
# Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem

Marat Valiev
Carnegie Mellon University
USA

Bogdan Vasilescu
Carnegie Mellon University
USA

James Herbsleb
Carnegie Mellon University
USA

## ABSTRACT

Open-source projects do not exist in a vacuum. They benefit from reusing other projects and themselves are being reused by others, creating complex networks of interdependencies, *i.e.*, software ecosystems. Therefore, the sustainability of projects comprising ecosystems may no longer by determined solely by factors internal to the project, but rather by the ecosystem context as well.

In this paper we report on a mixed-methods study of ecosystem-level factors affecting the sustainability of open-source Python projects. Quantitatively, using historical data from 46,547 projects in the PyPI ecosystem, we modeled the chances of project development entering a period of dormancy (limited activity) as a function of the projects' position in their dependency networks, organizational support, and other factors. Qualitatively, we triangulated the revealed effects and further expanded on our models through interviews with project maintainers. Results show that the number of project ties and the relative position in the dependency network have significant impact on sustained project activity, with nuanced effects early in a project's life cycle and later on.

## CCS CONCEPTS

• **Software and its engineering → Open source model**; **Software evolution**; Risk management;

## KEYWORDS

Software ecosystems, Open Source, Survival modeling

**ACM Reference Format:**
Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-Level Determinants of Sustained Activity in Open-Source Projects: A Case Study of the PyPI Ecosystem. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3236024.3236062

## 1 INTRODUCTION

While only twenty years ago open-source software (OSS) was simply a curiosity that attracted the attention of a few academics and was not seriously considered in the software industry, OSS infrastructure today is ubiquitous,[1] powering applications in virtually every domain. Economists refer to OSS as "digital dark matter" [34], to signify both its invisibility and importance. They also report valuations of OSS in the billions of dollars per year [20, 34], in terms of both direct reuse value and boosted productivity and efficiency.

Given the importance of OSS digital infrastructure to so much of the economy, one might expect that it is adequately staffed and maintained, *i.e.*, sustainable. Yet, this is often not the case. As a recent Ford Foundation report investigating the sustainability of OSS "digital infrastructure" [26] notes, most users of OSS infrastructure take it for granted, and society at large is unaware of the risks. A vivid example is OpenSSL, the OSS project critical to the secure operation of the majority of websites, recently in the spotlight for the "heartbleed" security bug [24]; at that time, Open SSL was severely understaffed. Another notable example is leftpad [1], the trivial 11-LOC JavaScript package that, when deleted by its author from the npm[2] registry, caused cascading disruption in thousands of other projects that relied on it being accessible on npm.

Besides emphasizing the importance of OSS sustainability issues, both examples illustrate a challenge with modern code reuse. Indeed, with vast amounts of high-quality OSS code available for reuse, one can declare dependencies on others' code instead of copying it into their own, taking advantage of the functionality without assuming the burden of maintenance. This leads to the formation of large code interdependency networks. However, this also means that local sustainability issues around individual projects can have widespread network effects. For example, breaking changes—changes that are not backwards compatible—are a significant source of instability, causing negative consequences for dependents downstream [5]. Another example is developer turnover. Contributors to and maintainers of OSS are often overworked volunteers, who can decide to stop contributing at any time [58, 74]. Consequently, OSS projects risk knowledge loss [52, 57], quality degradation [27], or even extinction [10], again with reverberations downstream.

These challenges are particularly visible in OSS package ecosystems like npm, PyPI,[3] and CRAN,[4] where packages form complex and often brittle dependency chains [21, 41]. With reuse so enticing and so much OSS code available, how can one make informed decisions about which packages to use? While OSS projects can be long-lived (*e.g.*, Linux, Apache, and Eclipse), relatively few reach a mature state [7, 12] and many that are active for a period of time are eventually abandoned [40], even once-popular ones [10]. Will a package still be maintained in a year? Which packages are


*ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA*

ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236062

[1] Already in 2015, less than 3% of respondents to a Black Duck Survey reported they *do not* use OSS in any way, https://bit.ly/2NgQNRH (slide 9).
[2] Node.js Package Manager, https://www.npmjs.com
[3] Python Package Index, https://pypi.python.org
[4] Comprehensive R Archive Network, https://cran.r-project.org

sustainable and which are at risk? How does a package's place in the ecosystem influence its survival chances? How do developer choices, ecosystem community norms, and social processes contribute to sustainability or extinction? The empirical evidence for the mechanisms and predictive factors of OSS project survival *in an ecosystem context* is, at best, fragmented and incomplete.

In this paper we report on a mixed-methods study of the Python PyPI ecosystem, that makes a step towards filling this gap. PyPI is the official third-party registry for Python packages and one of the most popular OSS ecosystems, with over 130,000 published packages as of March 2018. Specifically, we study the ecosystem-level factors impacting the chances of a package becoming dormant, *i.e.*, having very low or no development activity after some time. While not all dormant projects are abandoned (*e.g.*, some simply do not require any additional maintenance because they are feature complete [10]), being in an inactive state could signal sustainability risk. For example, for an external observer, lack of project activity may indicate abandonment and increased uncertainty about whether potential issues or feature requests would be dealt with.

We interview maintainers of PyPI packages; integrate data from PyPI and GitHub, mining repositories and their interdependencies to assemble an ecosystem-level longitudinal data set; identify which packages became dormant; and estimate Cox proportional hazards survival regressions [15, 49] to model the factors affecting a package's chances of entering this dormant state.

We find that the number of connections and the relative position in the dependency network are significant factors affecting the chances of a project becoming dormant; the organizational support a package receives, if any, has different effects depending on the type of supporting organization; and the practice of producing backwards compatible releases does not appear to influence project dormancy under our definition.

In summary, we contribute (1) a dependency-network-based survival analysis of packages in the PyPI ecosystem; (2) a series of interviews with project maintainers from this ecosystem to triangulate and refine the discovered relationships; and (3) an in-depth discussion of the effects revealed by the mixed-methods analysis.

## 2 DEVELOPMENT OF HYPOTHESES

As with natural systems, the sustainability of OSS projects (much like the success of OSS projects [17]) is also clearly a multi-faceted concept; *e.g.*, projects may be considered sustainable from a code maintainability perspective if they conform to modular and extensible architectures, from a community perspective if they successfully attract and retain newcomers; and from an economic perspective if they ensure low total cost of ownership and high added value. Our perspective in this paper is that of *OSS supply chains* [4]: Given the choice, should one depend on some OSS package? Will it be actively maintained in a year or will it show no signs of life?

To develop our hypotheses, we start by reviewing the literature on factors impacting the survival of OSS projects, defined here as the state of being actively maintained. We distinguish between project-level factors, of which having an appropriate supply of contributor effort is arguably most important, and ecosystem-level factors, induced by projects' position in an ecosystem and their relationship with other projects up and downstream. The project-level factors are relatively well studied, therefore we use the literature review to identify relevant control variables in our regression models. The ecosystem-level factors constitute our main contribution. For these we derive, and later test, explicit hypotheses.

**Project-level Factors.** In order to survive and thrive, OSS projects typically require a steady supply of contributor effort, and projects with more contributors tend to have higher survival rates [59]. But not all contributions are created equal. The communities supporting OSS projects are typically organized in layers, with different roles being recognizable among participants [38, 51]. Usually, project activity is driven by a few *core contributors*, who have commit (*i.e.*, "write") access to the repository and do most of the work. Ascension into the core group is a socio-technical process; earning committer status involves socializing with the core group [23, 30, 64] and demonstrated commitment through repeated, high-quality contributions [18, 70]. The next layer, larger, comprises *external contributors*, who submit occasional patches; on GitHub, these occasional contributions are popular with the pull-based development model [32, 55]. Next, there is typically a layer of *contributing users*, who may participate in discussions or report issues without contributing code [75]. Finally, the outermost layer consists of *external users* of the software, who do not necessarily participate in any project activities.

Core contributors, or maintainers, are paramount to the survival of OSS projects. They are highly active and have the deepest knowledge of the code base, making them the hardest to replace. Activity in OSS projects typically follows the Pareto principle [31, 50, 73], by which 20% of contributors are responsible for 80% of all activity; to capture this phenomenon, different measures of risk of knowledge loss due to developer turnover have been proposed [57, 69], including the popular "truck factor" [2, 14]. Other contributors and users are also important: future maintainers are frequently groomed or ascend from among external contributors [51]; external contributors also provide much needed testing and quality assurance ("given enough eyeballs, all bugs are shallow" [56]); and without users the software would quickly become obsolete.

When contributions come is also important. OSS projects, as with projects generally, have a life cycle, from inception to abandonment. The motivations for contributing to a project, the amount of effort a project may need, and the chances of attracting contributors will likely vary with the stage in the life cycle. The code growth curve of OSS projects often follows the typical pattern of rapid growth slowing and flattening as projects reach maturity and require less effort for adding features, as shown, *e.g.*, in GNOME [42] and Linux stable releases [65]. Relatively few OSS projects reach maturity [7, 12] and even once-popular projects can get abandoned, *i.e.*, no longer maintained [10, 40]. However, the factors associated with sustained activity can be different in early-stage projects compared to later on [60, 71]. For example, Comino *et al.* [12] found that fewer than 2% of a sample of SourceForge projects reached maturity, and that early-stage projects risked abandonment due to restrictive licenses and smaller communities. In contrast, Coelho and Valente [10] found that common reasons why mature and once-popular OSS projects are abandoned include losing out to a competitor, having become obsolete, and lack of time and interest from contributors.

**Ecosystem-level Factors.** OSS ecosystems have been an active research topic (for a review, see Franco-Bedoya *et al.* [28]) and different definitions exist [6, 36, 45, 47]. Here we follow Lungu's broad definition [46] of OSS ecosystems as collections of related software projects that co-evolve in the same environment. Python packages published on PyPI fit this definition: they coexist in the PyPI environment and, as we show below, they are often interdependent.

Within an OSS ecosystem, developers frequently contribute to multiple projects [68], often at the same time [66]. In addition to building social capital, bridging sub-communities also creates connections between projects, which can impact project sustainability. For example, Casalnuovo *et al.* [8] found that GitHub contributors are more likely to join projects with which they have prior social connections. Singh *et al.* [61] showed, using a longitudinal panel of 2,378 SourceForge projects, that social network ties between developers impact OSS project survival. Finally, Wang [71] analyzed 2,220 SourceForge projects to model survival factors at various project lifecycle stages. The author found that member social connections with other projects and active engagement of contributors present survival advantages at any stage, while permissive licenses and large contributor bases help especially early on.

A significant missing puzzle piece in prior research on OSS sustainability is the impact of a project's position in dependency networks on survival or extinction, though the heartbleed and leftpad incidents discussed above suggest this might be significant. Indeed, ecosystem connections between projects extend well beyond the social, with complex dependency networks being formed, *i.e.*, one project reusing functionality from another [1, 3, 21, 41].

We argue that the survival of an OSS project in an ecosystem, in addition to all the factors reviewed above, depends also on the survival of its dependencies upstream (*i.e.*, other packages the current package depends on) and downstream (*i.e.*, packages depending on the current package). Specifically, while the benefits of depending on others' code in an ecosystem—reusing functionality without assuming the responsibility of maintenance—are clear, we expect that in general having more upstream dependencies may create more points of failure, because of the costs associated with responding to breaking (*i.e.*, backward incompatible) changes. Indeed, while different OSS ecosystem communities have different practices in planning and deploying breaking changes [5], it is clear that in all ecosystems developers must constantly make dependency management decisions. We expect that:

**H$_1$**. *The number of upstream dependencies is related to a lower probability of project survival.*

However, depending on an upstream package may also create an incentive to step up to make changes that benefit you as a user or to help more generally, because an ill maintained upstream project could increase maintenance effort downstream. Therefore, the more downstreams, the larger the pool of potential resources available upstream if needed. This may help explain how the original issues were resolved within hours for leftpad and days for OpenSSL:

**H$_2$**. *The number of downstream dependencies is related to a greater probability of project survival.*

In addition, in both incidents, a catalyst to the massive disruption is also the fact that among immediately affected dependencies were important network actors. For OpenSSL these were popular web servers while leftpad was used by Babel, a core JavaScript package. The number of direct dependents may not fully reflect a package's importance. Consequently, we posit:

**H$_3$**. *Structural properties indicating more indirect connectivity through transitive dependencies are related to a greater probability of survival.*

In contrast, there are upstream practices that can help mitigate these downstream costs. As concrete evidence of expending effort to support a community of users we consider backporting, which may indicate more intense involvement with other projects in the extended dependency network. Hence, we posit that:

**H$_4$**. *Backporting is related to a higher probability of project survival.*

There is also a social organizational perspective to thriving in an ecosystem. Besides differences in roles, OSS contributors are also diverse in terms of background, demographics [67], and employment; they can be a mixture of volunteers, academics, and paid contributors [9, 76], with different motivations [22, 35, 43]. In particular, as ecosystem dependency management costs may become significant over time, we posit that whether a big organization (commercial, non-profit, or even academic) supports an OSS project will affect the project's survival chances, as this level of investment (*e.g.*, assigned employees) is likely beyond that found among volunteers:

**H$_5$**. *Projects supported by large organizations have a higher probability of survival.*

## 3 METHODOLOGY

To test our hypotheses we designed a mixed-methods study following a *concurrent triangulation strategy*, a common mixed-methods design [25]. We collected both quantitative and qualitative data concurrently and used findings from one source as cross-validation for findings from the other. Quantitatively, we collected a panel data set of Python PyPI packages and used survival analysis to model the factors that explain projects becoming dormant. Qualitatively, we interviewed package maintainers to triangulate the model results and refine the discovered effects. Because of a concurrent rather than sequential triangulation strategy, we could revisit and enhance the model to account for potential effects revealed by the interviews, which is a particular strength of this design [16].

### 3.1 Data Set

We assembled a large panel data set[5] of OSS packages part of the PyPI (Python Package Index) ecosystem. A distinctive feature of our data set is that it *accurately represents the network of dependencies between member packages* (details below).

We chose Python as it is a popular general purpose language; it is the second most popular on Github by number of pull requests.[6] PyPI is the official registry of Python packages, forming an ecosystem comprising over 130,000 packages (as of March 2018), with declarative-style dependencies. Unlike other languages (*e.g.*, Haskell), Python has only one package repository; as Figure 1, based on our data (details below), shows, PyPI is increasingly popular.

**Initial Filtering.** Assembling our data set involved integrating data from two sources: metadata from the PyPI registry and the packages' development history from their GitHub repositories.

[5] Available online at https://zenodo.org/record/1297925.
[6] https://octoverse.github.com/, retrieved February 2018

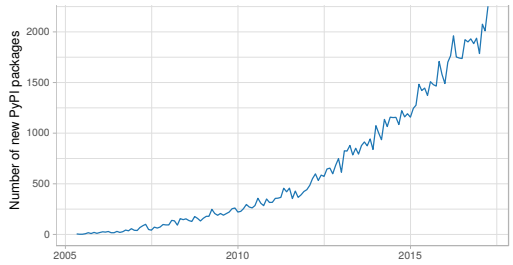


**Figure 1: Number of new PyPI packages per month.**

Linking the two required several steps. First, we obtained a list of all PyPI packages using PyPI's JSON API[7] on January 21, 2018, for a total of 125,699 packages, 116,687 of which had at least one release. Next, for each package, we checked if its *home page* field in the PyPI metadata matches any popular code hosting platforms (github.com, bitbucket.org, gitlab.com). If this failed, we extended the lookup to all other metadata fields. If no URL was found, we downloaded the last package release and looked for mentions of code hosting platform URLs across all package files, with the repository name, if any, matching the package name on PyPI. This approach revealed 91,728 package repositories overall, 89% (or 81,802) of which were hosted on GitHub; for simplicity, we subsequently only mined packages with GitHub repositories. These repositories were checked for existence and uniqueness to filter out project foundries (*i.e.*, repositories hosting hundreds of projects,[8] since these all point to the same PyPI package and the metadata would be impossible to disentangle), leaving 71,903 code repositories (packages) total. We further filtered out packages created before 2012, when GitHub became popular, as the data pre-2012 is sparse, and packages with less than a year of observable history, *i.e.*, created after January 2017, since we could not confidently label them as dormant or still maintained (see §3.2). Our final sample contains 46,547 packages.

**Dependency Network.** By "dependency", we mean a *declared technical dependency on another PyPI package*. That is, we do not count required system libraries nor Python packages copied to the source tree of a package. We also exclude optional extras and test dependencies, as the vast majority of installations do not use them. Given a package, we call "upstreams" those packages used by this package, *i.e.*, packages that the focal package depends on; conversely, we call "downstreams" those packages dependent on the focal one.

To extract dependencies, we mined the package metadata whenever possible, and used a sandbox installation as a fallback. PyPI supports several packaging formats, two of which (.egg and .whl) store machine-readable dependencies. For other formats, *e.g.*, source archives, we executed a package installation in a sandbox environment with an instrumented version of the package installer, logging the requested dependencies.

To capture network dynamics we extracted dependency information from *all releases of all PyPI packages*. Then, we generated historical snapshots of the network, using the latest non-testing, non-backported release at each time. Testing releases were inferred from version numbers, using semantic versioning assumptions (*i.e.*, not matching a pattern of dots and numbers only). Backporting releases are defined as a release with lower version number than the highest non-testing release (*e.g.*, 1.10.0 is a backporting release if 2.0 was released earlier). Note that we extracted dependencies over the whole PyPI network, not just for packages with GitHub repositories; this means our network structural measures are robust to the initial filtering above.

[7] https://wiki.python.org/moin/PyPIJSON
[8] E.g., https://github.com/micropython/micropython-lib, 220 projects

**Table 1: Data set summary statistics (Dec 2017 snapshot)**

| Quantile | Min | Mean | 0.5 | 0.7 | 0.8 | 0.9 | 0.95 | 0.97 | 0.99 | Max |
|---|---|---|---|---|---|---|---|---|---|---|
| **All projects** | | | | | | | | | | |
| Commits | 0 | 1.9 | 0 | 0 | 0 | 2 | 8 | 15 | 42 | 1144 |
| Core team | 0 | 0.07 | 0 | 0 | 0 | 0 | 0 | 1 | 2 | 97 |
| Non_dev_issues | 0 | 0.2 | 0 | 0 | 0 | 0 | 1 | 1 | 4 | 245 |
| Upstreams | 0 | 1.7 | 1 | 2 | 3 | 4 | 6 | 8 | 14 | 117 |
| Downstreams | 0 | 1.5 | 0 | 0.0 | 0 | 1 | 2 | 4 | 15 | 5487 |
| Katz centrality | 0 | 3e-4 | 4e-4 | 4e-4 | 4e-4 | 5e-4 | 5e-4 | 6e-4 | 1e-3 | 0.33 |
| Social ties | 0 | 1.0 | 0 | 0 | 0 | 0 | 1 | 3 | 12 | 240 |
| University | 0 | 0.01 | 0 | 0 | 0 | 0 | 0 | 0 | 0.14 | 1 |
| Commercial | 0 | 0.03 | 0 | 0 | 0 | 0 | 0 | 0.5 | 1 | 1 |
| **Non-dormant projects** | | | | | | | | | | |
| Commits | 0 | 11.6 | 4 | 8 | 14 | 27 | 48 | 66 | 125 | 1144 |
| Core team | 0 | 0.42 | 0 | 0 | 1 | 1 | 2 | 3 | 4 | 97 |
| Non_dev_issues | 0 | 0.9 | 0 | 0 | 1 | 2 | 4 | 6 | 14 | 245 |
| Upstreams | 0 | 2.2 | 0 | 2 | 4 | 6 | 9 | 12 | 20 | 100 |
| Downstreams | 0 | 5.3 | 0 | 0 | 0 | 2 | 5 | 12 | 67 | 5487 |
| Katz centrality | 0 | 5e-4 | 4e-4 | 4e-4 | 4e-4 | 5e-4 | 6e-4 | 1e-3 | 4e-3 | 0.33 |
| Social ties | 0 | 6.2 | 0 | 1 | 3 | 6 | 15 | 32 | 211 | 240 |
| University | 0 | 0.05 | 0 | 0 | 0 | 0 | 0.3751 | | 1 | 1 |
| Commercial | 0 | 0.18 | 0 | 0 | 0.3331 | | 1 | 1 | 1 | 1 |

## 3.2 Operationalizations of Concepts

In preparation for the quantitative survival analysis below (details in §3.3), we introduce the following operationalizations of the different project- and ecosystem-level factors discussed in §2 above.

**Dormant projects.** We consider project as dormant if it is no longer being maintained, *i.e.*, when it stops development (commit) activity, in line with prior work [10, 61, 71]. This suggests a straightforward approach to detect dormant projects: look for a long period of inactivity in the git history, and consider the timestamp of the latest commit as the dormant date. However, this is not always accurate: *e.g.*, there are instances when a seemingly dormant project, with an activity gap of one year or more, is "revived" by (a few) commits to officially indicate that the project has been deprecated; dormant projects may have also had little activity to begin with (thus long gaps are not unusual), changed owner, or were not as much abandoned as they were "completed" [10] (*i.e.*, they continue to deliver the intended value without active maintenance; 11% of developers interviewed by Coelho and Valente [10] reported this).

To increase robustness to residual development activity (*i.e.*, fewer false positives), we instead only label a project as dormant if it had less than one commit per month on average in the 12 months prior to its most recent commit. This fits well with our manual inspections of small samples of the data (tens of packages), though we do acknowledge this definition as a potential threat to validity.

**Project-level Control Variables.** As indicated in our literature review, several factors are known to impact survival rates, and we use the following variables to include these factors in our models:

*Project age*: the number of months since the first commit in the repository. Note that the earliest Git commit is sometimes dated unreasonably early, *e.g.*, because of a system time reset on a developer machine (dead CMOS battery). A true first commit is one without a parent in the Git history graph. We identified true first commits and filtered out outlying commits dated before their timestamps.

*Number of commits*: obtained via the GitHub API and aggregated per calendar month; consequently, since the first month may be incomplete, we exclude it from further analysis.

*Number of contributors*: counted as the number of GitHub users having authored commits within a given calendar month.

*Size of the core team*: the number of people responsible for 90% of contributions in a given month. This threshold was selected empirically as a typical "elbow" point in distribution of OSS activity, much like in other OSS projects (*e.g.*, Apache [50]).

*Number of issues*: obtained via the GitHub API, with pull requests filtered out, aggregated per calendar month. We distinguish between developer-reported issues, likely occurring internally during development, and non-developer-reported issues, likely reported by external users, as the latter are more indicative of the size of the user base; we call "developers" those contributors who authored prior commits and "non-developers" the rest.

*Number of non-developer issue reporters*: the number of non-developer GitHub users reporting issues in a given calendar month; may help distinguish communities with higher user engagement from those where few users do most issue reporting.

*License type*: extracted from PyPI package metadata; categorical variable, indicating whether a project is distributed under a strong copy-left license (GPL, Affero etc.), weak copy-left (LGPL, MPL, OPL, etc.) or non-copy-left license (Apache, BSD etc.), cf. [71].

*Social ties*: the total number of PyPI packages that contributors to the focal package also contributed to this month, as a proxy for the amount of OSS embeddedness of the contributors. Projects with more "seasoned" contributors may be more sustainable.

**Variables for Ecosystem-level Hypotheses.**

*Number of upstreams* (out-degree centrality): number of upstream dependencies used by the project ($\mathbf{H}_1$).

*Dormant upstreams*: binary variable indicating whether any of the upstream dependencies is itself dormant at the time ($\mathbf{H}_1$).

*Number of downstreams* (in-degree centrality): number of projects directly dependent on the focal project ($\mathbf{H}_2$). Both here and in the upstreams case, we do not differentiate between versions of a dependency, so even dependencies on old and unsupported versions of the package are counted. This is because more potential contributors dependent on this package and having a vested interest in its sustainability may positively affect survival.

*Katz centrality*: as a proxy for the importance of a package's position in the network structure ($\mathbf{H}_3$). Katz centrality [39] for node $i$ is defined as: $C_{Katz}(i) = \alpha \sum_{j \in J} C_{Katz}(j) + \beta$, where $J$ is the set of adjacent nodes to $i$. Parameter $\beta$ is an initial centrality (usually 1), and $\alpha$ is a discriminating factor applied at each step to down-weigh farther nodes (0.1 in this study). Developers further downstream in the dependency chain may be less likely to step in if needed than people from immediately dependent projects. We use Katz centrality to capture this and down-weigh farther downstream projects depending on their dependencies.

*Backporting*: binary variable indicating whether the project produced a backporting release in the last 12 months ($\mathbf{H}_4$).

*Organizational account*: binary variable indicating that the project is hosted under an organizational (rather than personal) GitHub account; organizations, even informal, may possess more resources than an individual developer, affecting survival differently ($\mathbf{H}_5$).

*University involvement*: share of commits where the top-level domain of the author's email is a university domain.[9] This is a conservative operationalization, as the list may be incomplete and not all academically affiliated contributors configure their Git clients with their institutional emails. Predominantly academic projects may be subject to specific survival risks, *e.g.*, student graduation, funding cycles, and shifting research interests ($\mathbf{H}_5$).

*Organizational involvement*: the share of commits from non-university, non-public, non-personal email domains. Public email providers (*e.g.*, gmail.com) were excluded based on a public list.[10] Personal domains are defined as those with only one known user across the entire ecosystem data. We manually validated the top-100 domains (by number of emails) labeled organizational and found no obvious mislabelling. Commercial companies or open-source foundations, both of which are "organizational", may act as a driving force and supply resources, *e.g.*, their employees' time ($\mathbf{H}_5$).

## 3.3 Survival Analysis (Quantitative)

We use survival analysis to model the effects of the different factors above on packages becoming dormant. Survival analysis, also known as event history analysis, is a branch of statistics that specializes in modeling of time to event data [49]. Typically only a single event occurs for each subject; in our case, the event is the package suspending its development activity. Survival analysis techniques are designed to deal with so-called right-censored observations: the time of the occurrence of the event of interest can only be recorded reliably for members of the population that already experienced the event; for others, all we know for certain is that the event hasn't happened yet; for some, it may never happen (hence the term right-censorship). In software engineering, survival analysis has been used to model, *e.g.*, defect survival in Eclipse (time to bug fixes) [72] and contributor survival in OSS projects [13, 44, 53].

**Cox Proportional-Hazards Model.** Different survival analysis techniques exist. The most common regression modeling framework for survival analysis is the Cox proportional-hazards model [15], which allows to estimate the effect of any one independent variable on the outcome, *while holding other covariates fixed*. This allows us to precisely isolate the effects of any given factor on survival.

In a general case, one may be interested in modeling state transitions in some system. Say we have a number of observation of some system, entering (*e.g.*, birth) and leaving (*e.g.*, death) a state of interest. For each alive subject, we thus have a *survival time* $T$ on record. The probability of reaching a given survival time $t$ will be defined by the *survival function* $S(t) = P(T > t)$. The probability of leaving the state at time $t$ will be given by *hazard rate* $h(t) = \frac{P(T<t+\Delta t | T \geq t)}{\Delta t}$.

[9] As per a public list https://github.com/Hipo/university-domains-list
[10] https://gist.github.com/tbrianjones/5992856/

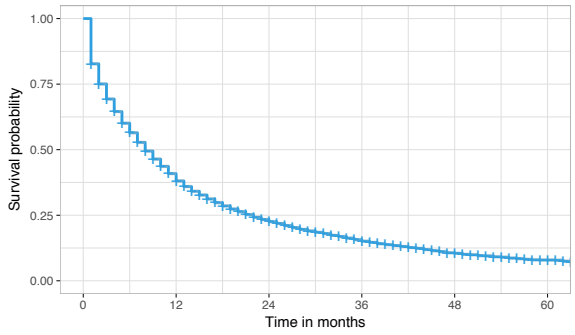


**Figure 2: Overall probability of survival across our sample.**

Given enough data, one can build a non-parametric regression to estimate all these functions.

Our goal, however, is to estimate the effect of some independent variables $X$ on the hazard rate: $h(t, X) = \theta(t)f(X)$. The problem in this case is that the baseline hazard rate $\theta(t)$ is non-parametric and thus does not have a functional equivalent. Cox's proportional hazard model allows to estimate coefficients of the regression $h(t, X) = \theta(t)\exp(\beta^T X)$ using partial likelihood, without any assumptions about the baseline hazard rate [37, 49].

A nice property of this model is that one can directly interpret the coefficients $\beta$. For example, if $\beta_i = 2$, then every unit increase in $X_i$ will increase the probability of survival by $\exp(2) = 7.4$ times.

**Modeling Considerations.** Recall that our data set is longitudinal, organized in monthly windows. Measures derived from OSS data (*e.g.*, number of commits) tend to be quite noisy, with high variation from one observation (*e.g.*, month) to the next. To increase the robustness of our models to potentially high window-to-window variance, we first smoothed out all numerical variables using a six-months sliding window, where each value was replaced by the average of the previous six.[11] Then, we split project data into six-month periods, taking only the last observation from each period.

As expected (§2), we also observed during initial exploration of our data that many packages become dormant early; see Figure 2, which shows the probability of survival (*i.e.*, non-dormant) over time, across all PyPI packages in our sample. To model how the different factors contribute to explaining the variability in package survival rates differently early-stage compared to later on, we split the data set into two parts: early-stage shutdowns (*i.e.*, stopping or nearly stopping development in the first six months, which matches well our sliding-window smoothing approach) and the rest.

The early-stage shutdowns, by definition given our smoothing, only contribute one observation each, while the other packages contribute more. Therefore, we model this group using logistic regression (`glm` in R), with the response variable being the likelihood of a project becoming dormant. For the remaining packages, the data contains monthly observations. A package's `dormant` variable is `True` in the last month, if we labeled the package as having stopped activity (see above), and `False` otherwise; surviving packages have, therefore, `dormant = False` in all windows. To model these, we estimate a Cox proportional-hazards model (`coxph` in R).

[11] Or fewer, in the first five months of observation.

In both cases, we follow a similar procedure for model fit and diagnostics. First, for predictors with highly skewed distributions, we conservatively removed the top 1% of values as high-leverage outliers, in line with statistical best practice [54]; high-leverage points would disproportionally affect regression slopes and reduce the model's robustness. Second, we log-transformed variables with skewed distributions, as needed, to reduce heteroscedasticity [29]; this helps stabilize the variance and can improve model fit. Third, we test for multicollinearity between the explanatory variables using the variance inflation factor (VIF), and remove variables, if any, with VIF scores above the recommended maximum of 5 [11]. We also performed graphical diagnostics: deviance residual plots for the logistic regression and Schoenfeld residuals [33] for the Cox model (which test the assumption of constant hazard ratios over time); none displayed any obvious signs of violations. In the Cox model, to account for the non-independence of repeated observations per package, we explored different options, all of which produced qualitatively similar results: transforming the data into count process format [62]; or using a cluster term for package.

When interpreting the models, we consider coefficients important if they are statistically significant at 0.05 level or lower, and we estimate their effect sizes from ANOVA type-2 analyses (column "LR Chisq" in Table 2). For the logistic model we also report McFadden's pseudo $R^2$ measure of goodness of fit.

## 3.4 Maintainer Interviews (Qualitative)

To triangulate and enhance our modeling results we conducted 10 semistructured interviews with package maintainers. Recruitment was done by soliciting via email, using addresses collected from GitHub profiles or personal websites. We used stratified sampling to identify potential interviewees: 3 packages with extreme feature values (large size), 2 projects that recently became inactive, 4 randomly selected to stratify the sample by project size, and 1 highly productive individual contributing to many projects of different sizes. For each project, the most active person in the last two years was solicited via email. We sent 32 emails, received 12 responses, and conducted 10 interviews; we reached theoretical saturation, meaning roughly that subsequent data all fit within the categories derived from the previous interviews, around the sixth interview.

The interview protocol was centered around the model features, asking about the predictive power of these features, their expected effect, and comments on the effects discovered by the model. Interviews also included several open ended questions about the definition of sustainability in OSS, project context, and missing factors that should be incorporated into the model.

Interviews were recorded, transcribed, translated (two cases), and coded. For three interviews, interview recordings were partially or completely corrupted due to technical glitches. Transcripts for these interviews were restored from partial recordings and notes, confirming accuracy with participants when necessary.

A lightweight qualitative coding was made by one author and discussed with the others. Codes were designed to match model features, their possible explanations, and threats to validity.

**Table 2: Regression models for early-stage survival and later-stage survival.**

| | **Early-stage survival** response: *dormant = TRUE* Pseudo $R^2$ = 43.6% | | **Later-stage survival** response: *dormant = TRUE* $R^2$ = 17% | | $R^2$ = 17.2% | |
|---|---|---|---|---|---|---|
| | Coeffs (Err.) | LR Chisq | Coeffs (Err.) | LR Chisq | Coeffs (Err.) | LR Chisq |
| (Intercept) | 3.96 (0.05)*** | | | | | |
| Log number of commits | 3.24 (0.02)*** | 7675.27*** | 1.77 (0.01)*** | 3317.06*** | 1.84 (0.01)*** | 3326.17*** |
| Log number of contributors | 0.00 (0.08)*** | 8276.50*** | 0.19 (0.05)*** | 1374.33*** | 0.19 (0.05)*** | 1372.20*** |
| Log number of non-developer issues | 1.91 (0.07)*** | 79.22*** | 0.55 (0.04)*** | 222.92*** | 1.07 (0.11) | 222.93*** |
| Social ties | 1.12 (0.03)*** | 16.42*** | 1.09 (0.02)*** | 17.58*** | 1.09 (0.02)*** | 18.30*** |
| Number of downstream projects | 1.60 (0.04)*** | 178.37*** | 0.89 (0.02)*** | 68.05*** | 0.89 (0.02)*** | 68.08*** |
| Number of upstream dependencies | 1.25 (0.01)*** | 380.27*** | 0.95 (0.01)*** | 68.25*** | 0.95 (0.01)*** | 19.05*** |
| Some upstreams are dormant | 0.69 (0.05)*** | 50.39*** | 1.11 (0.03)*** | 13.23*** | 1.11 (0.03)*** | 13.00*** |
| Katz centrality | 1.12 (0.02)*** | 35.29*** | 1.27 (0.02)*** | 221.55*** | 1.26 (0.02)*** | 171.36*** |
| High university involvement (>10%) | 1.08 (0.06) | 1.32 | 0.75 (0.05)*** | 30.88*** | 0.76 (0.05)*** | 30.35*** |
| High commercial involvement (>10%) | 1.51 (0.04)*** | 125.22*** | 1.15 (0.03)*** | 24.51*** | 1.15 (0.03)*** | 23.47*** |
| Had a backporting release in the last 12 months | 0.61 (0.12)*** | 16.63*** | 0.97 (0.07) | 0.21 | 0.97 (0.07) | 0.17 |
| Strong copy-left license (vs none) | 0.62 (0.06)*** | 78.42*** | 0.83 (0.04)*** | 35.83*** | 0.84 (0.04)*** | 34.54*** |
| Weak copy-left license (vs none) | 0.69 (0.10)*** | | 0.75 (0.07)*** | | 0.75 (0.07)*** | |
| Non-copy-left license (vs none) | 0.82 (0.04)*** | | 0.98 (0.03) | | 0.98 (0.03) | |
| Hosted under organizational account on GitHub | 0.55 (0.04)*** | 259.95*** | 0.77 (0.03)*** | 84.62*** | 0.78 (0.03)*** | 79.91*** |
| Number of upstream dependencies, squared | | | | | 1.00 (0.00) | 0.08 |
| Log num. commits × log num. issues | | | | | 0.81 (0.03)*** | 43.60*** |
| Log num. issues × log num. contributors | | | | | 0.91 (0.11) | 0.75 |

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

## 4 RESULTS AND DISCUSSION

We present an integrated discussion of quantitative and qualitative results, combining the survival analysis with interview insights. Table 2 presents the regression results. The first two models, logistic regression for projects becoming dormant in the early-stage (first six months), and Cox proportional-hazards for those becoming dormant later on, comprise the factors we reviewed or hypothesized in §2. The third model extends the Cox proportional-hazards model, to test for potential interaction effects emerging during our qualitative analysis. Logistic regression coefficients are odds ratios. Cox model coefficients are hazard ratios; a hazard ratio above 1 indicates a covariate that is positively associated with the event probability, and thus negatively associated with the length of survival.

### 4.1 Survival Models and Interview Insights

**Project-level Effects.** The **number of commits** is associated with higher chances to become dormant in the next time period in both groups (early- and late-stage), *i.e.*, packages with higher commit activity are more likely to become dormant, other variables held constant. Interviewees pointed out that commit squashing (merging large contributions into a single commit), which would reduce the number of commits, may help explain this effect: mature projects may use this practice more often, but are less likely to become dormant; at the same time, a contributor with direct commit access can contribute many small commits, which often happens in smaller projects, which are more likely to become dormant. Three interviewees indicated using recency of commits as the main indicator of sustainability. Future work should consider counting commit message lines instead (squashed commits tend to contain all original messages), and replacing the absolute number with commit dynamics, such as stability of monthly contributions. Two participants also suggested adjusting the number of commits to the project size, since bigger projects may need more maintenance.

The **number of contributors** has a negative effect on chances of becoming dormant in both groups, *i.e.*, packages with more contributors are less likely to become dormant. Six interviewees agreed unconditionally that having more contributors improves sustainability. Explanations included, in decreasing order of popularity: larger recruitment pool for the core team, better code reviews, and an indication of healthy onboarding practices. An important addition to the number of committers as a sustainability metric, pointed out by three interviewees (all are maintainers of big projects) is that it does not capture non-code contributions. For example, people contributing code reviews, issue triaging, and even evangelism and securing funding are essential for project sustainability.

The **size of the core team** is collinear with the number of contributors in our models, hence not included. Our interviewees also perceive it similarly. It was unanimously viewed as a positive factor by maintainers of big projects. All explanations of the effect either directly referred to the "bus factor" or closely resembled its definition. For small projects, this metric was either equivalent to the total number of contributors and still considered positive, or did not apply because the project was considered feature-complete.

The **number of issue reporters** was not included in our models due to multicollinearity. For our interviewees, this was perceived as another way to measure user base.

The **number of issues** was expected to be a positive indicator of user engagement. However, it was estimated as a risk factor in the first six months (*i.e.*, the early-stage model), while still decreasing

the chances of becoming dormant in the later-stage survival model. In the interviews, the discussion revealed several layers of interpretation for this metric. First, reported issues were unanimously considered to be a positive sign of an active user community. Two small project maintainers also noted that no issues in a small project could be an indicator of a project without quality issues rather than low usage. Four participants suggested looking into issue handling and discussions. Average response time, number of responses, and number of closed issues were proposed as indicators of activeness for the project team.

Another interpretation for issues, coming from three maintainers of big projects, suggested that issue triaging, although helpful for end users, takes resources and sometimes slows down development activities. Some projects were known to stop responding to issues completely to conserve developers' effort.[12] Based on this discussion, in the third model we introduce an *interaction between the number of commits and number of contributors*. The interaction effect was significant, rendering number of issues as a positive factor in projects with high volume of commits. Controlling for this interaction, the number of reported issues itself is not significant.

**$H_1$. Upstreams. Upstream dependencies** were expected to increase the chances of projects becoming dormant. The modeling effects are nuanced. In the first six months, a higher number of upstreams correlates with higher chances of dormancy, as hypothesized, although the presence of a *dormant* upstream reduces this risk. Later on, a higher number of upstreams correlates with lower chances of dormancy, but the presence of a *dormant* upstream increases this risk.

The interviews with utility library project maintainers offer a potential explanation. Such projects are considered dormant per our definition, while in fact they are feature complete. Reusing feature complete libraries can boost development of a new project, hence the reduced dormancy risk in the first six months. In the long-term, however, projects may start to incur higher costs of maintaining compatibility with dormant upstreams, hence the increased risk.

At the same time, having more upstreams overall in the long term enables more reuse, compensating for increased dormancy risks. Still, interviewees were cautious, stressing that it is better to limit upstream dependencies in the long-term to those that are really necessary, as there is a trade-off between development effort (lower with more reuse) and potential compatibility issues later on. The mention of "as few as possible within reason" suggested a non-linear effect, which we added to the third model as a *quadratic term of the number of upstreams*; however, this term did not have a statistically significant effect. We further illustrate this trade-off.

On the one hand, the positive effect of dependency adoption comes from saving resources on implementation. A striking example comes from an interview with a project maintainer who was able to reuse a domain-specific library from a similar project. The maintainer claimed that most projects in this domain die in an attempt to implement this very expensive piece of functionality, so adoption of this dependency was essential for the project success.

On the other hand, the compatibility issues come from the Python package installer (pip) not having a version resolver. For example, if package A requires B version 1.0 and C version 1.0, and B 1.0 requires C 2.0, after installing package A a user might end up with an incompatible setup B 1.0 and C 1.0. The Python community developed tools to build isolated, non-contradicting sets of dependent packages (*e.g.*, virtualenv and pipenv), but even compatibility within these environments requires effort from package developers.

All interviewees seem well aware of this trade-off and reported using different heuristics to find the right balance. Several maintainers, especially of larger packages, indicated that they have to spend substantial effort to stay compatible with a wide range of environments by supporting outdated versions of upstream packages. In case of smaller upstreams, it is often considered a lesser evil to either reimplement a dependency or copy a compatible version under the package source tree.

In summary, an indication of compatibility with potential upstreams is the biggest factor in dependency adoption, but overall the evidence for or against $H_1$ is inconclusive.

**$H_2$. Downstreams. Downstream dependencies** were expected to have a positive effect on project sustainability. Our models indicate that indeed they have a positive effect in the long term, but not in the first six months. Building a community of downstream projects takes time. The most likely scenario for a project to have downstream dependencies early in their lifecycle is to be chipped off from a bigger project into a small utility library, used by other projects of the same maintainer. Such projects are usually limited in scope, do not require further maintenance, and thus will be considered dormant. In the later stage, however, this metric is dominated by "natural" downstream dependencies, working as a positive survival factor in the later-stage survival models.

Across all interviews, downstream dependencies were characterized as a mostly positive factor, with the main trade-off between extra maintenance effort and resources brought by the dependent projects. Main benefits brought by downstream projects were described as code contributions (three participants), free testers (two), secure funding (two participants from academic projects). Two participants considered the number of dependent projects a proxy for user base, and two maintainers of feature-complete projects stated that it is not important at all.

The negative side of downstream dependencies was described as extra effort to maintain compatibility and triage issues. Only one participant stated that contributions from downstream projects are not worth this effort. These explanations, together with modeling results, mostly support $H_2$.

**$H_3$. Structural properties. Relative position in the dependency network** was only relevant for four projects in the interview pool having transitive downstream dependencies, and even for those the dependency network was not directly observable. Due to these constraints the role of the relative position in the dependency network was mostly discussed from a theoretical perspective rather than personal experience.

In most cases, transitive downstreams were interpreted in the same way as direct ones, as a user base. Two participants stated that projects higher in the dependency chain need to put more resources into sustainability because of their special position. One of them motivated this with an example of "extremely painful" debugging of a second-level upstream dependency (*i.e.*, an upstream of an upstream). For two maintainers of feature-complete projects

[12] E.g., the npm CLI team: https://twitter.com/maybekatz/status/953402549293350913

the relative position in the dependency network did not matter. Limitations of the interview sample prevent us from building a more robust qualitative interpretation of this metric.

However, in all three models higher Katz centrality correlates with increased chances of projects becoming dormant. This effect could be possibly explained by a higher reuse rate of feature complete libraries. It could also be attributed to an increased maintenance effort required in projects higher in the dependency chain, as indicated by interviewees. Overall, based on the modeling and interview results, we could not confirm $\mathbf{H}_3$.

**$\mathbf{H}_4$. Backporting. Backporting** was used as an indicator of project practices aimed at reducing the maintenance cost of dependent projects [5]. It was estimated to substantially reduce chances of becoming dormant in the first six months, where practical importance of backporting releases is questionable. The model estimate, suggesting increased likelihood of survival, could be explained by projects occasionally mislabeling releases; however, the fact that these projects produce multiple releases in the first six months is serving as an indicator of sustainability. In subsequent later-stage survival models this feature did not have a significant effect. We could not confirm $\mathbf{H}_4$.

**$\mathbf{H}_5$. Organizational Support. University involvement** has a special role in the Python community. Many signature Python projects are related to traditionally academic domains: Data Science, Machine Learning, Artificial Intelligence, Natural Language Processing, etc. Our expectation was that university projects have extra risks, such as students leaving after graduation, end of funding cycles, etc. However, modeling results indicated that university involvement is not a significant dormancy risk factor in the first six months, but in later-stage survival models it reduces the chance of becoming dormant by approximately 25%.

In our interview sample, four participants were university affiliates. Overall, all four indicated that their OSS work is currently funded through a university through a grant or contract, and their contributions are related to their position in academia. Two of them started their projects as students, one joined an existing project, and the last project was created as a practical tool to support existing research. Two interviewees transitioned into faculty positions during their projects, and thus had an extended perspective on the evolution of an academic project. They commented on three survival challenges in the lifecycle of a university project: surviving the student graduation cycle, surviving the academic funding cycle, and growing outside academia. Two out of four university affiliates we interviewed assumed that the diversity of institutions involved might also be used as an indicator of sustainability, where projects with multiple institutions involved are expected to be truly owned by the research community, in contrast to local research projects. It was also suggested that the university involvement effect might vary depending on the area of science.

The explanation above and the modeling results partially support $\mathbf{H}_5$. However, increased sustainability of university projects might be specific to the Python ecosystem due to its high share of "academic" projects and should be further tested in different ecosystems by future work.

**Commercial involvement** was expected to have a positive effect on sustainability (lower risk of dormancy), but the models suggest otherwise; the feature also elicited somewhat controversial interview explanations. A shared opinion about commercial involvement among interviewees was that companies bring resources to the project, but this support is not sustainable long term. Common concerns include misalignment of companies' priories with the project goals and sustainability issues caused by companies withdrawing from a project.

The overall extent of commercial involvement in PyPI seems small. Contrasting the results of the 2016 Future of Open Source Survey, which states that "1 in 3 companies have a full-time resource dedicated to open-source projects" and "67% of companies actively encourage developers to engage in and contribute to open-source projects",[13] only two participants knew about cases of companies paying developers to contribute to OSS projects of their choice. One participant also indicated that their project benefits from commercial contributions, but "... those engineers will have a finite time with us. So we can't put them on the critical path". This explanation, coupled with modeling results, is at odds with $\mathbf{H}_5$.

Another aspect to commercial contributions is licensing. One participant stated that commercial companies are sensitive to licensing terms. In particular, many product companies will not be able to work with GPL products, though service companies might.

Adding **license restrictiveness** as a control variable in our models, we find that presence of a license, whether strong-, weak- or non-copy-left, is a positive survival indicator. Although one interview participant indicated that strong copy-left licenses, such as GPL, restrict project adoption, the model indicates that strong- and weak copy-left licenses have higher positive impact on project chances of survival than non-copy-left licenses or no license.

**Hosting under an organizational account on GitHub** has a substantial positive effect. Otherwise equal, such projects are 22% less likely to be become dormant, which partially supports $\mathbf{H}_5$.

## 4.2 Other Indicators of Sustainability

During the interviews other indicators of sustainability emerged. For example, competition was listed as a major driving force behind one project. Users in this domain can easily switch between projects, so this project had to implement new features added by competing projects. Such competition increases the required maintenance effort to stay up to date with the user needs.

Across many interviews, participants indicated that maturity of project practices plays an important role in an evaluation of the project's sustainability from the end user perspective. Prior work by Trockman *et al.* [63] also found that developers rely on observable signals when making decisions about which project to use or contribute to. The factors mentioned during our interviews were related to software quality, backward compatibility, developer onboarding, and support for end users. The indicators of such practices include implemented autotests, CI and test runner configurations, documentation, number of pull requests, GitHub stars, project website, etc. Three participants mentioned the number of downloads as an indicator of an active user community, although noisy ("even one order of magnitude doesn't tell you very much").

[13] https://www.blackducksoftware.com/2016-future-of-open-source, slide 26

### 4.3 Implications

Our study provides a quantitative way (supported by qualitative insights) to identify and predict which OSS projects may become dormant and therefore pose a risk to developers choosing dependencies. Our survival models show that in addition to known project-level factors impacting sustainability, such as the number of contributors and the number of users, a project's chances of becoming dormant (having limited activity) is influenced by a series of ecosystem-level factors, such as its position in the ecosystem dependency network. These results have several implications.

First, these results may be *actionable for OSS researchers and platform designers*. The ecosystem-level variables we found to correlate with a project's risk of becoming dormant, despite being aggregations of publicly accessible data, are not readily observable on platforms like GitHub. One of the defining features of GitHub is transparency [19]; developers rely on *signals* [63] displayed on GitHub repository and user pages (*e.g.*, counts of stars and followers, repository badges) to form impressions about each other and their work [48]. Our approach shows how new signals to display these otherwise unobservable ecosystem-level qualities, such as a project's position in the ecosystem interdependency network or its level of organizational support, could be developed.

In turn, displaying these signals may help developers identify sustainable projects and projects at risk, steer developers and organizations towards contributing to central projects most in need of support, and overall help nudge the ecosystem towards more efficient allocation of contributor effort. Recall the Open SSL and leftpad examples discussed in the introduction. In these and other similar cases, arguably the centrality of these projects for the health of the rest of the ecosystem was not as clear before their respective prominent incidents as it has become after the fact. Newly developed signals, such as the ones our approach can inform, could have been used to reduce the information asymmetry. Therefore, it is not surprising that recently both GitHub and PyPI have started displaying information on dependents and dependencies for some OSS packages hosted or published there. We expect other signals will become available in the future.

Our results may also be *actionable for OSS practitioners*. If future research confirms that there is a causal relationship, not just the correlation we demonstrated in our work, between the variables we identified and a project's risk of becoming inactive, that may provide means for suggesting how to extend the life of projects that become inactive without being feature complete.

Still, we emphasize that the response variable in this study (dormancy, or low development activity) is not always indicative of project abandonment, and it therefore requires careful interpretation and should be adjusted to project context. While all abandoned projects are dormant, not all dormant projects are abandoned. For example, utility libraries with a well defined scope do not require further maintenance and thus will also be rendered as dormant, even though they are not abandoned. One interviewee used SMTPlib to illustrate the issue. This library implements a standard unchanged for 30 years and does not require maintenance. This claim is supported by prior research on reasons for OSS project failure, indicating that 11% of seemingly abandoned projects are just considered feature complete by their authors [10]. This suggests that when interpreting sustainability indicators, one should adjust at least to project class and size. *E.g.*, existing dependent projects early in the lifecycle might indicate a chipoff from another project, feature complete from birth, which is not a negative sustainability indicator. Likely, a dormant upstream project might not be an issue if it is feature complete, but can be a problem if it requires constant maintenance. In practice, it means that the presented survival model does not fully apply to feature complete projects and one should consider qualitative methods instead.

Future research should try to further distinguish feature complete projects from abandoned ones. Suggested ways to determine if a dormant project is complete rather than abandoned may include looking at: maintainers' activity outside the project, non-development activity (mailing lists, issue trackers, and community forum discussions), and dynamics of the project user base; some anecdotal evidence also suggests that projects abandoned by their maintainers continue to be used by existing adopters, but are rarely adopted by new projects, in contrast to feature complete projects, which continue to be adopted as dependencies in new projects.

## 5 CONCLUSIONS

Prior work revealed a number of project characteristics related to the sustainability of open-source projects. In this mixed-methods study, we have extended those results to include ecosystem factors. We theorized about expected effects and used survival analysis on a large set of PyPI projects hosted on GitHub, modeling risk of dormancy early in their life cycle and later on. We then triangulated the models through interviews with project maintainers, and modeled interaction effects informed by the qualitative analysis.

Our work shows the real impact ecosystem context has in how software is developed, and suggests that it brings new risks as well as clear benefits. As open-source projects are increasingly incorporated into software supply chains, organizations need to improve their ability to evaluate the risks they are taking on and learn strategies for mitigating them. It is also becoming clear, for example in our results about the effects of corporate participation, that it can have a destabilizing effect as well as simply providing more resources. In addition to becoming more informed consumers of open-source software, commercial firms should carefully consider the impact that inconsistent participation can have on the ecosystem.

Due to the size of our dataset, not all new features suggested by the interviewees were practical to test. Remaining, untested effects, left open for future work, include: issue response time, issue discussions, quality of commit messages, quality of documentation, automated tests, and use of CI. Future research should generalize and contrast our results on different ecosystems, better account for feature complete projects, and test the remaining features suggested by interviewees. For now, we note all these issues as potential threats to validity.

## ACKNOWLEDGEMENTS

The authors kindly acknowledge support from NSF awards 1633083 and 1546393, an award from the Alfred P. Sloan Foundation, and support from the Google Open Source Program Office.

## REFERENCES


[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. 2017. Why do developers use trivial packages? An empirical case study on npm. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 385–395.

[2] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. 2016. A novel approach for estimating truck factors. In *Proc. International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

[3] Kelly Blincoe, Francis Harrison, and Daniela Damian. 2015. Ecosystems in GitHub and a method for ecosystem identification using reference coupling. In *Proc. International Conference on Mining Software Repositories (MSR)*. IEEE, 202–207.

[4] Bradley C Boehmke and Benjamin T Hazen. 2017. The future of supply chain information systems: The open source ecosystem. *Global Journal of Flexible Systems Management* 18, 2 (2017), 163–168.

[5] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. International Symposium on Foundations of Software Engineering (FSE)*. ACM, 109–120.

[6] Jan Bosch. 2009. From software product lines to software ecosystems. In *Proc. International Software Product Line Conference (SPLC)*. 111–119.

[7] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. 2003. Characteristics of open source projects. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*. IEEE, 317–327.

[8] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. 2015. Developer onboarding in GitHub: the role of prior social links and language experience. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 817–828.

[9] Malgorzata Ciesielska and Ann Westenholz. 2016. Dilemmas within commercial involvement in open source software. *Journal of Organizational Change Management* 29, 3 (2016), 344–360.

[10] Jailton Coelho and Marco Tulio Valente. 2017. Why modern open source projects fail. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 186–196.

[11] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. 2013. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge.

[12] Stefano Comino, Fabio M Manenti, and Maria Laura Parisi. 2007. From planning to mature: On the success of open source projects. *Research Policy* 36, 10 (2007), 1575–1586.

[13] Eleni Constantinou and Tom Mens. 2017. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering* 13, 2-3 (2017), 101–115.

[14] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. 2015. Assessing the bus factor of Git repositories. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 499–503.

[15] David Roxbee Cox and David Oakes. 1984. *Analysis of survival data*. Vol. 21. CRC Press.

[16] John W Creswell and David J Creswell. 2017. *Research design: Qualitative, quantitative, and mixed methods approaches* (third ed.). Sage publications. 203–224 pages.

[17] Kevin Crowston, James Howison, and Hala Annabi. 2006. Information systems success in free and open source software development: Theory and measures. *Software Process: Improvement and Practice* 11, 2 (2006), 123–148.

[18] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. 2012. Free/Libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)* 44, 2 (2012), 7.

[19] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proc. ACM 2012 Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 1277–1286.

[20] Carlo Daffara. 2012. Estimating the economic contribution of open source software to the European economy. In *The First Openforum Academy Conference Proceedings*.

[21] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2018. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* (2018).

[22] Bert J Dempsey, Debra Weiss, Paul Jones, and Jane Greenberg. 2002. Who is an open source software developer? *Commun. ACM* 45, 2 (2002), 67–72.

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

[24] Zakir Durumeric, James Kasten, David Adrian, J Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, et al. 2014. The matter of heartbleed. In *Proc. 2014 Conference on Internet Measurement Conference*. ACM, 475–488.

[25] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering*. Springer, 285–311.

[26] Nadia Eghbal. 2016. *Roads and Bridges: The Unseen Labor Behind Our Digital Infrastructure*. Ford Foundation.

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

[28] Oscar Franco-Bedoya, David Ameller, Dolors Costal, and Xavier Franch. 2017. Open source software ecosystems: A Systematic mapping. *Information and Software Technology* 91 (2017), 160–185.

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

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

[31] Mathieu Goeminne and Tom Mens. 2011. Evidence for the Pareto principle in Open Source Software Activity. In *Proc. International Workshop on Model-Driven Software Migration (MDSM)*. 74.

[32] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An exploratory study of the pull-based software development model. In *Proc. International Conference on Software Engineering (ICSE)*. ACM, 345–355.

[33] Patricia M Grambsch and Terry M Therneau. 1994. Proportional hazards tests and diagnostics based on weighted residuals. *Biometrika* 81, 3 (1994), 515–526.

[34] Shane Greenstein and Frank Nagle. 2014. Digital dark matter and the economic contribution of Apache. *Research Policy* 43, 4 (2014), 623–631.

[35] Guido Hertel, Sven Niedner, and Stefanie Herrmann. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research policy* 32, 7 (2003), 1159–1177.

[36] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. 2009. A sense of community: A research agenda for software ecosystems. In *Proc. International Conference on Software Engineering (ICSE) - Companion*. IEEE, 187–190.

[37] Stephen P Jenkins. 2005. Survival analysis. *Unpublished manuscript, Institute for Social and Economic Research, University of Essex, Colchester, UK* (2005).

[38] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. 2011. The onion patch: migration in open source ecosystems. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 70–80.

[39] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18, 1 (1953), 39–43.

[40] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. 2013. Is it all lost? A study of inactive open source projects. In *Proc. IFIP International Conference on Open Source Systems*. Springer, 61–79.

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

[42] Stefan Koch and Georg Schneider. 2002. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal* 12, 1 (2002), 27–42.

[43] Karim R. Lakhani and Robert G. Wolf. 2005. *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects*.

[44] Bin Lin, Gregorio Robles, and Alexander Serebrenik. 2017. Developer turnover in global, industrial Open Source projects: Insights from applying survival analysis. In *Proc. International Conference on Global Software Engineering (ICGSE)*. IEEE, 66–75.

[45] Mircea Lungu, Romain Robbes, and Michele Lanza. 2010. Recovering inter-project dependencies in software ecosystems. In *Proc. International Conference on Automated Software Engineering (ASE)*. ACM, 309–312.

[46] Mircea F Lungu. 2009. *Reverse engineering software ecosystems*. Ph.D. Dissertation. Università della Svizzera italiana.

[47] Konstantinos Manikas and Klaus Marius Hansen. 2013. Software ecosystems–A systematic literature review. *Journal of Systems and Software* 86, 5 (2013), 1294–1306.

[48] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. 2013. Impression formation in online peer production: activity traces and personal profiles in GitHub. In *Proc. 2013 Conference on Computer Supported Cooperative Work (CSCW)*. ACM, 117–128.

[49] Rupert G Miller Jr. 2011. *Survival analysis*. Vol. 66. John Wiley & Sons.

[50] Audris Mockus, Roy T Fielding, and James D Herbsleb. 2002. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11, 3 (2002), 309–346.

[51] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. 2002. Evolution patterns of open-source software systems and communities. In *Proc. International Workshop on Principles of Software Evolution (IWPSE)*. ACM, 76–85.

[52] Mathieu Nassif and Martin P Robillard. 2017. Revisiting Turnover-Induced Knowledge Loss in Software Projects. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 261–272.

[53] Felipe Ortega and Daniel Izquierdo-Cortazar. 2009. Survival analysis in open development projects. In *Proc. ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*. IEEE, 7–12.

[54] Jagdish K Patel, CH Kapadia, Donald Bruce Owen, and JK Patel. 1976. *Handbook of statistical distributions.* Technical Report. M. Dekker New York.
[55] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. 2013. Creating a shared understanding of testing culture on a social coding site. In *Proc. International Conference on Software Engineering (ICSE).* IEEE, 112–121.
[56] Eric Raymond. 1999. The cathedral and the bazaar. *Knowledge, Technology & Policy* 12, 3 (1999), 23–49.
[57] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. 2016. Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. In *Proc. International Conference on Software Engineering (ICSE).* ACM, 1006–1016.
[58] Gregorio Robles and Jesus M Gonzalez-Barahona. 2006. Contributor turnover in libre software projects. In *Proc. IFIP International Conference on Open Source Systems.* Springer, 273–286.
[59] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. 2010. Survival analysis on the duration of open source projects. *Information and Software Technology* 52, 9 (2010), 902–922.
[60] Charles Schweik, Bob English, Qimti Paienjton, and Sandy Haire. 2010. Success and abandonment in open source commons: Selected findings from an empirical study of sourceforge.net projects. In *Proc. Workshop on Building Sustainable Open Source Communities (OSCOMM).*
[61] Param Vir Singh, Yong Tan, and Vijay Mookerjee. 2011. Network effects: The influence of structural capital on open source project success. *MIS Quarterly* (2011), 813–829.
[62] Terry Therneau, Cindy Crowson, and Elizabeth Atkinson. 2017. Using time dependent covariates and time dependent coefficients in the Cox model. *Survival Vignettes* (2017).
[63] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proc. International Conference on Software Engineering (ICSE).* ACM, 511–522.
[64] Jason Tsay, Laura Dabbish, and James Herbsleb. 2014. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proc. International Symposium on Foundations of Software Engineering (FSE).* ACM, 144–154.
[65] Qiang Tu et al. 2000. Evolution in open source software: A case study. In *Proc. International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 131–142.
[66] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. 2016. The sky is not the limit: multitasking across GitHub projects. In *Proc. International Conference on Software Engineering (ICSE).* IEEE, 994–1005.
[67] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. 2015. Perceptions of diversity on GitHub: A user survey. In *Proc. International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE).* IEEE, 50–56.
[68] Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. 2014. On the variation and specialisation of workload—a case study of the Gnome ecosystem community. *Empirical Software Engineering* 19, 4 (2014), 955–1008.
[69] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. 2013. The Babel of software development: Linguistic diversity in Open Source. In *Proc. International Conference on Social Informatics (SocInfo).* Springer, 391–404.
[70] Georg Von Krogh, Sebastian Spaeth, and Karim R Lakhani. 2003. Community, joining, and specialization in open source software innovation: a case study. *Research Policy* 32, 7 (2003), 1217–1241.
[71] Jing Wang. 2012. Survival factors for Free Open Source Software projects: A multi-stage perspective. *European Management Journal* 30, 4 (2012), 352–371.
[72] Michael Wedel, Uwe Jensen, and Peter Göhner. 2008. Mining software code repositories and bug databases using survival analysis models. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM).* ACM, 282–284.
[73] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. 2015. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *Proc. International Workshop on Principles of Software Evolution (IWPSE).* ACM, 46–55.
[74] Yiqing Yu, Alexander Benlian, and Thomas Hess. 2012. An empirical study of volunteer members' perceived turnover in open source software projects. In *Proc. Hawaii International Conference on System Science (HICSS).* IEEE, 3396–3405.
[75] Minghui Zhou and Audris Mockus. 2012. What make long term contributors: Willingness and opportunity in OSS community. In *Proc. International Conference on Software Engineering (ICSE).* IEEE, 518–528.
[76] Minghui Zhou, Audris Mockus, Xiujuan Ma, Lu Zhang, and Hong Mei. 2016. Inflow and retention in OSS communities with commercial involvement: A case study of three hybrid projects. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 25, 2 (2016), 13.