Giving Back: Contributions Congruent to Library Dependency Changes in a Software Ecosystem

Supatsara Wattanakriengkrai, Dong Wang, Raula Gaikovina Kula, Christoph Treude, Patanamon Thongtanunam, Takashi Ishio, and Kenichi Matsumoto

Abstract—The widespread adoption of third-party libraries for contemporary software development has led to the creation of large inter-dependency networks, where sustainability issues of a single library can have widespread network effects. Maintainers of these libraries are often overworked, relying on the contributions of volunteers to sustain these libraries. To understand these contributions, in this work, we leverage socio-technical techniques to introduce and formalise dependency-contribution congruence (DC congruence) at both ecosystem and library level, i.e., to understand the degree and origins of contributions congruent to dependency changes, analyze whether they contribute to library dormancy (i.e., a lack of activity), and investigate similarities between these congruent contributions compared to typical contributions. We conduct a large-scale empirical study to measure the DC congruence for the npm ecosystem using 1.7 million issues, 970 thousand pull requests (PRs), and over 5.3 million commits belonging to 107,242 npm libraries. We find that the most congruent contributions originate from contributors who can only submit (not commit) to both a client and a library. At the project level, we find that DC congruence shares an inverse relationship with the likelihood that a library becomes dormant. Specifically, a library is less likely to become dormant if the contributions are congruent with upgrading dependencies. Finally, by comparing the source code of contributions, we find statistical differences in the file path and added lines in the source code of congruent contributions when compared to typical contributions. Our work has implications to encourage dependency contributions, especially to support library maintainers in sustaining their projects.

Index Terms—Software Ecosystem, Dependency Changes, npm ecosystem

1 Introduction

The adoption of third-party libraries for contemporary software development has led to the emergence of massive library platforms (i.e., library ecosystems) such as npm for JavaScript[1] which is reported to be relied upon by more than 11 million developers worldwide, and contains more than one million libraries [3]. Other examples include the 427,286 Maven libraries for Java Virtual Machine languages,[2] and the 324,779 PyPI libraries for the Python community,[3] to name a few. These library ecosystems are comprised of a complex inter-connected network of dependencies, where developers adopt many other libraries for their work. As an example, npm libraries directly depend on between 5 to 6 other libraries in the ecosystem on average [40], [2].

Due to their open source nature, the key risk of using these libraries is that contributors (including maintainers) may become demotivated at any time [50], [25] thus interfering with their ability to continue contributions to that library. For instance, recently the maintainers of a vulnerable library expressed their frustration when maintaining a library in a tweet, *'... maintainers have been working sleeplessly on mitigation measures; fixes, docs, CVE, replies to inquiries, etc. Yet nothing is stopping people to bash us, for work we aren't paid for, for a feature we all dislike yet needed to keep due to backward compatibility concerns...'* [4].

Especially for highly dependent libraries, sustained contributions are crucial. This is because these libraries need to constantly change (add, remove, upgrade, and downgrade) to their dependencies, in response to breaking changes or applying critical security patches from elsewhere in the ecosystem [67]. In response to recent attacks on open source libraries [1], Google has deployed an "Open Source Maintenance Crew", tasked to assist upstream maintainers of critical open-source libraries that are used by major technology vendors, including Microsoft, Google, IBM and Amazon Web Services.

Our idea in this paper is to explore the match between contributions that are congruent with dependency changes among these libraries. We borrow the ideas of socio-technical congruence of Cataldo et al. [11], which is the congruency between task dependencies among people and coordination activities. Cataldo et al. find that the social structure of an organisation reflects how tasks are completed. Using the same intuition, we can explore the origins of contributions (i.e., whether they are able to commit to the library or not) that are congruent with dependency changes to a library. Hence, we can reveal insights into the social structure of who is updating these dependencies.

- *S. Wattanakriengkrai, R. Kula, T. Ishio, and K. Matsumoto are with Nara Institute of Science and Technology, Japan.*
  E-mail: {wattanakri.supatsara.ws3, raula-k, ishio, matumoto}@is.naist.jp.
- *D. Wang is with Kyushu University, Japan.*
  E-mail: wang.dong.vt8@is.naist.jp.
- *C. Treude and P. Thongtanunam are with the University of Melbourne, Australia.*
  E-mail: {christoph.treude, patanamon.t}@unimelb.edu.au.

1. https://www.npmjs.com/
2. https://search.maven.org/stats
3. https://pypi.org/
4. https://twitter.com/yazicivo/status/1469349956880408583

More precisely, we introduce and formalise our new concept of dependency-contribution (DC) congruence as the degree to which contributions are congruent with dependency changes.

Through the case of the npm ecosystem, we conduct a large scale empirical study to examine the DC congruence and the library dormancy using over 5.3 million commits, 1.7 million issues, and 970 thousand pull requests (PRs[5]) belonging to 107,242 libraries. We formulate the following research questions.

• **(RQ1)** *What contributions are congruent with dependency changes of libraries that they depend on?*

Motivation: Inspired by Cataldo et al. [11] and to better understand the contributors behind these contributions, we define contribution types based on four different types of contributions which are: (1) contributions from a contributor who commits to both client and library, (2) contributions from a contributor who commits to a client and submits to a library, (3) contributions from a contributor who submits to a client and commits to a library, and (4) contributions from a contributor who submits to both client and library. In terms of the dependency changes, we consider four types of different dependency changes which are (1) adding a dependency, (2) removing a dependency, (3) upgrading a dependency, and (4) downgrading a dependency. To answer RQ1, we break down the question into the following sub-questions: *RQ1a. To what extent do contributors contribute to the libraries they depend on?* and *RQ1b. How do different contribution types and dependency changes contribute to DC congruence?*

Results: Using our DC congruence metrics, we find that contributions most congruent to dependency changes are contributions from contributors that can only submit to both client and library (s-s). Higher congruence is observed with a dependency downgrade over time.

• **(RQ2)** *What is the relationship between the DC congruence and the likelihood of libraries becoming dormant? Motivation:* Since library sustainability heavily relies on contributions [52], we would like to investigate the association between the DC congruence and the library dormancy.

Results: Our survival analysis shows that the different types of DC congruence share an inverse association with the likelihood of a library becoming dormant. For instance, the higher the number of issues from specific types (i.e., s-s, c-s) that are congruent with a dependency upgrade, the lower the likelihood that a library becomes dormant.

• **(RQ3)** *Do the contributions differ depending on their congruence with dependency changes? Motivation:* In addition to measuring DC congruence (RQ1 and RQ2), since contributors may be motivated to submit congruent contributions based on changes in their dependencies, for RQ3, our motivation is to test the hypothesis that congruent contributions are similar based on their congruence (i.e., c-s, s-s, and c-c and s-c).

Results: Comparing contribution similarity in terms of source code and file paths, we find statistical differences in file path and added lines in source code of contributions that are congruent with dependency changes when compared to those that are not congruent. In other words, congruent contributions are not typical contributions by that contributor.

We provide an online appendix, containing datasets and source code related to (a) the ecosystem-level and library-level DC congruence results, (b) the metric data for our survival model analysis, and (c) the file path and source code similarities between contribution types, which is available at https://doi.org/10.5281/zenodo.5677371.

5. PR(s) stands for the pull request(s) throughout the entire paper.

2 Background and Related work

In this section we discuss the background and related work.

Socio-Technical Congruence (STC). Cataldo et al. [12] pioneered the concept of socio-technical congruence (STC), the match between task dependencies among people and coordination activities performed by individuals. Their follow-up studies (Cataldo et al. [11], Cataldo and Herbsleb [10]) then investigated software quality and development productivity. Further studies also presented STC measures from different perspectives. For instance, Valetto et al. [57] proposed a measure of STC in the view of network analysis, while Kwan et al. [41] measured weight STC and the impact of STC on software build success (Kwan et al. [42]). Additionally, Wagstrom et al. [59] measured individualized STC. Other studies then measured STC from other perspectives such as dependencies, knowledge, and resources [33], global software development [47], and for file-level analysis [69] and [44].

Different from these studies, we introduce and formalise metrics that measure the degree of contributions congruent to dependency changes. By borrowing the STC measure proposed by Cataldo et al. [12], we are able to reflect the origins of the contributions (i.e., characterized by whether the contributor has the ability to commit to the library repository aka, a maintainer). Likewise, we formalise the library-level DC congruence, making an adjustment on the STC measure proposed by Wagstrom et al. [59]. We propose that library-level DC congruence can also provide more insights: for example, it can be used to detect certain libraries in which contributions congruent to dependency changes have ceased or are on the decline.

Library Dependency Changes. The research community has carried out a large body of research on understanding the large networks of software library dependencies. Some studies focused on a single ecosystem, focusing on the updating and lags within that ecosystem. For instance, Wittern et al. [2] analyzed a subset of npm libraries, focusing on the evolution of characteristics such as their dependencies and update frequency. Abdalkareem et al. [5] also empirically analyzed "trivial" npm libraries, to suggest that depending on trivial libraries can be useful and risk-free if they are well implemented and tested. Zerouali et al. [68] analyzed the library update practices and technical lag to show a strong presence of technical lag caused by the specific use of dependency constraints. Recent studies include understanding the lag when updating security vulnerabilities [13] and dependency downgrades [16] for the npm ecosystem. There are works that studied other ecosystems. Robbes et al. [49] showed that a number of API changes caused by deprecation can have a large impact on the Pharo ecosystem.

Blincoe et al. [8] proposed a new method for detecting technical dependencies between projects on GitHub and IBM. Bavota et al. [7] studied Apache projects to discover that a client project tends to upgrade a dependency only when substantial changes (e.g., new features) are available. Kula et al. [39] revealed that affected developers are not likely to respond to a security advisory for Maven libraries, while Wang et al. [63] revealed that developers need convincing and fine-grained information for the migration decision. Most recently, He et al. [30] proposed a framework for quantifying dependency changes and library migrations based on version control data using Java projects.

Other studies focused on analyzing multiple ecosystems and comparing the different attributes between them. Decan et al. [21] showed the comparison of dependency evolution from seven different ecosystems. Similarily, Kikas et al. [35], studied the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems. Dietrich et al. [23] studied how developers declare dependencies across 17 different library managers. Stringer et al. [54] analyzed technical lag across 14 library managers and discovered that technical lag is prevalent across library managers but preventable by using semantic versioning based declaration ranges.

Differently, in our work, we focus on how contributions are congruent with dependency changes at the ecosystem level, our motivation is different, as we explore how dependency changes correlate with a library becoming dormant.

Sustainability of Software Ecosystem As with prior work [18], we view software ecosystems as socio-technical networks consisting of technical components (software libraries) and social components (communities of developers) that maintain the technical components. Sustainability is particularly visible in the complex and often brittle dependency chains in OSS library ecosystems like npm, PyPi, and CRAN [21]. Other studies have studied sustainability for R [24], Ruby [4], Python [58], and npm [22]. Through a longitudinal empirical study of Cargo's dependency network, Golzadeh [27] revealed that dependencies to a project lead to active involvement in that project. However, the effect of DC congruence from different types of contributions on ecosystem sustainability is still unknown.

Different from these studies, we take a quantitative approach to investigate the relationship between the DC congruence and the likelihood of libraries becoming dormant.

3 A Congruence of Two Graphs

In this section, we describe how we measure the DC congruence, i.e., the degree to which contributions are congruent with dependency changes as a graph. This is described as the overlap between two separate graphs of dependencies and contributions. Both graph edges have a temporal attribute which provides information when the dependency (or contribution) event occurs. Since the contribution and dependency relationships are dynamic (i.e., changing over time), we capture and analyze these relationships based on a specific period of time.

Dependency Graph (G_D) - Let $G_D = (N_{lib}, E_{dep})$ be a directed graph where N_{lib} refers to the set of software libraries in the ecosystem and $E_{dep} \subseteq N_{lib} \times N_{lib}$ refers to library dependencies. A library $l \in N_{lib}$ has attributes <name, time> where name is the unique name of the library and time is when the library was first created.

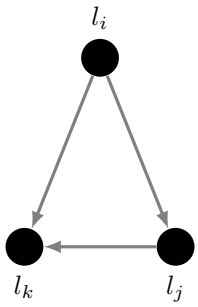
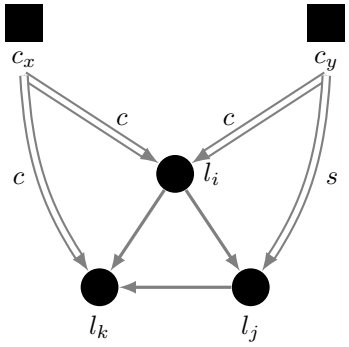
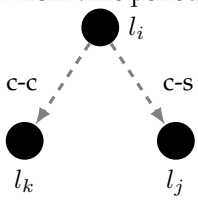


Figure 1: Dependency graph from time period Jan 1 to Mar 31, 2016



(a) Contribution graph from time period Jan 1 to Mar 31, 2016



(b) Dependency-Projected Contribution graph that is projected from Fig. 2a from time period Jan 1 to Mar 31, 2016. Note that the edges now contain directional dependency information

Figure 2: Example graph representation of the relationships between contributions and dependencies

Dependency-projected Contribution Graph (G_{DC}) - It is a projection of two different node types, i.e., developer node (C_x and C_y) and library node (l_i) in Fig. 2 (a). This is defined as a Contribution Graph (G_C), where $G_C = (N_{lib} \cup N_{con}, E_c)$ is a directed graph where N_{lib} refers to the set of software libraries in the ecosystem, N_{con} refers to the set of contributors in the ecosystem. An edge $(c, l) \in E_c$ indicates that a contributor c contributes to a library l. A contributor $c \in N_{con}$ has an attribute <userid> that is the unique identifier of a contributor. To calculate the congruence as an intersection between the contribution and dependency graph, we need to transform G_C from a two-node network (the graph consists of two different types of nodes, i.e., developer and library nodes) to match the dependency graph (single-node network consisting of one node type). Hence, we use a projection technique [29],

Table 1: Classification of contributions by maintainer role to both client and library

Contribution Type	Description	Notation
committer-committer (c-c)	Contributions from a contributor who commits to both client and library.	E_{con}^{c-c}
committer-submitter (c-s)	Contributions from a contributor who commits to a client and submits to a library.	E_{con}^{c-s}
submitter-committer (s-c)	Contributions from a contributor who submits to a client and commits to a library.	E_{con}^{s-c}
submitter-submitter (s-s)	Contributions from a contributor who submits to both client and library.	E_{con}^{s-s}

where the edges still retain the dependency information in G_C. The result is a new graph we define as G_{DC}. Let $G_{DC} = (N_{lib}, E_{con})$ be a directed graph where N_{lib} represents the set of libraries, while $E_{con} \subseteq N_{lib} \times N_{lib}$ represents their relationships with respect to contributors.

3.1 Graph Properties

In this section, we discuss three properties of both graphs. ***Temporal Properties.*** We define a temporal graph $G_{[t1,t2]}$ where $t1$ and $t2$ are two time points between which we capture the contribution and dependency relationships.

Definition 3.1.1. The temporal graph $G_{[t1,t2]}$ is formally defined as: $G_{[t1,t2]} = (N_{[t1,t2]}, E_{[t1,t2]})$ where $N_{[t1,t2]}$ is a set nodes that have relationships with others within a period of $t1$ until $t2$, and $E_{[t1,t2]}$ is a set of edges between the nodes in $N_{[t1,t2]}$ that occur during the period of $t1$ until $t2$.

It is important to note that for any time period (i.e., t1 to t2), a library node must have been created before t1. This means that any library created in one time period will only be considered in the next time period. Temporal operations can be applied to both G_D and G_{DC}.

Dependency Changes. Each dependency change is represented in our dependency graph. The dependency edge has the attributes <change, time> which is the current change of the dependency relationship.

Definition 3.1.2. The dependency edges E_{dep} of a graph G_D represent any dependency changes between two libraries. An edge $(l_i, l_j) \in E_{dep}$ means that a library l_i changed its dependency on library l_j, where l_i is a client to the l_j library.

We categorize dependency changes into four types: (i) added (E_{dep}^{ad}), i.e., a dependency has been added to a library, (ii) removed (E_{dep}^{rm}), i.e., the dependency has been removed from a library, (iii) upgraded (E_{dep}^{up}), i.e., a dependency version has been changed to a more recent version of a library, and (iv) downgraded (E_{dep}^{dn}), i.e., a dependency version has been changed to an older version of a library. It is important to note that the edge of a dependency change that occurs in a time period will be kept in the subsequent time periods until a new dependency event occurs.

Contribution Types. Each contribution is represented in our contribution graph. We identify two contributor roles based on the ability to merge changes: a committer, who has the ability to merge any submitted contributions, and a submitter, who can only submit contributions to a library but does not have the permission to merge the contributions, i.e., they do not have the right to commit changes directly to the main branch of the library repository.

Definition 3.1.3. The contribution edges E_c of a graph G_C have two types: $E_c = E_{comm} \cup E_{sub}$. An edge $(c, l) \in E_{comm}$ means that a contributor c contributes to the library l as a committer. An edge $(c, l) \in E_{sub}$ means that a contributor c contributes to a library l as a submitter.

We project the graph G_C into a one-node graph $G_{DC} = (N_{lib}, E_{con})$ as follows.

Definition 3.1.4. The contribution edges E_{con} of a projected graph G_{DC} have four types: $E_{con} = E_{con}^{c-c} \cup E_{con}^{s-c} \cup E_{con}^{c-s} \cup E_{con}^{s-s}$. An edge $(l_1, l_2) \in E_{con}$ is extracted if and only if there exists a contributor (c) who contributes to both libraries l_1 and l_2; i.e., $(c, l_1) \in E_C \wedge (c, l_2) \in E_C$ holds.

Through projection, edges combine dependency information with committer and submitter information, resulting in four edge combination (see Table 1) as follows: (i) E_{con}^{c-c}, i.e., a committer to both client and library, (ii) E_{con}^{c-s}, i.e., a committer to a client, submitter to library, (iii) E_{con}^{s-c}, i.e., a submitter to client, committer to library, (iv) E_{con}^{s-s}, i.e., submitter to both client and library.

In Figure 2a, the two node graph is shown to be $G_{C[Jan2016,Mar2016]}$, contributor c_x contributed to libraries l_i and l_k, and another contributor c_y contributed to l_i and l_j. Figure 2a is projected in Figure 2b, where c_y commits to l_i and c_y submits to l_j, making these contributions a c-s type.

3.2 Congruence Calculation

Ecosystem-level DC Congruence is the ratio of dependency changes that receive contributions divided by the total number of dependency changes in the ecosystem. Although prior work has represented the congruence using adjacency matrices, we exploit graph theory to define this intersection.

Definition 3.2.1 (Ecosystem-level DC Congruence)**.** This can be described as the generalized equation:

$$DC(dtype, ctype, G_{D[t1,t2]}, G_{DC[t1,t2]}) = \frac{|E_{dep[t1,t2]}^{dtype} \cap E_{con[t1,t2]}^{ctype}|}{|E_{dep[t1,t2]}^{dtype}|} \quad (1)$$

For a time period [t1, t2], dtype is a dependency change (added, removed, upgraded, or downgraded) and ctype is the type of contribution (c-c, c-s, s-c, c-c). For example, given $G_{D[Jan2016,Mar2016]}$ in Figure 1 and all the dependency relationships in the graph having a type of "added", we want to calculate the DC congruence between dependencies of added and c-s contributions (demonstrated in Figure 2b). The DC congruence at the ecosystem level of the graph $G_{D[Jan2016,Mar2016]}$ and $G_{DC[Jan2016,Mar2016]}$ is $DC(added, c-s, G_{D[Jan2016,Mar2016]}, G_{DC[Jan2016,Mar2016]}) = \frac{1}{3} = 0.333$.

Definition 3.2.2 (Library-level DC Congruence)**.** To quantify how the congruence related to a library contributed to the

Table 2: Dataset snapshot statistics

	# N_{pkg}	# PRs	# issues	# commits	# dev
2014	17,859	22,239	55,081	473,591	11,031
2015	57,534	98,975	222,146	1,255,065	46,029
2016	92,748	242,424	427,001	1,415,015	82,991
2017	105,058	186,257	352,411	886,796	96,916
2018	106,815	139,016	267,318	527,252	86,978
2019	107,146	166,588	259,560	428,191	69,285
2020	107,242	115,186	171,435	339,219	43,815
	Total	**970,685**	**1,754,952**	**5,325,129**	**437,045**

congruence value of the whole ecosystem, we also measure congruence of each library (i.e., the congruence at the library level):

$$DC(dtype, ctype, l, G_{D[t1,t2]}, G_{DC[t1,t2]}) =$$

$$\frac{|n(l, E^{dtype}_{dep[t1,t2]}) \cap n(l, E^{ctype}_{con[t1,t2]}|)}{|n(l, E^{dtype}_{dep[t1,t2]})|} \quad (2)$$

$$n(l, E) = \{l' \in N_{lib} \mid (l, l') \in E \vee (l', l) \in E\}$$

The function $n(l, E)$ represents the set of neighbors of a library l. The congruence related to a library l becomes higher if similar libraries are connected to l in the two graphs. Similar to the ecosystem congruence, we define the parameters as a time period [t1, t2], where dtype is a dependency change (added, removed, upgraded, or downgraded) and ctype is the type of contribution (c-c, c-s, s-c, c-c). For example, given $G_{D[Jan2016,Mar2016]}$ in Figure 1 and all the dependency relationships in the graph having a type of "added", we want to calculate the library-level DC congruence of the library l_k between dependencies of added and c-c contributions (demonstrated in Figure 2b). Finally, the library-level congruence of the library l_k is calculated as follows. $DC(added, c-c, l_k, G_{D[Jan2016,Mar2016]}, G_{DC[Jan2016,Mar2016]}) = \frac{1}{2} = 0.5$

Implementation. Following prior work, we translate our graph definitions into adjacency matrices (cf. replication package for details). Hence, for each time period [t1, t2], we calculate all parameter instances of dependency changes (dtype) and contribution types (ctype) to result in 16 combinations of DC congruence.

Intuitively, we can interpret the congruence values (ranging from 0 to 1) as the extent to which contributions are congruent with dependency changes. A congruence value close to 1 at the ecosystem level would indicate that a large number of dependency relationships receive congruent contributions, while a congruence value close to 0 would indicate that there is a small number of congruent contributions in the ecosystem. Similarly, at the individual level, a congruence value close to 1 would indicate that a large number of contributions is congruent with the dependency relationships of that library.

4 Data Preparation

To evaluate our congruence models and answer our RQs, we perform an empirical study on the npm library ecosystem. We select the npm JavaScript ecosystem as it is one of the largest library collections on GitHub, and also has been the focus of recent studies [35], [5], [20]. In this work, we focus on the npm libraries linked to their GitHub repositories, since we will analyze contributions (i.e., PRs and issues). Similar to 'all-the-package-repos' [26] and Chinthanet et al. [13], we use a listing of npm libraries from the npm registry and then match them to repositories available on GitHub. The listing of npm libraries was acquired in Nov. 2020. To collect the required information, we use the GitHub REST API to collect PRs, issues, commits, and other metadata (e.g., license type) of the library repositories included in the latest data snapshot. The issues are used to answer RQ1 and RQ2, while the commits are used to answer RQ2 and the PRs are used to answer all RQs. We do not consider commits when measuring the DC congruence since we cannot identify the direction of commits (i.e., which library they are from) and commits can be only created by maintainers of a library which exclude other contribution types. As part of our data collection, we accessed the API using five GitHub tokens (each belonging to the first five authors) that were generated by the first five authors during the collection process. To comply with the GitHub terms of usage, we only used the API rate limit at 25,000 (5,000 * 5) requests per hour, taking into account periodic rests between requests. In total, the data collection period took one month, lasting from November 1 to November 30, 2020.

Table 2 provides an overview of the number of npm libraries that have repositories available on GitHub for each year. Overall, the number of npm libraries starts from small (i.e., 17,859 libraries), but then reaches over a hundred thousand libraries (i.e., 107,242) as of Nov. 2020. Similar to previous work which considered the temporal dimension [45], [9], [46], we split the data of each year into quarters: (Q1) January 1 to March 31, (Q2) April 1 to June 30, (Q3) July 1 to September 30, and (Q4) October 1 to December 31 to establish the temporal intervals of our graphs (i.e., $G_{[t1,t2]}$).

Extracting Contributions and Dependency Changes Once the time-intervals are established, we then extract the contributions and dependency changes within each quarter. For the contribution activities, we generate the graph nodes by identifying PRs and issues that occurred during the quarter. To classify the role of the contributor, we use the GitHub REST API to identify whether or not the contributor who submitted a contribution (i.e., a PR or an issue) had committed any prior commits into the main branch of the repository. For the dependencies and their changes, similar to prior work [16], we parse the `package.json` file to extract both development and runtime dependencies for each library during that time period. To identify the dependency changes during the time period, we use the PyDriller package[6] to mine the dependency changes of `package.json` in the Git history. More specifically, we focus on the Modification object provided by the PyDriller to investigate commits to `package.json`.

Calculating Ecosystem-Level DC Congruence (RQ1) Answering RQ1 involves computing the DC congruence at the ecosystem level. Hence, for all time periods, we construct the adjacency matrices of the graphs (see Section 3) for

6. https://github.com/ishepard/pydriller

Table 3: Statistical comparisons of Ecosystem-level and Library-level DC congruence of contribution types

Contribution Type	Ecosystem Level	Library Level
s-s >c-c	Medium	-
s-s >c-s	Large	Small
s-s >s-c	Large	Small
c-c >c-s	Medium	Medium
c-c >s-c	Large	Medium
c-s >s-c	Small	-

Effect size: Negligible $|\delta| < 0.147$, Small $0.147 \leq |\delta| < 0.33$, Medium $0.33 \leq |\delta| < 0.474$, Large $0.474 \leq |\delta|$

PRs and issues separately. To ease computational power, we filter out all rows and columns of the adjacency matrices that have only zeros. Since we analyze PRs and issues separately, we generate a total of 32 instances for each quarter, resulting in 864 combinations of DC congruence. It took around 21 days of execution based on two mainframe servers. The first server is comprised of an Intel Core 2.60GHz, with 10 cores and 252GB of RAM, and the second server is comprised of AMD Core 3.70GHz, with 32 cores and 252GB of RAM.

Calculating Library-Level DC Congruence (RQ2) Answering RQ2 involves computing the library-level DC congruence. Using the constructed matrices (before filtering the rows and columns with zeros) from RQ1, we calculate the library-level DC congruences for each time period. Similar to RQ1, we generate a total of 32 DC congruence values for each library in each quarter. Since the experiment are conducted in parallel with RQ1, the computational time and resources are the same.

Sampling Contributions (RQ3) Answering RQ3 involves a quantitative analysis of contributions. Hence, we draw a statistically representative random sample of 1,841 contributors. Our sample size allows us to generalize conclusions about the ratio of contributors to all contributors with a confidence level of 99% and a confidence interval of 3%, as suggested by prior work [6] based on the Sample Size Calculator.[7] To collect the most recent contributions of the sampled contributors, we extract contributions belonging to the most recent quarter. To do so, we first identify the last contribution, i.e., the last submitted PRs in our dataset, and then we extract all their PRs submitted within that time period. In the end, we collect 27,554 PRs. Since our approach involves similarity analysis of the source code (at least one .js file) and file path components, we obtain 11,405 (41%) PRs from the 1,841 contributors with the four contribution types (see Section 3.1).

5 Results

In this section, we present the results to answer our RQs.

5.1 Assistance from the Ecosystem (RQ1)

Approach. To answer RQ1, we analyze the DC congruence in terms of contributions and dependency changes. In particular, we address each of the sub-questions as follows: To answer RQ1a: *To what extent do contributors contribute to the libraries they depend on?*, we summarize the DC congruence at the ecosystem level and library level in terms of the distribution by contribution type. To statistically confirm the differences in the four contribution types, we use the Kruskal-Wallis H test [38]. This is a non-parametric statistical test to use when comparing two or more types. We test the null hypothesis that *'DC congruence values of two different contribution types are the same.'* We also measure the effect size using Cliff's δ, a non-parametric effect size measure [14]. Effect size is analyzed as follows: (1) $|\delta| < 0.147$ as Negligible, (2) $0.147 \leq |\delta| < 0.33$ as Small, (3) $0.33 \leq |\delta| < 0.474$ as Medium, or (4) $0.474 \leq |\delta|$ as Large [51].

To answer RQ1b: *How do different contribution types and dependency changes contribute to DC congruence*, we analyze trends of DC congruence related to different contribution types and dependency changes. We use the Spearman rank-order correlation coefficient to analyze the correlations between the different dependency changes for each kind of contribution. A correlation between 0 – 0.09 is considered as negligible, 0.1 – 0.39 as weak, 0.4 – 0.69 as moderate, 0.7 – 0.89 as strong, and 0.9 – 1 as very strong [53].

Congruent contributions (RQ1a). Figure 3a and 3b show the relative congruence score between contributions at both the ecosystem level and the library level. It is important to note that the box-plot of each contribution type presents the congruence scores between the contributions and dependency changes of the different time periods (see Section 4). In the Figure, we can see that s-s has the highest congruence at the ecosystem level, however at the library level both s-s and and c-c have a higher relative DC congruence. As shown in Table 3, the effect sizes range from Small to Large, confirming statistical significance for both the ecosystem and library level congruence metrics.

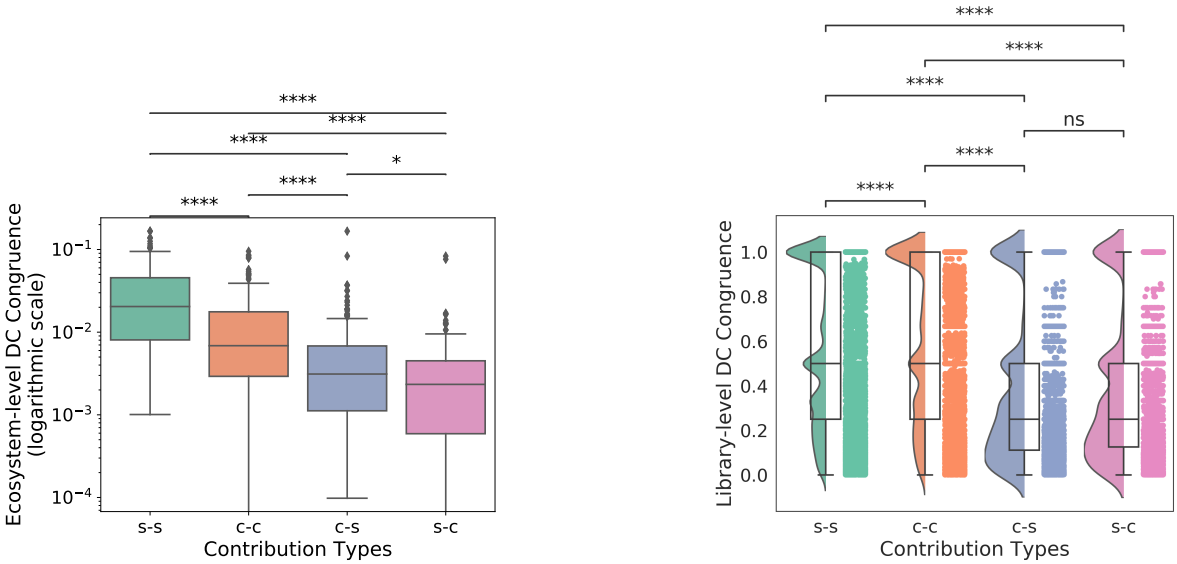
From a temporal perspective, as shown in Figure 4, we find that the highest congruence values occur generally in the second quarter of 2014, followed by the first quarter of 2015. The congruence values may be correlated with the ecosystem size.

Attributes that correlate with the DC congruence (RQ1b) From Figure 4, we are able to make two observations. First, we find that there is higher congruence with contributions when a dependency is downgraded (see the purple lines in Figure 4). This finding is consistent between all contribution types and between issues and PRs, except for the congruence between PRs of s-c and dependency upgrades. The second observation is that the s-s contributions are most congruent with dependency changes. This is consistent from both the issues and PRs. This provides evidence that library dependency changes are satisfied by contributors that do not have the ability to commit to these libraries.

Table 4 shows evidence that all dependency changes are correlated with each other. Statistically, we tested to find that any dependency changes are moderately to strongly correlated to each other for all contribution types, except for the s-c contributions (added to downgraded, and removed to downgraded). We now return to answer RQ1, which we summarize below:

7. https://www.surveysystem.com/sscalc.htm

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3225197



(a) Ecosystem-level DC congruence

(b) Library-level DC congruence

Figure 3: DC congruence of contribution types (ns: no significance, **: p-value < 0.01, ***: p-value < 0.001, ****: p-value < 0.0001)

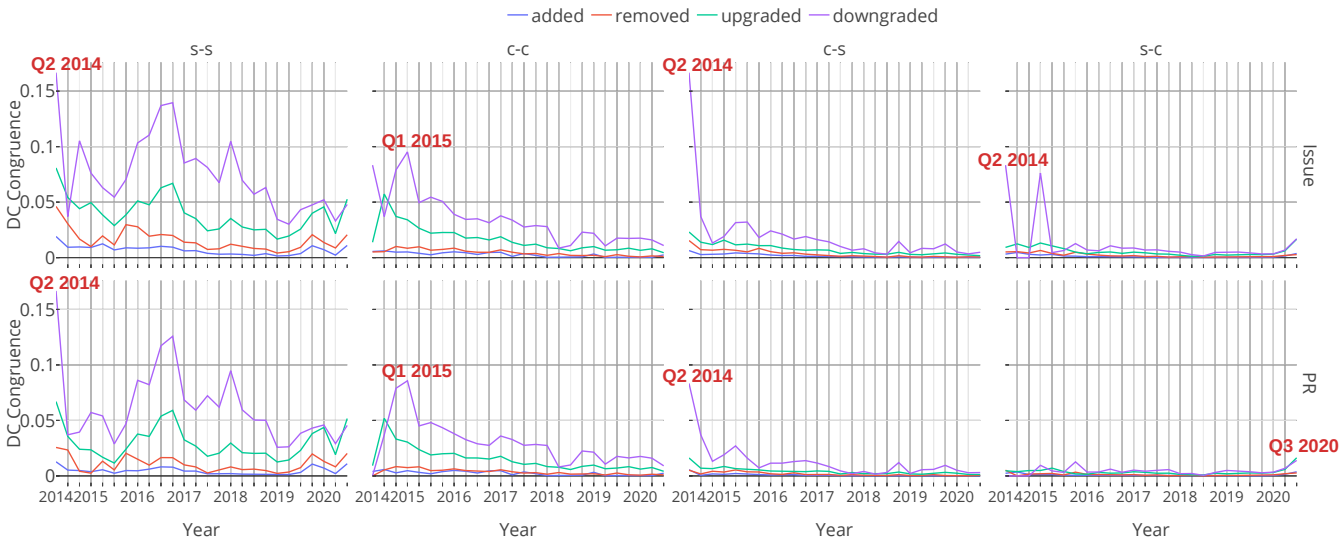


Figure 4: Ecosystem-level DC congruence over time. The red text indicates quarters in which congruence values are highest

RQ1 Summary

Using our DC congruence metrics, we find that contributions most congruent to dependency changes are contributions from contributors that can only submit to both client and library (s-s). Higher congruence is observed with a dependency downgrade over time.

5.2 Contribution Congruence and Library's Dormancy (RQ2)

Approach. To address RQ2, we analyze the relationship between the DC congruence and the likelihood of libraries becoming dormant. In particular, we leverage survival analysis techniques which allow us to model the relationship across different time periods. We use a Cox proportional-hazard model (i.e., a commonly-used survival analysis model) [19] to capture the risk of an event (i.e., a library becomes dormant) in relation to factors of interest (e.g., the DC congruence) in the elapsed time.

To do so, we extract the event and the factors for each library in each time period (i.e., a quarter). In our Cox model, the dependent variable is the event that a library becomes dormant at a particular time point. In line with prior work [15], [62], [58], a project is regarded as dormant if it is no longer being maintained or does not have development activity. Specifically, similar to the work of [58], we consider a library as *dormant* in the quarter q if it had less than one commit on average for four consecutive quarters (i.e., from the quarter q until the quarter $(q + 3)$). For example, in the first, second, third, and fourth quarters in 2018, a library

Table 4: Correlations between different dependency changes for each type of contribution

	s-s		c-c		c-s		s-c	
Dependency change to Dependency change	**PR corr. strength**	**issue corr. strength**	**PR corr. strength**	**issue corr. strength**	**PR corr. strength**	**issue corr. strength**	**PR corr. strength**	**issue corr. strength**
added to removed	Strong	Strong	Moderate	Strong	Strong	Very strong	Moderate	Strong
added to upgraded	Strong	Strong	Moderate	Strong	Very strong	Very Strong	Strong	Very strong
added to downgraded	Moderate	Moderate	Moderate	Strong	Strong	Strong	Weak	Weak
removed to upgraded	Strong	Strong	Strong	Strong	Strong	Very strong	Moderate	Strong
removed to downgraded	Moderate	Moderate	Strong	Strong	Moderate	Strong	Weak	Moderate
upgraded to downgraded	Strong	Strong	Strong	Moderate	Very Strong	Strong	Moderate	Moderate

Correlation: Negligible is 0 to 0.09, Weak is 0.1 to 0.39, Moderate is 0.4 to 0.69, Strong is 0.7 to 0.89, Very strong is 0.9 to 1.

that has 1, 0, 0, and 1 commits, respectively is considered as *dormant* since the first quarter in 2018. This is because the average number of commits for four consecutive quarters is 0.5 ($\frac{2}{4}$). Note that we exclude libraries that have the first commit after October 2019, since their historical data is not sufficient to identify whether the library is dormant or not (i.e., less than four consecutive quarters). As our RQ1 shows that the correlations between different dependency changes for all contribution types vary, it is also possible that these DC congruence may provide various signals to the library dormancy given different dependency changes and contribution types. Thus, for independent variables in our Cox model, we use 32 metrics which measure the library-level DC congruence (see Section 4). Since prior work has shown that project characteristics can be associated with the chance of libraries becoming dormant [58], we also include six project characteristics (see Table 5) as control variables in our Cox model.

To construct the Cox model, we use a similar approach as Valiev et al. [58], which has the following three steps. First, we perform log-transform on the independent variables with skewed distribution to stabilize the variance and reduce heteroscedasticity, which will improve the model fit. We manually observe the skewness of the distribution using histogram plots. Second, we check for the multicolinearity between independent variables using the variance inflation factor (VIF) test and the Spearman rank correlation (ρ). We remove the variables that have VIF scores above the recommended maximum of 5 [17] and we also remove the pairs of variables that have an absolute Spearman correlation coefficient of above or equal to 0.7 [37]. To improve the fit of the model, we apply the non-linear relationships between the dependent variable and independent variables, using restricted cubic splines. To avoid over-fitting, we use the Spearman multiple ρ^2 to determine the appropriate degrees of freedom (i.e., the number of regression parameters) to the variables that have more potential for sharing a nonlinear relationship with the dependent variable. Third, we test the assumption of constant hazard ratios overtime. To do so, we employ graphical diagnostics of Schoenfeld residuals [28]. We observe that Schoenfeld residuals of the remaining variables have a random pattern against time, which implies that the Cox model assumptions are not violated.

To analyze the association between each independent variable and the likelihood of a library becoming dormant, we examine the coefficient values and effect sizes of the independent variables. The coefficient values in the Cox model are hazard ratios, where a hazard ratio above 1 implies that a variable is positively associated with the event probability, while a hazard ratio below 1 indicates an inverse association. For the effect size, we perform ANOVA type-II to obtain Log-likelihood Ratio χ^2 (LR χ^2). The larger the LR χ^2 of the variable is, the stronger the relation of the variable to the likelihood of a library becoming dormant.

Table 5: Project characteristics [58] that are used as control variables in our Cox model

Metric	**Description**
#Commits	The number of commits occurring in the observed quarter.
#Contributors	The number of GitHub users who have authored commits in the observed quarter.
Core Team Size	The number of GitHub users responsible for 90% of contributions in the observed quarter.
#Issues	The number of issues created in the observed quarter.
#Non-Developer Issue Reporters	The number of GitHub users who did not author any commits but created an issue in the observed quarter.
License Type	The license type that the studied library used in the observed quarter: Strong copy-left (e.g., GPL, Affero), Weak-copy-left (e.g., LGPL, MPL, OPL), or non-copy-left (e.g., Apache, BSD)

Table 6: Survival analysis statistics

npm Ecosystem Survival Analysis
response: dormant = TRUE
R^2 = 42.2%
concordance index = 0.672

			Coeffs (Err.)	LR Chisq
Log number of commits			7.04 (0.01)***	39144***
Log number of commits'			0.15 (0.01)***	
Log number of non-developer issues			0.82 (0.01)***	2783***
Log number of non-developer issues'			0.78 (0.02)***	
Core size of the team			0.67 (0.01)***	976***
Strong copy-left license (vs none)			0.88 (0.02)***	757***
Weak copy-left license (vs none)			0.69 (0.03)***	
Non-copy-left license (vs none)			0.82 (<0.01)***	
Dependency Change	Contribution Type	Type	Coeffs (Err.)	LR Chisq
Upgraded	**c-c**	**PR**	**0.85 (0.05)****	**9****
	s-s	**issue**	**0.56 (0.04)*****	**214*****
	s-c	**PR**	**0.49 (0.20)*****	**14*****
	c-s	**issue**	**0.67 (0.10)*****	**15*****
Downgraded	**c-c**	**PR**	**0.64 (0.18)***	**6***
	s-s	**PR**	**0.58 (0.12)****	**23*****
	s-c	PR	0.11 (1.2)	7*
	c-s	issue	0.72 (0.27)	1
Added	**c-c**	**PR**	**0.74 (0.07)*****	**20*****
	s-s	**PR**	**0.79 (0.08)****	**9****
	s-c	PR	0.61 (0.26)	4*
	c-s	PR	1.02 (0.18)	0
Removed	**c-c**	**PR**	**1.23 (0.10)***	**4***
	s-s	PR	0.86 (0.09)	3
	s-c	issue	0.81 (0.24)	1
	c-s	PR	0.96 (0.22)	0

* p<0.05; ** p<0.01; *** p<0.001
# Issues, # Contributors, twelve congruence variables concerning issues (e.g., c-c added, c-c downgraded, s-c upgraded, etc.) and four congruence variables concerning PR (e.g., s-s upgraded, c-s upgraded, etc.) are removed during the multicolinearity analysis.

Table 6 shows that our Cox proportional-hazard model achieves an adjusted R^2 of 42.2%. Based on the prior quantitative empirical studies in open source projects [58], [36], [61], this score is considered as acceptable since our model is supposed to be explanatory not predictive. In addition, the concordance index (another popular metric

that quantifies the correlation between risk predictions and event times [29]) of our model is 0.672, suggesting that our model performs equally well.

Association between the contribution congruence and the chance of libraries becoming dormant. Our results highlight nine congruence-related variables that associate with the likelihood of a library becoming dormant, i.e., two variables concerning issue (*s-s upgraded* and *c-s upgraded*) and seven variables concerning PR (*s-s added*, *s-s downgraded*, *c-c added*, *c-c removed*, *c-c downgraded*, *c-c upgraded*, and *s-c upgraded*).

As shown in Table 6, we observe that congruence regarding a dependency upgrade or downgrade has a relatively great effect. For instance, *s-s upgraded (issue)* and *s-s downgraded (PR)* have the largest LR χ^2 values (see Table 6), being 214 and 23, respectively. Coefficient values indicate an inverse association with the likelihood of a library becoming dormant. For example, the coefficient value of *s-s upgraded (issue)* indicates that the higher the number of issues of s-s that are congruent with a dependency upgrade, the lower the likelihood that a library becomes dormant. Similarly, the coefficient value of *c-s upgraded (issue)* indicates that the higher the number of issues of c-s that are congruent with a dependency upgrade, the lower the likelihood that a library becomes dormant. While for the congruence regarding a dependency removal, the coefficient of one variable (*c-c removed*) indicates a positive association with the likelihood of a library being dormant, but showing little effect.

These results suggest that a library is less likely to become dormant if the contributions are congruent with upgrading dependencies. A possible reason for upgrading a dependency is to mitigate security vulnerabilities [39].

It is important to note that dormant projects do not mean that these projects have been abandoned. These projects might be referred to as being dormant as their features are completed. In the case of libraries, a dormant library might be still heavily used by other libraries. However, prior work [15] suggests that only 2% of open source projects reach this maturity.

RQ2 Summary

Our survival analysis shows that the different types of DC congruence share an inverse association with the likelihood of a library becoming dormant. For instance, the higher the number of issues from specific types (i.e., s-s, c-s) that are congruent with a dependency upgrade, the lower the likelihood that a library becomes dormant.

5.3 Similarities between Contributions (RQ3)

Approach. To answer RQ3, we investigate whether the type of contributions that are congruent with dependency changes are the same as other contributions by analyzing the source code similarity and file location similarity. Hence, we focus on contributions that are congruent with dependency changes of libraries that they depend on (i.e., c-s), comparing against contributions from the ecosystem (i.e., s-s), and then the rest of maintainer contributions (i.e., c-c and s-c). We compute similarity based on two common methods to compare the content of PRs, which are (i) source code and (ii) file path similarity [60], [43], [32]. More specifically, we use the source code similarity to measure whether the two contributions share similar source code; and use the file path similarity to measure whether the two contributions modify similar components. From our sample, we compare the similarities between PRs of the same contribution type of a contributor. As a PR may modify multiple files, we compare all components (i.e., source code lines and file paths).

Our source code similarity measure is based on the Jaccard index of tokens which approximates the edit distance between two PRs [56]. Following the analysis of repeated bug fixes [66], to compare a pair of PRs, we take only added lines by the PRs into account. Formally, for any PR (p), the source code similarity is defined as follows:

$$SourceCodeSim(p_1, p_2) = \frac{|trigrams(p_1) \cap trigrams(p_2)|}{|trigrams(p_1) \cup trigrams(p_2)|} \quad (3)$$

where $trigrams(p)$ is a multiset of trigrams (3-grams) of source code tokens extracted from all source code lines added by a PR (p). A higher similarity indicates that a larger amount of source code is shared. The similarity is an extension of file similarity defined by Ishio et al. [31] for PRs. This measure has been widely adapted in various software engineering studies [31], [34], [64] and is less affected by moved code in a PR, compared with other measures. On that same note, we apply the file location-based model that is used in typical code review recommendations [55], [65], [60] to compute file path similarity that is defined as follows:

$$StringSim_{LCx}(f_1, f_2) = \frac{LCx(f_1, f_2)}{max(Length(f_1), Length(f_2))}$$

where the $LCx(f_1, f_2)$ function has a parameter to specify how to compare file path components f_1 and f_2. Four different comparison techniques, i.e., Longest Common Prefix (LCP), Longest Common Suffix (LCS), Longest Common Substring (LCSubstr), and Longest Common Subsequence (LCSubseq) are used in the LCx function. The comparison function value is normalized by the maximum length of each file path in f_1 and f_2, i.e., the number of file path components. To calculate the file path similarity between two PRs, we do a pairwise comparison of all file paths in the two PRs and then summarize these by taking the average score. Since we use four different comparison techniques, four similarity scores between two PRs are retrieved and we also summarize these by taking the average score. A higher similarity indicates that a larger amount of components are shared in the PRs.

Once we compute the similarity between PRs in each contribution type (i.e., c-s, s-s, and c-c and s-c), we now analyze whether the similarity of the c-s contributions is different from the similarity of the other contributions. To visualize this, we use a box-plot to show the distributions of similarities between PRs in each of the contribution types. Similar to RQ1a, we use the Kruskal-Wallis H test to statistically confirm the differences in the three contribution types with the null hypothesis that *'two different types have similar contributions.'*, and measure the effect size using the Cliff's δ.

This article has been accepted for publication in IEEE Transactions on Software Engineering. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/TSE.2022.3225197

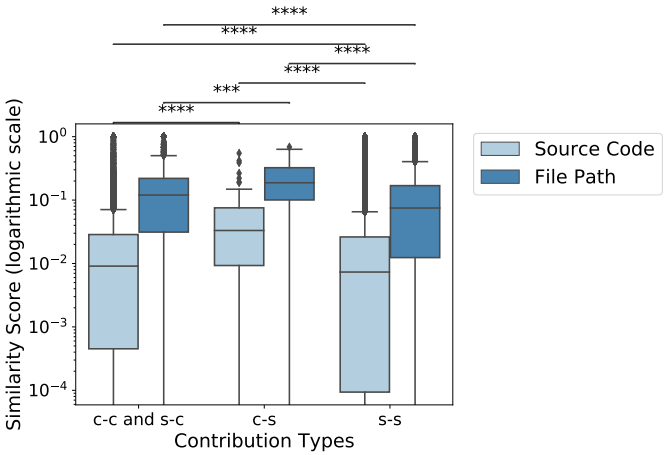


Figure 5: Comparisons of contribution types (ns: no significance, **: p-value < 0.01, ***: p-value < 0.001, ****: p-value < 0.0001)

Table 7: Statistical comparisons of contribution types

	Source Code	File Path
c-s > (c-c and s-c)	Medium	Small
s-s > (c-c and s-c)	-	-
c-s > s-s	Medium	Medium

Effect size: Negligible $|\delta| < 0.147$, Small $0.147 \leq |\delta| < 0.33$, Medium $0.33 \leq |\delta| < 0.474$, Large $0.474 \leq |\delta|$

Similarities/differences between congruent contributions and other contributions. Figure 5 shows the comparisons of source code and file path similarities between the three types of contributions ("c-c and s-c", "c-s", and "s-s"). We find that overall the similarities of contents of PRs in terms of the source code and file paths are relatively low (i.e., scores of 0.01 to 0.19). This could be explained by our method to aggregate all source code and file paths for comparison, which is different to typical file-level similarity comparison techniques [48]. Relatively, we see that the file path similarity is higher than the source code similarity.

In terms of statistical significance, from Figure 5, we make two key observations. First, we find significant differences in the file path and added lines in source code between c-s and other contributions. This suggests that the content of contributions that are congruent with dependency changes may differ from the other contributions made to either their own libraries or other libraries. On the same note, we find that there are significant differences in the file path and added lines of source code between c-c, compared to s-c and s-s contributions, which suggests that contributors may be submitting different contributions to libraries that they are maintaining, compared to contributions to other libraries. Confirming statistical significance, we show that the effect size ranges from small to medium strength as shown in Table 7.

RQ3 Summary

Comparing contribution similarity in terms of source code and file paths, we find statistical differences in file path and added lines in source code of contributions that are congruent with dependency changes when compared to those that are not congruent. In other words, congruent contributions are not typical contributions by that contributor.

6 Discussion

We now discuss our results, implications, and threats to validity.

6.1 Peaks of DC Congruence and Global Events

The results in RQ1 (i.e., Figure 4) show that DC congruence is changing over time. Interestingly, we find that the official npm blog[8] showed two dependency events that may correlate with the observed peaks. It is also important to note that we do not aim to draw a causal relationship, but only observe correlations.

• *June, 2015 - npm 3 is released*. According the npm blog,[9] the release has a notable breaking change when migrating to npm version 3. According to our DC congruence values in Figure 4, we see the peak changes in the congruence between s-s contribution and dependency remove (i.e., 0.019 for issues and 0.013 for PRs), and c-s contribution and dependency downgrade (i.e., 0.031 for issues and 0.027 for PRs).

• *March 2016 - Left-pad incident*. npm attracted press attention after a library called `left-pad`, which many popular JavaScript libraries depended on, was unpublished. It caused widespread disruption, leading npm to change its policies regarding unpublishing libraries. According to our DC congruence values in Figure 4, we see peak changes in the congruence between s-s contribution and dependency downgrade (i.e., 0.103 for issues and 0.086 for PRs), and s-s contribution and dependency upgrade (i.e., 0.051 for issues and 0.037 for PRs).

6.2 Implications

1. Finding contributors for dependency maintenance: To help project teams avoid dormancy, we suggest project teams to build on our findings from RQ1 which show that there is more potential for contributions from the ecosystem. As highlighted by RQ1, the most congruent contributions when dependency changes originate from the ecosystem itself (i.e., s-s contributions that do not have the ability to commit to either the library or a client library). Furthermore, RQ2 results show that associations indeed exist between levels of congruence and dormancy (e.g., significant congruency with a dependency upgrade from all studied contribution types). These results also raise another question, as to why these contributors would have the motivation

8. Available at https://blog.npmjs.org/
9. https://blog.npmjs.org/post/122450408965/npm-weekly-20-npm-3-is-here-ish.html

to contribute an upgrade. Speculation is that these contributions are from a user of both the library and the client who would like to mitigate the security vulnerabilities and further ensure their sustainability. Our work shows that these kinds of contributions are needed and highlights the importance of the ecosystem to both library and the client in terms of maintenance.

Another interesting avenue is in terms of helping newcomers make their first contribution to open source projects. Several approaches have been proposed to help newcomers find good first issues, e.g., Tan et al. [52]. Our results show that contributions congruent to dependency changes might be a potential type of contribution that newcomers might consider.

2. Libraries with low-dependency-congruent contributions are more likely to be dormant: Using our congruence metrics, we find a correlations between congruent contributions and libraries becoming dormant. In our RQ2, we found that a large number of libraries become dormant in the early stages, with more than 90% of libraries becoming inactive until the end-time of our study. Interestingly, this is the case for both upgrading or downgrading a dependency.

3. Awareness of automated dependency maintenance is evident: Another implication is that awareness and dependency maintenance tooling have become part of the ecosystem. From the results of RQ1, we find that congruent contributions may originate from automated sources like automated tooling. Currently there are numerous tools to assist with awareness of when a new version of an existing dependency is available (bots like dependabot).

4. Ecosystem-level governance for dependency maintenance: Since library survival has a relationship with contributions congruent to dependency maintenance, we believe that this has implications that affect ecosystem health, resilience, and governance. This points to implications such as implementing collective support into existing policies and governance. Individual libraries may suffer from dependency management issues, especially when security is involved. For instance, Log4J maintainers state how they became overwhelmed (as mentioned in Introduction).

The DC congruence metrics could be a potential indicator for such suffering libraries, with potential guidelines to keep libraries alive with healthy congruent contributions to dependency maintenance. For example, the ecosystem governors should put in place some kind of community-maintained libraries, i.e. libraries that are overlooked by npm, and that are maintained by a small group of well-defined community volunteers, in order to avoid those important libraries becoming dormant. One example is the faker library: when the maintainer self-sabotaged the library, the community had to step in to assist with its maintenance [10].

6.3 Threats to Validity

Internal Validity - We discuss five threats to internal validity. The first threat is the correctness of techniques used in our mining task. We use the listed dependencies and version numbers as defined in the package.json meta-file. Since we based our mining techniques on prior work, we are confident that our results are replicable. The second threat is related to the constructed matrices for measuring the DC congruence. Since we pioneer the investigation of DC congruence, we constructed binary matrices in this study. We will calculate weight matrices, e.g., a matrix captures the number of contributions (issues and PRs) to investigate the impact of contributions on the DC congruence, in future work. Also our work only considers the first change during the time period, hence, other changes during this period may not be counted, resulting in a threat in our results. We consider this as future work. Additionally, since we focus on issues and PRs for the relationships between contributors, we are unable to claim causation. There might be other factors (e.g., contributor-contributor relations through messaging and commenting) that explain the correlations that should be considered by future studies. The third threat is related to the results derived from the survival model. Although we observe association between explanatory and dependent variables, we cannot infer causal effect of explanatory variables. The fourth threat is related to the factors used in our survival model. Other factors might also influence the chance of libraries becoming dormant. We acknowledge that our factors are based on those proposed by Valiev et al. [58] as a baseline. Tool selection has its threats, as different tools and techniques may lead to different results (e.g., Jaccard index for the source code similarity calculation). We are confident in our results, since all employed tools and techniques are those that have been used in prior work and are well-known in the software engineering community. For instance, we use Pydriller and GitHub REST API and our technique for distinguishing the type of dependency changes relies on related work [16].

External Validity - The main threat to external validity is the generalizability of our results to other library ecosystems. Since the npm ecosystem is the largest library ecosystem, we believe that our model and measurements could easily be applied to other ecosystems that have similar library management systems, such as PyPI for Python and Maven for Java. Our metrics is robust enough to be applied to different ecosystems, however, we do acknowledge that we have no evidence that the results will be the same. We will explore this in future work.

Construct Validity - The main threat to construct validity is the identification of dependency changes. It is possible that some dependency changes that were not reflected in the commits are not identified when we construct the dataset. Furthermore, there might be corner cases when using version ranges might affect the "upgraded" classification. We collect the dependency changes from commits, i.e., an approach suggested by prior work, which is the state of the art [16]. We do rely on the SOTA, however, acknowledge this threat.

Conclusion Validity - We discuss two threats to conclusion validity. The first threat is the correctness of statistical tests. We apply non-parametric tests in our analyses. Since we verify normality assumptions, we are confident that proper statistical tests are selected in this work. The choice of survival analysis technique is another threat to conclusion validity. We use the Cox proportional-hazard model to answer the second research question. We are confident in our model as it achieves an adjusted R^2 of 42.2%, suggesting the

10. https://fakerjs.dev/about/announcements/2022-01-14.html

goodness-of-fit of the model is acceptable.

7 Conclusion and Future Directions

This work investigates how contributors whose code depends on other libraries in the npm ecosystem, submit contributions to assist these libraries. Borrowing socio-technical techniques, we were able to measure the congruence between these congruent contributions, and changes in dependencies for the ecosystem. Results indicate that libraries indeed rely on the ecosystem for contributions, and that these congruent contributions share a relationship with the likelihood of a library becoming dormant, suggesting that the congruency between contributions and dependency changes could be used as an indicator of library dormancy. Moreover, congruent contributions differ from typical contributions. Our results show that contributors do depend on the ecosystem, and should be encouraged to support libraries that they depend on.

Insights that are revealed from the study open many avenues for future work, such as investigating whether maintainers of a library wish to contribute to some of the libraries that depend on their libraries and their reasons to contribute to these libraries, introducing contributor-contributor relations (messaging or commenting among contributors) into the analysis of DC congruence, exploring more precise measures of the DC congruence such as weighting metrics and empirical studies into causal effects for DC congruence changes (i.e., using measures such as causal inference), and developing approaches that make it easier for contributors to understand where in the ecosystem their contributions can have the most impact.

Acknowledgement

This work is supported by Japanese Society for the Promotion of Science (JSPS) KAKENHI Grant Numbers 20K19774 and 20H05706. P. Thongtanunam was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE210101091).

References

[1] "Google: Here comes our 'open source maintenance crew' — zdnet," https://www.zdnet.com/article/google-here-comes-our-open-source-maintenance-crew/, (Accessed on 08/18/2022).
[2] "A look at the dynamics of the javascript package ecosystem."
[3] "npm," https://www.npmjs.com/, (Accessed on 05/16/2021).
[4] "socio-technical evolution of the ruby ecosystem in github."
[5] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, "Why do developers use trivial packages? an empirical case study on npm," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, p. 385–395.
[6] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," 2020.
[7] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "How the Apache community upgrades dependencies: an evolutionary study," *Empirical Software Engineering*, vol. 20, no. 5, p. 1275–1317, 2015.
[8] K. Blincoe, F. Harrison, N. Kaur, and D. Damian, "Reference coupling: An exploration of inter-project technical dependencies and their characteristics within large software ecosystems," *Information and Software Technology*, vol. 110, pp. 174–189, 2019.
[9] C. Brindescu, I. Ahmed, C. Jensen, and A. Sarma, "An empirical investigation into merge conflicts and their effect on software quality," *Empirical Software Engineering*, vol. 25, no. 1, p. 562–590, 2020.
[10] M. Cataldo and J. D. Herbsleb, "Coordination breakdowns and their impact on development productivity and software failures," *IEEE Transactions on Software Engineering*, pp. 343–360, 2013.
[11] M. Cataldo, J. D. Herbsleb, and K. M. Carley, "Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity," in *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2008, p. 2–11.
[12] M. Cataldo, P. A. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Identification of coordination requirements: Implications for the design of collaboration and awareness tools," in *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*, 2006, p. 353–362.
[13] B. Chinthanet, R. G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto, "Lags in the release, adoption, and propagation of npm vulnerability fixes," *Empirical Software Engineering*, pp. 1–28, 2021.
[14] N. Cliff, "Dominance statistics: Ordinal analyses to answer ordinal questions," *Psychological Bulletin*, vol. 114, pp. 494–509, 1993.
[15] J. Coelho and M. T. Valente, "Why modern open source projects fail," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 186–196.
[16] F. R. Cogo, G. A. Oliva, and A. E. Hassan, "An empirical study of dependency downgrades in the npm ecosystem," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.
[17] J. Cohen, P. Cohen, S. G. West, and L. S. Aiken, "Applied multiple regression," *Correlation Analysis for the Behavioral Sciences*, 1983.
[18] E. Constantinou and T. Mens, "An empirical comparison of developer retention in the RubyGems and npm software ecosystems," vol. 13, p. 101–115, 2017.
[19] D. R. Cox and D. Oakes, *Analysis of survival data*. CRC Press, 1984.
[20] A. Decan, T. Mens, and M. Claes, "An empirical comparison of dependency issues in OSS packaging ecosystems," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, 2017, pp. 2–12.
[21] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, 2019.
[22] T. Dey, Y. Ma, and A. Mockus, "Patterns of effort contribution and demand and user classification based on participation patterns in npm ecosystem," 2019, p. 36–45.
[23] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories*, 2019, pp. 349–359.
[24] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 243–252.
[25] M. A. Gerosa, I. S. Wiese, B. Trinkenreich, G. J. P. Link, G. Robles, C. Treude, I. Steinmacher, and A. Sarma, "The shifting sands of motivation: Revisiting what drives contributors in open source," *2021 IEEE/ACM 43rd International Conference on Software Engineering*, pp. 1046–1058, 2021.
[26] GitHub - nice-registry/all-the-package-repos, "Normalized repository URLs for every package in the npm registry. Updated daily." https://github.com/nice-registry/all-the-package-repos, 2017, (Accessed on 08/28/2021).
[27] M. Golzadeh, "Analysing socio-technical congruence in the package dependency network of Cargo," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 08 2019.
[28] P. M. Grambsch and T. M. Therneau, "Proportional hazards tests and diagnostics based on weighted residuals," *Biometrika*, pp. 515–526, 1994.
[29] F. E. Harrell Jr, K. L. Lee, and D. B. Mark, "Multivariable prognostic models: issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors," *Statistics in medicine*, vol. 15, no. 4, pp. 361–387, 1996.
[30] H. He, R. He, H. Gu, and M. Zhou, "A large-scale empirical study on Java library migrations: prevalence, trends, and rationales," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, p. 478–490.

[31] T. Ishio, Y. Sakaguchi, K. Ito, and K. Inoue, "Source file set search for clone-and-own reuse analysis," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*, 2017, pp. 257–268.

[32] J. Jiang, D. Lo, J. Zheng, X. Xia, Y. Yang, and L. Zhang, "Who should make decision on this pull request? analyzing time-decaying relationships and file similarities for integrator prediction," *Journal of Systems and Software*, vol. 154, pp. 196–210, 2019.

[33] L. Jiang, K. Carley, and A. Eberlein, "Assessing team performance from a socio-technical congruence perspective," *2012 International Conference on Software and System Process*, 2012.

[34] M. A. C. Jiffriya, M. A. C. A. Jahan, and R. G. Ragel, "Plagiarism detection on electronic text based assignments using vector space model," in *7th International Conference on Information and Automation for Sustainability*, 2014, pp. 1–5.

[35] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017, p. 102–112.

[36] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. De Water, "Studying pull request merges: a case study of Shopify's active merchant," in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 124–133.

[37] H. C. Kraemer, G. A. Morgan, N. L. Leech, J. A. Gliner, J. J. Vaske, and R. J. Harmon, "Measures of clinical significance," *Journal of the American Academy of Child & Adolescent Psychiatry*, vol. 42, no. 12, pp. 1524–1529, 2003.

[38] W. H. Kruskal and W. A. Wallis, "Use of ranks in one-criterion variance analysis," *Journal of the American Statistical Association*, pp. 583–621, 1952.

[39] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, pp. 384–417, 2018.

[40] R. G. Kula, A. Ouni, D. M. German, and K. Inoue, "On the impact of micro-packages: An empirical study of the npm Javascript ecosystem," *arXiv:1709.04638*, 2017.

[41] I. Kwan, A. Schröter, and D. Damian, "A weighted congruence measure," in *2nd International Workshop on Socio-Technical Congruence*, 2009, pp. 1–10.

[42] I. Kwan, A. Schroter, and D. Damian, "Does socio-technical congruence have an effect on software build success? a study of coordination in a software project," *IEEE Transactions on Software Engineering*, pp. 307–324, 2011.

[43] Z. Li, Y. Yu, T. Wang, G. Yin, X. jun Mao, and H. Wang, "Detecting duplicate contributions in pull-based model combining textual and change similarities," *Journal of Computer Science and Technology*, vol. 36, pp. 191–206, 2021.

[44] W. Mauerer, M. Joblin, D. Tamburri, C. Paradis, R. Kazman, and S. Apel, "In search of socio-technical congruence: A large-scale longitudinal study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[45] E. Mirsaeedi and P. C. Rigby, "Mitigating turnover with code review recommendation: Balancing expertise, workload, and knowledge distribution," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, p. 1183–1195.

[46] M. Nassif and M. Robillard, "Revisiting turnover-induced knowledge loss in software projects," in *2017 IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 261–272.

[47] J. Portillo-Rodríguez, "An agent architecture with which to improve coordination and communication in global software engineering," *University of Castilla-La Mancha, Ciudad Real, Spain*, 2013.

[48] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering*, pp. 2464–2519, 2018.

[49] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to api deprecation? the case of a smalltalk ecosystem," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.

[50] G. Robles and J. M. Gonzalez-Barahona, "Contributor turnover in libre software projects," in *IFIP International Conference on Open Source Systems*. Springer, 2006, pp. 273–286.

[51] J. Romano, J. D. Kromrey, J. Coraggio, J. Skowronek, and L. Devine, "Exploring methods for evaluating group differences on the NSSE and other surveys: Are the t-test and Cohen's d indices the most appropriate choices?" in *Annual Meeting of the Southern Association for Institutional Research*, 2006, pp. 1–51.

[52] I. Samoladas, L. Angelis, and I. Stamelos, "Survival analysis on the duration of open source projects," *Information and Software Technology*, vol. 52, no. 9, p. 902–922, 2010.

[53] P. Schober, C. Boer, and L. Schwarte, "Correlation coefficients: Appropriate use and interpretation," *Anesthesia and Analgesia*, vol. 126, p. 1, 2018.

[54] J. Stringer, A. Tahir, K. Blincoe, and J. Dietrich, "Technical lag of dependencies in major package managers," in *Proceedings of the 27th Asia-Pacific Software Engineering Conference*, 2020, pp. 228–237.

[55] P. Thongtanunam, C. Tantithamthavorn, R. G. Kula, N. Yoshida, H. Iida, and K.-i. Matsumoto, "Who should review my code? a file location-based code-reviewer recommendation approach for modern code review," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering*, 2015, pp. 141–150.

[56] E. Ukkonen, "Approximate string-matching with q-grams and maximal matches," *Theoretical Computer Science*, pp. 191–211, 1992.

[57] G. Valetto, M. Helander, K. Ehrlich, S. Chulani, M. Wegman, and C. Williams, "Using software repositories to investigate socio-technical congruence in development projects," in *4th International Workshop on Mining Software Repositories*, 2007, pp. 25–25.

[58] M. Valiev, B. Vasilescu, and J. Herbsleb, "Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, p. 644–655.

[59] P. Wagstrom, J. D. Herbsleb, and K. M. Carley, "Communication, team performance, and the indiviual: Bridging technical dependencies," *Academy of Management Proceedings*, vol. 2010, no. 1, pp. 1–7, 2010.

[60] D. Wang, R. G. Kula, T. Ishio, and K. Matsumoto, "Automatic patch linkage detection in code review using textual content and file location features," *Information and Software Technology*, 2021.

[61] D. Wang, T. Xiao, P. Thongtanunam, R. G. Kula, and K. Matsumoto, "Understanding shared links and their intentions to meet information needs in modern code review," *Empirical Software Engineering*, vol. 26, no. 5, pp. 1–32, 2021.

[62] J. Wang, "Survival factors for free open source software projects: A multi-stage perspective," *European Management Journal*, vol. 30, no. 4, pp. 352–371, 2012.

[63] Y. Wang, B. Chen, K. Huang, B. Shi, C. jian Xu, X. Peng, Y. Liu, and Y. Wu, "An empirical study of usages, updates and risks of third-party libraries in Java projects," *2020 IEEE International Conference on Software Maintenance and Evolution*, pp. 35–45, 2020.

[64] Y. Win and T. Masada, "Exploring technical phrase frames from research paper titles," in *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, 2015, pp. 558–563.

[65] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull requests in GitHub: What can we learn from code review and bug assignment?" *Information and Software Technology*, pp. 204–218, 2016.

[66] R. Yue, N. Meng, and Q. Wang, "A characterization study of repeated bug fixes," in *2017 IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 422–432.

[67] R. E. Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm Javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution*, 2018, pp. 559–563.

[68] A. Zerouali, E. Constantinou, T. Mens, G. Robles, and J. Gonzalez-Barahona, "An empirical analysis of technical lag in npm package dependencies," in *New Opportunities for Software Reuse*, 2018, pp. 95–110.

[69] W. Zhang, S.-C. Cheung, Z. Chen, Y. Zhou, and B. Luo, "File-level socio-technical congruence and its relationship with bug proneness in OSS projects," *Journal of Systems and Software*, pp. 21–40, 2019.