Empirical Software Engineering (2025) 30:164
https://doi.org/10.1007/s10664-025-10703-4

Underproduction analysis of open source software

Kaylea Champion[1] · **Benjamin Mako Hill**[1]

Accepted: 10 July 2025 / Published online: 23 September 2025

Abstract
Free/Libre and Open Source software is widely deployed and serves as digital infrastructure. This software is often developed through peer production, in which contributors select their own tasks. Although much of this software is innovative and well-regarded, in some cases quality and importance may be misaligned, such that highly important software is nonetheless low quality. In this paper, we describe a generalized approach for identifying important but poorly maintained FLOSS packages (Underproduction Analysis), then apply that approach to an important body of digital infrastructure. We validate our approach using alternate indicators of importance, quality, and risk. We then test a series of hypotheses about social and technical factors that correlate with underproduction. We find that older software is more likely to be underproduced, and we are surprised to find that additional maintenance resources are not always associated with lower levels of underproduction. Finally, we examine the community's collaboration network structure, and find that those people working on underproduced packages tend to be those who are more central to the community's collaboration network. These results have important implications both for engineering research seeking to understand outcomes of technical and social decisions, and the software development communities seeking to maintain FLOSS.

Keywords Open source · FLOSS · FOSS · OSS · Mining software repositories · Commons-based peer production · Software quality · Risk · Quantitative methods

✉ Kaylea Champion
kaylea@uw.edu

Benjamin Mako Hill
makohill@uw.edu

[1] Department of Communication, University of Washington, 4109 E Stevens Way NE, Seattle, WA 98185, USA

1 Introduction

Over the last three decades, millions of people working in Free/Libre Open Source Software (FLOSS) development communities have created an enormous body of software that has become digital infrastructure (Asay 2019). FLOSS communities have produced the GNU/Linux operating system, the Apache webserver, programming languages and source control systems, and more (Crowston et al. 2012). Hoffmann et al. (2024) assess the demand-side value of FLOSS as $8.8 trillion, and that firms would need to more than triple their spending on software if FLOSS did not exist. However, this history also includes some notable failure events. One such example is an event that occurred in 2014, when it was announced that the OpenSSL cryptography library contained a buffer over-read bug dubbed "Heartbleed" (Walden 2020). This vulnerability compromised the security of a large portion of secure Internet traffic, resulting in the virtual theft of millions of health records, private government data, and more. OpenSSL provides the cryptographic code that protects the majority of HTTPS web connections, many VPNs, and a variety of other Internet services.

OpenSSL was at risk of an event like Heartbleed because although it was an extraordinarily important piece of software, the architecture had fallen drastically out of step with modern engineering standards and had very little attention and labor devoted to its upkeep (Walden 2020; Eghbal 2016; Wheeler 2014). OpenSSL had been maintained through peer production, a common process in FLOSS where software development work is performed by anyone interested in taking on a particular task (Benkler 2002). For OpenSSL in early 2014, that had involved only four core developers, all volunteers.

How did it come to pass that such an important library was relatively poor quality and lacked a robust development community? How might we detect such problems in the future, before they become major crises? We argue that preventing vulnerabilities like Heartbleed requires a three-part approach: building scientific understanding of the production communities that make digital infrastructure software like OpenSSL possible in the first place, development of methods to measure the risks these communities face, and the creation and evaluation of interventions to remediate those risks. In the following work, we share progress on each of these goals; see Fig. 1 for an overview of the structure.

We first developed a method for measuring underproduction, applying it to the Debian GNU/Linux operating system. We then validated our approach and results from three perspectives: using an alternate indicator of importance, using an alternate indicator of quality, and using an alternate indicator of risk. We then performed a series of hypothesis tests to evaluate the social and technical factors associated with underproduction.

This work seeks to provide empirical guidance to the ongoing process of software evolution and re-engineering, in particular by offering frameworks for thinking about when and where this form of effort is most needed. This paper is an extension of two articles: one previously published as part of the 2021 *IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* (Champion and Hill 2021) and one published as part of *SANER* 2024 (Champion and Hill 2024), both of which also appear in the first author's dissertation. This article reproduces much of the results and text of these previous studies while presenting several new analyses that evaluate the relationship between inequality of contribution and underproduction, conduct new validations of our approach, and expand our discussion of the implications the work has for practitioners.

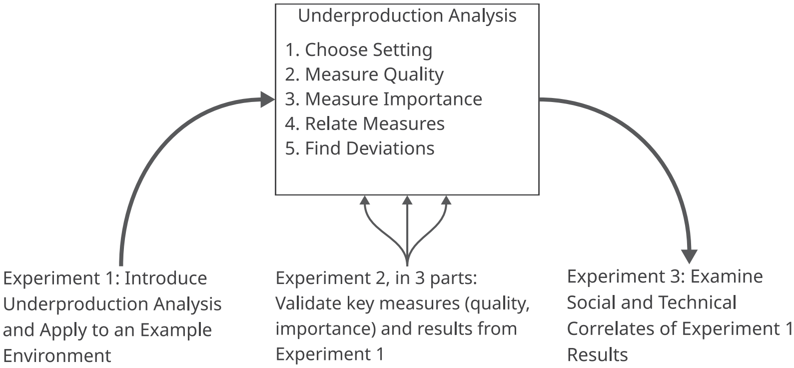

Fig. 1 An overview of the paper

2 Background

We situate our method as an approach to detect and measure software risk which takes insights from features of peer production as a software development strategy.

2.1 Detecting and Measuring Software Risk

Prevention, detection, and mitigation of risk in software development, maintenance, and deployment are the subject of substantial research interest [e.g., Natella et al. 2016; Gritzalis et al. 2018; Meidan et al. 2018]. Risk detection and management techniques examine overall system safety, develop predictive models to prioritize effort (such as reliability growth models), and seek development techniques to make a project less error-prone and more fault-tolerant (Bennett et al. 1996). One line of work in software quality analysis and risk detection seeks to identify issues by locating "bad smells" and antipatterns. This includes code smells (Sobrinho et al. 2018; Santos et al. 2018; Al Dallal and Abdin 2018) as well as architectural smells (ill-considered fundamental design decisions that may trouble the project later) (Le et al. 2018) and community smells (early warning signs of problems in a community). Of particular interest to both software engineering researchers and practitioners are smells that are empirically related to failures (Tamburri et al. 2019).

Various other strategies have been used to measure risk. Code-level metrics look at complexity, change-prone code, or defect-prone code. Other approaches consider the extent to which a codebase takes adequate preventative measures against risk, such as thorough testing (Wong et al. 2005). Finally, multifactor approaches, as in decision support analysis, take a risk management point of view and incorporate organizational factors, management practices, and areas of potential risk throughout the lifecycle of a project (Pasha et al. 2018; Verdon and McGraw 2004; Lyytinen et al. 1998)

2.2 Systematic Comparisons of FLOSS

The process of building and maintaining software is often collaborative and social, including not only code but code comments, commit messages, pull requests, and code reviews, as well as bug reporting, issue discussing, and shared problem solving (Robles et al. 2009). Non-code trace data may include signals of technical debt (Zampetti et al. 2020), signs that a given code commit contains bugs (Falcao et al. 2020), or serve as indicators of committed developers, a high-quality software project, or a healthy and sustainable project community (Dabbish et al. 2012; Coelho et al. 2018; Valiev et al. 2018). Previous research has found that digital trace data capturing online community activity can provide significant insight into the study of software (Dabbish et al. 2012).

These collaborative and social systems offer a data source for understanding both developer team productivity, as in Choudhary et al. (2020) study of "collaboration bursts" as well as for analyzing macro-level dynamics of software production. For example, Gonzalez-Barahona et al. (2014) used the repository of *glibc* as a site to evaluate Lehman's "laws of software evolution" including the law of organizational stability which states that work rates on a system are constant. The team found that the laws are frequently not upheld in FLOSS, especially when effort from outside a core team is considered. This work suggests that the effort available to maintain a piece of FLOSS software may increase as it grows in popularity.

Previous studies have suggested that bug resolution rate is closely associated with a range of important software engineering outcomes, including codebase growth, code quality, release rate, and developer productivity (Abou Khalil et al. 2019; Kim and Whitehead 2006; Michlmayr and Senyard 2006). In contrast, the lack of maintenance activity, as reflected in a FLOSS project's bug tracking system can be considered a sign of failure (Coelho and Valente 2017).

3 Conducting Underproduction Analysis

3.1 Conceptual Framework

Repositories of peer-produced FLOSS are susceptible to what we call underproduction—a concept proposed by several Wikipedia researchers who use the term to describe a dynamic that emerges when volunteers self-select into tasks. In particular, we are inspired by a study of Wikipedia by Warncke-Wang et al. (2015) who builds on previous work by Gorbatai (2011) to formalize what Warncke-Wang calls the "perfect alignment hypothesis" (PAH). The PAH proposes that the most heavily used peer produced information goods (for Warncke-Wang et al., articles in Wikipedia, for us, Free/Libre Open Source Software) should be the highest quality, that the least used will be the lowest quality, and so on.

In other words, the PAH proposes that if we rank peer production products in terms of both quality and importance—for example, in the simple conceptual diagram shown in Fig. 2—the ideal circumstance is one where the two ranked lists will be perfectly correlated. In software terms, if an ecosystem contained 100 packages, full alignment would mean the #1 most important package would be #1 in quality, packages whose important rank is #50 would be #50 in quality, and so on.

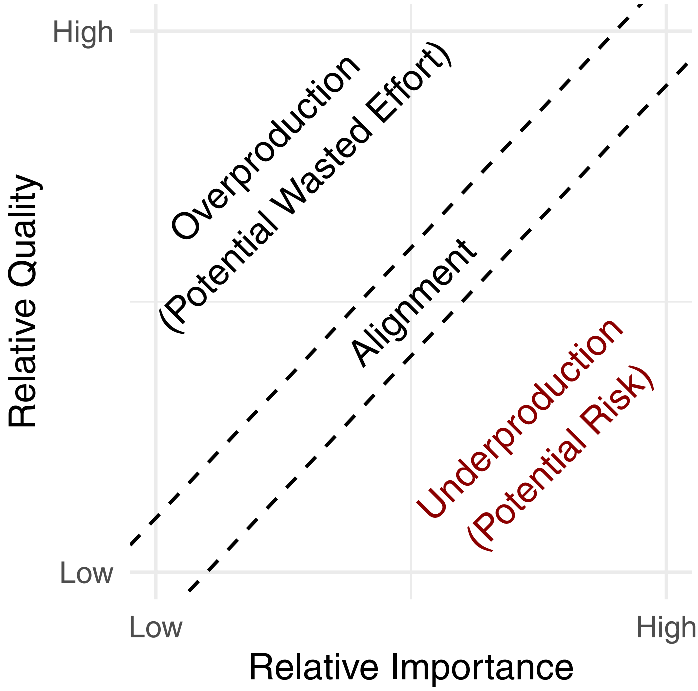

Fig. 2 A conceptual diagram locating underproduction in relation to quality and importance

In Gorbatâi's terminology, misalignment such that quality is high but demand is low results in 'overproduction.' Of greater interest is the case when peer-produced goods are 'underproduced': when demand is high but quality is low. In an economic market, supply and demand are said to be aligned through a price mechanism. Alignment is reached because lower demand decreases prices, which disincentivizes production and returns a market to equilibrium. Because there is no price signal in peer production to bring consumer demands and producer supply into equilibrium, it is unsurprising that peer production projects deviate substantially from the ideals of the PAH (Warncke-Wang et al. 2015).

"Perfect alignment" serves not as a prediction of the relationship between quality and importance, but rather as a baseline from which to identify lacunae in need of attention (Warncke-Wang et al. 2015). Research has sought to characterize the negative impacts on information consumers from divergence from the PAH baseline and to identify sociological processes through which underproduction might emerge (Warncke-Wang et al. 2015; Gorbatai 2011). Self-selection of tasks by volunteers may create a particularly challenging double bind for peer production projects like Wikipedia and FLOSS, driving both creative innovation and underproduction in ways that make alignment of quality and importance a constant struggle.

Despite the central role that FLOSS plays in peer production, we know of no efforts to conceptualize or measure underproduction in software prior to our efforts. We find this surprising for two reasons. First, widespread underproduction seems likely in FLOSS given that FLOSS is characterized by self-selection of software developers into tasks, varying motives among contributors, and the frequent absence of market forces to allocate producers' labor (Lakhani and Wolf 2005; Crowston et al. 2012; O'Neil 2009). Second, the consequences of underproduction are particularly stark in FLOSS, where popular software acts as infrastructure (Eghbal 2016, 2020). A low quality Wikipedia article on an extremely popular subject seems likely to pose much less risk to society than a bug like the Heartbleed vulnerability described earlier which could occur when FLOSS is underproduced. Underproduction in software reflects an important, if underappreciated, type of risk.

This exploration of underproduction also points out its opposite–overproduction. Overproduction may also represent a form of risk, in that resources spent on high-quality, low-importance software may to some extent represent wasted effort, or at least diminishing returns, particularly under conditions of resource constraint. However, the effort spent on maintaining such software may not be easily redirected, given differences in preference or skill among the available volunteers.

3.2 The Underproduction Analysis Method

Our approach to detecting underproduction in software is composed of five steps, adapted and extended from Warncke-Wang et al. (2015). We frame these in general terms here; in sections that follow, we will illustrate the approach, including using alternate measures and conducting validation.

1. Assemble a collection of software artifacts that can be consistently measured as described below. These might be software packages, modules, source files, etc.
2. Identify one or more measures of quality that can be recorded consistently and independently (perhaps repeatedly) across each software artifact in the collection.
3. Similarly, identify a measure of importance that can be recorded consistently and independently across the collection.
4. Propose an *ex ante* theoretical baseline relationship between the two measures that reflects alignment according to the values of the community or the priorities laid out in the research question. This might involve any number of assumptions about that ideal relationship.
5. Measure deviation from this theoretical baseline across artifacts.

The measure of deviation resulting from this process serves as our measure of misalignment between quality and importance (i.e., over- or underproduction).

In a sense, our conceptual approach involves laying out software packages in dimensions similar to those shown in Fig. 2, empirically identifying an ideal relationship that reflects general alignment (i.e., the zone in the lower left to upper right diagonal), and then measuring the deviation from that ideal. This basic conceptual framework can incorporate any number of ways of measuring quality and importance—both areas of active work in software engineering research. Our approach can be carried out using a range of techniques for identifying alignment including entirely nonparametric rank-based approaches (as we demonstrate in Experiment 1), machine learning-based ordinal categorization, or parametric regression-based techniques.

3.3 Ethics

These experiments were conducted with data made public by the Debian community and do not involve any interaction with human subjects. This type of research using these data has been reviewed by the IRB at the authors' institution and has been determined not to be human subjects research. However, by working with observational data, this work does remove data from its original context for analysis. That is, although bug reports and maintenance work are all publicly-viewable records published on the web by Debian, we collected and parsed them, and we acknowledge that these bug reports and maintenance activities were not done with our research project in mind. Therefore, our publication does not include information that would identify Debian contributors. We also sought advice from Debian developers on data use and interpretation. We presented the results of Experiment 1 at the Debian community conference as part of formulating Experiment 3.

4 Experiment 1: Identifying Underproduced Software

In this experiment, we apply our method from § 3 to the FLOSS distribution Debian GNU/Linux and report the results.

4.1 Empirical Setting

The first step of Underproduction Analysis involves assembling a collection of software artifacts. We draw our collection from the Debian GNU/Linux distribution which acts as the empirical setting for all of our experiments. GNU/Linux distributions are collections of software that have been integrated, configured, tested, and packaged with an installer. The contributor community producing the distribution focuses primarily on the production of packages and package management tools for managing the installation and updating of software products produced by others. Distributions like Debian play an important role in the FLOSS ecosystem and are the way that the vast majority of GNU/Linux users install operating system software as well as most applications and their dependencies. With a community in operation since 1993, Debian is widely used and is the basis for other widely-used distributions like Ubuntu. Debian had more than 1,400 different contributors in 2020[1] and contains more than 20,000 of the most important and widely used FLOSS packages.

Debian provides detailed and consistently measured longitudinal data on all its packages and maintainers in the form of released databases, datasets, and APIs (Caneill et al. 2016; Nussbaum and Zacchiroli 2010; Hindle et al. 2010). A body of research in software engineering has used open data from Debian to understand a range of software development practices. The Debian distribution has served as a basis for applying techniques to detect and mitigate defects (Chen and Wagner 2007; Michlmayr and Senyard 2006), predict bugs and vulnerabilities (Pati and Shukla 2014), detect the evolution of package incompatibilities (Claes et al. 2015), predict component reuse (Spaeth et al. 2008), demonstrate code clone detection techniques (Cordy and Roy 2011), develop generalizable QA techniques for complex projects (Nussbaum and Zacchiroli 2010), investigate package dependencies (Galindo

[1] https://contributors.debian.org/ (Archived: https://web.archive.org/web/20201107231239/https://contributors.debian.org/)

Duarte et al. 2010), and as an example of an information processing network (Villegas et al. 2020).

4.2 Application of Framework

We apply our framework using several measures grounded in and sourced from the Debian community itself: its role in the software supply chain as a packager and integrator of software developed by others, the standards and processes in use by its community, and the data it gathers. This provides three advantages. First, we scope the otherwise wickedly complex problem of maintaining and securing the software supply chain to a specific site where a coherent analysis can be conducted. Second, we hope by using Debian-derived measures our approach will have practical value and implications for the community of practitioners from whom much of our data is derived. Third, given that peer production communities often function in a self-organizing and self-assigning fashion, we propose that working within the standards and norms of the community itself may make those implications more likely to be taken up.

4.2.1 Step 1: Assemble a Collection of Artifacts

Our unit of analysis is the Debian *source package*. Source packages are the format that Debian package maintainers modify and publish, but they are not used directly by end-users. Instead, source packages are built by computers in a Debian network of "build daemons" into one or more *binary packages* that may or may not include architecture-specific executables. These binary packages are then distributed to end-users and installed on their computers. Debian also provides tools to allow users to download and build their own binary packages from the corresponding source packages. A single source package may produce many binary packages. For example, although it is an outlier, the Linux kernel source package produces up to 1,246 binary packages from its single source package (most are architecture-specific subcollections of kernel modules).

The one-to-many relationship between source and binary packages presented a challenge for our analysis. Although our chosen measure of quality (bug resolution, described in §4.2.2) uses information stored at the level of the source package (Davies et al. 2010), our chosen measure of importance (installations, described in §4.2.5), is aggregated at the binary level. To map source packages to binary packages, we used the Debian snapshot database's public APIs[2] to identify all binary packages produced by all versions of every source package. All source packages for which Debian had historical quality data and had importance data at our point of analysis (June 6, 2020) are included in the scope of this analysis. To prevent double-counting, we include only the importance data for the most-installed binary within each source package.

4.2.2 Step 2: Identify a Measure of Quality

The second step of our framework involves identifying a measure of quality for each Debian source package. Quality in software is difficult to measure and common strategies for measuring quality include analyzing bug counts [e.g., Ray et al. 2014] or assessing code internal

[2] https://snapshot.debian.org/ (Archived: https://perma.cc/HQW7-R4Y2)

design using a series of heuristics [e.g., Santos et al. 2018]. However, software engineering researchers have noted that the number of bugs reported against a particular piece of FLOSS may be more related to the number of users of a package (Herraiz et al. 2011; Davies et al. 2010), or the level of effort expended in finding bugs (Walden 2020) in ways that limit its ability to serve as a clear signal of software quality. In fact, Walden (2020) found that OpenSSL had a lower bug count before Heartbleed than after. Walden (2020) argued that measures of project activity and process improvements are a more useful sign of community recovery and software quality than bug count.

Additionally, techniques to assess codebases using code design heuristics are not oriented to the work of a distribution development community. The primary focus of work in a community like Debian is on configuring and testing packages for interoperability, rather than making changes to internal code design. In applying our method to Debian for this study, we follow a series of authors who have argued for a focus on a community's effectiveness at resolving the inevitable issues that arise instead of artifact-focused measures (Eghbal 2020; Ronchieri and Canaparo 2018; Adewumi et al. 2016; Crowston et al. 2012; Ruiz and Robinson 2011; Aksulu and Wade 2010). Specifically, our measure of quality is the speed at which bugs are resolved. We treat the difference between opening and closing times as the *time to resolution*. Time to resolution has been cited as an important measure of FLOSS quality by a series of software engineering scholars (Eghbal 2020; Abou Khalil et al. 2019; Crowston et al. 2012; Kim and Whitehead 2006). Although there may be many reasons for protracted resolution time in a distribution (e.g., maintainer skill, task complexity, report quality, lack of resources, bugs in the underlying software package causing packaging problems), the unresolved bug still represents an issue for the distribution's maintainer community and a problem for end users, whatever the reason.

Calculating this measure requires interpretation of bug reports and resolution data which Debian tracks using a database where each transaction takes the form of a specially formatted e-mail message. We extracted comprehensive data on bugs from the Debian Bug Tracking System which is also referred to as "debbugs" or the "BTS." After obtaining an archival copy of all bugs from the BTS, we queried the Ultimate Debian Database (Nussbaum and Zacchiroli 2010) to map bugs to packages. We parsed all actions (i.e., e-mail messages) associated with each bug into columnar data for analysis. Using BTS data, we identified the date and time when each bug was opened and closed. We treated the marking of a bug as "closed," "forwarded" (i.e., not solved by the maintainer but rather referred to the "upstream" development team for the package itself and therefore no longer Debian's responsibility), or "merged" (i.e., designated as a repeat report of an issue already in the database) as closed.

Our approach differs from the approach taken by Zerouali et al. (2019b) who measure bugs as closed based on when they are last modified. We diverge from their method because the final modification to a bug typically occurs when a bug is archived through an automated process that occurs about 30 days after closure. In our examinations of the database, we found that this process can be inconsistent and that unarchiving can occur as a function of administrative processing as well as due to true reopening of a bug.

4.2.3 Controlling for Bug Severity

A limitation relating to our measure of quality is related to the fact that not all bugs require equal attention. In recognition of this fact, the person submitting a bug in Debian assigns it *severity*, which can be modified by the package maintainer or others.[3] Severity in Debian is one of the following categories: wishlist, minor, normal, important, serious, grave, and critical (in that order) with normal as the default.[4] We extracted severity for every bug from the BTS to use as a control. We treat the 4,559 bugs where the severity is "not set" and the 2,239 bugs where the severity is listed as "fixed" as priority "normal."[5] We omit all "wishlist" severity bugs (124,961) because in addition to being used to track bugs that are "very difficult to fix due to major design considerations," this category may be used for a wide range of non-bug "to-do list" items.

Although there may be many other reasons for extended resolution time (e.g., skill, resources, inherent difficulty of the task), we do not include controls for these considerations. This is because our goal is to capture variation in resolution rate due to a range of reasons, which themselves may correspond to a level of risk.

Figure 3 shows non-parametric Kaplan-Meier survival curves for bugs of differing severities. These curves depict the probability of a given bug going from unresolved to resolved in days since it was filed. We observe in this plot that the curves are tightly clustered during the first few days after a bug is reported: about half of all bugs, regardless of severity, are solved shortly after they are reported. Bugs that are serious, grave, or critical are considered "release-critical" and must be fixed for the next "stable" release of Debian. Release-critical bugs include security problems, bugs that make packages unusable, or licensing requirements that are incompatible with Debian.[6] We observe that the curves in Fig. 3 cluster into two general shapes that correspond to release critical and non-release critical bugs, and that release-critical bugs are fixed more quickly.

4.2.4 Modeling *Time to Resolution* Using Bayesian Survival Models

Measuring quality as *time to resolution* poses two analytic challenges. The first challenge is the fact that a substantial portion of bugs remain unresolved at the point of data collection. Because bugs languishing in an open state is precisely the type of risk we hope to measure, omitting bugs that are open at the time of data collection would underestimate how quickly bugs are resolved. Following recent work in software engineering (Abou Khalil et al. 2019), we incorporate data on unresolved bugs by modeling the bug closure process directly using Cox proportional hazard models.[7]

[3] https://www.debian.org/Bugs/Reporting

[4] https://www.debian.org/Bugs/Developer#severities

[5] The "fixed" severity was used from 2000 to 2002 to indicate a non-maintainer upload and then deprecated in favor of a "tagging" approach for this type of fix.

[6] https://www.debian.org/Bugs/Developer (Archived: https://perma.cc/MX3T-36PP)

[7] Cox models are a type of survival model developed originally in epidemiology to estimate how a behavior or treatment might prolong or shorten patients' lives while incorporating data from individuals for whom data is censored (i.e., individuals who are still alive at the conclusion of data collection). In our case, data is censored for any open bug.

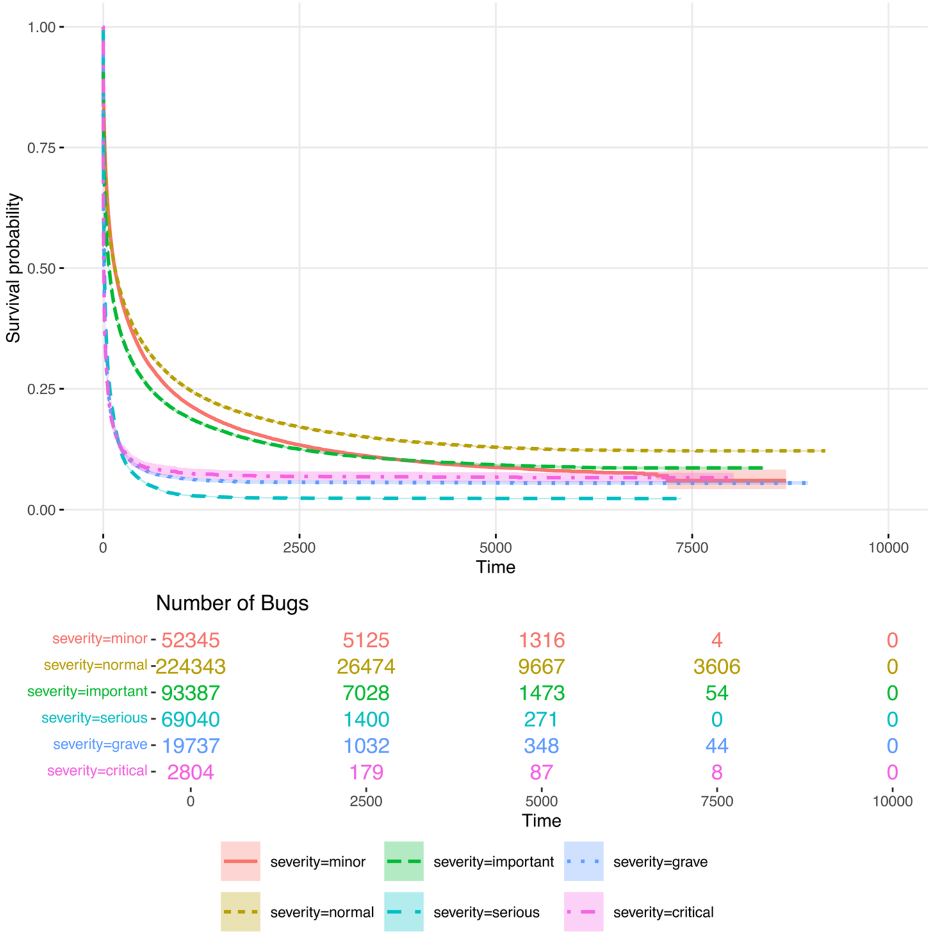

Fig. 3 A Kaplan-Meier curve that shows the number of bugs of different severities that remain open over time

A second challenge in measuring quality as *time to resolution* comes from the fact that the distribution of bugs across packages is highly unequal. Most of the packages we examine (14,604 of 21,902) have 10 or fewer bugs, and more than one out of six (3,857 of 21,902) have only one bug reported. In response, we use Bayesian hierarchical models fit using Stan. The intuition behind this choice is that when a package has few bugs, the overall problem resolution rate across Debian is more informative than the package's small number of data points. In general, the Bayesian approach allows us to become progressively more confident in an estimate when more bugs have been filed against a package.

Our approach finds support in the argument made by Ernst (2018) who elaborates the value of Bayesian hierarchical modeling as a method to carefully distinguish local and global characteristics in studies of software repositories. Our approach offers a "partial-pooling" model that allows us to both take advantage of the structure of our data (bugs are clustered within packages) and to update our prior assumption about expected resolution rates based on the new information that each bug supplies.

We incorporate our control for *severity* into a survival model of the following form, where bugs are the unit of analysis and where λ reflects the hazard function that captures whether bug *k* in package *j* will be resolved at time *t*. We represent the severity of each bug as x_{jk} and use z_j to reflect the log of the package-level random effect for package *j*.[8]

$$\lambda(t; x_{jk}; q_j) = \lambda_0(t) exp(\beta x_{jk} + q_j)$$

Our measure of quality (q_j) is the package-level random effect of the posterior distributions in our survival models. We estimate this quantity using 4,000 independent draws from each package's posterior distribution using Stan. We take 95% credible intervals of these empirical distributions to reflect uncertainty.

4.2.5 Step 3: Identify a Measure of Importance

Our third step involves identifying a consistent measure of importance for every artifact in our collection. In FLOSS contexts, importance can be measured as attention to hosting sites, number of active users, intensity of use, dependency networks, or criticality of function (e.g., life-safety systems) (Wiggins et al. 2009; Eghbal 2020; Stergiopoulos et al. 2016). We measure importance using data from the Debian "Popularity Contest" or "Popcon" application. Popcon is an opt-in survey that shares anonymous data from volunteer systems back with Debian. Popcon data has been used in a range of previous studies in software research (Wiggins et al. 2009; Herraiz et al. 2011; Davies et al. 2010). Popcon is particularly applicable in our case because it is a signal which Debian itself has developed and deployed and because it is displayed in multiple locations on Debian's maintenance platforms. One important limitation of Popcon data is that despite the millions of systems running Debian, only a fraction have opted to report installation data.

Our study includes data from 201,484 systems from a single-day snapshot on July 6, 2020, and only includes packages for which usage was being reported at that time. Popcon includes two measures: *inst* for *installation* (i.e. the presence of a package on a machine), which serves as our measure of importance for subsequent analysis (i_j, for each package *j*); and a measure called "vote" which attempts to capture if packages are being used. In our analysis, we use *installation*.

Unfortunately, because Popcon reports installations at the binary level, we cannot use this data to distinguish whether a single individual reported the installation of multiple binaries associated with a given source package. To avoid double counting, we use the binary-source mappings described in §4.2.1 to set i_j to the highest install count among all binary packages associated with a source package *j*. As a result of this construction, our installation measure is necessarily conservative, but we can be assured that a source package was installed at least i_j times.

4.2.6 Step 4: Select a Baseline Relationship

The fourth step in our conceptual framework involves comparing measures of quality and importance to some ideal baseline relationship. For our baseline, we take a nonparametric ranking approach that is similar to, but more granular than, the approach taken in Warncke-

[8] We use notation for random effects survival models drawn from Sargent (1998).

Wang et al.'s Wikipedia analysis which uses Wikipedia quality categories (Warncke-Wang et al. 2015). Our baseline definition of alignment is when the relative rankings of importance and quality are the same ($rq_j = ri_j$). We treat a ranking of 1 as describing the worst observed quality or the lowest number of installations, while a rank of 21,902 represents the highest.

4.2.7 Step 5: Measure Deviation

The final step of our conceptual framework involves measuring deviation from our theoretical baseline. We describe our measure of alignment as the "underproduction factor" (U_j) that we measure as the log of the ratio of rankings of importance and quality:

$$U_j = \log \frac{ri_j}{rq_j}$$

Given this construction, U_j will be zero when a package is fully aligned, negative if it is overproduced, and positive if it is underproduced. This approach means that the range of U_j is a function of repository size. In our dataset of 21,902 source packages, the range of $U_j = [-10, 10]$, where $U_j = 10$ for a theoretical maximally underproduced package where $ri_j = 21,902$ and $rq_j = 1$.

Although constructing U_j for a single value of q_j is straightforward, incorporating uncertainty into our measure of quality requires additional work. Because posterior draws in Stan are independent, we can incorporate uncertainty in our measure of quality by computing U_j using the quality ranking of each of the 4,000 posterior draws taken from the estimated random effect for each package j. U_j reported in our analysis reflect the 95% credible intervals from U_j computed separately for these draws. Because we only have a single measure of installation i_j, we do not attempt to incorporate uncertainty in this measure into our analysis.

4.3 Results from Experiment 1

The application of our method described in §4.2 suggests that a minimum of 4,327 packages in Debian are underproduced (20%).

Our approach provides evidence that underproduction is widespread in Debian. Figure 4 shows 95% credible intervals (CIs) for all 21,902 packages. We describe packages whose 95% credible interval for U_j includes zero ($0 \in U_j$) as "aligned" and packages where both ends of the credible interval have the same sign as "overproduced" ($U_j < 0$) and "underproduced" ($U_j > 0$). The wide credible intervals for many of the packages shown on the left panel of Fig. 4 reflect the fact that many of the aligned packages may be misaligned packages with high variance. The noise in our measures of q_j and U_j is partially attributable to the fact that many packages have few bugs.

Figure 5 displays a heatmap visualization showing the number of packages occupying a range of installation ranks (ri_j) along the x-axis and quality ranks (rq_j) along the y-axis in evenly spaced bins. In this way, this nonparametric data visualization seeks to reflect the basic intuition that goes into the construction of our measure of underproduction U_j. Because rq_j is a distribution, the figures show a ranking of the mean value of q_j. The top panel shows all packages in Debian and clearly shows data spread across the heatmap rather

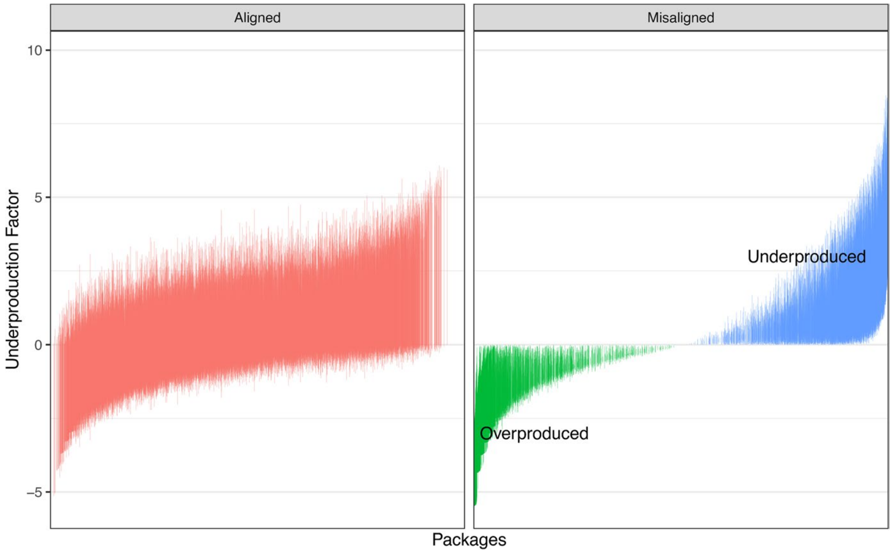

Fig. 4 Each line represents the credible intervals for U_j for a package in Debian. We treat all packages whose CIs include zero as "aligned"; those whose CIs are entirely above 0 are labeled "underproduced;" those whose CIs are entirely below zero are labeled "overproduced"

than clustered along the diagonal. The relative lack of density along the diagonal in the top figure is evidence that misalignment in Debian is widespread. The relatively high density of packages on the lower right indicates that underproduction in Debian is common.

The lower panel of Fig. 5 shows only packages for which the 95% credible interval excludes zero (i.e., the misaligned packages shown on the right panel of Fig. 4). Removing aligned packages reveals much more density among underproduced packages relative to overproduced packages. Removing 'aligned' packages visibly hollows out the center of the heatmap filled with packages of moderate quality because many of these packages have wide CIs. However, although there are almost no packages of middling quality that we can confidently say are overproduced, there are many that we can confidently say are underproduced. Further, there is substantial density—hundreds of packages—in the extreme bottom right corner. This area contains packages that are almost maximally underproduced according to our measure.

In Fig. 6, we present a list of the packages displaying the highest levels of underproduction (mean U_j) with per-package boxplots drawn from the posterior distribution associated with each package.

5 Experiment 2: Validation Using Alternate Indicators

To validate our approach, we consider that there are three components of our approach in need of scrutiny—our method relies on measuring importance, on measuring quality, and on an argument that the regulation of this relationship is connected to risk. In our validations, we tackle each of these in turn with empirical analysis.

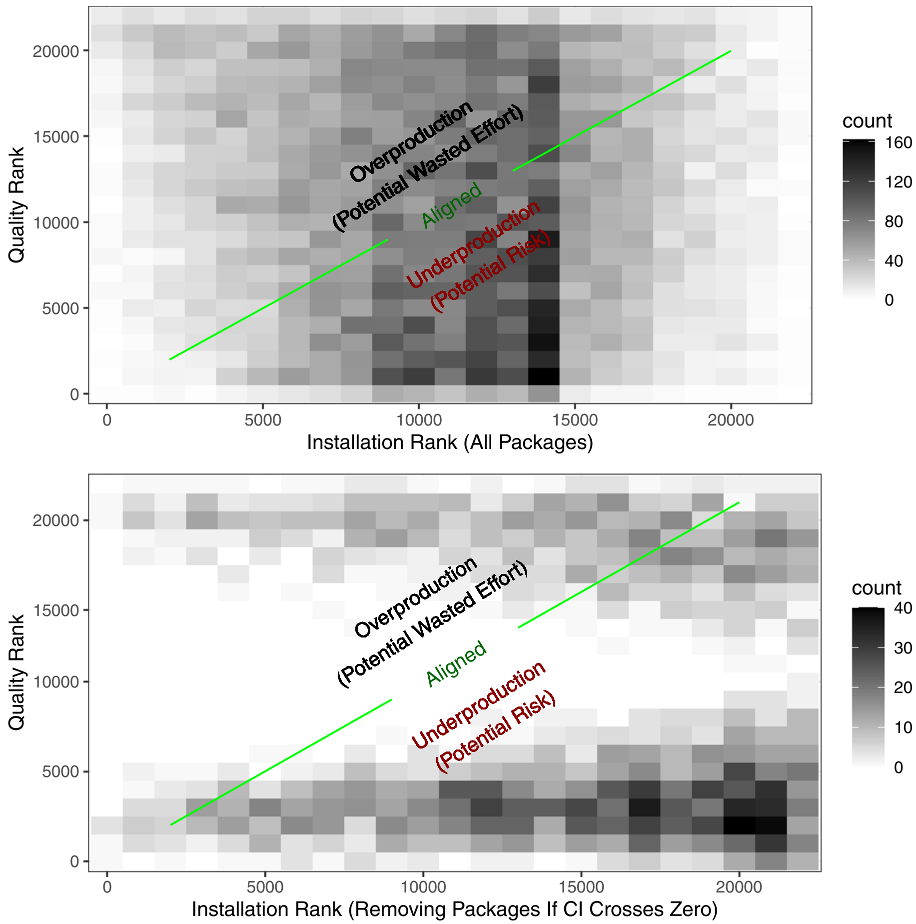

Fig. 5 A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and installation. Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero

We first compare the results of Experiment 1 with an Underproduction Analysis using an alternate operationalization of importance. Next, we examined the relationship between the results from § 4 and an alternative operationalization of quality. We then compared our results with an alternative indicator of risk akin to underproduction.

5.1 Analysis with an Alternate Measure of Importance

In §3, we described underproduction measurement in FLOSS as a five-step method that includes the identification of a measure of importance. In our illustration of the measure in §4.2.5, we offer an example of *installation count* as reported by the Debian project's Popcon tool as a measure of importance. Of course, installation count is not the only possible

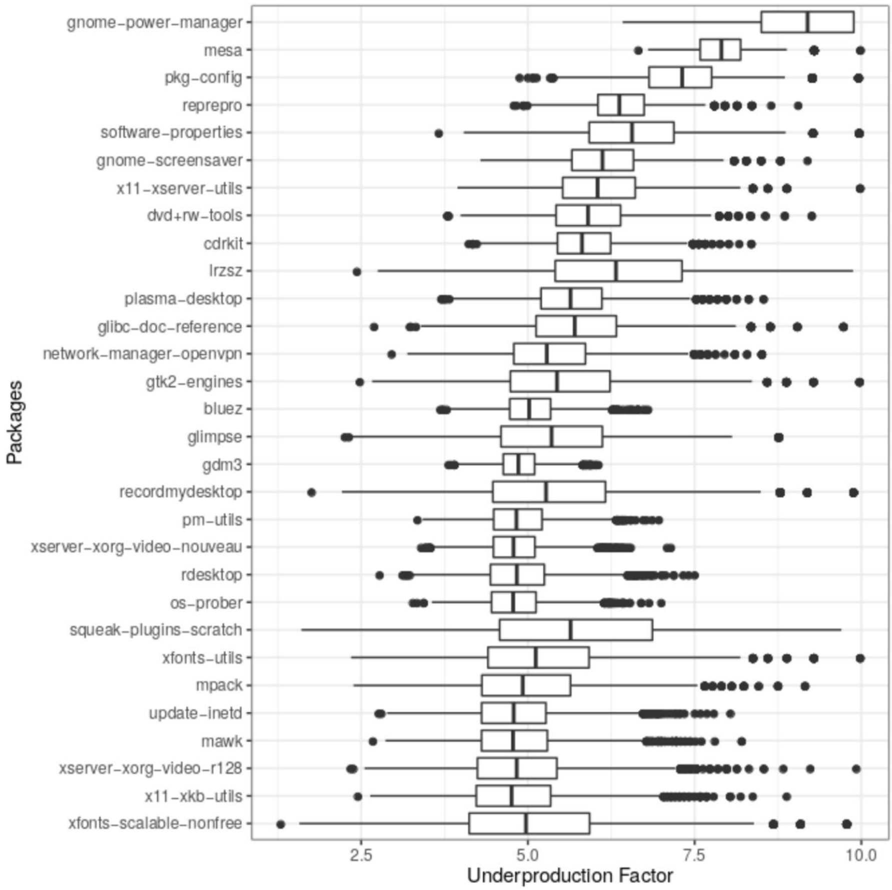

Fig. 6 Packages displaying the highest mean levels of underproduction. Boxplots show the mean and interquartile range of our distributions of U_j and reflect uncertainty in our model of package-level quality

measure of importance, or even the only measure of importance that can be derived from Popcon.

In addition to installation count, Popcon also reports a measure called *vote*. Popcon documentation explains that "A computer 'vote's for a package if according to the data provided in the report, a program provided or depending on the package was used less than thirty days ago."[9] However, one limitation of this measure is that the calculation of vote uses a comparison of a file's *atime* (access time) and *ctime* (creation time).[10]

A file's atime updates when the file is read or run, including not only direct usage, but also interactions occurring as part of some other process, e.g., some system backups or updates of the locate database. In addition, files or filesystems can be set not to update atime at all as a way of increasing performance. Hence, although the intention is for vote to indicate usage, it has the disadvantage that it relies on atime, which may both under- and

[9] https://popcon.debian.org/FAQ (Archived: https://perma.cc/2VXC-47YD)

[10] https://popcon.debian.org/README (Archived: https://perma.cc/4S3D-H2RU)

overcount usage in different situations. That said, the 30-day observation span and Debian's dependency analysis, although limited to dependencies of which Debian is aware of, both serve to mitigate this limitation: vote as a measure is widely used in the Debian community alongside installation and offers another measure of package importance.

To illustrate the results of applying our method using an alternate measure of importance, we followed the same process described in §4.2, substituting *vote* for *inst* in §4.2.5 as appropriate. Figure 7 illustrates a finding of substantial underproduction. The heatmap in Fig. 8 indicates concentrations of misaligned packages. The boxplot in Fig. 9 lists the most at-risk packages under this analysis.

We observe that the results of this analysis are similar—underproduction by this measure is likewise widespread, and in fact, in Fig. 9, the same 11 packages appear at the top (i.e., those with the worst underproduction risk) in both lists. However, the ordering beyond the 11th most badly underproduced is slightly shuffled. We also observe that the package "glibc-doc-reference" and "xfonts-scalable-nonfree" drop out of the 30-worst list when usage ("vote") is the measure of importance instead of installation, and "xserver-xorg-video-vesa" and "gpm" are added. These two importance measures are highly correlated, with installation taking all packages into account, and vote including only those that register atime changes and have been run sufficiently recently such that atime is updated.

5.2 Underproduction Analysis Using an Alternate Measure of Quality

We offer a further validation of our approach by demonstrating the use of the existence of bugs as a quality indicator.

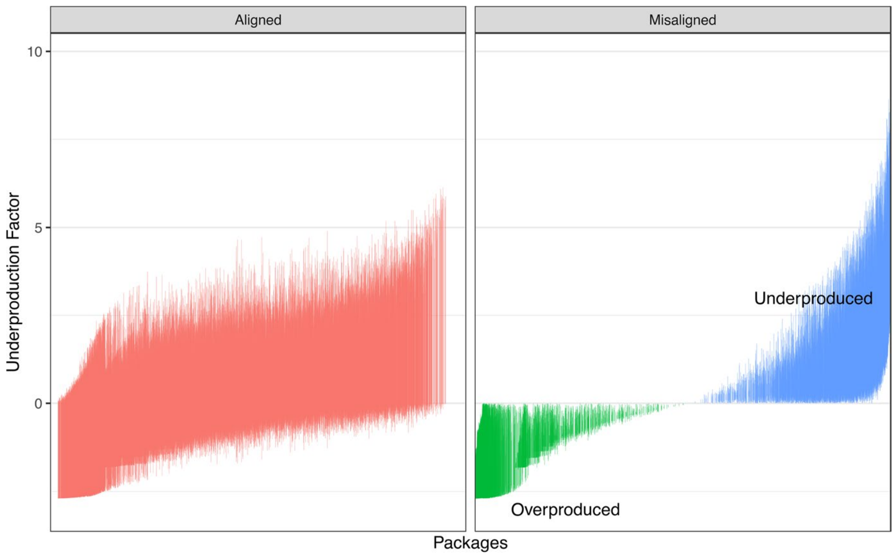

Fig. 7 Credible intervals for U_j for every package in Debian using "vote" (recent usage) as the measure of importance. We treat all packages whose CIs include zero as aligned; those whose CIs are entirely above 0 are labeled 'underproduced;' those whose CIs are entirely below zero are labeled 'overproduced'

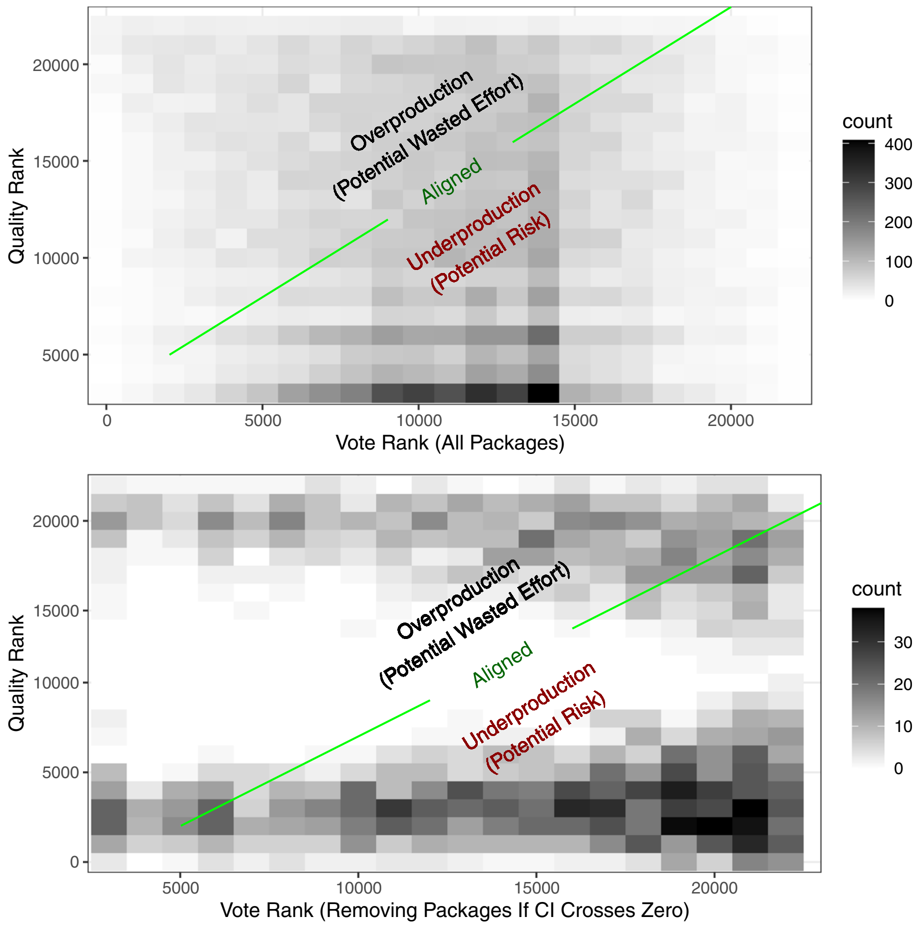

Fig. 8 A heatmap of software alignment. Color intensity indicates the number of packages occupying a given ranking of quality and importance ("vote"). Aligned packages appear along the lower-left to upper-right diagonal. The top heatmap includes all packages, while the bottom heatmap contains only those packages for which the 95% credible interval does not cross zero

5.2.1 Methods—Underproduction and Build Process

The CoLiS platform developed by Becker et al. (2022) analyzes the shell scripts Debian maintainers write to manage the installation process. CoLiS examines these scripts using syntactic analysis, semantic analysis, and symbolic execution to check compliance with Debian policy and technical validity (e.g., use of valid command line arguments) (Becker et al. 2022). The CoLiS project examined 27,324 scripts in 11,640 packages as of August 14, 2021, and made their data available in the form of 152 bug reports to the Debian project.

For **Step 1: Assemble a collection of artifacts**, we used the packages analyzed by Becker et al. (2022), the packages distributed by Debian in the August 14, 2021 release (nicknamed *bullseye*). We obtained the list of packages from the Ultimate Debian Database (Nussbaum and Zacchiroli 2010).

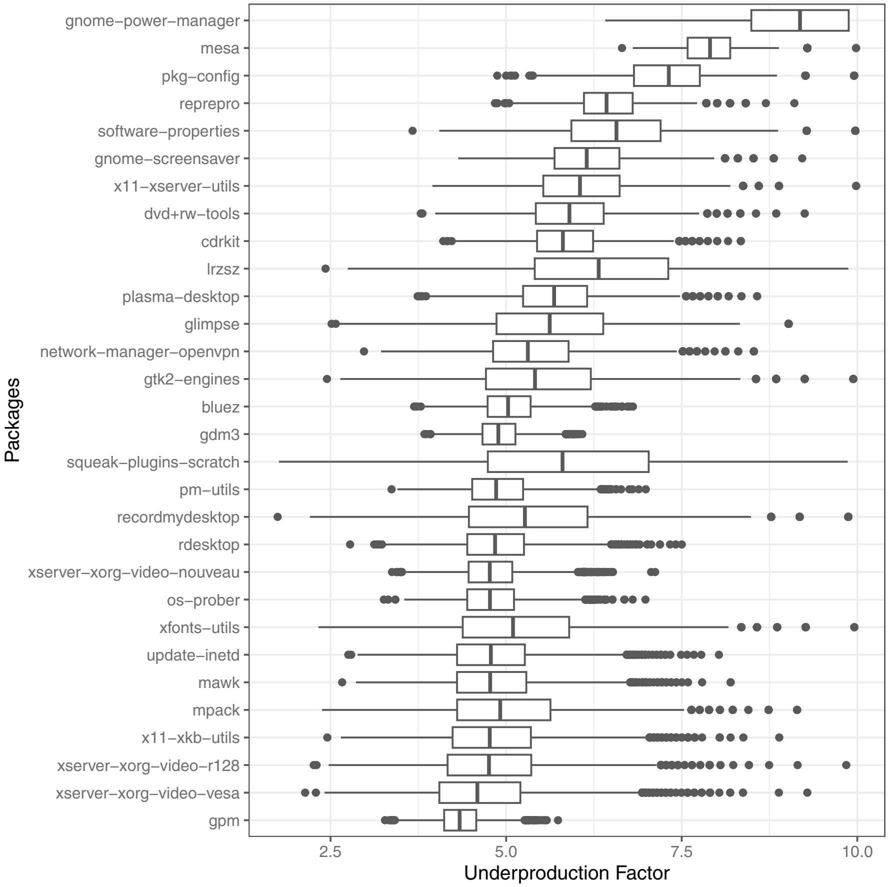

Fig. 9 Packages displaying the highest mean levels of underproduction using "vote" as a measure of importance. Boxplots show the mean and interquartile range of our distributions of U_j and reflect uncertainty in our model of package-level quality

For **Step 2: Identify a measure of quality**, we examined the details found in Becker et al. (2022), and the bug report data at the URL they publish (tagged with CoLiS and the name of the article's senior author). We observe that the Debian community triaged these bugs by assigning them with a severity (wishlist, minor, normal, and important).[11] We assign an overall score to each severity level—having no bug gives a quality problem score of -50, wishlist bugs are 0, minor are 1, normal are 10, and important are 50. Package-level scores are sums of the severities of all of the bugs found in the package. Other ways of translating these categories into metrics may yield different results.

To perform **Step 3: Identify a measure of importance**, we used the install count (*inst*) derived from *popcon* as described in § 4.2.5. This feels justified because everyone who installs a package with a CoLiS bug would be exposed to potential unwanted effects from the installation script. Becker et al. (2022) examines the state of the Debian project at the

[11] Debian has additional levels of severity—serious, grave, and critical—but none of the reported bugs have these severities.

point of release. Debian releases new versions approximately every two years, and we would expect that users might not update their software immediately. We select a time point six months after the release of *Debian bullseye*: February 15, 2023, and call the Popcon API for the total number of installations of each package.

Mapping the bugs in the maintainer scripts reported in Becker et al. (2022) with popcon data by the name of the binary succeeded in all but 25 cases. In 17 of those cases, the package had been removed from Debian, so these were dropped from the analysis. In two cases, the package with the bug was in an overarching source package (src:shibboleth-sp and src:freedict); we dropped these from analysis to maintain our unit of analysis at the binary level. In five cases, the bug was reported against a binary that did not have any popcon usage (lilo, debbugs, uim-common, uim-qt, websimba). In one case (notmuch-emacs), the package had been renamed (to elpa-notmuch), and we assigned the bug to the new name. In all, this approach brings our total to 152 bugs in 143 binaries, where some binaries were impacted by more than one bug.

The cases where Becker et al. (2022) found bugs in *Debian bullseye* which did not correspond to any installations reported via popcon are potentially illustrative examples that point out limitations in our approach: in particular, the *debbugs* package used inside of Debian's own infrastructure for bug management and the *lilo* bootloader. Both of these packages are "internal," but in very different ways. Debbugs is the software platform Debian runs on its own server farm to track its own bugs, i.e. it is internal to the development community itself, and serves as an enabling factor for Debian's development community. It is a crucial part of how Debian gets work done, but there is no particular expectation that the general public would run and use it. A bootloader is stored in system firmware and is invoked as part of system startup, and thus runs before the operating system. Therefore, it is perhaps not surprising that a bootloader might not show up in a package installation and usage survey like popcon (indeed, *grub*, the default bootloader in Debian, also does not indicate any usage via popcon); a bootloader must be installed before anything else can be installed (including popcon), and its usage cannot be recorded into the filesystem before the filesystem has been loaded, i.e. it is internal to the running of the system itself, and serves as an enabling factor for bridging the hardware-software divide. This suggests hints at potential for what Meyers et al. (2022) calls "software dark matter"—software on which our systems depend, either because like debbugs, they are serving as supporting infrastructure for the infrastructure organization itself, or because they are low-level, like the bootloader. These internal connecting and enabling pieces are critical, but may be less visible and more easily forgotten, or be difficult to track in the same way as other software.

To address **Step 4: Select a baseline relationship**, we reason, as we did in § 4.2.6, that it seems most preferable for installation script bugs to occur among the least used packages, scaled by the severity of the bug.

For **Step 5: Measure deviation**, we take the same approach as with § 4.2.7, taking the log of the ratio of rankings of importance and quality:

$$U_j = \log \frac{ri_j}{rq_j}$$

5.2.2 Results of Underproduction Analysis Using Build Process Bugs

We present the top 25 most underproduced packages in Table 1.

The impacted binaries vary widely. Some are clearly infrastructural—samhain is an intrusion alerting tool, guidedog is used to set up a system to serve as a packet router and port forwarder, unscd is a name service caching daemon, ceph-osd and ceph-mon are part of the scalable distributed ceph storage system. We also see underproduction in more user-facing tools. Binaries from UIM (Universal Input Method, providing support for multilingual user inputs) appear frequently in the top 25 results—in addition to a CoLis-identified bug in a core library, several of these binaries had their own CoLis-identified bugs. We also see bugs in eight of Debian's own reference guides, in the Java bindings and dev tools for the svn source code manager (libsvn-java and libsvn-dev), in the e-mail indexing search tool elpa-notmuch, in the scientific fonts package fonts-stix, in the cxref documentation generator, in the plotting libraries gnuplot-qt and gnuplot-x11, and in linpac—used by ham radio operators.

Finally, we treat this as an opportunity to further validate our approach in Experiment 1. How well is a later finding of underproduction associated with a prior finding, using these different operationalizations of quality? To examine this, we test the correlation between the underproduction factor from Experiment 1, measured using mean time to resolution since the date when the package was added to Debian and usage on July 6, 2020, to the underproduction factor in this subsection of Experiment 2, measured using presence of build script

Table 1 The 25 most underproduced packages in Debian, measuring quality based on the severity of bugs present in installation scripts

Package	COLIS Underproduction Factor
samhain	8.52
cxref	8.35
linpac	8.13
guidedog	7.73
debian-reference-common	7.18
fonts-stix	7.17
gnuplot-qt	7.15
gnuplot-x11	7.14
debian-reference-it	7.14
debian-reference-en	7.08
uim-gtk2.0	7.07
uim-gtk3	7.06
uim-qt5	7.06
libsvn-java	7.05
unscd	7.04
ceph-osd	7.01
ceph-mon	7.00
debian-reference-de	6.94
debian-reference-fr	6.94
debian-reference-pt	6.91
debian-reference-ja	6.91
debian-reference-zh-cn	6.91
debian-reference-zh-tw	6.91
libsvn-dev	6.90
jags	6.86

Table 2 Predicting non-maintainer upload (NMU)

Intercept	-3.03^*
	$[-3.10; -2.95]$
Mean U_j	0.47^*
	[0.40; 0.53]
Num. obs.	21902

* 0 outside the confidence interval.

bugs at time of release on August 14 2021 and usage as of February 15 2022 using Pearson's product-moment correlation. In other words, we examine the relationship between an underproduced source as of 2020 and an underproduced binary as of 2022. The correlation between these values is moderate, at 0.62, $p < .0001$, with 3,968 degrees of freedom. This provides evidence in support of the overall validity of our approach and also suggests that underproduction risk as an overall concern can manifest in multiple ways, any of which may be a vector for disruptive failures.

5.3 Underproduction Analysis with an Alternate Indicator of Risk

The Debian community already works actively to understand risks within their ecosystem. We would expect that our method and their methods would identify some of the same packages. Validating our measures with respect to theirs further serves to test whether our measure of underproduction can be used to predict community responses to risk. To do so, we collect data on the count of "non-maintainer uploads" (NMUs) which occur when any individual other than the denoted maintainer updates a version of a package in Debian.[12] Although some package maintainers may welcome NMUs, or even invite them, NMUs are widely understood as an indicator of risk in Debian. One of the criteria for "package salvage," that is, the taking over of maintainership without cooperation from the designated maintainer, is the presence of NMUs.[13] We hypothesize that any valid measure of underproduction in Debian will be positively associated with the number of NMUs that the package receives.

To select a regression model, we compared our data to the parametric assumptions required by applicable models. Our measure of NMUs is a count, however it is overdispersed (i.e. the variance exceeds the mean), meaning a Poisson regression framework is not appropriate. Instead, we conduct our analysis using a negative binomial regression framework, which can accept overdispersion.

Results from our model are reported in Table 2. We find that U_j (underproduction factor) is a statistically significant predictor of NMU count, and that increases in underproduction factor correlate with increased numbers of NMUs. Our model estimates suggest that there is a strong relationship between our measure of underproduction and NMU count ($\beta = 0.47$; CI $= [0.40, 0.53]$). To illustrate this effect, consider two prototypical packages: a fully aligned package ($U_j = 0$) and one of the most underproduced packages in the distribution, as depicted in Fig. 6 ($U_j = 5$). Our model predicts that, on average, a prototypical aligned package will be NMUed extremely rarely ($\widehat{\text{NMU}} = 0.048$; or less than 5% of pack-

[12] https://wiki.debian.org/NonMaintainerUpload (Archived: https://perma.cc/TYN9-WQ79)

[13] https://wiki.debian.org/PackageSalvaging (Archived: https://perma.cc/NBF9-6QSK)

ages will have a single NMU). On the other hand, it suggests that our prototypical underproduced package will be NMUed as often as not ($\widehat{\mathrm{NMU}} = 0.504$).

6 Experiment 3: Social and Technical Factors in Underproduction

Where does underproduction come from within GNU/Linux distributions? Answering this question is difficult because we lack knowledge in the literature about the role and success of distributions in the software supply chain. However, considering that distributions' work involves high levels of expertise in writing, patching, installing, and integrating software, we can draw lessons from the software engineering literature about what factors could be associated with success and failure. Therefore, to develop hypotheses about the sources of underproduction, we analyze both what we know about the most widely cited example of underproduction (the Heartbleed vulnerability in OpenSSL) and how underproduction might arise at the software project level, in general.

OpenSSL is open source software and the most widely used implementation of SSL.[14] The importance of OpenSSL is clear: global web traffic relies on SSL to encrypt traffic between web servers and browsers. In 2014, security researchers identified a vulnerability in OpenSSL that they nicknamed Heartbleed. At the time of its discovery, the vulnerability had been present for more than two years and security researchers estimated that 24-55% of the most visited websites were vulnerable (Durumeric et al. 2014). Estimates of the costs to remediate Heartbleed ranged in the hundreds of millions of US dollars (Kerner 2014). The aftermath of Heartbleed revealed that, despite its importance, OpenSSL had a variety of quality challenges (Walden 2020; Eghbal 2016; Perlroth 2014; Pagliery 2014). The Heartbleed security vulnerability is emblematic of underproduction because it involves extremely important software that was, at the time, of low quality in critical respects.

Scholars have identified several potential reasons for Heartbleed that point to reasons for OpenSSL's underproduction. First, OpenSSL had a complex technical structure that was difficult to analyze by both tools and experts, suggesting a need for widespread refactoring and modernization (Walden 2020; Wheeler 2014; Carvalho et al. 2014). Second, OpenSSL is written in C, which, like C++ but unlike Java, lacks built-in detection of buffer overreads—the bug that led to Heartbleed (Wheeler 2014). Third, its architecture was substantially out of step with modern engineering standards (for example, it implemented its own memory management) (Wheeler 2014). Bug reports related to the dangers of its memory architecture apparently were untriaged in the OpenSSL bug tracker for a substantial period prior to the release of the Heartbleed CVE.[15] Finally, there were a relatively small number of people involved with OpenSSL who were available to detect and respond to security vulnerabilities and very little funding devoted to its upkeep (Carvalho et al. 2014; Eghbal 2016; Walden 2020). Although scholars have presented these factors as causes or contributing factors to Heartbleed, the significance of Heartbleed is not simply in the details of how a buffer overread might arise, nor in how such an error might go undetected, but also in the broader problems of neglect and longstanding quality issues in OpenSSL that Heartbleed revealed.

[14] https://heartbleed.com

[15] E.g., https://flak.tedunangst.com/post/analysis-of-openssl-freelist-reuse

Several of these factors suggest that underproduction is ultimately a technical problem to be prevented and solved. We might expect that modern languages, modern libraries, modern architectures, and modern code analytic techniques could have detected the bug sooner or prevented the bug from being introduced in the first place. Although older packages written in older languages and according to older architectural ideas can be modernized, this is a substantial undertaking (Wheeler 2014). Examining the problem of underproduction from a technical perspective suggests that some codebases are simply harder to maintain adequately and more failure prone. Technical causes of underproduction might include the language in which a piece of software is written, the age of the code, or some combination of these. Although languages and code can be re-factored and modernized, the original language and architecture form at least an upper bound on how old the package can be. All things being equal, we might expect that the newer versions will be better as we take the opportunity to learn from the past. To test these ideas, we propose three hypotheses: **(H1) underproduction is associated with older software**, and **(H2) underproduction is associated with the use of older programming languages**. Furthermore, given improvements in engineering standards over time and our observation that OpenSSL was both an older package and written in an older language (C), we suggest **(H3) Underproduction associated with increased package age will be even stronger when the language is also old.**

Although age may be part of the explanation, some software is better maintained than other software of a similar age. There are numerous cases where old software continues to function as well as or better than newer software. Focusing only on the passage of time would not account for the important role of maintainer resources. The lack of developers who work on OpenSSL is often cited as part of the story of how Heartbleed happened (e.g., Walden 2020; Eghbal 2016; Paul 2014; Pagliery 2014). Proponents of open forms of collaboration point to Linus's law—the notion that "given enough eyeballs, all bugs are shallow" (Raymond 2001). Hence, presumably, the lack of eyeballs leads to undiscovered vulnerabilities. In their study of the PyPi ecosystem, Valiev et al. found that a higher number of contributors was associated with both short-term and long-term project survival (Valiev et al. 2018). A larger OpenSSL maintainer team might have prevented Heartbleed using defect detection tools or seeing the problem sooner. However, as Brooks famously argued, adding additional developer resources can be detrimental to progress (Brooks 1995). In their study of GitHub, Joblin and Apel suggest that small clusters of collaborators with low turnover may be associated with more successful projects compared to projects where larger groups of people are committing to the same functions (Joblin and Apel 2022). Although we acknowledge the presence of competing explanations, our fourth hypothesis draws from the former perspective emphasizing the value of a larger number of contributors and suggests that **(H4) underproduced software has fewer contributors.**

To the extent that open source projects behave in ways that are similar to other approaches to organizing labor, we might look to the management literature, which has observed that turnover in firms can be both positive and negative: high turnover is associated with lower relationship satisfaction, lower social cohesion, and may lead to an overall smaller team, but can have some advantages with respect to innovation (Bolt et al. 2022; Price 1989). That said, attributing underproduction to simply the number of developers seems likely to be incomplete, given that FLOSS developers vary widely in such traits as knowledge of the codebase. Turnover in leadership may be harmful, as found in Joblin and Apel (2022). In his analysis of a software development firm, Mockus found that staff departures were associated with lower software quality measured by code analysis and customer defect reports (Mockus 2010). Work by Hall et al. (2008) likewise found via surveys that low turnover was associated with greater success in software projects, although not conclusively. Some open source projects rely heavily on a single individual, and the loss of the maintainer may spell the end of a project. Coelho and Valente surveyed maintainers of GitHub projects that were once popular but have since been deprecated, and found that lack of maintainer interest and lack of maintainer time were some of the main reasons for project failure (Coelho and Valente 2017). In addition, Coelho and Valente found that although maintainers tried to overcome failure by transitioning ownership to a team, recruiting a new maintainer, or bringing in new contributors, these approaches often did not revive the project.

However, some FLOSS projects are organized such that high levels of maintainer turnover are not detrimental. For example, in their study of five open source projects composed of many modules in an overarching framework (Angular.JS, Ansible, Jenkins, JQuery, and Rails), Foucault et al. found that these successful projects all had relatively high contributor turnover (Foucault et al. 2015). Early work by Michlmayr and Hill observed that Debian tended to rely on single individuals (Michlmayr and Hill 2003). Robles et al. (2005) found that when a maintainer leaves the Debian project, others often adopt their packages. As a result of this process, Robles et al. found that the most important packages tend to be maintained by the most experienced maintainers. However, the adoption of a piece of software by others does not mean that the software will ultimately be maintained at a quality level commensurate with its importance. Nassif and Robillard's 2017 study of eight open source projects (GIMP, Assimp, TrinityCore, Gitlab CE, Linux, Chromium, Kodi, and Apereo CAS) found that the departure of maintainers led to substantial knowledge loss (Nassif and Robillard 2017). Further, Nassif and Robbilard found that these departures led to parts of the project going unmaintained: those who continued to participate did not tend to take on maintenance of the parts written by those who had left.

So, while there are reasons to think otherwise, we hypothesize that loss of a maintainer will make it more likely that a package will be underproduced and that **(H5) underproduction is associated with maintainer turnover.**

FLOSS developers organize their work in multiple ways. They may work alone, form a loose collection of collaborators, or assemble into a team (Christian and Vu 2021; Crowston et al. 2006; Crowston and Howison 2006; Robles et al. 2005). Although the team structure does not need to be large or complex to have an impact, the presence of a team suggests that a more stable collaboration has formed with a distinct identity (Studer 2007). In their study of the PyPi ecosystem, Valiev et al. found that having a project hosted in an "organization" account, rather than an individual one, was associated with the project being 22% less likely to become dormant (Valiev et al. 2018). Transitioning away from depending upon a single maintainer was also one of the techniques Coelho and Valente found that maintainers of failing projects attempted, suggesting that maintainers themselves feel that teamwork may be helpful in preventing failure (Coelho and Valente 2017). Team approaches offer the potential for the transition of leadership, technical help, and social rewards such as encouragement and recognition. This perspective suggests that **(H6) packages maintained by a team are less likely to be underproduced than those maintained by individuals.**

Ways of organizing work go beyond the team that forms around a single package. Developers work together across multiple packages and form broad collaborative networks. Two software packages inside these networks can be thought of as connected if the same person has engaged with both. These types of connections may be valuable because someone who works on several packages may be well placed to see how a bug in one package has implications for another. In fact, Joblin and Apel constructed collaboration networks from developers who commit to the same function and found that degree centrality strongly predicted success for the open source projects they examined on GitHub (Joblin and Apel 2022). We might expect that collaboration allows for greater resilience and productivity within the team supporting individual pieces of software because an individual who is highly connected to other collaborators may be able to access additional social and technical support (Bird et al. 2008). This availability of social connection may be highly valuable in driving project success. In their study of GitHub projects, Qiu et al. found that participants were more likely to persist in software projects with high potential to build social capital (Qiu et al. 2019). A struggling package that is more central within a collaborative network may be more likely to be noticed, and the resources to solve its problems are more likely to be available. These perspectives suggest that **(H7) underproduced packages are associated with collaborators who are farther from the central influential core of collaborators.**

Finally, it may be the case that underproduced packages lack collaborators who are aware of what is happening across Debian as a whole—those people who might notice cross-package trends or have a more accurate sense of the baseline rates of problems and resolution such that they can recognize that a package is struggling. This suggests that **(H8) underproduction is associated with collaborators who lack visibility into what is going on elsewhere in the project—that is, an absence of brokers.**

6.1 Methods

For our hypothesis tests, we use the results of Experiment 1. To identify appropriate sources of predictors, we examine the technical and social conditions of the Debian community. An operating system distribution serves a vital integration role in the world of software development by bringing together, configuring, and testing thousands of packages under a common framework. One of the key ways Debian members contribute to Debian is by serving as a package's *maintainer*. Although the term is used in several ways within Debian, we use the term to describe the person listed in a package's "Maintainer" field. This person (or group) will receive notifications of bugs and is responsible for the package.[16] In Debian, maintainers are not the only individuals who can update packages. Trusted community members (those granted "Debian Developer" status) can upload a Debian package in what is called a non-maintainer upload (NMU). Because they are a sign of problems, in Experiment 2 we used NMUs to validate our measure of underproduction. Many packages list a range of other people in package metadata who, while not the maintainer, can upload packages in what are not considered NMUs. We describe any person who uploads a version of a package outside of an NMU as an "uploader."

A Debian package maintainer is not necessarily the person who develops and maintains the software itself. Debian typically describes this latter person as the "upstream" maintainer. In this sense, Debian maintainers are the downstream recipients of problems that are located upstream in the software supply chain. Regardless of how hard a Debian maintainer works, they may have little control over the challenging behavior of the software they are seeking to integrate with other packages. Of course, the responsibilities of the Debian main-

[16] https://wiki.debian.org/Maintainers

tainer may be made substantially more difficult if the software they are packaging is complex, has fragile dependencies, or is not being maintained well (or at all) upstream. Although our study is of Debian, we examine only one supply chain segment. Some of the data that we collect directly reflects upstream conditions, which vary widely. Figure 10 illustrates a piece of this supply chain; upstream software developers produce software, which is packaged by operating system distribution communities such as Debian.

6.1.1 Data

Debian has a history of making high-quality data available to the public in ways that have been useful for software engineering research (e.g., Caneill et al. 2016; Nussbaum and Zacchiroli 2010). To test our hypotheses, we operationalize the concepts in each hypothesis using measures from this data (corresponding hypotheses are indicated in parentheses). We use the history of package uploads as recorded in the Ultimate Debian Database (Nussbaum and Zacchiroli 2010) maintained by Debian, as well as package release notes for package age **(H1)**, the number of people contributing **(H4)**, the turnover in maintainership **(H5)**, and the presence of a maintenance team **(H6)**. We use the Experiment 1 dataset, which contains estimates for 6,551 packages. We merge these data with measures that we build by creating a series of other datasets. To identify the upstream programming language of each package (**H2**), and we use a set of tags in the UDD of the format "implemented-in::<language>". These tags can be added when a new version of a package is uploaded; we parse approximately 1.7 million upload records and find these tags have been assigned at least once for 2,383 of the 6,551 packages in our study and refer to 24 different languages. Some packages are recorded as having been written in multiple languages; in this case, we take the mean language age. We also use the history of bug resolutions from the Debian Bug Tracking System (BTS) to build a collaboration network (**H7**, **H8**).

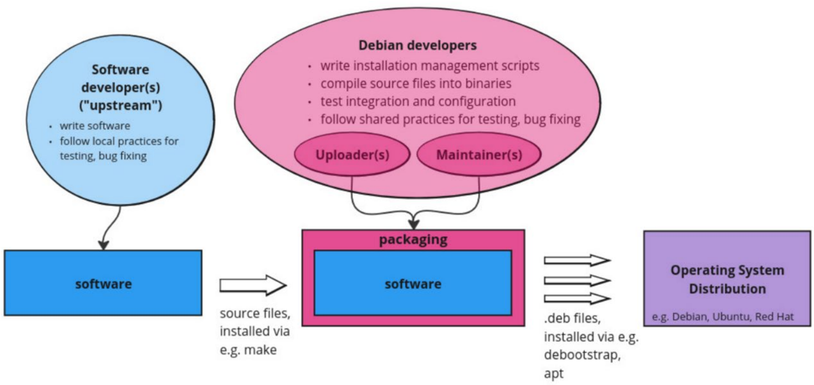

Fig. 10 A piece of the free/libre open source software supply chain. Software is typically developed "upstream", and then numerous software programs are packaged and integrated by Debian developers before being distributed as part of an operating system or using package management tools. Users may also directly install software from source files or precompiled binaries without the benefit of a package manager (not shown)

6.1.2 Measures

Our key outcome variable, *underproduction*, is drawn from Experiment 1. Positive values indicate underproduction, with a higher underproduction factor associated with higher levels of underproduction; the measure is a ratio between quality rank and importance rank. We treat *underproduction* as a Boolean value where true indicates that the underproduction factor is positive.

In **H1**, we proposed that underproduction is associated with older software. We operationalize *package age* as the length of time a package has been present in Debian. To do so, we face the challenge of incomplete data because the tracking database was introduced several years into Debian's history. To prevent left-censoring, we take the earlier of two sources of information: the date that first appears in the package changelog file associated with each package (reading the last five lines in reverse to identify the latest date in the file and manually extracting the date in the 160 cases where the parser could not identify a valid date) or the oldest entry in the uploads database. We then subtract the introduction date from our data collection year (2020). As a result, older packages will have a higher *package age*.

For **H2** about the language age of packages, identifying the language in which a package is written required additional processing because packages are, in some cases, implemented in multiple programming languages and are tagged accordingly. Additionally, packages can change their programming language over time or be tagged incorrectly in ways that are later corrected. Therefore, we calculated *mean language age* per package. We constructed this measure first by measuring language proportion, taking the total proportion of releases tagged with a given language, omitting instances where a release was uploaded without any language tags. In those cases where a single release was uploaded multiple times with differences in tagging, we took the union of the set of languages listed. For example, if a package was uploaded three times, once tagged only as Perl, once tagged Perl and shell, and once tagged only shell, we treated that release as tagged Perl and shell. For a package *i* and language *j*,

$$\text{Release Language Proportion}_{i,j} = \frac{\text{TaggedReleases}_{i,j}}{\text{TotalTaggedReleases}_i}$$

Hence, our measure of language is per package and per language, and a continuous value from 0 to 1.

To calculate *mean language age*, we take the number of years before 2020 in which the language was introduced (the year of introduction was sourced from the English Wikipedia article about the language; larger numbers describe older programming languages) and calculate language age at the package level using the proportion of releases tagged with the language (packages can be tagged with multiple languages in a given release), multiplied by the age of the language in 2020. This measure of language age is admittedly very coarse: languages change over time, and new versions may have very different features than were available in older versions. However, we argue that language authors desire to maintain some level of continuity between versions and that the design decisions made early in a language tend to set the overall tenor for what comes later. Because there are 24 different languages represented, for a package *i* and language *j*:

$$\text{Mean Lang Age}_i = \frac{\sum_{j=1}^{24} \text{Release Lang Prop}_{i,j} * \text{YearsOld}_j}{\#\text{Total Languages Present}_i}$$

In **H4**, we proposed that underproduced packages will have fewer contributors. We measure the number of people contributing using *uploader count*, the number of people contributing to a package by counting the number of unique individuals who have uploaded new versions of the package.

In **H5**, we proposed that underproduced packages will have higher rates of maintainer turnover. We measure maintainership turnover by looking at the maintainer field of all contributions uploaded for a given package. It is important to recall that the maintainer of a package may or may not be the person doing the upload. We use a Boolean value for turnover so that if a package has had more than one maintainer, *maintainer turnover* is true, otherwise false. As a robustness check on this assumption in operationalizing turnover, we also evaluated whether including the number of maintainers in the analysis provided better model fit.

In **H6**, we hypothesized that packages maintained by teams are less likely to be underproduced. Examples of teams are the Debian Games Team and the Debian Python Team. In Debian at the time of upload, the Debian archive system logs the identity of the uploader as well as the identity of the maintainer in the form of a name and email address pair. To detect whether a given upload occurred while the project was being maintained by a team, we examined the maintainer field of the log. If the maintainer is a mailing list (indicated by the mailing list email domain lists.alioth.debian.org), we record it as a team. We also identified three addresses representing the Debian-wide quality assurance teams. Because these teams are typically used as placeholders or as temporary maintainers, we did not treat the presence of these QA groups in the maintainer field as indicative of being maintained by a team and omitted these uploads from our dataset.

In **H7** and **H8**, we hypothesize that underproduced packages will tend to be worked on by people in less central positions within the broader collaborative network in Debian. To build a network that considers both packages and contributors, we considered that contributors may have some social connection by having worked on bugs within the same package.

Using the package *igraph* (Csardi and Nepusz 2006), we first generated a two-mode network from individuals doing bug work in packages. Two-mode networks are used when two types of nodes exist in a network analysis (Wasserman and Faust 1994). In our case, one set of nodes represents individuals, and the other set represents packages. The network is formed by generating an undirected edge between people and packages when they work on bugs in the package and an undirected edge between people who have worked on bugs in the same package. Our network included every bug in every package in Debian. We excluded from this network the internal-only automatic messages (e.g., “owner@bugs.debian.org”) as these do not indicate actual contributors.

To obtain network measures at the level of the package, we then project this two-mode network to a single-mode network such that packages are connected to other packages by means of the people who contribute to their bugs. Therefore, packages are ultimately connected by means of having contributors in common. This analysis yielded 78,955 contributors across 18,399 packages. Not all of these packages were present in the dataset from Experiment 1 due to insufficient bug resolution data to generate an underproduction estimate. Therefore, although we use all 18,399 packages in generating our network measures, our inference is limited to the 6,551 in Experiment 1. We use eigenvector centrality (which ranges from 0 to 1) to evaluate **H7** (proximity to the core). Eigenvector centrality (Bonacich 1972) is a weighted form of degree centrality: after assigning weights to nodes based on the number of connections to other nodes, the value of each connection is reweighted such that connections from high-degree nodes are worth more than those from low-degree nodes in a way that is similar to the Google PageRank algorithm. We use betweenness centrality to evaluate **H8** (cross-project visibility by means of brokerage). Betweenness centrality is the extent to which a given node lies on the shortest path between other nodes.

6.1.3 Identity Merging

Measuring the number of unique people contributing to a package in the form of package uploaders in **H4**, maintainer turnover in **H5**, as well as building the networks of bug collaborators in **H7** and **H8**, required identifying individuals. Doing this necessitated a process of identity merging. Bugs in Debian are represented as email messages, with addresses presented with a name part and an email part. Contributors sending messages with the system do not always use the same name and email address, and their name and address may change over time. A manual review of bug records revealed numerous examples in which an individual submitted a bug with their work or personal address and then worked on the bug and resolved it using their Debian-specific address. As a result, email address was not sufficient as a primary key. Unfortunately, a person's name alone is also not a reliable primary key since some names and nicknames are relatively common globally. The risk is not only from conflating two Johns Smith, but also from conflating Jack Johnson and Jennifer Juarez who have have both decided to set their name part of their email address to "JJ."

Given these concerns, we applied two heuristics as part of our merging process. First, we reasoned that in all cases, someone using the same email address but a different name was likely to be the same person (e.g., "J Doe <jdoe@example.com>" should be treated as the same person as "Jane Doe <jdoe@example.com>"). Further, within the same bug (but only within the same bug), we treated entries with the same name part but different email address part as also the same, e.g., "Jane Doe <jd@workaddress.com>" was treated as the same person as "Jane Doe <j.doe@homeaddress.com>". We also manually inspected this mapping to remove obvious dummy addresses and resolve circular references. This process found 8,741 instances where the same person used more than one address for their Debian bug work and identified aliases associated with 6,438 unique individuals. We used these aliases to perform identity merging on uploaders, maintainers, and collaborators on bugs prior to analysis.

6.1.4 Analytic Plan

We use logistic regression to test each of our hypotheses, which calculates the log odds of an outcome from a vector of explanatory variables. Because the outcome variable in logistic regression is the log odds of an outcome, our models were all of the form:

$$\log\left(\frac{P_{\text{underprod}}}{1 - P_{\text{underprod}}}\right) = \beta_0 + \beta \mathrm{X}$$

where **P** is the probability that a given package is underproduced, odds is defined as probability divided by 1 - probability, **X** describes a vector of variables from our dataset, and β describes the vector of fitted parameter estimates associated with our hypothesis tests.

One analytic challenge we faced was missing data in terms of package programming language (due to untagged packages) as well as missing network measures (due to isolates in the collaboration network, meaning we have no defined centrality measure for a package). In order to offer inference into the source of underproduction while managing the presence of confounders and missing data, we fit four models. M1 includes those predictors for which we have complete data (offering insight into **H1**, **H4**, **H5**, and **H6**). M2 omits only the language age predictors (**H1**, **H4-H8**). M3 omits only the network predictors (**H1-H6**). M4 is the full model (**H1-H8**) but, due to missing data, is estimated using only $\frac{1}{3}$ the observations used in M1. For each hypothesis, we evaluated the relationship between our predictors **X** and underproduction and measure significance at the $\alpha = .05$ significance level.

6.2 Results

The results of our models are presented in Table 3 and alternately in Fig. 11.

6.2.1 H1, H2, and H3: Age

In **H1**, we proposed that underproduction would be associated with older software. Our results in Table 3 provide support for this hypothesis. This finding is consistent across all four models.

We find a significant relationship between the number of years a package has been in Debian and underproduction—the older a piece of software is, the higher its underproduction risk. A visualization of our data with respect to language age appears in Fig. 12.

The coefficients from a model to test this relationship are presented in log-odds units in Table 3, and the individual coefficients should be interpreted as the log-odds ratio associated with a one-unit change in the predictor, all other factors being equal. Thus our model M4 predicts that a package older by one year has an odds ratio of 1.406, corresponding to odds of being underproduced that are 40.6% higher.[17]

In **H2**, we proposed that packages written in older languages would be more likely to be underproduced. Figure 13 shows the data associated with this measure. Our results in Table 3 provide support for this hypothesis across all models fit with this predictor (M3 and M4). M4 predicts that a package written in a language older by one year, all other factors being equal, has odds of being underproduced that are 17.3% higher.

Our hypothesis in **H3** suggests the presence of an interaction between language age and package age, such that underproduction associated with older packages will be more extreme when the language it is written in is also old. However, our models contradict this hypothesis, and we find that while the main effect of being an older package and being written in an older language both tend to increase the probability that a package is underproduced, the interaction between these two independent variables has a negative coefficient. To interpret these coefficients, we visualize the marginal effect of package age for two language ages (25 years, corresponding to Java, and 48 years, corresponding to C), with results

[17] Because our models are logistic regressions, e^{β} is the odds ratio associated with a one-unit change in β. In this case, $e^{.34} = 1.405$. We perform similar transformations for each of the other parameter estimates below.

Table 3 These logistic regression models assess the extent to which underproduction is a function of a range of social and technical factors

	M1: no lang/network measures	M2: No language measures	M3: No network measures	M4: Full model
(Intercept)	-1.90^*	-1.66^*	-6.57^*	-7.28^*
	$[-2.07; -1.73]$	$[-1.91; -1.41]$	$[-8.24; -4.89]$	$[-9.06; -5.50]$
Package Age (years)	0.14^*	0.08^*	0.32^*	0.34^*
	$[0.13; 0.15]$	$[0.06; 0.09]$	$[0.22; 0.43]$	$[0.23; 0.45]$
Uploader Count	0.21^*	0.13^*	0.26^*	0.18^*
	$[0.17; 0.24]$	$[0.09; 0.17]$	$[0.21; 0.32]$	$[0.12; 0.24]$
Did maintainer change?	0.32^*	0.35^*	0.27^*	0.22
	$[0.19; 0.45]$	$[0.19; 0.51]$	$[0.03; 0.51]$	$[-0.03; 0.47]$
Team proportion	0.17^*	0.03	-0.48^*	-0.20
	$[0.02; 0.33]$	$[-0.17; 0.23]$	$[-0.79; -0.16]$	$[-0.54; 0.13]$
Eigenvector Centrality		16.74^*		18.91^*
		$[13.08; 20.41]$		$[14.18; 23.64]$
Betweenness Centrality		-0.00		-0.00
		$[-0.00; 0.00]$		$[-0.00; 0.00]$
Mean Language Age			0.15^*	0.16^*
			$[0.11; 0.19]$	$[0.12; 0.20]$
Package Age : Mean Language Age			-0.01^*	-0.01^*
			$[-0.01; -0.00]$	$[-0.01; -0.01]$
AIC	6586.11	4373.82	2305.08	2088.71
BIC	6619.97	4418.64	2345.35	2140.37
Log Likelihood	−3288.05	−2179.91	−1145.54	−1035.35
Deviance	6576.11	4359.82	2291.08	2070.71
Num. obs.	6450	4459	2328	2299

Coefficients are untransformed log-odds estimates with a 95% confidence interval indicated in brackets. Note that the number of observations varies per model due to missing data

* Null hypothesis value outside the confidence interval

as seen in Fig. 14. Although the confidence intervals around the predictions are relatively wide, this result suggests that the effect on underproduction of being an older package is weaker when the language is also old.

6.2.2 H4: Contributor Count

In **H4**, we proposed that a higher number of contributors would decrease the probability that a package is underproduced. Our results across all four models in Table 3 contradict this hypothesis. Instead, we find that additional uploaders are associated with an increase in odds that a package is underproduced. Our model M4 predicts that an additional uploader is associated with a 19.7% higher odds of being underproduced. In other words, increased uploaders are associated with an increased risk of underproduction.

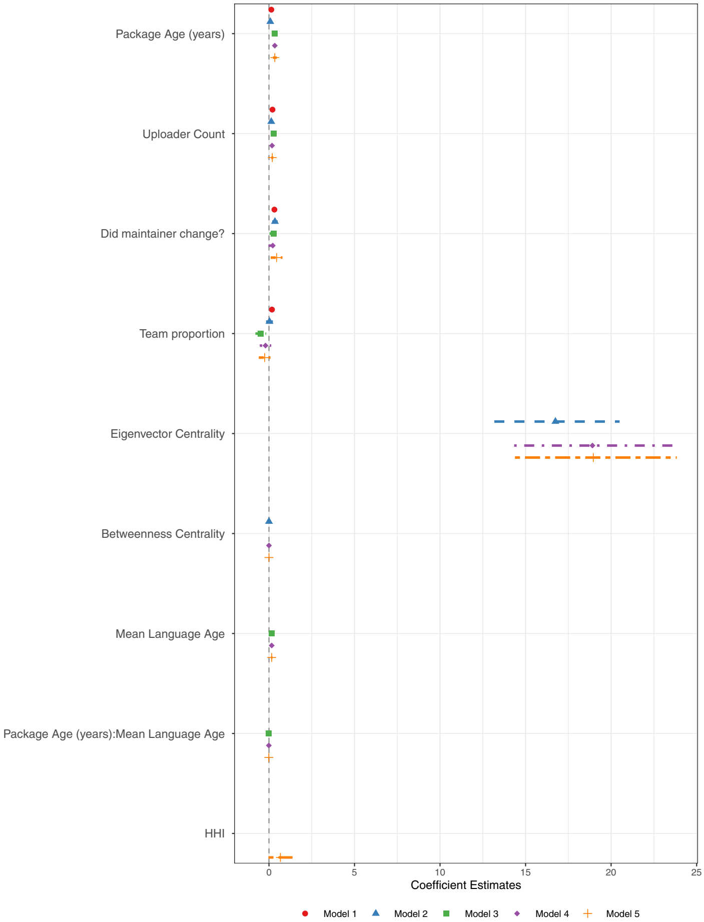

Fig. 11 Coefficients for each model fit as part of Experiment 3; bands represent confidence intervals

6.2.3 H5: Maintainer Turnover

In **H5**, we proposed that maintainer turnover would be associated with a higher probability of underproduction. Although three of the four model results (M1-M3) in Table 3 provide support for this hypothesis, we observe that once we include collaboration network measures, language age, and the interaction of package and language age, this effect is no longer statistically significant. This suggests that having experienced maintainer turnover is not associated with higher odds of underproduction once language age and collaboration network measures are held constant.

We also evaluated the robustness of our operationalization of maintainer turnover as a Boolean as opposed to using total maintainer count. We fit the full model as presented in Table 3 with an additional variable for maintainer count. Maintainer count was not statistically significant in this expanded model. We further evaluated the model with and without maintainer count using the Bayesian Information Criterion (BIC). When comparing nested models using BIC, smaller numbers are better. The BIC of the presented full model is 2140,

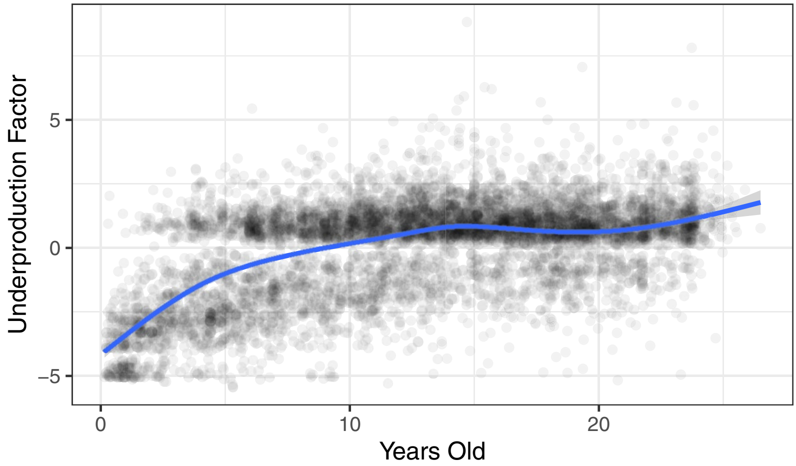

Fig. 12 Visualizing package age based on when the package was added to Debian, with a generalized additive model (GAM) line to indicate a moving average

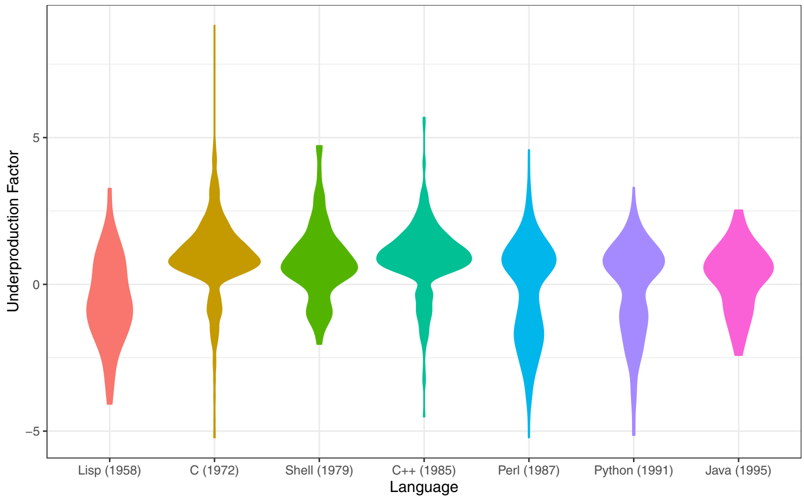

Fig. 13 Violin plot of our data distribution broken down by the most commonly appearing languages. See Table 1 for models which test the relationship between language age and underproduction. This visualization contains data for 2,280 packages. On 135 occasions, the same package appears multiple times because it was consistently tagged as having been implemented in more than one language

and adding maintainer count to the model yields a BIC of 2151. The magnitude of the difference between BIC values lets us characterize the impact of including the additional variable on model fit while balancing the risk of overfitting (Raftery 1995). According to this approach, differences under 2 are considered “weak” evidence, while values in excess of 10 are “very strong” evidence in favor of the simpler model. By this approach, we found very strong evidence in favor of our approach, a model treating maintainer turnover as a Boolean and not considering maintainer count.

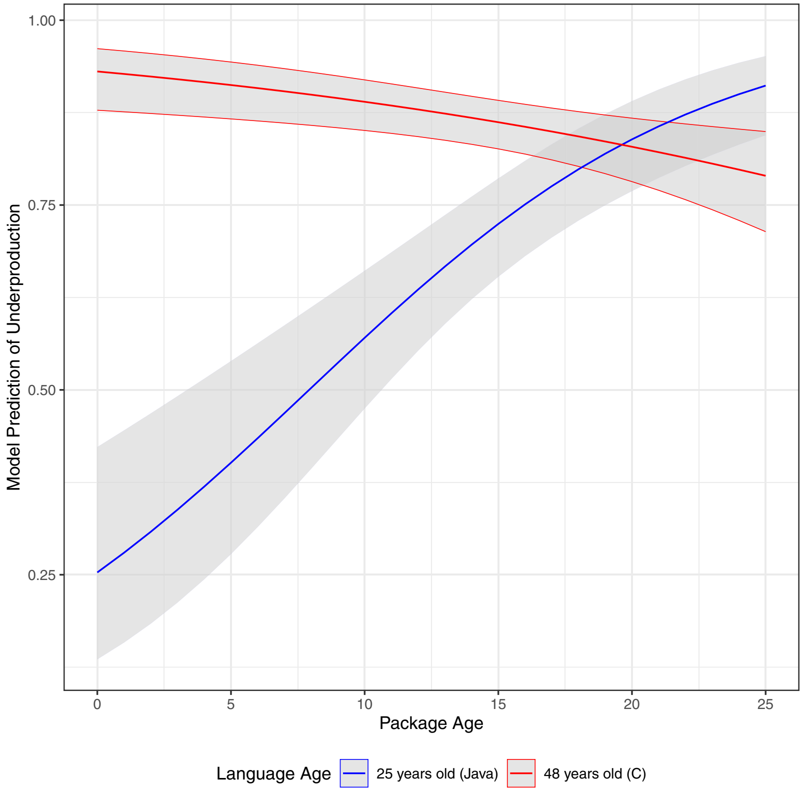

Fig. 14 This visualization shows predicted underproduction probability from model M4 for two prototypical packages of different programming language ages where package age varies as shown along the *x*-axis. The package shown in blue is 25 years old, corresponding to a package written in a language as old as Java, while the package shown in red is 48 years old, corresponding to a package written in a language as old as C. The gray ribbon shows a 95% confidence interval around the prediction

6.2.4 H6: Organizing into Teams

In **H6**, we proposed that being maintained by a team would diminish the likelihood that a given package was underproduced. Model results presented in Table 3 provide little support for this claim. Instead, we find contradictory results in models M1–M3. With all covariates present in the smaller data set in M4, we find that the impact of team proportion is not statistically significant. This suggests that, other factors being equal, the involvement of a maintenance team does not impact the odds of a package becoming underproduced. In fact, our results provide some evidence that the opposite may be true.

6.2.5 H7 and H8: Collaboration Networks

Finally, we examine the role of collaboration networks in underproduction. In **H7**, we test whether underproduced software packages are associated with an absence of influence (eigenvector centrality). Our results for M2 and M4 presented in Table 3 contradicts our expectation in **H7**. Instead, we find that an increase in eigenvector centrality is associated with an increased likelihood of underproduction. In **H8**, we test whether underproduced software packages are associated with an absence of brokerage (betweenness centrality). Our results for M2 and M4 presented in Table 3 contradict our expectation in **H8**. Instead, we find that all other factors being equal, the betweenness centrality of a package is not associated with an increase in the odds that the package is underproduced (Fig. 15).

7 Discussion

Despite the fact that Debian resolves most bugs quickly and that release-critical bugs in Debian are fixed much more quickly than non-release-critical bugs as shown in Fig. 3, our results suggest that underproduction is extremely widespread in Debian.

The presence of substantial underproduction in widely installed components of Debian exposes Debian's users to risk. We also found evidence against most of our hypotheses with respect to social factors. We explore several implications of these findings in the following sections.

7.1 The Long Tail of GUI Underproduction

One striking feature of our results from Experiment 1 is the predominance of visual and desktop-oriented components among the most underproduced packages (see Fig. 6). Of the 30 most underproduced packages in Debian, 12 are directly part of the XWindows, GNOME, or KDE desktop windowing systems. For example, the "worst" ranking package, GNOME Power Manager (*gnome-power-manager*) tracks power usage statistics, allows configuration of power preferences, screenlocking, screensavers, and alerts users to power events such as an unplugged AC adapter. Seven additional packages in Fig. 6 are also geared to desktop uses. For example, the *pm-utils* will suspend or hibernate a computer, *network-manager-openvpn* manages network connectivity through VPNs, Ethernet, and WiFi, *mesa* is a 3D graphics library, *bluez* is part of the Linux Bluetooth stack, *cdrkit* and *dvd+rw-tools* are both tools for creating CDs and DVDs, and *recordmydesktop* is a screen capture pro-

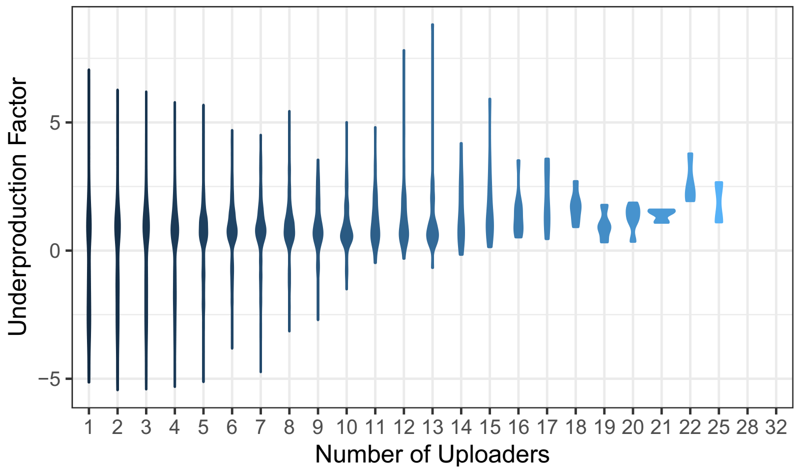

Fig. 15 Relationship between package underproduction and the number of uploaders a package has had

gram. Our finding of relative underproduction in these programs appears to be in line with the history of critiques of GNU/Linux with respect to desktop usability and visual tools (Paul 2009). Although some of the reported issues in these GUI tools may be aesthetic, inspection reveals flaws reported at a range of severities including many very serious bugs.

A critique of the significance of this result might be that visual components are less likely to be used in business-critical circumstances. However, these packages have enormous install bases and are relied upon by many other packages. These results might simply reflect the difficulty of maintaining desktop-related packages. For example, maintaining *gnome-power-manager* includes complex integration work that ranges from a wide range of low-level kernel features to high-level user-facing and usability issues. This pattern of underproduced GUI components suggests that although desktop GNU/Linux has made substantial progress, it remains a source of risk.

7.2 The Role of Technology Choices in Underproduction

In Experiment 3, we tested numerous hypotheses about correlates of underproduction, many of which were not sustained by the evidence. A summary of these hypotheses, as proposed, and an icon illustrating whether they were successful, appear in Fig. 16.

In **H1-H3**, we examined the relationship between older software, older languages, and underproduction. Our results suggest that we should think of software age in a nuanced way. All other things being equal, an older package or one written in an older language is more likely to be underproduced. However, having been written in an older language is associated with a weaker effect of package age. These results suggest that although software faces increased underproduction risk as it (and its language) grows older, thinking of software only in terms of language or age is insufficient. In fact, for older software in older languages to be present in our dataset, it must have survived for decades. Recently written packages in newer languages are also less likely to be underproduced. In addition, we observe that there is substantial variation within languages in Fig. 13 and in the width of the confidence interval in Fig. 14. Committed communities may be able to maintain the health of pieces of software regardless of their age and language, but there are no guarantees.

From the perspective of software users, underproduction risk due to age is also challenging. Older software may have a range of benefits not captured in the direct maintenance of

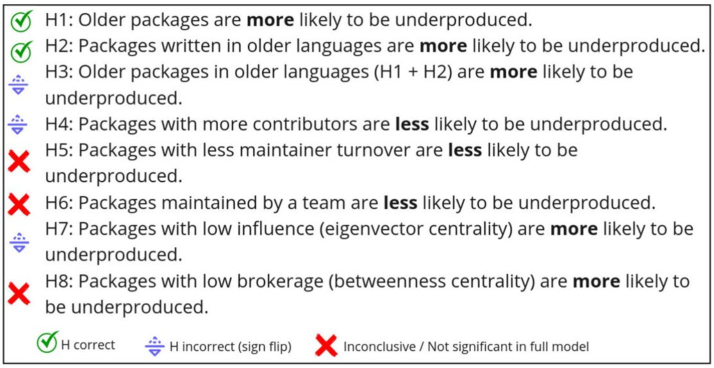

Fig. 16 All hypotheses tested in this experiment as originally phrased, and the outcome of the test

the software's quality: knowledge and trust from a history of use, availability of documentation, integration with other tools, embeddedness in a given process, or the existence of migration paths to an alternative. Change to a new service or paradigm has a cost even when the current solution is performing quite poorly. These factors may continue to elevate the importance of the software even after quality has fallen away substantially.

7.3 Organizing to Address or Prevent Underproduction

Several of the hypotheses we tested in Experiment 3 are less about technology and more about how human resources are organized (see Fig. 16 for a summary). In **H4**, we studied the number of contributors by examining uploader quantity. In **H5**, we examined maintainer turnover. In **H6**, we examined the question of how uploaders and maintainers are organized. Contrary to our expectations, additional uploaders to a package were associated with higher odds of underproduction. Maintainer turnover, in and of itself, is not associated with underproduction. Nor is the presence of a maintenance team in our full model. These are challenging results for communities seeking to organize effort in ways that resist or prevent underproduction. It may be that identifying a level of modularity where a single uploader can sustain effort for the package's life is helpful (hence, the community should orient itself to retaining that uploader's commitment). Awareness of the key role of individual effort and the risk involved in a transition from "one" to "more than one" may allow these projects to think differently about how they approach scale and burnout prevention. Or, it may be the case that detection and remediation are more achievable than prevention.

Our results in **H7** and **H8** suggest that the people working on bugs in underproduced packages are influential in the network formed by bug commentors. This suggests that, rather than being isolated from other software, underproduced software is drawing from a highly central resource pool—one that is perhaps spread too thinly. This suggests that the people involved with fixing bugs in these underproduced packages are in influential positions within Debian. However, our cross-sectional approach does not allow us to distinguish if the engagement with influential contributors predates the emergence of underproduction or followed after it. In sum, these findings suggest that the best-case scenario for a piece of software is to be maintained by a dedicated individual who does not work on many other pieces of software. This result is consistent with a finding from Xiao et al. (2023), who observed that early participants with substantial experience in open source but who are not broadly committed to many other projects are predictors of long-term sustainability for the project.

7.4 Extended Analysis: Productive Inequalities?

Our findings that more uploaders and central network position are associated with underproduction stand in tension with information systems and management literature emphasizing depth of support and the avoidance of single points of failure. We investigated this topic further and performed additional analysis. In developing our hypotheses **H4-H7**, we considered not only the possibility of rare negative events such as the sudden loss of a key contributor but also the potential for burnout from overreliance on single individuals or small teams.

Our model found that the relationship between maintainer turnover and underproduction is such that turnover predicts higher risk in some simpler models but is not statistically significant in the fully controlled model taking network position into account.

Previous work has explored the potential risk generated by inequality of participation. This risk is sometimes called the "truck factor" (Ferreira et al. 2019) or "bus factor" (Jabrayilzade et al. 2022)—the notion that some people are so personally engaged with an activity, and the work is so dependent on them—that if they were hit by a truck or bus on the way to work, the project might fail. In open source, this notion is also called the "boss factor"—volunteer projects may be so dependent on a single individual's efforts that their other life commitments may eventually make working on the open source project untenable (Villa 2019). Previous work has found that contributors report that they have disengaged from FLOSS following substantial transitions in their lives (Miller et al. 2019). Thus, too much reliance on a single individual may lead to neglect of a highly valued package. Such key individuals might also be prone to burnout (Geiger et al. 2021). However, we may also consider that too little reliance may have disadvantages as well–if no one is visibly "in charge," perhaps no one is in charge.

Previous work has found that popular packages are subject to a "bystander effect" in which it is generally assumed that such a widely used package has adequate resources (Eghbal 2020). Inequality of participation and contribution is also a long-standing observed phenomenon in open source and peer production more generally, with a small core group of participants and a substantial periphery of casual contributors (Crowston and Howison 2006; Christian and Vu 2021). However, it may be the case that substantial reliance on a single individual or a small team is, in fact, a highly productive form of inequality—and perhaps remain so right up until some point of failure. To better understand the role of inequality of participation in underproduction, we used an inequality measure developed in economics: the Herfindahl-Hirschman Index (HHI). HHI is widely used to measure concentration in terms of market share and household income (Rhoades 1993). It is also used to measure contribution inequality in software maintenance effort (Narayanan et al. 2009).

HHI is calculated by first determining each entity's market share as measured by proportion, then taking the sum of the squares of all entities' shares, placing relative concentration of a market on a 0 to 1 scale. This approach allows for a comparison between different markets (or in our case, projects) based on degree of consolidation. In our case, we calculate the market share of a given maintainer by determining how long the package has been in Debian, then measuring the duration of time each individual served as a maintainer (their "share" of the overall available "maintainability market.") We calculated an individual's maintainership spells as beginning when the package is first uploaded with their name listed as the maintainer, and ending when a subsequent version names a different maintainer. A given individual may have multiple maintainership spells, and we added these together to the extent that we could determine that two names were referring to the same person, using the same records of package uploads stored in the Ultimate Debian Database (Nussbaum and Zacchiroli 2010) and the identity merging procedure described in § 6.1.

7.4.1 Contextualizing Productive Inequalitives

We observe that HHI for these packages is very unequally distributed, with large numbers of projects with a concentration of 1; that is, they did not experience any maintainer turnover, while the remaining packages are widely distributed, as visualized in Fig. 17.

We consider the surprisingly small amount of maintainer turnover in light of two additional studies examining social and technical measures with respect to desirable outcomes in open source. Gaughan et al. (2024) examined projects in the ‘upstream’ of the current study, that is, they examined communities developing the software distributed by Debian. They found that power sharing was associated with lower underproduction risk—that is, a higher proportion of project committers who have pull request acceptance rights. Gaughan et al. (2024) also examined the association between formal engineering processes and underproduction, with a finding that this relationship was not statistically significant. Alami et al. (2024) also explored the tensions between broadly recommended community practices and technical outcomes, asking whether common community sustainability measures were actually related to code quality in the context of projects participating in the Apache Incubator program. Alami et al. (2024) found that their selected sustainability metrics were not consistently linked to software quality, and that indeed some facets of sustainability may have negative impacts on quality. These results suggest that communities face difficult trade-offs as they seek to maintain both their ongoing health and technical achievements.

In Table 4, we present the results of the model presented in Experiment 3, with the addition of HHI. These results indicate that HHI is not statistically significant in the fully controlled model. Of course, the difference between 1 and >1 maintainer is already tested in our models in H5 using our dichotomous measure of maintainer turnover. In this full model with HHI, the effect of any turnover is positive, larger in magnitude than any of our previous models in Table 3, and statistically significant. The results for HHI in Table 4 shows that

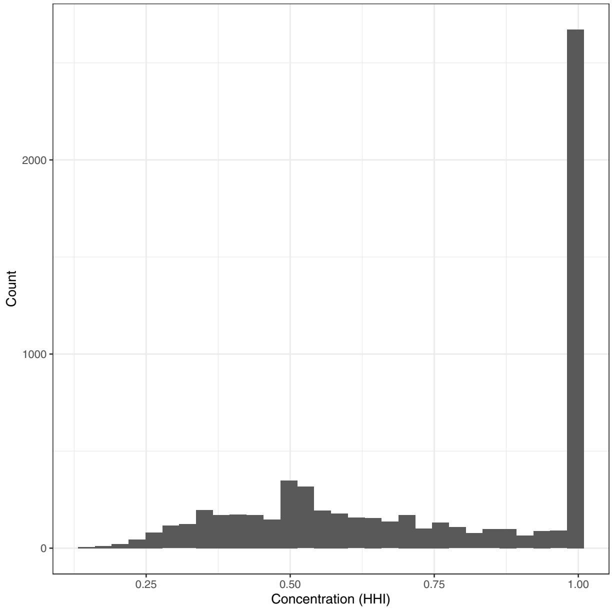

Fig. 17 Histogram of HHI levels observed in our data

Table 4 This is a variation of the full model from Table 2, with the addition of HHI

	Model 1
(Intercept)	−8.00*
	[−9.94; −6.06]
Package Age (years)	0.34*
	[0.23; 0.45]
Mean Language Age	0.16*
	[0.12; 0.20]
Uploader Count	0.19*
	[0.13; 0.26]
Did maintainer change?	0.44*
	[0.10; 0.79]
Team proportion	−0.25
	[−0.60; 0.09]
Eigenvector Centrality	18.96*
	[14.23; 23.69]
Betweenness Centrality	−0.00
	[−0.00; 0.00]
HHI	0.67
	[−0.03; 1.37]
Package Age : Mean Language Age	−0.01*
	[−0.01; −0.01]
AIC	2087.20
BIC	2144.61
Log Likelihood	−1033.60
Deviance	2067.20
Num. obs.	2299

Coefficients are untransformed log-odds estimates with a 95% confidence interval indicated in brackets

* 0 outside the confidence interval

we cannot reject the null hypothesis that there is no marginal effect associated with further decreases in the concentration of maintainer duties. We also observe that the addition of the HHI variable results in a worse model fit, as evaluated by the Bayesian Information Criterion (BIC), in which smaller values are better and values on the order of 4 are associated with what Raftery (1995) characterizes as "positive" evidence in favor of the model in Table 3 rather than the model we present here in Table 4.

So although we do not find evidence that concentration is associated with underproduction either positively or negatively, the rest of our results are largely unchanged. Inequality—such as having relatively few contributors and low turnover of maintainers—may indeed be productive. Although we should be concerned about the human cost of such inequality, we might also consider how to intervene in ways that diminish the toll without sacrificing potential advantages. Deep absorption with one's life work deserves appreciation, celebration, and support. Part of countering underproduction and improving the sustainability of digital infrastructure may lie not in removing these inequalities but rather in the building of institutions that can support maintainers in a more comprehensive way.

7.5 Implications for Practice

FLOSS communities may find it useful to employ our technique to identify underproduced software in their repositories and to allocate resources and developer attention accordingly.

However, any such intervention should consider that part of the success of open source has been its flexibility and the ability of volunteers to select their own tasks. Any intervention should take the value of that self-motivation into account, and be driven by community norms and practices. For example, the Debian project has a volunteer QA team who might benefit from using our analysis to allocate its effort. These individuals are already motivated to perform broad QA tasks and can do so within community guidelines. Other potential audiences for underproduction information are maintainers and projects seeking to work with funding agencies who support maintainers of key infrastructure, such as the Sovereign Technology Agency (Germany), Alpha-Omega (Linux Foundation), and the Safe Open Source Ecosystems program (United States). Although underproduction is likely to be a particularly acute problem in FLOSS projects due to developer self-selection into tasks, it may also exist in non-FLOSS contexts. We look forward to working with managers of software repositories in both FLOSS and non-FLOSS contexts to help them implement and take action based on measures of underproduction.

Although FLOSS practitioners may be particularly concerned about underproduction in their projects, the software infrastructure risk that results from underproduction in FLOSS is of broader concern. FLOSS acts as global digital infrastructure. Failures in that infrastructure ripple through supply chains and across sectors. Technology leaders may see opportunities to improve their own risk profiles by offering support to FLOSS components identified as relatively underproduced through the type of analysis we have described.

Many studies have examined the effect of funding when firms participate in FLOSS. Mixed and often ineffective results might be improved if underproduction is taken into account when directing resources to FLOSS projects. Trade-offs associated with the potentially demotivating effect of money can be mitigated if funds are targeted at high-impact areas with little existing volunteer labor. The influx of successful investment that followed the Heartbleed vulnerability (Walden 2020) may offer concerned organizations some hope that intervention in response to underproduction is possible and can be effective. Our work seeks to help guide these types of interventions before events like Heartbleed.

Our study draws both from the distribution level and from upstream development communities, asking whether underproduction at the distribution level is attributable to technical factors such as the age of the package and the language in which the package is written or to how the distribution organizes effort. Although software developers and maintainers of distributions such as Debian take on different roles in the supply chain, they have an important relationship. Software developers benefit from the additional testing, dependency management, visibility, and support that distributions provide. To the extent that their goal is to serve end users, making their package easy for distributions to maintain is in their best interest. For their part, distribution maintainers depend on upstream developers to make good quality software that can be easily installed and readily integrated with the other packages. Both groups have a role to play in preventing and addressing underproduction.

For developer communities, our finding that underproduction seems to be an inevitable consequence of age and language age suggests that all projects need to confront the march of time. New projects should be careful about using older languages. However, the negative interaction term brings a sign of hope: long-standing projects may continue to pass the test of time.

For distributions like Debian, our findings with respect to organizational structure should be particularly helpful. As described, the best-case scenario may be to support dedicated and focused individuals rather than pushing for simply “more eyeballs” or large volumes of new contributors pitching in casually. Although much of the literature on peer production communities emphasizes the power of these casual contributors as part of the long tail, distributions like Debian are an important counterexample.

Benkler (2006) identified modularity of contribution as a key factor in commons-based peer production. It may be the case that identifying the right level of granularity and coordination of that modularity is key to sustaining open source more broadly. The work of sustaining FLOSS is both an opportunity for individuals to make important personal contributions and for them to band together to build teams within larger communities. Although communities producing FLOSS distributions have little control over the age or language of the software they package, they have control over other elements, including whether they package the software at all. Our work points to ways that FLOSS organizations can use information on relative underproduction and its correlates to allocate resources and make difficult choices about when to retire or omit software when further intervention may not address the likely underlying causes of problems. However, distributions should consider omitting software that falls below their standards, including not only consideration of whether quality is sufficient relative to importance but also the impact of relying on core contributors to maintain it.

7.6 Implications for Software Engineering Research

Although our conceptual model and experiments demonstrate that underproduction can be measured, the detection and measurement of underproduction and the modeling of its predictors, causes, and remedies reflect a series of open challenges for the software engineering research community. More work is needed to validate our conceptual framework and our statistical approach. We also hope that future work will extend our approach and implement our framework in other FLOSS repositories.

Our findings on the tensions among different desirable outcomes—e.g., reliable software and depth of support—have important implications for research seeking to understand how to build and manage successful peer production projects. Attention to the quantity of participation and shared responsibility at the expense of personal investment may serve to ultimately undermine peer production.

While our evidence suggests that packaging software is well served by single individuals, some peer production activities are likely to be impossible without a team effort. Understanding the relative modularity of production tasks and the conditions that make it necessary to set aside the advantages of unitary leadership in favor of collaborative effort is a key area for future work. Future work should seek to further unpack the factors that affect quality or importance to understand how these factors ultimately affect underproduction. In addition, we should explore what social and structural factors might affect communities’ underlying ability to ensure more or less alignment between quality and importance. Additional methods need to be developed to understand underproduction not only in a cross-sectional and cumulative manner but also longitudinally, to support prioritization and intervention.

8 Limitations

Although Debian is widely used in software engineering research and the study of distributions has been identified as preferable to code repository hosting platforms (Spaeth et al. 2007), it is only a single empirical setting. Distributions are a popular way of obtaining packages and managing updates, as well as serving as an important signal of which packages to install, however they are not the only way to obtain software. Users can also make use of 3rd party distributors, language-specific distributors, or obtain software directly from the communities producing it (Legay et al. 2020). Scoping the ecosystem under study (Step 1 of Underproduction Analysis) also sets a scope on how the results can be interpreted in terms of risk to an individual system. Additionally, although our sample in Experiment 1 captures a broad time span of bug reports in Debian, it only reflects one snapshot in time for package installation. Our validation work in Experiment 2 seeks to mitigate these limitations.

Although prior research has implicated lack of maintenance as increasing the likelihood of failure, (e.g.,Walden 2020; Coelho et al. 2018; Eghbal 2016), we do not develop evidence to support a causal claim. Our approach is quantitative and examines a broad range of packages; although we can therefore assess the ecosystem in general terms, qualitative approaches such as interviews, or case studies of individual packages remains for future work. We hope that further work will demonstrate whether underproduction is predictive of significant failure. The validity of our approach is also subject to threats related to construct validity. Underproduction is a concept borrowed from economics and involves a relationship between supply and demand. Although we have leveraged existing studies of underproduction in Wikipedia as part of our process of conceptualization, there remains space for further discussion about what should constitute ‘supply’ and ‘demand’ in FLOSS. For example, we treat supply as quality but it might also be conceptualized as code quantity or developer effort. Should demand be a raw measure of consumption, as we have conceptualized it, or should we explore alternate approaches like central positions in software dependency networks? Our validation work in Experiment 2 seeks to mitigate this limitation.

Our measures of quality are limited to the performance of the Debian community in resolving bugs that they accept and triage, and does not directly address the quality of the code itself, nor the internal history of the bug as the community works to address it. The wide confidence intervals around many of our package-level estimates argue for caution in using them; our validation work in Experiment 2 seeks to mitigate this limitation.

Our measure of importance is constrained to the systems whose administrators have volunteered data using Popcon. As with all opt-in surveys, this results in a nonrandom sample. Bias in this sample is possible because the installation of certain packages is possibly correlated with participation in the survey. Further, our measure of importance is limited to the most-installed binary of a given source package at a single point in time as measured via an opt-in survey tool running automatically; dependencies are not taken into account. The impact of this conservative approach is that we may underestimate the importance of a given package; in this case, a package may appear to be lower risk than it actually is. It is also the case that importance and install base may be measured in different ways and our choice of metric may influence our results (Zerouali et al. 2019a). Although we consulted with Debian community members in designing and interpreting our experiments, there is no ground truth that we can use to ensure that we got it right. This limitation is common across open source research. With open licensing, open source code, and no point of sale, it is quite difficult to determine where and how most FLOSS packages are being used. A single download might represent one or thousands of uses; that download might be destined for a hobbyist laptop or a life safety server. Research into this topic often relies on proxy or indirect measures of importance, including download counts, indicators of social attention such as repository stars, system probes, and surveys.

Finally, we conducted a validation of our approach using non-maintainer upload (NMU) count in Experiment 2: the number of times a package is updated by someone other than its maintainer. However, this measure is not an entirely independent measure: packages with long resolution times and high user bases may also be more likely to draw outside contributions in the form of NMUs. Furthermore, our results should be characterized within the context of how underproduction manifests itself at the distribution level. Although Debian is an important part of the software supply chain, it is only one link in the chain. Different links may be characterized by different concerns.

Experiment 3 also has limitations in its operationalization. Our measure of package age does not consider how long a package existed before it was added to Debian and is thus only a lower bound on the age of the software. Our assessment of language age only considers the year in which a given language emerged. Although this is an important part of the context of a given language and the paradigms under which it was designed, languages evolve, and code is rewritten and refreshed. Our use of language tagging in Debian omits variation in how important a given language may be to a package, and these tags are unlikely to be missing at random.

We have taken up four different perspectives on ways that people collaborate—maintainership, uploading, declaring a team, and working on bugs. However, each of these measures is relatively coarse and does not take into account the history of the package. This omits multiple forms of variation, which may be important, such as how the team functions. Further, our analysis's cross-sectional nature omits the maintenance structure's history. Teams and overworked individuals may adopt packages because they are underproduced. Or, what begins as a team effort may fall into disarray with an individual left picking up the pieces. We have sought to limit the impact of confounders like these by including a full model with all predictors. However, because our collection of predictors is necessarily incomplete, our ability to infer the causes of underproduction is limited.

9 Conclusion

Our work makes three important contributions to software engineering research: 1) we present a broad conceptual framework for identifying relative underproduction and illustrate our approach using data from Debian; 2) we validate our method from multiple perspectives; and 3) we conduct a series of hypothesis tests to consider potential social and technical correlates of underproduction. The results of our experiments revealed significant underproduction in the widely used Debian distribution and that although some common explanations are in part correct (the passage of time is associated with increased risk), others are not sustained (more resources are not associated with less risk).

Given that older software, or software written in older programming languages, is at greater risk, confronting underproduction risk is an inevitable task for software in the longer term. One strategy to confront underproduction risk is to consider how best to organize maintenance effort. Although individual contributors and teams may be viable, our results suggest that underproduction risk is associated with projects with higher resources. We find no evidence that maintainer turnover is associated with higher risk or that teams are associated with lower risk in our full model.

Productive inequality can be powerful, and we see this model echoed in many levels of society—consider the driven founder, the heroic civic volunteer, the department administrator who seems to know how everything works, the everywhere-at-once activist, or the family member who serves to bind everyone together. Caring for these maintaining leaders among us, working to scope their burden to a level they are inspired to sustain, and building a pathway for them to transition their work to a new committed leader, is a key challenge for peer production communities.

Acknowledgements This paper extends work previously published as part of the first author's dissertation. Debian Developers Paul Wise and Don Armstrong provided valuable assistance. The Ford/Sloan Digital Infrastructure Initiative, (Sloan Award 2018-11356) and the US NSF (IIS-2045055) provided financial support.

Author Contributions Kaylea Champion led the conduct of this work, including design, data collection, analysis, and writing. Benjamin Mako Hill supervised, advised, and assisted with all parts of this work.

Funding This work was supported by the Sloan Foundation through the Ford/Sloan Digital Infrastructure Initiative, Sloan Award 2018-11356 as well as the National Science Foundation (Grant IIS-2045055). We conducted the analysis using the Hyak supercomputer at the University of Washington as well as research computing resources at Northwestern University.

Data Availability Full code and data for replicating our results are available via the Harvard Dataverse at https://doi.org/10.7910/DVN/OGLTJ9.

Declarations

Ethical Approval This type of research using these data has been reviewed by the IRB at the authors' institution and has been determined not to be human subjects research.

Informed Consent These experiments were conducted with data made public by the Debian community and do not involve any interaction with human subjects.

Conflict of interest The authors declare that they have no conflict of interest.

Clinical Trial Number Clinical trial number: not applicable.

References

Abou Khalil Z, Constantinou E, Mens T, Duchien L, Quinton C (2019) A Longitudinal Analysis of Bug Handling Across Eclipse Releases. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME). Cleveland, OH, USA, IEEE, pp 1–12. https://doi.org/10.1109/ICSME.2019.00010

Adewumi A, Misra S, Omoregbe N, Crawford B, Soto R (2016) A systematic literature review of open source software quality assessment models. SpringerPlus 5(1):1–13. https://doi.org/10.1186/s40064-016-3612-4

Aksulu A, Wade M (2010) A Comprehensive Review and Synthesis of Open Source Research. J Assoc Inf Syst 11(11/12): 576–656. https://doi.org/10.17705/1jais.00245

Alami A, Pardo R, Linåker J (2024) Free open source communities sustainability: Does it make a difference in software quality? Empir Softw Eng 29(5):114. https://doi.org/10.1007/s10664-024-10529-6
Al Dallal J, Abdin A (2018) Empirical Evaluation of the Impact of Object-Oriented Code Refactoring on Quality Attributes: A Systematic Literature Review. IEEE Trans Softw Eng 44(1):44–69. https://doi.org/10.1109/TSE.2017.2658573
Asay M (2019) The real number of open source developers. https://www.infoworld.com/article/3452881/the-real-number-of-open-source-developers.html
Becker B, Jeannerod N, Marché C, Régis-Gianas Y, Sighireanu M, Treinen R (2022) The CoLiS platform for the analysis of maintainer scripts in Debian software packages. Intl J Softw Tools Technol Trans 24(5):717–733. https://doi.org/10.1007/s10009-022-00671-1
Benkler Y (2002) Coase's penguin, or, Linux and the nature of the firm. Yale Law Journal 112(3):369–446. https://doi.org/10.2307/1562247
Benkler Y (2006) The wealth of networks: How social production transforms markets and freedom. Yale University Press, New Haven London
Bennett JC, Bohoris GA, Aspinwall EM, Hall RC (1996) Risk analysis techniques and their application to software development. Eur J Oper Res 95(3):467–475. https://doi.org/10.1016/S0377-2217(96)00171-3
Bird C, Pattison D, D'Souza R, Filkov V, Devanbu P (2008) Latent Social Structure in Open Source Projects. Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (pp. 24–35). New York, NY, USA: ACM
Bolt EET, Winterton J, Cafferkey K (2022) A century of labour turnover research: A systematic literature review. Intl J Manag Rev 24(4):555–576. https://doi.org/10.1111/ijmr.12294
Bonacich P (1972) Technique for Analyzing Overlapping Memberships. Sociolog Methodol 4:176–185. https://doi.org/10.2307/270732
Brooks FP (1995) The Mythical Man-month: Essays on Software Engineering. Addison-Wesley Pub. Co, Reading, Mass
Caneill M, German DM, Zacchiroli S (2016) The Debsources Dataset: Two Decades Of Free And Open Source Software. Zenodo. Retrieved: 2018-11-07. https://zenodo.org/record/61089
Carvalho M, DeMott J, Ford R, Wheeler DA (2014) Heartbleed 101. IEEE Security & Privacy 12(4):63–67. https://doi.org/10.1109/MSP.2014.66
Champion K, Hill BM (2021) Underproduction: An approach for measuring risk in open source software. In: 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). Honolulu, HI, USA, IEEE, pp 388–399. https://doi.org/10.1109/SANER50967.2021.00043
Champion K, Hill BM (2024) Sources of Underproduction in Open Source Software. The IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 740–751). Rovaniemi, Finland. https://doi.org/10.1109/SANER60148.2024.00081
Chen K, Wagner D (2007) Large-scale analysis of format string vulnerabilities in Debian Linux. In: Proceedings of the 2007 workshop on Programming languages and analysis for security - PLAS '07 (p 75). https://doi.org/10.1145/1255329.1255344
Choudhary S, Bogart C, Rose C, Herbsleb J (2020) Using Productive Collaboration Bursts to Analyze Open Source Collaboration Effectiveness. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 400–410). London, ON, Canada: IEEE. https://doi.org/10.1109/SANER48275.2020.9054852
Christian J, Vu AN (2021) Task-based structures in open source software: revisiting the onion model. R &D Management 51(1):87–100. https://doi.org/10.1111/radm.12428
Claes M, Mens T, Di Cosmo R, Vouillon J (2015) A Historical Analysis of Debian Package Incompatibilities. 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories (pp 212–223). Florence, Italy: IEEE. https://doi.org/10.1109/MSR.2015.27
Coelho J, Valente MT (2017) Why modern open source projects fail. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017 (pp 186–196). Paderborn, Germany: ACM Press. https://doi.org/10.1145/3106237.3106246
Coelho J, Valente MT, Silva LL, Shihab E (2018) Identifying unmaintained projects in github. In: Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (pp 1–10). Oulu Finland: ACM. https://doi.org/10.1145/3239235.3240501
Cordy JR, Roy CK (2011) DebCheck: Efficient Checking for Open Source Code Clones in Software Systems. In: 2011 IEEE 19th International Conference on Program Comprehension (pp 217–218). Kingston, ON, Canada: IEEE. https://doi.org/10.1109/ICPC.2011.27
Crowston K, Howison J (2006) Hierarchy and centralization in free and open source software team communications. Knowl Technol Policy 18(4):65–85. https://doi.org/10.1007/s12130-006-1004-8
Crowston K, Wei K, Howison J, Wiggins A (2012) Free/Libre open-source software development: What we know and what we do not know. ACM Computing Surveys 44(2):1–35. https://doi.org/10.1145/2089125.2089127

Crowston K, Wei K, Li Q, Howison J (2006) Core and periphery in free/libre and open source software team communications. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences, 2006. HICSS '06 (Vol 6, p118a). Kauai, Hawaii: IEEE Computer Society. https://doi.org/10.1109/HICSS.2006.101

Csardi G, Nepusz T (2006) The igraph software package for complex network research. InterJournal, Complex Systems, 1695. Retrieved from https://igraph.org/

Dabbish L, Stuart C, Tsay J, Herbsleb J (2012) Social coding in GitHub: transparency and collaboration in an open software repository. Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work - CSCW '12 (p 1277). Seattle, Washington, USA: ACM Press. https://doi.org/10.1145/2145204.2145396

Davies J, Zhang H, Nussbaum L, German DM (2010) Perspectives on bugs in the Debian bug tracking system. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (pp 86–89). Cape Town, South Africa: IEEE. https://doi.org/10.1109/MSR.2010.5463288

Durumeric Z, Li F, Kasten J, Amann J, Beekman J, Payer M, Halderman JA (2014) The Matter of Heartbleed. In: Proceedings of the 2014 Conference on Internet Measurement Conference (pp 475–488). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2663716.2663755

Eghbal N (2016) Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure. Ford Foundation

Eghbal N (2020) Working in Public: The Making and Maintenance of Open Source Software. Stripe Press, San Francisco

Ernst NA (2018) Bayesian hierarchical modelling for tailoring metric thresholds. In: Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18 (pp 587–591). Gothenburg, Sweden: ACM Press. https://doi.org/10.1145/3196398.3196443

Falcao F, Barbosa C, Fonseca B, Garcia A, Ribeiro M, Gheyi R (2020) On Relating Technical, Social Factors, and the Introduction of Bugs. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 378–388). London, ON, Canada: IEEE. https://doi.org/10.1109/SANER48275.2020.9054824

Ferreira M, Mombach T, Valente MT, Ferreira K (2019) Algorithms for estimating truck factors: a comparative study. Softw Quality J 27(4):1583–1617. https://doi.org/10.1007/s11219-019-09457-2

Foucault M, Palyart M, Blanc X, Murphy GC, Falleri J-R (2015). Impact of developer turnover on quality in open-source software. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (pp 829–841). Bergamo Italy: ACM. https://doi.org/10.1145/2786805.2786870

Galindo Duarte JA, Benavides Cuevas DF, Segura Rueda S (2010) Debian Packages Repositories as Software Product Line Models. Towards Automated Analysis, First International Workshop on Automated Configuration and Tailoring of Applications

Gaughan M, Champion K, Hwang S (2024) Engineering Formality and Software Risk in Debian Python Packages. In: 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 1005–1010). (ISSN: 2640-7574).https://doi.org/10.1109/SANER60148.2024.00108

Geiger RS, Howard D, Irani L (2021) The labor of maintaining and scaling free and open-source software projects. In: Proceedings of the ACM on Human-Computer Interaction 5(CSCW1), 175:1–175:28. https://doi.org/10.1145/3449249

Gonzalez-Barahona JM, Robles G, Herraiz I, Ortega F (2014) Studying the laws of software evolution in a long-lived FLOSS project: Studying the Laws of Software Evolution. J Soft Evol Process 26(7):589–612. https://doi.org/10.1002/smr.1615

Gorbatai AD (2011) Exploring Underproduction in Wikipedia. Proceedings of the 7th International Symposium on Wikis and Open Collaboration (pp 205–206). https://doi.org/10.1145/2038558.2038595

Gritzalis D, Iseppi G, Mylonas A, Stavrou V (2018) Exiting the Risk Assessment Maze: A Meta-Survey. ACM Computing Surveys 51(1):1–30. https://doi.org/10.1145/3145905

Hall T, Beecham S, Verner J, Wilson D (2008) The Impact of Staff Turnover on Software Projects: The Importance of Understanding What Makes Software Practitioners Tick. SIGMIS-CPR – Management Information Systems – Computers and People Research (pp 30–39)

Herraiz I, Shihab E, Nguyen TH, Hassan AE (2011) Impact of Installation Counts on Perceived Quality: A Case Study on Debian. In: 2011 18th Working Conference on Reverse Engineering (pp 219–228). Limerick, Ireland: IEEE. https://doi.org/10.1109/WCRE.2011.34

Hindle A, Herraiz I, Shihab E, Jiang ZM (2010) Mining Challenge 2010: FreeBSD, GNOME Desktop and Debian/Ubuntu. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (pp 82–85). Cape Town, South Africa: IEEE. https://doi.org/10.1109/MSR.2010.5463350

Hoffmann M, Nagle F, Zhou Y (2024) The Value of Open Source Software. SSRN Electronic Journal. https://doi.org/10.2139/ssrn.4693148

Jabrayilzade E, Evtikhiev M, Tüzün E, Kovalenko V (2022) Bus factor in practice. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice (pp 97–106). Pittsburgh Pennsylvania: ACM. https://doi.org/10.1145/3510457.3513082

Joblin M, Apel S (2022) How do successful and failed projects differ? A socio-technical analysis. ACM Transactions on Software Engineering and Methodology 31(4):1–24. https://doi.org/10.1145/3504003

Kerner SM (2014) Heartbleed SSL flaw's true cost will take time to tally. eWEEK. Retrieved: 2022-10-09. https://www.eweek.com/security/heartbleed-ssl-flaw-s-true-cost-will-take-time-to-tally/

Kim S, Whitehead EJ (2006) How long did it take to fix bugs? In: Proceedings of the 2006 International Workshop on Mining Software Repositories - MSR '06 (p 173). Shanghai, China: ACM Press. https://doi.org/10.1145/1137983.1138027

Lakhani KR, Wolf B (2005) Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In: Feller J, Fitzgerald B, Hissam SA, Lakhani KR (eds) Perspectives on Free and Open Source Software. MIT Press, pp 3–22

Le DM, Link D, Shahbazian A, Medvidovic N (2018) An Empirical Study of Architectural Decay in Open-Source Software. In: 2018 IEEE International Conference on Software Architecture (ICSA) (pp 176–17609). https://doi.org/10.1109/ICSA.2018.00027

Legay D, Decan A, Mens T (2020) On Package Freshness in Linux Distributions. In: 2020 IEEE International Conference on Software Maintenance and Evolution (pp 682–686). Adelaide, SA, Australia

Lyytinen K, Mathiassen L, Ropponen J (1998) Attention Shaping and Software Risk–A Categorical Analysis of Four Classical Risk Management Approaches. Inf Syst Res 9(3):233–255. https://doi.org/10.1287/isre.9.3.233

Meidan A, García-García JA, Ramos I, Escalona MJ (2018) Measuring Software Process: A Systematic Mapping Study. ACM Comput Surv 51(3):1–32. https://doi.org/10.1145/3186888

Meyers JS, Veytia A, Luhring D, Newman Z, Torres-Arias S (2022) Software dark matter is the enemy of software transparency. Retrieved from https://www.chainguard.dev/unchained/software-dark-matter-is-the-enemy-of-software-transparency

Michlmayr M, Hill BM (2003) Quality and the Reliance on Individuals in Free Software Projects. 3rd Workshop on Open Source Software Engineering

Michlmayr M, Senyard A (2006) A Statistical Analysis of Defects in Debian and Strategies for Improving Quality in Free Software Projects. The Economics of Open Source Software Development (pp 131–148). Elsevier. Retrieved 2020-07-18. https://doi.org/10.1016/B978-044452769-1/50006-8

Miller C, Widder DG, Kastner C, Vasilescu B (2019) Why do people give up FLOSSing? A study of contributor disengagement in open source. In: Bordeleau F, Sillitti A, Meirelles P, Lenarduzzi V (eds) Open Source Systems (Vol 556, pp 116–129). Springer International Publishing. https://doi.org/10.1007/978-3-030-20883-7_11

Mockus A (2010) Organizational volatility and its effects on software defects. In: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (pp 117–126). Santa Fe New Mexico USA: ACM. https://doi.org/10.1145/1882291.1882311

Narayanan S, Balasubramanian S, Swaminathan JM (2009) A Matter of Balance: Specialization, Task Variety, and Individual Learning in a Software Maintenance Environment. Management Science 55(11):1861–1876. https://doi.org/10.1287/mnsc.1090.1057

Nassif M, Robillard MP (2017) Revisiting turnover-induced knowledge loss in software projects. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME) (pp 261–272). Shanghai: IEEE. https://doi.org/10.1109/ICSME.2017.64

Natella R, Cotroneo D, Madeira HS (2016) Assessing Dependability with Software Fault Injection: A Survey. ACM Computing Surveys 48(3):1–55. https://doi.org/10.1145/2841425

Nussbaum L, Zacchiroli S (2010) The Ultimate Debian Database: Consolidating bazaar metadata for Quality Assurance and data mining. In: 2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010) (pp 52–61). Cape Town, South Africa: IEEE. https://doi.org/10.1109/MSR.2010.5463277

O'Neil M (2009) Cyberchiefs: Autonomy and authority in online tribes. Pluto Press, London

Pagliery J (2014) Your Internet security relies on a handful of volunteers. CNNMoney Retrieved 2021-11-11. https://money.cnn.com/2014/04/18/technology/security/heartbleed-volunteers/

Pasha M, Qaiser G, Pasha U (2018) A Critical Analysis of Software Risk Management Techniques in Large Scale Systems. IEEE Access 6:12412–12424. https://doi.org/10.1109/ACCESS.2018.2805862

Pati J, Shukla KK (2014) A comparison of ARIMA, neural network and a hybrid technique for Debian bug number prediction. In: 2014 International Conference on Computer and Communication Technology (ICCCT) (pp 47–53). Allahabad, India: IEEE. https://doi.org/10.1109/ICCCT.2014.7001468

Paul CL (2009) A Survey of Usability Practices in Free/Libre/Open Source Software. In: Boldyreff C, Crowston K, Lundell B, Wasserman AI (eds) Open Source Ecosystems: Diverse Communities Interacting, vol 299. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 264–273

Paul I (2014) In Heartbleed's wake, tech titans launch fund for crucial open-source projects. PCWorld, Retrieved: 2021-11-11. https://www.pcworld.com/article/438906

Perlroth N (2014) Heartbleed highlights a contradiction in the web. The New York Times, Retrieved: 2021-11-11. https://www.nytimes.com/2014/04/19/technology/heartbleed-highlights-a-contradiction-in-the-web.html
Price JL (1989) The Impact of Turnover on the Organization. Work and Occupations 16(4):461–473. https://doi.org/10.1177/0730888489016004005
Qiu HS, Nolte A, BrownA, Serebrenik A, Vasilescu B (2019) Going farther together: The impact of social capital on sustained participation in open source. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) (pp 688–699). Montreal, QC, Canada: IEEE. Retrieved:2023-03-25. https://ieeexplore.ieee.org/document/8812044/
Raftery AE (1995) Bayesian model selection in social research. Sociol Methodol 25:111. https://doi.org/10.2307/271063
Ray B, Posnett D, Filkov V, Devanbu P (2014) A large scale study of programming languages and code quality in github. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014 (pp 155–165). Hong Kong, China: ACM Press. https://doi.org/10.1145/2635868.2635922
Raymond ES (2001) The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary. In: O'Reilly T (ed) Sebastopol. CA, O'Reilly and Associates
Rhoades SA (1993) The Herfindahl-Hirschman Index Note. Federal Reserve Bulletin 79(3):88–189
Robles G, Gonzalez-Barahona JM, Michlmayr M (2005) Evolution of Volunteer Participation in Libre Software Projects: Evidence from Debian. In: Proceedings of the First Internation Conference on Open Source Systems (pp 100–107). Genova
Robles G, González-Barahona JM, Izquierdo-Cortazar D, Herraiz I (2009) Tools for the Study of the Usual Data Sources found in Libre Software Projects. Intl J Open Source Softw Proc 1(1):24–45. https://doi.org/10.4018/jossp.2009010102
Ronchieri E, Canaparo M (2018) Metrics for Software Reliability: a Systematic Mapping Study. J Int Design Proc Sci 22(2):5–25. https://doi.org/10.3233/JID180008
Ruiz C, Robinson WN (2011) Measuring Open Source Quality: A Literature Review. Int J Open Source Softw Proc 3(3):48–65. https://doi.org/10.4018/jossp.2011070104
Santos JAM, Rocha-Junior JB, Lins Prates LC, do Nascimento RS, Freitas MF, de Mendonca MG (2018) A systematic review on the code smell effect. J Syst Softw 144:450–477. https://doi.org/10.1016/j.jss.2018.07.035
Sargent DJ (1998) A General Framework for Random Effects Survival Analysis in the Cox Proportional Hazards Setting. Biometrics 54(4):1486. https://doi.org/10.2307/2533673
Sobrinho EVdP, De Lucia A, Maia MdA (2018) A systematic literature review on bad smells 5 W's: which, when, what, who, where. IEEE Trans Softw Eng 47(1):17–66. https://doi.org/10.1109/TSE.2018.2880977
Spaeth S, Stuermer M, Haefliger S, von Krogh G (2007) Sampling in Open Source Software Development: The Case for Using the Debian GNU/Linux Distribution. In: 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07). Waikoloa, HI: IEEE. https://doi.org/10.1109/HICSS.2007.471
Spaeth S, von Krogh G, Stuermer M, Haefliger S (2008) A Lightweight Model of Component Reuse: A Study of Software Packages in Debian GNU/Linux [Working Paper]. SSRN. https://ssrn.com/abstract=1153968
Stergiopoulos G, Kotzanikolaou P, Theocharidou M, Lykou G, Gritzalis D (2016) Time-based critical infrastructure dependency analysis for large-scale and cross-sectoral failures. Int J Critical Infrast Protect 12:46–60. https://doi.org/10.1016/j.ijcip.2015.12.002
Studer M (2007) Community structure, individual participation and the social construction of merit. In: Feller J, Fitzgerald B, Scacchi W, Sillitti A (eds) Open Source Development, Adoption and Innovation. Springer, US, pp 161–172
Tamburri DAA, Palomba F, Kazman R (2019) Exploring Community Smells in Open-Source: An Automated Approach. IEEE Trans Softw Eng 1–1. https://doi.org/10.1109/TSE.2019.2901490
Valiev M, Vasilescu B, Herbsleb J (2018) Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018 (pp 644–655). Lake Buena Vista, FL, USA: ACM Press. https://doi.org/10.1145/3236024.3236062
Verdon D, McGraw G (2004) Risk Analysis in Software Design. IEEE Security and Privacy 2(4):79–84
Villa L (2019) Bus factor, boss factor, and the economics of disappearing maintainers. https://blog.tidelift.com/bus-factor-boss-factor-and-the-economics-of-disappearing-maintainers

Villegas P, Munoz MA, Bonachela JA (2020) Evolution in the Debian GNU/Linux software network: analogies and differences with gene regulatory networks. J Royal Soc Interface 17(163):20190845. https://doi.org/10.1098/rsif.2019.0845

Walden J (2020) The Impact of a Major Security Event on an Open Source Project: The Case of OpenSSL. In: Proceedings of the 17th International Conference on Mining Software Repositories (pp 409–419). New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3379597.3387465

Warncke-Wang M, Ranjan V, Terveen L, Hecht B (2015) Misalignment between supply and demand of quality content in peer production communities. In: Proceedings of the Ninth International AAAI Conference on Web and Social Media (pp 493–502)

Wasserman S, Faust K (1994)Social Network Analysis: Methods And Applications. Cambridge University Press

Wheeler DA (2014) Preventing Heartbleed. Computer 47(8):80–83. https://doi.org/10.1109/MC.2014.217

Wiggins A, Howison J, Crowston K (2009) Heartbeat: Measuring Active User Base and Potential User Interest in FLOSS Projects. In: Boldyreff C, Crowston K, Lundell B, Wasserman AI (eds) Open Source Ecosystems: Diverse Communities Interacting, vol 299. Springer, Berlin Heidelberg, Berlin, Heidelberg, pp 94–104

Wong WE, Qi Y, Cooper K (2005) Source code-based software risk assessing. In: Proceedings of the 2005 ACM symposium on Applied computing -SAC '05 (p1485). Santa Fe, New Mexico: ACM Press. https://doi.org/10.1145/1066677.1067014

Xiao W, He H, Xu W, Zhang Y, Zhou M (2023) How Early Participation Determines Long-Term Sustained Activity in GitHub Projects? In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp 29–41). San Francisco CA USA: ACM. https://doi.org/10.1145/3611643.3616349

Zampetti F, Serebrenik A, Di Penta M (2020) Automatically Learning Patterns for Self-Admitted Technical Debt Removal. In: 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 355–366). London, ON, Canada: IEEE. https://doi.org/10.1109/SANER48275.2020.9054868

Zerouali A, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 589–593). Hangzhou, China: IEEE. https://doi.org/10.1109/SANER.2019.8667997

Zerouali A, Mens T, Robles G, Gonzalez-Barahona JM (2019) On the Relation between Outdated Docker Containers, Severity Vulnerabilities, and Bugs. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER) (pp 491–501). Hangzhou, China: IEEE. https://doi.org/10.1109/SANER.2019.8668013