Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints

Huiyan Wang*
State Key Lab. for Novel Software Technology
Nanjing University
Nanjing, China
why@nju.edu.cn

Shuguan Liu
State Key Lab. for Novel Software Technology
Nanjing University
Nanjing, China
liu_shuguan@126.com

Lingyu Zhang
State Key Lab. for Novel Software Technology
Nanjing University
Nanjing, China
zly@smail.nju.edu.cn

Chang Xu*
State Key Lab. for Novel Software Technology
Nanjing University
Nanjing, China
changxu@nju.edu.cn

ABSTRACT

Python projects grow quickly by code reuse and building automation based on third-party libraries. However, the version constraints associated with these libraries are prone to mal-configuration, and this forms a major obstacle to correct project building (known as *dependency-conflict (DC) building failure*). Our empirical findings suggest that such mal-configured version constraints were mainly prepared manually, and could essentially be refined for better quality to improve the chance of successful project building. We propose a LooCo approach to refining Python projects' library version constraints by automatically loosening them to maximize their solutions, while keeping the libraries to observe their original behaviors. Our experimental results with real-life Python projects report that LooCo could efficiently refine library version constraints (0.4s per version loosening) by effective loosening (5.5 new versions expanded on average) automatically, and transform 54.8% originally unsolvable cases into solvable ones (i.e., successful building) and significantly increase solutions (21 more on average) for originally solvable cases.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Dependency conflict, version constraint, loosening resolution

ACM Reference Format:
Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23), December 3–9, 2023, San Francisco, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3611643.3616264

1 INTRODUCTION

Python projects depend extensively on third-party libraries for resource and code reuse, and have gained increasing popularities in recent years. Until Aug 2023, there are over four million release versions of libraries in PyPI [17], a well-known centralized Python repository of third-party libraries. To build Python projects anywhere and anytime, developers write configuration scripts (e.g., `setup.py` or `requirements.txt`) with version constraints for dependent third-party libraries, each of which specifies expected versions for a specific library (e.g., `numpy>=1.8`, meaning all versions equal to, or over, `1.8`). When such version constraints are properly specified, a Python project can be restored or built with a compatible execution environment via importing those third-party libraries satisfying these constraints, which can be done in a recursive way by any Python library installer, e.g., `Pip` [14].

However, Python projects and their dependent third-party libraries are developed independently, and this causes developers to have to write version constraints manually without proper guidance. Besides, the quick growth of third-party libraries requires developers to spend non-trivial efforts [29, 41, 42] on keeping these constraints up-to-date along with library evolution. When failed to do so, the version constraints can conflict with each other and the concerned projects would result in building failures (named *dependency-conflict building failure*, or *DC building failure* for short [50, 53]) due to the lack of a compatible execution environment [36, 47]. Fig. 1 gives a real building failure example from BugInPy [47, 54]. The failure steams from the fact that a project installing library `Clifford==1.3.1` requires a `llvmlite` library version satisfying constraint (`<0.36`, `>=0.35.0`) to be installed first (transitively required by `numba>0.46`), but this installation conflicts with the project's another constraint on library `llvmlite==0.32.1`.

Existing work has investigated into this problem and proposed various ways to address the concerned DC issues for Python projects. For example, Wang et al. [53] proposed building dependency graphs for Python projects to detect DC issues. Ye et al. [56] and Cheng

[*] Corresponding authors.

ESEC/FSE '23, December 3–9, 2023, San Francisco, CA, USA

ACM ISBN 979-8-4007-0327-0/23/12…$15.00
https://doi.org/10.1145/3611643.3616264

```
Clifford==1.3.1
----numba>0.46 (from Clifford==1.3.1)
--------llvmlite<0.36,>=0.35.0 (from numba>0.46->Clifford==1.3.1)
llvmlite==0.32.1----**ERROR**: numba 0.52.0 has requirement
llvmlite<0.36,>=0.35.0, but you'll have llvmlite==0.32.1 which is incompatible.
```

Figure 1: Building failure #pandas_106_0 in BugsInPy [47, 54]

et al. [27] optimized detecting DC issues by searching constraints from building necessities besides explicit libraries (e.g., Python interpreter and system libraries). This line of work focuses on detecting and reporting Python DC issues. Another relevant line of work emphasizes alleviating DC building failures by improving library-installing strategies. For example, Pip (after 20.3) presented a backtracking strategy [12] to better restore a building process when failed; sMARTPIP [50] further optimized this process by controlling the backtracking cost with the aid of a dedicated constructed PyPI dependency database. However, all these efforts assume sticking to original library version constraints, and can unfortunately gain still limited benefits in resolving originally failed project building (e.g., completely helpless to 30% originally-unsolvable failures [50]).

We hence dig into Python's project building process to see how one can substantially resolve such DC building failures. Our in-depth study discloses that some failures are indeed inevitable since their associated library version constraints are themselves unsolvable (i.e., no solution satisfying all constraints). We refer to such failures as *type-A failure* (or *A-failure* for short), which have to be resolved by rewriting constraints. We also observe that some failures' associated library version constraints are solvable in some cases, but the used library installers must support backtracking (otherwise, the installation could be stuck in a local unsolvable situation). We refer to such failures as *B-failure*. Finally, the remaining failures' associated version constraints are essentially solvable in all cases, but some installers may still fail due to their specialized search mechanisms. We refer to such failures as *C-failure*.

Since DC building failures are so heavily associated with library version constraints, we naturally ask: *instead of living with very limited or even no solutions by sticking to original low-quality version constraints associated with project libraries (mainly existing work's focus, e.g., [42, 50]), can one step further by proactively refining these constraints for better quality to substantially resolve DC building failures with a better chance, and if yes, can this be done automatically?*

To answer the question, we first empirically study existing Python DC issues to investigate how DC building failures have occurred due to improper library version constraints. We observe that strict version constraints (e.g., pinned ones like `llvmlite==0.32.1`) dominate the most unsolvable cases and limited solutions, and can be refined to greatly alleviate failures by slightly loosening such constraints (e.g., loosening to one/five more version(s) already resolving 15.5%/27.5% unsolvable cases). However, loosening library version constraints can be risky, since allowing importing versions other than originally specified ones might introduce unexpected behaviors. As such, an immediate challenge is how to loosen version constraints while still assuring libraries' consistent behaviors with respect to each specific project that uses these libraries.

With this regard, we propose our behavior-consistent loosening approach, LooCo, which automatically decides maximal boundaries in loosening version constraints based on its sliced call graphs that capture truly invoked methods in these used libraries. Based on another observation from our empirical study, only very limited library methods (less than 10% on average) are actually invoked in a Python project dependency. Then our loosening approach can be very attractive by providing a great chance to effectively resolve DC building failures via simply loosening library version constraints.

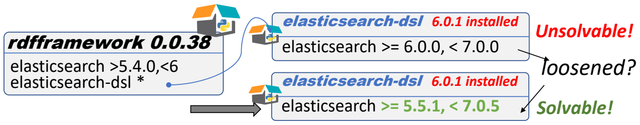

rdfframework 0.0.38
elasticsearch >5.4.0,<6
elasticsearch-dsl *
elasticsearch-dsl 6.0.1 installed
elasticsearch >= 6.0.0, < 7.0.0
Unsolvable!
loosened?
elasticsearch-dsl 6.0.1 installed
elasticsearch >= 5.5.1, < 7.0.5
Solvable!

Figure 2: Loosening illustration for DC issue #99

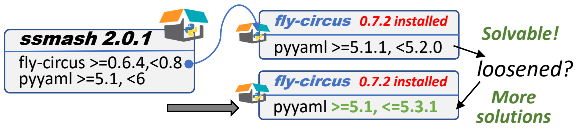


Figure 3: Loosening illustration for DC issue #131

With LooCo, one can: (1) solve originally unsolvable cases relating to A-failures and B-failures, effectively resolving DC building failures, (2) produce more solutions for originally solvable cases relating to B-failures and C-failures, potentially helping future failure resolution when projects and libraries evolve. One example from Fig. 2 illustrates DC issue #99 [8] reported by WATCHMAN [53], whose induced building can be resolved by LooCo: building `rdfframework 0.0.38` requires intalling an `elasticsearch` version satisfying "`>5.4.0, <6`", and this request conflicts with `rdfframework`'s indirect version constraint "`>=6.0.0, <7.0.0`" on `elasticsearch` (transitively introduced by `elasticsearch-dsl`); then LooCo suggests loosening the library `elasticsearch-dsl`'s version constraint on library `elasticsearch` to "`>=5.5.1,<7.0.5`" (behavior-consistency validated), and the building failure is resolved successfully. Another example [5] from Fig. 3 suggests that `ssmash 2.0.1` can be built without any problem, and LooCo can further loosen its version constraints (e.g., library `pyyaml` can expand its original constraint "`>=5.1.1,<5.2.0`" to "`>=5.1, <=5.3.1`") with more solutions to avoid potential DC building failures in future.

We experimentally evaluate LooCo's performance and usefulness with real-life Python projects and their DC issues. The evaluation results report that: (1) LooCo could effectively loosen projects' library version constraints, by 5.5 new versions expanded for each library on average, with library behaviors successfully validated by built-in test cases and developer feedbacks; (2) LooCo could solve 54.8% (46/84) originally unsolvable cases in A- and B-failures, and enhance 50.2% (101/201) from "with solutions" to "with more solutions" (21 new solutions on average) in B- and C-failures; (3) LooCo was highly efficient and its time overhead was 0.4s on average per library version loosening.

The remainder of this paper is organized as follows. Section 2 introduces the dependency conflict background. Section 3 presents our empirical study and findings for motivating a constraint loosening approach. Section 4 elaborates on our LooCo's methodology for automatically resolving DC building failures. Section 5 evaluates

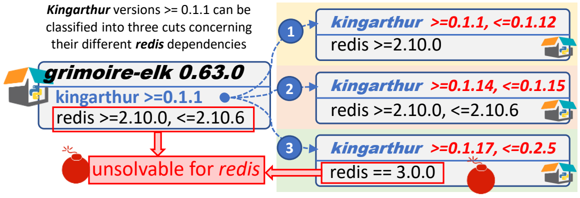

Kingarthur versions >= 0.1.1 can be classified into three cuts concerning their different *redis* dependencies
grimoire-elk 0.63.0
kingarthur >=0.1.1
redis >=2.10.0, <=2.10.6
unsolvable for *redis*
1
kingarthur >=0.1.1, <=0.1.12
redis >=2.10.0
2
kingarthur >=0.1.14, <=0.1.15
redis >=2.10.0, <=2.10.6
3
kingarthur >=0.1.17, <=0.2.5
redis == 3.0.0

Figure 4: DC issue #272 with its three dependency cuts

LooCo's performance and usefulness. Sections 6 and 7 discuss our work and related work, and finally Section 8 concludes the paper.

2 PRELIMINARIES

2.1 Dependency in Python Projects

A Python project usually depends on multiple third-party libraries via its configuration script (e.g., `setup.py` or `requirements.txt`). In the script, a list of *version constraints* would be written by developers, each of which declares expected versions for a specific library. This facilitates to better saving and restoring a correct execution environment for the project, therefore running anytime and anywhere. A version constraint usually follows PEP specifications [11] and can be divided into three classifications [47]: *pinned*, *constrained*, and *unconstrained*. A *pinned version constraint* refers to declaring a specific library version, e.g., "`request ==2.20.0`". A *constrained version constraint* refers to declaring library versions under constrained ranges (>, >=, <, <=, !=, ==1.*), e.g., "`pyyaml >=5.1,<6.0`", and an *unconstrained version constraint* refers to only declaring a library by the name only without any specific version request, e.g., `elasticsearch-dsl`.

Note that code reuse is so pervasive in the Python world, and not only Python projects depend on third-party libraries (*directly dependent library*) to reuse library code, libraries also depend on other libraries (a.k.a., *indirectly dependent library* for the project) as well via version constraint declaration. We consider those declarations in the project's configuration script for its directly dependent libraries as its *direct* version constraints, and version constraints declared for its indirectly dependent libraries transitive imported as *indirect* ones. Based on version constraints declared for the project's directly or indirectly dependent libraries, Python library installers like `Pip` [14] and `Conda` [2] can accordingly install suitable versions for a project's all concerned libraries transitively, thus saving and restoring a suitable execution environment for Python projects. Due to popularity, we consider `Pip` as the typical library installer in this work and other installers like `Poetry`, `pipenv`, `virtualenv` also share similar environment management mechanisms.

2.2 Building Failures and Dependency Conflicts

However, restoring a suitable execution environment is observed to be so difficult that developers usually spend hours or still fail to set up a correct execution environment [36]. That is because for the non-trivial NPC problem of dependency solving [23, 46], a typical installer like `Pip` does not guarantee to always achieve a solution for each library due to its specialized strategy, thus possibly leading to building failures in restoring execution environment.

Many building failures [50, 53] have been observed and affect restoring correct execution environments for Python projects due to the conflicted version constraints for installing certain libraries. This is known as *dependency conflict issues* (*DC issues*) [50, 53], which occur due to the conflict among a Python project's concerned version constraints (either direct or indirect) during installation. For example, as aforementioned earlier, Fig. 1 gives a real-world building failure [47, 54] caused by DC issues. This building failure occurred because to install library "`Clifford==1.3.1`" for this project, `llvmlite` satisfying "`<0.36, >=0.35.0`" would be transitively installed, which would result in conflicting when installing the project's another direct dependency on `llvmlite` latter, i.e., "`==0.32.1`". This exhibits a clear DC issue between the project's direct version constraint "`llvmlite==0.32.1`" and indirect one "`llvmlite<0.36, >=0.35.0`", transitively introduced by `Clifford`.

Regarding resolving DC building failures, there are two typical lines of work. One line of work [27, 56] focuses on detecting DC issues more effectively by collecting complete dependencies for projects, such as obtaining exhaustive dependency of projects besides third-party libraries, e.g., local environment, the Python interpreter and system libraries. Another line focuses on solving existing version constraints better with improved installing strategy, e.g., `Pip` backtracking [12] and SmartPip [50]. Different from existing researches generally on "sticking to projects' original version constraints to build execution environments", we focus on how to "refine version constraints automatically to resolve building failures", thus greatly complementing existing researches.

3 EMPIRICAL STUDY AND MOTIVATION

3.1 Research Questions

We raise the following research questions:

RQ1 (Study on DC issues): How did DC issues occur and lead to building failures? What version constraints are desirable in resolving such failures?

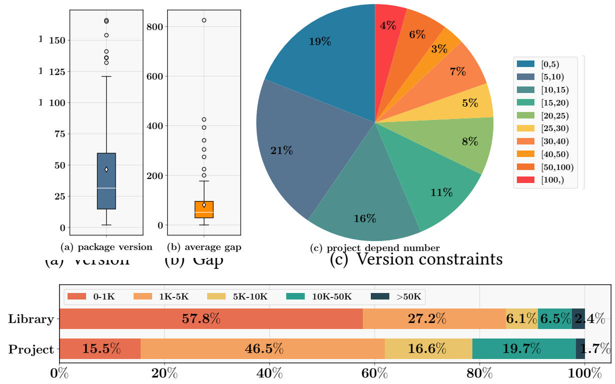
RQ2 (Study on developer practices): How did developers write library version constraints in Python projects? What common practices can be leveraged for resolving DC building failures?

3.2 Design and Setup

We collected two subjects for our empirical study on DC issues and common developer practices, respectively.

3.2.1 Subject-A on DC Issues. We tracked all DC issues reported by Watchman [53], which have also been studied in SmartPip [50]. By removing un-reproducible cases until Jan 2023 if either the code repository of the related Python project was removed or the bug report did not contain necessary reproducing information for analyses, we finally obtained 83 DC issues from 82 Python projects.

For example, issue #272 [6] is reported because when building `grimoire-elk 0.63.0` requires to install library `kingarthur` satisfying "`kingarthur>=0.1.1`", and this request might transitively introduce constraint on "`redis==3.0.0`", supposing the situation when `kingarthur  0.1.18` is truly installed. In this situation, it conflicts with `grimoire-elk`'s other direct version constraint, i.e., "`redis>=2.10.0, <=2.10.6`", as shown in Fig. 4. This makes an installer impossible to find a suitable version to install for `redis` (i.e., *root library* that fails to be installed) in this situation. The issue



(a) version (b) Gap (c) Version constraints

(d) LOC statistics

Figure 5: Description about Subject-B

describes a concrete building situation when dependency conflicts happen and can help us specify the root library that fails to be installed due to conflicting version constraints. To simulate all possible building situations related to this issue, we slice the related dependent and independent version constraints and simulate exhaustive building possibilities by different dependency cuts. Each *dependency cut* slices a conjunction situation of version constraints for the root library during building with some necessary library ranges specified. As in Fig. 4, all `kingarthur` versions in declaration `>=0.1.1` can be divided into three cuts concerning their different dependencies on the root library `redis`. When `kingarthur` is installed with a version satisfying “`>=0.1.1`, `<=0.1.12`” (cut #1), “`redis>=2.10.0`” would be introduced. When `kingarthur` is installed satisfying “`>=0.1.14`, `<=0.1.15`” (cut #2), “`redis>=2.10.0`, `<=2.10.6`” would be introduced. When `kingarthur` is installed satisfying “`>=0.1.17`, `<=0.2.25`” (cut #3), “`redis ==3.0.0`” would be introduced. Cut #3 leads to an unsolvable conjunction for installing `redis`, which is known as a dependency conflict resulting in a typical DC building failure. The three cuts compose all situations that an installer might meet for dependency solving upon the root library `redis`’s version constraints related to this DC issue. We believe such cut analyses can help us study all building situations extensively and potential threats for those DC issues.

Therefore, we analyzed and obtained a total of 285 dependency cuts for all the studied 83 DC issues. We consider these 285 cuts to represent all possible situations for the studied DC issues that can meet during building and work as our subject (a.k.a., *Subject-A*) for answering RQ1 on symptoms and causes of DC building failures.

3.2.2 Subject-B on Open Projects. We also collected open projects and libraries to investigate developers’ common practices in the real world to facilitate our resolution. First, we scanned Python repositories from GitHub with the top 500 stars with the following requirements: (1) use Python as its main language (allowing typical script language like Shell, PowerShell, Bash, etc.), and can be suitably deprecated by Python 3.8, (2) evolve with more than one version during its versioning history, and (3) be imported as a library by at least one down-stream project (a.k.a., dependent project). We thus obtained 104 libraries in total. Then, we accordingly collected dependent projects for these libraries. Considering the limitations of storage space, for libraries with over 300 dependent projects, we collected the top 300 dependent projects sorted by stars. Finally, we obtained 4,511 dependent projects (avg LOC: 12K, max LOC: 5M) for 104 libraries (avg LOC: 8K, max LOC: 104K). Regarding the 104 libraries, we tracked all available versions across its development from both GitHub and PyPI repo and obtained a total of 4,829 different library versions (avg: 46 versions, max: 166 versions). Fig. 5 illustrates the statistical information for the number of library versions, versioning gaps, the number of dependent libraries by each project, and LOC statistics in Subject-B. Then, we investigate how developers usually write and maintain version constraints for their imported third-party libraries and investigate common practices to guide our approach. More details about subjects can be found on our website [9].

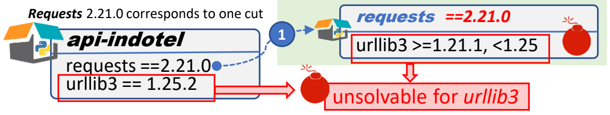


Figure 6: DC issue #56 with its only dependency cut

3.3 RQ1: Issue Symptom and Cause

Considering all 285 dependency cuts in Subject-A (introduced in Section 3.2.1), we note that each cut specifies a conjunction of several version constraints for the DC issue’s root library. If the conjunction is unsolvable, this dependency cut can never result in a solution of the root library satisfying all related version constraints, thus leading to a building failure.

For example, cut #3 in Fig. 4 composes a conjunction with two version constraints on `redis`, i.e., “`==3.0.0`” and “`>=2.10.0`, `<=2.10.6`”. They conflict with each other and make it unsolvable for a `redis` solution. In this case, some installers may directly cause a building failure (e.g., `Pip` legacy strategy before `Pip 20.3` [15]), while some would proceed with fallbacks and further search the space simulated by remaining cuts for possible solutions (e.g., `Pip` backtracking strategy after `Pip 20.3` [12]). However, if unfortunately, all dependency cuts are unsolvable, even a perfect installer cannot find a solution for installation. For example, Fig. 6 gives DC issue #56 [7] for `api-indotel` corresponding to one cut only. It is unsolvable for finding a solution of `urllib3` because in its only dependency cut, library `request`’s introduced dependency, i.e., “`urllib3>=1.21.1`, `<1.25`”, is conflicting to the project’s another direct dependency, i.e., “`urllib3==1.25.2`”. Such dependency conjunction cannot be solved by any of the existing installers because for this project, library `urllib3` is always unsolvable for all situations during building (as illustrated by the only cut). We refer to such building failures separately and obtain the following symptoms.

> *Symptom 1: 9.6% (8/83) DC issues are unsolvable unless their associated version constraints are rewritten, since all their corresponding dependency cuts (8/285) are unsolvable (a.k.a., A-failure).*

> *Symptom 2: 31.3% (26/83) DC issues contain at least one unsolvable cut (76/285), and resolving their building failures has to install fallback or backtracking (a.k.a., B-failure).*

We observe that 9.6% DC issues are unsolvable because all their dependency cuts are unsolvable, and thus even a perfect installer cannot resolve such unavoidable building failures (A-failure). Besides, 31.3% DC issues contain at least one unsolvable cut, and thus some existing installers without backtracking may result in building failures locally (B-failure). We chose three representative installers (`Pip` legacy [15], `Pip` backtracking [12], and the state-of-the-art smartPip [50]). We experimentally confirmed that none of them can build smoothly for the eight DC issues causing A-failures, because resolving such unsolvable DC issues requires rewriting declarations of version constraints among the Python project and related libraries, as echoed in existing work [50]. Meanwhile, for DC issues causing B-failures, `Pip` legacy suffers from 88.5% (23/26) building failures while `Pip` backtracking and the state-of-the-art smartPip can theoretically solve them all by allowing backtracking strategies or spending more search resources to maintain an up-to-date dependency database. However, in practice, backtracking costs and database maintainence are non-negligible. For example, `Pip` backtracking usually needs to retry 20 times on average to obtain its eventual solution and may still result in building failure for complex scenarios when resources are limited [50].

Finding 1: None of existing efforts on improving installing strategies could resolve A-failures, while smart strategies with backtracking could resolve B-failures with extra resources.

Considering all 84 unsolvable dependency cuts that cause either A-failures or B-failures, we measure the number of library versions to loosen at least in order to connect associated version constraints of the concerned unsolvable cuts. For example, for the cut #3 shown in Fig. 4, there is only version `redis 3.0.0` to directly connect the two constraints, i.e.,"`>=2.10.0, <=2.10.6`" and "`==3.0.0`". We call the *distance* (i.e., how far are the concerned version constraints connect to each other with the least expansions) for this unsolvable cut being one. We observe that among the 84 unsolvable cuts, 13 cuts are with distance being one and can become solvable if one more version is allowed for version constraints, e.g., `redis 3.0.0` can be allowed if loosening "`>=2.10.0, <=2.10.6`" to "`>=2.10.0, <=3.0.0`" for cut #3 in Fig. 4. In our study, 23/38 cuts can become solvable with no more than five/ten neighboured versions being loosened. For example, cut #1 in Fig. 6 can become solvable if two neighboured versions are allowed for version constraints, e.g., constraint "`urllib3<1.25`" being loosened to "`urllib3<=1.25.2`".

Finding 2: Loosening version constraints can be useful. For unsolvable dependency cuts, 15.5% (13/84) can become solvable by loosening one more adjacent version, and 27.4%/45.2% can become solvable by loosening no more than five/ten versions.

For the remaining DC issues that contain no unsolvable cuts, we also observe that not all installers can smoothly install the root library due to their specialized installing strategies. For example, `Pip` legacy strategy [15] (before 20.3) would typically choose the latest version for the library that satisfies the concerned version constraints, and conduct its installation following in a BFS-alike order with guarantee upon a topological order only. Due to its specialized installing order, we observe that it met 33 building failures to those 49 DC issues. Fortunately, with extra resources allowed for backtracking and knowledge base analyses, `Pip` backtracking (after 20.3) and smartPip can build all these cases smoothly.

Symptom 3: 59.0% (49/83) DC issues contain no unsolvable cut. Still, some installers may incur building failures due to their specialized installing strategies (a.k.a., C-failure). Fortunately, SOTA installers can resolve them smoothly.

By digging into the total of 201 solvable dependency cuts associated with both B-failures and C-failures, we observe that although solvable, over 50% dependency cuts (111/201) are eventually with limited solutions (no more than five acceptable versions), which can leave threats for evolution. For example, as in Fig. 3, there are only two acceptable solutions (`pyyaml 5.1.1` and `pyyaml 5.1.2`) for this solvable case. More severely, 35.8% (72/201) dependency cuts result in only one unique solution when solving the conjunction of the concerned version constraints. This is mainly due to the involvement of a pinned version constraint (e.g., "`urllib3==1.25.2`") which specifies one unique library version.

Finding 3: Over 50% solvable cuts can result in limited solutions in dependency solving, leaving potential threats for evolution.

Although limited solutions do not result in building failures at the moment, it leaves potential threats for future evolution since any additional version constraint that may be introduced in the future can easily cause the unsolvable problem. Moreover, among all unsolvable cuts, there are over 40% cuts involving at least one pinned version constraint and can result in extremely limited solutions (only one acceptable library version). Therefore, we can see that strict version constraints like the pinned ones can not only lead to unsolvable cuts but also limit solutions to solvable ones.

Finding 4: Strict version constraints like pinned ones can lead to unsolvable or limited solutions in dependency solving, and thus version constraint loosening can be desirable.

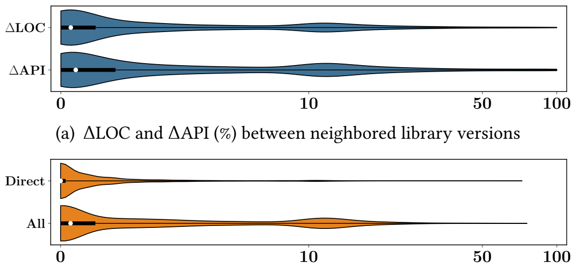
Answering RQ1: Loosening version constraints (especially pinned ones) can be useful for both resolving DC building failures caused by unsolvable cuts and allowing more solutions for solvable cuts in dependency solving.

3.4 RQ2: Developing Status and Practice

From Fig. 5, we can see that: (1) the number of dependent libraries by a project ranges from several to hundreds, and vary across projects, (2) over 50% projects specify more than 10 libraries in their dependency, and around 10% projects import more than 50 libraries, which may incur dependency conflict issues with high possibilities, and (3) libraries evolve quickly with versioning gaps of only several months for new versions, with 48 total versions on average. These together put extreme pressures on developers to maintain suitable version constraints for importing third-party libraries.

Observation 1: Developers tend to import a large amount of third-party libraries and these libraries usually evolve quickly, resulting in non-trivial efforts on dependency maintenance.

From Subject-B, we collect all written version constraints for those 104 libraries from projects' configuration scripts (i.e., a total of 109,129 version constraints). Among all, we observe that 93,345 (85.5%) version constraints are pinned (using "==" to declare one



(a) ΔLOC and ΔAPI (%) between neighbored library versions

(b) Proportion (%) for directed and all traced callings

Figure 7: Statistical data about library versioning or usage

specific library version only). Such rates also hold consistently over 80% concerning different scales of projects, e.g., 1k–5k: 84.6%, >50k: 81.9%. We conjecture that this may because in practice developers usually directly freeze and export their default execution environment during development into a default configuration script by "*pip freeze*". In this way, default pinned version constraints [47] can be generated [13], and one can later restore this environment easily by "*pip install*". In this case, as long as any upstream or downstream projects depend on the same library but require any other version, "unsolvable" disasters (leading to A-failures and B-failures as studied in RQ1) would happen.

Observation 2: Developers prefer to write strict version constraints, and this can easily cause unsolvable cuts leading to both A- and B-failures.

As we studied in RQ1, loosening might be optional for relaxing such strict version constraints as pinned ones to resolve DC building failures. However, loosening any version constraint safely needs to protect the project's behavior, and it requires quite an effort for analyzing realizations for the concerned project and the library's all optional versions. It can be tedious and somehow infeasible in practice to conduct a full analysis due to code scales. We then dig into how libraries evolve and how developers call library APIs in practice to facilitate a feasible loosening mechanism.

When investigating how each library evolves between its two neighbored versions, we emphasize on those functions and lines of codes have been either modified or newly added (a.k.a., ΔAPI and ΔLOC) in the latter version, are shown in Fig. 7. We observe that after discarding the first version for each library, 34% versions evolve less than 1% functions only, and 61% library versions evolve less than 5%. Only 10% versions evolve more than 20% functions. This denotes that during evolution, libraries tend to evolve limited functions only. We also investigate how projects usually invoke functions through APIs from third-party libraries. For each project-library pair in Subject-B (a total of 4,511 subjects, concerning 4,511 projects for 104 third-party libraries) as aforementioned, we dig into how the project invokes APIs from its associated library in the collection. Typically, one would use "`import`" or "`from import`' to import module(s) from libraries, and then invoke APIs whenever needed by developers in developing the project. For each subject, we recursively track all modules from its associated project that import the concerned library and then analyze how many APIs/functions from the library are truly invoked in the project (clearly specified in the project code, and definitions successfully located in the library code). For example, project "`likyoo/change_detection.pytorch`" requires library "`albumentations`" with constraint "`==1.0.3`". We track 8 APIs directly invoked in the project, and by manually inspecting their definitions in the library, we can together track another 11 inner functions indirectly called in this library, which only occupies a minuscule portion of the library's functionalities (with over 700 functions supported in total). Similar observations hold in most of our subjects as shown in Fig. 7. Directly invoked APIs usually take less than 1% among all library functions and indirectly invoked ones may together increase the ratio to still no more than 5% (can be also analyzed by our LooCo later).

Practice 1&2: Libraries typically evolve quickly but with very limited modifications, and projects usually invoke only limited functionalities from libraries during development.

Therefore, we can see that upon Practice 1 and 2, a feasible loosening mechanism to maintain the project's consistent behavior does not require a full analysis of the whole realization of both the project and library with candidate versions. We only need to protect the library contexts that are truly used in the project, thus motivating our loosening mechanism upon on-demand call graphs starting from entrance APIs.

Answering RQ2: Only limited library functionalities participate in project development, and this inspires one to potentially resolve DC building failures by loosening library versions with behavior-consistency validation.

4 APPROACH

4.1 Insight and Overview

To resolve DC building failures by loosening version constraints, the kernel question is "*how to examine whether a project can behave consistently upon a new version v_2 of its dependent library l?*" Generally, suppose a project depends on a library l with an acceptable version v_1 (base version) to reuse library functionalities by a few l's APIs. We consider that if all called APIs and any parts transitively called in the library have not been modified between v_1 and v_2, the project should behave consistently in between. Thus, LooCo can loosen the project's version constraint for l to v_2. LooCo's workflow is shown in Fig. 8. It would first scan the whole project to obtain all entrance APIs for how project p uses library l based on a base version v_1 (Step 1), and then, tracking such entrance APIs, LooCo would trace the remaining library parts that are transitively called via constructing an on-demand call graph (Step 2). Then, LooCo would generate version diffs between v_1 and v_2 (Step 3). Combining both version diffs and the constructed call graph, LooCo can suggest to loosen version constraints to v_2 as long as they do not have any shared content, suggesting that any library part in l that are directly and transitively used by the project have not been evolved between v_1 and v_2. Otherwise, loosening is risky (Step 4).

4.2 Step 1: Entrance API Extraction

This step aims to extract all project p's entrance APIs for using library l. Typically, to reuse library functionalities, Python projects can invoke the library APIs by both long qualified names like `img.img_to_graph`, `T.img_to_graph`, `np.reshape` and shortened

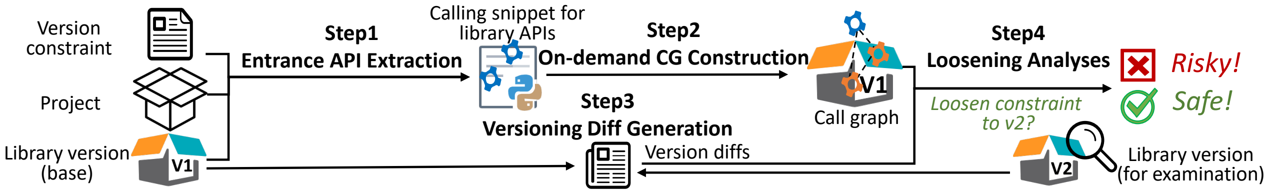


Figure 8: LooCo workflow

ones like `API_Name` (`img_to_graph`). We model the entrance APIs for the library no matter whether they are called by long or shortened API names to a consistent style of a fully qualified name `A.B.C.API_Name`, which specifies the path, module, class, and API name for any concerned library API. Therefore, given project p and library l with base version v_1, this step tries to exact entrance APIs with fully qualified names to exhibit l's functionality used by p.

Considering that developers may follow diverse ways to import library, LooCo adopts a two-phase exaction mechanism ("extract-first-localize-next") for obtaining such entrance APIs. First, LooCo adopted and refined an existing tool DLocator [51] to extract all l's APIs that have been called in the project code. Note that, we addressed DLocator's limitations which can lead to unexpected entrance APIs being missed during its extraction, e.g., failing in processing the "`import *`" cases, and ignoring APIs called by non-call AST nodes. Note that, solely scanning the project code may not precisely exact library APIs and obtain the corresponding fully qualified names since the library codes are invisible now. Therefore, in this phase, LooCo aims to exact all possible entrance APIs for l that may be called by the project and aim to exact entrance APIs conservatively first. Second, to avoid possible mistakes in API extraction, LooCo then conducts a localization phase to localize the realization body of any exacted library API in the last phase by scanning library l's code of base version v_1. This manages to filter out any mistakenly extracted calls for library APIs, and assign the expected fully qualified API names for all successfully filtered ones.

As such, LooCo now exacts a set of entrance APIs for library l concerning its API usage in project p, i.e., $S_A = \{api_0, api_1, \ldots, api_n\}$, each of which specifies a specific API api_i in l (with its fully qualified name) being called by p.

4.3 Step 2: On-demand CG Construction

This step aims to construct a call graph slicing from the entrance APIs obtained in the last step. Considering the set of entrance APIs being extracted to be $S_A = \{api_0, api_1, \ldots, api_n\}$, LooCo would track each entrance API in l's v_1 to slice all internal parts (e.g., function, method, class, etc.) transitively called by project p.

To obtain such, LooCo would not need to construct a full call graph for library l, but only a partial one that called from the obtained entrance APIs in S_A. Considering our observations in empirical study, i.e., limited l's APIs are normally called by projects, we can expect that such an on-demand call graph would be much easier and feasible in practice. Therefore, to slice a call trace including possible concerned library functions, class methods, and classes themselves (indicating constructors called), LooCo analyzes each obtained entrance APIs transitively, and thus composes an on-demand call graph by merging all call traces for these obtained entrance APIs. Since LooCo aims to loosen p's version constraints for l, we consider capturing the possible calls in the sliced call trace for l's directly dependent library as well. Therefore, for each sliced node in the call trace, we logged a tuple of its necessary information including node name, caller, callee, and dependency location. For example, for a inner-called node (function f, method m, class c) that is called and realized by l itself, LooCo would collect $< f/m/c, f_2, A.B.C.f/m/c, l >$, representing $f/m/c$ is called by f_2 and realized by library l in $A.B.C.f/m/c$. For a outer-called node $< f/m/c, f_2, -, l_2 >$ represents that $f/m/c$ is called by f_2 and realized by l's dependent library l_2, with realization unknown.

To do so, LooCo refines code2flow [1] to analyze possible calls traced from the obtained entrance APIs in Step 1 by tracking all possible calls in the AST structure with an additional definition-mapped domain supported. As a result, LooCo's call graph analyses can exhibit comparable performance to the state-of-the-art Python analyzer PyCG [49], which would be further discussed in Section 6.

4.4 Step 3: Versioning Diff Generation

This step is to construct the differences for library l's realization between its base version (v_1) and the version to loosen (v_2). We consider the contents of both l's code and configuration scripts. Concerning the difference of l's code, LooCo compares l's v_1 realization to v_2 by constructing code diffs with the aid of filecmp [4] and difflib [3], and records the full names of functions/methods/classes that are defined in v_1, but modified (except modifying comments only) or deleted in v_2. This would produce a list of all modified components (function, method, class) during l's evolving from v_1 to v_2, namely, $Diff_{code} = \{f_1, m_2, c_3\}$. Concerning the difference of l's configuration, LooCo compares l's configuration scripts between v_1 to v_2 and obtained a list of l's dependent libraries whose dependencies have been modified during l's evolution from v_1 to v_2, either deleted or modified with a new version constraint, $Diff_{lib}$.

4.5 Step 4: Loosening Analyses

This step would analyze the on-demand call graph obtained in Step 2 and versioning diffs ($Diff_{code}$ and $Diff_{lib}$) obtained in Step 3 to examine whether p's dependency on l can be further loosened to v_2. LooCo would suggest to loosen as long as all nodes in constructed call graph satisfy conditions: (1) if the node associates to an outer call f from an indirectly dependent library l_2, l_2 must not be contained in $Diff_{lib}$; (2) if the node associates to an inner call f from the library l under examination, f must not be contained in $Diff_{code}$. Otherwise, LooCo would not suggest loosening p's dependency on l to v_2. Note that, LooCo works with its behind insight of only loosening p's dependency on l to v_2 as long as all p's called parts in l have not been modified. However, we do admit that even some modifications (i.e., equivalent code changing) for these parts may not affect p's behaviors as well when using l's new version. For safety, we leave LooCo as such in a conservative way to facilitate its feasibility and automation at the same time.

4.6 LooCo Realization and Application

LooCo is proposed to solve the kernel question for examining whether the analyzed Python project can behave consistently upon its dependent library l with a new version v_2, a.k.a., safe for loosening v_2. In practice, suppose the project p and any of its imported library l declared in the configuration script. Considering different types of version constraints originally written by developers, LooCo can be applied as follows. (1) For the pinned constraint which specifies one concrete version only, e.g., $l == v_0$, LooCo can treat v_0 as the base version and examine any l's accessible versions (recommend to start forward and backward from v_0) in the market for possible loosening. (2) For the constrained one which may give a range of acceptable versions, e.g., $v_1 <= l <= v_2$, LooCo can choose any acceptable version satisfying the original constraint (e.g., $v3$ when $v_1 <= v_3 <= v_2$) as the base version, and any version can be loosened as long as LooCo believes safe for loosening, suggesting the project's consistent behavior between v_1 and v_2.

By doing so, upon the project's existing configuration script, LooCo can examine all accessible library versions that are not included originally, and loosen version constraints automatically as relaxed as possible. After that, LooCo would suggest revising the configuration script with concrete suggestions with loosened versions for any specific library. Moreover, due to LooCo's superiority on tracking the project's called APIs for each library (e.g., l), LooCo additionally suggests removing some original version constraints if none of l's APIs is called in p, i.e., $S_A = \emptyset$.

5 EVALUATION

5.1 Experimental Preparation

5.1.1 Research Questions. We raised the following two research questions to evaluate LooCo's performance on loosening version constraints and resolving DC building failures.

RQ3 (Loosening Performance): Can LooCo effectively loosen version constraints for Python projects' imported libraries? How efficiently can this be done?

RQ4 (Resolving Usefulness): How useful is LooCo for resolving DC building failures by its automatic constraint loosening?

5.1.2 Design and Setup. **For RQ3**, to evaluate LooCo's performance on loosening version constraints for projects when importing third-party libraries, we use Subject-B in Section 3, which contains 4,511 open projects for the collected 104 third-party libraries, thus 4,511 project-library pairs for experiments with a version constraint written by developers originally. For each pair, we use LooCo to loosen the project's concerned version constraint upon the associate library as possible. We evaluate LooCo's effectiveness by measuring the number of new library versions LooCo can loosen and use the consistency of behavior between the loosened version and the original version on test cases to validate the correctness of LooCo's loosening. And we evaluate LooCo's efficiency by the time cost when applying LooCo to examine whether a project's version constraint can be loosened to a certain library version. **For RQ4**, to evaluate LooCo's loosening usefulness on resolving DC building failures, we use Subject-A in Section 3, containing real-world DC issues with 285 dependency cuts (three removed due to only one version constraint for the root library) representing dependency among all practical building possibilities. By similarly applying LooCo to loosen each issue's related version constraints, we examine whether and how the concerned DC building failures are resolved and thus evaluate LooCo's usefulness.

Table 1: Loosen levels of LooCo's results (#)

Level	L0	L1	L2	L3
Pinned (3,754)	2,513 (66.9%)	919 (24.5%)	165 (4.4%)	157 (4.2%)
Constrained (757)	493 (65.1%)	206 (27.2%)	32 (4.2%)	26 (3.4%)
All (4,511)	3,006 (66.6%)	1,125 (24.9%)	197 (4.4%)	183 (4.1%)

5.1.3 Configuration. All experiments were conducted on a server with two 12-core Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz and 503GB RAM, installed with Ubuntu 20.04.1 LTS and Python 3.8.

5.2 RQ3: Loosening Performance

We evaluate LooCo's loosening effectiveness by measuring how many versions can be loosened for each project concerning the associated library in the 4,511 pairs. Then, we validate LooCo's loosening results by combining both test suite validation and developers' feedbacks. Finally, we measure LooCo's time overhead.

5.2.1 Loosening Results. For the studied 4,511 project-library pairs, we first measured how many new versions can be further loosened when applying LooCo, to investigate its loosening effectiveness. By classifying LooCo's loosen effectiveness by the number of loosened versions, we design four levels with increasing numbers of loosened versions, i.e., L0 (no new version loosened), L1 (1–5 versions loosened), L2 (6–10 versions loosened), and L3 (over 10 versions loosened). As shown in Table 1, LooCo successfully loosened version constraints for 1,505 (33.4%) projects' imported libraries, with 1–5 versions loosened (L1) for 1,125 (24.9%) projects, 6–10 versions loosened (L2) for 197 (4.4%) projects, and over 10 versions loosened (L3) for 183 (4.1%) projects, with 5.5 loosened versions on average.

5.2.2 Loosening Validation. To validate LooCo's loosening results, we filtered out 47 pairs among all collected project-library pairs with the following requirements: (1) the associated project is provided with built-in test suites (e.g., Pytest); (2) built-in test suites can work smoothly under the original dependency configuration, and work inconsistently when uninstalling the associated library, thus covering necessary functionality for testing this library dependency; (3) LooCo suggests loosening new library versions. For 47 collected pairs, the average test code coverage of those obtained projects' test suites' is 69.5%(min: 28%, max: 100%). Then, for each filtered project-library pair, we ran test suites under both library versions satisfying its original configuration and installing LooCo's suggested versions. We examined whether test suites produce consistent behaviors between such two executions. Among all, LooCo in total suggests loosening existing version constraints for 254 new library versions, and test suites behave completely consistently for 248 (97.6%) library versions. The only exceptions are pair #598 (i.e., project `ServerlessChalicePlatform` upon library `chalice`) and #325 (i.e., project `mlcomp` upon library `albumentations`). We dig a little deeper and find out that both their inconsistency behaviors under test are due to the conflict between LooCo's suggested versions and our local experimental environment (i.e., PyTest and Python 3.8). After neglecting such conflicts and installing LooCo's suggested versions forcibly, consistency behaviors remain hold.

5.2.3 Feedback on LooCo. We sent LooCo's suggestions to developers with loosened versions. Until Aug 2023, we randomly sent over 600 reports and received 66 responses from developers (covering 60 project-library pairs). Among these 66 responses, we achieved encouraging feedback, with details on our website [9]. 49 developers (74.2%) have confirmed LooCo's loosening plans, and most developers (38/49) have already refined their dependency following LooCo's suggestions, either asking us for pull requests (27/38, with 21 merged already) or resolving themselves (11/38) exactly following LooCo's suggestions.

We also received valuable responses from developers. Developers do admit that they typically use "*pip freeze*" for convenience and may lack sufficient dependency analyses, e.g., "*Using strict version dependency does seem like a mistake or rather something I overlooked while using* pip freeze. *Loosening is a good idea.*" in report #2688 [19]. They indeed appreciate LooCo's loosening suggestions, e.g., "*Thanks for the suggestion about the automatic tool for dependency analysis. I appreciate the thought, . . . we don't have anything like that in place.*" in report #2135 [18], "*PR always welcome, and really sorry that I just notice this issue.*" in report #700 [21], and "*You're right. I tested* `Slips` *with* `redis 3.5.2` *and it's working fine.*" in report #282 [20].

5.2.4 Overhead. We also measured the time cost when applying LooCo to loosen a project's version constraint for a certain library. On average, LooCo spent 0.4s (min: 0.04ms, max: 54s) to make its examination per candidate version to loosen. Despite of LooCo's analyses, LOCs of each project-library pair also affect the final time cost, e.g., the LOC of the project in pair #4470 is about 3.7M, somehow explaining its relatively large time overhead (54s). Considering the varying scales, i.e., project (avg LOC: 12K, max LOC: 5M), library (avg LOC: 8K, max LOC: 104K), we believe LooCo is efficient with nice scalability, owing to its on-demand CG analyses.

Answering RQ3: LooCo can loosen version constraints for Python projects' imported libraries effectively (avg: 5.5 versions expanded) and efficiently (0.4s overhead per loosening). We received encouraging feedback (74.2% confirmed in responses) from developers.

5.3 RQ4: Resolving Usefulness

5.3.1 Loosening Constraints upon Dependency Cuts. To investigate how LooCo's constraint loosening can help resolve DC building failures in practice. We use Subject-A introduced in Section 3, containing 83 real-world DC issues with 285 dependency cuts for building. For each dependency cut, LooCo is conducted to loosen version constraints accordingly. Table 2 illustrates LooCo's results.

For the 84 dependency cuts that are originally unsolvable, LooCo can effectively loosen version constraints for 68 cuts (81.0%, besides those gray ones). Among all unsolvable cuts, 46 (54.8%) cuts can become solvable (R1), and 11 (13.1%) cuts, although still unsolvable, can bring closer distances for solving the conjunctions in dependency (R2). For example, concerning the unsolvable cut #6 for issue #99 as shown in Fig. 2, LooCo would perform its analyses to both loosen project `rdfframework`'s dependency on `elasticsearch` and also `elasticsearch-dsl`'s dependency on library `elasticsearch`. As a result, the former one would be loosened from `>5.4.0`, `<6`" to "`>5.4.0`, `<=6.1.1`" and the latter one would be loosened from "`>=6.0.0`, `<7.0.0`" to "`>=5.5.1`, `<7.0.5`", thus solvable. This is tagged as #99, C6, rdfframework in Table 2.

For the remaining 201 dependency cuts that are solvable originally, LooCo effectively loosens version constraints for 125 cuts (62.2%). Among all solvable cuts, LooCo's loosening brings new solutions for 101 cuts (50.2%) with 21 more solutions on average (R4). For example, concerning the solvable cut #2 for issue #131 as shown in Fig. 3, as a result, `fly-circus`'s dependency on `pyyaml` would be loosened from "`>=5.1.1`, `<5.2.0`" to "`>=5.1`, `<=5.3.1`'. Therefore, four more solutions can be obtained for dependency solving, i.e., from `pyyaml 5.1` to `5.3.1`. This case is also tagged as #131, C2, ssmash in Table 2. Therefore, we can observe that LooCo's loosening can effectively resolve unsolvable cuts to become solvable and bring more solutions to the solvable ones.

5.3.2 Resolving DC Building Failures. Note that, when all associated dependency cuts are unsolvable, none of the existing optimizations on installing strategies can help, and they would all suffer from severe building failures as studied in Section 3.

This relates to 8 DC issues with none solvable cut (a.k.a., causing A-failures). LooCo can successfully resolve 6 (75%) of them by achieving at least one solvable cut now. In this way, state-of-the-art installers like smartPip can now build them smoothly. For the 26 DC issues that contain at least one unsolvable cut (a.k.a., causing B-failures), LooCo can resolve 13 (50%) of them from being unsolvable anymore, and alleviate the remaining (50%) by either letting any of its "unsolvable" cut with a closer distance or "solvable" cuts with more solutions (avg: 12 more solutions).

For the remaining 49 DC issues (although with all solvable ones initially, but may still fail under some specific installing strategies like `Pip` legacy, a.k.a., C-failures), LooCo can alleviate 31 (63.3%) of them by making their associated cuts with more solutions (avg: 10 more solutions) and thus letting any accompanied installing strategy to find a possible solution much easier. By comparing LooCo's suggestions with reports sent by Watchman, we observe that LooCo suggests similarly as Watchman with concrete suggestions of loosened version constraints for 20 cases (42.5%) among 46 cases that have been confirmed and resolved by developers. There are 10 cases when LooCo's suggestions are precisely the same as the developers' true resolutions, supporting LooCo's usefulness on resolving DC building failures.

Answering RQ4: LooCo can effectively resolve 54.8% unsolvable cases to be solvable and allow 21 more solutions on average for solvable cases, thus successfully resolving DC building failures (unsolvable cases resolved for 75% A-failures and 50% B-failures, and solutions expanded for 50% B-failures and 63.3% C-failures).

Table 2: Loosening effectiveness upon dependency cuts for LooCo with comparisons

	Loosening results for all 285 dependency cuts (with the name shortened to save space)
Unsolvable Cuts (84)	#23,C2,Runcible #35,C1,cert-issu #40,C1,whats-bot #52,C1,CoinMarke #55,C1,COCO-Styl #56,C1,api-indot #58,C3,jawfish #58,C4,jawfish #58,C5,jawfish #58,C6,jawfish #58,C7,jawfish
	#58,C8,jawfish #66,C1,django-cu #66,C7,django-cu #66,C8,django-cu #99,C6,rdfframew #116,C2,django-te #116,C3,django-te #116,C4,django-te #116,C5,django-te #116,C6,django-te
	#116,C7,django-te #116,C8,django-te #116,C9,django-te #116,C10,django-te #116,C11,django-te #128,C1,reana-clu #260,C1,Osmedeus #261,C3,dork #261,C4,dork #261,C5,dork
	#261,C6,dork #270,C2,Scriptax- #272,C3,grimoirel #278,C13,transifex #278,C14,transifex #278,C15,transifex #339,C3,docker-cr #339,C4,docker-cr #347,C2,django-we #349,C3,cert-mail
	#349,C4,cert-mail #354,C6,agora-wot #364,C3,trio #364,C4,trio #364,C5,trio #50,C1,discord-w #77,C4,imgsync #77,C5,imgsync #99,C3,rdfframew #99,C7,rdfframew #117,C2,textX-lan
	#117,C3,textX-lan #121,C1,django-gl #121,C4,django-gl #122,C1,django-gl #122,C4,django-gl #37,C2,crema #37,C3,crema #41,C1,zarp #59,C2,ltiauthen #99,C4,rdfframew
	#124,C7,mockerena #124,C8,mockerena #124,C9,mockerena #124,C10,mockerena #337,C3,superdesk #366,C2,zelt #77,C1,imgsync #77,C6,imgsync #120,C3,django-gl #121,C2,django-gl
	#121,C3,django-gl #121,C5,django-gl #121,C6,django-gl #122,C2,django-gl #122,C3,django-gl #122,C5,django-gl #122,C6,django-gl #354,C3,agora-wot #354,C5,agora-wot
	#362,C2,piicatche #362,C3,piicatche #362,C4,piicatche
Solvable Cuts (201)	#3,C1,aucome #3,C2,aucome #5,C1,kindred #7,C1,crypto-wh #8,C1,OrcaSong #9,C2,pypmml-sp #9,C3,pypmml-sp #9,C4,pypmml-sp #9,C5,pypmml-sp #12,C1,twitterbo
	#15,C1,AWSBucket #19,C1,unblock_y #20,C1,auto_craw #23,C1,Runcible #37,C1,crema #46,C1,Hidden-Fr #58,C1,jawfish #58,C2,jawfish #65,C1,dedis-clu #66,C6,django-cu
	#73,C1,flask-mon #73,C2,flask-mon #86,C1,program-y #96,C1,python-bi #96,C2,python-bi #96,C3,python-bi #96,C4,python-bi #96,C5,python-bi #96,C6,python-bi #98,C1,py-redis-
	#99,C1,rdfframew #99,C2,rdfframew #99,C5,rdfframew #108,C1,gmssl #114,C1,target-da #114,C2,target-da #116,C1,django-te #117,C1,textX-lan #118,C1,twitter_m #124,C1,mockerena
	#124,C6,mockerena #125,C1,nornir #128,C2,reana-clu #130,C1,spartacus #130,C2,spartacus #131,C1,ssmash #131,C2,ssmash #133,C1,TMO4CT #133,C2,TMO4CT #257,C1,bakerydem
	#257,C2,bakerydem #257,C3,bakerydem #257,C4,bakerydem #257,C5,bakerydem #257,C6,bakerydem #261,C1,dork #261,C2,dork #266,C1,pymacaron #266,C2,pymacaron
	#270,C1,Scriptax- #271,C1,1a23-tele #271,C2,1a23-tele #271,C3,1a23-tele #271,C4,1a23-tele #271,C5,1a23-tele #271,C6,1a23-tele #278,C1,transifex #278,C2,transifex #278,C3,transifex
	#278,C4,transifex #278,C5,transifex #278,C6,transifex #278,C7,transifex #278,C8,transifex #278,C9,transifex #278,C10,transifex #278,C11,transifex #278,C12,transifex #278,C16,transifex
	#278,C17,transifex #278,C18,transifex #329,C1,fossor #331,C1,video-fun #331,C2,video-fun #331,C3,video-fun #331,C4,video-fun #337,C1,superdesk #339,C1,docker-cr
	#339,C2,docker-cr #346,C1,djangoplu #347,C1,django-we #349,C1,cert-mail #349,C2,cert-mail #351,C1,antinex-u #363,C1,superdesk #363,C2,superdesk #364,C1,trio #364,C2,trio
	#365,C1,incubator #22,C1,Indy-node #59,C1,ltiauthen #65,C2,dedis-clu #65,C3,dedis-clu #65,C4,dedis-clu #65,C5,dedis-clu #66,C2,django-cu #66,C3,django-cu #66,C4,django-cu
	#66,C5,django-cu #124,C2,mockerena #124,C3,mockerena #124,C4,mockerena #124,C5,mockerena #125,C2,nornir #125,C3,nornir #272,C1,grimoirel #272,C2,grimoirel
	#335,C1,tensor2te #337,C2,superdesk #341,C1,molo.surv #365,C2,incubator #366,C1,zelt #367,C1,zvt #367,C2,zvt #367,C3,zvt #2,C1,django-el #2,C2,django-el #2,C3,django-el
	#2,C4,django-el #2,C5,django-el #2,C6,django-el #3,C3,aucome #8,C2,OrcaSong #8,C3,OrcaSong #8,C4,OrcaSong #8,C5,OrcaSong #8,C6,OrcaSong #9,C1,pypmml-sp #10,C1,toolium
	#10,C2,toolium #10,C3,toolium #10,C4,toolium #10,C5,toolium #10,C6,toolium #10,C7,toolium #10,C8,toolium #10,C9,toolium #10,C10,toolium #10,C11,toolium #10,C12,toolium
	#11,C1,WavesGate #11,C2,WavesGate #18,C1,ScrapyRed #18,C2,ScrapyRed #18,C3,ScrapyRed #18,C4,ScrapyRed #18,C5,ScrapyRed #77,C2,imgsync #77,C3,imgsync #78,C1,iprange-p
	#78,C2,iprange-p #78,C3,iprange-p #78,C4,iprange-p #78,C5,iprange-p #82,C1,musco-tf #82,C2,musco-tf #83,C1,musco-pyt #83,C2,musco-pyt #92,C1,pyclics-c #92,C2,pyclics-c
	#92,C3,pyclics-c #96,C7,python-bi #98,C2,py-redis- #98,C3,py-redis- #98,C4,py-redis- #98,C5,py-redis- #98,C6,py-redis- #107,C1,scvelo #107,C2,scvelo #107,C3,scvelo #107,C4,scvelo
	#107,C5,scvelo #111,C1,sockeye #120,C1,django-gl #120,C2,django-gl #130,C3,spartacus #130,C4,spartacus #130,C5,spartacus #130,C6,spartacus #130,C7,spartacus #330,C1,xontrib-r
	#331,C5,video-fun #353,C1,agora-py #354,C1,agora-wot #354,C2,agora-wot #354,C4,agora-wot #357,C1,aiocontex #359,C1,django_cl #359,C2,django_cl #359,C3,django_cl
	#362,C1,piicatche

R1: loosening with new solutions for the unsolvable cuts; R2: loosening but still with no solution for the unsolvable cuts (closer distance); R3: loosening but still with no solution for the unsolvable cuts (same distance); R4: loosening with new solutions for the solvable cuts; R5: loosening but without new solution for the solvable cuts; R6: no loosening by LooCo.

6 THREAT ANALYSES AND DISCUSSION

Threat Analyses. The subject selection for DC issues and open projects may threaten the validity of our empirical study. To alleviate this threat, for DC issues, we leveraged DC issues collected and studied in the prior studies [50, 53] and manually ensured their reproducibility for nice data quality. For open projects, we select libraries with high popularities (top 500 stars on GitHub), multiple dependent projects (avg: 43 dependent projects), and varying program scales for both projects and libraries to be representative.

Limitations of Static Analyses. Impreciseness in static analyses may also threaten the validity of LooCo, e.g., unsupported language features and complex Python semantics. Therefore, some analyzed calls for third-party libraries may be missed during API entrances and CG tracking in LooCo. This is a widely-admit challenge for static analysis research (not our focus in this paper). We try to alleviate this by optimizing our static analyzer in LooCo for comparable performance to the state-of-the-art tool PyCG [49] (we do not directly use it since it does not support on-demand CG construction and occasionally run into exceptions when analyzing large programs [16]). We compare LooCo’s constructed call graphs with those generated by PyCG upon its released benchmark, where we achieve a 99.6% precision and 70.1% recall, comparable to PyCG (precision: 98.9%, recall: 68.2%). Still, we admit that such limitations of static analyses may bring both false positives/negatives to our analyses, and this should be further investigated in the future to better show and improve the safety of LooCo’s loosening.

LooCo’s Application. In this work, we use LooCo to analyze a project’s directly dependent library for possible version loosening. Therefore, it is possible that some outer calls are basically equivalent even version constraints for some indirectly dependent library change. However, since this would add obvious stresses to LooCo’s analyses, for simplicity, we do not perform LooCo’s analyses iteratively for outer calls in the analyzed library. In practice, one may choose to perform LooCo’s loosening analyses with the control of certain depths for its loosening analyses along the dependency tree. Moreover, due to the request of constructing call graphs, LooCo is suitable when both source codes for projects and libraries are available at the moment. We are working on adapting LooCo for analyzing third-party libraries with binary releases only.

SemVer Comparison. Semantic versioning (SemVer) [22] uses structured versioning schemes (MAJOR.MINOR.PATCH) to suggest loosening plans for developers, which is similar to our approach. We discuss their differences from two aspects. First, in order to loosen, SemVer indeed relies on developers to first follow certain specifications in versioning during the development, which can hardly hold in practice. In existing work [30], around 75%/30% libraries are observed to violate SemVer specifications in some/all versions, thus unreliable to obtain compatible versions for loosening directly in this way. When we attempt to loosen the studied 47 projects with built-in test suites as aforementioned following SemVer, we observed that loosening using SemVer suggests versions with obviously inconsistent behaviors for 19.1% projects, exhibiting its unreliable loosening.

Second, since SemVer normally suggests loosening version constraints to new versions with consistent MAJOR or greater MINOR numbers, this can be quite general and not sensitive with respect to each dependent project, possibly too strict. Owing to LooCo's project-aware analyses, our loosening plan can be more delicate and relaxed with higher possibilities for loosening. For example, in our evaluation, LooCo can loosen an original version constraint "`telepot==12.7`" for pair#4297 to versions a smaller version of MINOR, e.g., `telepot` 12.0, or to versions with different MAJOR numbers, e.g., `telepot` 11.0 and 10.5.

7 RELATED WORK

Software Ecosystem and Library Versioning. Third-party libraries and sheets contribute to code and data reuse in developing intelligent and evolving software [48, 52, 58]. In practice, projects co-evolve with each other based on their intra-dependencies in software ecosystems and many existing work studied these dependencies. Kula et al. [42] investigated dependency migrations for Java and observed severe outdated dependency problems [52]. Latendresse et al. [43] studied the differences between installed dependencies and production dependencies in JavaScript projects and suggested that production dependencies should be given a higher priority. Hejderup et al. [34] constructed an exhaustive dependency CG for the ecosystem. Zhang et al. [57] studied API compatibility issues by analyzing the characteristics of Python API usages. Xing et al. [59] and Johannes et al. [35] emphasized on avoiding API-usage issues like API-breaking changes [25], and Eric [37, 38] focused on API-breaking changes by identifying problematic codes during versioning. Jia et al. [40] proposed DepOwl to avoid incompatible library versions, e.g., with backward or forward incompatible changes (e.g., removing or adding an interface). Ma et al. [45] focused on cross-project bugs in the ecosystem and how they affect downstream and upstream projects. Wu et al. [55] studied potential threats of upstream vulnerabilities to downstream projects in the Maven ecosystem by a fine-grained analysis of calls from downstream projects to upstream projects.

Dependency Inference and Conflict Resolution. Towards inferring dependencies for Python projects, Ye et al. [56] proposed PyEGo to construct knowledge graphs and inference environment dependency for Python projects. Similarly, PyCRE [27] infers more exhaustively by analyzing the runtime environment via domain knowledge graphs. Based on inferred dependencies, different strategies target resolving potentially conflicting issues. Wang et al. [53] detected typical patterns of DC issues in PyPI ecosystem, and Artho et al. [24] conducted an empirical study on conflict defects and made recommendations for prevention and detection. Jafari et al. [39] surveyed a classification of dependency smells in JavaScript and proposed DependencySniffer for detecting dependency smells. Cao et al. [26] investigated three types of dependency smells in Python projects, i.e., missing dependency, bloated dependency, and version constraint inconsistency, and proposed PyCD to extract such dependencies respectively. `Maven` [10] leverages its package management for the Java ecosystem by a nearest-win strategy to resolve dependency for DC issues, which is also studied by Foo et al. [30] for resolution via an efficient static checking. `Pip` [12] optimized its dependency-solving strategy with backtracking, and smartPip [50] further optimizes it with a powerful strategy with controllable costs. In addition, some DC issues are due to conflicts with existing local dependencies, in which case they can be solved by having the project use a separate Python environment [53]. Different from existing work that sticks to original version constraints, we choose to automatically refine them for better quality to resolve DC building failures. Therefore, we believe that LooCo can be a great complement to existing work with its refined version constraints.

Static Analyses for Python Projects. LooCo constructs call graphs for Python projects. Despite Python popularities, there are only a few Python static analyzers [28, 32, 33, 44, 49]. Pyan [28] parses ASTs to extract Python projects' call graphs but faces drawbacks for inter-procedural flows of values and module imports, later optimized with visualization in Code2graph [32, 33]. Depends [31] obtains syntactical relations among source entities to generate call graphs more precisely but does not support higher-order programming. The state-of-the-art PyCG [49] performs the best but does not support on-demand call graph generation. We realizes LooCo's static analyses with comparable performance to it.

8 CONCLUSION

Python projects commonly suffer DC issues and thus incurred building failures. In this paper, we have proposed a constraint refinement approach LooCo to automatically loosen version constraints for dependent libraries, assisting in resolving building failures without sacrificing libraries' behavioral consistency. The approach was inspired by our empirical findings from real-world DC issues and characteristics of their associated library version constraints, and exhibited promising performance by effective loosening (avg. 5.5 versions expanded) and efficient execution (0.4s overhead per loosening). Such automatic constraint loosening contributed to significant resolution of DC building failures, by transforming 54.8% originally unsolvable cases into solvable ones and producing more solutions (21 more on average) for original solvable cases. Nevertheless, LooCo is currently still restricted by syntactic consistency in its code analysis (conservative in expanding versions), and we plan to extend its capability by exploring potential semantic consistency (finding further loosening space) in future.

9 DATA AVAILABILITY

The source code of LooCo and other resources are available on [9].

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers for their insightful comments and suggestions. This work was supported by the Natural Science Foundation of China under Grant No. 61932021, and the Natural Science Foundation of Jiangsu Province under Grants (No. BK20202001 and BK20220771). The authors would also like to thank the support from the Fundamental Research Funds for the Central Universities of China (020214380102 and 020214912220), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

REFERENCES

[1] 2023. Code2flow. https://code2flow.com/, Accessed: 2023-08-24.

[2] 2023. Conda. https://docs.conda.io/, Accessed: 2023-08-24.

[3] 2023. Dfflib. https://docs.python.org/3/library/difflib.html, Accessed: 2023-08-24.

[4] 2023. Filecmp. https://docs.python.org/3/library/filecmp.html, Accessed: 2023-08-24.

[5] 2023. Issue #131. https://github.com/garyd203/ssmash/issues/39, Accessed: 2023-08-24.

[6] 2023. Issue #272. https://github.com/chaoss/grimoirelab-manuscripts/issues/136, Accessed: 2023-08-24.

[7] 2023. Issue #56. https://github.com/ivanubi/api-indotel/issues/2, Accessed: 2023-08-24.

[8] 2023. Issue #99. https://github.com/KnowledgeLinks/rdfframework/issues/24, Accessed: 2023-08-24.

[9] 2023. LooCo website. https://agnes-u.github.io/LooCo/, Accessed: 2023-08-24.

[10] 2023. Maven. https://mvnrepository.com/repos, Accessed: 2023-08-24.

[11] 2023. PEP specifications. https://peps.python.org/, Accessed: 2023-08-24.

[12] 2023. PIP backtracking. https://pip.pypa.io/en/stable/topics/dependency-resolution/, Accessed: 2023-08-24.

[13] 2023. PIP freeze. https://pip.pypa.io/en/stable/cli/pip_freeze/, Accessed: 2023-08-24.

[14] 2023. PIP installer. https://pypi.org/project/pip/, Accessed: 2023-08-24.

[15] 2023. PIP legacy. https://pip.pypa.io/en/stable/user_guide/, Accessed: 2023-08-24.

[16] 2023. PyCG issues. https://github.com/vitsalis/PyCG/issues, Accessed: 2023-08-24.

[17] 2023. Python Package Index. https://pypi.org/, Accessed: 2023-08-24.

[18] 2023. Report #2135. https://github.com/andylokandy/rqalpha-mod-minute/issues/1, Accessed: 2023-08-24.

[19] 2023. Report #2688. https://github.com/gkeep/spotify-stats/issues/3, Accessed: 2023-08-24.

[20] 2023. Report #282. https://github.com/stratosphereips/StratosphereLinuxIPS/issues/163, Accessed: 2023-08-24.

[21] 2023. Report #700. https://github.com/yihong0618/running_page/issues/282, Accessed: 2023-02-03.

[22] 2023. SemVer. https://semver.org/, Accessed: 2023-08-24.

[23] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*. 547–551.

[24] C. Artho, R. D. Cosmo, K. Suzaki, R. Treinen, and S. Zacchiroli. 2012. Why Do Software Packages Conflict?. In *Proceedings of the 2012 IEEE/ACM 9th International Conference on Mining Software Repositories (MSR 2012)*.

[25] Aline Brito, Laerte Xavier, Andre Hora, and Marco Valente. 2018. APIDiff: Detecting API breaking changes. In *Proceedings of the 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER 2018)*. 507–511.

[26] Yulu Cao, Lin Chen, Wanwangying Ma, Yanhui Li, Yuming Zhou, and Linzhang Wang. 2023. Towards Better Dependency Management: A First Look at Dependency Smells in Python Projects. *IEEE Transactions on Software Engineering* 49, 4 (2023), 1741–1765. https://doi.org/10.1109/TSE.2022.3191353

[27] Wei Cheng, Xiangrong Zhu, and Wei Hu. 2022. Conflict-aware Inference of Python Compatible Runtime Environments with Domain Knowledge Graph.

[28] Fraser D., Horner E., Jeronen J., and Massot P. [n. d.]. Pyan3: Offline call graph generator for Python 3. https://github.com/davidfraser/pyan.

[29] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *Proceedings of the 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR 2019)*. 349–359.

[30] Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*. ACM Press, Lake Buena Vista, FL, USA, 791–796.

[31] Zhang G. and Wuxia J. 2023. Depends. https://github.com/multilang-depends/depends, Accessed: 2023-08-24.

[32] G. Gharibi, R. Alanazi, and Y. Lee. 2018. Automatic Hierarchical Clustering of Static Call Graphs for Program Comprehension. In *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data)*. 4016–4025.

[33] G. Gharibi, R. Tripathi, and Y. Lee. 2018. Code2graph: Automatic Generation of Static Call Graphs for Python Source Code. In *Proceedings of the 2018 IEEE/ACM International Conference on Automated Software Engineering (ASE 2018)*. 880–883.

[34] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software Ecosystem Call Graph for Dependency Management. In *Proceedings of the ACM/IEEE 40nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER 2018)*. ACM, New York, NY, USA, 101–104.

[35] Johannes Henkel and Amer Diwan. 2005. CatchUp! Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)* (St. Louis, MO, USA). Association for Computing Machinery, New York, NY, USA, 274–283.

[36] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME 2018)*. 217–227.

[37] Eric Horton and Chris Parnin. 2019. DockerizeMe: Automatic Inference of Environment Dependencies for Python Code Snippets. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE 2019)*. IEEE, Montreal, QC, Canada, 328–338.

[38] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. IEEE, San Diego, CA, USA, 477–488.

[39] Abbas Javan Jafari, Diego Elias Costa, Rabe Abdalkareem, Emad Shihab, and Nikolaos Tsantalis. 2022. Dependency Smells in JavaScript Projects. *IEEE Trans. Softw. Eng.* 48, 10 (oct 2022), 3790–3807. https://doi.org/10.1109/TSE.2021.3106247

[40] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *Proceedings of the 43rd International Conference on Software Engineering (ICSE 2021)*. 86–98.

[41] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl. 2017. Structure and Evolution of Package Dependency Networks. In *Proceedings of the 2017 IEEE/ACM International Conference on Mining Software Repositories (MSR 2017)*. 102–112.

[42] Kula, R. Gaikovina, German, M. Daniel, Ouni, Ali, Ishio, Takashi, Inoue, and Katsuro. 2018. Do Developers Update Their Library Dependencies? An Empirical Study on the Impact of Security Advisories on Library Migration. *Empirical Software Engineering* 23 (2018), 384–417.

[43] Jasmine Latendresse, Suhaib Mujahid, Diego Elias Costa, and Emad Shihab. 2023. 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* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 73, 12 pages. https://doi.org/10.1145/3551349.3556896

[44] Y. Li. 2019. Empirical Study of Python Call Graph. In *Proceedings of the 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE 2019)*. 1274–1276.

[45] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. 2020. Impact Analysis of Cross-project Bugs on Software Ecosystems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020)*. ACM, Seoul South Korea, 100–111.

[46] Fabio Mancinelli, Jaap Boender, Roberto di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. 2006. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*. IEEE, Tokyo, 199–208.

[47] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, Virtual Envent, Denmark, 439–451.

[48] Yuen Tak YU Sau-Fun TANG Pak-Lok POON, Man Fai LAU. 2024. Spreadsheet quality assurance: a literature review. *Frontiers of Computer Science (FCS)* 18, 2 (2024), 182203.

[49] Vitalis Salis, Thodoris Sotiropoulos, Panos Louridas, Diomidis Spinellis, and Dimitris Mitropoulos. 2021. PyCG: Practical Call Graph Generation in Python. In *Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*. IEEE, Madrid, ES, 1646–1657.

[50] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu Shu, and Guoqing Li. 2022. SmartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE 2022)*. Oakland Center, Michigan, United States.

[51] Jiawei Wang, Li Li, Kui Liu, and Haipeng Cai. 2020. Exploring How Deprecated Python Library APIs Are (not) Handled. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*. ACM, Virtual Event USA, 233–244.

[52] 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 *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME 2020)*. 35–45.

[53] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE 2020)*. ACM, Seoul South Korea, 125–135.

[54] R. Widyasari, Q. S. Sheng, C. Lok, H. Qi, and E. L. Ouh. 2020. BugsInPy: A Database of Existing Bugs in Python Programs to Enable Controlled Testing and Debugging studies. In *Proceedings of the 28th ACM Joint European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020). 1556–1560.

[55] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem.

[56] Hongjie Ye, Wei Chen, Wensheng Dou, Guoquan Wu, and Jun Wei. 2022. Knowledge-Based Environment Dependency Inference for Python Programs. (May 2022), 12.

[57] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *Proceedings of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*. 81–92.

[58] Ze-Lin Zhao, Di Huang, and Xiao-Xing Ma. 2022. TOAST:Automated Testing of Object Transformers in Dynamic Software Updates. *Journal of Computer Science and Technology (JCST)* 37, 1 (1 2022), 50–66.

[59] Zhenchang Xing and E. Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (Dec. 2007), 818–836.

Received 2023-03-02; accepted 2023-07-27