Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem

Yun Peng
The Chinese University of Hong Kong
Hong Kong, China
ypeng@cse.cuhk.edu.hk

Ruida Hu
Harbin Institute of Technology
Shenzhen, China
200111107@stu.hit.edu.cn

Ruoke Wang
Harbin Institute of Technology
Shenzhen, China
200110930@stu.hit.edu.cn

Cuiyun Gao*
Harbin Institute of Technology
Shenzhen, China
gaocuiyun@hit.edu.cn

Shuqing Li
The Chinese University of Hong Kong
Hong Kong, China
sqli21@cse.cuhk.edu.hk

Michael R. Lyu
The Chinese University of Hong Kong
Hong Kong, China
lyu@cse.cuhk.edu.hk

ABSTRACT

Python is the top popular programming language used in the open-source community, largely owing to the extensive support from diverse third-party libraries within the PyPI ecosystem. Nevertheless, the utilization of third-party libraries can potentially lead to conflicts in dependencies, prompting researchers to develop dependency conflict detectors. Moreover, endeavors have been made to automatically infer dependencies. These approaches focus on version-level checks and inference, based on the assumption that configurations of libraries in the PyPI ecosystem are correct. However, our study reveals that this assumption is not universally valid, and relying solely on version-level checks proves inadequate in ensuring compatible run-time environments.

In this paper, we conduct an empirical study to comprehensively study the configuration issues in the PyPI ecosystem. Specifically, we propose PyConf, a source-level detector, for detecting potential configuration issues. PyConf employs three distinct checks, targeting the setup, packing, and usage stages of libraries, respectively. To evaluate the effectiveness of the current automatic dependency inference approaches, we build a benchmark called VLibs, comprising library releases that pass all three checks of PyConf. We identify 15 kinds of configuration issues and find that 183,864 library releases suffer from potential configuration issues. Remarkably, 68% of these issues can only be detected via the source-level check. Our experiment results show that the most advanced automatic dependency inference approach, PyEGo, can successfully infer dependencies for only 65% of library releases. The primary failures stem from dependency conflicts and the absence of required libraries in the generated configurations. Based on the empirical results, we derive six findings and draw two implications for open-source developers and future research in automatic dependency inference.

*Corresponding author

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639077

ACM Reference Format:
Yun Peng, Ruida Hu, Ruoke Wang, Cuiyun Gao, Shuqing Li, and Michael R. Lyu. 2024. Less is More? An Empirical Study on Configuration Issues in Python PyPI Ecosystem. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639077

1 INTRODUCTION

Python has experienced a remarkable 22.5% year-over-year surge in usage, positioning it as the second most favored programming language within the GitHub open-source community [11]. The popularity of Python is primarily established by its flexible and readable syntax, making it easier for developers to maintain complicated software. Nowadays, the success of Python owes much to its thriving and supportive community, which plays a pivotal role in fostering its prosperity. The accessibility and utility of Python are further amplified by the public libraries available on the Python Package Index (PyPI) platform. With over 470 thousand Python projects and more than 4.7 million releases [9], PyPI serves as the primary repository for numerous third-party libraries. By encapsulating reusable functionalities with APIs in third-party libraries, developers can easily build complicated applications.

In the dynamic ecosystem of third-party libraries hosted on PyPI, multiple releases of the same library are often available, distinguished by version numbers. To use a specific library release, developers must specify both the library's name and the desired version. Utilizing the official library management tool, `pip` [7], for PyPI, developers can effortlessly retrieve and install the intended release based on the associated configurations. Once installed in the current run-time environment, the library release can be accessed through import statements within the source code. Compared with static programming languages such as Java and C/C++, third-party library usage in Python is much simpler and requires no compilation. However, even with this streamlined approach, the presence of any configuration issues in the third-party libraries can lead to potential run-time failures.

Numerous research efforts [1, 24, 28, 39–41] are dedicated to detecting potential dependency conflicts among diverse third-party libraries during the constraint-solving process. As a library may rely on others, a dependency graph can be established to represent the interconnected libraries with nodes and version constraints with edges. Based on the dependency graph, these approaches use

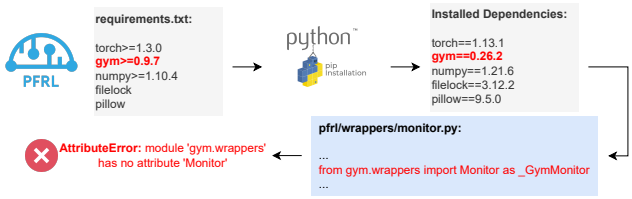

Figure 1: A configuration issue of the third-party library PFRL.

SMT solvers to determine an available version assignment for each library. In addition, some other work [4, 13, 44] develops knowledge graphs for third-party libraries on PyPI and then builds run-time environments for new Python projects based on the knowledge graphs.

The aforementioned version-level approaches have been established and evaluated under the assumption that the configurations of existing Python projects are accurate, as they solely examine version constraints in configurations without inspecting the source code. However, we have discovered instances where this assumption does not hold, and we present an illustrative example in Fig. 1. In this example, the third-party library `PFRL` [10] implements several well-known reinforcement learning algorithms. It records all required third-party libraries in the `requirements.txt` file. During installation, the library manager `pip` resolves the constraints in `requirements.txt` and installs the latest available version for each library. A widely-used library, `gym`, is among the dependencies specified in `requirements.txt`. The configuration for `gym` merely requires a version newer than 0.9.7. As a result, `pip` installs the latest version, 0.26.2, into the project as it satisfies the constraint[1]. Since version 0.26.2 of `gym` does not conflict with other libraries in `requirements.txt`, it passes the regular conflict check and becomes part of the run-time environment. However, when running the code in `PFRL`, an `AttributeError` is raised as the `Monitor` class from `gym.wrappers` in the file `pfrl/wrappers/monitor.py` cannot be found. This issue is widely discussed on `PFRL`'s GitHub issues [30] and Stack Overflow [18]. The root cause is that `gym` removed the `Monitor` class starting from version 0.23.0. Since this change only affects the source code and is not detected by version-level checks, the problem remains unnoticed. This scenario highlights the inadequacy of version-level checks in ensuring the compatibility of source code and run-time environments. To address this problem, the library `gym` should be constrained to versions `gym>=0.9.7, gym<0.23.0`. However, predicting such changes in `gym` during the development of `PFRL` is not feasible since version 0.23.0 of `gym` had not been released at that time. Therefore, the configurations in Python projects can be outdated despite being correct at the release time.

The above-mentioned challenge of version-level dependency checks may pose big threats to the development and evaluation of automatic dependency inference approaches that heavily depend on PyPI library configurations. To address this challenge, we first comprehensively study the potential configuration issues in the PyPI ecosystem (RQ1) and then construct a source-level compatible dataset to facilitate the evaluation of existing automatic dependency inference approaches (RQ2).

To answer RQ1, we introduce PyConf, an automatic approach designed to identify both version-level and source-level configuration issues in third-party libraries on the PyPI platform. PyConf incorporates three distinct checks, namely *Installation Check*, *Dependency Check* and *Import Validation*, to detect configuration issues during the setup stage, packing stage and usage stage of third-party libraries, respectively. Through an analysis of PyConf's results, we identify 183,864 (54%) library releases among the 338,069 checked releases that exhibit potential configuration issues. Notably, 68% of these issues are newly detected by the source-level check, i.e., the *Import Validation*. We identify 15 kinds of configuration issues based on the run-time error types and classify them into three major categories: *Incomplete Configuration*, *Incorrect Configuration* and *Incorrect Code*. For RQ2, we construct a benchmark, VLibs, consisting of 131,720 library releases that successfully pass all three checks implemented by PyConf. We then evaluate the correctness of the inferred run-time environments by the three state-of-the-art automatic dependency inference approaches Pipreqs [26], Dockerizeme [13] and PyEGo [44], respectively.

Key Findings. Based on a thorough analysis of the experiment results pertaining to RQ1 and RQ2, we have summarized the following key findings:

1) Developers tend to provide inadequate configurations for the usage of libraries, especially for Python versions and direct imports in source code.

2) Developers make mistakes in writing configurations since 19% of configuration issues are incorrect configurations. What's more, about 50% incorrect configuration issues can only be detected by *Import Validation*, indicating the importance of source-level validation.

3) Current automatic dependency inference approaches fail to infer about 35% of Python projects. Among the failures, the majority are attributed to dependency conflicts and the absence of required libraries in the generated configurations.

Based on the findings, we conclude two implications for the developers of third-party libraries on the PyPI platform and the future research of automatic dependency inference. Specifically, we find that "*less is more*", i.e., fewer dependency constraints can lead to more configuration errors, so we suggest developers avoid employing open constraints such as `version>1.0`, but set complete and strict dependency constraints limiting the versions of dependencies to the verified ones before the release dates. For future research on automatic dependency inference, we suggest researchers add more conflict checks to avoid generating incorrect configurations.

Contributions. To sum up, we list our contributions as follows.

- To the best of our knowledge, we are the first to study the source-level configuration issues in the PyPI ecosystem systematically.
- We propose an automatic approach PyConf that incorporates *Installation Check*, *Dependency Check* and *Import Validation* to detect configuration issues for Python projects.
- We build a benchmark VLibs that includes 131,720 library releases to facilitate the evaluation of automatic dependency inference approaches.

[1]The installation was performed in July 2023.

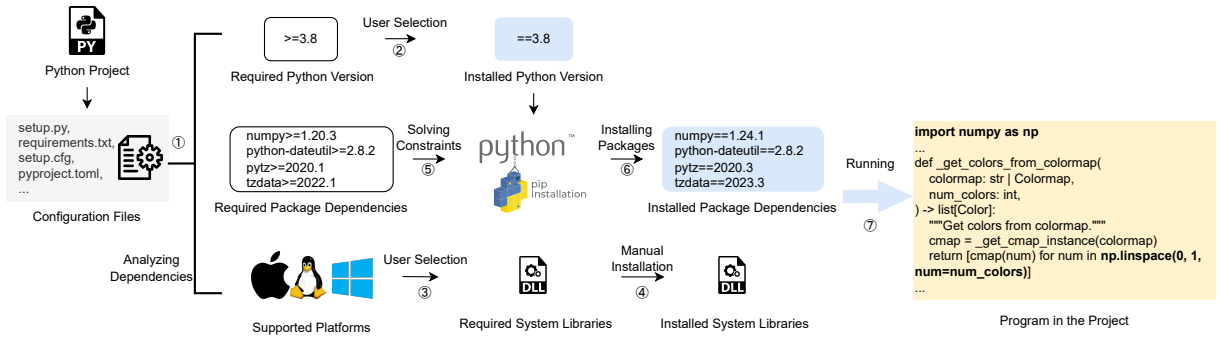

Figure 2: The typical process of run-time environment installation for Python projects.

2 PYTHON RUN-TIME ENVIRONMENT

Python, as an interpreted programming language, offers the advantage of not requiring compilation prior to execution. This attribute facilitates fast prototyping and enables Python programs to be executed on different platforms. However, benefiting from rich support from external libraries, nearly all Python projects depend on multiple third-party or system libraries to avoid redundant implementations of common functionalities. Therefore, it becomes imperative to establish the appropriate run-time environment, comprising all necessary libraries, before running a Python project effectively.

We illustrate the process of installing the run-time environment for a Python project based on its source code in Fig.2. Typically, developers document all project dependencies in configuration files. As the Python community evolves, various configuration formats like `requirements.txt` and `setup.py` have emerged. Additionally, there are diverse developer tools, such as `setuptools`, available to analyze these configuration files (① in Fig.2). The configuration files contain three types of dependencies: 1) the required Python version, 2) the necessary third-party libraries, and 3) the required platform and corresponding system libraries.

In most cases, users must first select the appropriate Python version and platform (② and ③ in Fig.2) before proceeding with the installation of other dependencies. If the Python project relies on some system libraries of the selected platform, users may also need to install them manually (④ in Fig. 2). Since Python third-party libraries are hosted on the PyPI platform [9], Python Software Foundation also provides a dedicated tool named `pip` [7] to facilitate automated installation. `Pip` first resolves the constraints of third-party libraries provided in the configuration file (⑤ in Fig. 2), and then selects the latest valid version for each library (⑥ in Fig. 2). By ensuring the presence of the appropriate Python version, third-party libraries, and system libraries, users can successfully execute certain Python projects (⑦ in Fig. 2).

3 METHODOLOGY

In this section, we introduce how we collect the metadata of PyPI libraries and how PyConf works to detect potential configuration issues.

3.1 Data Preparation

As of July 2023, the PyPI ecosystem boasts a substantial collection of approximately 471,000 libraries, encompassing over 4,712,000 releases [9]. It is quite difficult to perform a comprehensive analysis of all the libraries and their releases on a single machine. To address this challenge, we employ a well-established strategy used in prior studies [4, 13, 44] and collect data from the top 10,000 most popular libraries, as reported by libraries.io [32]. Libraries with only one or two releases, which generally do not necessitate automatic version determination, are excluded from our analysis, resulting in a dataset comprising 8,282 libraries and 338,069 releases[2]. The first column of Table 1 provides statistics on these libraries. In accordance with Python Enhancement Proposal (PEP) 508 [5], names of PyPI libraries are case-insensitive, and distinctions between dash, dot, and underscore are disregarded. To ensure consistency and avoid multiple names for the same library, we normalize all library names to lowercase and replace all dots and underscores with dashes.

Initial Python Version Assignment. As mentioned in Sec. 2, a smooth `pip` installation requires the correct Python version. Hence, we begin by assigning an initial Python version to each library release in the dataset. To acquire the Python version constraints for each library release, we examine the classifiers set by developers on the project web page of the PyPI platform. PyPI offers a set of classifiers for developers to denote the compatibility status of library releases. Among these classifiers, those categorized under the programming language category specify the Python versions with which a library release is compatible. For instance, the developers of library release `pipreqs-0.4.13` add classifier `Python::3.7` in the web page [23], indicating that `pipreqs-0.4.13` can be used in Python version 3.7. By collecting such classifiers from the web pages, we determine the latest Python version applicable to each library release as the initial Python version.

The initial Python versions inferred from classifiers provide relatively reliable insights into developers' intentions regarding library usage. However, setting classifiers is not mandatory when developers publish a new release on PyPI, so we cannot assign initial Python versions for certain library releases lacking appropriate classifiers. To tackle this problem, we collect the release dates of such library releases and select the latest Python version released

[2]Data was collected in November 2022.

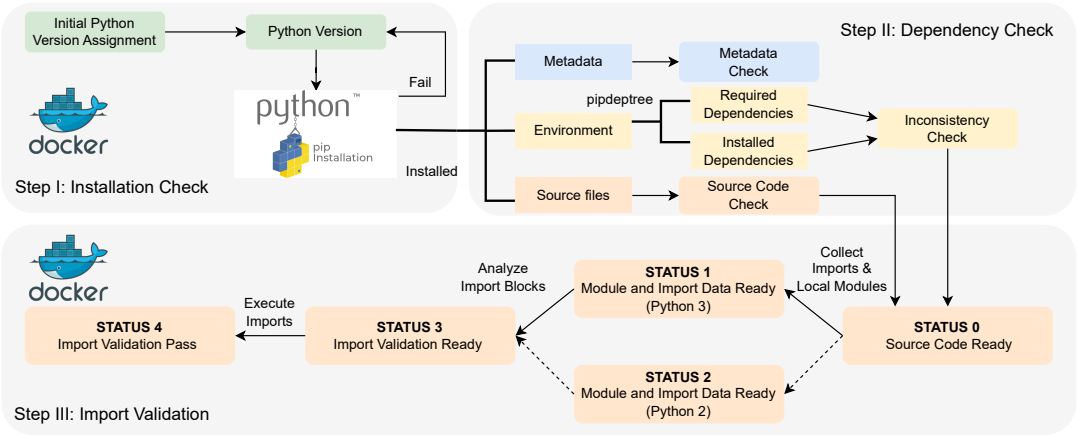

Figure 3: The overview of PyConf.

Table 1: The statistics of the PyPI libraries in our study. "Installed" and "Validated" indicates the libraries passing the Dependency Check and all checks of PyConf, respectively. #Stars indicate the number of GitHub Stars of libraries. #Stars, #Classes, #Functions and #Imports are shown in the format of Avg/Max/Min. The data of #Stars is calculated per library and others are calculated per release. Note that the source code data in the first column is not available as the libraries are not installed.

	All	Installed	Validated (VLibs)
#Libraries	8,282	7,830	5,371
#Releases	338,069	303,377	131,720
#Modules	-	368,304	144,250
#Stars (k)	2.3/159.0/0.0	-	-
#Classes (k)	-	0.3/88.1/0.0	0.2/21.9/0.0
#Functions (k)	-	1.5/261.4/0.0	0.6/50.3/0.0
#External Imports	-	49/2207/0	23/386/0
#Lines of Code (k)	-	18.2/7455.3/0.0	6.5/551.3/0.0

180 days before the release dates of the library releases. This assignment may not be accurate but can be fixed by PyConf when the installation fails.

3.2 PyConf: Detecting Configuration Issues

PyConf checks both version-level and source-level configuration issues for libraries in the PyPI ecosystem. We present the overview of PyConf in Fig. 3. PyConf conducts three checks, namely *Installation Check*, *Dependency Check* and *Import Validation*, to discover potential configuration issues in the setup stage, the packing stage and the usage stage of libraries, respectively. The *Installation Check* verifies the availability of the library releases and detects fatal configuration errors, such as dependency conflicts, that even prevent successful library installation. The *Dependency Check* verifies the consistency of the installed environment with the specified configuration, correctness of the library metadata and syntactic correctness of the source code, which are threatened by mistakes made during the packing stage before a library is published. The *Import Validation* verifies the compatibility of the source code with the installed run-time environment to discover run-time errors during the usage of libraries.

Installation Check. Upon receiving the name and version of a library, PyConf initiates an empty run-time environment within a docker container using the initial Python version. The library release is then installed using the command `pip install <library> == <version>`. However, due to certain initial Python version assignments being estimated based on release dates, and the existence of erroneous configurations authored by developers, some library releases may fail to be installed under the initial Python version. For libraries with Python version constraints, PyConf retries the installation using another valid Python version. For libraries lacking such constraints or failing on all versions indicated by the constraints, PyConf adopts a heuristic searching approach to minimize overhead. Specifically, PyConf first copies the Python version of other successfully installed releases of the same library as different releases of the same library require similar run-time environments. In cases where the installation still fails, PyConf attempts commonly used versions such as 2.7, 3.6, and 3.10. The heuristic searching strategy can handle most installation failures and PyConf resorts to trying all possible Python versions only when the heuristic search proves unsuccessful. Therefore, the installation check fails only when there is no compatible Python version for the given library release or when there are critical errors in applying the configurations provided by developers. This indicates that the library release is not available for use under any Python version.

Dependency Check. During the installation process of a library release via `pip`, three types of data are downloaded into the system:

1) *Metadata.* The metadata is stored in the format of a folder named `<package>-<version>.dist-info`. To analyze this metadata, PyConf focuses on the `top_level.txt` file, which enumerates all modules that can be imported from the library release.

2) *Run-time environment.* PyConf captures information regarding the installed run-time environment, including the versions of

Algorithm 1 Import Block Analysis

```
Input: Abstract Syntax Tree (AST) of the current source file, ast;
Output: Import blocks, B; Block-free Imports, D;
1: function getImportBlocks(block)          ▷ The main function
2:     importBlocks ← {}; subBlocks ← divideBlock(block)
3:     for sb ∈ subBlocks do
4:         curIB ← {}; curBFI ← {}
5:         for node ∈ sb.importnodes do
6:             if isIforTryOutside(node) then
7:                 bNode ← getOutmostIforTryNode(node)
8:                 curIB ← curIB + {getImportBlocks(bNode)}
9:             else
10:                curBFI ← curBFI + {node}
11:            end if
12:        end for
13:        curB ← curIB + {curBFI}
14:        importBlocks ← importBlocks + {curB}
15:    end for
16:    return importBlocks
17: end function
18: blocks ← { }; D ← { }; B ← { }          ▷ The overall algorithm
19: for node ∈ ast.importnodes do
20:    if isIforTryOutside(node) then
21:        blocks ← blocks + {getOutmostIforTryNode(node)}
22:    else
23:        D ← D + {node}
24:    end if
25: end for
26: for b ∈ blocks do
27:    B ← B + {getImportBlocks(b)}
28: end for
```

installed third-party libraries and the version constraints of the required third-party libraries, via the `pipdeptree` [33] tool. PyConf then proceeds to resolve the version constraints of the required third-party libraries and cross-checks them against the installed versions to detect potential inconsistencies.

3) *Source files.* To validate the syntactic correctness of the source code, PyConf locates source folders or files based on the modules collected from the metadata. It employs the `ast` module [8] to parse all source files and identifies the presence of any syntax errors.

Import Validation. Successful installation and consistent runtime environment do not necessarily guarantee the smooth usage of the library, since the execution still fails if some external import requirements in the source code cannot be fulfilled. PyConf conducts *Import Validation* to detect these issues. PyConf leverages a finite state machine (FSM) with four states to guide the process of *Import Validation*, as shown in step III of Fig. 3.

1) Collect Imports and Local Modules (STATUS 0 → STATUS 1/2). Initially, all library releases enter STATUS 0 if PyConf can successfully locate their source code in *Dependency Check*. For library releases with STATUS 0, PyConf collects import statements in the source code. Import statements in the source code can be of two types: *internal imports*, which introduce local modules within the project, and *external imports*, which require third-party libraries from the run-time environment. PyConf employs different approaches to handle the two kinds of import statements.

Local modules are required to distinguish internal imports and external imports. Different source files also have different available local modules. For each source file in the library release, PyConf collects the names of all Python source files and the sub-directories with `__init__.py` file in the same directory, as well as image files such as `.so` and `.pyd`, as local modules. Next, PyConf checks all import statements in the source file and compares the imported module with the local modules to identify internal imports. Since internal imports are not pertinent to the run-time environment, they are excluded from the *Import Validation* process. The remaining import statements are regarded as external imports. PyConf executes external imports in the installed run-time environment to detect potential compatibility issues.

If the above process succeeds under Python 3, the library release enters STATUS 1. Otherwise, PyConf retries the similar process under Python 2 with some small adaptations to Python 2 syntax. The library release analyzed under Python 2 enters STATUS 2.

2) Analyze Import Blocks (STATUS 1/2 → STATUS 3). Developers may handle different run-time environments by utilizing branch statements, such as `if-else` and `try-except`, to wrap the import statements in the code. We term this practice as *multiple version control*. In such scenarios, not all imports are executed during program execution, making it essential to discern whether failures of certain imports indicate configuration issues. To address this challenge, PyConf introduces import block analysis, which effectively categorizes imports under *multiple version control* into import blocks. The main algorithm for import block analysis is detailed in Alg. 1. Additionally, Fig. 4 provides an illustrative example to enhance comprehension of import block analysis.

The import block analysis takes the abstract syntax tree (AST) of the current source file as input and generates two outputs: import blocks B, which are sets of imports grouped based on the branch statements, and block-free imports D, which are import statements unaffected by any branch statements. Specifically, PyConf collects all import nodes present in the AST and verifies whether they are enclosed within branch statement nodes (line 20). Import nodes not associated with branch statements are grouped as block-free imports (line 23). For import nodes associated with branch statements, PyConf identifies the outermost branch statement node to facilitate further analysis (line 21). In the code of Fig. 4, all import statements are included in a `if-else` statement, so there is no block-free import.

To accommodate nested branch statements, such as the `try-except` statement within the true branch of the `if-else` statement in Fig. 4, PyConf adopts a recursive approach (lines 1~17) to handle them, where the branch statement is divided into different blocks based on the branches (line 2). Each block is treated as a new virtual source file, and PyConf recursively gathers the current import blocks and block-free imports for the given branch (lines 3~15). These current import blocks and block-free imports are then consolidated into a larger import block representative of the entire branch. This recursive process continues until all branch statements are effectively handled. The generated import blocks may exhibit nested structures due to this recursive nature. For instance, in Fig. 4, PyConf partitions the `if-else` statement into two blocks, highlighted in

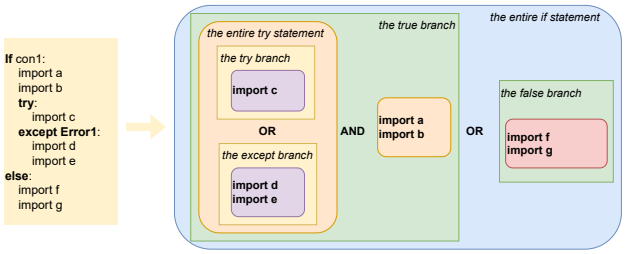

Figure 4: An example of block analysis for external imports.

green. It then recursively handles statements in the two blocks. In the true branch block, PyConf collects all current block-free imports and the `try-except` statement as two sub-blocks, highlighted in orange. The `try-except` block is further processed to different sub-blocks, highlighted in yellow, based on the branch `try` and `except`.

After handling all branch statements, PyConf removes duplicate block-free imports and duplicate imports in the same import block. It then regards all the block-free imports D as a single block and combines it with import blocks B to form the final block. Therefore, there are two relationships "AND" and "OR" between blocks. The "AND" relationship exists between the block of block-free imports and the import blocks, indicating that both blocks will be executed in reality. The "OR" relationship exists between the sub-blocks inside the import block, signifying that only one sub-block will be executed. These relationships are arranged in alternating fashion at different levels of the final block. Initially, block-free imports and import blocks are distinguished, followed by further differentiation of various sub-blocks within an import block. This hierarchy allows the connection of blocks, from the outermost level to the innermost level, using a boolean expression comprising relationships of "AND→OR→AND→OR→ · · ·". Therefore, PyConf facilitates the validation of imports in the source file through the observation of the entire boolean expression. Upon completion of import block analysis, all library releases enter STATUS 3.

3) Execute Imports (STATUS 3 → STATUS 4). Given the boolean expression, PyConf executes the imports one by one and calculates the final value of the expression. Each successful import is regarded as *True* and each failed import is regarded as *False*. The value of a block with no sub-block is *True* only if all the contained imports are *True*. Recognizing the possibility of one run-time error masking another, PyConf executes one import at a time to capture as many run-time errors as possible. Library releases whose boolean expressions for all source files are *True* enter STATUS 4, indicating that they pass the *Import Validation*.

4 EXPERIMENT SETUP

In this section, we introduce the built benchmark VLibs, the baselines in the evaluation and the experiment environment.

Benchmark. We include the 5,371 libraries and their 131,720 releases that pass the three checks of PyConf in our benchmark VLibs. As PyPI libraries themselves are Python projects and have dependencies, verified PyPI libraries can form a good benchmark to evaluate the effectiveness of automatic dependency inference approaches. We show the statistics of VLibs in the last column of Table 1.

Baselines. We select three state-of-the-art automatic dependency inference approaches as our baselines:

Pipreqs [26]: It generates `requirements.txt` files for Python projects based on the import statements in code.

Dockerizeme [13]: It generates `Dockerfile` files for Python projects by scanning the source code. The `Dockerfile` files contain dependencies of the Python version and third-party libraries.

PyEGo [44]: It generates all information required to set up the run-time environments, including the Python version, the third-party libraries and system libraries. It utilizes knowledge graphs to store the information of PyPI libraries and invokes SMT solvers to solve the most proper version for each dependency.

Metric. We use **Pass Rate** to evaluate the performance of automatic dependency inference approaches. Pass Rate is defined as the rate of library releases whose run-time environments inferred by the approach pass all the checks of PyConf.

Environment. To avoid potential attacks on the host machine, PyConf utilizes Docker [16] to install run-time environments. We re-implement all baselines using the replicate packages provided by the authors. We conduct all experiments on a Linux machine (Ubuntu 20.04 LTS) with a 112-core Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz and 256GB memory.

5 RESULT ANALYSIS

5.1 Research Questions

We focus on the following research questions:

- **RQ1:** What are the configuration issues detected by PyConf?
- **RQ2:** How effective are existing automatic dependency inference approaches on VLibs?

To answer RQ1, we run PyConf on the 8,282 libraries and their 338,069 releases, as depicted in the first column of Table 1, to detect configuration issues. During the *Installation Check* and *Import Validation*, PyConf executes the libraries' code, capturing and logging run-time errors like `ImportError` encountered during the execution for analysis. In *Dependency Check*, PyConf collects library releases that violate the pre-defined rules in Sec. 3. To summarize potential configuration issues, we categorize and group the reported run-time errors based on their types. We then review the error messages to identify recurring issue patterns. Regarding RQ2, due to the time-consuming nature of building run-time environments for all baselines using the complete benchmark, we opted to sample 5,000 library releases from VLibs for analysis. To prevent potential bias during sampling, we initially select one release from each library, excluding a few that do not require configurations in VLibs, ensuring representation from all libraries. We then randomly sample the remaining releases to reach a total of 5,000 releases in the sample dataset. We run the three baselines on the sampled dataset and calculate the Pass Rates of the output configurations for each baseline. Moreover, we conduct a comprehensive analysis to identify the primary reasons behind the failure of baselines to provide accurate configurations.

5.2 RQ1: Configuration Issues

Overall Results on Top Popular PyPI Libraries. We present the statistics of libraries that successfully pass the *Dependency Check* and all three checks of PyConf in the second and last columns of Table 1, denoted as *installed libraries* and *validated libraries*, respectively. *Installed libraries*, which are verified by PyConf along with the specified run-time environments, can be correctly set up and are available to users without encountering fatal errors. We observe that there are 7,830 (95%) installed libraries with 303,377 (90%) releases, indicating that the setup configurations of most PyPI libraries are correct. However, the situation becomes less favorable when examining the compatibility of imports in the source code with the specified run-time environments. Only 5,371 (65%) libraries, comprising 131,720 (39%) releases, successfully pass all three checks of PyConf. This indicates that approximately 30% of libraries and 51% of releases on the PyPI platform can be installed but may encounter source-level compatibility problems. Although these issues may not be severe enough to entirely prevent the usage of the library, they can adversely affect specific functionalities.

We categorize the configuration issues identified from the library releases that failed in the three checks of PyConf into three groups: *Incomplete Configuration*, *Incorrect Configuration*, and *Incorrect Code*. All these configuration issues are presented in Table 2. We define a configuration issue as "fatal" if it hinders the usage of the entire library release, and a configuration issue as "not fatal" if it only impacts a portion of the library's functionality. In the rest section of RQ1, we provide a detailed exploration of each configuration issue.

Incomplete Configuration. The issues under this category are raised due to the lack of some important information in the configurations. Specifically, the four issues are classified based on the missing information.

1) Missing configuration files. As mentioned in Sec. 2, most libraries use configuration files such as `requirements.txt` to record the required dependencies. However, PyConf identifies 251 library releases missing necessary configuration files, which directly results in failures in the *Installation Check*. One such instance is the installation failure of *PyAstronomy-0.10.0*, as it requires another library, `numpy`, before the successful execution of `setup.py`. However, the absence of a proper configuration file indicating the dependencies results in the failure to install the library.

2) Missing required libraries for setup. PyConf identifies 3,318 library releases that encounter installation failures due to the absence of libraries required for setup in their configurations. This configuration issue is distinguished by the occurrence of `ModuleNotFoundError` and `ImportError` during the *Installation Check*. For example, in the library release *translators-4.0.4*, a `ModuleNotFoundError` is triggered due to a missing module `requests`. This happens when the installer tries to obtain the version from `__init__.py`, but there are some external imports that are not specified in the `setup_requires` field of `setup.py`.

3) Missing Python versions. PyConf identifies 55,138 library releases that do not indicate the required Python versions in their configurations during *Dependency Check*. The absence of specified Python versions presents significant risks to the reliability of the libraries, as the breaking changes introduced in different Python versions can impact the functionality of the libraries. A notable example is the introduction of new keywords `async` and `await` in Python version 3.5. Identifiers `async` and `await` valid in Python versions < 3.5 become invalid in Python versions > 3.5.

4) Missing required libraries for direct imports. We define **direct imports** as the import statements in the source code of the current library release, and **indirect imports** as the import statements that are called by direct imports in the source code of third-party libraries required in the configurations. PyConf identifies 142,521 library releases where modules required by direct imports are not installed because of missing corresponding library dependencies in the configurations. This issue is characterized by `ModuleNotFoundError` and `ImportError` occurring in direct imports in *Import Validation*. For example, in the library release *claripy-7.8.8.1*, there is an import statement "import celery" in the file `backends/remotetasks.py`. However, the corresponding library *celery* for the module `celery` is not included in the configuration.

Finding 1: Developers tend to provide inadequate configurations for the usage of libraries, especially for Python versions and direct imports in source code.

Incorrect Configuration. The issues under this category are raised due to incorrect information in the configurations. Specifically, eight types of issues are classified based on incorrect information.

1) Dependency conflicts in setup. Dependency conflict in the setup occurs when the dependency constraints of third-party libraries cannot be resolved to valid versions on the PyPI platform. PyConf identifies 6,318 library releases with dependency conflicts during the *Installation Check*, as indicated by the error message "*Could not find a version that satisfies the requirement*". For instance, the library release *accountant-0.0.6* requires `enum>=1.1.5`, but the latest version of *enum* available on the PyPI platform is 0.4.7, which does not satisfy the specified constraint.

2) Incorrect Python versions. For library releases with Python version constraints, PyConf initially selects the latest Python version in the constraint for the installation and retries other Python versions in the constraints if the initial Python version fails. However, PyConf finds 4,155 library releases with Python version constraints but all the Python versions in the constraints fail in *Installation Check*. This suggests that the Python version constraints written by developers for these library releases are incorrect.

3) Other run-time errors in setup. In addition to dependency conflicts and Python version issues, PyConf identifies two types of run-time errors occurring during the setup process. Specifically, there are 966 library releases associated with `AttributeError` and 2,498 library releases associated with `FileNotFoundError` in the *Installation Check*. The `AttributeError` is caused by incorrect setup dependencies, while the `FileNotFoundError` is a result of some non-configuration files being absent. As an example, the library release *aiodocker-0.1* requires `README.md`, but it does not exist.

4) Inconsistent configurations with metadata. PyConf checks the potential inconsistencies between the configurations and the library metadata. It identifies 592 library releases with such inconsistencies. The inconsistencies primarily result from the naming errors of files

Table 2: Configuration issues detected by PyConf. There may be multiple issues occurring in one release.

Category	Issue	Check	#Releases	Fatal?	Possible Reasons
Incomplete Configuration	Missing configuration files	Installation Check	251	✔	Missing required information
	Missing required libraries for setup	Installation Check	3,318	✔	
	Missing Python versions	Dependency Check	55,138	✘	
	Missing required libraries for direct imports	Import Validation	142,521	✘	
Incorrect Configuration	Dependency conflicts in setup	Installation Check	6,318	✔	Unsolvable constraints
	Incorrect Python versions	Installation Check	4,155	✔	Incorrect dependencies
	Other run-time Errors in setup	Installation Check	3,464	✔	Missing files
	Inconsistent configurations with metadata	Dependency Check	592	✘	Naming error
	Inconsistent version numbers with release dates	Dependency Check	12,018	✘	Confusing version orders
	Missing required modules for indirect imports	Import Validation	11,023	✘	Incorrect dependencies
	Inconsistent modules in direct imports with installed depenencies	Import Validation	6,678	✘	
	Other run-time Errors in imports	Import Validation	8,178	✘	
Incorrect Code	Missing source code	Dependency Check	2,588	✔	Creating placeholders
	Parsing error	Dependency Check	431	✔	Invalid syntax/encoding
	Multiple version control failure	Import Validation	15,507	✘	Incorrect dependencies

or folders. For example, the metadata folder in the library release *kfp-0.1.23* is named `kfp-0.1.22.dist-info`.

5) Inconsistent version numbers with release dates. When resolving version constraints of third-party libraries, `pip` installs the latest versions that meet the constraints. The selection of the latest version is determined by comparing the version number strings. However, we have discovered cases where the version number order does not align with the release date order. For example, the library *multipart* released version 2.0 in 2019 and version 0.1.1 in 2020. Developers who used this library in 2019 expected that future versions would be greater than 2.0 and thus set the constraint `multipart<0.2`. However, `pip` still considers version 0.1.1 as valid for this constraint, leading to the selection of an unexpected version. As a result, the inconsistency between the version number order and the release date order can undermine the validity of constraints set by developers. In our analysis, PyConf identifies 12,018 library releases that depend on third-party libraries with this issue.

Finding 2: Inconsistencies between version number order and release date order are prevalent in the PyPI ecosystem, undermining the validity of developers' dependency constraints.

6) Missing required modules for indirect imports. PyConf identifies 11,023 library releases where modules in indirect imports are not installed, as indicated by `ModuleNotFoundError` and `ImportError` in *Import Validation*. This issue arises due to two possible reasons.

Firstly, the required third-party libraries may not properly handle their own dependencies. For instance, the library release *keras-bert-0.10.0* requires *keras* in the configuration and has an import statement "`import keras.backend`". However, when importing `keras.backend`, *tensorflow* is also required, but *keras* does not list it as a dependency in its configuration, resulting in import failure.

Secondly, incorrect dependencies for third-party libraries in the configurations may be the cause. For example, the library release *replit-1.4.0* has an external import statement "`import flask`", which, in turn, includes an import statement "`from markupsafe import soft_unicode`". However, `soft_unicode` is removed starting from version 2.1.0 of `markupsafe`, and there is no constraint preventing `pip` from getting the latest version of `markupsafe`, leading to the import failure.

Finding 3: Ignoring indirect dependencies is one of the major (~ 18%) incorrect configuration issues, indicating that developers

often ignore indirect dependencies and only focus on the modules directly used in the source code.

7) Inconsistent modules in direct imports with installed dependencies. PyConf identifies 6,678 library releases where modules in direct imports have corresponding library dependencies in the configurations but fail to be imported. This issue arises because `pip` automatically acquires the latest available version of the required libraries, which may lead to the exclusion of certain required modules in the direct imports if they have been removed in the latest version. A prime example of this is the library *jtskit*, which is deprecated, and its developers create an empty release 0.5.0 to install another library *jsontableschema*, resulting in the failure of the direct import "import jtskit".

8) Other run-time errors in imports. We include all other run-time errors in this case. PyConf identifies 8,178 library releases with run-time errors other than `ModuleNotFoundError` and `ImportError` in the *Import Validation*, which include `TypeError`, `ValueError`, and so on. These run-time errors are induced by the execution of global statements in the module, resulting in the failure of the module import. For instance, in the library release *pandas-market-calendars-1.6.0*, there is an import statement "import trading_calendars". Upon executing this import, a global statement "NP_NAT = np.array([pd.NaT], dtype=np.int64)[0]" in the module `trading_calendars` leads to a `TypeError` due to the use of `int()` on `NaTType` type.

Finding 4: Developers make mistakes in writing configurations since 19% of configuration issues are incorrect configurations. What's more, about 50% incorrect configuration issues can only be detected by *Import Validation*, indicating the importance of source-level validation.

Incorrect Code. The issues under this category are raised due to the incorrect source code. Specifically, there are three cases classified based on source code errors.

1) Missing source code. PyConf identifies 2,588 library releases whose source code cannot be located in *Dependency Check*. PyConf cannot further validate the import statements without the source code. One possible reason we observed through manual analysis is that some library releases are published as placeholders on the PyPI platform without any actual source code. For example, the library release `mypy-protobuf-1.0` contains no source code and the `top_level.txt` file in it indicates there is no available module in the library.

2) Parsing error. PyConf identifies 431 library releases with Python files that cannot be parsed due to syntax errors, such as incorrect use of semicolons in Python code and encoding errors. Libraries with parsing errors cannot be handled by the Python interpreter, rendering them infeasible to be imported and used by users.

3) Multiple version control failure. In Sec. 3, PyConf conducts import block analysis to partition import statements in different branches into separate blocks and generate boolean expressions to validate the correctness of imports. We identify 15,507 library releases whose generated boolean expressions evaluate to *False* in *Import Validation*, indicating that none of the branches in the branch statements successfully handle the specified run-time environments. We refrain from analyzing the run-time errors in individual branches as they may not be executed in practice. Instead, we collectively refer to these cases as "multiple version control failures," highlighting the incompatibilities between the version control in the source code and the actual run-time environments.

Table 3: The Pass Rates (%) of three baselines on the sampled 5,000 releases from our benchmark.

Python Version?	Pipreqs	Dockerizeme	PyEGo
✔	52.9	26.6	60.7
✘	-	23.2	65.0

Finding 5: Incorrect configurations can hardly be handled by the multiple version control logic in source code, as there are 5% of library releases suffering from multiple version control failures.

5.3 RQ2: Effectiveness of Automatic Dependency Inference Approaches

We evaluate the configurations provided by three baselines in two different settings, considering that the baseline Pipreqs does not output Python versions. In the first setting, we utilize the validated Python versions obtained in RQ1 and rely solely on the third-party library dependencies provided by the baselines to build run-time environments. In the second setting, we do not provide the validated Python versions and use those supplied by the baselines for building run-time environments. Table 3 presents the Pass Rates of the three baselines under these two settings. Notably, PyEGo achieves the highest Pass Rate of 65.0% when using its own inferred Python versions. This suggests that approximately 35% of library releases cannot be successfully inferred by PyEGo. On the other hand, for Pipreqs and Dockerizeme, their performance is limited, covering only 20% to 50% of library releases, despite the slight improvement when provided with the correct Python versions.

To investigate the primary reasons behind the failures of the three baselines in inferring correct dependencies, we present the major issues with at least 50 occurrences (>1%) during the check process of PyConf in Table 4. Surprisingly, we find that for Pipreqs and PyEGo, approximately 68% and 51% of the failures, respectively, come from dependency conflicts during setup. This suggests that some dependencies provided by these baselines are not valid on the PyPI platform. Since these baselines rely on import statements to determine which libraries should be included in the configurations, there are instances where local modules share names with third-party modules or different libraries share module names, confusing their inference. In the case of Dockerizeme, around 86% of the failures arise from missing required libraries for direct imports. This issue is also the second most common cause of failures for Pipreqs and PyEGo. One possible explanation is that the baseline databases cannot cover all libraries. For instance, PyEGo's database only includes the top 10,000 popular PyPI libraries [44], while PyPI hosts over 471 thousand libraries. Regarding the other three issues

Table 4: The issues that three baselines fail to pass the checks of PyConf when we provide the Python versions. Only issues with more than 50 occurrences are included.

Issue	Pipreqs	Dockerizeme	PyEGo
Missing required libraries for setup	71	168	13
Missing required libraries for direct imports	589	3,099	597
Dependency conflicts in setup	1,675	310	823
Missing required modules for indirect imports	14	20	147
Multiple version control failure	124	15	24

in Table 4, we observe that they only frequently occur in one specific baseline, suggesting that they might arise from inappropriate designs in that particular baseline's approach.

Finding 6: Current automatic dependency inference approaches fail to infer about 35% of Python projects. Most failures come from dependency conflicts and the absence of required libraries in the generated configurations.

6 IMPLICATIONS

Fewer dependency constraints lead to more configuration issues. "Less is More" seems to be a widely-used strategy to cut costs in software development. However, our findings from RQ1 reveal that 74% of configuration issues arise from insufficient dependency constraints. While these constraints may be valid and correct during the initial release of third-party libraries, they can become outdated over time as dependencies evolve. Therefore, run-time errors may occur when using certain functionalities, which cannot be detected during the setup process of run-time environments. As a result, these issues are challenging to detect without a comprehensive evaluation of the source code. Fortunately, the resolution for these issues is relatively straightforward – by adding more strict dependency constraints. We advise third-party library developers to avoid setting open constraints like `version>1.0`. Instead, they should opt for complete and strict dependency constraints that restrict Python versions and library dependencies to the verified versions at the time of release. By doing so, developers can enhance the reliability of their libraries and mitigate potential run-time errors caused by evolving dependencies.

Fewer conflict checks result in more dependency inference failures. During our analysis of why the three baselines fail to infer correct configurations, we have identified two major issues. First, some required libraries are missing, which can be resolved by updating the databases to align with the PyPI ecosystem. Second, we have observed dependency conflicts in the generated configurations. It indicates that the baselines lack sufficient conflict checks to validate the generated configurations thoroughly. For example, they do not handle the potential conflicts between the local modules inside the project and the external modules from the PyPI platform. Therefore, we recommend that future research on automatic dependency inference should incorporate more extensive conflict checks between local projects and libraries on the PyPI platform.

7 THREATS TO VALIDITY

The experiments and conclusions in our paper may face the following threats.

Threats to Internal Validity. During the *Installation check* of PyConf, we encountered some library releases that could not be installed due to timeout errors or downloading errors. These issues are primarily caused by unstable network connections. Additionally, a few cases involved extremely large libraries (>1GB) that exceeded the 600-second time limit for handling by dockers. To mitigate the impacts of this threat, we retried the installation of the library releases that failed due to network issues. For libraries encountering timeout errors, we extended the timeout limit of dockers from 600 seconds to 1,200 seconds. These methods reduce the number of failed library releases to about 22,485. However, due to limited time and competing resources, we could not address the problems for all library releases in a short timeframe. Therefore, we did not classify these library releases as having configuration issues to ensure that all configuration issues discussed in RQ1 are supported by solid and direct empirical evidence.

Threats to External Validity. During the preparation of the dataset for PyConf, we select 10,000 third-party libraries from the PyPI platform. This selection is necessary as it is infeasible to check all libraries on PyPI. Additionally, we sample 5,000 library releases to evaluate the effectiveness of existing automatic dependency inference approaches. The process of data selection could potentially impact the generality of our experimental results and findings. To minimize this impact, we select the 10,000 most popular libraries that are well-maintained and widely recognized in the Python community as the dataset by following work [4, 13, 44], ensuring that our study is significant as the top 10,000 libraries have a substantial influence on the PyPI ecosystem. When sampling library releases for the evaluation of current dependency inference approaches, we make the sampled releases cover all verified libraries in VLibs. By doing so, we guarantee that these approaches are evaluated in diverse libraries with various functionalities rather than on different releases of the same libraries.

8 RELATED WORK

8.1 Software Ecosystem

For the Python software ecosystem, Valiev *et al.* [34] study the ecosystem-level factors impacting the sustainability of Python projects. Bommarito *et al.* [2] conduct an empirical analysis on the Pypi ecosystem. Chen *et al.* [3] and Peng *et al.* [25] analyze the language features of Python projects. Vu *et al.* [35] identify the typosquatting and combosquatting attacks on the Python ecosystem. In this paper, we focus on the configuration issues in the PyPI ecosystem. For software ecosystems of other programming languages, Serebrenik *et al.* [27] study different tasks in the software ecosystem and identify six types of challenges. Mens [21]

study software ecosystem on the aspect of software maintenance and evolution. Lertwittayatrai *et al.* [19] study the topology of the JavaScript package ecosystem. Zimmermann *et al.* [45] study the security threats in the npm [17] ecosystem. There was also a lot of work [6, 20, 22, 31] studying configuration problems of software ecosystems.

8.2 Dependency Inference

There are a lot of efforts [15, 38] being devoted to automatically inferring environment dependencies for software. Most recently, DockerizeMe [13] infers third-party and system libraries via static analysis and dynamic analysis. V2 [14] enhances DockerizeMe and explores possible environment dependencies based on feedback-directed search. Pipreqs [26] builds the *requirements.txt* files for Python projects by analyzing the *import* statements in code. SnifferDog [37] builds the execution environments for Python Jupyter notebooks. PyEGo [44] and PyCRE [4] utilize knowledge graphs to represent and analyze the dependencies between the third-party packages used by Python programs.

8.3 Dependency Conflict Detection

To improve the reliability of software, some researchers work on detecting potential dependency conflicts of software. Artho *et al.* [1] conduct a case study for conflict defects on software packages. Patra *et al.* [24] propose to detect the dependency conflicts between JavaScript libraries. Soto-Valero *et al.* [28] study the problem of multiple versions of the same library co-existing in Maven Central. LibHarmo [15] detects library version inconsistencies for Java Maven projects. Wang *et al.* [38–42] conduct a series of empirical analyses and develop several tools to facilitate dependency conflict issue diagnosis for the ecosystem of different programming languages. These approaches focus on version-level checks while PyConf conducts source-level checks by validating the import statements in source code.

There are also some research efforts on repairing dependency conflict issues. Su *et al.* [29] propose to repair the inconsistencies between file systems and configuration scripts. Weiss *et al.* [43] capture and replay developer changes to repair the system configuration. HireBuild [12] repairs failing gradle build scripts based on the patterns from TravisTorrent dataset. SmartPip [36] proposes to address the efficiency problem of previous approaches on the PyPI [9] ecosystem.

9 CONCLUSION

In this paper, we conduct an empirical study on configuration issues in the PyPI ecosystem. We propose PyConf to automatically identify configuration issues in the setup stage, the packing stage and the usage stage of third-party libraries. We also build a benchmark VLibs for the evaluation of automatic dependency inference approaches. We discover six findings and conclude two implications to facilitate the development of third-party libraries and future research on automatic dependency inference.

10 DATA AVAILABILITY

The proposed tool PyConf and benchmark VLibs are released at https://github.com/JohnnyPeng18/PyConf.

11 ACKNOWLEDGEMENT

The authors would like to thank the efforts made by anonymous reviewers. The work described in this paper was supported by the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14206921 of the General Research Fund). The work was also supported by National Natural Science Foundation of China under project (No. 62002084), Natural Science Foundation of Guangdong Province (Project No. 2023A1515011959), Shenzhen Basic Research (General Project No. JCYJ20220531095214031), Shenzhen International Cooperation Project (No. GJHZ20220913143008015), and Key Program of Fundamental Research from Shenzhen Science and Technology Innovation Commission (Project No. JCYJ20200109113403826). Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the above sponsoring entities.

REFERENCES

[1] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why do software packages conflict?. In *9th IEEE Working Conference of Mining Software Repositories, MSR 2012, June 2-3, 2012, Zurich, Switzerland*, Michele Lanza, Massimiliano Di Penta, and Tao Xie (Eds.). IEEE Computer Society, 141–150. https://doi.org/10.1109/MSR.2012.6224274

[2] Ethan Bommarito and Michael J. Bommarito II. 2019. An Empirical Analysis of the Python Package Index (PyPI). *CoRR* abs/1907.11073 (2019). arXiv:1907.11073 http://arxiv.org/abs/1907.11073

[3] Zhifei Chen, Yanhui Li, Bihuan Chen, Wanwangying Ma, Lin Chen, and Baowen Xu. 2020. An Empirical Study on Dynamic Typing Related Practices in Python Systems. In *ICPC '20: 28th International Conference on Program Comprehension, Seoul, Republic of Korea, July 13-15, 2020*. ACM, 83–93. https://doi.org/10.1145/3387904.3389253

[4] Wei Cheng, Xiangrong Zhu, and Wei Hu. 2022. Conflict-aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 451–461. https://doi.org/10.1145/3510003.3510078

[5] Robert Collins. 2015. PEP 508 – Dependency specification for Python Software Packages. https://peps.python.org/pep-0508/ https://peps.python.org/pep-0508/.

[6] Clemens Dubslaff, Kallistos Weis, Christel Baier, and Sven Apel. 2022. Causality in Configurable Software Systems. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 325–337. https://doi.org/10.1145/3510003.3510200

[7] Python Software Foundation. 2023. The pip tool. https://pypi.org/project/pip/ https://pypi.org/project/pip/.

[8] Python Software Foundation. 2023. The Python AST module. https://docs.python.org/3/library/ast.html https://docs.python.org/3/library/ast.html.

[9] Python Software Foundation. 2023. The Python Package Index. https://pypi.org/ https://pypi.org/.

[10] Yasuhiro Fujita, Prabhat Nagarajan, Toshiki Kataoka, and Takahiro Ishikawa. 2023. PFRL: a PyTorch-based deep reinforcement learning library. https://github.com/pfnet/pfrl https://github.com/pfnet/pfrl.

[11] Inc. GitHub. 2022. GitHub Octoverse report on programming languages. https://octoverse.github.com/2022/top-programming-languages https://octoverse.github.com/2022/top-programming-languages.

[12] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: an automatic approach to history-driven repair of build scripts. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 1078–1089. https://doi.org/10.1145/3180155.3180181

[13] Eric Horton and Chris Parnin. 2019. DockerizeMe: automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 328–338. https://doi.org/10.1109/ICSE.2019.00047

[14] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 477–488. https://doi.org/10.1109/ASE.2019.00052

[15] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *ESEC/FSE*

'20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 518–529. https://doi.org/10.1145/3368089.3409689

[16] Docker Inc. 2023. Docker. https://www.docker.com/ https://www.docker.com/.

[17] NPM Inc. 2023. NPM. https://www.npmjs.com/ https://www.npmjs.com/.

[18] learner. 2022. how to solve module 'gym.wrappers' has no attribute 'Monitor'? https://stackoverflow.com/questions/71411045/how-to-solve-module-gym-wrappers-has-no-attribute-monitor https://stackoverflow.com/questions/71411045/how-to-solve-module-gym-wrappers-has-no-attribute-monitor.

[19] Nuttapon Lertwittayatrai, Raula Gaikovina Kula, Saya Onoue, Hideaki Hata, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. 2017. Extracting Insights from the Topology of the JavaScript Package Ecosystem. In *24th Asia-Pacific Software Engineering Conference, APSEC 2017, Nanjing, China, December 4-8, 2017*, Jian Lv, He Jason Zhang, Mike Hinchey, and Xiao Liu (Eds.). IEEE Computer Society, 298–307. https://doi.org/10.1109/APSEC.2017.36

[20] Jens Meinicke, Chu-Pan Wong, Bogdan Vasilescu, and Christian Kästner. 2020. Exploring Differences and Commonalities between Feature Flags and Configuration Options. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice* (Seoul, South Korea) *(ICSE-SEIP '20)*. Association for Computing Machinery, New York, NY, USA, 233–242. https://doi.org/10.1145/3377813.3381366

[21] Tom Mens. 2016. An Ecosystemic and Socio-Technical View on Software Maintenance and Evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, Raleigh, NC, USA, October 2-7, 2016*. IEEE Computer Society, 1–8. https://doi.org/10.1109/ICSME.2016.19

[22] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2014. Mining Configuration Constraints: Static Analyses and Empirical Results. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 140–151. https://doi.org/10.1145/2568225.2568283

[23] The open source community. 2023. The PyPI web page of library pipreqs 0.4.13. https://pypi.org/project/pipreqs/ https://pypi.org/project/pipreqs/.

[24] Jibesh Patra, Pooja N. Dixit, and Michael Pradel. 2018. ConflictJS: finding and understanding conflicts between JavaScript libraries. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 741–751. https://doi.org/10.1145/3180155.3180184

[25] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 24–35. https://doi.org/10.1109/SANER50967.2021.00012

[26] pipreqs. 2023. pipreqs. https://github.com/bndr/pipreqs.

[27] Alexander Serebrenik and Tom Mens. 2015. Challenges in Software Ecosystems Research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops, Dubrovnik/Cavtat, Croatia, September 7-11, 2015*, Ivica Crnkovic (Ed.). ACM, 40:1–40:6. https://doi.org/10.1145/2797433.2797475

[28] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 333–343. https://doi.org/10.1109/MSR.2019.00059

[29] Ya-Yunn Su, Mona Attariyan, and Jason Flinn. 2007. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, Thomas C. Bressoud and M. Frans Kaashoek (Eds.). ACM, 237–250. https://doi.org/10.1145/1294261.1294284

[30] ThePeshMod. 2022. "from gym.wrappers import Monitor" has been deprecated. https://github.com/pfnet/pfrl/issues/172 https://github.com/pfnet/pfrl/issues/172.

[31] Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. 2014. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Comput. Surv.* 47, 1, Article 6 (jun 2014), 45 pages. https://doi.org/10.1145/2580950

[32] Inc Tidelift. 2023. Libraries.io - The Open Source Discovery Service. https://libraries.io/ https://libraries.io/.

[33] Open Source Tool. 2023. pipdeptree. https://github.com/tox-dev/pipdeptree https://github.com/tox-dev/pipdeptree.

[34] Marat Valiev, Bogdan Vasilescu, and James D. Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: a case study of the PyPI ecosystem. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 644–655. https://doi.org/10.1145/3236024.3236062

[35] Duc Ly Vu, Ivan Pashchenko, Fabio Massacci, Henrik Plate, and Antonino Sabetta. 2020. Typosquatting and Combosquatting Attacks on the Python Ecosystem. In *IEEE European Symposium on Security and Privacy Workshops, EuroS&P Workshops 2020, Genoa, Italy, September 7-11, 2020*. IEEE, 509–514. https://doi.org/10.1109/EuroSPW51379.2020.00074

[36] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. 2022. smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 93:1–93:12. https://doi.org/10.1145/3551349.3560437

[37] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1622–1633. https://doi.org/10.1109/ICSE43902.2021.00144

[38] Ying Wang, Liang Qiao, Chang Xu, Yepang Liu, Shing-Chi Cheung, Na Meng, Hai Yu, and Zhiliang Zhu. 2021. HERO: On the Chaos When PATH Meets Modules. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 99–111. https://doi.org/10.1109/ICSE43902.2021.00022

[39] 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 *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 125–135. https://doi.org/10.1145/3377811.3380426

[40] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 319–330. https://doi.org/10.1145/3236024.3236056

[41] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 572–583. https://doi.org/10.1109/ICSE.2019.00068

[42] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Trans. Software Eng.* 48, 7 (2022), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767

[43] Aaron Weiss, Arjun Guha, and Yuriy Brun. 2017. Tortoise: interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 625–636. https://doi.org/10.1109/ASE.2017.8115673

[44] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1245–1256. https://doi.org/10.1145/3510003.3510127

[45] Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, Nadia Heninger and Patrick Traynor (Eds.). USENIX Association, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman