE)*.
[22] Jailton Coelho, Marco Tulio Valente, Luciana L Silva, and Emad Shihab. 2018. Identifying unmaintained projects in GitHub. In *Proc. Int'l Symp. Empirical Software Engineering and Measurement (ESEM)*.
[23] Lucian Constantin. 2018. Npm Attackers Sneak a Backdoor into Node.js Deployments through Dependencies. https://thenewstack.io/npm-attackers-sneak-a-backdoor-into-node-js-deployments-through-dependencies/. Accessed: 2024-02-28.
[24] John W Creswell and Cheryl N Poth. 2016. *Qualitative inquiry and research design: Choosing among five approaches*. Sage publications.
[25] Lee J Cronbach. 1946. Response sets and test validity. *Educational and psychological measurement* 6, 4 (1946), 475–494.
[26] 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.
[27] 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.
[28] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me updated: An empirical study of third-party library updatability on Android. In *Proc. Conf. Computer and Communications Security (CCS)*.
[29] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The Llama 3 Herd of Models. *arXiv preprint arXiv:2407.21783* (2024).
[30] Nadia Eghbal. 2016. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation.
[31] Nadia Eghbal. 2019. The rise of few-maintainer projects. https://increment.com/open-source/the-rise-of-few-maintainer-projects/. Accessed: 2024-08-15.
[32] Nadia Eghbal. 2020. *Working in public: the making and maintenance of open source software*. Stripe Press.
[33] Linda Erlenhov, Francisco Gomes de Oliveira Neto, and Philipp Leitner. 2022. Dependency management bots in open-source systems—prevalence and adoption. *PeerJ Computer Science* 8 (2022), e849.
[34] Linda Erlenhov, Francisco Gomes De Oliveira Neto, and Philipp Leitner. 2020. An empirical study of bots in software development: Characteristics and challenges from a practitioner's perspective. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 445–455.
[35] Nicole Forsgren et al. 2021. 2020 State of the Octoverse: Securing the World's Software. *arXiv preprint arXiv:2110.10246* (2021).
[36] Jill J Francis et al. 2010. What is an adequate sample size? Operationalising data saturation for theory-based interview studies. *Psychology and Health* (2010).
[37] howpublished = https://deepmind.google/technologies/gemini/flash/ note = Accessed: March 2025 Google Inc, title=Gemini Flash. [n. d.].
[38] Margherita Grandini, Enrico Bagli, and Giorgio Visani. 2020. Metrics for multi-class classification: An overview. *arXiv preprint arXiv:2008.05756* (2020).
[39] Bruce Hanington and Bella Martin. 2019. *Universal methods of design expanded and revised: 125 Ways to research complex problems, develop innovative ideas, and design effective solutions*. Rockport publishers.
[40] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on GitHub Dependabot. *IEEE Transactions on Software Engineering* (2023).
[41] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. GPT-4o System Card. *arXiv preprint arXiv:2410.21276* (2024).
[42] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 672–681.
[43] Daniel Kahneman, Paul Slovic, and Amos Tversky. 1982. *Judgment Under Uncertainty: Heuristics and Biases*. Cambridge University Press.
[44] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. 2013. Is it all lost? A study of inactive open source projects. In *IFIP Int'l Conf. on Open Source Systems*. Springer, 61–79.
[45] Jon Kleinberg, Himabindu Lakkaraju, Jure Leskovec, Jens Ludwig, and Sendhil Mullainathan. 2018. Human decisions and machine predictions. *The Quarterly Journal of Economics* 133, 1 (2018), 237–293.
[46] Amy J Ko, Robert DeLine, and Gina Venolia. 2007. Information needs in collocated software development teams. In *Proc. Int'l Conf. Software Engineering*

(ICSE). IEEE, 344–353.
[47] Raula Gaikovina Kula et al. 2018. Do developers update their library dependencies? *Empirical Software Engineering* 23, 1 (2018), 384–417.
[48] Stefano Lambiase, Gemma Catolino, Fabio Palomba, and Filomena Ferrucci. 2024. Motivations, Challenges, Best Practices, and Benefits for Bots and Conversational Agents in Software Engineering: A Multivocal Literature Review. *Comput. Surveys* 57, 4 (2024), 1–37.
[49] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not all dependencies are equal: An empirical study on production dependencies in npm. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.
[50] Tobias Lauinger et al. 2018. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. *arXiv preprint arXiv:1811.00918* (2018).
[51] Lucas Layman, Laurie Williams, and Robert St Amant. 2007. Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 176–185.
[52] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.
[53] Sarah Lewis. 2015. Qualitative inquiry and research design: Choosing among five approaches. *Health promotion practice* 16, 4 (2015), 473–475.
[54] Xiaozhou Li, Sergio Moreschini, Fabiano Pecorelli, and Davide Taibi. 2022. OSSARA: abandonment risk assessment for embedded open source components. *IEEE Software* 39, 4 (2022), 48–53.
[55] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 Technical Report. *arXiv preprint arXiv:2412.19437* (2024).
[56] Fang Liu, Yang Liu, Lin Shi, Houkun Huang, Ruifeng Wang, Zhen Yang, Li Zhang, Zhongqi Li, and Yuchi Ma. 2024. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971* (2024).
[57] Yuxing Ma et al. 2021. World of Code: Enabling a research workflow for mining and analyzing the universe of open source VCS data. *Empirical Software Engineering* 26 (2021).
[58] Suvodeep Majumder, Joymallya Chakraborty, Amritanshu Agrawal, and Tim Menzies. 2019. Why software projects need heroes (lessons learned from 1100+ projects). *arXiv preprint arXiv:1904.09954* (2019).
[59] Pia Mancini et al. 2021. *Sustain: A One Day Conversation for Open Source Software Sustainers – The Report*. Technical Report. Sustain Conference Organization. https://sustainoss.org/assets/pdf/Sustain-In-2021-Event-Report.pdf
[60] 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).
[61] Tyler McDonnell, Baishakhi Ray, and Miryung Kim. 2013. An empirical study of API stability and adoption in the Android ecosystem. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 70–79.
[62] Tom Mens and Alexandre Decan. 2024. An Overview and Catalogue of Dependency Challenges in Open Source Software Package Registries. *arXiv preprint arXiv:2409.18884* (2024).
[63] Tom Mens, Mathieu Goeminne, Uzma Raja, and Alexander Serebrenik. 2014. Survivability of software projects in gnome–a replication study. In *7th International seminar series on advanced techniques & tools for software evolution (SATToSE)*. 79–82.
[64] Courtney Miller, Mahmoud Jahanshahi, Audris Mockus, Bogdan Vasilescu, and Christian Kästner. 2025. Understanding the response to open-source dependency abandonment in the npm ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*.
[65] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. 1281–1293.
[66] 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*.
[67] Samim Mirhosseini and Chris Parnin. 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies?. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE.
[68] Martin Monperrus. 2019. Explainable software bot contributions: Case study of automated bug fixes. In *2019 IEEE/ACM 1st international workshop on bots in software engineering (BotSE)*. IEEE, 12–15.
[69] Suhaib Mujahid et al. 2023. Where to Go Now? Finding Alternatives for Declining Packages in the npm Ecosystem. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*.
[70] 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).
[71] Michael J Muller and Sarah Kuhn. 1993. Participatory design. *Commun. ACM* 36, 6 (1993), 24–28.
[72] Brad A Myers, Amy J Ko, Thomas D LaToza, and YoungSeok Yoon. 2016. Programmers are users too: Human-centered methods for improving programming tools. *Computer* 49, 7 (2016), 44–52.
[73] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 2015. *Principles of Program Analysis*. Springer.
[74] npm. 2016. kik, left-pad, and npm. https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm. Accessed: 2022-10-04.
[75] 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.
[76] OpenSSF. [n. d.]. FLOSS Best Practices Criteria (All Levels). https://www.bestpractices.dev/en/criteria Accessed: 2024-03-17.
[77] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2018. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–10.
[78] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. 2020. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering* 48, 5 (2020), 1592–1609.
[79] Donald Pinckney, Federico Cassano, Arjun Guha, and Jonathan Bell. 2023. A Large Scale Analysis of Semantic Versioning in NPM. *Proc. Conf. Mining Software Repositories (MSR)* (2023).
[80] Gede Artha Azriadi Prana et al. 2021. Out of sight, out of mind? How vulnerable dependencies affect open-source projects. *Empirical Software Engineering* 26 (2021).
[81] 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)*.
[82] Steven G Rogelberg et al. 2003. Profiling active and passive nonrespondents to an organizational survey. *Jrnl. of Applied Psych.* (2003).
[83] Benjamin Rombaut, Filipe R Cogo, Bram Adams, and Ahmed E Hassan. 2023. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the Greenkeeper dependency bot in npm. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* 32, 1 (2023).
[84] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.
[85] shelly. 2022. https://twitter.com/codebytere/status/1567437988908392455. Accessed: 2024-03-17.
[86] Sonatype. [n. d.]. Automate your dependnecy management. https://www.sonatype.com/sonatype-developer. Accessed Mar. 2025.
[87] Sonatype. 2023. *9th Annual State of the Software Supply Chain*. Technical Report. Sonatype. https://www.sonatype.com/state-of-the-software-supply-chain/about-the-report
[88] Sonatype. 2024. *10th Annual State of the Software Supply Chain*. Technical Report. Sonatype. https://www.sonatype.com/state-of-the-software-supply-chain/Introduction
[89] Margaret-Anne Storey and Alexey Zagalsky. 2016. Disrupting developer productivity one bot at a time. In *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*. 928–931.
[90] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical lag of dependencies in major package managers. In *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*. IEEE, 228–237.
[91] Synopsys. 2024. *2024 Open Source Security and Risk Analysis Report*. Technical Report. Synopsys. https://www.synopsys.com/software-integrity/engage/ossra/ossra-report
[92] Gareth Terry, Nikki Hayfield, Victoria Clarke, Virginia Braun, et al. 2017. Thematic analysis. *The SAGE handbook of qualitative research in psychology* 2, 17-37 (2017), 25.
[93] Tidelift. 2024. *The 2024 Tidelift State of the Open Source Maintainer Report*. Technical Report. Tidelift. https://tidelift.com/open-source-maintainer-survey-2024
[94] Alexandros Tsakpinis. 2023. Analyzing Maintenance Activities of Software Libraries. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 313–318.
[95] 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)*.
[96] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems* 35 (2022), 24824–24837.

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

[98] Martin Woodward. 2022. Octoverse 2022: 10 years of tracking open source. https://github.blog/news-insights/research/octoverse-2022-10-years-of-tracking-open-source/. Accessed: 2025-03-02.

[99] Xiaoya Xia, Shengyu Zhao, Xinran Zhang, Zehua Lou, Wei Wang, and Fenglin Bi. 2023. Understanding the archived projects on GitHub. In *Proc. Int'l Conf. Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 13–24.

[100] 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)*.

[101] Nusrat Zahan, Philipp Burckhardt, Mikola Lysenko, Feross Aboukhadijeh, and Laurie Williams. 2024. Leveraging Large Language Models to Detect npm Malicious Packages. *arXiv preprint arXiv:2403.12196* (2024).

[102] Ahmed Zerouali et al. 2018. An empirical analysis of technical lag in npm package dependencies. In *Proc. Int'l Conf. Software Reuse (ICSR)*. Springer.

[103] Zhiqing Zhong, Shilin He, Haoxuan Wang, Boxi Yu, Haowen Yang, and Pinjia He. 2025. An Empirical Study on Package-Level Deprecation in Python Ecosystem. In *Proc. Int'l Conf. Software Engineering (ICSE)*.

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