Characterizing Deep Learning Package Supply Chains in PyPI: Domains, Clusters, and Disengagement

KAI GAO, School of Software & Microelectronics, Peking University, Beijing, China and Key Laboratory of High Confidence Software Technologies, Ministry of Education, , China
RUNZHI HE, BING XIE, and MINGHUI ZHOU, School of Computer Science, Peking University, Beijing, China and Key Laboratory of High Confidence Software Technologies, Ministry of Education, , China

Deep learning (DL) frameworks have become the cornerstone of the rapidly developing DL field. Through installation dependencies specified in the distribution metadata, numerous packages directly or transitively depend on DL frameworks, layer after layer, forming DL package supply chains (SCs), which are critical for DL frameworks to remain competitive. However, vital knowledge on how to nurture and sustain DL package SCs is still lacking. Achieving this knowledge may help DL frameworks formulate effective measures to strengthen their SCs to remain competitive and shed light on dependency issues and practices in the DL SC for researchers and practitioners. In this paper, we explore the domains, clusters, and disengagement of packages in two representative PyPI DL package SCs to bridge this knowledge gap. We analyze the metadata of nearly six million PyPI package distributions and construct version-sensitive SCs for two popular DL frameworks: TensorFlow and PyTorch. We find that popular packages (measured by the number of monthly downloads) in the two SCs cover 34 domains belonging to eight categories. *Applications*, *Infrastructure*, and *Sciences* categories account for over 85% of popular packages in either SC and TensorFlow and PyTorch SC have developed specializations on *Infrastructure* and *Applications* packages, respectively. We employ the Leiden community detection algorithm and detect 131 and 100 clusters in the two SCs. The clusters mainly exhibit four shapes: Arrow, Star, Tree, and Forest with increasing dependency complexity. Most clusters are Arrow or Star, while Tree and Forest clusters account for most packages (Tensorflow SC: 70.7%, PyTorch SC: 92.9%). We identify three groups of reasons why packages disengage from the SC (i.e., remove the DL framework and its dependents from their installation dependencies): dependency issues, functional improvements, and ease of installation. The most common reason in TensorFlow SC is dependency incompatibility and in PyTorch SC is to simplify functionalities and reduce installation size. Our study provides rich implications for DL framework vendors, researchers, and practitioners on the maintenance and dependency management practices of PyPI DL SCs.

CCS Concepts: • **Software and its engineering** → **Software evolution**; **Maintaining software**; • **Human-centered computing** → **Open source software;**

Additional Key Words and Phrases: Software supply chain, PyPI ecosystem, deep learning, software structure and evolution

This work is sponsored by the National Natural Science Foundation of China 61825201 and 62332001.

Authors' addresses: K. Gao, School of Software & Microelectronics, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, 100871, China and Key Laboratory of High Confidence Software Technologies, Ministry of Education, China; e-mail: gaokai19@pku.edu.cn; R. He, Bing Xie, and M. Zhou (Corresponding author), School of Computer Science, Peking University, No. 5 Yiheyuan Road, Haidian District, Beijing, 100871, China and Key Laboratory of High Confidence Software Technologies, Ministry of Education, China; e-mails: {rzhe, xiebing, zhmh}@pku.edu.cn.

ACM 1049-331X/2024/04-ART97
https://doi.org/10.1145/3640336

ACM Reference Format:
Kai Gao, Runzhi He, Bing Xie, and Minghui Zhou. 2024. Characterizing Deep Learning Package Supply Chains in PyPI: Domains, Clusters, and Disengagement. *ACM Trans. Softw. Eng. Methodol.* 33, 4, Article 97 (April 2024), 27 pages. https://doi.org/10.1145/3640336

1 INTRODUCTION

In the past decade, **deep learning (DL)** techniques have been widely adopted in diverse tasks such as face recognition [34], machine translation [22], and code generation [64] with exhilarating performance. The burst of DL techniques is indispensable from DL frameworks, which provide a collection of APIs to allow developers to design and train DL models more easily and quickly and have become the cornerstone of the DL field. Many DL frameworks have been launched by various organizations, among which TensorFlow [14] and PyTorch [11] are the most popular [13].

Research has revealed that developers have different needs when using DL frameworks to perform different tasks [37]. To satisfy developers' diverse needs, substantial packages have been released by DL enthusiasts in the Python community [28]. These packages provide a variety of specialized functionalities based on the APIs provided by DL frameworks directly or transitively. To allow package management tools like `pip` to automatically install DL frameworks at the installation time, DL frameworks are specified as direct or transitive *installation dependencies* in these packages' distribution metadata [6].[1] Gradually, starting from DL frameworks and through layer-by-layer installation dependencies, a large number of packages form DL package ***supply chains*** **(SCs)**.

Numerous DL frameworks have emerged as DL thrives. However, remaining competitive amidst rivals is not that easy for DL frameworks. Recent years have witnessed the stop of development of several DL frameworks such as CNTK [3] and Chainer [4] for their lack of competitiveness. Given the wide adoption of DL techniques in diverse tasks and the prevalence of reusing third-party packages [5], we argue that a prosperous DL package SC is critical to a competitive DL framework. First, packages in the SC provide a variety of off-the-shelf specialized functionalities for developers to use, which can help DL frameworks better meet developers' diverse needs [37]. Second, the number of dependent packages is perceived by practitioners as an indicator of the package's popularity and influences the selection of third-party packages [53]. In this sense, a prosperous SC is helpful for DL frameworks to attract users. Moreover, the significance of packages in the SC is acknowledged by DL frameworks, e.g., the PyTorch website has a dedicated page presenting packages depending on PyTorch [16]. Therefore, it is important for DL frameworks to nurture and sustain their package SCs.

Previous work has primarily studied SCs at the whole packaging ecosystem level rather than the specific field level like DL. These studies constructed SC in packaging ecosystems as a directed graph and designed metrics to measure its structure and evolution [31–33, 50, 55, 73]. However, these metrics are built mainly from the graph's perspective and can not well reflect SC's nature, such as the domain of packages in the SC. A few studies investigated code repositories that declare dependencies on DL frameworks in the manifest files (i.e., installation dependencies) [39] or in the import statements (i.e., *import dependencies*) [36, 65], e.g., application domains [39, 65], update behaviors [36, 39], and popularity factors [65]. However, a code repository may or may not be released as a package in **The Python Package Index (PyPI)** and the import dependency differs from the installation dependency, leaving the knowledge on how to nurture and sustain DL package SCs still fragmented and incomplete. The primary significance of achieving such knowledge is to help *DL framework vendors* formulate effective measures to strengthen their SCs to remain

[1] According to **Python Packaging Authority (PyPA)** [7], the distribution is a specific version of a package.

competitive (as argued above). Besides, such knowledge can also shed light on dependency management issues and practices in the DL SC for *researchers* and *DL practitioners*.

To make a step towards filling this knowledge gap, this paper investigates two representative DL package SCs in the PyPI packaging ecosystem, which start from TensorFlow and PyTorch respectively (TensorFlow SC and PyTorch SC for short). TensorFlow and PyTorch are the two most widely adopted DL frameworks [37] and PyPI is the official third-party package registry [9] for Python, the mainstream programming language in the DL field [28] and the main language for which TensorFlow and PyTorch provide APIs [15, 20]. Considering that a prosperous DL SC should help meet DL developers' diverse needs and requires the engagement of numerous packages, we formulate the following research questions:

RQ1 (Domains): *What domains do packages in TensorFlow SC and PyTorch SC cover?* The domain of a package refers to what kinds of tasks the package can solve. Investigating the domains of packages in the SC can pinpoint domains in which a DL SC should provide support to help meet developers' needs. Besides, it also helps us gain a better understanding of the SC's capability, e.g., its strengths, weaknesses, and common usage scenarios. Such understanding can help DL framework vendors strengthen their SCs to better meet developers' needs and help DL practitioners make more informed decisions when selecting DL framework (SC) for their tasks. Prior work has studied application domains of code repositories that directly depend on DL frameworks [39] and domains of packages in the DL repository SC based on import dependencies [65]. However, they cover only a small part of the DL package SC (more details in Section 3) and a more complete understanding of domain distributions in the DL package SC is still lacking.

RQ2 (Clusters): *What kinds of package clusters are formed in the two SCs?* DL package SC is a complex graph involving numerous interdependent packages, which makes its structure mysterious. Decomposing DL package SC into package clusters where packages are densely connected in the same cluster but sparsely connected between clusters, is a good means to understand how packages engage in the SC and identify critical packages that attract other packages to the SC, thus helping DL frameworks take effective actions to attract packages. For example, packages on which the other packages depend in each cluster significantly impact the SC's proper functioning and should be prioritized for security measures. Besides, packages in the same cluster may share similar functionality, thus package clusters can serve as a valuable source for practitioners to search for suitable packages in the SC.

RQ3 (Disengagement): *To what extent and why do packages disengage from the two SCs?* It is common for software to remove some dependencies during its evolution [41]. If a package completely removes dependencies on the DL framework and its dependents since a certain version, we refer to that package disengaging from the DL package SC. The disengagement of packages may affect the SC's sustainability and stability and lead to user churn, thus reducing the DL framework's competitiveness. Moreover, the disengagement of packages could also signal potential dependency management problems faced by package developers in the SC. Therefore, understanding why packages disengage from the SCs helps DL frameworks take countermeasures to keep packages engaged. However, not much is known about the prevalence and rationale for package disengagement in the DL package SC.

To answer the questions, we first build an ecosystem-wise and version-sensitive installation dependency database by analyzing the metadata of nearly six million PyPI package distributions. Based on the database, we construct TensorFlow and PyTorch SC by iteratively retrieving packages that declare direct or transitive installation dependency on TensorFlow or PyTorch. We conduct a thematic analysis on the description of popular packages (measured by the number of monthly downloads) in the two SCs and identify 34 package domains spanning eight categories. Over 85% of popular packages in either SC fall into *Applications*, *Infrastructure*, and *Sciences* categories.

Infrastructure and *Applications* categories account for the most popular packages in TensorFlow SC and PyTorch SC respectively, uncovering the two SCs' different specializations. We employ the Leiden community detection algorithm and detect 131 and 100 package clusters in the two SCs. The clusters mainly exhibit four shapes: Arrow, Star, Tree, and Forest, with increasing dependency complexity. Despite most clusters being Arrow or Star, Tree and Forest clusters account for the majority (70.7% and 92.9% respectively) of packages, indicating that a small number of packages attract most of the other packages to the SC. We find the number of packages that have disengaged from either SC shows an increasing trend. We identify seven disengagement reasons related to three aspects: dependency issues, functional improvements, and ease of installation. The most common disengagement reasons differ between the two SCs, suggesting different dependency management problems in them.

In a nutshell, this paper makes the following contributions:

— A comprehensive understanding of the domains, clusters, and disengagement of packages in two representative DL package SCs in the PyPI packaging ecosystem.
— An automated approach to construct version-sensitive PyPI package SCs and a public dataset and scripts[2] to facilitate future research.
— Rich suggestions for DL framework vendors, researchers, and DL practitioners on the maintenance and dependency management practices of PyPI DL package SCs.

2 BACKGROUND AND RELATED WORK

2.1 Background

Definition of Software SC. In traditional production, SC refers to the network of companies involved in the process of converting raw materials to final products delivered to customers, where supply relationships between companies are established by the flow of products [59]. Similar supply relationships exist between software [21]. Specifically, if a piece of software uses the functionality provided by another piece of software (e.g., package APIs), a supply relationship (i.e., dependency) is assumed to be established between them. As software development becomes increasingly dependent on external software, supply relationships among software become more and more complex, gradually forming a network of software connected by dependencies, i.e., software SC.

Code Repository vs. Package Distribution. Prior work [65] investigated code repositories that depend on DL frameworks and analyzed the domains of code repositories that release packages to PyPI by matching the repository's URL with the code repository URL declared in the PyPI package's distribution metadata. However, substantial (about 1/3) PyPI packages do not declare their code repository information in the distribution metadata.[3] Therefore, identifying PyPI packages from code repositories will omit considerable PyPI packages. In this paper, we choose **package distributions** published in PyPI since we can easily obtain a complete list of PyPI packages and dependency relationships between them, which allows us to construct as complete package SCs in PyPI as possible.

Installation Dependency vs. Import Dependency. Existing literature mainly investigates two kinds of dependencies: installation dependency and import dependency. Installation dependency is specified in the software's manifest files such as `setup.py`, `requirements.txt`, and package distribution metadata [72]. Import dependency is specified in the code files' import statements instead [65], e.g., `import torch`.

[2] https://github.com/gaokai320/PyPI-DLSC
[3] The script that calculates the ratio is provided in the replication package.

In this paper, we choose to construct DL package SCs by analyzing **installation dependencies** between package distributions for four reasons. First, installation dependency is the most widely used in prior work on analyzing package SCs (e.g., [33, 50, 73, 76]). Second, Python has provided an official package, `packaging` [19] which can accurately parse installation dependencies specified in the package distribution metadata. Third, it is challenging to accurately identify import dependencies (i.e., map the package's import name in the import statement to its package name) since a Python package's import name is not necessarily the same as its package name and different packages may share the same import name [38]. Fourth, installation dependency reserves dependency version information, which enables us to analyze packages' disengagement. However, installation dependency suffers from the bloated dependency issue [24], that is, some installation dependencies are not used by the software. To evaluate the impact of this issue, we manually check some packages, as detailed in Section 3.

Definition of PyPI DL Package SC. We conceptualize *PyPI package SC* as a directed graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$ where vertices in $\mathcal{P}$ are PyPI packages, edges in $\mathcal{E}$ represent installation dependencies between packages. Formally, installation dependency can be defined as an ordered pair $(P_u[u_i], P_d[d_j])$, indicating that to install version d_j of the package P_d, version u_i of the package P_u should be installed first. We refer to P_u as P_d's dependency and P_d as P_u's dependent. We refer *PyPI DL package SC* as a sub-graph of PyPI package SC which starts from a DL framework and includes all packages directly or transitively depending on the framework, as detailed in Section 3.

2.2 Related Work

Our work lies in the intersection between software SC and DL and hence has two groups of related work: research on SCs in packaging ecosystems and research on **software engineering (SE)** for DL.

2.2.1 Research on SCs in Packaging Ecosystems. Due to the prevalence of reusing third-party packages in software development [41], package SCs in various packaging ecosystems have been a popular topic in the SE community. Wittern et al. [73] investigated the evolution of NPM and found that the number of packages and dependency relationships among packages increased rapidly. Decan et al. [30, 31, 33] and Kikas et al. [50] analyzed the distribution of direct and transitive dependencies in package SCs in multiple packaging ecosystems. They also found growing packages and dependency relationships among packages in these SCs, and that transitive dependencies lead to the fragility of these SCs.

Substantial work is conducted to identify and mitigate risks in package SCs. Valiev et al. [68] found that PyPI packages' relative position in the SC has a significant impact on their sustainability. Vu et al. [70] investigated the differences between the code in PyPI packages and their corresponding code repositories to identify potentially malicious code injections. Liu et al. [55] investigated the vulnerability propagation and propagation evolution in NPM. Zahan et al. [78] proposed six security weakness indicators to identify potential SC attacks in NPM. Wang et al. investigated dependency conflicts introduced by transitive dependencies in PyPI [72] and NuGet [54].

Despite these efforts, knowledge of how to nurture and sustain DL package SCs is still limited.

2.2.2 Research on SE for DL. A large body of work has been devoted to SE for DL [37] with the data from GitHub, **Stack Overflow (SO)**, interviews, and so on. Specifically, DL bugs are extensively investigated. Zhang et al. [82] explored the symptoms, root causes, and fix challenges of bugs in TensorFlow applications. Islam et al. [46, 47] and Humbatova et al. [45] studied bugs in applications of several DL frameworks. Bugs associated with DL jobs [79], DL frameworks [48, 49, 77], DL dependency stack [44], training [81], and deployment [27] are also explored. Some research identified challenges for developing [37, 40, 80] and deploying [26] DL applications. This line of work provides insights for the SE community to tackle DL problems but none touches the problem from SC's perspective.

Little work investigated the characteristics of code repositories depending on DL frameworks. Han et al. [39] studied the project purposes, application domains, update behaviors, and dependency version distribution of code repositories that declare DL frameworks as dependencies in the manifest files. Dilhara et al. [36] investigated the usage and update of DL frameworks in code repositories that import them in the code. The two studies suffer from two major limitations. First, they only focused on code repositories that directly depend on the DL frameworks and ignored the network structure of SC. Second, the code repositories they collected may or may not be released as PyPI packages. Therefore, the two studies cover only a small part of the DL package SC. Tan et al. [65] constructed TensorFlow and PyTorch repository SCs by repeatedly retrieving code repositories that directly or transitively import TensorFlow or PyTorch. They located code repositories released as PyPI packages in the SC and investigated the two SCs' structure, application domains, and popularity factors. However, their SCs differ a lot from package SCs for two reasons. First, their SCs miss substantial PyPI packages as discussed in Section 2.1. About 70% of packages in our SCs are not covered by theirs (more details in Section 3). Second, import dependencies between the packages' code repositories may over-represent installation dependencies between package distributions. Specifically, not every file (e.g., test files and example files) in the repository will be packaged into the distribution, so import dependencies declared in these files are not installation dependencies of the distribution. A manual inspection of 100 packages in the SCs of [65] indicates that the import dependencies declared in the code repository of 27 packages are not installation dependencies of their distributions. The two differences lead to the differences in the findings of domain distribution between their study and ours, as discussed in Section 4.2.

To conclude, earlier studies cover only a small part of the PyPI DL package SC and their results can not sufficiently reveal how to nurture and sustain PyPI DL package SCs.[4]

3 SUPPLY CHAIN CONSTRUCTION

The **Python Packaging Authority (PyPA)**, a working group responsible for Python packaging, hosts a public dataset on BigQuery containing the metadata of all distributions released on PyPI [10], which enables us to construct as complete PyPI DL package SCs as possible. We download the data dump on Nov 4, 2021, which consists of 354,636 packages and 5,743,721 distributions.

We first build a dependency database $\mathcal{D}$ to enable the construction of PyPI DL SCs. The database is ecosystem-wise since it covers all PyPI packages. Compared with the approaches proposed in prior work [30, 68] on analyzing PyPI package SCs, it also takes the dependency's version information into account to obtain dependency relationships between PyPI package releases (i.e., version-sensitive), which enables us to construct finer-grained DL package SCs and investigate the package's disengagement. Specifically, the PyPI package distribution specifies its installation dependencies in the `requires_dist` field in its metadata. Each item in the field contains a package name P_u and an optional version constraint C. A version constraint specifies a **Python Enhancement Proposal (PEP)** 508 version range [2] for a dependency. We implement a version constraint parser on top of the `packaging` [19] package, a tool released by PyPA to parse dependency specifications, to locate satisfying dependency versions. Specifically, for each version u_i of P_u released in PyPI (by querying the PyPI distribution metadata dump), we check if it satisfies C. If satisfied, we insert a record $(P_u[u_i], P_d[d_j])$ to $\mathcal{D}$. Finally, $\mathcal{D}$ contains 281,443,215 records.

[4] In the following sections, if not specified, "SC" refers to PyPI package SC, and "dependency" refers to installation dependency

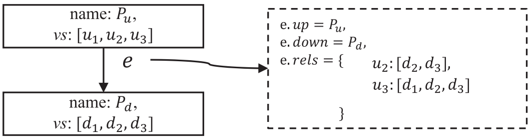

Fig. 1. An example of SC.

ALGORITHM 1: Constructing PyPI DL SC

```
Input: A list of package names: $\mathcal{N}$
Output: A supply chain: $\mathcal{G}$
$\mathcal{P} \leftarrow Vertices(), \mathcal{E} \leftarrow Edges(), unvisited \leftarrow Vertices()$
// Input packages as initial unvisited packages
foreach $n \in \mathcal{N}$ do
    $p \leftarrow Vertex(name = n, vs = list())$
    $p.update\_versions(GetAllPyPIVersions(n))$
    $\mathcal{P}.merge(p)$
    $unvisited.merge(p)$
while $unvisited \neq \emptyset$ do
    $dependents \leftarrow Vertices()$
    foreach $u \in unvisited$ do
        foreach $version \in u.vs$ do
            $dps \leftarrow GetDependents(u.name, version)$
            $dependents.merge(dps)$
            foreach $d \in dps$ do
                // Merge new dependencies to $\mathcal{E}$
                $e \leftarrow Edge(up = u.name, down = d.name, rels = \{version : d.vs\})$
                $\mathcal{E}.merge(e)$
    // Obtain new $unvisited$ packages and versions
    $unvisited \leftarrow dependents - \mathcal{P}$
    // Merge newly found packages and versions to $\mathcal{P}$
    $\mathcal{P} \leftarrow \mathcal{P}.merge(dependents)$
$\mathcal{G} \leftarrow DiGraph(\mathcal{P}, \mathcal{E})$
return $\mathcal{G}$
```

Based on $\mathcal{D}$, we construct PyPI DL SCs as follows. To be version-sensitive, we add attributes to packages and dependency relationships in the SC. For each package, we maintain an attribute vs which is a list storing all the package's <u>v</u>ersion<u>s</u> appearing in the SC. For each dependency relationship, we maintain an attribute $rels$ which uses a map to store the version dependency <u>rel</u>ation<u>s</u> between package P_u and its dependent P_d. Each key in $rels$ represents a version of P_u and the value represents all versions of P_d that depend on the version of P_u corresponding to the key. Take Figure 1 as an example, suppose there is a package P_u and a package P_d depending on P_u in the SC. u_1, u_2, u_3 versions of P_u and d_1, d_2, d_3 versions of P_d appear in the SC as indicated by the vs attribute. The $rels$ attribute of dependency e between P_u and P_d shows that P_d only depends on the u_2 and u_3 versions of P_u. Specifically, the d_2 and d_3 versions of P_d depend on the u_2 version of P_u and all versions of P_d depend on the u_3 version of P_u.

Based on the definition in Section 2.1, we iteratively construct version-sensitive PyPI DL SC following the procedure described in Algorithm 1. Considering that a DL framework may release multiple packages, e.g., TensorFlow framework releases three packages: `tensorflow`,

`tensorflow-cpu`, and `tensorflow-gpu`, the algorithm takes a list of package names $\mathcal{N}$ as input. *unvisited* is a queue of unexplored packages. Our algorithm retrieves direct dependents of packages in *unvisited* and updates *unvisited* with unexplored direct dependents in each iteration. *unvisited* is initialized as all input packages with all their released versions in PyPI (lines 2~6). For each version of each package in *unvisited*, we query all versions of packages that directly depend on it from $\mathcal{D}$ (lines 9~12) and merge dependencies between them to $\mathcal{E}$ (lines 13~15). The merge takes two forms: adding new edges, or updating *rels* of existing edges (adding new key-value pairs or updating existing values). *dependents* denotes all versions of packages that directly depend on packages in *unvisited*. We obtain new *unvisited* packages by comparing *dependents* with $\mathcal{P}$ (line 16) and merge newly found packages (*dependents*) to $\mathcal{P}$ (line 17). The merge operation either adds new packages or adds new versions to *vs* of existing packages. The above process is repeated until *unvisited* is empty (line 7).

We construct TensorFlow and PyTorch SC with the input of `[tensorflow, tensorflow-cpu, tensorflow-gpu]` and `[torch]`, respectively. Finally, TensorFlow SC contains 2,567 packages, 19,116 versions, and 3,232 edges; PyTorch SC contains 3,278 packages, 26,563 versions, and 5,440 edges. We also construct TensorFlow SC and PyTorch SC by Nov. 2019 to make a comparison with [65]. There are 1,087 unique packages in the two SCs and 772 (71.0%) are not covered by [65]. We investigate the monthly downloads and the number of dependents for the uncovered packages and find that: 1) 185 packages have above-average monthly downloads, accounting for 79.4% of (233) packages with above-average monthly downloads; 2) 85 packages have dependents, such as `torchvision` with 130 dependents, `torchtext` with 21 dependents, `pytorch-transformers` with 13 dependents, and `tensorflow-probability` with 12 dependents. Therefore, our SCs cover more popular and critical packages than [65] and provide a more complete picture of DL package SCs in PyPI.

As discussed in Section 2.1, the package may not use its installation dependencies (i.e., bloated dependency), affecting the accuracy of the constructed DL package SC. To evaluate the prevalence of this issue, we randomly select 334 and 344 packages from TensorFlow SC and PyTorch SC, respectively (95% confidence level, 5% confidence interval).[5] For each sampled package, we download one distribution of its latest version appearing in the SC and check if this distribution imports its installation dependencies in the SC. Among the sampled packages, 7 (out of 334) and 6 (out of 344) packages have been removed from PyPI at the time of inspection. Among the remaining 327 and 338 packages, 305 (93.3%) and 334 (98.8%) packages import their installation dependencies, suggesting that the bloated dependency issue is rare in the two SCs we construct. Besides, it seems that the bloated dependency issue is more common in TensorFlow SC than in PyTorch SC, which coincides with the finding in Section 6.2.

4 RQ1: WHAT DOMAINS DO PACKAGES IN TENSORFLOW SC AND PYTORCH SC COVER?

4.1 Method

While manual checking is a feasible approach to identifying the package's domain, it is almost unaffordable to go through all packages due to the large number. We, therefore, choose to focus on popular packages in the two SCs measured by the number of monthly downloads, a widely used popularity metric [35, 69, 70, 76], for three reasons. First, intuitively, a higher number of monthly downloads indicates that the package or the package's dependents are more used by developers, which can better reflect the SC's core capabilities. Second, prior work [65] also selected popular

[5] https://www.surveysystem.com/sscalce.htm

packages to analyze domain distribution. Third, many packages are rarely used, e.g., toy or dummy packages [65, 71], which can not well reflect the SC's core capabilities. Considering newly created packages, we retrieve the number of downloads of all PyPI packages from the public PyPI download statistics dataset one month after we obtain the distribution metadata dump, i.e., between Nov. 4, 2021, and Dec. 4, 2021. Then, we select packages in the two SCs with over 3,825 monthly downloads, the average monthly downloads of PyPI packages. We use the average monthly downloads as the selection metric to balance the representativeness of selected packages and the time costs of conducting the manual inspection. In this way, we obtain 219 and 259 packages in TensorFlow and PyTorch SC respectively, which are feasible for manual labeling and are higher than the number of packages investigated in prior work [65]. Perhaps not surprisingly, the popular packages in the two SCs are not disjoint with 41 overlapped packages, leaving 437 unique packages.

We follow the general thematic analysis [29] procedure to determine package domains. First, two inspectors (the first and second author) inspect each package's summary, description, and repository README to collect relevant texts about their domains. Then, they independently read and re-read all materials, generate initial codes describing each package's domain, and merge similar codes into themes. If a package can not be labeled due to the lack of information on its domain, it is marked as *Unclear*. Next, they compare the lists of codes and themes until an agreement is reached and develop a coding guide with definitions and examples for each identified code. In the process, we refer to the topics listed in the call for paper of NeurIPS,[6] one of the top-tier conferences in the DL field to ensure the rationality of generated domains and categories. After that, the inspectors independently use the coding guide to label all materials, and the inter-rater reliability measured by Cohen's Kappa [58] is 0.953. The disagreements are resolved by a discussion. Finally, 201 packages in TensorFlow SC and 247 packages in PyTorch SC are properly labeled. 28 packages are marked as *Unclear* and are excluded in the following discussion. The entire process takes about 200 man-hours and related materials are provided in the replication package.

4.2 Results

Figure 2 presents the domain distribution covered by the popular packages in either SC. In total, we identify 34 package domains under eight categories.

Applications category includes nine domains that have long been studied in computer science, such as **natural language processing (NLP)** and **computer vision (CV)**. Packages in this category provide functionalities that facilitate the application of DL techniques on specific types of data (e.g., images, text, and audio), such as data preprocessing and model architectures. This category accounts for the highest (52.6%) number of popular packages in PyTorch SC and the second-highest (36.3%) in TensorFlow SC, among which *NLP* and *CV* are predominant. We also notice differences between the two SCs. Generally, the proportion of *Applications* packages in PyTorch SC is significantly higher than that in TensorFlow SC, as confirmed by the one-sided proportion z-test ($p < 0.001$). It is known that PyTorch offers better programmabilities than TensorFlow, and our finding further suggests that PyTorch SC has developed competitive advantages in *Applications* domains with many popular packages. Specifically, PyTorch SC provides a higher percentage of packages on *NLP*, *CV*, *Time Series*, and *Tabular Prediction*, while TensorFlow SC provides a higher percentage of ***IR*** (**Information Retrieval**) packages. Besides, the number of *NLP* packages is slightly less than the number of *CV* packages in PyTorch SC, but in TensorFlow SC, the former is twice as many as the latter. Such differences may be attributed to the availability of related tools. For example, `torchvision` is the most popular package in PyTorch SC. It provides *popular datasets, model architectures, and common image transformations for computer vision* [12] and is dependent

[6] https://neurips.cc/Conferences/2023/CallForPapers

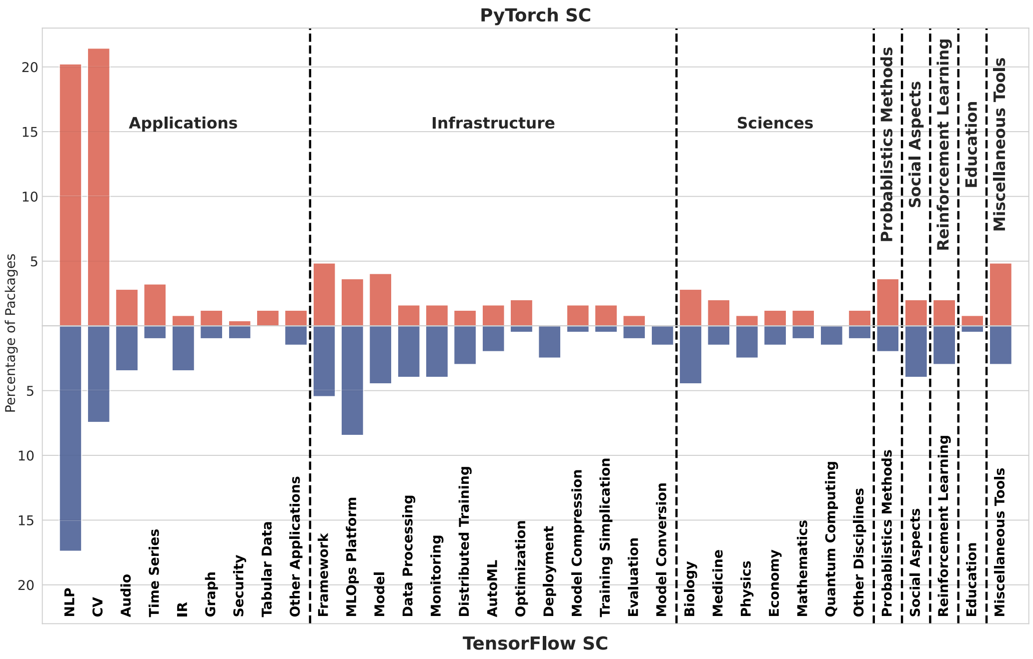

Fig. 2. Distribution of the 34 package domains and eight categories. The upper part (red bars) corresponds to PyTorch SC and the lower part (blue bars) corresponds to TensorFlow SC.

by 943 packages in PyTorch SC. That is, `torchvision` lays the foundation for the prosperity of *CV* packages in PyTorch SC. But similar *CV* packages are missing in TensorFlow SC.

Infrastructure is the second-largest (24.7%) category in PyTorch SC and the largest (37.8%) category in TensorFlow SC. Packages in this category provide functionalities applicable to multiple domains in other categories. It includes 13 domains. The distribution of domains in this category differs greatly between the two SCs. First, TensorFlow SC provides a significantly higher percentage of popular *Infrastructure* packages than PyTorch SC ($p < 0.01$), suggesting the competitive advantage of TensorFlow SC in *Infrastructure* domains. Second, a notable difference is that TensorFlow SC has a much higher percentage of popular *MLOps Platform* packages. These packages are released by various production platform vendors such as Azure Machine Learning[7] and TensorFlow Extended[8] and provide functionalities that facilitate the development, deployment, and management of DL models in these platforms. It is recognized that TensorFlow offers more features for productionization, and our finding further confirms the prosperity of productionization tools in TensorFlow SC. Third, the proportion of *Data Preprocessing*, *Monitoring*, *Distributed Training*, *Deployment*, and *Model Conversion* packages in TensorFlow SC is higher than that in PyTorch SC but PyTorch SC provides a higher percentage of *Optimization*, *Model Compression*, and *Training Simplification* packages. Such differences not only indicate the abundance of tool support in the SC but also reflect the functional deficiencies of the framework. For example, it is acknowledged that TensorFlow SC provides a collection of powerful deployment tools, e.g., TensorFlow Serving and TensorFlow Lite [26]. However, TensorFlow does not offer native support for ONNX [8], an open standard format of DL models that powers interoperability among frameworks, while PyTorch

[7] https://azure.microsoft.com/en-us/products/machine-learning
[8] https://www.tensorflow.org/tfx

does. As a result, three popular *Model Conversion* packages in TensorFlow SC emerge to enable the conversion between TensorFlow and ONNX models. Another example is the *Training Simplification* packages in PyTorch SC. Despite the convenience of building complex models with PyTorch, developers have to write a lot of boilerplate code to train DL models. To address these issues, many *Training Simplification* packages like `pytorch-ignite` are proposed to simplify intricate training configurations. Overall, these differences suggest that TensorFlow SC provides stronger capabilities in productionizing DL models.

Amazingly, *Sciences* category has the third most packages (13.4% in TensorFlow SC and 9.3% in PyTorch SC). Packages in this category facilitate the application of DL techniques in many scientific disciplines, especially *Biology*, *Medicine*, and *Physics*. Next, *Probabilistic Methods*, *Reinforcement Learning*, and *Social Aspects* take their turn. *Probabilistic Method* packages provide various Bayesian inference methods and Bayesian optimization models, especially the Gaussian Process. PyTorch SC provides a higher percentage (3.6%) of related packages than TensorFlow SC (2.0%). ***Reinforcement Learning* (RL)** packages provide various RL environments and models. 3.0% of the popular packages in TensorFlow SC and 2.0% in PyTorch SC fall into this domain. *Social Aspects* is an emerging hot research topic aiming to reduce the adverse effects such as discrimination and unreliable decisions brought by DL. It involves many aspects such as robustness, explainability, fairness, and privacy. It seems that TensorFlow SC provides more (4.0%) widely used packages on *Social Aspects* than PyTorch SC (2.0%), possibly due to that social aspects are of great concern when productionizing DL models and TensorFlow SC is more used to deploy DL model into production environments. After that, 3.0% and 4.9% of the popular packages in TensorFlow SC and PyTorch SC are *Miscellaneous Tools*, which provide commonly used utilities and wrappers. Finally, we notice three popular packages in the two SCs are supporting materials for DL books and courses that teach developers DL techniques, e.g., `nlpia`, a package containing code examples for the *Natural Language Processing in Action* book.

We also make a comparison with the findings reported by [65], which constructed DL SCs based on import dependencies between code repositories and analyzed application domains of repositories that have released PyPI packages. For comparison, we operate as follows: (1) keep packages released before Nov. 2019, the time of their data collection; (2) map *Applications*, *Sciences*, *Probablistic Methods*, and *RL* packages to **domain-related (DR)** packages and the rest to **non-domain related (NDR)** packages, based on their coding guide of package domains. Our study confirmed some findings by [65], e.g., TensorFlow SC provides more *RL* packages; *CV* and *NLP* are the two most popular DR package types. However, there are also noteworthy differences. First, different from their finding that the numbers of DR and NDR packages are almost the same in either SC, we find that the number of DR packages is about 1.6 times of NDR packages in PyTorch SC, suggesting different specializations of the two SCs. Second, they found that there are more *CV* packages than *NLP* packages in either SC. In contrast, we find that the number of *NLP* packages is comparable and about two times that of *CV* packages in PyTorch SC and TensorFlow SC, indicating a relatively greater focus on NLP in either SC. Third, instead of more experiment result analysis packages provided by TensorFlow SC, we find that PyTorch SC provides a higher proportion of *Monitoring* packages, possibly because experiment result analysis is very important whereas PyTorch does not provide native support as TensorFlow does (e.g., the *tensorboard* visualization tool). The differences can be ascribed to the differences between repository SCs and package SCs discussed in Section 2.2.2.

Among the 41 overlapped packages, 18 are about *Applications* and 12 are about *Infrastructure*. These overlapped packages depend on packages from both SCs, either providing functionalities to support both DL frameworks or using packages from one SC to complement the other. For example, `tempeh` is a framework to track the model's memory usage and run time. It depends on both TensorFlow and PyTorch to be able to support models written with TensorFlow and PyTorch. As an example for the latter case, memcnn is a framework built upon PyTorch and uses tensorboard from TensorFlow SC for logging since PyTorch SC did not provide related tools at that time.

Summary for RQ1: The popular packages measured by the number of monthly downloads in the two SCs cover 34 domains belonging to eight categories. Over 85% of packages in either SC fall into *Applications*, *Infrastructure*, and *Sciences* categories. PyTorch SC provides a higher percentage of *Applications* packages such as *NLP* and *CV*. TensorFlow SC favors *Infrastructure* packages such as *MLOps Platform* and *Deployment*, suggesting that TensorFlow SC and PyTorch SC have developed competitive advantages in *Applications* and *Infrastructure* domains, respectively.

5 RQ2: WHAT KINDS OF PACKAGE CLUSTERS ARE FORMED IN THE TWO SCS?

5.1 Method

Conceptually, SC is a directed graph, enabling us to employ graph community detection techniques to decompose SC's complex structure into package clusters. First, in this RQ, we want to evaluate the impact of the derived packages in the SC. Therefore, the root nodes (i.e., TensorFlow and PyTorch framework) and their connecting edges are removed from further analysis, leaving us with 892 nodes with 938 edges in the TensorFlow SC, and 2,044 nodes with 2,661 edges in the PyTorch SC. A considerable number of nodes (65.2% in TensorFlow SC and 37.6% in PyTorch SC) are isolated points, suggesting that their only dependency in the SC is the DL framework. Next, we run the Leiden community detection algorithm on the pruned SCs. This algorithm is chosen for its efficiency, stability, and capability of forming dense clusters, which captures the structural characteristics of SC well [67].

We aim to understand how packages engage in the SC and identify critical packages that attract other packages to the SC. To that end, we focus on the shape and complexity (measured by two widely used metrics [30, 33]: *cluster size*, i.e., the number of nodes, and *average degree*, i.e., the number of edges over the number of nodes) of the cluster. They demonstrate the distribution of the packages in the whole picture, which provides a basic understanding of the SC's characteristics. Specifically, the cluster's shape describes how packages within the cluster are interdependent overall, and the average degree of the cluster indicates the average number of dependency relationships of packages in the cluster. They can well reflect how packages engage in the SC through dependencies. The cluster size refers to the number of packages in the cluster and helps to identify packages that increase package engagement in the SC.

To categorize clusters by shapes, we first visualize all clusters as figures[9] with igraph, a package providing common graph analysis tools [17]. Next, the same inspectors as Section 4.1 conduct open coding [63] to determine cluster shapes. Specifically, they observe how packages are interdependent in the clusters' visualization figures repeatedly. Then one inspector assigns each cluster to an initial code that describes the dependency structure among packages in the cluster and discusses with the other inspector to iteratively refine the codes until reaching an agreement. After that, they together develop a coding guide. To verify the reliability, they conduct independent coding on all clusters with the coding guide and the Cohen's Kappa value is 0.907. The conflicts are resolved by a meeting. The above process takes about 150 man-hours and related materials are provided in the replication package.

[9] Please refer to the figures/TensorFlow folder and figures/PyTorch folder in the replication package.

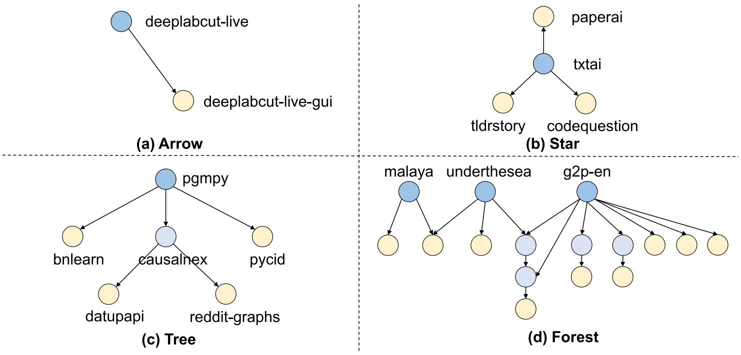

Fig. 3. Example of the four cluster shapes.

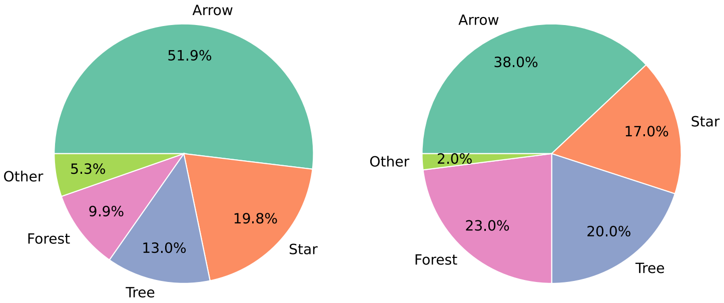

Fig. 4. Distribution of cluster shapes in TensorFlow SC (left) and PyTorch SC (right).

5.2 Results

Cluster Shape. The Leiden algorithm detects 131 and 100 package clusters in the pruned TensorFlow and PyTorch SC, respectively. The clusters mainly exhibit four shapes: **Arrow**, **Star**, **Tree**, and **Forest**. Figure 3 illustrates the four shapes and Figure 4 presents the proportion of each cluster shape in the two SCs.

- **Arrow** cluster contains only two packages with one depending on the other. For example, in an Arrow cluster of TensorFlow SC, package `deeplabcut-live-gui` depends on package `deeplabcut-live` and provides GUI for it (Figure 3(a)). 68 (51.9%) and 38 (38.0%) clusters in the two SCs are in this shape and take 15.2% and 3.7% of packages, respectively. Notably, packages in these clusters do not depend on packages in other clusters.
- **Star** cluster presents a centralized shape that contains at least three packages with a package dependent by the remaining packages. In Figure 3(b), the package `txtai` is directly dependent by the remaining three packages. 26 (19.8%) and 17 (17.0%) clusters are in this shape and account for 12.1% and 3.3% of packages in the two SCs. Most (23 and 14) Star clusters contain no more than five packages.
- **Tree** cluster presents a hierarchy shape with a *root package* on which the other packages in the cluster depend directly or transitively. Compared with Star clusters, transitive dependencies exist among packages in Tree clusters. In Figure 3(c), `pgmpy` is the cluster's root package that is directly or transitively dependent by the remaining five packages. 17 (13.0%) and 20 (20.0%) clusters are in this shape and take 26.8% and 38.6% of packages, respectively. Specifically, the largest cluster in either SC is in this shape.

- **Forest** cluster shows a more complex hierarchy shape with more than one root package on which the other packages in the cluster depend. Figure 3(d) shows a Forest cluster in TensorFlow SC. This cluster contains three root packages, i.e., `malaya`, `underthesea`, and `g2p-en`, dependent by the remaining 13 packages in the cluster. 13 (9.9%) and 23 (23.0%) clusters are in this shape and take 43.9% and 54.3% of packages, respectively. Over 75% of Forest clusters contain more than 10 packages in the two SCs.

In addition to the above four shapes, very few (17 and 2) packages form clusters in **Other** Shape. Specifically, three clusters in TensorFlow SC and two clusters in PyTorch SC contain only one package whose later version depends on their early version. One cluster in TensorFlow SC contains two packages that depend on each other. Three clusters in TensorFlow SC contain a package directly depending on the rest packages. Since the number of other clusters is small and they contain very few packages, in the following, we focus on the four main shapes.

Cluster Complexity. To understand the complexity of different cluster shapes, we compare their sizes and average degrees. Firstly we apply the Mann–Whitney U test [57] to compare the sizes of Arrow, Star, Tree, and Forest clusters. Since we perform multiple tests, we use the Holm–Bonferroni method to adjust the p-values to control the family-wise error rate [43]. The results suggest that in the two SCs Forest clusters contain more packages than Tree clusters (adjusted p-value: 0.009 and 1.40e-4) and Star clusters contain more packages than Arrow clusters (adjusted p-value: 6.57e-21 and 1.32e-12). However, the sizes of Tree clusters and Star clusters are not significantly different (adjusted p-value: 0.188 and 0.112). Figure 5 presents the distribution of the average degree for different shapes of clusters (each data point represents the average degree of a cluster). We can observe that the average degree of Arrow, Star, Tree, and Forest clusters increases gradually with the median as 0.50, 0.71, 1.00, 1.08 for TensorFlow SC and 0.50, 0.67, 1.00, 1.08 for PyTorch SC. The Mann–Whitney U test with Holm–Bonferroni method also suggests a significant difference. Particularly, the average degree for most Forest clusters (11 of 13 and 20 of 23) is no less than one, indicating that on average a package in the Forest cluster has dependency relations with at least two packages. We also investigate the depth of Tree and Forest clusters considering their complexity. The cluster's depth is defined as the length of the longest path between root packages and other packages in the cluster and reflects the ability of root packages to attract transitive dependents. Overall, the average depth of Forest clusters is higher than Tree clusters in either SC (TensorFlow SC: 2.46 vs. 2.06, PyTorch SC: 2.39 vs. 2.30), suggesting that root packages in Forest clusters are more likely to attract transitive dependents. To conclude, Arrow, Star, Tree, and Forest clusters show increasingly complex dependency relations. The proportion of Tree and Forest clusters and the proportion of packages in Tree and Forest clusters in PyTorch SC (43.0%, 92.9%) is much higher than that in TensorFlow SC (22.9%, 70.7%). It suggests that PyTorch SC has more complex dependency relations among packages and has a more centralized structure. Two reasons may account for this. First, PyTorch SC has a higher percentage (429, 13.1%) of packages with dependents than TensorFlow SC (276, 10.8%). Second, packages in PyTorch SC tend to depend on more packages than in TensorFlow SC, as suggested by the Mann-Whitney U test (p-value: 2.17e-99).

To gain more insights into SC's structural characteristics, we further inspect large clusters with more than 10 packages in each SC, resulting in 15 clusters in TensorFlow SC and 20 clusters in PyTorch SC. We find 8 (of 15) and 18 (of 20) large clusters in the shape of Forest in TensorFlow SC and PyTorch SC, respectively. We also inspect the package with the most dependents in these large clusters (core packages for short). One notable thing is that the core packages for 1/3 of large clusters in TensorFlow SC and 1/4 of large clusters in PyTorch SC are official packages, i.e., packages released by the DL framework team. It suggests that official packages play an important role in forming the SC. We also find that these core packages can well reflect the differences in the domain distribution of the two SCs revealed in RQ1. First, 5 (of the 20) core packages in PyTorch SC are

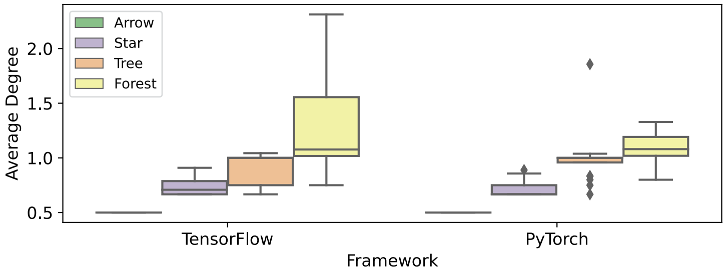

Fig. 5. Distribution of the average degree.

about *CV* but none of the 15 core packages in TensorFlow SC are about *CV*, which coincides with the huge difference in the proportion of *CV* packages in the two SCs. Second, 40% of the 15 core packages in TensorFlow SC fall into the *Infrastructure* category, and the percentage in PyTorch SC is 25%, which is consistent with the higher percentage of *Infrastructure* packages in TensorFlow SC.

Summary for RQ2: The Leiden algorithm detects 131 and 100 clusters in TensorFlow and PyTorch SC, respectively. The clusters mainly present four shapes: Arrow, Star, Tree, and Forest, with increasing dependency complexity. *Tree* and *Forest* clusters account for about 22.9% and 43.0% of clusters but contain the majority (70.7% and 92.9%) of packages, indicating that a small number of packages attract most packages to the SC. The higher percentage of Tree and Forest clusters in PyTorch SC suggests that PyTorch SC has more complex dependency relationships among packages than TensorFlow SC. The core packages of large clusters tend to be official packages in either SC and demonstrate the (targeted) specialty of each SC.

6 RQ3: TO WHAT EXTENT AND WHY DO PACKAGES DISENGAGE FROM THE TWO SCS?

6.1 Method

We consider a package to have disengaged from a PyPI DL SC if its latest version in the SC (V_{sc}) is earlier than its latest version in PyPI (V_{pypi}). In other words, it removes the DL framework and its dependents from installation dependencies after version V_{sc}. In this way, we find that 364 and 201 packages have disengaged from TensorFlow and PyTorch SC, respectively. We identify disengagement reasons from the package's code repository since the code repository documents rich development activity data (e.g., commits, issues, pull requests, and release notes [74, 75]) that have been used to identify reasons for various software development scenarios in prior work, e.g., library migration [41] and Dependabot deprecation [42]. Specifically, we take three steps as follows:

❶ *Locate packages' code repository.* For each package that has disengaged from the SC, we first identify its code repository from the repository link on its PyPI page following prior work [61, 69, 70]. If the link is broken or not present, we Google its name and description for its code repository. About 75% of the packages are linked to their repositories in either SC (TensorFlow SC: 275/364, PyTorch SC: 150/201). The ratio is comparable with previous work [68].

❷ *Extract disengagement-relevant text.* For each located code repository, we first identify the commit that removes the DL framework and its dependents from the dependency specification files such as `setup.py` and `requirements.txt`. If the commit message justifies the disengagement (i.e., removal of dependencies), we record relevant texts. Otherwise, we go through the issues, pull requests, and release notes that contain the name of the dependency and pick relevant texts. Finally,

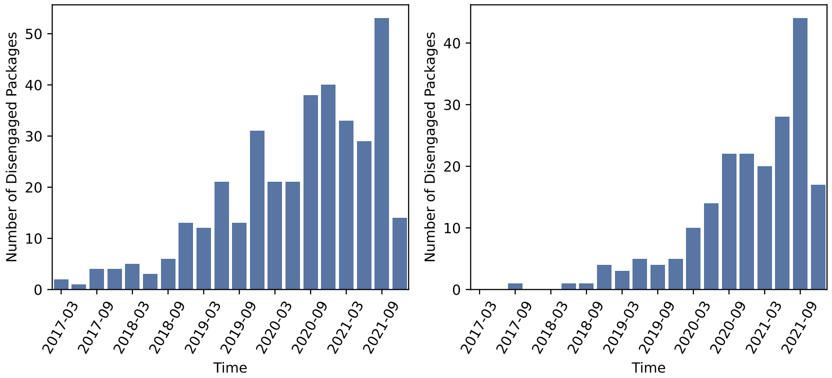

Fig. 6. Number of disengaged packages each quarter in TensorFlow SC (left) and PyTorch SC (right).

we extract disengagement-relevant texts for about 30% of packages in the two SCs (TensorFlow SC: 83/275, PyTorch SC: 44/150). The ratio is low because developers do not document the rationale behind their work, as revealed in prior work [66]. Despite this, the ratio is comparable with prior work that identified migration reasons [41] and Dependabot deprecation reasons [42] from code repositories.

❸ *Conduct thematic analysis.* The first two authors conduct a thematic analysis on the extracted texts to mine reasons for package disengagement. Each inspector independently reads and re-reads the extracted texts, generates initial codes describing disengagement reasons, and merges similar codes into themes. Then they discuss the generated codes and themes until they reach an agreement and develop a coding guide. Next, each inspector independently uses the coding guide to code all extracted texts. Note that multiple codes can be assigned to one text fragment. In this case, we use MASI distance [60] to measure the distance between the two inspectors' codes and Krippendorff's alpha [51] to measure inter-rater reliability. Krippendorff's alpha is 0.927, which is higher than the recommended threshold of 0.8 [52]. A meeting is held to resolve conflicts and to assign the final codes. The process takes about 200 man-hours and related materials can be found in the replication package.

6.2 Results

Figure 6 shows the trend of disengaged packages in the two SCs. We can see that the number of packages that disengaged from either SC each quarter shows an increasing trend overall (note that we only have partial data for the last quarter of 2021). In the 3rd quarter of 2021, both TensorFlow and PyTorch SC experienced a sharp increase in disengaged packages. These findings indicate that TensorFlow and PyTorch SC both suffer from increasingly severe package disengagement issues.

We identify seven disengagement reasons related to three aspects from the 83 and 44 disengaged packages in TensorFlow and PyTorch SC, respectively. Table 1 shows the number and proportion of disengaged packages mentioning each reason in either SC. Note that the sum of the proportions exceeds 100% because a package may disengage from the SC for multiple reasons. In the remainder of this section, we will elaborate on each reason.

Dependency. 38 (45.78%) and 13 (29.55%) package disengagements in TensorFlow and PyTorch SC are due to dependency issues. This category consists of two reasons: the removed dependency causes incompatibility or is bloated.

❖ *Incompatibility.* Some packages disengage from the SC because their dependencies induce incompatibility *issues or errors* with other required packages, hardware, or operating systems. It is the

Table 1. Reasons for Package Disengagement

Reason	TensorFlow SC[1]	PyTorch SC[2]
Dependency	38 (45.78%)	13 (29.55%)
Incompatibility	23 (27.71%)	10 (22.73%)
Bloated	15 (18.07%)	4 (9.09%)
Functionality	27 (32.53%)	24 (54.55%)
Performance	16 (19.28%)	7 (15.91%)
Simplification	9 (10.84%)	13 (29.55%)
Framework-Independent	3 (3.61%)	5 (11.36%)
Installation	25 (30.12%)	14 (31.82%)
Flexibility	15 (18.07%)	1 (2.27%)
Size Trimming	10 (12.05%)	13 (29.55%)

[1] 83 disengaged packages in total in TensorFlow SC.
[2] 44 disengaged packages in total in PyTorch SC.

most common reason in TensorFlow SC mentioned by 23 (27.71%) disengaged packages. The predominant cause is the conflicts among TensorFlow packages. As mentioned in Section 3, the TensorFlow framework releases three packages: `tensorflow`, `tensorflow-cpu`, and `tensorflow-gpu`. The three packages share the same API names, meaning that installing one of them overrides the other already installed package.[10] Developers often adopt two strategies to resolve this issue. The first is to remove TensorFlow packages from dependencies and guide end users to install the correct package for their environments. The other is to check the environment (e.g., the presence of GPUs) in `setup.py` and install the corresponding TensorFlow package. The downside of this strategy is that the package must be installed from the source.

❖ *Bloated.* 15 (18.07%) and 4 (9.09%) packages disengage from the two SCs because they do not use the removed dependencies at all, or package developers assume that the dependencies have been already installed in the environment. For example, a package that has removed PyTorch from its dependencies explained that *If you set up the development environment to work on PyTorch, you would probably have it installed already.*[11] Bloated dependencies increase the package's installation size and import time. Once found, developers will remove them from dependency specification files. It appears that the bloated dependency issue is more common among packages in TensorFlow SC, which is consistent with the manual inspections in Section 3.

Functionality. 27 (32.53%) and 24 (54.55%) packages disengage from TensorFlow and PyTorch SC respectively for functionality improvements.

❖ *Performance.* Sixteen and seven packages disengage from TensorFlow and PyTorch SC for better performance, e.g., simpler model, faster import time and execution time. Some packages just remove dependencies, for example, *Replace MTCNN with Mediapipe to remove TensorFlow + Keras dependency.*[12] Some choose to migrate to other packages, e.g., *The code may become cleaner and faster and more flexible if we strip out PyTorch, and instead (maybe) use concurrent.futures.ProcessPoolExecutor.*[13]

[10] https://github.com/tensorflow/tensorflow/issues/7166
[11] https://github.com/Quansight/pytest-pytorch/issues/13#issue-861367114
[12] https://github.com/apangasa/bestof/issues/21
[13] https://github.com/openclimatefix/nowcasting_dataset/issues/86

❖ *Simplification.* Some packages provide cumbersome functionalities depending on a variety of packages. They disengage from the SC to simplify their functionalities for developers easier to use. 9 (10.84%) and 13 (29.55%) disengaged packages in TensorFlow and PyTorch SC mention this reason. This reason together with *Size Trimming* are the two most common disengagement reasons in PyTorch SC. Packages usually take three approaches to simplify their functionalities. Some packages turn certain functionalities into optional features and the dependencies required by these functionalities into extra dependencies [2]. For example, *These ... may not be needed if the user needs to explain sklearn models. If needed, they can be installed as optional dependencies using pip, pip install dice-ml[deeplearning].*[14] Some packages split their functionalities into multiple packages. For example, *We have recently migrated the pipeline deeplearning-prepare-data to our sister project ClinicaDL which handles everything related to deep-learning.*[15] Some packages take a rather aggressive approach of completely removing functionalities that require these dependencies. For example, *We will remove all dependencies and code related model training and experiments to keep datasetinsights focused on io and statistics.*[16]

❖ *Framework-Independent.* Some packages disengage from a particular DL SC to support more DL frameworks. For example, *Tonic is now free from direct PyTorch and PyTorch Vision dependencies, meaning that if someone wanted to use it with another pipeline (e.g., TensorFlow), they would be able to do so.*[17] Three and five disengaged packages in the two SCs mention this reason.

Installation. 25 (30.12%) and 14 (31.82%) packages disengage from TensorFlow and PyTorch SC for two reasons related to installation.

❖ *Flexibility.* Packages disengage from the SC to ease and simplify the installation procedure to adapt to various environments. The difference between *Flexibility* and *Compatibility* is that the latter is passively triggered by errors while the former is proactively performed to relax installation requirements. For example, *There are multiple official and community TensorFlow packages out there ... The main TensorFlow package works in most cases but ultimately we should allow users to install an alternative package.*[18] 15 disengaged packages in TensorFlow SC and only one disengaged package in PyTorch SC mention this reason. As discussed above, TensorFlow provides separate packages for different hardware platforms, whereas PyTorch provides only a single package. Therefore, packages in TensorFlow SC care more about hardware requirements.

❖ *Size Trimming.* Either TensorFlow or PyTorch is large whose built distributions occupy hundreds of megabytes of space, resulting in a large installation size for packages in the SC as well. Therefore, some packages disengage from the SC to reduce their installation size. Different from *Bloated* where packages disengage from the SC because they do not use the dependencies, dependencies are used by disengaged packages in the case of *Size Trimming*. In addition to removing or turning dependencies into extra dependencies, we also find packages that choose to migrate to ONNX in either SC. For example, 1) *The PyTorch dependency is huge ... Let's try to replace it by ONNX, which is hopefully smaller.*[19] 2) *switched from TensorFlow to ONNX runtime, smaller binary size.*[20] 10 (12.05%) and 13 (29.55%) packages disengage from the two SCs to trim their installation size, suggesting packages in PyTorch SC are more likely to suffer from large installation sizes.

We further analyze the trends of the seven reasons to discern why the number of disengaged packages has increased in the two SCs (as shown in Figure 6). We find that simplifying functionality

[14] https://github.com/interpretml/DiCE/pull/74
[15] https://github.com/aramis-lab/clinica/issues/458
[16] https://github.com/Unity-Technologies/datasetinsights/issues/143
[17] https://tonic.readthedocs.io/en/latest/about/release_notes.html#id4
[18] https://github.com/OpenNMT/OpenNMT-tf/pull/802
[19] https://github.com/raymon-ai/raymon/issues/50
[20] https://github.com/paradigmn/ultrastar_pitch/issues/5

and trimming installation size are the major reasons for the increase of disengaged packages in PyTorch SC, possibly due to the increasing size of the PyTorch framework and the increasingly cumbersome functionalities provided by these disengaged packages. We also find that increasing packages disengage from TensorFlow SC due to incompatibility and performance issues, possibly because more and more packages encounter conflicts among the three TensorFlow packages and TensorFlow fails to meet the disengaged packages' performance requirements such as import time and state-of-the-art models.

> ***Summary for RQ3:*** The number of disengaged packages in either SC shows an increasing trend. Packages disengage from the SC for seven reasons related to three aspects: dependency issues, functional improvements, and ease of installation. The most common disengagement reason in TensorFlow SC is dependency incompatibility issue, and that in PyTorch SC is to simplify functionalities and reduce installation size, indicating packages in different DL SCs face different dependency management problems.

7 IMPLICATIONS

Our study provides rich suggestions for DL framework vendors, researchers, and stakeholders on the maintenance and dependency management practices of PyPI DL SCs. To elicit the practitioners' opinions on our suggestions, we interviewed five people, including two contributors to the PaddlePaddle DL framework, one contributor to the MindSpore DL framework, and two DL researchers from two large IT companies. Overall, they recognize that our suggestions are clear and informative and some suggestions coincide with their practices. They also provide additional information on some suggestions. We elaborate on these suggestions in this section.

7.1 DL SC Maintenance

Framework improvement. A decent DL framework is the foundation of a prosperous DL SC. Our findings reveal two issues that DL frameworks should consider.

- *Package size.* DL frameworks are very large packages whose pre-built distributions occupy hundreds of megabytes of space. As a result, installing DL packages that directly or transitively depend on them consumes a considerable amount of disk space and network traffic. Many packages disengage from DL SCs, especially PyTorch SC, to reduce their installation size (RQ3). Therefore, DL frameworks may consider controlling their package size to retain packages in their SCs. We envision that DL frameworks may split into a base package with many plugin components. The base package contains commonly used functionalities such as CNN layers while the plugin components contain functionalities for specific tasks, e.g., visualization. The PaddlePaddle contributors point out that the inclusion of multiple versions of CUDA binaries is the primary reason for the overwhelming size of DL framework packages. They further reveal that they control PaddlePaddle's package size by setting a range for supported CUDA versions.
- *Multi-hardware support.* A prominent disengagement reason in TensorFlow SC is the conflicts among different TensorFlow packages released for CPU and GPU, respectively (RQ3). Currently, there are two approaches to support multiple hardware platforms. The first is to build separate packages for different hardware platforms. Most DL frameworks, e.g., TensorFlow, MxNet, and PaddlePaddle, adopt this approach. The second is to provide a single package that supports a wide range of hardware platforms, as PyTorch does. Both approaches have their limitations: the former leads to conflicts between different packages, while the latter increases package size. The PaddlePaddle and MindSpore contributors justify that they

choose the first approach because PyPI sets a size limit (60MB) for package distributions and they have to request an increase to the distribution size limits from the PyPI maintainers.[21] The dilemma is largely due to PyPI's lack of support for uploading pre-built distributions for different hardware platforms.[22] PyTorch wheel repository utilizes local version identifiers [1] to differentiate distributions for different hardware platforms.[23] However, local version identifiers are not permitted on PyPI, and even worse, the `packaging` package has removed the support of local version identifiers recently [18]. In summary, current Python packaging specifications pose tons of challenges to DL package maintainers from supporting multiple hardware platforms elegantly. Thus, the Python community may propose adjustments to the packaging specifications (e.g., reintroducing local version identifiers or having `pip` understand the conflicts among packages).

Package Diversity. DL is widely adopted in our society, motivating diverse applications and tools, as evidenced by the 34 package domains identified in RQ1. The diversity of packages in turn boosts the SC's prosperity. We recommend DL framework vendors diversify the chain from two aspects.

— *Competitive advantages.* It is noteworthy that both TensorFlow and PyTorch SC have developed their specializations on *Infrastructure* and *Applications* packages, respectively (RQ1). We also find that official packages play an important role in forming the SC (RQ2). Therefore, for emerging DL frameworks, they can start by providing official packages for domains that account for the most packages in the two SCs such as *NLP* and *CV*. These official packages could provide common datasets and model architectures. However, we notice that not all DL frameworks prioritize CV and NLP-related packages. For example, MindSpore does not provide these packages (i.e., `MindCV` and `MindNLP`) until 2023. Then they may spend efforts on specific domains (e.g., Social Aspects and Reinforcement Learning) to increase their popularity and unique competitiveness in the DL community. This suggestion coincides with what MindSpore does as confirmed by the MindSpore contributor. Specifically, MindSpore has provided many *Sciences* packages such as `MindEarth` and `MindSPONGE` to gain a competitive advantage in the *Sciences* domains.

— *Emerging areas.* DL framework vendors may track the emerging and promising areas in which the adoption of DL techniques makes a difference. For example, as SE researchers, we are surprised at the absence of SE-related popular packages in either SC. Given that DL techniques are increasingly applied in diverse SE tasks (e.g., code search [25] and code generation [64]) and show promising results, we anticipate in the near future that both DL and SE communities could together build tools to facilitate the development of DL for SE, e.g., the "ImageNet" of SE, common data (e.g., abstract syntax tree) preprocessing tools, and large pre-trained models for various SE tasks. The PaddlePaddle and MindSpore contributors recognize the deficiency of AI4SE packages and consider devoting more efforts to this area.

Collaboration. First, DL is widely adopted in various scientific disciplines as revealed by RQ1. Applying DL techniques to these areas usually requires domain expert knowledge. Therefore, DL framework vendors can establish collaboration with researchers and practitioners from other disciplines. Second, DL models are growing dramatically in their size, involving billions of parameters, hundreds of gigabytes of training data, and large GPU clusters. Training such models is barely affordable for any single organization and necessitates collaboration with other organizations. Finally, we find some packages depend on packages from both SCs (RQ1), which

[21] https://www.dampfkraft.com/code/distributing-large-files-with-pypi.html
[22] https://discuss.python.org/t/what-to-do-about-gpus-and-the-built-distributions-that-support-them/7125
[23] https://download.pytorch.org/whl/torch_stable.html

reflects the lack of tool support in one particular SC. Therefore, collaborations with packages in other DL SCs to complement tool support are also strongly encouraged. The PaddlePaddle contributors also emphasize the collaboration between DL framework vendors and hardware vendors to facilitate the deep fusion of hardware and software and to fully unlock the computing potential of the hardware.

⚠ **Risk management.** Tree and Forest clusters assemble about 70% ~ 90% packages through their root packages (RQ2), indicating dense dependency relationships among most DL packages. Such relationships magnify SC risks, e.g., malicious code injections and vulnerabilities in a popular root package trigger the panic of maintainers of a large number of its dependents. Therefore, for DL SCs and SCs in other fields and packaging ecosystems, more attention should be paid to these root packages to avoid and reduce risks. On the one hand, vulnerabilities in root packages will propagate along the SC and impact other packages within the same cluster. Therefore, vulnerability notification and patch propagation tools can be designed to mitigate the cascading impact of security issues of root packages. Such tools may integrate with package managers (e.g., `npm audit`[24]) or code hosting platforms (e.g., Dependabot[25]) to offer timely update notifications to downstream developers and end users. On the other hand, if root packages disengage from the SC, i.e., remove dependencies on the DL framework and its dependents, their dependents will also disengage from the SC. Therefore, the retention of root packages is also critical to SC's stability and sustainability.

7.2 DL SC Dependency Management

⇄* **Interoperability.** We find TensorFlow and PyTorch SC differ greatly in the distribution of package domains (RQ1), indicating that the resources of the DL community are scattered among different DL SCs. This may add extra burden to DL developers. For example, if a developer uses package A in PyTorch SC for prototype development and package B in TensorFlow SC for deployment, she/he must be familiar with both frameworks and perhaps write redundant code. Therefore, we consider interoperability among frameworks can help developers harness the strengths of different DL SCs [56]. Current interoperability tools such as ONNX and MMdnn [56] mainly focus on converting models between different DL frameworks by defining a common set of operators. However, due to the discrepancies among operators supported by different DL frameworks, the engineering efforts are formidable and the application scenarios are still limited. The PaddlePaddle contributors add that the different and evolving implementations of the same operator by different frameworks also pose challenges to existing interoperability tools. Ideally, DL frameworks could conform to a common interoperability tool. However, it is not easy for DL framework vendors to reach a consensus due to different incentives [83]. Therefore, DL practitioners could collectively complement the support of DL frameworks for existing tools. Moreover, novel techniques to achieve interoperability among DL frameworks can be explored.

🐍 **Dependency division.** DL packages usually introduce many dependencies, which increases the risk of dependency issues like dependency incompatibility, cumbersome functionalities, and large installation sizes (RQ3). To simplify their dependency tree and resolve the above issues, some packages split non-core functionalities that require dependencies into "extras". Specifically, these packages choose to split their functionalities into a base package with main dependencies and multiple optional components with corresponding extra dependencies they require. By default, only dependencies required by the base package are installed. Extra dependencies are installed only if the user explicitly declares the use of optional components. To divide dependencies, a good modular architecture is a prerequisite. That is, extra dependencies are not imported by Python code

[24] https://docs.npmjs.com/cli/v9/commands/npm-audit
[25] https://github.com/dependabot

files corresponding to the functionalities provided in the base package so that the base package works properly without extra dependencies. Therefore, practices on how to properly specify extra dependencies and related automation tools can be explored. The two DL practitioners consider this suggestion very useful since they have developed packages supporting different frameworks and this suggestion can help them better organize dependencies and free users from downloading unnecessary dependencies. We consider that similar practices can be applied to SCs in other packaging ecosystems.

Dependency debloating. We find 15 and 4 packages that have disengaged from TensorFlow SC and PyTorch SC mentioned that they did not use the dependencies declared in the metadata at all (RQ3), suggesting that bloated dependencies may be common among PyPI packages. Bloated dependencies increase the package's installation size and slow the import process. Although some work has investigated bloated dependencies in other ecosystems like Java/Maven, e.g., debloating tools [23] and evolution [61, 62], bloated dependencies in Python/PyPI and other packaging ecosystems is still under-investigated [24] and tools for automatically debloating Python dependencies is lacking. We recommend researchers bridge these knowledge gaps in the future.

In addition to DL frameworks, we consider that the implications of controlling package size, risk management, dependency division, and dependency debloating are also applicable to packages in other packaging ecosystems such as NPM. Take the implication of controlling package size as an example, NPM developers often suffer from failures in publishing large packages to NPM and it is recommended that large packages *"should be avoided or broken down into smaller packages"*.[26]

8 THREATS TO VALIDITY

Internal validity concerns how we draw our findings. The main threat comes from author bias introduced by manual labeling of package domains, cluster shapes, and disengagement reasons. To mitigate author bias, two inspectors perform labeling independently, develop coding guides through a series of discussions, and measure the inter-rater reliability throughout the process. The inter-rater reliability results all indicate a high agreement. Then, extracting dependency relationships between packages from the package's distribution metadata is a standard practice in many recent studies (e.g., [33, 55]). However, not all runtime dependencies are listed in the distribution metadata. To evaluate how severe this case is, we randomly sample 100 packages in the two SCs, download their latest distributions, and compare packages imported in the code with dependencies declared in the metadata. We find that six packages import packages that are not declared in the metadata, but all six packages do not have dependent packages and have a monthly download of less than 900. Thus, we believe the impact of this case should be minor and will not invalidate our results. Future research may analyze the packages' source code for a more precise dependency resolution. The selection of popular packages for domain distribution analysis also poses threats to internal validity. To evaluate the representativeness of the selected packages, we further sample 334 and 344 packages in TensorFlow and PyTorch SC, respectively (95% confidence level and 5% confidence interval). We use the derived domains to label the sampled packages and obtain similar findings: over 85% (TensorFlow SC: 89.6%, PyTorch SC: 87.4%) of packages in either SC fall into *Applications*, *Infrastructure*, and *Sciences* categories; the two SCs favor *Infrastructure* (TensorFlow SC: 32.9%, PyTorch SC: 22.2%) and *Applications* (TensorFlow SC: 39.4%, PyTorch SC: 51.2%), respectively. Therefore, we believe that the selected popular packages are representative. Only about 30% of disengaged packages are used to identify disengagement reasons, which poses another threat. Although we do not know how representative these packages are, packages whose code repository documents the rationale behind disengagement may be of better quality than

[26] https://github.com/npm/npm/issues/12750

those without. So their practices should also provide valuable insights into package disengagement. Finally, the use of not-so-recent data may threaten our findings since our findings may not reflect the recent progress of DL such as **large language models (LLMs)** and diffusion models. If thinking it carefully, we can see the use of not-so-recent data mainly threatens the findings of RQ1. However, we argue that their impact on our findings (especially RQ1) is limited for three reasons: (1) The progress (e.g., LLMs) is mostly shared via pre-trained model weights in the Hugging Face platform instead of software packages. (2) The underlying techniques behind them have already been proposed before 2021 and are provided by packages in our dataset. For example, the transformer architecture is widely used by LLMs and was proposed in 2017. Our dataset contains many packages providing transformer architectures such as `pytorch-transformers`. (3) The progress mainly concentrates on the NLP and CV domains, which are the two most popular domains in our dataset. It suggests that the domain distribution may not change greatly even taking the progress into account. Therefore, despite the use of not-so-recent data, we believe our findings still provide valuable insights into DL package SCs.

External validity concerns the generalization of our findings. This paper investigates the SCs of the two most popular and representative DL frameworks, TensorFlow and PyTorch. As a comparison, we also construct SCs for another three DL frameworks, i.e., MindSpore, PaddlePaddle, and MxNet, which include 6, 33, and 87 packages respectively, far less than the number of packages in TensorFlow SC (2,567) and PyTorch SC (3,278). Therefore, we believe the results obtained from the two prosperous DL SCs should also provide valuable insights for other DL frameworks. Still, considering the unique characteristics of the DL field (e.g., the diverse hardware platforms) and PyPI packaging ecosystem, researchers must be cautious before applying our findings to other fields (e.g., web frameworks) or other packaging ecosystems (e.g., Maven or NPM). However, our SC construction approach and analysis framework should be generalizable to other fields and other packaging ecosystems. Further investigations can be conducted to understand the similarities and specifics among SCs in different fields and different ecosystems. Then, PyPI does not cover all Python packages since not every code repository would release packages to PyPI. However, considering that PyPI is the largest and de facto package registry for developers to find, install, and publish Python packages and hosts over 488 thousand packages to date, we believe that package SCs in PyPI are representative. Taking packages released in other platforms (e.g., GitHub code repositories) into account will lead to a more complete package SC, but it needs nontrivial efforts and is beyond the scope of the current study, e.g., how to collect as complete Python code repositories as possible, how to accurately identify packages released by a code repository, and how to identify dependents of these packages. We leave it as future work.

Construct validity is the degree to which our metrics of the cluster size and average degree measure the complexity of clusters. The two metrics are widely used in prior work on analyzing SCs in packaging ecosystems, e.g., ([30, 33]). We believe that the two metrics have a high potential to reflect the cluster's complexity.

9 CONCLUSION

Through analyzing nearly six million PyPI package distributions, we construct version-sensitive SCs of TensorFlow and PyTorch and analyze their package domains, clusters, and disengagement. We find the popular packages in the two SCs cover 34 domains spanning eight categories and the two SCs have developed their specializations on *Infrastructure* and *Applications,* respectively. We find that package clusters in the two SCs are mainly in the shape of Arrow, Star, Tree, and Forest, with increasing dependency complexity. Most packages are concentrated in Tree and Forest clusters. We also find packages disengage from the SC for seven reasons in three groups: dependency issues, functionality improvements, and ease of installation. The most common disengagement reasons in TensorFlow SC and PyTorch SC are different. Overall, the two SCs share similar package domains and cluster shapes but differ in their specialized packages and dependency management issues. Our findings provide rich implications for DL framework vendors, researchers, and practitioners on the maintenance and dependency management practices of PyPI DL SCs.

REFERENCES

[1] 2013. PEP 440 – Version Identification and Dependency Specification. https://peps.python.org/pep-0440/ (Accessed on 02/02/2023).

[2] 2015. PEP 508 – Dependency Specification for Python Software Packages. https://peps.python.org/pep-0508 (Accessed on 2022-08-04).

[3] 2019. Microsoft/CNTK: Microsoft Cognitive Toolkit (CNTK), an Open Source Deep-Learning Toolkit. https://github.com/microsoft/CNTK#disclaimer (Accessed on 12/15/2022).

[4] 2019. Preferred Networks Migrates its Deep Learning Research Platform to PyTorch - Preferred Networks, Inc. https://www.preferred.jp/en/news/pr20191205/ (Accessed on 12/15/2022).

[5] 2021. SolarWinds Orion Security Breach: A Shift in the Software Supply Chain Paradigm. https://snyk.io/blog/solarwinds-orion-security-breach-a-shift-in-the-software-supply-chain-paradigm/ (Accessed on 2022-08-02).

[6] 2022. Core Metadata Specifications – Python Packaging User Guide. https://packaging.python.org/en/latest/specifications/core-metadata/ (Accessed on 08/02/2022).

[7] 2022. Glossary – Python Packaging User Guide. https://packaging.python.org/en/latest/glossary/ (Accessed on 08/03/2022).

[8] 2022. onnx.ai. https://onnx.ai/ (Accessed on 2022-09-01).

[9] 2022. PyPI · The Python Package Index. https://pypi.org/ (Accessed on 12/22/2022).

[10] 2022. Python Package Index (PyPI) – Marketplace – Google Cloud Console. https://console.cloud.google.com/marketplace/product/gcp-public-data-pypi/pypi (Accessed on 2022-08-04).

[11] 2022. PyTorch. https://pytorch.org/ (Accessed on 12/27/2022).

[12] 2022. PyTorch/Vision: Datasets, Transforms and Models Specific to Computer Vision. https://github.com/pytorch/vision (Accessed on 08/14/2022).

[13] 2022. Stack Overflow Developer Survey 2022. https://survey.stackoverflow.co/2022/#section-most-popular-technologies-other-frameworks-and-libraries (Accessed on 01/31/2023).

[14] 2022. TensorFlow. https://www.tensorflow.org/ (Accessed on 12/27/2022).

[15] 2023. API Documentation | TensorFlow v2.12.0. https://www.tensorflow.org/api_docs (Accessed on 06/13/2023).

[16] 2023. Ecosystem | PyTorch. https://pytorch.org/ecosystem/ (Accessed on 02/03/2023).

[17] 2023. Igraph – Network Analysis Software. https://igraph.org/ (Accessed on 06/27/2023).

[18] 2023. Let's Permit '+Local.Version.Label' in Version Specifiers - Packaging. https://discuss.python.org/t/lets-permit-local-version-label-in-version-specifiers/22781 (Accessed on 02/02/2023).

[19] 2023. Packaging. https://packaging.pypa.io/en/stable/ (Accessed on 02/03/2023).

[20] 2023. PyTorch C++ API – PyTorch main documentation. https://pytorch.org/cppdocs/ (Accessed on 06/13/2023).

[21] Sadika Amreen, Bogdan Bichescu, Randy Bradley, Tapajit Dey, Yuxing Ma, Audris Mockus, Sara Mousavi, and Russell Zaretzki. 2019. *A Methodology for Measuring FLOSS Ecosystems*. Springer Singapore, Singapore, 1–29. https://doi.org/10.1007/978-981-13-7099-1_1

[22] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2015. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.).

[23] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth investigation into debloating modern Java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 135–146.

[24] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2022. Towards better dependency management: A first look at dependency smells in Python projects. *IEEE Transactions on Software Engineering* (2022), 1–26.

[25] Qingying Chen and Minghui Zhou. 2018. A neural framework for retrieval and summarization of source code. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 826–831.

[26] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 750–762.

[27] Zhenpeng Chen, Huihan Yao, Yiling Lou, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, and Xuanzhe Liu. 2021. An empirical study on deployment faults of deep learning based mobile applications. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 674–685.

[28] Francois Chollet. 2017. *Deep Learning with Python* (1st ed.). Manning Publications Co., USA.

[29] Daniela S. Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284.

[30] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proceedings of the 10th European Conference on Software Architecture Workshops* (Copenhagen, Denmark) *(ECSAW '16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 4 pages.

[31] Alexandre Decan, Tom Mens, and Maëlick Claes. 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2–12.

[32] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the impact of security vulnerabilities in the npm package dependency network. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 181–191.

[33] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (01 Feb. 2019), 381–416.

[34] Jiankang Deng, Jia Guo, Niannan Xue, and Stefanos Zafeiriou. 2019. ArcFace: Additive angular margin loss for deep face recognition. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 4685–4694. https://doi.org/10.1109/CVPR.2019.00482

[35] Tapajit Dey and Audris Mockus. 2018. Are software dependency supply chain metrics useful in predicting change of popularity of NPM packages?. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering* (Oulu, Finland) *(PROMISE'18)*. Association for Computing Machinery, New York, NY, USA, 66–69.

[36] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding software-2.0: A study of machine learning library usage and evolution. *ACM Trans. Softw. Eng. Methodol.* 30, 4, Article 55 (Jul. 2021), 42 pages. https://doi.org/10.1145/3453478

[37] Kai Gao, Zhixing Wang, Audris Mockus, and Minghui Zhou. 2023. On the variability of software engineering needs for deep learning: Stages, trends, and application types. *IEEE Transactions on Software Engineering* 49, 2 (2023), 760–776. https://doi.org/10.1109/TSE.2022.3163576

[38] Haiqiao Gu, Hao He, and Minghui Zhou. 2023. Self-admitted library migrations in Java, JavaScript, and Python packaging ecosystems: A comparative study. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 627–638. https://doi.org/10.1109/SANER56733.2023.00064

[39] Junxiao Han, Shuiguang Deng, David Lo, Chen Zhi, Jianwei Yin, and Xin Xia. 2020. An empirical study of the dependency networks of deep learning libraries. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 868–878.

[40] Junxiao Han, Emad Shihab, Zhiyuan Wan, Shuiguang Deng, and Xin Xia. 2020. What do programmers discuss about deep learning frameworks. *Empirical Software Engineering* 25, 4 (01 Jul. 2020), 2694–2747.

[41] Hao He, Runzhi He, Haiqiao Gu, and Minghui Zhou. 2021. 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* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 478–490.

[42] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on GitHub Dependabot. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4004–4022. https://doi.org/10.1109/TSE.2023.3278129

[43] Sture Holm. 1979. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics* 6, 2 (1979), 65–70. http://www.jstor.org/stable/4615733

[44] Kaifeng Huang, Bihuan Chen, Susheng Wu, Junming Cao, Lei Ma, and Xin Peng. 2023. Demystifying dependency bugs in deep learning stack. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 450–462. https://doi.org/10.1145/3611643.3616325

[45] Nargiz Humbatova, Gunel Jahangirova, Gabriele Bavota, Vincenzo Riccio, Andrea Stocco, and Paolo Tonella. 2020. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1110–1121.

[46] Md. Johirul Islam, Giang Nguyen, Rangeet Pan, and Hridesh Rajan. 2019. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 510–520.

[47] Md. Johirul Islam, Rangeet Pan, Giang Nguyen, and Hridesh Rajan. 2020. Repairing deep neural networks: Fix patterns and challenges. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1135–1146.

[48] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2020. An empirical study on bugs inside TensorFlow. In *Database Systems for Advanced Applications*, Yunmook Nah, Bin Cui, Sang-Won Lee, Jeffrey Xu Yu, Yang-Sae Moon, and Steven Euijong Whang (Eds.). Springer International Publishing, Cham, 604–620.

[49] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The symptoms, causes, and repairs of bugs inside a deep learning library. *Journal of Systems and Software* 177 (2021), 110935.

[50] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories* (Buenos Aires, Argentina) *(MSR '17)*. IEEE Press, 102–112.

[51] Klaus Krippendorff. 2011. Computing Krippendorff's alpha-reliability. (2011).

[52] Klaus Krippendorff. 2018. *Content Analysis: An Introduction to its Methodology*. Sage publications.

[53] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. 2020. Selecting third-party libraries: The practitioners' perspective. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 245–256. https://doi.org/10.1145/3368089.3409711

[54] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: Escape from NuGet dependency maze. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 1545–1557.

[55] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 672–684.

[56] Yu Liu, Cheng Chen, Ru Zhang, Tingting Qin, Xiang Ji, Haoxiang Lin, and Mao Yang. 2020. Enhancing the interoperability between deep learning frameworks by model conversion. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1320–1330.

[57] H. B. Mann and D. R. Whitney. 1947. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics* 18, 1 (1947), 50 – 60.

[58] Mary L. McHugh. 2012. Interrater reliability: The Kappa statistic. *Biochemia Medica* 22, 3 (2012), 276–282.

[59] John T. Mentzer, William DeWitt, James S. Keebler, Soonhong Min, Nancy W. Nix, Carlo D. Smith, and Zach G. Zacharia. 2001. Defining supply chain management. *Journal of Business Logistics* 22, 2 (2001), 1–25. https://doi.org/10.1002/j.2158-1592.2001.tb00001.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/j.2158-1592.2001.tb00001.x

[60] Rebecca Passonneau. 2006. Measuring agreement on set-valued items (MASI) for semantic and pragmatic annotation. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*. European Language Resources Association (ELRA), Genoa, Italy.

[61] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated Java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1021–1031.

[62] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering* 26, 3 (25 Mar. 2021), 45.

[63] Klaas-Jan Stol, Paul Ralph, and Brian Fitzgerald. 2016. Grounded theory in software engineering research: A critical review and guidelines. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16)*. Association for Computing Machinery, New York, NY, USA, 120–131.

[64] Zeyu Sun, Qihao Zhu, Yingfei Xiong, Yican Sun, Lili Mou, and Lu Zhang. 2020. TreeGen: A tree-based transformer architecture for code generation. *Proceedings of the AAAI Conference on Artificial Intelligence* 34, 05 (Apr. 2020), 8984–8991. https://doi.org/10.1609/aaai.v34i05.6430

[65] Xin Tan, Kai Gao, Minghui Zhou, and Li Zhang. 2022. An exploratory study of deep learning supply chain. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 86–98.

[66] Yingchen Tian, Yuxia Zhang, Klaas-Jan Stol, Lin Jiang, and Hui Liu. 2022. What makes a good commit message?. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2389–2401.

[67] V. A. Traag, L. Waltman, and N. J. van Eck. 2019. From Louvain to Leiden: Guaranteeing well-connected communities. *Scientific Reports* 9, 1 (26 Mar. 2019), 5233.

[68] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 644–655.

[69] Duc-Ly Vu. 2021. py2src: Towards the automatic (and reliable) identification of sources for PyPI package. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1394–1396.

[70] Duc-Ly Vu, Fabio Massacci, Ivan Pashchenko, Henrik Plate, and Antonino Sabetta. 2021. LastPyMile: Identifying the discrepancy between sources and packages. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 780–792.

[71] Duc-Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and combosquatting attacks on the Python ecosystem. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. 509–514. https://doi.org/10.1109/EuroSPW51379.2020.00074

[72] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 125–135.

[73] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. 2016. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 351–361.

[74] Jianyu Wu, Hao He, Wenxin Xiao, Kai Gao, and Minghui Zhou. 2022. Demystifying software release note issues on GitHub. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension* (Virtual Event) *(ICPC '22)*. Association for Computing Machinery, New York, NY, USA, 602–613. https://doi.org/10.1145/3524610.3527919

[75] Jianyu Wu, Weiwei Xu, Kai Gao, Jingyue Li, and Minghui Zhou. 2023. Characterize software release notes of GitHub projects: Structure, writing style, and content. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 473–484. https://doi.org/10.1109/SANER56733.2023.00051

[76] Weiwei Xu, Hao He, Kai Gao, and Minghui Zhou. 2023. Understanding and remediating open-source license incompatibilities in the PyPI ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 178–190. https://doi.org/10.1109/ASE56229.2023.00175

[77] Yilin Yang, Tianxing He, Zhilong Xia, and Yang Feng. 2022. A comprehensive empirical study on bug characteristics of deep learning frameworks. *Information and Software Technology* 151 (2022), 107004.

[78] Nusrat Zahan, Thomas Zimmermann, Patrice Godefroid, Brendan Murphy, Chandra Maddila, and Laurie Williams. 2022. What are weak links in the npm supply chain?. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. 331–340.

[79] Ru Zhang, Wencong Xiao, Hongyu Zhang, Yu Liu, Haoxiang Lin, and Mao Yang. 2020. An empirical study on program failures of deep learning jobs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1159–1170.

[80] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. 104–115.

[81] Xiaoyu Zhang, Juan Zhai, Shiqing Ma, and Chao Shen. 2021. AutoTrainer: An automatic DNN training problem detection and repair system. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 359–371.

[82] Yuhao Zhang, Yifan Chen, Shing-Chi Cheung, Yingfei Xiong, and Lu Zhang. 2018. An empirical study on TensorFlow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 129–140.

[83] Yuxia Zhang, Minghui Zhou, Audris Mockus, and Zhi Jin. 2021. Companies' participation in OSS development–An empirical study of OpenStack. *IEEE Transactions on Software Engineering* 47, 10 (2021), 2242–2259.

Received 26 June 2023; revised 11 November 2023; accepted 18 December 2023