Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem

Ying Wang
wangying@swc.neu.edu.cn
Software College, Northeastern University, China

Ming Wen*
mwenaa@hust.edu.cn
School of Cyber Science and Engineering, HUST, China

Yepang Liu*
liuyp1@sustech.edu.cn
Department of Computer Science and Engineering, SUSTech, China

Yibo Wang, Zhenming Li, Chao Wang
{wybneu,lzmneu,wangcneu}@163.com
Software College, Northeastern University, China

Hai Yu
yuhai@mail.neu.edu.cn
Software College, Northeastern University, China

Shing-Chi Cheung
scc@cse.ust.hk
Department of Computer Science and Engineering, HKUST, China

Chang Xu
changxu@nju.edu.cn
State Key Lab for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, China

Zhiliang Zhu
zzl@mail.neu.edu.cn
Software College, Northeastern University, China

ABSTRACT

The PyPI ecosystem has indexed millions of Python libraries to allow developers to automatically download and install dependencies of their projects based on the specified version constraints. Despite the convenience brought by automation, version constraints in Python projects can easily conflict, resulting in build failures. We refer to such conflicts as Dependency Conflict (DC) issues. Although DC issues are common in Python projects, developers lack tool support to gain a comprehensive knowledge for diagnosing the root causes of these issues. In this paper, we conducted an empirical study on 235 real-world DC issues. We studied the manifestation patterns and fixing strategies of these issues and found several key factors that can lead to DC issues and their regressions. Based on our findings, we designed and implemented Watchman, a technique to continuously monitor dependency conflicts for the PyPI ecosystem. In our evaluation, Watchman analyzed PyPI snapshots between 11 Jul 2019 and 16 Aug 2019, and found 117 potential DC issues. We reported these issues to the developers of the corresponding projects. So far, 63 issues have been confirmed, 38 of which have been quickly fixed by applying our suggested patches.

*Ming Wen and Yepang Liu are the corresponding authors of this paper. HUST, SUSTech, and HKUST are short for Huazhong University of Science and Technology, Southern University of Science and Technology, and The Hong Kong University of Science and Technology, respectively.

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380426

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*;

KEYWORDS

Python, dependency conflicts, software ecosystem

ACM Reference Format:
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 42nd International Conference on Software Engineering, Seoul, Republic of Korea, May 23–29, 2020 (ICSE '20),* 11 pages.
https://doi.org/10.1145/3377811.3380426

1 INTRODUCTION

Python projects are commonly shared as third-party libraries in a server-side central repository PyPI [42], and reused by other projects with a client-side library installer `pip` [36, 46, 55]. By June 2019, the PyPI ecosystem (PyPI for short) has indexed over 1.43 million Python libraries together with their metadata (e.g., version information, dependencies on other libraries, etc.).

To use a library on PyPI, developers need to specify the desired version constraint [51] in a configuration script such as `setup.py` and `requirements.txt` [44]. When a library is reused by another project, this library and other libraries on which it depends will be automatically installed at the project's build time. The automation smartly combines a server-side central repository and a client-side library installer to manage library dependencies. It considerably simplifies the build process of Python projects. Besides, the version constraint mechanism for a required library allows developers to restrict the dependencies to a set of compatible versions and enables automatic library evolution [3]. However, such automation comes with the risk of potential Dependency Conflict (DC) issues, which

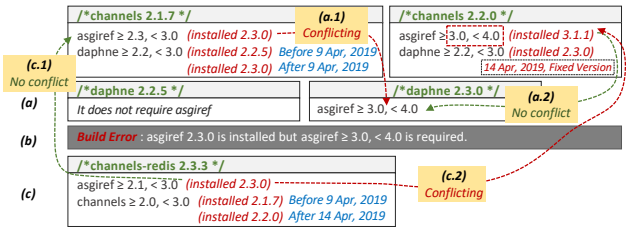

Figure 1: Illustrative examples of dependency conflict issues

can cause build failures when the installed version of a library violates certain version constraints on the library.

Figure 1 gives a real example: issue #1277 [6] in `channels`. As shown in `channels` 2.1.7's configuration script, it directly requires libraries `asgiref` (version constraint: $\langle \geq 2.3 \wedge < 3.0 \rangle$) and `daphne` (version constraint: $\langle \geq 2.2 \wedge < 3.0 \rangle$). Note that when downloading a library, the `pip` installer always chooses the latest version on `PyPI` that satisfies the library's version constraint [37]. No DC issues occurred when `channels` 2.1.7 was built before 9 Apr 2019. Both `asgiref` 2.3.0 and `daphne` 2.2.5 selected for the build satisfy the concerned constraints. However, issue #1277 [6] arose after 9 Apr 2019 when `channels` 2.1.7 was built via selecting the newly released library `daphne` 2.3.0, which additionally requires library `asgiref` (version constraint: $\langle \geq 3.0 \wedge < 4.0 \rangle$). The DC issue (the red curve a.1) happened because `pip` selected `asgiref` 2.3.0 to satisfy the direct dependency constraint $\langle \geq 2.3 \wedge < 3.0 \rangle$, but this version violated the constraint $\langle \geq 3.0 \wedge < 4.0 \rangle$ specified in `daphne` 2.3.0. This issue caused a build error as shown in Figure 1(b).

To fix the issue, `channels`'s developers released version 2.2.0 on 14 Apr 2019, which updated the requirement on `asgrief`'s version to $\langle \geq 3.0 \wedge < 4.0 \rangle$. This update led to the installation of `asgrief` 3.1.1 (the latest version under 4.0) when building `channels` 2.2.0, thus resolving the failure (the green curve a.2). However, this fix induced another DC issue in `channels-redis` (issue #152 [7]), as Figure 1(c) shows. After the upgrade of `channels`, to build `channels-redis` 2.3.3, `pip` still selected `asgiref` 2.3.0 to satisfy the direct dependency constraint $\langle \geq 2.1 \wedge < 3.0 \rangle$. Unfortunately, this version violates the constraint $\langle \geq 3.0 \wedge < 4.0 \rangle$ that is transitively introduced by `channel` 2.2.0, causing a build failure (red curve c.2).

To understand the scale of DC issues in Python projects and their characteristics, we empirically studied 235 DC issues in 124 popular Python projects, which were reported on `GitHub` in the last five years. We explored the following two research questions:

- **RQ1 (Manifestation Patterns)**: *How do DC issues manifest themselves in Python projects? Are there common patterns that can be leveraged for automated diagnosis of these issues?*
- **RQ2 (Fixing Strategies)**: *How do developers fix DC issues in Python projects? Are there common practices that can be leveraged for automated repair of these issues?*

Through investigating the research questions, we observe that DC issues mainly arise from conflicts caused by remote dependency updates or local environments (see Section 3.2). We also found common strategies for fixing DC issues and key factors that can lead to DC issues and their regressions (see Section 3.3).

As a real-world Python developer commented on the report of `pyenv` issue #3118 [21], the dependency resolution in the Python world is far from being easy. The difficulties are mainly attributed to the complex dependencies across projects. Developers often specify version constraints on the dependent libraries of their projects without considering the constraints specified in other projects. To be specific, we summarize three major challenges as follows.

First, the version of a library installed for a Python project can vary over time. Recall that for each required library of a project, `pip` will install its latest version satisfying the concerned constraint. Therefore any update of libraries on `PyPI` can affect the version of the libraries installed for the downstream projects (i.e., the projects that depend on these libraries), causing potential build failures.

Second, when a library updates its version constraints on other libraries, its downstream projects might be affected. The impact can be further propagated to a wide range of projects.

Third, it is difficult for Python developers to obtain a full picture of their projects' dependencies with version constraint information. State-of-the-art tools like `pipenv` and `Poetry` only show which libraries have been installed, rather than their dependencies.

To address the challenges and help Python developers combat DC issues, we designed a technique, Watchman, which performs a holistic analysis from the perspective of the entire `PyPI` ecosystem, to continuously monitor dependency conflicts caused by library updates. For each library on `PyPI`, Watchman builds a Full Dependency Graph (FDG), a formal model that simulates the process of installing dependencies for Python projects. The FDGs can be incrementally updated as the libraries evolve on `PyPI`. Watchman then analyzes them to detect and proactively prevent DC issues. Since FDGs record full dependencies of Python projects with version constraints, they can also provide useful diagnostic information to help developers understand the root causes of the detected DC issues, thus facilitating issue fixing.

To evaluate Watchman, we played back the evolution history of all libraries on `PyPI`, from 1 Jan 2017 to 30 Jun 2019 and deployed Watchman to detect DC issues. After analyzing `PyPI` snapshots during this time period, Watchman detected 515 DC issues and 502 (97.5%) of them were indeed fixed by developers during the evolution of the libraries. To evaluate the usefulness of Watchman, we ran it to monitor dependency conflicts for the `PyPI` ecosystem between 11 Jul 2019 and 16 Aug 2019. During the time period, it detected and reported 117 previously-unknown DC issues, 63 of which (53.8%) have been confirmed by developers. Further, 38 (60.3%) confirmed issues have been fixed by applying our suggested patches. Developers also expressed great interests in Watchman. In summary, our work makes three major contributions:

- **Originality:** To the best of our knowledge, we conducted the first empirical study of DC issues in open-source Python projects. Our findings help understand the characteristics of DC issues and provide guidance to future studies related to this topic.
- **Dataset:** We release the dataset for empirical study, comprising 235 DC issues collected from 124 real-world Python projects, to facilitate future research.
- **Technique:** We proposed a formal model to simulate the build process of Python projects and developed a DC issue diagnostic technique Watchman (**http://www.watchman-pypi.com/**) based on the model. Experimental results show that Watchman can monitor the entire `PyPI` ecosystem and detect DC issues with a high precision.

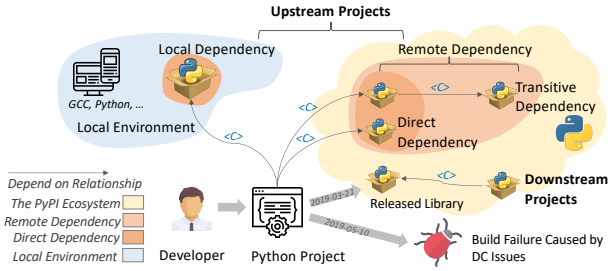

Figure 2: Dependencies of a Python project

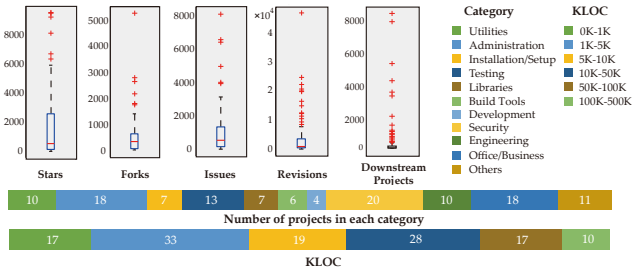

Figure 3: Statistics of the projects used in our empirical study

2 PRELIMINARIES

2.1 Dependencies of Python Projects

Figure 2 illustrates the concept of Python project dependencies. Code reuse is pervasive in the Python world, where projects often reuse other projects as libraries. The configuration script of a project P explicitly constrains the versions of *direct dependencies* that P may use. If these direct dependencies further rely on other libraries, such libraries are called *transitive dependencies* of P. In this paper, all direct and transitive dependencies are collectively referred to as the *upstream projects* of P. Correspondingly, we call P a *downstream project* of its dependencies.

Python projects are often developed in a self-contained environment, which can be created by tools such as `virtualenv` [43], `conda` [2], and `pipenv` [38]. When building a Python project, the library installer `pip` downloads most of the required libraries from `PyPI`. We refer to such libraries that need to be downloaded as *remote dependencies*. For each required remote dependency, `pip` downloads it according to its name and version constraint. If multiple releases of a library on `PyPI` satisfy the version constraint, `pip` downloads and installs the latest version of the library [36].

Besides remote dependencies, the development of a Python project can be affected by its *local environment*, including the local development tool chains (e.g., the Python interpreter and GCC) and *local dependencies* (i.e., libraries that are already installed). Local dependencies exist when the development environment is not clean (e.g., the project is not developed in an isolated virtual environment). If any version of a required dependency has been installed locally, `pip` will not download the dependency from `PyPI`.

2.2 Library Version Constraints

To use a library, a Python project needs to specify a constraint on its desired library versions as shown in Figure 2 (i.e., the C annotated on some edges). To facilitate subsequent discussions, we formally define *version constraint* using the grammar below ($version_{id}$ refers to a specific version of a library, e.g., 1.24.1):

$$\begin{aligned} \text{C} &::= \epsilon \mid \text{range} \wedge \text{extra} \mid = version_{id} \\ \text{range} &::= \text{range} \wedge op\ version_{id} \mid op\ version_{id} \\ \text{extra} &::= \epsilon \mid \neq version_{id} \mid \text{extra} \wedge \neq version_{id} \\ op &::= > \mid \geq \mid < \mid \leq \end{aligned} \tag{1}$$

A constraint C could be empty, in which case `pip` will choose to download the latest version of the library from `PyPI` if the library is not installed in the local environment. Developers may also specify a specific version that is desired (e.g., = 1.24.1) or undesired (e.g., $\neq$ 1.24.1). In practice, developers mostly specify a range of versions in a constraint (e.g., $\leq 1.24.1 \wedge > 1.11.0$). To understand how frequent ranges are used in version constraints, we investigated the top 1,000 popular Python projects on `PyPI` based on the number of downstream projects. We found that 92.2% of these projects' direct dependencies are constrained to a range of versions. In comparison, this ratio is only 0.03% for Java projects managed by `Maven` following the same investigation method. Such heavy uses of ranges in version constraints for dependencies make the diagnosis of DC issues in the Python world complicated and challenging (see Section 3).

3 EMPIRICAL STUDY

3.1 Data Collection

Following the data collection process of existing studies [47, 49, 58], we prepared our dataset in two steps.

Step 1: Selecting subjects. To understand the manifestation patterns and fixing strategies of DC issues, we need to study the issue reports (with discussions if any), dependency configuration scripts, issue-fixing patches, and related code revisions. For this purpose, we searched `GitHub` for Python projects that satisfy three conditions: (1) *popular*: having more than 50 stars or forks, (2) *being used as libraries*: containing more than three direct downstream projects, and (3) *well-maintained*: having over 500 code revisions or over 50 issue reports. With this process, we obtained 1,596 open-source Python projects.

Step 2: Identifying DC issues. To locate DC issues in the 1,596 projects, we searched for the issue reports that contain keywords "dependency conflict" or "dependency hell" (case insensitive), filed between Jul 2014 and Jul 2019 (i.e., in the past five years). We obtained 2,593 and 334 issue reports by searching with the two keywords, respectively. Next, we removed duplicates and noises from the search results and kept only those issue reports that satisfy the following three conditions. First, the report is related to valid DC issues. Second, the report contains descriptions of issue root causes. Third, we can find code revisions that fix the reported issue(s) in the concerned project's code repository or there is an explicit consensus on the fixing solutions among developers as documented in the issue report.

Eventually, we obtained 235 DC issues from 124 projects, and 201 of the 235 issues have been fixed. Figure 3 shows the statistics of the 124 projects. As we can see, they are: (1) large in size (around 38 KLOC on average), (2) well-maintained (containing 78 revisions and 92 issues on average), (3) popular (83% of them have over 100 stars), (4) impactful (86% of them have more than 5 direct

downstream projects), and (5) diverse (covering over 10 categories). In the following, we study the 235 issues to answer RQ1–2.

3.2 RQ1: Manifestation Patterns

DC issues in Python projects manifest themselves due to different causes. Our studied issues can be divided into two categories according to whether the issues are caused by *remote dependencies* or *local environment*. In the following, we discuss the manifestation patterns of the issues in detail with illustrative examples.

Finding 1: *211 out of the 235 (89.8%) DC issues that involve the violation of library version constraints were introduced by the updates of remote dependencies on PyPI.*

3.2.1 Pattern A: Conflicts caused by remote dependency updates. The root cause of the 211 issues is that the updates of some remote dependency change the version of the concerned library to be installed by `pip`, which is hardly perceptible to project developers. Suppose that a Python project P requires a library β with a constraint C. If C does not specify an upper bound on β's version (e.g., $C = \langle \geq 3.0 \rangle$), or the specified upper bound is greater than the latest version of β on `PyPI` (e.g., $C = \langle 2.0 \leq \wedge \leq 4.0 \rangle$, while the latest version of β is 3.0), then the version of β used to build P may not be controllable in developers' perspective, meaning that β can be upgraded when there are new versions on `PyPI`. Such upgrading can easily induce DC issues: β's new version may not satisfy the constraints specified by other dependencies of P; the version constraints specified by β for its own libraries may also change in new versions, causing potential conflicts with the constraints for the same libraries introduced by P's other dependencies.

The 211 issues can be further categorized based on where the dependency conflicts come from. Theoretically, conflicts could happen in three different cases: (1) among direct dependencies, (2) between direct dependencies and transitive dependencies, and (3) among transitive dependencies. However, developers usually will not introduce conflicts among direct dependencies by including two conflicting libraries in the configuration script (such mistakes can be easily caught). Indeed, we did not observe any conflicts of the first case. In the following, we discuss the latter two cases.

a. Conflicts between direct and transitive dependencies (139/211). Suppose that a Python project P directly depends on two libraries α and β with the version constraints $C_{P\rightarrow\alpha}$ and $C_{P\rightarrow\beta}$, respectively, and β further depends on α with the version constraint $C_{\beta\rightarrow\alpha}$. In other words, α is not only a direct dependency of P, but also required by other direct dependencies of P (i.e., α can also be seen as a transitive dependency of P). When building P, `pip` will always install the latest version v of the library α that satisfies $C_{P\rightarrow\alpha}$, as α is at the top level of P's dependency tree [36]. If v falls into the version range(s) specified by $C_{\beta\rightarrow\alpha}$, P will be built successfully. However, once α gets updated on `PyPI`, the update may cause `pip` to install another version v' of α. If v' falls out of the range(s) specified by $C_{\beta\rightarrow\alpha}$, P will not be built successfully. For instance, in issue #229 [15], the project `gallery-dl` directly requires the libraries `requests` $\langle \geq 2.11.0 \rangle$ and `urllib3` $\langle \geq\ 1.16 \wedge \neq\ 1.24.1 \rangle$. `pip` installed the version 2.13.0 of `requests`, which also depends on `urllib3` $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$. Things worked smoothly when `gallery-dl` was first released on `PyPI`, as the latest version of `urllib3` at that time was 1.24.2, which satisfies the constraints $\langle \geq 1.16 \wedge \neq 1.24.1 \rangle$ and $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$. However, when the project `urllib3` was updated to 1.25.0 on 18 Apr 2019, `gallery-dl` began to suffer from build failures. This is because when building `gallery-dl`, `pip` will install the latest version (i.e., 1.25.0) of `urllib3`. However, the version 1.25.0 violates the constraint $\langle < 1.25.0 \wedge \geq 1.21.1 \rangle$ specified in `requests`.

b. Conflicts between transitive dependencies (72/211). Suppose that a Python project P directly depends on two libraries α and β, both of which depend on another library θ but with two different version constraints $C_{\alpha\rightarrow\theta}$ and $C_{\beta\rightarrow\theta}$, respectively. If the version v of θ downloaded by `pip` according to $C_{\alpha\rightarrow\theta}$ (suppose that it has a higher priority) also satisfies $C_{\beta\rightarrow\theta}$, the project P can be built successfully. However, since α and β are two separate projects, their dependency relationship on θ may evolve over time. There can be cases where the updates of α or β would result in conflicting version constraints of θ, consequently causing DC issues when building P. We observed 72 such issues in our study. For example, the issue report #3826 [24] of `rasa` documented an incident that a project was forced to introduce multiple version constraints of the library `requests` by its direct dependencies `rasa` and `sagemaker`. The reason is that `rasa` released a new version 1.0.4 and added a constraint $\langle = 2.22.0 \rangle$ on `requests`. However, this constraint is in conflict with another constraint on `requests` $\langle \geq 2.20.0 \wedge < 2.21.0 \rangle$ introduced by `sagemaker`.

Finding 2: *24 out of the 235 (10.2%) DC issues arose due to the conflicts between remote dependencies and the tools/libraries installed in the local environment.*

3.2.2 Pattern B: Conflicts affected by local environment. Such issues can happen when the required tool of a remote dependency is incompatible with the local installed one (e.g., requiring Python 3.7.* but installed Python 3.6.*). They can also happen when the version of a dependency, which is already installed in the local environment, does not satisfy the constraint specified by a remote dependency. Take issue #25316 [17] of `gradient` as an example. The project failed to be built because there was already one version (1.13.3) of the library `numpy` installed in the local environment before the build, and this version is in conflict with the constraint $\langle \geq 1.15 \rangle$ specified by `pandas` v0.24.1, a direct dependency of `gradient`.

3.2.3 Dependency Smells. By further analyzing developers' discussions in the issue reports and the dependency configuration scripts of the project versions that were not affected by the reported issues of Pattern A,[1] we observed several types of "dependency smells". These smells are interesting as they do not immediately cause DC issues but are likely to induce issues as the projects evolve.

Finding 3: *Restricting dependencies to specific versions for common libraries could easily induce DC issues to downstream projects.*

Build failures can easily happen if library version constraints are too restrictive (e.g., only accepting specific versions), especially for those common libraries. 59 of our studied 235 issues belong to this case. For instance, the project `molecule` [41] depends on

[1] We ignored Pattern B issues as they are affected by developers' local environments, which are often unknown to us.

a specific version of `ansible-lint` (i.e., version 3.4.23), a library that is used by many other projects. This makes `molecule`'s downstream projects that also depend on `ansible-lint` particularly sensitive to the updates of `ansible-lint`. We observe that whenever there was a new version of `ansible-lint` released on `PyPI`, `molecule`'s developers would receive requests from downstream projects to upgrade its version constraint on `ansible-lint` (e.g., [8, 11]). As there were too many such requests, `molecule` developers finally chose to update and loosen the version constraint on `ansible-lint` to a range $\langle \geq 4.0.1 \wedge < 5 \rangle$ [39], thus allowing more downstream projects to work well with it.

Finding 4: *DC issues can easily occur when the installed version of a library satisfying one version constraint is close to the upper bound specified in another version constraint.*

67 out of the 235 issues belong to this case. As a library version installed by `pip` in the concerned project is close to the upper bound that another version constraint imposed on the library, updates of the library will likely induce build failures. For instance, the projects that directly require both `request` and `urllib3` have often encountered DC issues (e.g., [23, 27–29]). The reason is that these projects always install the latest version of `urllib3` since the direct dependency constraints on `urllib3` do not set an upper bound. Besides, `request` also depends on `urllib3` with a version constraint $\langle \geq 1.21.1 \wedge < 1.23 \rangle$. These projects were built successfully when `urllib3`'s latest version was 1.22.4, which satisfies $\langle \geq 1.21.1 \wedge < 1.23 \rangle$. However, the installed latest version 1.22.4 was close to the upper bound 1.23. In such cases, DC issues can easily arise when there comes a newer version of `urllib3`.

These findings are useful. We will show that identifying the two types of smells can help perform predictive analysis to proactively prevent DC issues before they cause real build failures.

3.3 RQ2: Fixing Strategies

To answer RQ2, we studied: (1) the patches of the 201 fixed issues, (2) the planned fixing solutions of the remaining 34 issues, and (3) the comments in the issue reports. We observed seven fixing strategies, which altogether resolved 93.6% of our collected issues.

Strategy 1: *Adjusting the version constraints of direct dependencies (98/235).* The conflicts between direct and transitive dependencies were commonly fixed by adjusting the version constraints of direct dependencies to be compatible with those of transitive dependencies. For example, in issue #32 [22] of project `valinor`, there were two conflicting version constraints on the library `pyyaml`. One constraint $\langle \geq 3 \wedge < 5 \rangle$ was directly specified by `valinor`. The other constraint $\langle < 6.0 \wedge \geq 5.1 \rangle$ was transitively introduced by `pyOCD`, a dependency of `valinor`. In such a case, any version of `pyyaml` installed by `pip`, which satisfies the former constraint, will violate the latter one. To fix the problem, the developers of `valinor` revised the version constraint of `pyyaml` to $\langle < 6.0 \wedge \geq 5.1 \rangle$.

Strategy 2: *Upgrading or downgrading the direct dependencies that require conflicting libraries (27/235).* Dependency conflicts between transitive dependencies can be solved by upgrading or downgrading the direct dependencies that introduce the transitive dependencies. Take issue #66 [31] of `zhmcclient` as an example. The two conflicting version constraints $\langle = 4.0.3 \rangle$ and $\langle \geq 4.4 \rangle$ on `coverage` were transitively introduced by `zhmcclient`'s direct dependencies `python-coveralls` $\langle = 2.9.1 \rangle$ and `pytest-cov` $\langle \geq 2.4.0 \rangle$, respectively. Since the installed version `pytest-cov` 2.6.0 added `coverage` $\langle \geq 4.4 \rangle$ as its direct dependency, which caused the conflict, `zhmcclient`'s developers downgraded `pytest-cov` by changing its version constraint to $\langle \geq 2.4.0 \wedge < 2.6.0 \rangle$. After revising the constraint, `pytest-cov` 2.5.1, which requires `coverage` $\langle \geq 3.71 \rangle$, was installed. This constraint $\langle \geq 3.71 \rangle$ is not in conflict with $\langle = 4.0.3 \rangle$ and thus the DC issue was resolved.

Strategy 3: *Coordinating with upstream projects to adjust conflicting version constraints (51/235).* DC issues can also be fixed via coordinating with upstream projects. Take issue #740 [33] of the project `yotta` as an example. Although the conflict can be resolved by adjusting the direct dependency's version constraint (i.e., following Strategy 1), the developers chose to coordinate with the upstream projects to solve the problem. This avoids changing the version of the directly required library.

Strategy 4: *Removing conflicting direct dependencies and keeping the transitive ones (8/235).* When it is difficult to make a project's direct dependencies in line with its transitive ones, developers may choose to remove the conflicting direct dependencies. For example, as described in issue #407 [25] of the project `wandb`, conflicts occurred when an upstream project updated its version constraint on a direct dependency `PyYAML`, and this happened several times. As `wandb` developers had no direct control on the upstream projects, they removed the conflicting direct dependency from the configuration script, and used the transitively introduced one instead.

Strategy 5: *Adding direct dependencies (16/235).* There are cases when the version constraints $C_{\alpha \to \theta}$ and $C_{\beta \to \theta}$ of two conflicting transitive dependencies overlap, meaning that one can find some versions of the concerned library θ to satisfy both constraints. The DC issue in such a case can be resolved by adding θ as a direct dependency with a constraint that entails both $C_{\alpha \to \theta}$ and $C_{\beta \to \theta}$. This will instruct `pip` to install the version specified by the direct dependency that satisfies both the transitive dependencies. For instance, in issue #1586 [9] of `crossbar`, there exist two conflicting transitive dependencies: `urllib3` $\langle < 1.25 \wedge \geq 1.21.1 \rangle$ and `urllib3` $\langle \geq 1.24.2 \rangle$. To resolve the conflict, developers added `urllib3` $\langle \geq 1.24.2 \wedge < 1.25 \rangle$ as a direct dependency to avoid build failures.

Strategy 6: *Upgrading/downgrading development tools (12/235).* The dependency conflicts between the local environment and the remote dependencies are often solved by upgrading or downgrading the development tools (e.g., issue #409 [26] of `bandit`).

Strategy 7: *Creating an isolated environment (8/235).* This is a viable solution for resolving the dependency conflicts between remote and locally installed dependencies. As recommended by developers of `spyder` [35] and `pandas` [18], there are several tools such as `virtualenv` [43], `conda` [2], and `pipenv` [38], which can create virtual environments to isolate the impacts of locally installed dependencies to avoid such DC issues.

There are nine issues that were fixed by restricting the conflicting library to a specific version. However, this is not a good practice and can induce new issues (e.g., [13, 14, 16]) as discussed in Finding 3. The remaining six issues were fixed by specific workarounds.

Table 1 summarizes how each pattern of issues were fixed. From the statistics, we can observe that there can be multiple ways to fix DC issues of a certain pattern. In particular, issues of Pattern A.a

Table 1: Statistics of manifestation patterns and fixing strategies

Pattern \ Strategy	1	2	3	4	5	6	7
A.a	94	9	19	8			
A.b		18	32		16		
B	4					12	8

can be fixed by adopting four different strategies, among which Strategy 1 is the most wildly adopted one. This is because Python developers have full control of the version constraints of their projects' direct dependencies. If adopting Strategy 1 will cause side effects such as security loopholes, developers may solve the conflicts by upgrading or downgrading the direct dependencies of their projects (Strategy 2) or coordinating with upstream projects to adjust conflicting version constraints (Strategy 3). For Pattern A.b, developers often adopt Strategies 2, 3, and 5 to resolve the issues. Issues of Pattern B are mainly resolved via dealing with the local environments. Due to the page limit, we do not make further discussions and we summarize our observations in the following.

Finding 5: *There can be multiple fixes for a DC issue. The solutions can be affected by the issue's manifestation pattern, the topological structure of the project's dependency graph, `pip`'s installation rules, and the interference between the version constraints of upstream projects and those of downstream projects.*

4 DEPENDENCY CONFLICT DIAGNOSIS

In view of many DC issues induced by complex dependencies among upstream and downstream projects on PyPI, we further propose a technique, Watchman, to continuously monitor dependency conflicts from the perspective of the entire ecosystem.

Figure 4 gives an overview of our technique. A major challenge is to perform a holistic analysis of the huge number of projects on PyPI and model their interdependent relationships, which are subject to change over time. To address the challenge, Watchman first collects the metadata for each library version, including its direct dependencies with version constraints and their declaration orders. Second, it consolidates the metadata of all libraries hosted on PyPI into a single repository to enable the analysis of the interference between the version constraints across upstream and downstream projects. Then, by continuously monitoring library release information on PyPI, Watchman synchronously updates the metadata repository to precisely model the dependency relationships. For the captured library updates, Watchman uses a depth-first searching strategy to identify the affected downstream projects. It also performs a breadth-first search of the metadata repository to construct a full dependency graph for each potentially affected downstream project, according to the library installation mechanism of `pip`. Finally, Watchman performs the automatic DC issue diagnosis.

4.1 Constructing Metadata Repository

To model the dependency relationships among libraries, Watchman uses the *metadata structure* to capture the version constraints of the direct dependencies of each library version and the declaration orders of these direct dependencies. For ease of understanding, in the subsequent discussions, we shall use lowercase Greek letters to denote libraries and superscripts to denote versions.

Algorithm 1: Identifying Affected Downstream Projects

```
Input: L_up and G
Output: L_af
1 L_af ← {};
2 foreach ζ^v ∈ L_up do
3     identifyAffectedLibrary(ζ^v, L_af, G);
4 Function identifyAffectedLibrary(ζ^v, L_af, G)
5 foreach G(δ^u) = (D, R, P) ∈ G do
6     if ζ ∈ D && v satisfies the constraint C_{δ^u→ζ} then
7         L_af ← L_af ∪ {δ^u};
8         identifyAffectedLibrary(δ^u, L_af, G);
```

Definition 1 (Metadata Structure): For a library version ζ^v, i.e., the version v of library ζ, Watchman captures a collection of information $G(\zeta^v) = (D, R, P)$, where

- $D = \{\alpha, \beta, \gamma \cdots\}$ is a set of direct dependencies of ζ^v.
- $R = \{C_{\zeta^v \to \delta} | \ \delta \in D\}$, where $C_{\zeta^v \to \delta}$ denotes the version constraint on the dependency δ specified by ζ^v.
- P maps each dependency $\delta \in D$ to its declaration order.

In our experiments to detect unknown DC issues, Watchman extracted 1,423,291 versions of 191,787 distinct libraries from a snapshot of PyPI on 15 Jun 2019. For each library version $\zeta^v \in L$, where L represents all library versions, it obtained the structured metadata $G(\zeta^v)$ via analyzing the dependency configuration script of ζ^v. Such metadata of all extracted library versions formed an initial *metadata repository* $\mathcal{G}$, which is defined as $\{G(\zeta^v) | \zeta^v \in L\}$. This repository enables the queries of dependency relationships among all upstream and downstream projects on PyPI.

4.2 Analyzing the Impacts of Library Updates

The analysis mainly consists of two steps as explained below.

Step 1: Monitoring library updates. Library updates on PyPI often cause DC issues. There are two types of library updates on PyPI: *new versions of an existing library being released* and *new libraries being released*. Watchman computes L_{up} by monitoring the two types of library updates on a daily basis. For each library version $\zeta^v \in L_{up}$, Watchman collects the metadata $G(\zeta^v)$ and adds it to the repository $\mathcal{G}$. In this manner, the metadata repository $\mathcal{G}$ can be synchronized with the evolution of the libraries on PyPI.

Step 2: Identifying affected downstream projects. Watchman performs backward search for identifying the set of downstream projects affected by L_{up}, denoted L_{af}, following the process as described in Algorithm 1. The algorithm works as follows. First, it initializes L_{af} to an empty set (Line 1). For each library $\zeta^v \in L_{up}$, Watchman analyzes which libraries in the ecosystem may be directly affected by the update via calling the function identifyAffectedLibrary (Lines 2–3), which takes ζ^v, L_{af}, and $\mathcal{G}$ as input and updates L_{af} when needed. For each piece of metadata $G(\delta^u) = (D, R, P)$ in $\mathcal{G}$, if ζ is directly referenced by δ^u (i.e., $\zeta \in D$) and the version number v satisfies the version constraint $C_{\delta^u \to \zeta}$, then δ^u is possibly affected by ζ^v and thus added to L_{af}. Then, Watchman performs a depth-first search to recursively find more downstream projects affected by δ^u and update L_{af} accordingly.

4.3 Detecting DC Issues

As discussed earlier, the topological structure of a Python project's dependency tree determines the installed library versions. In order

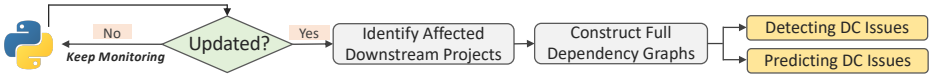

Figure 4: The overall architecture of WATCHMAN

to diagnose DC issues for each library version $\zeta^v \in L_{af}$, we need to analyze the relationships among all library versions that would be installed by `pip` to build ζ^v. To capture such relationships, we propose a formal model *Full Dependency Graph* (FDG).

Definition 2 (Full Dependency Graph): The full dependency graph of a library version ζ^v, denoted $FDG(\zeta^v)$, is a three-tuple, (N, E, FR), where

- $N = N' \cup \{\zeta^v\}$ is the set of nodes in the graph, and N' denotes a set of library versions that `pip` installs for building ζ^v. The libraries here include both direct and transitive dependencies.
- $E = \{\langle \alpha^x, \beta^y \rangle | \alpha^x, \beta^y \in N\}$ is a set of directed edges, where the edge from α^x to β^y represents that the version x of library α directly depends on library β.
- FR maps each edges $e = \langle \alpha^x, \beta^y \rangle \in E$ to the version constraint that the library version α^x sets on the library β, i.e., $C_{\alpha^x \to \beta}$.

Note that the FDG of a library version may change overtime when the library's upstream projects are updated on `PyPI`. Algorithm 2 describes the process of constructing the FDG for a library version ζ^v. WATCHMAN constructs $FDG(\zeta^v)$ following `pip`'s breadth-first installation strategy: `pip` first installs direct dependencies for a project, and then installs dependencies at the next level according to the project's dependency tree, and this process continues until all dependencies are installed. In the algorithm, we use a queue named *Queue* to record the order of traversing and installing dependencies, and ζ^v is initially added to the queue. When visiting each dependency α^x in *Queue*, WATCHMAN first retrieves its metadata $G(\alpha^x) \equiv (D, R, P)$. It then tries to add each dependency β in D to the FDG. If β has not yet been loaded (or installed), WATCHMAN determines the version to be loaded based on constraint $C_{\alpha^x \to \beta}$ (recorded in R) following `pip`'s installation rules (Line 8). N and *Queue* are then updated accordingly (Line 9). If β has already been added to the FDG, WATCHMAN will retrieve the loaded version (Line 11). A new edge $\langle \alpha^x, \beta^y \rangle$ is then added to E (Line 12). The algorithm uses another queue $VisitedEdges$ to record the order in which the edges are traversed (Line 13). WATCHMAN also sets the version constraint of this edge (Line 14), which can be retrieved from R. After traversing all dependencies in *Queue*, the FDG of a library is completely constructed.

DC Issue Detection. WATCHMAN detects DC issues by analyzing the FDG of each project $\zeta^v \in L_{af}$ in the following steps. First, WATCHMAN traverses $FDG(\zeta^v)$ and locates those nodes with multiple incoming edges. A node has multiple incoming edges when there are multiple projects that directly depend on the library represented by the node. Next, for each such node α^x, WATCHMAN analyzes the set of its incoming edges, denoted E_α. Note that there is one edge e in E_α that is traversed first when constructing $FDG(\zeta^v)$. Suppose that x is the latest version number of the library α that satisfies the constraint $FR(e)$. To detect DC issues, WATCHMAN checks x against the set of constraints associated with other edges in E_α, i.e., $\{FR(e') \mid e' \in E_\alpha \backslash \{e\}\}$. If x violates any such constraints, WATCHMAN will report a DC issue to the project ζ.

Algorithm 2: Constructing FDG via Breath-First Search

Input: ζ^v and $\mathcal{G}$
Output: $FDG(I^v) = (N, E, FR)$

```
1  N ← {ζ^v}; E ← {}; FR ← {};
2  Queue.add(ζ^v); Loaded ← {ζ}; VisitedEdges ← {};
3  while !Queue.isEmpty() do
4      α^x ← Queue.pop(); Loaded ← Loaded ∪ {α};
5      G(α^x) ≡ (D, R, P) ← getMetadata(α^x, G);
6      foreach β ∈ D do
7          if β ∉ Loaded then
8              β^y ← getToLoadVersion(β, C_{α^x→β});
9              N ← N ∪ {β^y}; Queue.add(β^y);
10         else
11             β^y ← getLoadedVersion(β, N);
12         E ← E ∪ {⟨α^x, β^y⟩};
13         VisitedEdges.add(⟨α^x, β^y⟩);
14         FR(⟨α^x, β^y⟩) ← C_{α^x→β};
```

4.4 Predictive Analysis for DC Issues

The constructed FDGs by WATCHMAN can also enable us to perform predictive analysis for proactive prevention of DC issues via detecting the two types of smells discussed earlier (Findings 3–4).

Type 1: *Restricting a dependency to a specific version.* If a project restricts a dependency to a specific version, its downstream projects may suffer from DC issues. Specifically, DC issues may arise if the following conditions hold:

(1) There is a version v of project ζ, denoted ζ^v, that restricts its direct dependency α to a specific version x.

(2) There is a version y of a downstream project β that depends on both ζ and α, and ζ^v and α^x are the installed library versions for β^y at the time of analysis.

Let DP be the set of downstream projects (e.g., β) thus found. The larger $|DP|$ is, the more likely that DC issues can arise. This is because each project in DP independently sets its own version constraints on α. If ζ^v only accepts the version x of α, the possibility that the constraint $\langle = x \rangle$ conflicts with other constraints on α set by the projects in DP is high, especially when DP is large. In our experiments, we will warn the developers of the project ζ, if $|DP|$ is larger than a threshold value, which is set empirically.

Figure 5(a) gives an illustrative example. In project $\texttt{C}^{2.0}$, the constraint for library `A` is restricted to $\langle = 2.0 \rangle$. In addition, `C`'s downstream project $\texttt{B}^{5.0}$ depends on both $\texttt{C}^{2.0}$ and $\texttt{A}^{2.0}$. In such a case, it is very likely that the restrictive constraint `C` sets on `A` would cause conflicts for `B` (e.g., when `A` gets updated on `PyPI`). The risk of conflicts gets higher if we find more such downstream projects. WATCHMAN will find such cases and suggest project `C`'s developers to relax its constraint on `A`, to avoid potential DC issues.

Type 2: *The installed version of a library is close to the upper bound specified in the version constraint.* If the installed version of a library satisfies the concerned version constraint but is close to the upper bound specified in the constraint, build failures may occur when the library evolves. WATCHMAN deems that a project ζ^v has a potential DC issue, if the following conditions hold:

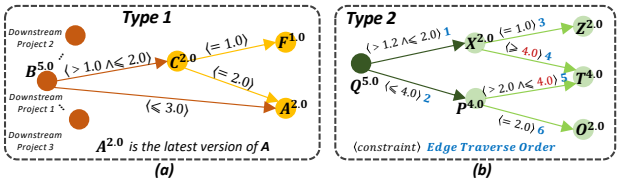

Figure 5: Illustrative examples of potential DC issues

(1) In $FDG(\zeta^v) = (N, E, FR)$, there exists a node α^u with multiple incoming edges, where α is a dependency of ζ^v and u is the installed version of α. Let E_α be the set of incoming edges to α^u.

(2) The version constraint of the first traversed edge e in E_α does not specify an upper bound on α (e.g., the constraint is of the form $\langle \geq y \rangle$) or the specified upper bound is greater than α' latest version z on PyPI. In this case, any updates of α on PyPI will affect the version of α to be installed.

(3) There exists another edge e' in E_α ($e' \neq e$), of which the associated constraint $FR(e')$ specifies an upper bound on the version of α (e.g., the constraint is of the form $\langle \leq x \rangle$) and the upper bound is greater than or equal to the latest version of α, i.e., z.

Figure 5(b) gives an illustrative example. In the FDG of project $\mathtt{Q}^{5.0}$, there are two incoming edges to project $\mathtt{T}^{4.0}$, one from project $\mathtt{X}^{2.0}$ and the other from project $\mathtt{P}^{4.0}$. Suppose that the former edge is traversed before the latter. Since the constraint that $\mathtt{X}^{2.0}$ sets on T has no upper bound, the latest version 4.0 of T will be installed. There is no dependency conflict at the time of analysis. However, since the constraint associated with latter edge, i.e., $\langle > 2.0 \wedge \leq 4.0 \rangle$, restricts T to a version range, build failures may occur if developers release a newer version (e.g., 4.1) of T on PyPI.

5 EVALUATION

To evaluate Watchman, we study two research questions:

- **RQ3 (Effectiveness):** *How effective is Watchman in detecting real DC issues and predicting potential ones?*
- **RQ4 (Usefulness):** *Can Watchman monitor DC issues in PyPI and provide useful diagnostic information to developers?*

To answer RQ3, we replayed the evolution history of all libraries on PyPI from 1 Jan 2017 to 30 Jun 2019. We first constructed a metadata repository for PyPI's snapshot on 1 Jan 2017, and then conducted incremental analysis to extract daily updates of all libraries until 30 Jun 2019. For each library update, we applied Watchman to detect DC issues and predict potential ones via identifying dependency smells. Since we have the whole evolution history, we could evaluate Watchman's effectiveness by checking whether the detected DC issues have been resolved and whether the predicted ones have indeed evolved into real issues subsequently.

To answer RQ4, we deployed Watchman to monitor PyPI since 1 Jul 2019, and configured it to detect new DC issues of Pattern A, as well as potential ones that could be induced by the smells of Type 1 and Type 2. Note that issues of Pattern B can hardly be detected since they are affected by developers' local environments, on which we have no knowledge.

We then consolidated the detected DC issues and submitted reports to the concerned projects' issue tracking systems; if: (1) the detected issues have not been reported or fixed in the unreleased master branches of the projects and (2) the concerned projects have maintenance records in the last two years (still active). In each issue report, we pointed out the detected conflicts and explained how they arose. Such diagnostic information can be easily provided by Watchman since it simulates the build process of each project. The report also includes fixing suggestions generated by Watchman based on our observed common fixing strategies.

Table 2: Basic information of experimental subjects

	Period 1	Period 2	Period 3	Period 4	Period 5
# Projects	1,454	1,535	2,279	2,398	2,673
# Releases	11,759	13,202	18,418	18,984	19,746
# Commits	530	646	338	740	694

5.1 RQ3: Effectiveness

Data collection. A project's evolution history provides useful information about how DC issues manifested themselves (and got fixed). To ease experiments, we divided the whole time period from 1 Jan 2017 to 30 Jun 2019 into the following five sub-periods: 1 Jan 2017–30 Jun 2017 (*Period 1*), 1 Jul 2017–31 Dec 2017 (*Period 2*), 1 Jan 2018–30 Jun 2018 (*Period 3*), 1 Jul 2018–31 Dec 2018 (*Period 4*), and 1 Jan 2019–30 Jun 2019 (*Period 5*). For each sub-period, we collected open-source Python projects satisfying the following two criteria as our experimental subjects: (1) having more than five release versions during this sub-period (active), and (2) having more than 300 commits during this sub-period (well-maintained). Table 2 lists the basic information of these subjects. On average, there are 16,421 releases of 2,067 projects for each sub-period. We then applied Watchman to detect DC issues of Pattern A and predict potential ones that may be induced by smells of Type 1 (Type 1 issues) and Type 2 (Type 2 issues), during each of the five sub-periods on a daily basis.

Evaluation metrics. To evaluate Watchman's effectiveness, we define two metrics, ***resolving ratio*** and ***lasting time***:

- For each detected issue of Pattern A, we checked whether it had been resolved in the latest version of the project released on PyPI, up to the date 20 Jul 2019. The metric *resolving ratio* measures the proportion of resolved DC issues in those detected by Watchman. Higher resolving ratios indicate better effectiveness of Watchman. The metric *lasting time* measures the gap between the detection time of a DC issue and the resolving time of this issue. A longer lasting time indicates a wider impact caused by a DC issue on the concerned downstream projects.
- For the predicted issues, we checked whether they had turned into real ones. There are two cases: (1) the predicted issue indeed arose (reported) in history due to library updates, and was fixed by developers in subsequent project releases; (2) the predicted issue was not reported in history but developers still fixed it to avoid certain undesirable consequences. Accordingly, the metric *resolving ratio* measures the proportion of the predicted DC issues that belong to either case. The metric *lasting time* measures the gap between the time an issue was predicted and the time it was reported for *Case (1)*, and the gap between the time an issue was predicted and the time it was resolved by developers for *Case (2)*.

Results. Table 3 presents the experimental results. For all the five sub-periods, Watchman detected a total of 369 DC issues of Pattern A, and all of them had been fixed by developers (i.e., *resolving ratio* = 100%). This strongly suggests that Watchman can precisely

Table 3: Results of DC issues reported by WATCHMAN from 1 Jan 2017 to 30 Jun 2019

	Period 1	Period 2	Period 3	Period 4	Period 5	Summary
Pattern A	56	42	84	72	115	369★
Fixed	56	42	84	72	115	369★
Resolving ratio	100%	100%	100%	100%	100%	100%♮
Lasting time (days)	25.2	27.3	25.0	20.8	31.6	26.0♮
Type 1	10	13	12	11	15	61★
Type 2	16	18	19	21	21	95★
Case (1)	2	2	3	4	2	13★
Case (2)	22	25	26	26	31	130★
Resolving ratio	92.3%	87.1%	93.5%	93.8%	91.7%	91.7%♮
Lasting time (days)	101.6	77.1	100.8	51.0	63.9	78.9♮

♮ denotes the average value while ★ denotes the sum.

detect DC issues. WATCHMAN also predicted a total of 156 Type 1 and Type 2 issues, 143 of which had been resolved by developers, resulting in an average *resolving ratio* of 91.7% (= (13+130)/(61+95)). The *resolving ratio* of the predicted issues for different periods ranges from 87.1% to 93.8%, which are generally satisfactory. This suggests that WATCHMAN is also effective in predicting potential DC issues. Besides, we observed that all detected 369 DC issues were resolved by developers within a month (on average, 26 days). WATCHMAN may help reduce this delay since it can detect DC issues timely (it performs analysis on a daily basis) and report them to developers along with fixing suggestions. If developers are able to fix WATCHMAN's detected issues in due course, the side effect of these issues on downstream projects will be largely diminished.

As at 20 Jul 2019, 13 (8.7%) of the 156 DC issues predicted by WATCHMAN had not evolved into real ones. By further analyzing the concerned projects, we found that the dependencies introducing these issues were no longer active. For instance, in the project `finance-dl` [5], WATCHMAN found multiple version constraints for library `idna`. When building `finance-dl`, `pip` would install version 2.8 of `idna`, which is equal to the upper bound of the constraint $\langle \geq 2.5 \wedge \leq 2.8 \rangle$ introduced by the latest version of the library `selenium-requests`. However, this potential DC issue (Type 2) did not evolve into a real one, since `selenium-requests` had stopped its update on `PyPI` (at our study time).

5.2 RQ4: Usefulness

WATCHMAN detected and predicted a total of 189 DC issues since we started our online monitoring on 1 Jul 2019. We filtered out 23 issues that had been reported in the corresponding projects' issue tracking systems and 49 issues whose associated projects had no maintenance record in the last two years. After filtering, we reported the remaining 117 DC issues to developers. As shown in Table 4, 63 issues (53.8%) were confirmed by developers as real DC issues within a few days. 38 out of the 63 confirmed issues (60.3%) were quickly fixed, and 25 confirmed issues (38.7%) are under fixing. The remaining 54 issues are still pending, mainly due to inactive maintenance of the associated projects. We provide a detailed analysis in the following.

5.2.1 Feedback on reported issues. For 64 detected issues of Pattern A, which were caused by library updates, WATCHMAN got a higher confirmation rate (60.9% = 39/64), which is within our expectation. For these 39 confirmed DC issues, developers agreed that the detected conflicts would cause build failures, and invited us to submit patches to help resolve them. For instance, in issue report #70 [32] of the project `Osmedeus`, the developer mentioned that they had indeed encountered our reported DC issue when deploying the project, and left a comment "*I also get that error when installing the project but my server works fine. Just submit a PR and I will review the patch*".

For the 21 predicted DC issues of Type 1, 11 of them have already been spotted by developers and resolved in the master branches of the corresponding projects (but not yet released on `PyPI`) before we reported them. For instance, the project `MycroftAI adapt` relaxed its version constraint on library `six` from $\langle = 1.10.0 \rangle$ to $\langle \geq 1.10.0 \rangle$ in commit `7eeadeb` [1] with a log "*to avoid incompatibility with downstream projects `adapt-parser` and `jsonschema`*". Therefore, we reported only the remaining 10 issues of Type 1 to the developers of the corresponding projects, and 4 of them have been confirmed by developers. Encouragingly, in the issue report #182 [12] of the project `dynamic-prefer -ences`, we got the following comment from developers after they resolved our reported issue: "*It is a hazzle to keep track of all the frozen versions of some dependencies, especially for larger projects. I think it would be good to get an automatic notification as maintainer somehow, if one of your dependencies has locked its own libraries on a specific version.*"

For the 43 predicted Type 2 issues, 20 of them have been confirmed by developers, although these issues may not cause build failures immediately. We observed that some of WATCHMAN's warnings quickly caught developers' attention, and they added labels "bug" and "deployment problem" to our issue reports (e.g., [19, 20]).

Among the 63 confirmed DC issues (including both detected and predicted ones), developers resolved 54 (85.7%) of them following our suggested solutions. For example, we reported an issue of Pattern A.a to the developers of `webinfo` along with three fixing solutions (issue #9 [34]). The developer chose the one WATCHMAN generated based on Strategy 2 to resolve this conflict. For the remaining nine confirmed issues, for which our fixing solutions have not been adopted by developers, we found that these projects can be sensitive to certain library updates and our suggested changes might introduce other side effects, such as vulnerability or compatibility issues (e.g., issue #16 [10] in project `kindred`).

5.2.2 Feedback on WATCHMAN. Besides confirming our reported DC issues, some developers expressed interests in our tool WATCHMAN. For example, a developer left the following comment in the pull request #71 [40] of the project `arxiv-submission-core`:

"*A better mechanism of maintaining the dependency constraints among projects on PyPI like what you did, is much-needed!*"

In issue report #492 [30] of the project `pywb`, we found an encouraging comment from an experienced lead developer who is also the founder of the `webrecorder` community [4]:

"*Are you an 'automation' written by Github community to help resolve dependency conflict issues for Python projects? If so, a piece of nice work! I'd say this is a good approach, a nice friendly bot to inform of potential dependency issues.*"

Such feedback indicates that monitoring library updates and detecting/predicting dependency conflicts is indeed important to and welcomed by real-world Python developers. The information provided by WATCHMAN is also useful to help developers diagnose DC issues in practice.

Table 4: Results of 117 DC issues reported by WATCHMAN from 11 Jul 2019 to 16 Aug 2019

Manifestation	Issue reports (Each item gives the issue report ID and the project name)
Pattern A.a	Issue#1, aucome; Issue#110, crypto; Issue#1, OrcaSong; Issue#2, pypmml-spark; Issue#138, toolium; Issue#26, GatewayFramework; Issue#56, Airbnb-data; Issue#2, Runcible; Issue#95, identification; Issue#96, identification; Issue#1813, tasking-manager; Issue#356, Archery; Issue#325, bocadillo; Issue#21, crema; Issue#4, what-digit-you-write; Issue#9, webinfo-crawler; Issue#35, zarp; Issue#4, open-helpdesk; Issue#5, languagecrunch; Issue#103, account-creator; Issue#9, jawfish; Issue#212, openpose-plus; Issue#16, kindred; Issue#13, Generator-GUI; Issue#3, tabular; Issue#5, whats-bot; Issue#65, armory-bot; Issue#39, derrick; Issue#16, Historical-Prices; Issue#688, dxr; Issue#18526, erpnext; Issue#1, scrapy-qtwebkit; Issue#4778, InstaPy; Issue#2, api-indotel; Issue#145, cert-issuer; Issue#146, django; Issue#4, pymacaron; Issue#1, mgz-db; Issue#1, twitterbots; Issue#2, gremlin; Issue#17, AWSBucketDump; Issue#198, fabric-cli; Issue#1, BlockCluster; Issue#3, gateway; Issue#2, beauty_image; Issue#1389, Indy-node; Issue#130, swapi; Issue#279, explorer; Issue#34 footmark; Issue#3, driver-acs; Issue#56, driver-napi; Issue#11, simulator; Issue#9, Friends-Finder; Issue#1, chatbot-template; Issue#545, djangopackages; Issue#2048, cadasta-platform; Issue#122, adminset; Issue#45, Wallpaper; Issue#21, ltiauthenticator; Issue#28, cryptography;
Pattern A.b	Issue#243, bakerydemo; Issue#4, pytools; Issue#70, Osmedeus; Issue#101, aldryn-search;
Type 1	Issue#182, dynamic-preferences; Issue#20, ldapdomaindump; Issue#326, py-cluster; Issue#986, faker; Issue#717, newspaper; Issue#120, mixer; Issue#3, client-python; Issue#75, PyInquirer; Issue#953, compressor; Issue#26, certstream;
Type 2	Issue#8, AutoCrawler; Issue#31, BBScan; Issue#492, pywb; Issue#8, ct-exposer; Issue#71, EagleEye; Issue#1179, mythril; Issue#1, frida-util; Issue#34, python-urwid; Issue#4, SecurityManageFramwork; Issue#295, sherlock; Issue#2077, freqtrade; Issue#36, trains; Issue#298, glastopf; Issue#5, Machine-Learning-with-Python; Issue#569, kalliope; Issue#98, bless; Issue#70, arxiv-submission-core; Issue#2729, plaso; Issue#17, oauth-dropins; Issue#303, ripping-machine; Issue#27, ChannelBreakoutBot; Issue#167, tldextract; Issue#183, messytables; Issue#9, kuberdock-platform; Issue#42, python-weixin; Issue#25, NoDB; Issue#146, Photon; Issue#911, pyspider; Issue#7, fan; Issue#126, historical; Issue#49, stephanie-va; Issue#979, subliminal; Issue#56, WPSeku; Issue#3, zhihu-crawler; Issue#38, network-topology; Issue#647, marathon-lb; Issue#9, Konan; Issue#181, JBOPS; Issue#962, hangoutsbot; Issue#41, GyoiThon; Issue#120, automation-tools; Issue#4, start-vm; Issue#10, ahmia-index;

Status 1: The issues have already been fixed using our suggested solutions; Status 2: The issues have already been fixed using other solutions; Status 3: The issues have been confirmed and are under fixing using our suggested solutions. Status 4: The issues have been confirmed and are under fixing using other solutions. Status 5: The issues are still pending.
We do not present the link of these issues due to page limit. The detailed information of them can be found on our project website (**http://www.watchman-pypi.com/buglist**).

6 DISCUSSIONS

Threats to validity. Keyword search can introduce irrelevant issues into our dataset. Such noises pose a threat to the validity of our study results. The errors in our manual analysis of the DC issues may also affect our study results. To reduce these threats, three co-authors independently investigated our collected DC issues and cross-validated their analysis results.

Limitations. Our work has three limitations. First, we focus on the DC issues that cause build failures. However, in some cases, dependency conflicts may lead to semantic inconsistencies, runtime errors, or other consequences in Python projects. Second, the rules adopted in the predictive analysis can only help find a subset of all possible DC issues that may be induced by the two types of dependency smells. The rule set is designed based on our observed real cases in the empirical study and is not meant to be complete. Third, WATCHMAN currently is not able to detect all patterns of DC issues observed in our empirical study. We will address these limitations in future work.

7 RELATED WORK

Studies of dependency conflicts. Pradel et al. [52] studied the dependency conflicts among JavaScript libraries and proposed a detection strategy. Suzaki et al. [45] conducted an extensive case study of conflict defects, including conflicts on resource access, conflicts on configuration data, and interactions between uncommon combinations of packages. Soto-Valero et al. [54] studied the problem of multiple versions of the same library co-existing in `Maven Central`, and presented empirical evidence about how the immutability of artifacts in `Maven Central` supports the emergence of natural software diversity. Wang et al. [56] conducted an empirical study to characterize dependency conflicts in Java projects and developed RIDDLE to generate tests to collect crashing stack traces to facilitate DC issue diagnosis [57]. To the best of our knowledge, there is no previous work focusing on characterizing and detecting DC issues in the Python world.

Studies of software ecosystem. Serebrenik et al. [53] performed a meta-analysis of the difficult tasks in software ecosystem research and identified six types of challenges, e.g., how to scale the analysis to a massive amount of data. Mens [50] studied software ecosystem from the socio-technical view on software maintenance and evolution. Zimmermann et al. [59] studied the security risks in the `npm` ecosystem by analyzing data such as dependencies between packages and publicly reported security issues. Another study by Lertwittayatrai et al. [48] used network analysis techniques to study the topology of the JavaScript package ecosystem and extracted insights about dependencies and their relations. Our work studies software ecosystem from a novel perspective by taking into account the interference between the version constraints of upstream and downstream projects. We also propose a technique to continuously monitor dependency conflicts for Python projects.

8 CONCLUSION AND FUTURE WORK

In this work, we first conducted an empirical study of 235 real dependency conflict issues in Python projects to understand the manifestation patterns and fixing strategies of dependency conflict issues. Motivated by our empirical findings, we then designed a technique,WATCHMAN, to continuously monitor dependency conflicts for the `PyPI` ecosystem. Evaluation results show that WATCHMAN can effectively detect dependency conflict issues with a high precision and provide useful diagnostic information to help developers fix the issues. In future, we plan to further improve the detection capability of WATCHMAN and generalize our technique to other Python library ecosystems such as `Anaconda` to make it accessible to more developer communities.

ACKNOWLEDGMENTS

The authors would like to sincerely thank the anonymous reviewers of ICSE 2020 for their constructive comments that helped improve this paper. Part of the work was conducted during the first author's internship at HKUST in 2018. This work is supported by the National Natural Science Foundation of China (grants #61932021, #61902056, #61802164, #61977014), the Hong Kong RGC/GRF grant #16211919, MSRA grant, and the Program for University Key Laboratory of Guangdong Province (Grant #2017KSYS008).

REFERENCES

[1] 2020. Commit 7eeadeb of adapt. https://github.com/MycroftAI/adapt/commit/7eeadeb4744b7e2dd7a9aa61e0350c4e22350eba. (2020). Accessed: 2020-02-06.
[2] 2020. conda. https://conda.io/. (2020). Accessed: 2020-02-06.
[3] 2020. Dependency specification for Python. https://www.python.org/dev/peps/pep-0508/. (2020). Accessed: 2020-02-06.
[4] 2020. An experienced developer. https://github.com/ikreymer. (2020). Accessed: 2020-02-06.
[5] 2020. finance-dl. https://github.com/jbms/finance-dl. (2020). Accessed: 2020-02-06.
[6] 2020. Issue #1277 of channels. https://github.com/django/channels/issues/1277. (2020). Accessed: 2020-02-06.
[7] 2020. Issue #152 of channels. https://github.com/django/channels_redis/issues/152. (2020). Accessed: 2020-02-06.
[8] 2020. Issue #1525 of molecule. https://github.com/ansible-community/molecule/issues/1525. (2020). Accessed: 2020-02-06.
[9] 2020. Issue #1586 of crossbar. https://github.com/crossbario/crossbar/issues/1586. (2020). Accessed: 2020-02-06.
[10] 2020. Issue #16 of kindred. https://github.com/jakelever/kindred/issues/16. (2020). Accessed: 2020-02-06.
[11] 2020. Issue #1607 of molecule. https://github.com/ansible-community/molecule/issues/1607. (2020). Accessed: 2020-02-06.
[12] 2020. Issue #182 of dynamic-preferences. https://github.com/EliotBerriot/django-dynamic-preferences/issues/182. (2020). Accessed: 2020-02-06.
[13] 2020. Issue #1824 of allennlp. https://github.com/allenai/allennlp/issues/1824. (2020). Accessed: 2020-02-06.
[14] 2020. Issue #2195 of allennlp. https://github.com/allenai/allennlp/issues/2195. (2020). Accessed: 2020-02-06.
[15] 2020. Issue #229 of gallery-dl. https://github.com/mikf/gallery-dl/issues/229. (2020). Accessed: 2020-02-06.
[16] 2020. Issue #2483 of allennlp. https://github.com/allenai/allennlp/issues/2483. (2020). Accessed: 2020-02-06.
[17] 2020. Issue #25316 of pandas. https://github.com/pandas-dev/pandas/issues/25316. (2020). Accessed: 2020-02-06.
[18] 2020. Issue #25487 of pandas. https://github.com/pandas-dev/pandas/issues/25487. (2020). Accessed: 2020-02-06.
[19] 2020. Issue #2729 of plaso. https://github.com/log2timeline/plaso/issues/2729. (2020). Accessed: 2020-02-06.
[20] 2020. Issue #295 of sherlock. https://github.com/sherlock-project/sherlock/issues/295. (2020). Accessed: 2020-02-06.
[21] 2020. Issue #3118 of pipenv. https://github.com/pypa/pipenv/issues/3118. (2020). Accessed: 2020-02-06.
[22] 2020. Issue #32 of valinor. https://github.com/ARMmbed/valinor/issues/32. (2020). Accessed: 2020-02-06.
[23] 2020. Issue #36 of dbxfs. https://github.com/rianhunter/dbxfs/issues/36. (2020). Accessed: 2020-02-06.
[24] 2020. Issue #3826 of rasa. https://github.com/RasaHQ/rasa/issues/3826. (2020). Accessed: 2020-02-06.
[25] 2020. Issue #407 of wandb/client. https://github.com/wandb/client/issues/407. (2020). Accessed: 2020-02-06.
[26] 2020. Issue #409 of bandit. https://github.com/PyCQA/bandit/issues/409. (2020). Accessed: 2020-02-06.
[27] 2020. Issue #4669 of requests. https://github.com/psf/requests/pull/4669. (2020). Accessed: 2020-02-06.
[28] 2020. Issue #4674 of requests. https://github.com/psf/requests/pull/4674. (2020). Accessed: 2020-02-06.
[29] 2020. Issue #4675 of requests. https://github.com/psf/requests/pull/4675. (2020). Accessed: 2020-02-06.
[30] 2020. Issue #492 of pywb. https://github.com/webrecorder/pywb/issues/492. (2020). Accessed: 2020-02-06.
[31] 2020. Issue #66 of pythoncoveralls. https://github.com/z4r/pythoncoveralls/issues/66. (2020). Accessed: 2020-02-06.
[32] 2020. Issue #70 of Osmedeus. https://github.com/j3ssie/Osmedeus/issues/70. (2020). Accessed: 2020-02-06.
[33] 2020. Issue #740 of yotta. https://github.com/ARMmbed/yotta/issues/740. (2020). Accessed: 2020-02-06.
[34] 2020. Issue #9 of webinfocrawler. https://github.com/lubosson/webinfocrawler/issues/9. (2020). Accessed: 2020-02-06.
[35] 2020. Issue #9090 of spyder. https://github.com/spyder-ide/spyder/issues/9090. (2020). Accessed: 2020-02-06.
[36] 2020. pip. https://pypi.org/project/pip/. (2020). Accessed: 2020-02-06.
[37] 2020. pip documentation. https://pip.pypa.io/en/stable/reference/pip_install. (2020). Accessed: 2020-02-06.
[38] 2020. pipenv. https://docs.pipenv.org/. (2020). Accessed: 2020-02-06.
[39] 2020. PR #1675 of molecule. https://github.com/ansible-community/molecule/pull/1675. (2020). Accessed: 2020-02-06.
[40] 2020. PR #71 of arxiv-submission-core. https://github.com/arXiv/arxiv-submission-core/pull/71. (2020). Accessed: 2020-02-06.
[41] 2020. Project molecule. https://github.com/ansible-community/molecule. (2020). Accessed: 2020-02-06.
[42] 2020. PyPI. https://pypi.org/. (2020). Accessed: 2020-02-06.
[43] 2020. virtualenv. https://virtualenv.pypa.io/en/latest/. (2020). Accessed: 2020-02-06.
[44] Pietro Abate and Roberto Di Cosmo. 2011. Predicting upgrade failures using dependency analysis. In *Proceedings of the IEEE 27th International Conference on Data Engineering Workshops*. 145–150.
[45] Cyrille Artho, Kuniyasu Suzaki, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Why do software packages conflict?. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'18)*. 141–150.
[46] Alexandre Decan, Tom Mens, and Maelick Claes. 2016. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *Proccedings of the 10th European Conference on Software Architecture Workshops*. 21.
[47] Jiajun Hu, Lili Wei, Yepang Liu, Shing-Chi Cheung, and Huaxun Huang. 2018. A Tale of Two Cities: How WebView Induces Bugs to Android Applications. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. 702–713.
[48] 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 *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC'17)*. 298–307.
[49] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering, (ICSE'14)*. 1013–1024.
[50] 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'16)*. 1–8.
[51] Fabio Nelli. 2015. Python data analytics. *Berkeley: Apress* (2015).
[52] 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'18)*. 741–751.
[53] Alexander Serebrenik and Tom Mens. 2015. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. 40.
[54] 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 IEEE Working Conference on Mining Software Repositories (MSR'19)*. 1–11.
[55] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 644–655.
[56] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. 319–330.
[57] 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 41th International Conference on Software Engineering (ICSE'19)*. 572–583.
[58] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2016. Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, (ASE'16)*. 226–237.
[59] Markus Zimmermann, Cristianalexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *arXiv: Cryptography and Security* (2019).