PyTrim: A Practical Tool for Reducing Python Dependency Bloat

Konstantinos Karakatsanis[1], Georgios Alexopoulos[2], Ioannis Karyotakis[1], Foivos Timotheos Proestakis[1]
Evangelos Talos[1], Panos Louridas[1], Dimitris Mitropoulos[2]
[1]Athens University of Economics and Business, [2]University of Athens
{karakatsanis, karyotakisg, proestakis, vtalos, louridas}@aueb.gr, {grgalex, dimitro}@ba.uoa.gr

***Abstract*—Dependency bloat is a persistent challenge in Python projects, which increases maintenance costs and security risks. While numerous tools exist for detecting unused dependencies in Python, removing these dependencies across the source code and configuration files of a project requires manual effort and expertise. To tackle this challenge we introduce PYTRIM, an end-to-end system to automate this process. PYTRIM eliminates unused imports and package declarations across a variety of file types, including Python source and configuration files such as `requirements.txt`, and `setup.py`. PYTRIM's modular design makes it agnostic to the source of dependency bloat information, enabling integration with any detection tool. Beyond its contribution when it comes to automation, PYTRIM also incorporates a novel dynamic analysis component that improves dependency detection recall. Our evaluation of PYTRIM's end-to-end effectiveness on a ground-truth dataset of 37 merged pull requests from prior work, shows that PYTRIM achieves 98.3% accuracy in replicating human-made changes. To show its practical impact, we run PYTRIM on 971 open-source packages, identifying and trimming bloated dependencies in 39 of them. For each case, we submit a corresponding pull request, 6 of which have already been accepted and merged. PYTRIM is available as an open-source project, encouraging community contributions and further development.**

Video demonstration: https://youtu.be/LqTEdOUbJRI
Code repository: https://github.com/TrimTeam/PyTrim

I. Introduction

Python has emerged as one of the most widely adopted programming languages globally [1], [2] partly due to its large ecosystem of third-party packages, with the Python Package Index (PyPI) – currently hosting over 654,000 projects [3]. While these packages accelerate development and foster reuse they also contribute to a notable software engineering challenge: *dependency bloat*. As projects evolve, obsolete dependencies are often left in the codebase. This creates technical debt, which in turn leads to increased maintenance overhead, expanded attack surface, and slower pipelines [4]–[7].

Consider a real-world example from the `softlayer-python` project [8]. In a commit from May 2022, developers refactored the command-line interface to use the `rich` library, replacing the older `prettytable` library. While all source code usages of `prettytable` were removed, the dependency declaration itself was left behind in four configuration files, two of them shown in Fig. 1: (1a) `setup.py`, which defines package metadata and installation configuration for a Python project, and (1b) `tools/requirements.txt`, which lists the Python dependencies needed for the project's tooling components.

```
install_requires = [
  'prettytable',
  ...
  'rich==14.0.0',
]
```

(a) `setup.py`

```
prettytable
...
rich==14.0.0
```

(b) `tools/requirements.txt`

Fig. 1: Two of the four configuration files of the `softlayer-python` package. Its `prettytable` dependency has been unused since commit `1238377` [8].

The Python ecosystem provides numerous open-source tools that focus on bloat detection. Some of these tools operate at the file level, identifying unused imports within individual Python files [9], [10]. Others operate at the project level, looking for unused package dependencies by comparing configuration files against the imports used in the source code [11], [12]. While valuable for detection, these tools do not provide a removal solution.

Contributions: We present PYTRIM, which introduces:

- An end-to-end pipeline for the detection and removal of unused dependencies from Python projects, including integration with three existing detection tools.
- A novel and practical dynamic analysis technique for calculating a project's dependencies, which it uses to enhance the static-based approaches of integrated bloat detection tools.
- Removal logic for dependency declarations from a wide array of file formats, including Python source files, `requirements` files, TOML (Tom's Obvious, Minimal Language) [13] configurations, and `setup.py` files.
- Automation of the maintenance lifecycle by creating a new Git branch, committing the changes, and generating a pull request with a detailed report, enabling a "review-and-merge" approach to dependency cleanup.

II. Background & Motivation

For the remainder of this paper, we use the term *configuration files* to refer to auxiliary files that declare a package's dependencies or define its installation behavior. Such files include `requirements.txt`, `pyproject.toml`, and `setup.py`. Accordingly, we use the term *source files* to refer to source code files that implement the main functionality of a package. Finally, we use the term *project* to refer to the source code (usually published on GitHub) of a corresponding Python *package*, which is in turn published on PyPI.

```
f = open('reqs/core.txt')
REQS = f.read().splitlines()
...
install_requires = REQS
```

(a) `setup.py`

```
pyrsistent>=0.16.0
...
```

(b) `reqs/core.txt`

Fig. 2: `optimizely-sdk` builds its dependencies dynamically, causing static analysis-based detectors to miss its `pyrsistent` dependency and fail to identify it as unused.

We define a dependency package as *unused* (or *bloat*) if it is declared in the project's configuration files but is not used anywhere in the project beyond, at most, a direct import. While we acknowledge that importing a package causes Python to implicitly execute its `__init__.py` file [14], which could be part of the program logic, we encounter no such cases during our evaluation.

To detect unused dependencies such as the `prettytable` dependency in `softlayer-python` from Fig. 1, developers must *run* a dependency bloat detection tool, such as `fawltydeps` [11], on the package's source code and manually *examine* the tool's output to identify bloated dependencies. Note that the process of detecting unused dependencies comprises two distinct phases: (i) computing the set of the package's dependencies, and (ii) determining the extent to which each of them is used.

Once unused dependencies are identified, developers must undertake a manual removal process that involves *removing* dependency declarations from configuration files and imports from Python files, and finally *submitting* a pull request to fix the problem. This remediation workflow requires careful attention to ensure that all references to the unused dependency are properly eliminated from the codebase without affecting the package's functionality.

Challenge 1: Project dependency resolution: To compute the dependency set, existing detection tools rely on static analyses of configuration files. While tools such as `fawltydeps`, correctly parse standard configuration files such as the ones depicted in Fig. 1, they fail when dependencies are programmatically constructed. For example, in the `optimizely-sdk` package (Fig. 2), the `setup.py` file dynamically builds the `install_requires` list by reading an external file, making static analysis approximation difficult and causing missed dependencies. Given that over 50% of PyPI packages published in April 2024 [15] use `setup.py` as their configuration file, the practical approximation of its behavior has become essential.

Challenge 2: Multitude of configuration files and types: To remove dependency declarations developers have to search for and edit all related configuration files (e.g. all four aforementioned configuration files in the case of `softlayer-python`). Notable, open-source projects have previously encountered issues [16] with partial updates of configuration files, leading to broken builds.

Challenge 3: Human error: While valuable for detection, existing tools do not provide a removal solution. That is, developers have to perform the remediation workflow manually, making the procedure prone to human error. This gap between detection and remediation represents a bottleneck in automated software maintenance.

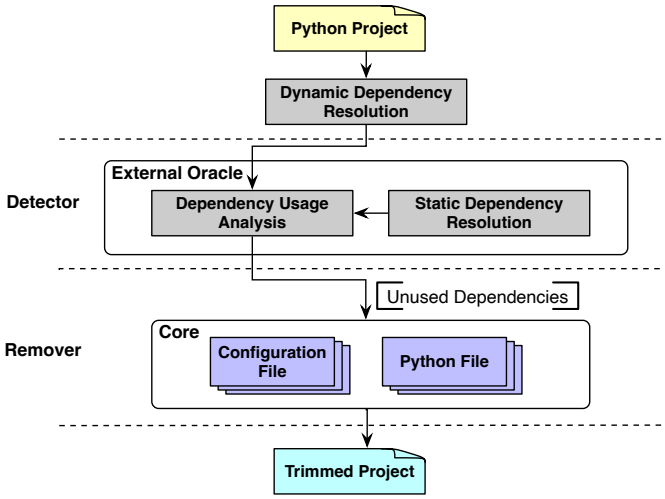

Fig. 3: High-level architecture of PYTRIM.

III. PYTRIM

A. Overview

PYTRIM receives as input the path to a Python project and then proceeds to (a) dynamically infer the project's dependencies by installing it in isolation (b) identify the subset of unused dependencies by invoking an unused dependency detector, and (c) locate and remove their occurrences (declarations and imports) across a wide range of file formats. Its design is detector-agnostic, allowing any dependency analysis tool to be integrated.

B. System Architecture

PYTRIM incorporates three primary modules:

- **Dynamic dependency resolver:** A component that generates a list of dependencies based on install-time information.
- **Detector:** An external dependency analysis tool (e.g., extended PyCG [17]) to determine which dependencies are unused.
- **Remover:** A component that implements file-specific logic to locate and remove unused dependency occurences (declarations and imports).

The interaction between these modules and the overall workflow is shown in Fig. 3.

Dynamic dependency resolver: To address Challenge 1, we perform a source installation of the project under analysis by invoking `pip` with the "`-t`" argument, to install artifacts in an isolated environment. This is a dynamic operation, meaning that it yields no false positives, while at the same time requires no inputs. This step is inspired by recent work that has been introduced to support cross-language call-graph construction [18]. Then, we utilize the `pipdeptree` [19] utility to produce a directed graph representation of installed packages in this environment, where an edge from `x` to `y` denotes that `x`

depends on `y`. The package under analysis is the vertex with no incoming edges, and its dependencies are the destinations of its outgoing edges. In this manner, we dynamically obtain a list of the project's dependencies. A current limitation of our approach is that it only considers the default installation options, and thus only follows one concrete execution path in the configuration file.

Detector: As a default, PYTRIM utilizes a call-graph based detector [17]. It also provides out-of-the-box support for two other popular detectors, namely `fawltydeps` [11] and `deptry` [12]. We modify each detector's dependency resolution component to take into account the union of its built-in static dependency resolver and PYTRIM's dynamic dependency resolver, to decrease missed dependencies.

Remover: To address Challenge 2 we follow a two-fold approach. For Python source files (`.py`), PYTRIM employs Abstract Syntax Tree (AST) analysis to identify and remove unused imports. For configuration files, it uses a combination of methods based on file type: structured file types such as `TOML`, `YAML`, and `INI/CFG` are managed by parsing libraries such as `toml`, `PyYAML`, and `configparser`; line-based formats including `requirements.txt` and their common `.in` counterparts are handled with regular expressions; and executable `setup.py` files are processed via AST analysis.

To ensure dependency integrity after these modifications, PYTRIM identifies when a lock file (e.g., `poetry.lock`) may be out of sync and prints a message to inform the user to regenerate it manually, as automatic regeneration could be unsafe due to project-specific versioning or hashes. Similarly, less structured files like shell scripts or Dockerfiles, are only analyzed for reporting and are not modified automatically due to their inherent complexity and risk of reliably altering their syntax programmatically.

C. Implementation and Usage

PYTRIM is implemented in ∼4,200 lines of Python code and is provided as a command-line tool, installable from PyPI via `pip install pytrim` and accessible via the `pytrim` command. Apart from end-to-end automated analysis (Challenge 3), it can also work in removal-only mode, bypassing its built-in detectors, accepting the list of unused packages as input alongside the project's path. Further, PYTRIM can also generate markdown reports and pull requests.

IV. Evaluation

A. Dynamic Dependency Resolution

We compare PYTRIM's dependency resolver with the static resolvers employed by the three integrated detection tools, to quantify the missed dependencies it helps uncover. To this end, we evaluate each resolver on a dataset of 1300 popular GitHub projects studied in previous work [17]. From these, we only keep a subset of 971 packages that are successfully installed. Compared to the static resolvers, the dynamic resolver is able to uncover missed dependencies in 48/971 (5%) of cases. The main causes for the static resolvers' misses are: (a) parsing limitations for edge-cases of supported declarative files, such as `pyproject.toml` (18 cases), (b) arbitrary `setup.py` code, such as function invocations (22 cases), and (c) miscellaneous bugs (8 cases). For example, the dependency of `optimizely-sdk` on `pyrsistent`, shown in Fig. 2, is detected only through our dynamic resolution technique.

TABLE I: PYTRIM's replication accuracy on manually created PRs from previous work [17].

Metric	Value
Total Pull Requests Analyzed	37
Total Files with Dependency Changes	76
Files Excluded (e.g., Documentation)	16
Relevant Files for Comparison	**60**
Files Correctly Replicated by PYTRIM	59
Files with Mismatched Output	1
Replication Accuracy	**98.33%**

B. Remover effectiveness

To validate PYTRIM's effectiveness, we design an automated pipeline to measure its accuracy against real-world software maintenance tasks. As a ground truth, we use the dataset from Drosos et al. [17], which provides a curated set of pull requests (PRs) where bloated dependencies were manually removed from open-source Python projects.

The ground truth for each PR consists of two parts: the list of dependencies the developer removed, and the final, correct state, of the modified files. Our pipeline first runs PYTRIM on the pre-PR version of each project, providing the ground-truth list of dependencies as input. It then compares the files modified by PYTRIM against the ground-truth files to measure replication accuracy, ignoring non-semantic differences such as changes in white space or comments.

The results of our analysis are summarized in Table I. Out of 60 relevant files where dependencies were removed, PYTRIM successfully matches the final state of 59, achieving a replication accuracy of 98.33%. It is also efficient; when provided with the pre-computed list of unused dependencies, it processes the entire evaluation dataset of 37 projects in less than 10 seconds.

The only mismatch occurred in a PR [20] on `simple-salesforce`, revealing a limitation of PYTRIM's scope: dependency refactoring. While PYTRIM correctly removed `cryptography`, the human developer also refactored `pyjwt` into `pyjwt[crypto]`, to include optional cryptographic features. Such semantic changes require domain-specific knowledge and are currently out of scope for PYTRIM, which focuses solely on removing unused dependencies, not refactoring them.

C. Real-World Deployment

To evaluate PYTRIM's practical applicability, we deploy it on the 971 installable open-source Python projects studied in previous work [17]. For each project, we run PYTRIM using the call-graph-based detector to identify unused dependencies. Then, we manually verify all changes and submit pull requests to the respective repositories.

TABLE II: Merged pull requests contributed by PYTRIM.

Project	PR ID	Files Modified
pytroll/pyresample	#669	1
furlongm/patchman	#689	1
pycqa/prospector	#781	2
jamesoff/simplemonitor	#1646	4
softlayer/softlayer-python	#2230	5
napalm-automation/napalm	#2243	1

PYTRIM removes unused dependencies in 39 projects, resulting in 39 submitted pull requests. At the time of writing, 6 of these PRs have been merged by the project maintainers, 29 are still under review, and 4 were closed due to project-specific policies.

As an example consider the `softlayer-python` project (Fig. 1). PYTRIM correctly identified that the `prettytable` dependency is unused. Then, it generated a PR to remove the dependency from four configuration files (`setup.py` and three separate `requirements.txt` files). PYTRIM also identified the unused package name in the `README.rst` file and correctly flagged it for manual review. The resulting PR, with the manual edit for the README, was merged by the project maintainers. All merged PRs are summarized in Table II.

Notably, bloated dependencies in `edx/edx-lint` and `optimizely/python-sdk` (Fig. 2) would have gone undetected without PYTRIM, as existing detectors fail to accurately identify their dependencies.

V. Related Work

While software debloating is a broad research area, this section focuses on work most directly related to Python dependency analysis. For comprehensive surveys of debloating techniques in other ecosystems, we refer readers to Drosos et al. [17] and the Systematization of Knowledge (SoK) work by Alhanahnah et al. [21]. Most relevant to our work is Drosos et al. [17], who conducted a large-scale study on dependency bloat in the Python ecosystem and established a methodology for identifying unused dependencies through fine-grained call graph analysis using the PyCG static call graph generator [22]. This research provides the foundational understanding of the problem space that PYTRIM builds upon.

Existing Python tools focus primarily on detection and can be broadly categorized into two groups. The first consists of source code linters, such as `autoflake` [9] and `pylint` [10], that identify unused imports directly within `.py` files. While some of these, such as `autoflake`, can remove the identified import lines, their scope is strictly limited to the source code; they do not modify project-level configuration files. The second group operates at the project level; tools such as `deptry` [12], and `FawltyDeps` [11] compare declared dependencies against source code imports to find unused packages, but they do not automate the removal of the corresponding `import` statements.

VI. Conclusion

We presented PYTRIM, a system for end-to-end removal of unused dependencies and imports. We outlined its detector-agnostic architecture, its novel dynamic dependency resolution technique, and its support for a wide range of configuration formats. Our evaluation demonstrates its effectiveness in uncovering previously undetectable dependency bloat and automatically removing it.

References

[1] S. Cass, "The top programming languages 2024," *IEEE Spectrum*, Aug. 2024. [Online]. Available: https://spectrum.ieee.org/top-programming-languages-2024

[2] "Octoverse: Ai leads python to top language as the number of global developers surges," accessed: 2025-07-03. [Online]. Available: https://github.blog/news-insights/octoverse/octoverse-2024

[3] Python Software Foundation, "Python package index (pypi)," https://pypi.org, 2025.

[4] R. Cox, "Surviving software dependencies," *Commun. ACM*, vol. 62, no. 9, p. 36–43, Aug. 2019. [Online]. Available: https://doi.org/10.1145/3347446

[5] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, p. 45, 2021.

[6] B. A. Azad, R. Jahanshahi, C. Tsoukaladelis, M. Egele, and N. Nikiforakis, "AnimateDead: Debloating Web applications using concolic execution," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5575–5591.

[7] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *Proceedings of the 29th USENIX Conference on Security Symposium*, ser. SEC'20. USA: USENIX Association, 2020.

[8] "Adding support for fancy tables," https://github.com/softlayer/softlayer-python/commit/1238377, 2022, accessed: 2025-07-23.

[9] "autoflake: Removes unused imports and unused variables from Python code," accessed: 2025-07-23. [Online]. Available: https://github.com/PyCQA/autoflake

[10] Pylint contributors, "Pylint," accessed: 2025-07-23. [Online]. Available: https://github.com/pylint-dev/pylint

[11] "Fawltydeps: Python dependency checker." accessed: 2025-07-06. [Online]. Available: https://github.com/tweag/FawltyDeps

[12] "deptry: Find unused, missing and transitive dependencies in a python project." accessed: 2025-07-23. [Online]. Available: https://deptry.com/

[13] T. Preston-Werner, "TOML: Tom's obvious minimal language," https://toml.io/en/, 2024, accessed: 2025-07-24.

[14] "The import system (regular packages)," https://docs.python.org/3/reference/import.html#regular-packages, 2022, accessed: 2025-07-23.

[15] Tom Forbes, "The contents of PyPI, in numbers," https://py-code.org/stats, 2025, accessed: 2025-07-23.

[16] "Requirements in setup.py are redundant, and partially inconsistent, with requirements.txt," https://github.com/pvlib/pvanalytics/issues/113, 2022, accessed: 2025-07-23.

[17] G.-P. Drosos, T. Sotiropoulos, D. Spinellis, and D. Mitropoulos, "Bloat beneath python's scales: A fine-grained inter-project dependency analysis," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 2584–2607, 2024.

[18] G. Alexopoulos, T. Sotiropoulos, G. Gousios, Z. Su, and D. Mitropoulos. (2025) PyXray: Practical cross-language call graph construction through object layout analysis. To appear at the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE). Accessed: 2025-07-23. [Online]. Available: https://grgalex.gr/assets/pdf/pyxray_icse26.pdf

[19] pipdeptree contributors, "pipdeptree," accessed: 2025-07-23. [Online]. Available: https://github.com/tox-dev/pipdeptree

[20] "Remove unused dependency: cryptography," accessed: 2025-07-23. [Online]. Available: https://github.com/simple-salesforce/simple-salesforce/pull/680

[21] M. Alhanahnah, Y. Boshmaf, and A. Gehani, "Sok: Software debloating landscape and future directions," in *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation*, 2024, pp. 11–18.

[22] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical call graph generation in python," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1646–1657.