The Software Librarian: Python Package Insights for Copilot

Jasmine Latendresse
Data-driven Analysis of Software (DAS) Lab
Concordia University
Montreal, Canada
jasmine.latendresse@mail.concordia.ca

Nawres Day
ISSAT Sousse
Sousse, Tunisia
nawresday121@gmail.com

SayedHassan Khatoonabadi
Concordia University
Montreal, Canada
sayedhassan.khatoonabadi@concordia.ca

Emad Shihab
Data-driven Analysis of Software (DAS) Lab
Concordia University
Montreal, Canada
emad.shihab@concordia.ca

ABSTRACT

Software packages form the backbone of software systems, significantly influencing their functionality, efficiency, and long-term maintainability. As developers increasingly turn to Large Language Models (LLMs) to streamline software development tasks, the ability of these models to accurately recommend suitable packages becomes critical. However, LLMs lack the ability to provide real-time information about package details such as license, dependencies, or even their existence. This can lead to the integration of outdated, incompatible, or legally restrictive packages, which could compromise the software's quality and legal standing. In this paper, we introduce the Software Librarian, a tool that provides real-time information about Python packages recommended as part of the generated code by GitHub Copilot, including license details, deprecation status, and package health. Our tool ensures that the recommended packages are not only valid but are also suitable for integration. To support future research, we have made the Software Librarian available on the Visual Studio Marketplace[1] and released the code online.[2] A demonstration can be viewed at https://youtu.be/hnPr0rvL8lk.

KEYWORDS

Open Source, Machine Learning, Software Development

ACM Reference Format:
Jasmine Latendresse, Nawres Day, SayedHassan Khatoonabadi, and Emad Shihab. 2024. The Software Librarian: Python Package Insights for Copilot. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3639478.3640030

[1]https://marketplace.visualstudio.com/items?itemName=jaslatendresse.software-librarian
[2]https://github.com/jaslatendresse/software-librarian-prod

ICSE-Companion '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0502-1/24/04
https://doi.org/10.1145/3639478.3640030

1 INTRODUCTION

Open source software packages are the cornerstone of modern software development as they significantly reduce development time and effort [1, 6]. However, software packages introduce the concept of dependencies, which are interconnections between code components, increasing complexity and dependency management challenges [3, 5, 7].

Large Language Models (LLMs) emerging as powerful programming assistants [4] that offer potential to simplify and streamline these processes, but they lack real-time critical insights into the packages they import in their code completions, such as license information, deprecation status, or even whether the package still exists [2]. This poses significant risks to software projects, as developers may unknowingly integrate outdated, legally restrictive, or unsupported packages into their codebase. To address these challenges, we introduce the **Software Librarian**, a Visual Studio Code (VS Code)[3] extension that provides developers with real-time analysis of Python packages recommended by GitHub Copilot.[4] Our tool provides insights into the key aspects of packages including license details, dependency data, and deprecation status based on the package's type (i.e., third-party or standard).

To evaluate the Software Librarian, we curated a dataset of third-party and standard packages, as well as packages that do not exist. We focused on the tool's ability to accurately classify these packages into their respective categories- standard, third-party, or invalid. The tool achieved 99% accuracy, which shows its potential to enhance AI-based programming assistants like Copilot. This paper details the tool's design, evaluation, and future improvements to streamline dependency management in the context of LLMs.

2 BACKGROUND

In this section, we introduce key concepts that are necessary to understanding the rest of the paper. To make these concepts more intuitive, we will use the "file directory" analogy, which closely mirrors how packages, modules, and subpackages are organized in Python. We will also define concepts like package aliases, placeholders, and package types.

Package, Module, and Subpackage: In Python, packages and modules are organized like folders and files in a directory:

[3]https://code.visualstudio.com/
[4]https://github.com/features/copilot

- **Package**: A folder that contains a collection of modules and/or subpackages. It is defined by the presence of an `__init__.py` file in the directory.
- **Module**: A single Python file within a package, containing functions, classes, and other Python code.
- **Subpackage**: A package contained within another package, similar to a subfolder within a directory.

For the rest of this paper, a subpackage, module, function, or class, will be referred to as an **importable resource**.

Package Alias: In Python, the name used to import a package in code sometimes differs from the package's distribution name (the name under which the package is published and distributed). This difference is known as an **alias**. For example, the package `opencv-python` is imported in Python as `cv2`. This can lead to confusion, as developers might not know immediately that `cv2` refers to the distribution name `opencv-python`.

Placeholder: A placeholder in the context of code refers to a generic or illustrative name used as an example. They are often used to demonstrate concepts without referring to actual packages, acting as temporary stand-ins meant to be replaced. However, it can be confusing if developers mistakenly interpret a placeholder as an actual library or vice versa, especially in the case where a placeholder also happens to be a published package (e.g., `my_module`).

Package Types: Python packages fall into two broad categories: standard and third-party packages.

- **Standard Packages**: These are packages that come pre-installed with Python and are developed and maintained by the Python core team.
- **Third-Party Packages**: These packages are developed and maintained by the broader open-source community and must be installed separately, typically from PyPi.

3 A WALKTHROUGH OF THE SOFTWARE LIBRARIAN

The Software Librarian is a VS Code extension designed to analyze the software packages generated by GitHub Copilot. Specifically, it assists developers in managing package recommendations made by GitHub Copilot during code completions by verifying their correctness and by providing key metadata such as licensing and maintainability metrics. Figure 1 shows an overview of the Software Librarian, which has two main components: 1) a frontend integrated into the VS Code editor to view the results of the package analysis and 2) a backend for analyzing the package from Copilot code completions.

3.1 Frontend of the Software Librarian

The Software Librarian has a dedicated tab located on the side of the VS Code editor to display the results of a package analysis. Below, we discuss the four main features in the Software Librarian UI.

1. Package Information Display. As shown in Figure 2, each analyzed package appears as a folder-like entry in the Software Librarian tab. These entries feature a dropdown menu that allows the user to expand or collapse the detailed information for each package. When expanded, the dropdown shows the collected metadata for the package.

2. Exporting and Managing Package Information. Users can manage package information with several options. For indivudal packages, they can download the analysis results as a JSON file by clicking the "Download" button located within each package's dropdown, or remove the results by clicking the "Clear" button. For bulk actions, users can download results for all analyzed packages at once by pressing the "Download All" button at the top of the extension tab, or clear all results by selecting the "Clear All" button.

3. Status Bar Indicators and Notifications. When the extension is actively analyzing a package, a progress indicator appears in the status bar of the VS Code editor. Once the analysis is complete, a completion message pops up to inform the user that the results are ready to be viewed in the extension tab.

4. Manual Trigger for Analysis. In addition to the automatic analysis triggered by Copilot code completions, the user can also manually initiate an analysis using the "Analyze" button located in the status bar by highlighting any portion of code and press "Analyze". This triggers an analysis of the packages imported within the highlighted section.

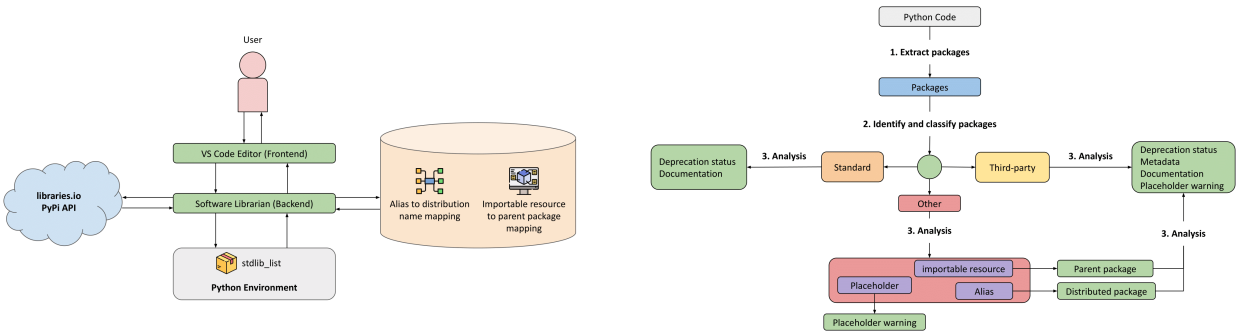
3.2 Backend of the Software Librarian

To provide information on the software packages from GitHub Copilot code completions, the Software Librarian performs three sequential steps as illustrated in Figure 1: 1) obtaining packages from Copilot code completions, 2) package identification and classification, and 3) package analysis. The Software Librarian's backend component receives an import statement from a Copilot code completion and extracts the imported package name. The backend then interacts with the `stdliblist`[5] package and the PyPi API[6] to determine the type of the package (standard Python or third-party) and consequently, the existence of the package. In case the package is a standard or third-party package, the backend interacts with the Python documentation API for metadata on the standard package and the libraries.io API[7] for the third-party package metadata. In case the package is neither standard nor third-party (classified as *other*), the software librarian performs additional analysis to determine the source of the Copilot suggestion and potential explanation for such a case. The results of the analysis are then forwarded to the UI where they are organized for each individual package. We describe each step below.

Step 1: Obtaining packages from Copilot Code Completions. The first step in the backend is to detect and collect packages that are generated in Copilot Code Completions. To achieve this, the Software Librarian monitors the user's interactions with the code editor and detects lines of code that are 1) auto-completed, and 2) contain an `import` statement. When the developer accepts the code completion that includes an `import` statement (triggered by pressing the "Tab" key followed by the "Enter" key to go to the next line), the extension retrieves the `import` statement, extracts the package, and analyzes it.

Step 2: Identifying and Classifying packages. Once an `import` statement is detected, the extension identifies the package and classifies it into one of the types described in Section 2: *standard* or

[5] https://pypi.org/project/stdlib-list/
[6] https://pypi.org/
[7] https://libraries.io/api



(a) Frontend and Backend Interaction **(b) Backend Workflow**

Figure 1: An Overview of the Software Librarian

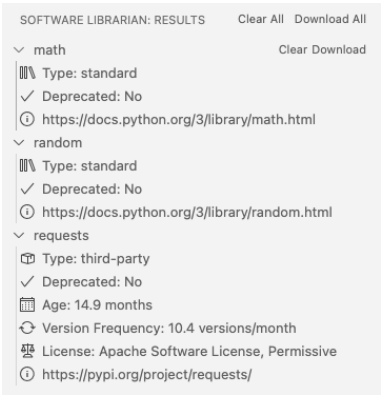


Figure 2: An Example of Output of the Software Librarian for Three Packages.

third-party. To identify *standard* packages, we use the `stdliblist` Python module, which provides a list of all standard packages included in each Python version since 2.6. For packages that are not classified as *standard*, we use the PyPi API to verify whether the package is published in the PyPi repository, in which case it is classified as *third-party*. When a package cannot be classified as either *standard* or *third-party*, it falls under the *other* category, which indicates that the package could potentially lead to installation errors.

Step 3. Package Analysis. The Software Librarian provides different types of information based on the classification of each package into one of the three aforementioned categories (*standard*, *third-party*, or *other*). Below, we discuss how the tool responds for each category.

(1) **Third-Party Package Analysis**: For *third-party* packages, the Software Librarian provides different metadata related to the package's maintenance, legal status, and health.
- **Metadata Retrieval**: The tool retrieves metadata from external sources namely libraries.io API and PyPi. The following metadata is returned for each third-party package:
  - **Number of dependencies**: The number of packages a given package depends on.
  - **Number of dependents**: The number of packages that depend on a given package.
  - **License information**: The type of license associated with a given package.
  - **Version frequency**: How often new versions of a given package are released in a month.
  - **Package age**: How long the package has been available since its first release (in months).
  - **Source rank**: The SourceRank metric of a given package as provided by libraries.io which is an indicator of its general health status.
- **Deprecation Status**: The deprecation status indicates whether a package is outdated or no longer supported in the current version of Python. For this, the Software Librarian creates a temporary virtual environment, installs, and imports the package. Then, it scans logs during the installation and import processes for any deprecation warnings.
- **Documentation**: The tool provides the link to the official PyPi documentation of the package.

(2) **Standard Package Analysis**: For *standard* Python packages, the focus is primarily on checking the deprecation status, as detailed metadata like number of dependencies does not apply in this context.
- **Deprecation Status**: The Software Librarian determines the deprecation status on *standard* packages by querying the official Python documentation. If a package has been renamed in Python 3 (e.g., `urllib2` to `urllib.request`), it is flagged as deprecated. Additionally, if a package is only found in the Python 2 documentation and not in Python 3, it is also marked as deprecated.
- **Documentation**: The tool provides the link to the official Python documentation of the package.

(3) **Other Package Analysis**: The *other* classification is assigned when the package cannot be identified as either *standard* or *third-party*. These packages fall under one of the following subcategories: aliases, importable resources, and placeholders, which are described in Section 2.
- **Alias Detection**: To detect if a package was imported using an alias, the Software Librarian leverages an internal database to map aliases to their corresponding distribution names. The data for this was collected using the

`johnnydep` Python package. Upon identifying an alias, the Software Librarian retrieves its associated distribution package and performs a regular `third-party` analysis. For example, for the alias `cv2`, the Software Librarian would retrieve its distribution `opencv-python`[8] and return its metadata.

- **Importable Resource Mapping**: The Software Librarian checks whether the package is an importable resource (as described in Section 2). To do this, the tool uses an internal database that maps modules, subpackages, and any other item to its parent package. This database is constructed by listing all items defined in the `__init__.py` files of Python packages using the built-in Python `dir()` function. Upon retrieving the parent package, the tool performs a regular *third-party* package analysis. For example, if the tool receives the subpackage `webapp2_extras`, it will retrieve the parent package `webapp2`[9] and return its metadata to the user.
- **Placeholder Detection**: As discussed in Section 2, a placeholder is a generic name that is not meant to represent a real package. To identify placeholders, the Software Librarian uses a predefined list of common placeholder patterns, curated from Stack Overflow, and augmented with data from prompting large language models such as ChatGPT and Gemini (e.g., my_package, module1). If a suspected placeholder corresponds to a real package (e.g., my_module,[10] the extension checks the package's number of downloads. If this number is below 1,000, the extension alerts the developer that, while the package exists, it may have been intended as a placeholder in this context, while still providing the package's metadata.

4 EVALUATION

In this section, we present the evaluation of the Software Librarian's ability to correctly identify and classify Python packages extracted from Copilot code completions. The goal of this evaluation is to assess the tool's accuracy in distinguishing between types of packages, such as *standard*, *third-party*, and packages that we define as *other* (i.e., aliases, importable resources, and placeholders) when they do not fall into either of the first two categories. We did not evaluate the correctness of the metadata retrieved by the Software Librarian since that information is dynamically fetched from external sources in real-time.

To perform our evaluation, we manually curated a dataset consisting of representative mix of 100 packages commonly encountered in Python development, including edge cases that could lead to installation issues. Specifically, the dataset contains 35 third-party packages, 38 standard packages, 16 placeholders, 9 aliases, and 2 importable resources. Then, we ran the Software Librarian on these input packages and measured its performance in accurately identifying each package type. This also includes flagging incorrectly named packages as placeholders, aliases, or importable resources.

[8]https://pypi.org/project/opencv-python/
[9]https://pypi.org/project/webapp2/
[10]https://pypi.org/project/my_module/

The tool achieved a high level of accuracy, with a total of 99 correct package classifications and only one incorrect classification. The single misclassification occurred with the input `webapp2_extras`, which was incorrectly classified as an alias, while in reality, it is a subpackage of the `webapp2` package. This edge case can be linked to the `johnnydep` package, which we used to map package aliases to their distribution names. In certain cases, `johnnydep` lists importable subpackages or components, which may lead to misclassifications like the one observed.

Overall, the Software Librarian achieved an accuracy rate of 99%, which demonstrates that it can reliably identify and classify Python packages correctly.

5 CONCLUSION AND FUTURE WORK

In this paper, we introduced the Software Librarian, a Visual Studio Code extension designed to identify and classify Python packages in Copilot code completions. It also provides developers with critical, real-time information about these packages, including licensing details, the number of dependencies, and the deprecation status. Our tool bridges the gap between the capabilities of LLMs and the practical needs of developers when integrating LLM-generated code into their codebases. Our evaluation demonstrated that the Software Librarian is highly effective in classifying packages, achieving a 99% accuracy rate in distinguishing between standard and third-party packages, aliases, importable resources, and placeholders. This accuracy ensures that developers receive reliable information to make informed decisions about the packages they integrate, reducing the risk of incorporating outdated or legally restrictive packages. In future work, we plan to further refine the logic for distinguishing between aliases and importable resources. One potential enhancement involves accessing the user's workspace to inspect the contents of installed packages and their internal structure directly.

REFERENCES

[1] 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 (2019), 381–416.
[2] Jasmine Latendresse, SayedHassan Khatoonabadi, Ahmad Abdellatif, and Emad Shihab. 2024. Is ChatGPT a Good Software Librarian? An Exploratory Study on the Use of ChatGPT for Software Library Recommendations. *arXiv preprint arXiv:2408.05128* (2024).
[3] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2022. Not All Dependencies are Equal: An Empirical Study on Production Dependencies in NPM. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE'22)*.
[4] Jenny T Liang, Chenyang Yang, and Brad A Myers. 2024. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 1–13.
[5] Suhaib Mujahid. 2021. *Effective Dependency Management for the JavaScript Software Ecosystem.* Ph. D. Dissertation. Concordia University.
[6] Emerson Murphy-Hill, Ciera Jaspan, Caitlin Sadowski, David Shepherd, Michael Phillips, Collin Winter, Andrea Knight, Edward Smith, and Matthew Jorde. 2019. What predicts software developers' productivity? *IEEE Transactions on Software Engineering* 47, 3 (2019), 582–594.
[7] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 35–45.