Novelty Begets Long-Term Popularity, But Curbs Participation

A Macroscopic View of the Python Open-Source Ecosystem

Hongbo Fang
hongbofa@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

James Herbsleb
jdh@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA, USA

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

ABSTRACT

Who creates the most innovative open-source software projects? And what fate do these projects tend to have? Building on a long history of research to understand innovation in business and other domains, as well as recent advances towards modeling innovation in scientific research from the science of science field, in this paper we adopt the analogy of innovation as emerging from the novel recombination of existing bits of knowledge. As such, we consider as innovative the software projects that recombine existing software libraries in novel ways, i.e., those built on top of *atypical combinations* of packages as extracted from import statements. We then report on a large-scale quantitative study of innovation in the Python open-source software ecosystem. Our results show that higher levels of innovativeness are statistically associated with higher GitHub star counts, i.e., novelty begets popularity. At the same time, we find that controlling for project size, the more innovative projects tend to involve smaller teams of contributors, as well as be at higher risk of becoming abandoned in the long term. We conclude that innovation and open source sustainability are closely related and, to some extent, antagonistic.

CCS CONCEPTS

• **Software and its engineering** → **Open source model**.

KEYWORDS

Open-source software, innovation

ACM Reference Format:
Hongbo Fang, James Herbsleb, and Bogdan Vasilescu. 2024. Novelty Begets Long-Term Popularity, But Curbs Participation: A Macroscopic View of the Python Open-Source Ecosystem. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE 2024), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3597503.3608142

1 INTRODUCTION

It has long been recognized that open-source software development is an avenue for innovation and creative expression — "how creative a person feels when working on the project is the strongest and most pervasive driver" of participation in open source [29]. Unsurprisingly, we have seen an explosion in production of open-source software, especially in the last decade, with the proliferation of the social coding philosophy [10]. Nowadays, open-source software seems more popular than ever before [12], and it is hard to find any sectors of the economy that do not rely heavily on open-source infrastructure [15].

At the same time, with growing use and ubiquity of open source, there are growing concerns about the maintainability and sustainability of this digital infrastructure [21, 34, 38, 45]. It is not always without barriers for newcomers to join projects [33, 42], turnover rates for open-source contributors are high [27, 35], and even widely-used projects can end up being maintained by a single person or, sometimes, by no one at all [2, 9]. The insufficient maintenance of open-source projects can have disastrous consequences, as prominent security incidents like Heartbleed [47], the Equifax breach [7], and Log4Shell [23] have shown, just to name a few.

These days, significant attention is being paid by policy makers, practitioners, and researchers to understanding open source health and improving open source sustainability, with many open questions remaining around determinants of project success and failure, governance models, procurement and allocation of resources, and others. In this general context we focus on one important but poorly understood concept — innovation. While open source as a whole is a catalyst for innovation [14] (e.g., technology startup companies would have much slower start without access to open-source infrastructure) and understanding, and being able to identify, innovations in other fields has always been of great interest to investors, governments, etc, we know very little about how innovation emerges at the individual project level and what are its consequences, in either open-source or commercial software development.

Taking one step in this direction, in this paper we begin to study innovation in open source in the Schumpeterian tradition [40] of viewing innovation as emerging from the novel recombination of existing bits of knowledge, a typical perspective in the science of science field [18]. Operationalizing innovation at code level as a function of the libraries and packages a project imports (see Section 3.3) — projects built on top of more *atypical combinations* of libraries are considered to be more innovative — we find that in the Python open-source ecosystem projects with higher levels of innovation tend to be more popular on average, in terms of GitHub star counts. Stated differently, novelty begets popularity. At the same time, we find that controlling for project size, projects with higher levels of innovation tend to involve smaller teams and are more likely to become abandoned sooner, suggesting that the benefits of increased popularity also carry a cost of limiting the available labor pool [16] of potential maintainers of a code base that, on average, fewer people may be familiar with.

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3608142

Based on these results, we argue that innovation and open source sustainability are closely related and, to some extent, antagonistic. With creative expression being such a dominant driver of contributing to open source, as discussed above, one can expect that there will always be an incentive to *create* ever-new open-source software (innovation seems to be rewarded with increased popularity) over doing the work of *maintaining* existing systems through bug fixes and the like, which is often perceived as "grunt work" [25, 30]. This can contribute to exacerbating the resource allocation problem at the global, ecosystem level, by making it even harder to ensure that sufficient maintenance attention (developers, funding, etc) is being allocated to the projects that need it the most.

2 THEORETICAL FRAMEWORK

We start by reviewing prior work and developing our hypotheses. Innovation has long been an important but elusive construct in all domains, including the literature reporting on the software industry [13]. Various theoretical models of innovation and innovativeness of firms have been proposed [1, 43], touching on multiple dimensions such as process, product, and organization. And while innovation is often operationalized in terms of patents [32], as of 2020 "there is [still] little consensus on how innovation measurement should be carried out" [3].

Instead, we adopt the Schumpeterian [40] view of innovation as a novel recombination of existing bits of knowledge, and draw inspiration from studies of innovation in research publications and its relationship to scientific impact. In a widely cited paper, Uzzi et al. [44] analyzed the list of references in research articles indexed by the Web of Science database, measuring the extent to which papers cite atypical versus conventional combinations of prior work as a proxy for how innovative the papers are — more innovative papers are expected to combine existing bits of knowledge, i.e., cite prior papers or publication venues, in novel ways. For example, aggregating at the level of publication venues, a paper that is early to cross disciplinary boundaries would tend to rank as innovative because within a discipline researchers tend to publish in relatively small and disjoint sets of venues. Similarly, a later paper in an established interdisciplinary area would be perceived as less innovative, if there is a history of prior work citing combinations of venues from both disciplines. A similar argument can be made at the individual publication level, instead of aggregating at the level of publication venues. A key finding from this work by Uzzi et al. [44] is that papers citing some amount of atypical combinations of prior work at publication time "are unusually likely to have high impact," as measured by citation counts to the focal papers over the following years post publication.

By analogy, software is also rarely created from scratch but, rather, by recombining existing bits of functionality in novel ways. These days, there is a wealth of open-source code that developers can reuse, and mature package managers and package registries like PyPI (Python) and npm (JavaScript) to facilitate such reuse. Naturally, not every piece of code using some libraries will be hugely innovative. Conversely, at some level almost all open-source projects are at least somewhat innovative, except in some relatively rare instances of hard forking [49] or code copying [36]. However, similar to the previous argument about scientific innovations, we expect that combining existing software libraries in novel ways indicates, on average, a higher degree of creativity and innovation. Consequently, we hypothesize that:

$\mathbf{H}_1$. *Open-source projects reusing more atypical combinations of software libraries tend to be more popular.*

At the same time, higher levels of innovation may come at a cost and not every team may be equally positioned and equally able to pursue innovations. In the economics literature it is generally understood that large business organizations tend to be more risk adverse when it comes to untested ideas, which may have potential for greater returns if successful, but also carry higher risk of failure [8]. Similar effects are present in scientific research. Indeed, Wu et al. [48] have recently shown that it is the smaller teams that most innovate, "disrupt[ing] science and technology with new problems and opportunities."

One can expect a similar effect in open source through at least two complementary mechanisms. One mechanism is related to a similar willingness to experiment and take greater technological risks in smaller teams compared to larger ones, as expected in commercial firms and scientific research. Another mechanism in open source could be related to the availability of appropriately skilled maintainers and contributors — the more atypical a project is in its technology stack, the fewer people may have the relevant knowledge to participate in it [16]. Either way, regardless of the mechanism, we hypothesize that:

$\mathbf{H}_2$. *Open-source projects reusing more atypical combinations of software libraries tend to involve smaller groups of contributors.*

In the long term, both having a more atypical technology stack and having smaller teams may pose a sustainability risk for projects. Except in rare cases of "feature completeness" [45], open-source projects rely on a constant stream of contributors for survival. There is a rich literature on attracting and retaining contributors to open source, exploring a diversity of topics ranging from motivations to participate [22] and barriers to placing a first contribution [33, 42], to engagement [6, 39] and disengagement factors [27, 35]. Similarly, prior research has also tried to explain the factors associated with project abandonment and survival [2, 9, 45]. We add to this literature by exploring the link between project innovativeness and project long-term survival. On the one hand, there is prior evidence suggesting that higher project popularity is associated with higher attractiveness to new contributors [19]; in turn, this should increase a project's chances of long-term survival. On the other hand, while possibly more popular, we expect that more innovative projects will involve smaller teams per $\mathbf{H}_2$ above and will have a harder time recruiting contributors, because there may be fewer people with the right expertise in their potential contributor pools. In the long term, we hypothesize that:

$\mathbf{H}_3$. *Open-source projects reusing more atypical combinations of software libraries tend to be at greater risk of abandonment.*

In the remainder of the paper, we describe how we tested these hypotheses at scale, using a large longitudinal dataset of open-source projects part of the Python ecosystem.

3 METHODS

3.1 Data Collection

The main source of data in this study is the *World of Code* [31] dataset. This dataset records the development activities of millions of open-source projects hosted in public git repositories online, including all of GitHub, Gitlab, and BitBucket. World of Code also includes timestamped package dependency information for many programming languages, extracted by parsing import statements, which we use to compute the atypicality of the projects' combinations of imported packages. This information is usually not available in other open-source software datasets. The closest are package manager dependency datasets (e.g., libraries.io), which contain only within-package-manager dependencies; in World of Code we can observe also dependencies in projects that are not themselves libraries hosted by a package manager registry.

To keep the analysis tractable (our data collection involves aggregation of raw data across platforms and complex, computationally expensive measurement steps), we focus within World of Code only on Python projects hosted on GitHub as the subjects of study. Python is one of the top languages used for open-source projects [4] and supports a wide range of applications and projects of different purposes.[1] It is a large, diverse, and interesting ecosystem. GitHub is the largest platform for hosting open-source development. Therefore, the effects observed in our sample may generalize to other open-source projects and programming language ecosystems, though that remains to be tested.

We consider projects with over 50% of their source code files written in Python as Python projects in the World of Code dataset and we query the complete commit activities for all projects before the end of 2021. To address the risk of including in our sample personal projects that are not intended to be used or developed by others (e.g., class projects), we require projects to have at least one release on GitHub to be considered for further analysis.

We obtain the commit activities of projects directly from the World of Code dataset, which provides de-aliased developer information through the approach developed by Fry et al. [20]. Commits from bots are removed following the approach described by Dey et al. [11]. In addition, we compute timestamped GitHub star counts, and extract public GitHub release dates and data to compute other control variables through the GitHub API.[2]

Lastly, for projects that implement a Python package, we also use their number of downloads from the PyPI package manager as an outcome measure of project usage. We obtained the monthly download data from public PyPI download statistics,[3] collected using the Google BigQuery API. Download counts from common mirroring tools are excluded to reflect more accurate download behaviors by users.

Overall, there are 70,891 projects left in our sample after this procedure, and those projects will be further filtered for each specific analysis as we describe below.

[1]https://www.python.org/about/apps/
[2]https://docs.github.com/en/graphql
[3]http://www.lesfleursdunormal.fr/static/informatique/pymod_stat_en.html

3.2 Pre-processing of Package Dependencies

We extracted package dependency information from the World of Code dataset by parsing the content of source code files.[4] To ensure accuracy, we de-aliased packages that were imported with different names but provided very similar functions (e.g., *urllib* and *urllib3*). We accomplished this by collecting the package homepage URL from libraries.io,[5] and we merged two packages if they pointed to the same homepage.

To focus on commonly used packages, we removed all packages that were imported by no more than one hundred projects, as well as Python standard libraries. The latter tend to provide little information about project functions and appear in many projects (e.g., os). We obtained a list of Python standard libraries by crawling the standard library page.[6]

After pre-processing, our sample consisted of 1,055 packages.

3.3 Quantifying Project Atypicality

At a high level, we first compute an atypicality score for all pairwise package combinations, i.e., all distinct pairs of packages imported in the same project.[7] We then aggregate the package-pairwise atypicality scores at the project level by taking the average atypicality of all package combinations in a project. Moreover, since the atypicality of package combinations can change over time (e.g., two packages that were previously used separately may become commonly imported in the same project years later), we compute the atypicality of packages and projects for each year.

More concretely, to measure the atypicality of a package combination (i.e., pair), we use a variation of the Markov Chain Monte Carlo algorithm following Uzzi et al. [44]. We construct simulations of importing events where projects import the same number of packages in the same year, but the choice of packages to import is decided randomly. This allows us to create an artificial source of counterfactual evidence, such that we can compare how often two packages are actually used together (the reality) with how often one could have expected the two packages to be used together, given how much each package has been used individually (the counterfactual) – naturally, the more widely used two packages are, the more likely they could end up being used together, simply by chance. Using this strategy, we can therefore compare the empirically-observed frequency of any pair of two packages with the frequency resulting from the simulation runs, to estimate if the two packages in a given pair are used together more, less, or about as much as could be expected by random chance. Specifically, a low observed frequency of a pair of two packages relative to the hypothetical, simulated one indicates an atypical combination of those two packages. Note how this random shuffling of links in the package dependency network preserves the total numbers of importing projects and imported packages, the global frequency of use of each imported package, and the number of imported packages per project – that is, it implicitly controls for many confounding factors that could bias the estimates of atypicality of a package combination.

[4]https://github.com/woc-hack/tutorial
[5]https://libraries.io/
[6]https://docs.python.org/3/library
[7]If a project imports n packages, the binomial coefficient of n and 2 gives the total number of distinct pairs.

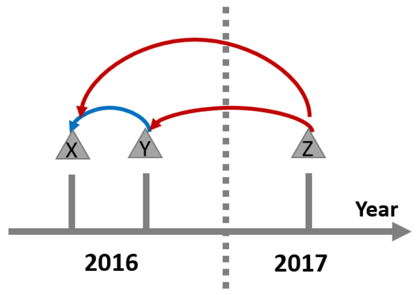

Figure 1: Package combinations within a project

Next, we describe how we computed and aggregated the atypicality scores in detail.

3.3.1 Obtaining Timestamped Project Dependency Information. We use the World of Code dataset to extract information about the packages that projects import. Specifically, we mine all Python source files in a repository and parse out the packages imported therein, recording also the earliest importing time for every package imported in a project (a package could be imported multiple times, in different files part of the same repository). The earliest time a package is imported across all projects indicates how "old" the package is. Therefore, each project and a package it imports can be represented as a tuple $[P, p, t_P, t_p]$, where P is the project, p is the package, and t_P and t_p represent the time when project P first imports package p and when package p was first imported by any project, respectively. To simplify the representation, t_P and t_p only record the year instead of the exact time.

3.3.2 Measuring the Frequency of Pairwise Package Combinations. Based on the timestamped dependency information, we compute the number of times that two packages are imported in the same project. In Figure 1, the focal project imports three packages X, Y and Z, with package X and Y being imported in 2016 and package Z in 2017. There are three possible pairwise combinations of packages (i.e., X and Y, Y and Z, and X and Z), and we consider the later time that one package was imported into a project as the time when new package combinations appear. For the focal project in Figure 1, there is one package combination in 2016 (X-Y, shown in blue), and two package combinations in 2017 (X-Z and Y-Z, shown in red).

For each given year, we aggregate over all projects in our sample and measure the frequency that two packages appear in the same project until that year, which gives us a dynamic measurement of package combinations and their evolution over time.

3.3.3 Simulating Package Importing Events. In the description below, we use *importing events* to refer to the importing activities represented by the tuple in Section 3.3.1. In this step, we generate random simulations of importing events which change the packages that a project imports, but preserve the other properties of the global project-package dependency network as much as possible, through a variation of the Markov Chain Monte Carlo algorithm. *Importing events* refer to the importing activities represented by the tuple in Section 3.3.1; the process is illustrated in Figure 2.

In Figure 2, there are three project nodes A, B, and C (project C appears two times), and three package nodes X, Y, Z. A directed edge

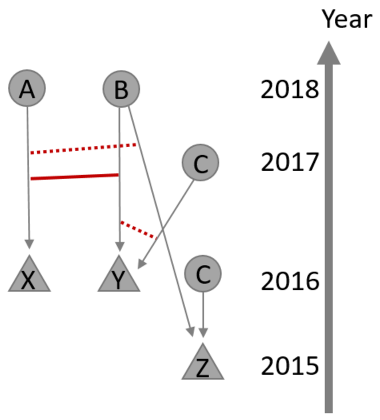

Figure 2: Switching package importing events

from one project to a package corresponds to an importing event, where the year that the project node is in represents the earliest time when the project imports the package (or t_P in the tuple), and the year that the package node is in corresponds to t_p and stands for the earliest time that the package was ever imported by any projects. For example, the edge from C (project) to Z (package) indicates that project C imported package Z (first ever imported in 2015) in 2016, and one year later project C imported package Y, which was first ever imported in 2016. For simplicity, we use P-p links to describe an importing event where project P imports package p (e.g., the A-X link in Figure 2)

A random switch of importing events is defined as randomly taking two importing events, and exchanging the package being imported between the two events. Given the A-X and B-Y links in Figure 2, a switch between those two links will lead to simulated importing events where project A imports package Y, and project B imports package X (or, the A-Y and B-X links). This random switch changes the packages that a project uses (and the package combination within a project), but keeps fixed the number of packages that a project imports, and the number of projects that a package was imported in.

In addition, because the atypicality of package combinations is computed longitudinally, we preserve the longitudinal pattern of imports by restricting the switch to be between t_P and t_p. Therefore, a switch between the A-X and B-Y links is allowed, because both importing events happened in 2018, and the imported packages were both first-imported in 2016. In contrast, a switch between the B-Y and C-Y links will not be allowed, because project B imported the package in 2018 while project C made the import in 2017. Similarly, switching between A-X and B-Z is not allowed because package X was first imported in 2016, which differs from 2015 for package Z. Finally, we further require that a given package be imported at most once per project

For each given t_P and t_p, and the set of events selected by the two timestamps, we conduct the described switch within this set of events many times to simulate random dependency relationships. There is no guarantee when the Monte Carlo algorithm will converge, but Uzzi et al. [44] suggest to run the switch $100 * E$ times, where E corresponds to the number of links (or importing events).

3.3.4 Comparing the Empirically-Observed Events with the Simulated Ones. We run the same simulation twenty times which results in twenty different randomly generated importing event sets. For each simulated set, we measure the frequency of package combinations until a given year as described in Section 3.3.2. Given twenty simulated event sets and the empirically observed pairwise package combination frequency, we compute a z-score for each package combination with equation 1,

$$z_{ijt} = (obs_{ijt} - exp_{ijt})/(\sigma_{ijt}) \tag{1}$$

where obs_{ij} corresponds to the empirically observed number of times that packages i and j appeared in the same project until year t, exp_{ijt} is the average number of times (or the expected times) that packages i and j appear in the same project until year t over twenty simulated event sets, and σ_{ijt} is the standard deviation of the co-appearance frequency of packages i and j in those sets as well.

Intuitively, the z-score represents how many standard deviations more (or less) the observed combination frequency between two packages is, compared to that expected by random chance. Since the numeric value of the z-score can be quite dispersed, we transform it with equation 2 to smoothen it.

$$Z_{ijt} = \begin{cases} \log(z_{ijt} + 1) & z_{ijt} \geq 0 \\ -\log(-z_{ijt} + 1) & z_{ijt} < 0 \end{cases} \tag{2}$$

A high Z-score corresponds to low atypicality and vice versa.

3.3.5 Aggregating the Package Combination Atypicality Scores at the Project Level. Given the package combination atypicality score (measured by the Z-score above) until a given year, we aggregate it at the project level by taking the average atypicality of all its package combinations, with package atypicality measured in the year when the combination appears in the focal project. Using Figure 1 again as an example, the atypicality score of the focal project is measured by averaging over the atypicality of the X-Y combination measured until 2016, and the X-Z and Y-Z combinations both measured until 2017.

3.4 Studying the Association between Project Atypicality and Interest from Developers and Users

We conducted regression analysis to understand the association between project atypicality scores and the outcomes of interest, as described in Section 2. Only projects with at least X years (where X is a hyper-parameter) of historical commit data before 2022-01-01 were selected for this analysis. Both the outcome and independent variables were computed based on activities in the first X-year after the projects' first commit. Table 1 provides a full list of variables used in the model.

To test the robustness of our results when modeling the effect of project atypicality on the number of project developers and project stars, we selected $X = 1, 2, 3, 4,$ and 5. We report the results for all values of X.

Table 1: Variables used in the regression analysis

Dependent variables	
Developer count	The total number of developers who made at least one commit to the project within its first X-year (after its first commit). Also used as controls when *Star count* being the outcome variable.
Star count	The total number of GitHub stars the project received within its first X-year. Also used as controls when *Developer count* being the outcome variable.
Download count	The total number of PyPI downloads the project received within its first X-year.
Control variables	
Package imported	The number of packages that the project imported within the first X-year.
Is owned by organization	A binary variable indicating whether the project was owned by an organizational account.
Time in GitHub	The number of days since project creation in GitHub, it differs from the project development lifespan as some projects only migrated to GitHub in later stage of project development.
Time in PyPI	The number of days since project registration in PyPI, and it is different from the project development lifespan.
Year of first commit	Fixed effect variable representing the year when the project received its first commit.
Atypicality measurement	
Conventionality	The aggregated Z-score at the project level as described in Section 3.3, with the value computed based on activities within its first X-year.

3.5 Survival Analysis on the Sustainability of the Project and its Relationship to Atypicality

In addition to measuring the amount of attention a project receives from the community, we also use a Cox proportional-hazards model to conduct survival analysis and understand how project conventionality influences its long-term activity. To do so, we sample projects that received at least one commit in year Y (where Y is a hyper-parameter) and analyze how long they are active before no longer receiving commit contributions. Following Qiu et al. [39], we consider projects with no commit activity for a period of 12 months to be abandoned, as the duration between a large majority of two consecutive commits is less than 12 months. As we have commit activity data until the end of 2021, we can detect project abandonment events that occurred before the end of 2020. Projects that are still receiving contributions by the end of this period are labeled as censored and considered to be actively maintained. We use the same control variables and atypicality measurement as described in Table 1, with the exception that they are computed based on activities until the end of year Y instead of within the first X years of the project's first commit. Additionally, we include the number of commits a project receives in year Y and the number of developers who made at least one commit to the project in year Y as controls

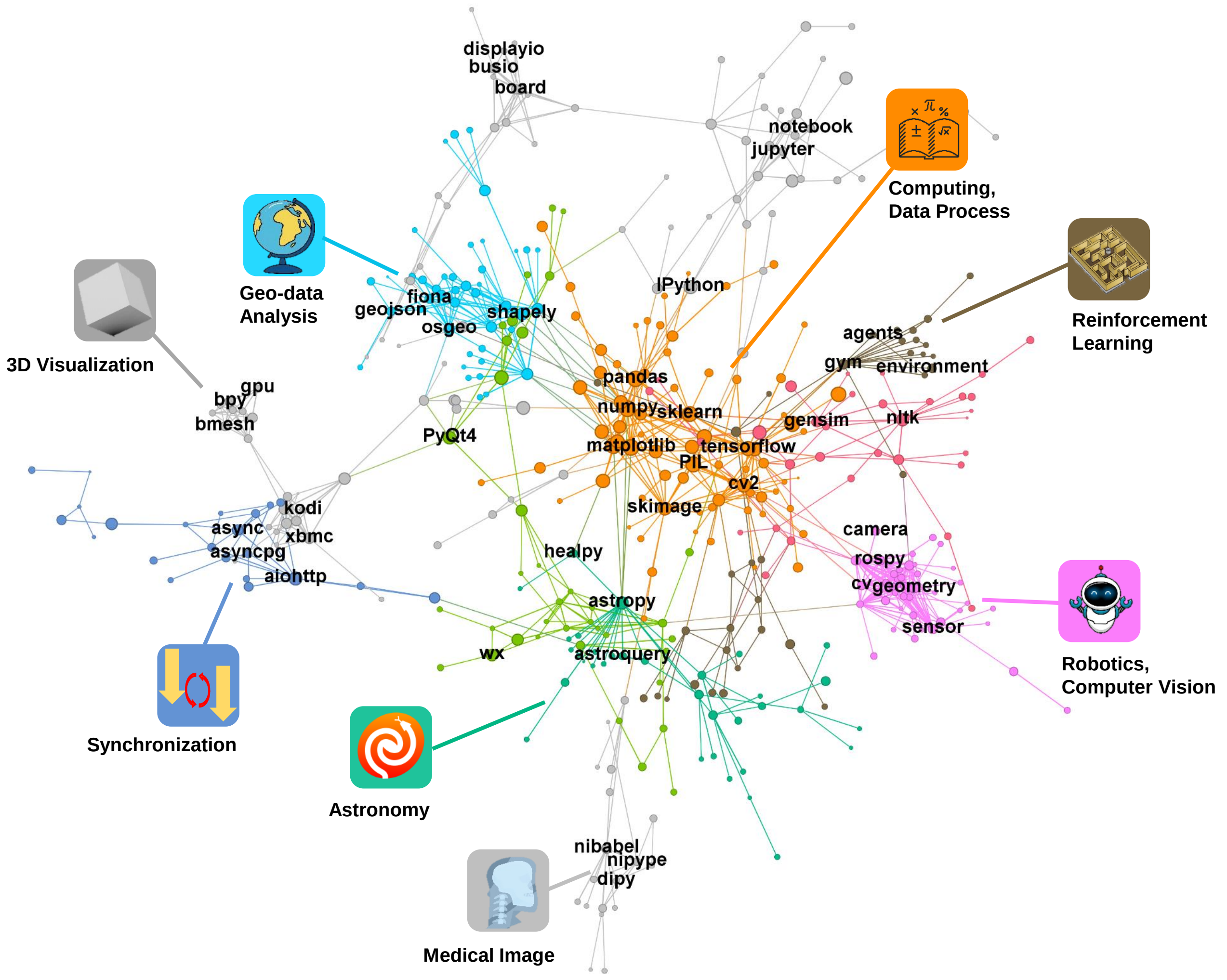

Figure 3: Typical combination of Python packages (until 2019)

to account for recent project activity. We select $Y = 2016, 2018$ and find that the results are qualitatively similar. We report the survival analysis effects with $Y = 2016$ in Section 4 with the other results available in the replication package.

The processed dataset and the scripts to replicate our results are available in the replication package.

4 RESULTS

4.1 Understanding the Atypicality Measurement

The atypicality measurement, or the Z-score, is the key variable of interest in the study. To validate the effectiveness of this measurement and provide more context, we describe what this measurement captures for both individual package combinations and clusters of package connections.

4.1.1 Understanding the Conventional (Atypical) Package Combinations. High Z-score package combinations indicate that two packages are frequently used together in the same project. To evaluate this, we manually examined instances of Python packages used for different purposes and identified their most commonly combined packages. These results are shown in Table 2.

Our measurement captures reasonable pairwise package combinations. For example, *numpy* is a general statistical computing package and is frequently used together with other computing packages such as *scipy*, machine learning packages including *tensorflow* and *sklearn*, data processing packages such as *pandas*, and visualization packages like *matplotlib*.

Another package, *OpenSSL*, provides a Python interface to the *OpenSSL* library, which is a popular network security package. It is commonly used in combination with other packages related to network connections and communications, such as *ntlm* (a network authentication package), *cryptography* (a package that provides

Table 2: Examples of typical package combinations

Focal package	Top five mostly combined pacakges
numpy	matplotlib, scipy, sklearn, pandas, tensorflow
tensorflow	keras, cv2, numpy, sklearn, scipy
django	rest, dj, south, whitenoise, celery
OpenSSL	ntlm, cryptography, ndg, pyasn1, idna
pymysql	MySQLdb, aiomysql, pstat, psycopg2, pymssql

cryptographic recipes and primitives), *ndg* (an HTTPS client implementation based on PyOpenSSL), *pyasn1* (packages related to telecommunications), and *idna* (support for domain names).

Interestingly, packages with similar functions, such as *pymysql* and *MySQLdb* may also be imported in the same project often. Manual inspections over a sample of projects importing both packages suggest that package replacement likely to contribute to the phenomenon. For example, project *brendanberg/f5* initially use *MySQLdb* in its development, but later switch to *pymysql*. It is also possible that projects import both packages to accommodate different users. Project *INWTlab/dbrequests* imports both *MySQLdb* and *pymysql*, and its users can choose one version of implementation as *MySQLdb* is more efficient but harder to install.

At a higher level, we visualize the combination of Python packages by constructing a topological graph shown in Figure 3. Each node in the graph represents a Python package, and an edge between two nodes indicates that the two packages have been imported in the same project, with the weight of the edge being its Z-score. The packages shown and their Z-scores are computed based on combination information until the end of 2019. We only keep the top 0.006% of edges with the highest weights and present the largest connected component to obtain a visually readable graph.

The size of each node represents the number of projects that imported the package by 2019. We used a fast unfolding community detection algorithm [5] to identify the discernible clusters ("communities") in the network. The communities are presented in different colors in the graph, and our community partition has a modularity level of 0.82, indicating a strong community structure.

Overall, the graph reveals a core-periphery structure, with a large orange cluster in the middle and other communities surrounding it. Our manual evaluation of packages suggests the existence of semantically meaningful clusters.

The orange community in the middle represents the core cluster and consists of many widely used packages, including statistical computing packages like *numpy*, data processing packages like *pandas*, visualization packages like *matplotlib*, and popular machine learning libraries like *tensorflow*. This observation indicates that these packages are used in diverse projects and are essential building blocks for many different applications.

The core community connects to several smaller but more specified package clusters. For instance, the pink community mostly comprises packages related to robotics, motion control, and computer vision, which are connected to the core community through image processing packages such as *cv2*. Similarly, the brown community includes packages important for reinforcement learning projects, such as *gym*, and connects to the core community through machine learning packages like *tensorflow*.

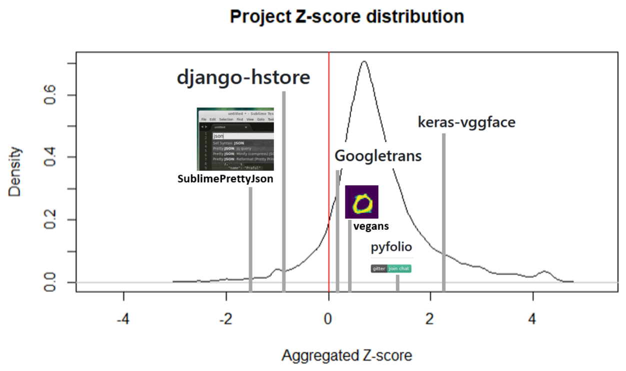

Figure 4: Distribution of Z-score and example of projects

There are several package clusters that are distant from the core community. For example, there is a package cluster for 3D visualization on the right side of the graph, and another cluster for asynchronous programming.

In summary, the graph based on the most conventional combination of packages represents meaningful clusters of communities, and the atypicality measurement captures how packages are commonly used together in projects.

4.2 Distribution of Project Z-scores

Figure 4 shows the distribution of Z-scores aggregated at the project level. As the figure illustrates, the distribution is skewed towards the positive end of the axis, indicating that most projects are not very atypical. This result is expected given the definition of atypicality. The project density peaks at around 0.7 Z-score, and very few projects have an extremely high Z-score. This suggests that the majority of projects are marginally conventional, meaning they are neither very atypical nor very conventional.

We also provide examples of atypical and conventional projects as captured by our measurement. *SublimePrettyJson* is a plugin supporting *sublime*, and it combines *simplejson* library with *sublime* which makes it atypical projects by our measurement. In the same vein, *django-hstore* provides PostgreSQL HStore support for Django, which makes it marginally atypical. Typical examples for conventional projects are machine learning repositories. For example, the *keras-vggface* repository implements VGG framework with keras, and it is done together with other statistical computing packages that are commonly used together.

4.3 Project Atypicality and the Outcome of Interest

Table 3 presents the results for the regression analysis when selecting X =5 (see Section 3.4). We visualize the estimated coefficient for the atypicality effect for each value of X in Figure 5. Table 4 summarizes our survival analysis results for Y = 2016 (see Section 3.5). The models behave similarly for other values of Y, as demonstrated in the replication package.

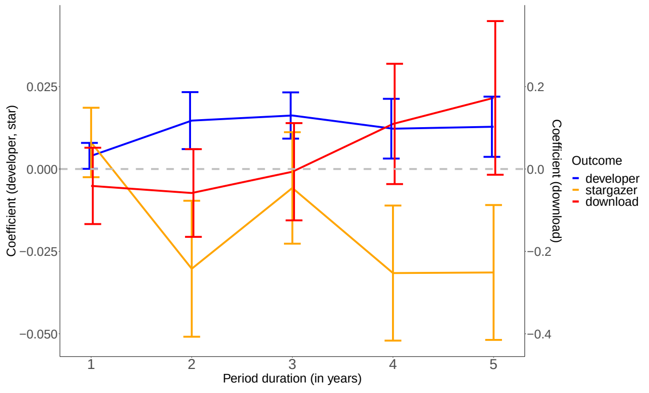

Figure 5: Estimated coefficient of Z-score with different outcome variables across different period duration (Error bar represents 95% confidence interval)

4.3.1 Project Atypicality and Popularity. Table 3 shows the results of Model I, which examines the relationship between project atypicality and the number of stars that the project receives. After controlling for the number of commits, the number of developers, project owner identity, and the number of imported packages, we find that the conventionality of a project's package combinations is negatively associated with the number of project stars on GitHub.

The conventionality variable is standardized to have a mean of zero and a standard deviation of one. The coefficient indicates that, after controlling for other variables, a one-standard-deviation increase in project conventionality (or a decrease in atypicality) corresponds to a 3% decrease in the number of project stars.

For example, the 25% most atypical projects in our sample have a conventionality that is 0.5 standard deviations below the mean, while the 25% most typical projects have a conventionality that is 0.4 standard deviations above the mean. Given that the mean number of stars for projects in our sample is 123 (median 7.0), shifting from the 25% most conventional projects to the 25% most atypical ones leads to an average increase of 3.3 stars, controlling for other variables.

The orange line in Figure 5 presents the estimated atypicality effects on project star counts measured for different time periods. The estimated coefficient is significantly below zero in most cases, suggesting that atypical projects tend to have more stars. The exception appears in the early stage when we only measure the project star counts within one year after project creation – in that case atypical projects tend to receive fewer stars during the period. One possible explanation is that atypical projects take longer time to be recognized and appreciated, which is known as the *sleeping beauty* effect in the science of science literature [46].

Next, we measure the relationship between project atypicality and the number of downloads that the project receives. In contrast to the effect on star counts and as shown in Model II, we do not observe a statistically significant effect at 95% confidence level, and the insignificant correlation holds for all the time period lengths measured in our study, as shown by the red line in Figure 5. Therefore, we report no conclusive association between project atypicality and the number of downloads.

Given our results, we partially confirm $\mathbf{H}_1$ by suggesting that the project atypicality associates with higher GitHub star counts in the long term, but possibly a lower number of stars in the short run. In addition, we find no significant association between project atypicality and project download counts.

4.4 Project Atypicality and the Size of the Developer Team

Model III, presented in Table 3, investigates the relationship between project atypicality and the size of the development team. After controlling for the number of commits, project owner, project popularity, and the number of imported packages, we find that the conventionality of a project's package combination is positively associated with the number of developers working on the project.

The estimated coefficient indicates that, after controlling for other variables, a one-standard-deviation increase in project conventionality (or a decrease in atypicality) corresponds to a 1% increase in the number of developers working on the project. A similar change from the quarter of most conventional projects to the 25% most atypical projects corresponds to a change in conventionality of -0.9 standard deviations, which would lead to a decrease of 0.14 developers per project, or approximately one fewer developer per every seven projects, after controlling for other variables.

Importantly, the number of developers is positively associated with the project conventionality only when controlling for *Commit count* and *Package imported*, which are two measures of project size. Without them as controls, this correlation is reverted, becoming statistically significantly negative. Indeed there is both a negative correlation between project size (measured by the number of packages imported and the number of commits) and project conventionality, and a negative correlation between project conventionality and the number of developers. Conventional projects tend to have fewer developers, but each developer contributes fewer commits compared to atypical projects, hence the positive correlation as shown in Table 3 after controlling for the project size.

This observation aligns with the result reported by Uzzi et al. [44] on scientific paper novelty, where the authors found that larger researcher teams tend to generate more novel, or atypical, papers. In open-source software, we found that novel projects tend to be developed by larger teams, but have fewer developers for the same amount of work, which may influence the sustainability of the project as it requires more effort from each developer on average, as we discuss in Section 4.5 below.

Therefore, $\mathbf{H}_2$ is partially confirmed by our findings that controlling for the size of the project, novel projects tend to have fewer developers, but the correlation is reverted without the project size as a control variable.

4.5 Project Atypicality and its Sustained Development

Table 4 presents the results of our survival analysis when setting $Y = 2016$. Our findings show that, when controlling for the number of project developers, the number of commits to the project, and the project-level popularity, conventional projects tend to have a lower

Table 3: Results for regression analysis (X=5)

	Dependent variable:		
	Model I (stars)	Model II (downloads)	Model III (developers)
Control variables			
Commit count (log)	0.09^{***} (0.01)	-0.06 (0.12)	0.29^{***} (0.005)
Package imported (log)	-0.07^{***} (0.02)	-0.60^{***} (0.18)	-0.03^{**} (0.01)
Developer count (log)	1.36^{***} (0.01)	1.24^{***} (0.11)	
Star count (log)			0.26^{***} (0.003)
Is owned by organization	-0.73^{***} (0.02)	-0.28 (0.20)	0.46^{***} (0.01)
Time in GitHub	1.58^{***} (0.05)		
Time in PyPI		-0.10 (0.08)	
Atypicality			
Conventionality (scaled)	-0.03^{**} (0.01)	0.17 (0.10)	0.01^{**} (0.005)
Observations	17,211	1,114	17,211
Adjusted R^2	0.55	0.49	0.69

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

Table 4: The effect of project atypicality on the sustained development (Y=2016)

	Coefficient	Exp(coefficient)
Commit count (log)	-0.09^{***} (0.01)	0.92
Package imported (log)	0.01 (0.03)	0.01 1.01
Developer count(log)	-0.31^{***} (0.03)	0.73
Star count (log)	-0.11^{***} (0.01)	0.90
Is owned by organization	-0.40^{***} (0.03)	0.67
Conventionality (scaled)	-0.05^{***} (0.01)	0.95
Observations	10,997	10,997

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

hazard rate or a lower probability of being abandoned. This is indicated by the significant negative coefficient of the conventionality variable.

Specifically, we found that a one standard deviation increase in the conventionality score changes the probability of project abandonment to 95% of its original value. In this sample, shifting from the 25% most atypical projects to the quarter of most conventional projects would result in a likelihood of project abandonment that is a 95.5% of its original value ($0.95^{0.9}$). Therefore, we confirm $\mathbf{H}_3$.

5 DISCUSSION

As one of the first studies on innovation in the open-source software context, our work begins to reveals the impact that atypical combinations of packages have on outcomes related to the success of the project. We summarize the main findings below and discuss their implications and future research directions.

5.1 Atypical Projects Draw More User Attention

Innovation is often praised and encouraged as it leads to success in business [37], technology [41], and scientific publications [44]. With atypical combinations of packages as one possible way to measure innovation in open-source software, we show that more atypical projects tend to draw more user attention and receive more GitHub stars. This suggests that the pursuit of atypical, innovative projects does come with rewards in terms of project popularity.

5.2 Atypical Projects Can Face Difficulty in Development

While developing atypical projects brings higher levels of user attention, it also comes with challenges in project development. We observe that controlling for the project size, atypical projects tend to be developed by a smaller set of developers, suggesting that the workload on individual developers is larger compared to conventional projects as the latter have more contributors to share the workload with.

While projects developed by small teams may enjoy advantages such as reduced communication overhead and easier coordination [28], they also face greater risk to their long-term sustainability as their development and maintenance are dependent on a smaller set of developers and the overall maintenance status of the project may be strongly influenced by the activity status change of one or two of those developers [2]. This is indeed what we observe empirically for projects in our analysis, as the conventionality of a project is positively associated with an decreased probability of project abandonment, or increase in the project sustainability.

5.3 The Tension Between Popularity and Participation

It is often a puzzle why many open-source projects, while popular and widely adopted by users, are only maintained by a very small set of developers. This puzzle is not only of scientific interest, but also embodies practical implications towards a more sustainable open-source development and the construction of more reliable open-source projects [14].

In our work, we provide a new perspective to explain this puzzling tension between project popularity and developer participation through the measurement of project atypicality and innovation. Empirical evidence suggests that more atypical projects tend to draw more attention from users, possibly due to a competitive advantage relative to other packages available to the open-source community. However, more atypical projects tend to be developed by a smaller set of contributors, which could lead to problems in their sustainability. Following this observation, new questions arise. How to reduce the seemingly negative consequences of atypical combinations (fewer contributors with relevant expertise) while maintaining the positive consequences (higher popularity)? And how to reduce the long-term abandonment risk of open-source projects through the identification of atypical projects and understanding the mechanisms behind the tension between project popularity and developer participation?

5.4 Future Work: The Context Surrounding Innovativeness

Our paper suggests a way of measuring innovation in open-source and studying its association with measures of open-source project success. This opens up the discussion around how innovation happens in an open-source context. What developers or teams are most likely to produce innovative projects and what are their characteristics? And how can we support innovative teams and address the challenges faced by emerging innovations? In the value-in-diversity literature outside of software engineering, the diversity-innovation relationship is well studied: surface-level differences between team members in socio-demographic attributes are linked to cognitive diversity, conceptualized as the different ways in which people represent and solve problems when working in teams; in turn, cognitive diversity in a team can increase creativity and foster innovation, with empirical evidence from corporate and public organizations [26], scientific research [24], and many others. Does this relationship also hold in open source? If so, how strongly and under what conditions?

5.5 Future Work: Atypicality and Measurements of Innovation

Measuring innovation has been a challenge in almost every field that it was touched in. Borrowing an established measure from other fields, particularly the study of scientific publications and new knowledge discovery [44], we propose the atypicality of package combinations as a measure of software project innovativeness.

Still, we need more evidence of the extent to which it captures the concept of innovation in a software context in general and in open-source ecosystems in particular. We consider our study as a starting point for future research into measuring innovation in software and we call for more effort into the discussion of measurement and the development of validation approaches and datasets.

6 CONCLUSIONS

In this project, we built a theoretical framework around innovation in open-source software and its relationship to project popularity and sustained development. A main contribution of our work is a new measure of innovation in software, based on the atypicality of imported package combinations. We explained the scientific intuition behind it and provided empirical evidence on what the measure captures. We further carried out a large-scale study of open-source projects part of the Python ecosystem. We found that innovative projects tend to draw more attention, primarily in the form of GitHub stars, and more contributors, especially in the long term. At the same time, we found that innovative projects require more effort from developers on average and face greater challenges with sustaining their activity. We hope that our theoretical framework, measurement approach, and empirical evidence on the association between innovation and project outcomes will help attract more research attention to empirically studying innovation in software engineering.

7 DATA AVAILABILITY

The data and scripts to reproduce our results are available in the replication package [17]. DOI 10.5281/zenodo.8364651

ACKNOWLEDGMENTS

The authors kindly acknowledge partial support from the NSF awards 1901311, 2107298, and 2120323, the Google Open Source Programs Office, Google Faculty Research Awards, and a Google Award for Inclusion Research. We are additionally grateful to Courtney Miller, Audris Mockus, and Mahmoud Jahanshahi, for their help with data collection and discussions around our study design.

REFERENCES

[1] Richard Adams, John Bessant, and Robert Phelps. 2006. Innovation management measurement: A review. *International journal of management reviews* 8, 1 (2006), 21–47.

[2] Guilherme Avelino, Eleni Constantinou, Marco Tulio Valente, and Alexander Serebrenik. 2019. On the abandonment and survival of open source projects: An empirical investigation. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12.

[3] Nauman bin Ali, Henry Edison, and Richard Torkar. 2020. The impact of a proposal for innovation measurement in the software industry. In *International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–6.

[4] Tegawendé F Bissyandé, Ferdian Thung, David Lo, Lingxiao Jiang, and Laurent Réveillere. 2013. Popularity, interoperability, and impact of programming languages in 100,000 open source projects. In *2013 IEEE 37th annual computer software and applications conference*. IEEE, 303–312.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment* 2008, 10 (2008), P10008.

[6] Fabio Calefato, Marco Aurelio Gerosa, Giuseppe Iaffaldano, Filippo Lanubile, and Igor Steinmacher. 2022. Will you come back to contribute? Investigating the inactivity of OSS core developers in GitHub. *Empirical Software Engineering* 27, 3 (2022), 76.

[7] Brandon Carlson, Kevin Leach, Darko Marinov, Meiyappan Nagappan, and Atul Prakash. 2019. Open source vulnerability notification. In *IFIP International Conference on Open Source Systems (OSS)*. Springer, 12–23.

[8] Clayton M Christensen and Clayton M Christensen. 2003. *The innovator's dilemma: The revolutionary book that will change the way you do business.* HarperBusiness Essentials New York, NY.

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

[10] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. 2012. Social coding in GitHub: transparency and collaboration in an open software repository. In

ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW). 1277–1286.

[11] Tapajit Dey, Sara Mousavi, Eduardo Ponce, Tanner Fry, Bogdan Vasilescu, Anna Filippova, and Audris Mockus. 2020. Detecting and characterizing bots that commit code. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 209–219.

[12] Thomas Dohmke. 2023. Blog post: 100 million developers and counting. https://github.blog/2023-01-25-100-million-developers-and-counting/. Retrieved: March 2023.

[13] Henry Edison, Nauman Bin Ali, and Richard Torkar. 2013. Towards innovation measurement in the software industry. *Journal of systems and software* 86, 5 (2013), 1390–1407.

[14] Nadia Eghbal. 2016. Roads and bridges. *The Unseen labor behind our digital infrastructure* (2016).

[15] Nadia Eghbal. 2020. *Working in public: The making and maintenance of open source software.* Stripe Press San Francisco.

[16] Hongbo Fang, James Herbsleb, and Bogdan Vasilescu. 2023. Matching Skills, Past Collaboration, and Limited Competition: Modeling When Open-Source Projects Attract Contributors. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. ESEC/FSE '23.

[17] Hongbo Fang, James Herbsleb, and Bogdan Vasilescu. 2023. Replication package. https://doi.org/10.5281/zenodo.8364651

[18] Santo Fortunato, Carl T Bergstrom, Katy Börner, James A Evans, Dirk Helbing, Staša Milojević, Alexander M Petersen, Filippo Radicchi, Roberta Sinatra, Brian Uzzi, et al. 2018. Science of science. *Science* 359, 6379 (2018), eaao0185.

[19] Felipe Fronchetti, Igor Wiese, Gustavo Pinto, and Igor Steinmacher. 2019. What attracts newcomers to onboard on OSS projects? tl;dr: Popularity. In *Open Source Systems: 15th IFIP WG 2.13 International Conference, OSS 2019, Montreal, QC, Canada, May 26–27, 2019, Proceedings 15*. Springer, 91–103.

[20] Tanner Fry, Tapajit Dey, Andrey Karnauch, and Audris Mockus. 2020. A dataset and an approach for identity resolution of 38 million author ids extracted from 2b git commits. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 518–522.

[21] R Stuart Geiger, Dorothy Howard, and Lilly Irani. 2021. The labor of maintaining and scaling free and open-source software projects. *Proceedings of the ACM on Human-Computer Interaction* 5, CSCW1 (2021), 1–28.

[22] Marco Gerosa, Igor Wiese, Bianca Trinkenreich, Georg Link, Gregorio Robles, Christoph Treude, Igor Steinmacher, and Anita Sarma. 2021. The shifting sands of motivation: Revisiting what drives contributors in open source. In *2021 IEEE/ACM 43th International Conference on Software Engineering (ICSE)*. 1046–1058.

[23] Joseph Hejderup. 2022. On the Use of Tests for Software Supply Chain Threats. In *ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 47–49.

[24] Bas Hofstra, Vivek V Kulkarni, Sebastian Munoz-Najar Galvez, Bryan He, Dan Jurafsky, and Daniel A McFarland. 2020. The diversity–innovation paradox in science. *Proceedings of the National Academy of Sciences* 117, 17 (2020), 9284–9291.

[25] James Howison, Keisuke Inoue, and Kevin Crowston. 2006. Social dynamics of free and open source team communications. In *IFIP Conference on Open Source Systems (OSS)*. Springer, 319–330.

[26] Charlie Karlsson, Jonna Rickardsson, and Joakim Wincent. 2021. Diversity, innovation and entrepreneurship: where are we and where should we go in future studies? *Small Business Economics* 56, 2 (2021), 759–772.

[27] Rajdeep Kaur and Kuljit Kaur Chahal. 2022. Exploring factors affecting developer abandonment of open source software projects. *Journal of Software: Evolution and Process* 34, 9 (2022), e2484.

[28] Aniket Kittur and Robert E Kraut. 2008. Harnessing the wisdom of crowds in wikipedia: quality through coordination. In *ACM Conference on Computer Supported Cooperative Work & Social Computing (CSCW)*. 37–46.

[29] Karim Lakhani and Robert Wolf. 2005. *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects.* MIT Press, Cambridge.

[30] Jeff Luszcz. 2017. How maverick developers can create risk in the software and IoT supply chain. *Network Security* 2017, 8 (2017), 5–7.

[31] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. 2019. World of code: an infrastructure for mining the universe of open source VCS data. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 143–154.

[32] Edwin Mansfield. 1986. Patents and innovation: an empirical study. *Management Science* 32, 2 (1986), 173–181.

[33] Christopher Mendez, Hema Susmita Padala, Zoe Steine-Hanson, Claudia Hilderbrand, Amber Horvath, Charles Hill, Logan Simpson, Nupoor Patil, Anita Sarma, and Margaret Burnett. 2018. Open source barriers to entry, revisited: A sociotechnical perspective. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1004–1015.

[34] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. 2023. "We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM. ESEC/FSE '23.

[35] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. 2019. Why do people give up FLOSSing? A study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems (OSS)*. Springer, 116–129.

[36] Audris Mockus. 2007. Large-scale code reuse in open source software. In *First International Workshop on Emerging Trends in FLOSS Research and Development (FLOSS'07: ICSE Workshops 2007)*. IEEE, 7–7.

[37] Andy Neely and Jasper Hii. 1998. Innovation and business performance: a literature review. *The Judge Institute of Management Studies, University of Cambridge* (1998), 0–65.

[38] Cassandra Overney, Jens Meinicke, Christian Kästner, and Bogdan Vasilescu. 2020. How to Not Get Rich: An Empirical Study of Donations in Op€n $our¢e. In *International Conference on Software Engineering*. ACM, 1209–1221. ICSE '20.

[39] Huilian Sophie Qiu, Alexander Nolte, Anita Brown, Alexander Serebrenik, and Bogdan Vasilescu. 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)*. 688–699.

[40] Joseph A Schumpeter et al. 1939. *Business Cycles: A Theoretical, Historical and Statistical Analysis of the Capitalist Process*. Vol. 1. Mcgraw-hill New York.

[41] Stanley F Slater and Jakki J Mohr. 2006. Successful development and commercialization of technological innovation: Insights based on strategy type. *Journal of product innovation management* 23, 1 (2006), 26–33.

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

[43] Ashok Subramanian. 1996. Innovativeness: redefining the concept. *Journal of engineering and technology management* 13, 3-4 (1996), 223–243.

[44] Brian Uzzi, Satyam Mukherjee, Michael Stringer, and Ben Jones. 2013. Atypical combinations and scientific impact. *Science* 342, 6157 (2013), 468–472.

[45] 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 *Joint Meeting on the Foundations of Software Engineering (ESEC/FSE)*. 644–655.

[46] Anthony FJ Van Raan. 2004. Sleeping beauties in science. *Scientometrics* 59, 3 (2004), 467–472.

[47] James Walden. 2020. The impact of a major security event on an open source project: The case of OpenSSL. In *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*. 409–419.

[48] Lingfei Wu, Dashun Wang, and James A Evans. 2019. Large teams develop and small teams disrupt science and technology. *Nature* 566, 7744 (2019), 378–382.

[49] Shurui Zhou, Bogdan Vasilescu, and Christian Kästner. 2020. How has forking changed in the last 20 years? A study of hard forks on GitHub. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 445–456.