

Analyzing the Accessibility of GitHub Repositories for PyPI and NPM Libraries

Alexandros Tsakpinis

fortiss - Research Institute of the Free State of Bavaria
Munich, Germany
tsakpinis@fortiss.org

Alexander Pretschner

Technical University of Munich
Munich, Germany
alexander.pretschner@tum.de

ABSTRACT

Industrial applications heavily rely on open-source software (OSS) libraries, which provide various benefits. But, they can also present a substantial risk if a vulnerability or attack arises and the community fails to promptly address the issue and release a fix due to inactivity. To be able to monitor the activities of such communities, a comprehensive list of repositories for the libraries of an ecosystem must be accessible. Based on these repositories, integrated libraries of an application can be monitored to observe whether they are adequately maintained. In this descriptive study, we analyze the accessibility of GitHub repositories for PyPI and NPM libraries. For all available libraries, we extract assigned repository URLs, direct dependencies and use the page rank algorithm to comprehensively analyze the ecosystems from a library and dependency chain perspective. For invalid repository URLs, we derive potential reasons. Both ecosystems show varying accessibility to GitHub repository URLs, depending on the page rank score of the analyzed libraries. For individual libraries, up to 73.8% of PyPI and up to 69.4% of NPM libraries have repository URLs. Within dependency chains, up to 80.1% of PyPI libraries have URLs, while up to 81.1% for NPM. That means, most libraries, especially the ones of increasing importance, can be monitored on GitHub. Among the most common reasons for invalid repository URLs is no URLs being assigned at all, which amounts up to 17.9% for PyPI and up to 39.6% for NPM. Package maintainers should address this issue and update the repository information to enable monitoring of their libraries.

CCS CONCEPTS

• **Software and its engineering** → **Maintaining software.**

KEYWORDS

Maintenance Activities, OSS Libraries, Repository Mining

ACM Reference Format:

Alexandros Tsakpinis and Alexander Pretschner. 2024. Analyzing the Accessibility of GitHub Repositories for PyPI and NPM Libraries. In *28th International Conference on Evaluation and Assessment in Software Engineering (EASE 2024)*, June 18–21, 2024, Salerno, Italy. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3661167.3661231>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

EASE 2024, June 18–21, 2024, Salerno, Italy

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1701-7/24/06...\$15.00

<https://doi.org/10.1145/3661167.3661231>

1 INTRODUCTION

Over time, Open Source Software (OSS) became a standard practice in the product life cycle due to commercial, engineering, and quality reasons [13]. Roughly 80% of code in commercial products comprises OSS components, reflecting the widespread utilization in the software industry [24]. Software libraries as key element of OSS play a key role in modern software development. Libraries, also called packages, enable the reuse of established, well-tested code for specific functionalities, reducing the need for coding from scratch [3]. After a library has been included in a project, it can be defined as a dependency [7]. OSS dependencies can be affected by vulnerabilities (e.g., Log4j¹) or supply-chain-attacks (e.g., npm colors²) representing significant challenges in the software industry [10]. If a vulnerability or an attack is identified, the library's community generally releases a fixed version soon after [26].

But, it can happen that support for a library is terminated or suspended resulting in the harmful situation of no maintenance activities being available [3, 25]. Consequently, it is crucial that direct and transitive dependencies of an application are monitored by the application maintainers accordingly [19, 28]. Monitoring the maintenance activities of dependencies requires accessibility to corresponding repositories which are reachable via URLs linking to code management systems (CMS), such as GitHub. However, an exhaustive investigation into the accessibility of GitHub repository URLs for libraries of an ecosystem is currently missing.

In this work, we focus on GitHub as the CMS, given its ongoing popularity for hosting OSS libraries [14]. Similar to other studies, we select PyPI and NPM as ecosystems due to their widespread use in modern software development and strong integration of libraries in their development practices [1, 9]. Thus, available GitHub repository URLs are collected for each library from their package managers. For libraries without valid URLs, possible reasons for their absence are documented. In this context, a valid URL is confirmed by an HTTP status code 200 and conforms syntactically to a standard GitHub URL scheme. Since libraries are heavily interconnected, they should not only be analyzed individually, but also include an analysis of their dependency chains. Thus, direct dependencies are collected for each library to create a dependency graph that is enriched with the available repository URL information. As the importance of individual libraries within their ecosystem can differ significantly, the dependency graph is used to calculate the importance of each library, as measured by its page rank score, and incorporate it in the analysis. This graph serves as the basis to gain insights into the accessibility of valid GitHub repository URLs from a library and dependency chain perspective.

¹<https://logging.apache.org/log4j/2.x/>

²<https://snyk.io/de/blog/open-source-npm-packages-colors-faker/>

To enable the monitoring of maintenance activities for PyPI and NPM libraries on GitHub, insights into the accessibility of valid GitHub repository URLs are needed. The following research questions (RQ) aim to provide evidence for these insights:

RQ1: What is the ratio of libraries with valid GitHub repository URLs when evaluating the libraries of each ecosystem individually?

RQ2: What are possible reasons for missing valid GitHub repository URLs and what does the corresponding distribution look like?

RQ3: What is the ratio of libraries with valid GitHub repository URLs when focusing on dependency chains within each ecosystem?

The descriptive analysis of the PyPI and NPM ecosystem shows mixed accessibility to valid GitHub URLs for individual libraries and libraries in dependency chains depending on their page rank score. For individual libraries, up to 73.8% of PyPI and up to 69.4% of NPM libraries contain valid GitHub URLs. For libraries within dependency chains, up to 80.1% of PyPI and up to 81.1% of NPM libraries have valid URLs. As the accessibility is particularly positive for more important libraries, the maintenance activities for most of these libraries can be monitored. One of the main reasons for invalid URLs is the absence of any URL, which accounts for up to 17.9% for PyPI and up to 39.6% for NPM. To enable monitoring of maintenance activities for even more libraries and dependency chains, maintainers should update the repository information of their libraries in the package managers accordingly. Additionally, the analysis reveals insights into the complexity of the PyPI and NPM ecosystems. On average, there are around 3 direct and 37 transitive dependencies with URLs in PyPI, while 3 and 32 in NPM. Furthermore, PyPI shows maximum values of 270 direct and 667 transitive dependencies, while NPM holds 967 direct and 7780 transitive dependencies with URLs. Given the high complexity and the need to monitor the maintenance activities of dependencies, we recommend an automated monitoring approach in the future.

2 RELATED WORK

In the following, related work is presented to highlight the importance of monitoring OSS libraries and to point out the differences of this work compared to other studies that analyze OSS ecosystems.

Recently, the US Cybersecurity and Infrastructure Agency (CISA) published a roadmap, focusing on the development of a framework for prioritizing OSS risks, also based on the level of maintenance. However, it lacks a specific definition, such as monitoring maintenance activities in a CMS. This roadmap highlights the importance of our research even for government-level organizations [8].

Numerous studies investigate package dependency networks for different ecosystems, focusing on security issues such as vulnerabilities [2, 12], intrusion of malicious code [15] and evolution of ecosystem robustness [16]. Another well-researched area is package versioning, which includes updating strategies [17], configuration issues [23], dependency conflicts [31], smells [5] and errors [22]. In addition, the characteristics of highly selected NPM packages [20], the determinants of sustainable OSS Python projects [30] as well as the usage of trivial packages in PyPI and NPM [1] are investigated. All these studies analyze various aspects of ecosystems but none has investigated the accessibility of valid GitHub repository URLs for their analysis, so our study aims to fill this gap.

Unlike the studies mentioned before focusing on specific aspects of an ecosystem, some studies examine broader characteristics such as dependencies [4, 9], with a focus on transitive dependencies [11, 18]. While numerous studies analyze libraries and their dependencies for different ecosystems, none has specifically addressed the accessibility of valid repository URLs for libraries within dependency chains. Thus, our research also seeks to close this gap.

3 RESEARCH METHOD

In the following, our research method for this descriptive study is explained in more detail, with a focus on how the data collection and data analysis have been conducted. The primary goal is to gather evidence to assess the accessibility of GitHub repository URLs enabling a CMS-based monitoring of maintenance activities.

3.1 Data Collection

To assess the accessibility of valid GitHub repository URLs within an ecosystem, we automatically collect assigned URLs for each library using different Python scripts. Therefore, a comprehensive list of available libraries is required which is provided by corresponding endpoints^{3,4}. For details about each library, such as assigned URLs and dependency information, additional endpoints^{5,6} are consumed. As dependency information, we collect all direct and optional dependencies for the latest published library version to create a dependency graph fully capturing the dependency tree without limiting the transitive depth. For each assigned GitHub URL, automated validity checks are performed, saving valid URLs or documenting one of the following reasons for invalid URLs:

- No URL assigned at all
- 404 status code returned by ecosystem endpoints^{5,6}
- 404 or other GitHub related status codes
- Use of private or unknown CMS
- Use of GitLab or BitBucket as CMS
- Malformed CMS URLs

The data collection results in a JSON file for each ecosystems linking library names with their direct dependencies, and either a repository URL or a potential reason for the absence of a valid URL.

3.2 Data Analysis

In our data analysis, we use standard terms such as nodes, dependency chains and dependency graph. Each library represents a node in the dependency graph and is connected to its direct dependencies by dependency chains. Together, this network of nodes and dependency chains forms the dependency graph representing an ecosystem. The data from Section 3.1 serve as the basis for various analyses providing insights from two perspectives. First, all libraries should be represented as individual nodes without considering dependency chains providing a library perspective on the ecosystems. Second, the data should be analyzed from a dependency chain perspective, focusing on all dependency chains originating from existing nodes within the dependency graph. Therefore, the following metrics are calculated for the library perspective:

³PyPI: <https://pypi.org/simple/>

⁴NPM: https://skimdb.npmjs.com/registry/_all_docs/

⁵PyPI: https://pypi.org/pypi/<package_name>/json

⁶NPM: https://skimdb.npmjs.com/<package_name>

- Ratio of packages with valid repository URLs
- Distribution of reasons for invalid URLs (see Section 3.1)

These are the metrics for the dependency chain perspective:

- Ratio of libraries in dependency chains having URLs
- Average and maximum number of direct and transitive dependencies with URLs

Analyzing nodes in isolation provides a basic understanding of individual libraries and focuses on their standalone properties. In contrast, looking at dependency chains provides a more comprehensive view on the structure of the ecosystem leading to insights that are not apparent when looking at individual libraries.

Note that the metrics for both perspectives are calculated for different subsets of the dependency graph. As a measure for defining subsets, we use the page rank algorithm inspired by Mujahid et al., where each node within a dependency graph is assigned an importance value based on its incoming and outgoing dependency chains [21]. This approach of categorizing libraries according to their importance enables a more differentiated understanding and improves the analysis of libraries using different levels of importance.

Since the importance value of the page rank algorithm is largely incorporated in the data analysis, we also conduct a correlation analysis to determine whether any of the calculated variables show a correlation with the importance value.

4 RESULTS

The data collection was carried out on February 16, 2024 resulting in 514,535 PyPI and 2,679,007 NPM libraries. When analyzing the dependencies for each library, we neglect libraries that can not be found in the initial ecosystem's library list^{3,4}. In total, we were not able to find 0.3% of PyPI and 0.8% of NPM libraries from a node and each 0.2% from a dependency chain perspective. Also we exclude 3.0% of PyPI and 33.9% of NPM libraries which return a status code (mostly 404) when querying the detailed ecosystem endpoints^{5,6}. These errors arise when the package name retrieved from the comprehensive library list endpoint^{3,4} could not be found at the detailed endpoint^{5,6}, indicating inconsistencies between the two endpoints which should be fixed by the ecosystems. The final dataset has 499,274 PyPI libraries with 1,068,768 dependency chains, while 1,770,463 NPM libraries with 5,408,141 dependency chains.

The analysis focuses on libraries with increasing importance measured by the page rank score, as these nodes have a greater influence within the dependency tree and are used more frequently. The x-axis of the shown figures defines the subset on which the characteristics are calculated, e.g., $x = 10$ specifies a subset of more important libraries, which contains the top 10% of libraries ranked by page rank score, while $x = 100$ describes the top 100% of libraries, which corresponds to the entire data set also containing less important libraries. Consequently, the results are often described as value ranges relating to the different subsets. If not stated otherwise, the observations mentioned in the following apply to both ecosystems, and the term URL refers to a valid GitHub repository URL.

4.1 RQ1+RQ2: Library perspective

Figure 1 presents the results of the data analysis of how the characteristics of the nodes in the dependency graph behave independently without looking at dependency chains. In particular, the

distribution of valid GitHub repository URLs and various reasons for invalid URLs are computed, as introduced in Section 3.1.

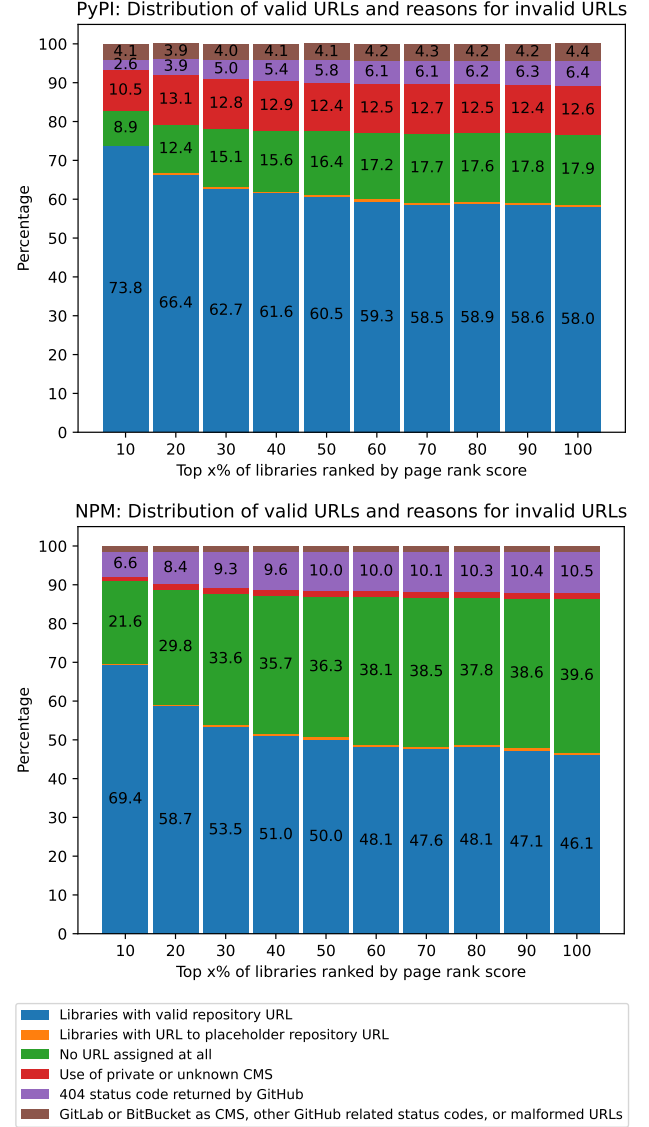


Figure 1: Distribution of URLs and reasons for invalid URLs

Valid URLs are further divided into "real" valid and placeholder URLs (orange), which are used for sample projects⁷ in PyPI, or for insecure⁸ or deprecated⁹ packages in NPM. The overall ratio of placeholder URLs is 0.7% for both PyPI and NPM. Even if these ratios are neglectable, developers should avoid such libraries, as these repositories do not reflect the actual code base of the library, which prevents monitoring of maintenance activities in a CMS.

Figure 1 shows, that more important libraries tend to have an increasing percentage of URLs (blue), with values ranging from 58.0%

⁷<https://github.com/pypa/sampleproject>

⁸<https://github.com/npm/security-holder>

⁹<https://github.com/npm/deprecate-holder>

to 73.8% for PyPI and from 46.1% to 69.4% for NPM. Consequently, the accessibility from a node perspective is not too promising even for libraries with increasing importance, evidenced by maximum values of 73.8% for PyPI and 69.4% for NPM. Additionally, there are some severe URL gaps for less important NPM libraries preventing the monitoring of maintenance activities in a CMS. This observation can be seen by the increase in libraries with URLs when moving from $x = 100$ to $x = 10$, where the values increase by 50.5%, while less important libraries are more and more neglected.

A significant fraction of packages, especially visible for NPM, have no URL assigned at all (green), which accounts for 8.9%-17.9% for PyPI and 21.6%-39.6% for NPM. The ratios of both ecosystems decrease with increasing library importance. Such occurrences of missing URLs might be an indicator of poor maintenance practices when updating package metadata. Maintainers could take these findings to review their packages and ensure that the repository URLs are valid and reflect the current state of their codebase. Another interpretation may be that some libraries are closed-source and do not allow access to their repositories on purpose.

There are also GitHub-related problems with status code 404 (purple) representing a syntactically valid GitHub repository URL that cannot be found at this location. With increasing importance, PyPI's ratios decrease from 6.4% to 2.7%, while NPM's ratios decrease from 10.5% to 6.6%. The widespread occurrence of such errors indicates a need for improvement in the package update processes, as numerous repositories have been moved or deleted without the package information being updated accordingly.

PyPI shows small percentages for URL-related errors such as malformed URLs, various GitHub related status codes, or GitLab or Bitbucket as CMS (brown) consistently having values below 4.4%. But, between 10.5% and 13.1% are assigned to unknown CMS providers (red) meaning that occasionally, links lead to not widely known CMS providers, such as those not affiliated with GitHub, GitLab, or Bitbucket, or even to private homepages instead of publicly accessible CMSs. We recommend that package maintainers replace these not publicly accessible CMSs, as they hinder the monitoring of maintenance activities. For the NPM ecosystem, errors related to URLs (brown) or unknown CMS providers rarely occur (red) with values consistently below 1.7%. In particular, the absence of the latter category suggests that more publicly accessible CMS providers are linked in NPM, which is a positive sign regarding the accessibility of valid GitHub repository URLs.

4.2 RQ3: Dependency chain perspective

In the following, different characteristics of the libraries including their dependency chains are analyzed. First, Figure 2 analyzes direct, transitive and full chains regarding the ratio of libraries within the dependency chains to which an URL is assigned describing a measure for the accessibility of URLs. In this context, *direct chains* refer to dependencies that are explicitly chosen by a library. *Transitive chains* include all dependencies that are transitively connected but cannot be directly influenced. A *full chain* represents the sum of direct and transitive dependencies, plus the characteristics of the root library itself.

Figure 2 shows that with increasing importance, between 68.7% and 80.1% of the libraries within full dependency chains have URLs

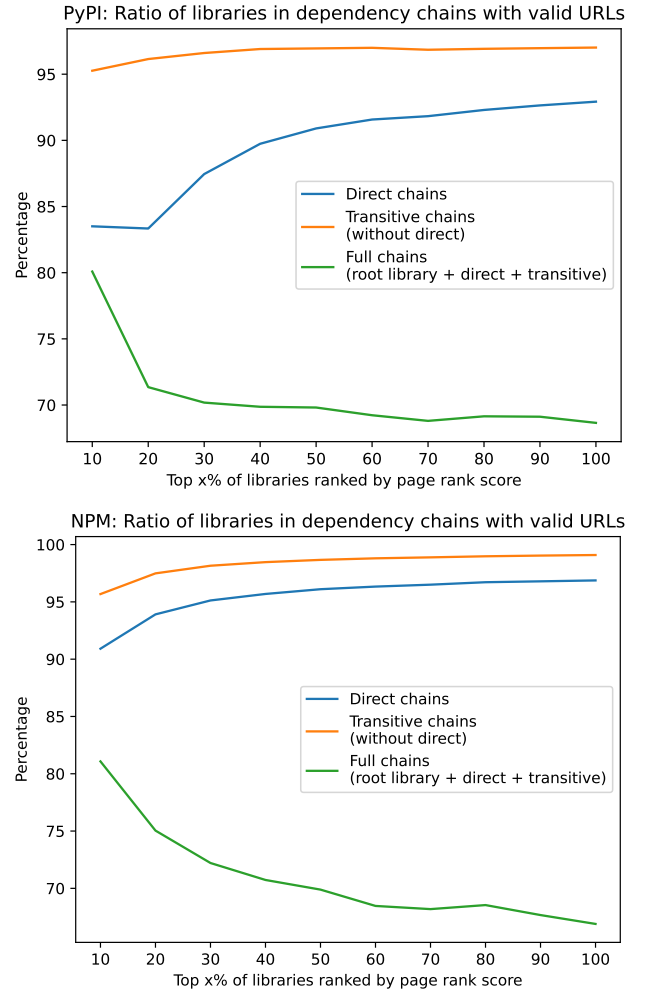


Figure 2: Ratio of libraries in chains having URLs

for PyPI, while between 66.9% and 81.1% for NPM. Within direct dependency chains, between 83.3% and 92.9% of PyPI libraries and between 90.9% and 96.9% of NPM libraries have access to URLs. In transitive chains, between 95.3% and 97.0% of PyPI libraries and 95.7% to 99.1% of NPM libraries can access URLs. Overall, the high ratios, especially for more important libraries, are a good indication that most libraries within the dependency chains provide access to valid GitHub URLs enabling monitoring of maintenance activities. Interestingly, the ratios for direct and transitive dependency chains are always higher than for full ones. This means that, the root libraries often lack URLs, but their direct and transitive dependencies have URLs assigned. Thus, most missing URLs in libraries within full dependency chains can be corrected by the package maintainers themselves, as they are in control over the repository information of the library uploaded to the ecosystems.

Second, the average number of direct and transitive dependencies with URLs is calculated indicating the complexity of each ecosystem. Figure 3 shows the average number of direct and transitive dependencies with URLs having values up to around 3 and 37 for

PyPI, and up to 3 and 32 for NPM. Note, that direct dependencies are not included in the transitive ones. The large number of transitive dependencies with URLs multiplied with the number of direct ones shows the high complexity of the dependency graphs and provides evidence that manual monitoring is not feasible describing the need for an automated monitoring approach in the future.

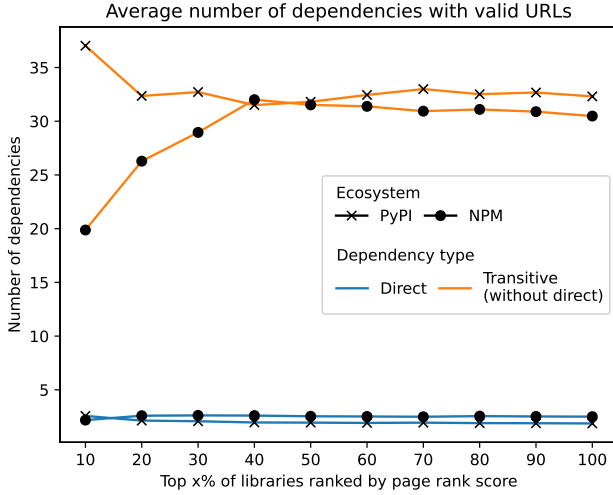


Figure 3: Direct and transitive dependencies with URLs

Third, the maximum number of direct and transitive dependencies with URLs is measured for each ecosystem which additionally provides evidence for the need for an automated monitoring approach in the future. Note, that these observations are not shown in any Figure. PyPI’s maximum values are 270 direct and 667 transitive dependencies with URLs, while NPM’s maxima are 967 and 7780.

4.3 Correlation analysis

To evaluate different hypotheses regarding the importance value of the page rank algorithm, which plays a central role in this study, we conduct various correlation analyses between different features of a library and its importance value. Therefore, we use the Pearson coefficient in order to find linear relationships [6].

In particular, we carry out three correlation analyses which are motivated by the results presented in Figure 1, 2 and 3. Intuitively, our hypotheses are that more important libraries would have a higher chance of being assigned an URL (Figure 1), would contain higher ratios of libraries in direct, transitive and full dependency chains with URLs (Figure 2), and would exhibit a higher number of direct and transitive dependencies with URLs (Figure 3). Surprisingly, all our hypotheses about expected correlations could not be confirmed with Pearson coefficients ranging from -0.0029 to 0.0075 for PyPI, and values between -0.0023 and 0.011 for NPM. These results indicate that there is almost no relationship between the analyzed features as the values are close to zero.

The weak correlations can be explained by the study’s focus on libraries with increasing importance, while less important libraries are neglected for deeper analysis. Further investigations show that less important libraries receive more URLs than expected, and thus

relativize the positive results of more important libraries, which finally leads to such low Pearson coefficients.

5 DISCUSSION

We investigated the accessibility of valid GitHub repository URLs from a node and dependency chain perspective by using different subsets of libraries varying their importance to finally gain a deeper understanding of the ecosystems. In this context, the results of the most important libraries are of greater relevance as they play a central role in their ecosystems and exert a significant influence.

From a library perspective, the results for PyPI and NPM libraries are not overly promising regarding the accessibility of valid GitHub repository URLs, not even for their most important libraries. This applies particularly for less important NPM libraries where a significant portion of URLs is missing. For both ecosystems we identified clear areas where improvements could be made. This may include better tooling for repository URL verification, automated checks for link validity, and more robust community practices for updating package metadata. In addition, library ecosystems should focus on eliminating inconsistencies between the comprehensive package name lists and the detailed package endpoints they provide.

But, even more important than the library perspective is the dependency chain perspective when assessing the accessibility of URLs. This is because the ratio of how many libraries within a dependency chain have an URL assigned provides a more realistic view since libraries rarely appear alone without having dependencies themselves. The ratio of libraries with valid GitHub repository URLs within direct, transitive and full dependency chains shows positive signs for both ecosystems, especially for libraries with increasing importance. Interestingly, most missing URLs can be found in root libraries, which could be immediately corrected by the package maintainers themselves. The results on existing dependency chains also reflect the high complexity of these ecosystems, given by the large average and maximum number of direct and transitive dependencies with valid GitHub repository URLs, making an automated monitoring approach for maintenance activities indispensable in the future.

6 THREATS TO VALIDITY

Threats to validity are divided into the following four aspects [27]:

Construct validity: For the primary study objective of analyzing the accessibility of valid GitHub repository URLs, there is no threat to construct validity, as the URLs are checked against a standard GitHub URL scheme and an HTTP status code 200 without any interpretation. A minor threat to construct validity arises when interpreting and deriving reasons for invalid URLs. This threat is mitigated by deriving reasons only from objective patterns (e.g., status codes, CMS provider names, URL format) found in the dataset. Another threat arises from our approach to resolving dependencies for a particular library. We disregarded version information for any dependency, consistently opting for the latest version. This may not accurately reflect the entire ecosystem’s dynamics. Nevertheless, we contend that the resulting discrepancies in the calculated page rank score of a library are negligible.

Internal validity: No threat to internal validity, as we describe ecosystem characteristics without investigating causal relations.

External validity: We focus on GitHub as CMS, and PyPI and NPM as ecosystems. Although the results may not generalize to other CMS or ecosystems, the applied research method can be transferred to any publicly crawlable ecosystem where libraries can include links, such as CMS references, in their metadata.

Reliability: While all data are collected from publicly accessible sources, which theoretically ensures reproducibility of the results, a threat to reliability still arises due to the nature of the ecosystems. The collected data reflect a specific timestamp from the collection day, posing a challenge for reproducibility as these ecosystems typically lack built-in time-travel functionality. To ensure complete reproducibility, the collected data and analysis code are published on figshare [29]. Another threat is the frequent emergence or modification of libraries posing a risk of gaps or outdated information in our dataset. To address this threat, the study must be repeated.

7 CONCLUSION AND FUTURE WORK

The paper analyzed the accessibility of valid GitHub repository URLs for the PyPI and NPM ecosystems. This is necessary for monitoring the maintenance activities of OSS libraries within their respective CMS, which enables the identification of poorly maintained libraries representing a potential security risk. We analyzed the PyPI and NPM ecosystem from a library and a dependency chain perspective providing a comprehensive view on the accessibility of valid GitHub repository URLs. Particularly, the high ratios of libraries with URLs in full dependency chains (PyPI: up to 80.1%, NPM: up to 81.1%) serve as proof that the maintenance activities of most libraries in dependency chains can be monitored on GitHub, which is especially true for libraries of increasing importance.

Future work should explore the reasons for missing, outdated or invalid URLs to understand why maintainers neglect such maintenance practices. Additionally, we aim to periodically repeat this study to observe ecosystem evolutions regarding GitHub repository URL accessibility. Given the high number of direct and transitive dependencies with GitHub repository URLs, we also plan to investigate what concrete metrics capture the maintenance activities of OSS libraries on GitHub, which can be used to develop an automated monitoring approach identifying poorly maintained libraries.

REFERENCES

- [1] Rabe Abdalkareem, Vinicius Oda, Suhaib Mujahid, and Emad Shihab. 2020. On the impact of using trivial packages: An empirical case study on npm and pypi. *Empirical Software Engineering* 25 (2020), 1168–1204.
- [2] Mahmoud AlFadel, Diego Elias Costa, and Emad Shihab. 2023. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering* 28, 3 (2023), 59.
- [3] Veronika Bauer, Lars Heinemann, and Florian Deissenboeck. 2012. A structured approach to assess third-party library usage. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 483–492.
- [4] Ethan Bommarito and Michael J Bommarito II. 2019. An Empirical Analysis of the Python Package Index (PyPI). CoRR abs/1907.11073 (2019). *arXiv preprint arXiv:1907.11073* (2019).
- [5] 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).
- [6] Ton J Cleophas, Aeilko H Zwinderman, Ton J Cleophas, and Aeilko H Zwinderman. 2018. Bayesian Pearson correlation analysis. *Modern Bayesian statistics in clinical research* (2018), 111–118.
- [7] Russ Cox. 2019. Surviving software dependencies. *Commun. ACM* 62, 9 (2019), 36–43.
- [8] Cybersecurity and Infrastructure Security Agency. 2023. *CISA Open Source Software Security Roadmap*. <https://www.cisa.gov/resources-tools/resources/cisa-open-source-software-security-roadmap> Accessed: January 2, 2024.
- [9] 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*. 1–4.
- [10] 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*. 181–191.
- [11] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24 (2019), 381–416.
- [12] Johannes Dösing and Ben Hermann. 2022. Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories. *Digital Threats: Research and Practice* 3, 4 (2022), 1–25.
- [13] Christof Ebert. 2008. Open source software in industry. *IEEE Software* 25, 3 (2008), 52–53.
- [14] Nadia Eghbal. 2020. *Working in public: the making and maintenance of open source software*. Stripe Press.
- [15] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. 2023. An Empirical Study of Malicious Code In PyPI Ecosystem. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 166–177.
- [16] Andrej Hafner, Anže Mur, and Jaka Bernard. 2021. Node package manager’s dependency network robustness. *arXiv preprint arXiv:2110.11695* (2021).
- [17] Abbas Javan Jafari, Diego Elias Costa, Emad Shihab, and Rabe Abdalkareem. 2023. Dependency update strategies and package characteristics. *ACM Transactions on Software Engineering and Methodology* 32, 6 (2023), 1–29.
- [18] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. 2017. Structure and evolution of package dependency networks. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 1025–112 (2017).
- [19] Raula Gaikovina Kula, Coen De Roover, Daniel German, Takashi Ishio, and Katsuro Inoue. 2014. Visualizing the evolution of systems and their library dependencies. In *2014 Second IEEE Working Conference on Software Visualization*. IEEE, 127–136.
- [20] Suhaib Mujahid, Rabe Abdalkareem, and Emad Shihab. 2023. What are the characteristics of highly-selected packages? A case study on the npm ecosystem. *Journal of Systems and Software* 198 (2023), 111588.
- [21] Suhaib Mujahid, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, Mohamed Aymen Saied, and Bram Adams. 2021. Toward using package centrality trend to identify packages in decline. *IEEE Transactions on Engineering Management* (2021).
- [22] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT international symposium on software testing and analysis*. 439–451.
- [23] Yun Peng, Ruida Hu, Ruoke Wang, Cuiyun Gao, Shuqing Li, and Michael R Lyu. 2023. Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem. *arXiv preprint arXiv:2310.12598* (2023).
- [24] Mike Pittenger. 2016. Open source security analysis: The state of open source security in commercial applications. *Black Duck Software, Tech. Rep* (2016).
- [25] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2011. Exploring risks in the usage of third-party libraries. In *of the BELgian-NETHERlands software eVOLUTION seminar*. 31.
- [26] Kristiina Rahkema and Dietmar Pfahl. 2022. SwiftDependencyChecker: Detecting Vulnerable Dependencies Declared Through CocoaPods, Carthage and Swift PM. In *2022 IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 107–111.
- [27] Per Runeson and Martin Höst. 2009. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering* 14, 2 (2009), 131–164.
- [28] Alexandros Tsakpinis. 2023. Analyzing Maintenance Activities of Software Libraries. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*. 313–318.
- [29] Alexandros Tsakpinis and Alexander Pretschner. 2024. Analyzing the Accessibility of GitHub Repositories for PyPI and NPM Libraries. (4 2024). <https://doi.org/10.6084/m9.figshare.25101428>
- [30] 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*. 644–655.
- [31] Ying Wang, Ming Wen, Yeping 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 ACM/IEEE 42nd International Conference on Software Engineering*. 125–135.