Guarding the Lifeline: A First Look and Automated Defect Diagnosis for ROS Central Index

WEIJIE SUN, Nanjing University, China
HUIYAN WANG✉, Nanjing University, China
YING WANG, Northeastern University, China
CHANG XU, Nanjing University, China

The Robot Operating System (ROS) relies on a centralized dependency index, the `rosdistro` *central index*, to manage packages across its heterogeneous software ecosystem, which integrates independently evolving Operating System (OS) repositories for system libraries, ROS repositories for domain-specific support, and Programming Language (PL) repositories for functional modules. While this design enables portability, it introduces a critical source of fragility since ROS dependency management depends entirely on this manually curated, static index that must map packages across independently evolving, multi-source repositories. This leads to persistent defects in the central index, such as missing, incorrect, or outdated installation rules, which undermine the reliability of ROS dependency management. To address this problem, we conducted the first in-depth empirical study of 863 real-world maintenance cases involving the ROS central index. We categorize defects as either coverage or correctness defects, identify their underlying structural causes, and demonstrate that the primary bottleneck in manual maintenance is the difficulty of identifying equivalent packages across repositories. Motivated by these findings, we propose RosdepAuditor, an automated auditing framework that introduces a cross-repository mapping mechanism with hybrid scoring to infer package equivalence and detect defects. Evaluated on a ground-truth dataset, RosdepAuditor achieves 94.7% mapping accuracy without recommending non-existent packages, outperforming existing pattern-based and upstream-based approaches as well as leading large language models (LLMs). When applied to the live index, it uncovered 3,249 potential defects across 2,233 entries, 46 of which have been confirmed and fixed, demonstrating its practical usefulness in strengthening ROS dependency management.

CCS Concepts: • **Software and its engineering → Software libraries and repositories**; **Software evolution**.

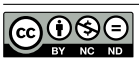
Additional Key Words and Phrases: ROS, Dependency Management, Package Management, Empirical Study, Software Ecosystems

ACM Reference Format:
Weijie Sun, Huiyan Wang, Ying Wang, and Chang Xu. 2026. Guarding the Lifeline: A First Look and Automated Defect Diagnosis for ROS Central Index. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA032 (October 2026), 23 pages. https://doi.org/10.1145/3832123

1 Introduction

The Robot Operating System (ROS) has emerged as a de facto standard for robotics research and development [1, 27]. Its success largely relies on a rich ecosystem of open-source, reusable packages

Authors' Contact Information: Weijie Sun, Nanjing University, State Key Lab of Novel Software Technology and School of Computer Science, Nanjing, China, sunweijie@smail.nju.edu.cn; Huiyan Wang, Nanjing University, State Key Lab of Novel Software Technology and Software Institute, Nanjing, China, why@nju.edu.cn; Ying Wang, Northeastern University, Software College, Shenyang, China, wangying@swc.neu.edu.cn; Chang Xu, Nanjing University, State Key Lab of Novel Software Technology and School of Computer Science, Nanjing, China, changxu@nju.edu.cn.



ACM 2994-970X/2026/10-ARTISSTA032
https://doi.org/10.1145/3832123

that provide tools, libraries, and conventions to accelerate the development of complex robotic behaviors [22]. Unlike ecosystems that use a single package source, ROS relies on three different types of package repositories, including Operating System (OS) repositories for foundational system libraries, ROS repositories for domain-specific support and integration, and Programming Language (PL) repositories for high-level functional modules. While this heterogeneity offers great flexibility and access to a wide range of software packages, it also significantly increases the complexity of ROS's dependency management, as the same package may be distributed under different names and versions across these independently evolving repositories. Therefore, ensuring feasible dependency resolution across such a fragmented landscape poses a major challenge.

To manage dependencies across complex, heterogeneous repositories, ROS adopts a centralized dependency management mechanism, built around its core structure, i.e., the `rosdistro` *central index*. By providing equivalent-package mappings across different repositories, the central index decouples high-level dependency declarations from platform-specific details. Thus, instead of specifying concrete package names, ROS developers only need to declare abstract *dependency keys*, and the ROS package manager `rosdep` consults the central index to resolve these keys into corresponding *dependency entries* with the correct, platform-specific packages and installation rules suited to developers' local environments. For instance, the central index maps the key `log4cxx` to `liblog4cxx-dev` on Ubuntu and `log4cxx-devel` on Fedora, as used by `rosdep` based on the local environment.

While this centralized design supports portability, it creates *a single point of fragility in ROS dependency management*, as the entire management system largely relies on a manually curated, static mapping to unify a dynamic, heterogeneous ecosystem, resulting in frequent maintenance issues [31]. These issues manifest in two main challenges. On one hand, *maintaining accurate cross-repository mappings in the central index is inherently difficult*, as it requires deep, expert-level familiarity with heterogeneous software repositories. For example, functionally equivalent packages can appear under entirely different names. The sound library packaged as `alsa-lib` on Fedora corresponds to `libasound2` on Ubuntu, with `lib32asound2` used in some older Ubuntu releases and `libasound2t64` used on Ubuntu Noble (24.04). Packages with similar names may offer incompatible functionality. A notable case is the Ubuntu package `python3-bson`, which shares a misleadingly similar name with the PyPI package `bson`, even though they are completely incompatible. Such incorrect mappings in the central index can lead to runtime failures in downstream projects [30]. On the other hand, *keeping the index synchronized with independently evolving repositories is practically infeasible under the current manual process*. For example, the central index has been reported to contain over 350 invalid and outdated dependency entries for Ubuntu Noble alone [31]. Broader package evolution, such as systematic renaming, exacerbates this problem. For example, the recent 64-bit time_t transition in Debian Trixie and Ubuntu Noble necessitated widespread library renaming [9], e.g., `libqt5core5a` to `libqt5core5t64`. Without proper renaming, downstream projects encounter build failures [40], further highlighting the limitations of the current maintenance process for the central index.

Despite the critical role of the central index, the ROS ecosystem lacks a systematic investigation into its defects and obstacles that impede effective maintenance. To address this, we conduct the first in-depth empirical study of 863 real-world maintenance cases from existing pull requests and issues. Our analysis reveals that index defects fall into two main categories stemming from the dual challenges of repository heterogeneity and evolution, i.e., resolution coverage defects (Type A.1 and A.2), which occur when the index contains no rule for a requested package, and resolution correctness defects (Type B.1 and B.2), where the retrieved rule is incorrect. We further characterize the maintenance burden and consistently observe bottlenecks stemming from the key difficulty of identifying equivalent packages across the diverse repositories used in ROS. Finally,

to inform potential automated maintenance solutions, we analyze 1,737 cross-repository index entries and identify key barriers (e.g., divergent package names and packaging policies) that render heuristic-based matching unreliable, underscoring the need for more sophisticated automation.

Therefore, motivated by our study, we propose RosdepAuditor, an automated auditing framework for detecting potential defects in the ROS central index to enhance dependency management. RosdepAuditor incorporates a cross-repository equivalent-package mapping mechanism that consolidates multi-dimensional package metadata to identify candidate equivalent packages across ROS-supported repositories, utilizing hybrid scoring to effectively prioritize and infer accurate equivalence mappings. By comparing inferred mappings against the entries in the central index, RosdepAuditor effectively identifies discrepancies and detects defects. Experimental evaluation shows that RosdepAuditor produced high-quality equivalence mappings, achieving a 94.7% match rate on a ground-truth dataset without referring to non-existent packages. RosdepAuditor outperformed all evaluated baselines, including pattern-based approaches such as RosdepCI [7][1] (67.6%), upstream-based approaches such as Repology [34] (81.4%), and leading LLMs, which, despite competitive matching, introduce significant proportions of unexpected non-existent packages among the unmatched, e.g., Gemini-3-Pro (92.4% accuracy, 38.1% non-existent packages). Leveraging these high-quality mappings, RosdepAuditor achieved 98.8% recall and 99.0% precision in detecting real defects in historical PRs. When applied to the live central index, it uncovered 3,249 potential defects across 2,233 dependency entries, providing actionable insights to strengthen ROS dependency resolution. Furthermore, 46 of the reported defects have been verified and promptly resolved, demonstrating RosdepAuditor's practical utility.

In summary, we make the following contributions:

- ***The First In-depth Empirical Study of the ROS Central Index***: We conducted a comprehensive examination of 863 real-world maintenance cases. This study systematically classifies four major defect types into two primary groups, while also uncovering their underlying root causes and maintenance bottlenecks.
- ***An Effective Technical Framework***: We propose an automated auditing framework that introduces a cross-repository mapping mechanism with hybrid scoring to accurately infer package equivalence and accordingly identify potential defects in the central index.
- ***Tool Implementation and Practical Utility:*** We developed and extensively evaluated RosdepAuditor. The tool outperformed existing baselines and uncovered thousands of potential defects in the live index, dozens of which have been verified and resolved. These results demonstrate strong real-world utility.

The remainder of this paper is organized as follows. Section 2 introduces the background of ROS dependency management. Section 3 conducts an empirical study to explore defects and maintenance bottlenecks of the ROS central index. Section 4 elaborates on RosdepAuditor to diagnose defects in the central index based on its core equivalence mapping mechanism. Section 5 evaluates RosdepAuditor with real-world datasets. Sections 6 and 7 discuss the threats to validity and the related work. Finally, Section 8 concludes the paper.

2 Background

2.1 Heterogeneous Software Repositories

Software is structured into *packages*, the fundamental units for building, distributing, and reusing code, which are usually hosted in and retrieved from repositories. Each *repository* refers to a structured storage location or server that distributes software metadata and packages. As illustrated in Figure 1, the ROS ecosystem relies on three types of heterogeneous package repositories:

[1]We refer to the CI tool embedded in the `rosdistro` repository as RosdepCI.

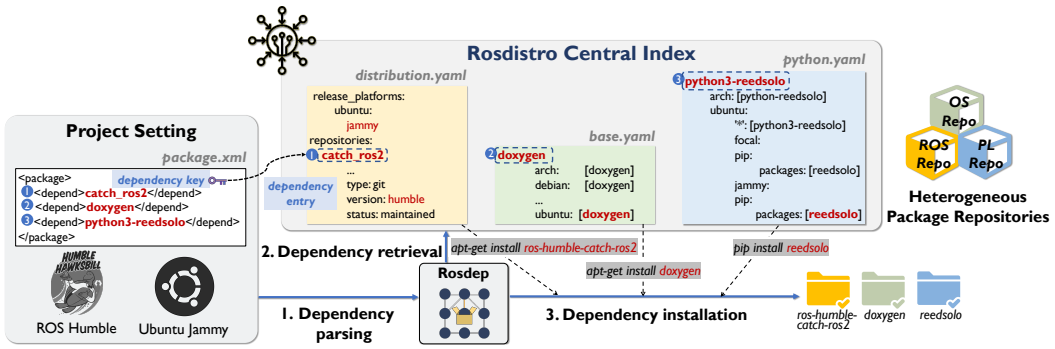


Fig. 1. Workflow of ROS Dependency Management

(1) ROS Repositories. ROS repositories are dedicated to hosting ROS packages and are maintained as release-based distributions. They provide officially released ROS libraries and tools that are compatible with specific ROS distributions (e.g., `humble`).

(2) Operating System (OS) Repositories. Since ROS runs on general-purpose operating systems, it inherently relies on packages provided by OS repositories. Each operating system (e.g., Ubuntu, Fedora) maintains its own native repository and package manager (e.g., `apt`, `dnf`), which supply core system libraries and utilities required by ROS projects.

(3) Programming Language (PL) Repositories. Since ROS projects are implemented in languages like Python or C++, they often depend on language-specific libraries in PL repositories (e.g., PyPI for Python). These repositories provide functional modules for application development.

The coexistence of these three repository types contributes to the flexibility of ROS software, with ROS repositories providing domain-specific support, OS repositories supplying foundational libraries, and PL repositories delivering functional modules. However, since these repositories are maintained independently, this creates a complex landscape for ROS dependency management.

2.2 ROS Package and Dependency Management

To declare and use packages from the heterogeneous repositories, ROS supports cross-repository package management via `rosdep`, which depends on the core `rosdistro` *central index* to obtain package mappings in dependency entries. Each *dependency entry* associates an identifier (its *entry key*) with different *installation rules* that specify which packages to install and which package manager to use for each specific repository. Typically, the central index categorizes dependency keys into two major groups based on the repositories they target. For ROS dependencies, as illustrated in the `distribution.yaml` section of Figure 1, each entry records the source repository, release configuration, and versioning metadata. For non-ROS dependencies provided by OS or PL repositories, dependency entries are organized into separate `YAML` files by repository type, e.g., `base.yaml` for OS repositories. In this way, ROS employs a coordinated, centralized dependency management mechanism.

We provide an illustrative example in Figure 1. A ROS project usually declares its package dependencies in a manifest file, `package.xml`, using abstract identifiers called *dependency keys*, rather than platform-specific package names. The manifest does not specify which `YAML` file should be used for each key. Instead, `rosdep` searches the files exhaustively for a matching dependency entry. To resolve these keys for installing packages, the ROS package manager generally follows a three-step process:

(1) **Dependency Parsing.** `rosdep` parses dependency keys from `package.xml`. For example, the project in the figure declares dependencies with three dependency keys, i.e., `catch_ros2`, `doxygen`, and `python3-reedsolo`, and is to be built in a local environment running ROS Humble on Ubuntu Jammy (22.04).
(2) **Dependency Retrieval.** For each key, `rosdep` searches the central index for a matching dependency entry and retrieves its installation rules. Specifically, the located dependency entry provides the platform-specific package names and package manager to use in the local environment. For example, `catch_ros2` is matched in `distribution.yaml` for package `catch_ros2` in ROS repositories, `python3-reedsolo` in `python.yaml` for `reedsolo` in the PyPI repository, and `doxygen` in `base.yaml` for `doxygen` in OS repositories.
(3) **Dependency Installation.** `rosdep` then installs the packages with retrieved names via the corresponding package managers. For example, `rosdep` resolves the dependency keys `catch_ros2` and `doxygen` by installing `ros-humble-catch-ros2` from the ROS repository and `doxygen` from the Ubuntu repository, both via `apt`. For the local Ubuntu Jammy environment, `rosdep` resolves the key `python3-reedsolo` by installing the `reedsolo` package from PyPI via `pip`.

Overall, the `rosdistro` central index decouples *declaration* (stable dependency keys in projects) from *realization* (platform-specific packages and installation rules) and serves as the *core* design supporting ROS's complex dependency management across heterogeneous repositories.

3 Empirical Study

To investigate the challenges of maintaining the `rosdistro` central index, we conducted an empirical study guided by three research questions:

- **RQ1 (Taxonomy of Defects):** What are the primary types of defects in the `rosdistro` central index?
- **RQ2 (Maintenance Bottleneck):** How costly are defects in the central index to maintain in practice, and what bottlenecks hinder efficient maintenance?
- **RQ3 (Automation Barriers):** What factors affect effective automation in mapping equivalent packages across heterogeneous repositories?

3.1 Data Collection Process

To answer these questions, following prior empirical studies on software change and issue repositories [24, 71, 76], we collected and analyzed data from both the maintenance history and the current status of the `rosdistro` central index.

Step 1: Curation of the Maintenance Dataset. By mining the official `rosdistro` GitHub repository [50] (the central index for ROS), we collected all historical issues and `rosdep`-related pull requests (PRs), totaling 534 issues and 1,002 PRs as of April 2025. To construct a suitable dataset for defect analysis, we retained only cases that were resolved or had a clear resolution plan involving modifications to `rosdep` index-related fields, such as installation rules, since `rosdistro` also serves as the publishing repository for ROS packages. After removing duplicate cases and excluding issues that were later resolved by already captured pull requests, we obtained a final set of 863 unique maintenance cases, including 28 issues and 835 PRs.

Step 2: Analysis of Defects and Maintenance Bottleneck. We utilized this curated dataset to answer the first two research questions. To address **RQ1**, we employed a qualitative analysis of all 863 maintenance cases. This involved a manual inspection of the detailed modifications to the central index (e.g., introducing new dependency keys, revising existing rules, or broadening platform support) to derive defect taxonomies. To address **RQ2**, we analyzed 835 PRs by tracing

Table 1. Taxonomy and Frequency of Central Index Defects

Category	Defect Type	Frequency
A. Resolution Coverage Defects	A.1 Missing Dependency Definition	625 (72.4%)
(No Rule Found)	A.2 Incomplete Platform Coverage	181 (21.0%)
B. Resolution Correctness Defects	B.1 Invalid Package Specification	111 (12.9%)
(Incorrect Rule Found)	B.2 Suboptimal Repository Prioritization	12 (1.4%)

their maintenance lifecycles, including commit histories, maintainer discussions, and continuous integration logs, to identify potential bottlenecks to efficient maintenance. As will be discussed in Section 3.3, the primary bottleneck to effective maintenance arises from the difficulty of finding equivalent packages across the heterogeneous repositories supported by `rosdep`. Thus, given this finding, we further raised RQ3 to investigate the key factors underlying these mapping challenges.

Step 3: Analysis of Cross-Repository Packaging Inconsistencies. To address **RQ3**, we analyzed the current status of the central index. Since RQ3 focuses on cross-repository packaging inconsistencies, we filtered the entries and retained only those whose package mappings span different repositories, resulting in 1,737 dependency entries for this analysis.

To avoid analysis bias arising from inconsistencies due to additional hierarchical information attached to all packages in certain repositories, such as category affixes for Gentoo and OpenEmbedded, we isolated the core package names from entries. We then investigated these entries alongside their package mappings and identified hidden factors that impede cross-repository package identification, thereby motivating our solution later.

3.2 RQ1: Taxonomy of Defects in the Central Index

Our analysis of the 863 maintenance cases shows that defects in the `rosdistro` central index fall into two categories: inability to locate a rule (*resolution coverage defects*) and provision of an incorrect rule (*resolution correctness defects*). Table 1 summarizes the subtypes and their frequencies. Note that a single maintenance case may involve multiple defect types (e.g., adding a missing key while also extending platform coverage).

Type A: Resolution Coverage Defects. These defects arise when `rosdep` fails to retrieve any installation rule from the central index for a requested dependency key. They indicate gaps in the index's completeness and primarily reflect the potential effort required to maintain mappings across the ROS ecosystem. We identified two representative subtypes:

Type A.1: Missing Dependency Definition (625/863). The defect occurs when a dependency key declared in a project's manifest file lacks a corresponding dependency entry in the central index, preventing this key's dependency resolution for all platforms. The root cause typically lies in the scale of the ecosystem, as maintainers cannot proactively include every possible package and often rely on community contributions to expand coverage. For example, in issue #36442 [37], the package `sparkfun-qwiic` needed to be installed but was completely absent from the central index, along with its installation rules for all platforms. To fix this, the maintainer instructed the project developers to add the missing entry themselves by following the contribution guidelines.

Type A.2: Incomplete Platform Coverage (181/863). The defect occurs when the dependency entry exists for the requested package but is missing the installation rule for the target platform. This defect primarily arises because contributors often set up rules only for popular platforms, while less common platforms are overlooked due to unfamiliarity or lower priority. This imbalance is reinforced by `rosdistro` contribution guidelines [51], which only require installation rules for Debian and Ubuntu. For example, in issue #34332 [36], the entry for `libopenni2-dev` had rules for Debian and Ubuntu but none for `rhel:8` (Red Hat Enterprise Linux 8), preventing users from

building their projects on that platform. This lack of platform coverage may be exacerbated as repositories evolve. When a package becomes available on a new platform, the central index should be updated accordingly, which requires substantial manual effort. For example, in PR #38540 [44], multiple `flake8`-related entries were updated because the packages had become available on Debian and Fedora after their initial release, leaving incomplete entries until they were manually added.

Type B: Resolution Correctness Defects. These defects occur when an installation rule exists for a dependency key and platform in the central index, but the rule itself is incorrect. This category distinguishes between rules that identify invalid packages within a target repository and those that identify packages from a non-preferred repository.

Type B.1: Invalid Package Specification (111/863). The defect occurs when the installation rule exists, but refers to an incorrect package for installation in the target repository. This is an intra-repository mismatch, typically caused when a rule becomes outdated after a change in the external repository, such as a package being renamed or removed. For instance, in issue #42698 [39], the index specified `pybind11-devel` for installation on RHEL 8. However, this package was later removed from the RHEL 8 repository. The rule became outdated, causing a Type B.1 defect and preventing developers from resolving the dependency on that platform.

Type B.2: Suboptimal Repository Prioritization (12/863). The installation rule specifies installation from a suboptimal repository. This defect occurs when an installation rule refers to a package from a non-OS repository even though an equivalent native OS package exists. This violates the ROS policy, which requires using native OS packages when available to prevent incompatibilities. This violation poses a risk to ROS because it can cause projects to link against an incompatible mix of packages, potentially resulting in runtime failures. The defect often happens when OS repositories include new packages, but the central index is not updated in time. For example, in issue #41622 [38], the key `octomap` still resolved to a non-OS package after `octomap` was added to the Ubuntu Noble repository. Because the native OS and non-OS versions were incompatible with each other, downstream projects like `moveit2` had to add workarounds to avoid linking to the wrong package (issue #2862 in moveit/moveit2) [28].

Overall, these two categories of defects reflect two different sources of fragility within the central index. Together, these findings show that the central index in the ROS ecosystem is inherently prone to errors due to two main reasons: the impracticality of manually tracking the vast and diverse software ecosystem, and the difficulty of keeping the index synchronized with the independently evolving external repositories it depends on.

Answer to RQ1: *Across 863 maintenance cases in the central index, we observe two defect categories: (1)* ***Resolution Coverage Defects*** *(806 cases), where the index fails to locate any installation rule for the requested key, and (2)* ***Resolution Correctness Defects*** *(123 cases), where the located rule is erroneous. These defects stem from the dual challenges of repository heterogeneity and evolution.*

3.3 RQ2: Maintenance Bottleneck

While RQ1 identifies common defects in the central index, RQ2 explores how these defects become maintenance burdens and pinpoints the recurring bottlenecks that slow down the dependency maintenance process in practice. To answer RQ2, we analyzed 835 merged PRs. By reviewing their timelines, revision histories, discussion threads, and CI/CD feedback, we traced their progress and identified the key bottleneck that stalled maintenance.

Maintenance Burdens. Defect resolution is labor-intensive and time-consuming. Our analysis of 835 pull requests shows an average resolution time of 248.6 hours (10.4 days). The process was highly review-intensive, with 3,915 comments across 835 PRs (4.69 comments per PR on average) and manual review comments appearing in 808 PRs (96.8%). Moreover, 555 PRs (66.5%) required

post-review iterations, indicating the need for multiple rounds of feedback and revision after the initial submission. This is largely because the developers' suggested equivalent packages to install may be incorrect. Even when a PR initially passes CI/CD checks and maintainer review, it may still require revision, turning the intended linear maintenance workflow into a trial-and-error cycle and incurring time-consuming maintenance iterations.

To identify where maintenance iterations typically occur, we analyzed changes made during revisions within each PR (*within-PR revisions*). Among the 555 PRs that required post-review changes, 402 PRs (72.4%) required changes to the package-mapping details for installation rules, suggesting that post-review iterations more often concern the refinement of the correctness and completeness of dependency entries. For example, in issue #49424 [43], contributors struggled to find the correct equivalent package, `glibc-langpack-de`, for RHEL, due to limited platform familiarity. This case illustrates that obtaining complete and accurate installation rules remains challenging because of the inherent mismatch between the central index's requirements for cross-platform validity and contributors' inherently localized knowledge. We also evaluated maintenance efforts by repeatedly examining the same specific dependency entries across PRs (*cross-PR revisiting*). Of the 1,921 unique entries analyzed, 990 (51.5%) were modified in more than one PR, with an average of 2.94 updates per entry. Although some repetitions can be explained by repository evolution that changes package availability or naming, 137 entries (13.8% of the repeatedly modified ones) were even updated within a short period, e.g., within a month. For instance, the entry key `libboost-python` underwent 12 revisions in total and exhibited burstiness in its maintenance process, including one period with three updates within a single month and another with four updates within three months. Since repository evolution is unlikely to occur so frequently within such short intervals, this indicates significant efforts to correct earlier mistakes or incompleteness, even after initial reviews and CI/CD verification.

Root Cause and Bottleneck. These observations on the repeated within-PR revisions and cross-PR revisiting for the same dependency keys suggest that contributors and maintainers often lack ecosystem-wide, up-to-date information needed to reliably map equivalent packages across ROS's repositories. While automated CI/CD tools offer some assistance, their capabilities are limited to name-based heuristics and often fail to identify valid mappings when packages are named differently. Our analysis further confirms this bottleneck: when package names differ, 66.2% of related PRs require post-review revisions, compared with 26.4% otherwise. For example, in PR #32002 [35], the contributor mapped package `python3-gi-cairo` for Debian, Ubuntu, and Arch, but missed the Fedora mapping, where the equivalent package is named `python3-gobject`. Identifying such inconsistencies still relies heavily on manual searches of distribution repositories, a time-consuming and error-prone process, as noted in maintainer discussions [52]. This highlights a critical need for an autonomous, accurate, and sustainable solution that can track package mapping across repositories. Such a solution would support reliable identification of equivalent packages, enabling more effective defect resolution and significantly reducing the central index's current maintenance burden.

Answer to RQ2: *Defect handling incurs clear maintenance burdens, manifested as repeated revisions within the same PR (66.5% of PRs requiring post-review iterations) or updates to the same dependency entries across multiple PRs (2.94 updates on average for 51.5% of entries) in the central index. The key bottleneck in maintenance is the difficulty of identifying equivalent packages across ROS's heterogeneous repositories.*

3.4 RQ3: Automation Barriers

To inform a potential solution for accurately and sustainably identifying equivalent packages across repositories, we examine the barriers to automation in RQ3. We analyzed the 1,737 cross-repository

entries collected in Section 3.1 and found that 76.6% (1,331 entries) could not be matched directly by name, a task that traditional CI/CD methods usually fail to handle. By examining package metadata, we identified four main barriers that prevent autonomous mapping, whether using names or package contents. These barriers are not mutually exclusive: a single entry may exhibit multiple barriers.

Divergent Affixation Conventions (1,007/1,331). Different repositories use different naming conventions, often adding varied prefixes or suffixes to indicate details like Python modules or development libraries. For instance, package `meson-python` on PyPI is named `python3-mesonpy` on Ubuntu and `python3-meson-python` on Fedora. Likewise, development libraries conventionally use the `-dev` suffix in Ubuntu but `-devel` in Fedora. Such divergent affixation poses a challenge for name-based automation. We compiled affixation patterns, including role prefixes and suffixes, and cross-distribution correspondences into an affixation rule base, available in our replication package. For example, `python3-`, `py3-`, `python3Packages.` are just a few of the prefixes used for Python 3 packages in Ubuntu, Alpine, and NixOS, respectively.

Evolving Upstream Repository (240/1,331). Package names may change in response to upstream evolution, such as ABI transitions or major functionality upgrades. For instance, the 64-bit `time_t` ABI transition in the GNU C Library triggered substantial downstream adaptation in Debian Trixie and Ubuntu Noble, affecting hundreds of library packages and requiring many of them to be rebuilt or renamed with the `t64` suffix [9]. In Debian-based systems, such upstream changes are often reflected in package names through ABI-related suffixes or API/ABI version identifiers, as illustrated by `libwxgtk3.0-gtk3-dev` and `libwxgtk3.2-dev`.

Differing Packaging Policies (526/1,331). Different repositories may also employ different packaging policies, such as varying granularities and naming philosophies.

(1) Granularity. Package granularity refers to how much functionality a distribution bundles into a single unit. It usually reflects the distribution's packaging philosophy. Some repositories, such as Ubuntu, follow a modular policy, splitting large libraries into multiple smaller packages for convenient modular installation, while others, such as Fedora, prefer a more consolidated policy, bundling related components into larger packages. For example, Ubuntu splits Qt6's related functionality into `libqt6qml6`, `libqt6quick6`, and `qml6-module-qtquick`, whereas Fedora provides it as a single package `qt6-declarative`.

(2) Naming Philosophy. Beyond packaging granularity, repositories apply different naming philosophies, such as naming after the upstream project (e.g., `alsa-lib`), and naming after core functionality (e.g., `libasound2`). This indeed leads to complex cross-repository package correspondence, e.g., `liblzma-dev` (Ubuntu) corresponding to `xz` (Fedora), `lib32asound2` (Ubuntu) to `alsa-lib` (Fedora and Arch), and `libabsl-dev` (Ubuntu) to `abseil-cpp-devel` (Fedora).

Inconsistent Structural Content (568/1,331). Beyond naming, a more fundamental challenge is the structural difference in package contents, which further complicates content-based automation. We observed that, for 42.7% of entries with naming inconsistencies, packages considered equivalent nevertheless differed in structural content. We identify two sub-factors:

(1) Path Specialization: File paths are sometimes specialized differently in package contents across varied repositories, such as different root path *prefixes* (e.g., "`/usr/lib64/`" vs. "`/usr/lib/`" vs. "`/usr/lib/x86_64-linux-gnu`"), special identifier *infixes* (e.g., the arch identifier "aarch" within filepath "`.../atlas-aarch64-base/zmm.h`" or the version identifier "6.4.2" within filepath "`.../qt6/QtCore/6.4.2/QtCore/private/minimum-linux_p.h`"), and version *suffixes* (e.g., the version suffix for compiled module "`.../libQt6Core5Compat.so.6.10.1`").

(2) File Heterogeneity: Repositories may distribute the same packages in different forms. For example, the package `pygobject` is distributed as source code (i.e., `.py` files) on PyPI but as

precompiled `.so` binaries on Fedora, resulting in substantially different filelists despite referring to the equivalent underlying package.

These barriers hinder automated equivalence identification across repositories, limiting maintenance of the central index. While divergent affixation conventions can be mitigated by pattern-based heuristics, other barriers in repository evolution, differing packaging policies, and structural content inconsistencies still undermine straightforward name- and content-based mapping.

Answer to RQ3: *Equivalent-package identification across ecosystems is challenging, with four key barriers noted among 1,331 entries investigated: divergent affixation conventions (1,007), evolving upstream repositories (240), differing packaging policies (526), and inconsistent structural content (568). These barriers undermine both name-based and content-based automated identification, motivating a more sophisticated, multi-dimensional approach to improving central index maintenance.*

4 Methodology

This section presents RosdepAuditor, an automated framework for diagnosing defects in the `rosdistro` *central index*, which serves as the core infrastructure supporting ROS dependency maintenance.

4.1 Formal Model

Before elaborating on the methodology, we first define the formal model underlying the problem.

Platforms, Repositories, and Packages. Let PF be the set of platforms officially supported by `rosdep`. For each platform $pf \in \mathsf{PF}$, we define the function $\mathrm{ReposOf}(pf)$ that identifies the set of repositories compatible with pf from which packages can be obtained. These include OS-native repositories, programming language-specific repositories, and ROS-specific repositories[2]. For example, projects on the Ubuntu Noble platform can use packages from the native Ubuntu repository, the Python Package Index (PyPI) repository, and the ROS Jazzy repository.

Here, we use rid to refer to each repository that distributes packages. Let $\mathcal{R}_{rid}$ denote the set of packages in repository rid. Each package can be represented as a metadata tuple:

$$p = \langle rid, name, desc, files, upstream \rangle.$$

We use $\mathrm{Rid}(p)$, $\mathrm{Name}(p)$, $\mathrm{Desc}(p)$, $\mathrm{Files}(p)$, and $\mathrm{Up}(p)$ to access these fields for the repository identifier, package name, description, filelist, and the upstream project from which the package is built.

Central Index. The central index functions as a centralized mapping, denoted $\mathcal{RM}$, that associates a dependency key with a platform and its specific installation rules. For a dependency key dk, we refer to its corresponding mapping as $\mathcal{RM}_{dk}$, and $\mathcal{RM}_{dk}(pf)$ is used to retrieve the corresponding installation rule for platform pf. If no rule exists for the key on the given platform, then $\mathcal{RM}_{dk}(pf) = \bot$. A key dk is considered *undefined* if $\mathcal{RM}_{dk}(pf) = \bot$ for all $pf \in \mathsf{PF}$.

Otherwise, the installation rule returned by $\mathcal{RM}_{dk}(pf)$ identifies a package p to be installed, with the corresponding repository identifier and package name populated in the tuple. We then define $\textsc{Locate}(p)$, which searches repository $\mathcal{R}_{\mathrm{Rid}(p)}$ for $\mathrm{Name}(p)$ and returns the complete package if it exists, and $\bot$ otherwise.

4.2 RosdepAuditor Workflow

As illustrated in Figure 2, RosdepAuditor follows a four-step framework. First, for the given dependency key, RosdepAuditor looks up its current dependency entry in the central index and resolves a *reference package* from that entry, which serves as the starting point for searching equivalent packages across supported repositories. Second, using normalized, multi-dimensional metadata

[2]ROS distributions officially target specific OS platforms to ensure compatibility with the underlying system.

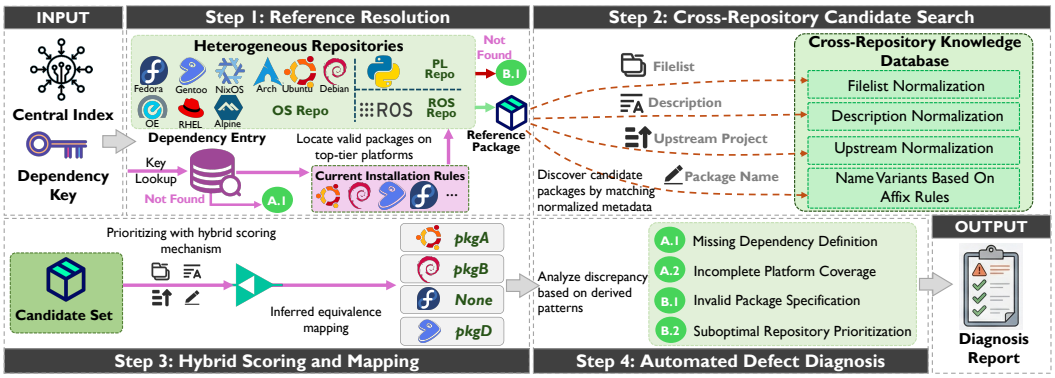


Fig. 2. Workflow of RosdepAuditor

stored in a cross-repository knowledge database, RosdepAuditor searches supported repositories for packages that may be equivalent to the reference package, as inspired by Section 3.4. Third, by adopting a hybrid scoring mechanism, RosdepAuditor prioritizes package candidates across repositories and derives its suggested equivalence mapping. Finally, RosdepAuditor compares the original package mappings with their inferred counterparts, diagnoses defects according to the types identified in Section 3.2, and outputs the final diagnosis report. We elaborate on each step below.

4.2.1 Step 1: Reference Resolution. Consider a requested dependency key dk. RosdepAuditor first looks up the corresponding dependency entry in the central index. RosdepAuditor examines the packages in the entry according to a priority order $\langle pf_1, pf_2, \dots \rangle$[3], starting with Ubuntu, because higher-priority platforms are generally better maintained. For each platform pf in this order, RosdepAuditor checks whether an installation rule exists in the central index (i.e., $\mathcal{RM}_{dk}(pf) \neq \perp$) and whether the package specified by that rule can be successfully located in the corresponding repository via Locate. RosdepAuditor selects the package associated with the first platform that satisfies both conditions as the reference package s, and denotes this platform by $pf^{\star}$.

If RosdepAuditor fails to find a reference package on any top-tier platform, the search stops. Specifically, if the entire dependency entry is missing (i.e., dk is undefined), a **Type A.1** defect is reported, and if none of the packages specified by the rules can be located using Locate, a **Type B.1** defect is reported.

4.2.2 Step 2: Cross-Repository Candidate Search. In this step, RosdepAuditor searches for package candidates possibly equivalent to the reference package s across supported repositories. Because relying solely on name-based or content-based metadata can create blind spots across heterogeneous repositories, RosdepAuditor retrieves package metadata across four dimensions for candidate search and applies dedicated normalization procedures to address the barriers discussed in Section 3.4.

Metadata Retrieval and Normalization. For repository rid, we scan all packages and retrieve metadata for each package p, including repository identifier $\mathrm{Rid}(p)$, package name $\mathrm{Name}(p)$, description $\mathrm{Desc}(p)$, filelists $\mathrm{Files}(p)$, and upstream information $\mathrm{Up}(p)$. Consider a package from the `ubuntu_noble` repository, whose metadata tuple is $p = \langle$`ubuntu_noble`, `liblzma5`, `XZ-format...`,

[3]Platform priority follows the official `rosdistro` guidelines: Ubuntu and Debian are top-priority, Fedora and Gentoo are recommended when available, and other platforms may also be supported [51].

{/usr/lib/x86_64-linux-gnu/liblzma.so.5.8.1, ...}, xz-utils⟩. Because repositories follow different metadata conventions, package names and contents can differ substantially. We normalize the metadata to mitigate these inconsistencies. Specifically, we lowercase the description text in Desc(p) and remove its punctuation, and we use the affixation knowledge derived from our empirical study to strip repository-specific affixes from the paths in the filelist Files(p). We further normalize upstream information by grouping packages derived from the same source project, relying on an external database [34] that aggregates packaging information using a ruleset to map distribution-specific packages to their canonical upstream source. Therefore, for package liblzma5, its normalized metadata is p' = ⟨ubuntu_noble,liblzma5,xz-format...,{liblzma.so, ...},xz⟩, facilitating subsequent content-based mapping. We store the normalized metadata for all packages in supported repositories in a database of package tuples.

Candidate Search. Then, based on the normalized metadata database, we search for candidates equivalent to the reference package s. Before the search, we normalize the metadata for s and obtain its normalized representation s'. A package that shares an identical normalized description, an identical upstream project, or at least one common functional file is regarded as possibly equivalent and is thus treated as a candidate. We also perform a name-based candidate search by comparing Name(p') and Name(s'). To alleviate name inconsistencies caused primarily by affixation divergence, we generate variants of Name(s') by exhaustively appending possible prefixes and suffixes using the affixation knowledge, such as x-devel and libx-devel when Name(s') = x. Any package whose Name(p') matches a generated variant of Name(s') is also treated as a candidate. In this way, we obtain the final candidate set C_{rid} by considering all metadata dimensions.

4.2.3 Step 3: Hybrid Scoring and Mapping. While Step 2 focuses on identifying potential candidates for equivalent packages, it may yield a large candidate set. In this step, RosdepAuditor adopts a hybrid scoring mechanism to prioritize candidates and identify the most likely equivalent package in each repository, thereby obtaining the final equivalence mapping.

Scoring Dimensions. We design a multi-dimensional scoring mechanism to balance the different dimensions of information and their contributions, considering each package candidate and its normalized metadata tuple. First, we decompose each package candidate's name, i.e., Name(c), into role tags roles(c), a core string core(c), and version attributes ver(c). The core string represents the core package identifier, while the other components reveal auxiliary information such as library roles or versions. To do so, we use our affixation rule base to map the affixes matched in a candidate name to the corresponding roles of c, i.e., roles(c). Next, we interpret any numeric segments as the version value associated with the immediately preceding substring key, forming c's version attribute dictionary, i.e., ver(c). We finally designate the remaining string components as the core string, the most important part in the package name. Consider the example qt6-qtmultimedia-dev. The parser first removes the suffix -dev to identify the development role. It then interprets 6 as the version for the substring qt and retains qt-qtmultimedia as the core string.

Following such name decomposition, we calculate three name-based equivalence scores between the reference package s and each candidate c: $S_{\text{name\_base}}(s,c)$, $S_{\text{name\_role}}(s,c)$, and $S_{\text{name\_ver}}(s,c)$, which quantify equivalence along the three name-specific dimensions:

$$S_{\text{name\_base}}(s,c) = \frac{2 \cdot \text{LCS}(\text{core}(s), \text{core}(c))}{|\text{core}(s)| + |\text{core}(c)|} \quad \text{and} \quad S_{\text{name\_role}}(s,c) = \frac{|\text{roles}(s) \cap \text{roles}(c)|}{|\text{roles}(s)|},$$

where LCS(a, b) denotes the length of the longest common subsequence [18]. We calculate version compatibility $S_{\text{name\_ver}}(s,c)$ as 0 if any shared dictionary element has conflicting versions and 1 otherwise.

For the remaining metadata dimensions—description, filelists, and upstream identifiers—we compute description similarity $S_{\text{desc}}(s,c)$ as the cosine similarity between sentence embeddings

generated by the commonly used `all-MiniLM-L6-v2` model. We calculate the filelist overlap proportion $S_{\text{file}}(s, c)$ and upstream consistency $S_{\text{up}}(s, c)$ using a set-overlap ratio and an indicator function, respectively, as follows:

$$S_{\text{file}}(s, c) = \frac{|\textsc{Files}(s) \cap \textsc{Files}(c)|}{|\textsc{Files}(s)|} \quad \text{and} \quad S_{\text{up}}(s, c) = \mathbb{I}[\text{Up}(s) = \text{Up}(c)].$$

If any dimension is unavailable, we set the corresponding score to 0.

Prioritization and Mapping. RosdepAuditor operates by scoring two groups of factors. Core name similarity, file overlap, and upstream agreement serve as primary factors, providing definitive evidence for identity matching. In contrast, name role/version attributes and description text serve as secondary factors, offering supplementary evidence. The primary score is calculated as

$$S_{\text{pri}}(s, c) = \text{Mean}\left(\{S_{\text{name\_base}}(s, c), S_{\text{file}}(s, c), S_{\text{up}}(s, c)\}\right)$$

and the secondary score as

$$S_{\text{sec}}(s, c) = \text{Mean}\left(\{S_{\text{desc}}(s, c), S_{\text{name\_role}}(s, c), S_{\text{name\_ver}}(s, c)\}\right).$$

Then, RosdepAuditor combines the scores from both groups, weighted by a parameter $w \in [0, 1]$ (with a default value of 0.7), which balances the contributions of the primary and secondary factors.

$$S(s, c) = w \cdot S_{\text{pri}}(s, c) + (1 - w) \cdot S_{\text{sec}}(s, c).$$

For each repository rid, RosdepAuditor prioritizes candidates and selects the one in C_{rid} that achieves the highest $S(s, c)$ as the chosen package for this specific repository. Then, RosdepAuditor aggregates all the chosen packages for supported repositories and forms its final equivalence mapping $\mathcal{EM}_{dk}$, representing the mapping it considers optimal for the dependency key dk across all supported repositories. In rare instances where the reference s from Step 1 is a set of packages, we treat each package as an independent reference and take the union of all highest-scoring candidate selections.

4.2.4 Step 4: Automated Defect Diagnosis. Based on the equivalence mapping $\mathcal{EM}_{dk}$ generated in Step 3, RosdepAuditor systematically audits the central index. By comparing the original mapping in the central index $\mathcal{RM}_{dk}$ against RosdepAuditor's inferred equivalence mapping $\mathcal{EM}_{dk}$, we can identify typical defects according to the defect types defined in Section 3.2:

- **Type A.1: Missing Dependency Definition.** This defect is flagged when the dependency key dk is entirely undefined in the central index across all supported platforms.
- **Type A.2: Incomplete Platform Coverage.** This defect is reported when the central index lacks a rule for a specific platform pf, while RosdepAuditor successfully infers a valid package from a repository associated with pf.
- **Type B.1: Invalid Package Specification.** This defect is reported when the existing rule in the central index conflicts with RosdepAuditor's inferred equivalence mapping. Formally, we report this defect if for the repository rid from which the platform pf installs packages, the originally specified package differs from the package in the inferred equivalence mapping.
- **Type B.2: Suboptimal Repository Prioritization.** This defect is reported when the index directs users to an auxiliary repository (e.g., PyPI) even though a valid package exists in the native OS repository. We report this when $\mathcal{RM}_{dk}(pf)$ utilizes packages from a non-native repository, whereas RosdepAuditor identifies a valid package in the native OS repository.

5 Evaluation

We evaluate the performance of RosdepAuditor by answering three research questions.

- **RQ4 (Equivalence Mapping Effectiveness):** *How effective is RosdepAuditor's core mechanism at identifying equivalent-package mappings across repositories?*
- **RQ5 (RosdepAuditor Effectiveness):** *How effective is RosdepAuditor in diagnosing defects within the central index?*

- **RQ6 (RosdepAuditor Usefulness):** *How useful is RosdepAuditor for the ongoing maintenance of the central index?*

5.1 Experimental Setup

5.1.1 Dataset Construction and Ground Truth. To evaluate the performance of RosdepAuditor in core equivalence mapping (RQ4) and defect diagnosis (RQ5), we collected real-world dependency entries and pull requests to build separate ground-truth datasets. For the former, we selected actively maintained dependency entries that were manually verified between May 2025 and September 2025, a period that does not overlap with our empirical study. These entries met two criteria: (1) they were mapped to Ubuntu Noble, a tier-one platform officially supported by ROS, and (2) they covered installable packages across at least two package repositories, ensuring that the dataset supported reliable evaluation of cross-repository equivalence mapping. To facilitate evaluation and metric calculation, we decomposed each entry into a list of *package-mapping instances* $I = (K, R, P)$, where K represents the dependency key, R denotes the target repository, and P is the ground-truth package name. For example, the dependency entry of `libqt6multimedia6` was decomposed into (`libqt6multimedia6`, Ubuntu Noble, `libqt6multimedia6`) and (`libqt6multimedia6`, Fedora 41, `qt6-qtmultimedia`). We regarded the resulting 854 instances as the ground-truth dataset $Dataset_{em}$ for evaluating equivalence mapping across repositories. For the latter, we collected PRs submitted to the `rosdistro` repository during the same period to address real-world defects in the central index identified by developers. Because the selected period contained too few Type B.2 defects, we supplemented $Dataset_{df}$ with Type B.2 cases from our empirical study. The resulting ground-truth dataset $Dataset_{df}$ covers 75 PRs with 83 verified real defects and is used to evaluate defect diagnosis in the central index.

5.1.2 Metrics and Baselines. To evaluate RosdepAuditor's effectiveness in inferring equivalence mappings, we designed the following metrics:

- **Exact Match Rate:** An instance $I = (K, R, P)$ is considered *exactly matched* if the approach outputs exactly and only package P for dependency key K in repository R. This metric measures the percentage of exactly matched instances in $Dataset_{em}$ under a given approach.
- **Total Match Rate:** An instance $I = (K, R, P)$ is considered *partially matched* if the approach outputs multiple packages, including P for dependency key K in repository R. This metric measures the percentage of both exactly and partially matched instances.

To evaluate RosdepAuditor's effectiveness in diagnosing defects, we designed the following metrics:

- **Precision:** The proportion of RosdepAuditor's reported defects that are true defects (either included in $Dataset_{df}$ or confirmed through additional manual inspection).
- **Recall:** The proportion of true defects in $Dataset_{df}$ that are successfully reported.

For comparison with RosdepAuditor, we chose the following baselines:

- **RosdepCI (pattern-based):** The current industry standard used in `rosdistro` CI/CD, which relies primarily on regex-based string matching against package names.
- **Repology (upstream-based):** A comprehensive aggregation service that maps packages by identifying shared upstream projects [34].
- **LLMs:** We adapted and prompted LLMs to predict target package names directly using the metadata of the reference package. We utilized four models that ranked highly on the Chatbot Arena Leaderboard [3] as of January 1, 2026: two proprietary models (Gemini-3-Pro [17] and Claude-Opus-4.5 [2]) and two open-source models (GLM-4.7 [73] and DeepSeek-V3.2 [10]).

Table 2. Performance Comparison of RosdepAuditor's Equivalence Mapping against Baselines (854 instances)

Approach	Exact Match	Partial Match	Exact Match Rate	Total Match Rate	Extra (Non-existent)
RosdepAuditor (Ours)	808	1	**94.6%**	**94.7%**	**334 (0, 0.0%)**
RosdepCI	577	0	67.6%	67.6%	129 (0, 0.0%)
Repology	173	522	20.3%	81.4%	109,957 (0, 0.0%)
LLM (Gemini-3-Pro)	785	4	91.9%	92.4%	362 (138, 38.1%)
LLM (GLM-4.7)	766	0	89.7%	89.7%	355 (150, 42.3%)
LLM (Claude-Opus-4.5)	702	0	82.2%	82.2%	282 (70, 24.8%)
LLM (DeepSeek-V3.2)	700	0	82.0%	82.0%	460 (257, 55.9%)

5.1.3 Process. For RQ4, we evaluated RosdepAuditor's core mechanism for inferring equivalent-package mappings, which are critical to maintaining the central index and identifying its defects. We compared RosdepAuditor with the existing approaches by measuring the exact and total match rates on $Dataset_{em}$, using its dependency keys and target repositories as inputs. For RQ5, we evaluated RosdepAuditor's effectiveness in diagnosing practical defects reported in $Dataset_{df}$. Specifically, for each defect, we extracted the corresponding dependency entry from its PR and ran RosdepAuditor using a version of the central index reverted to its state immediately before the corresponding fix commit to determine whether it could report the defect. We measured how many defects RosdepAuditor could report. For RQ6, we used RosdepAuditor to assess the state of the official `rosdistro` central index as of August 2025, at commit `592178c`, and to validate reported defects with developer feedback.

5.2 RQ4: Equivalence Mapping Effectiveness

This section compares RosdepAuditor's equivalence-mapping performance with that of the baselines.

Performance Comparisons in Equivalence Mappings. Table 2 presents the comparative results of all evaluated approaches. We observe that RosdepAuditor achieved the highest overall performance, with exact and total match rates of 94.6% and 94.7%, respectively. RosdepAuditor significantly outperformed the traditional approaches: RosdepCI (both rates 67.6%) and Repology (20.3% exact, 81.4% total). RosdepAuditor also demonstrates clear superiority over state-of-the-art LLMs, improving the total match rate by 2.3–12.7 percentage points and the exact match rate by 2.7–12.6 percentage points. Notably, although some LLMs, such as Gemini-3-Pro and GLM-4.7, achieve comparable matching rates, they frequently suggest packages that do not exist in the target repositories. Note that an unmatched case where the mapping did not fully or partially align with the ground-truth dataset $Dataset_{em}$ does not necessarily indicate an incorrect mapping; instead, it may reflect a discrepancy with the established ground truth, leaving actual correctness uncertain. To quantify this, we conservatively verified the existence of each reported package in unmatched cases. For example, among unmatched mapping instances, Gemini-3-Pro produced non-existent packages in approximately 38.1% of cases, primarily due to LLM hallucination. Similar issues were observed in other models, i.e., GLM-4.7 with 42.3% non-existent packages, Claude-Opus-4.5 with 24.8%, and DeepSeek-V3.2 with 55.9%. In contrast, RosdepAuditor produced zero non-existent packages, ensuring both high accuracy and functional reliability in dependency mapping.

Performance Influenced by Internal Factor w. Since RosdepAuditor adopts an internal weighting parameter w, which is used to infer equivalence mappings through its scoring mechanism, we also performed a sensitivity analysis to assess how its performance changes under different values of w. We varied w from 0.0 to 1.0 in steps of 0.1 and measured the exact and total match rates on the ground-truth dataset. The results are summarized in Figure 3. As shown, RosdepAuditor's performance improves as w increases from 0.0 (87.0%), reaching peak exact and total match rates

Table 3. Defect Diagnosis Results on Dataset $Dataset_{df}$ (83 defects)

Defect Type	Reported	Ground Truth	Recall	Precision
A.1 Missing Dependency	48	48	100.0% (48/48)	100.0% (48/48)
A.2 Incomplete Platform	26	16	100.0% (16/16)	100.0% (26/26)
B.1 Invalid Package Specification	16	11	90.9% (10/11)	93.8% (15/16)
B.2 Suboptimal Prioritization	10	8	100.0% (8/8)	100.0% (10/10)
Total	**100**	**83**	**98.8% (82/83)**	**99.0% (99/100)**

of 94.6% and 94.7%, respectively, at $w = 0.7$. This indicates that while primary structural features provide the most reliable signals for equivalence, incorporating secondary metadata is necessary to resolve ambiguities. However, when w increases beyond 0.7, the performance begins to degrade, dropping to 90.0% for the exact match rate at $w = 1.0$, indicating that ignoring semantic metadata (like package descriptions) entirely leads to a loss in precision for edge cases. These findings support the design rationale for RosdepAuditor's weighting scheme, which balances different metadata dimensions.

Answer to RQ4: *RosdepAuditor demonstrates strong performance for its core equivalence-mapping mechanism, achieving a total match rate of 94.7% without any non-existent package mappings and outperforming all the studied baselines.*

5.3 RQ5: RosdepAuditor Effectiveness

This section evaluates the effectiveness of RosdepAuditor in identifying real defects in the central index using the ground-truth dataset $Dataset_{df}$.

Table 3 summarizes the results. Of the 83 defects included in $Dataset_{df}$, RosdepAuditor successfully detected 82, achieving a 98.8% recall rate. Of the 100 defects reported by RosdepAuditor (including 18 defects not included in $Dataset_{df}$), we manually confirmed 99 as correct, resulting in a 99.0% precision rate. Moreover, with respect to Type A.1, A.2, and B.2 defects, RosdepAuditor demonstrated both 100% recall and precision. The only false positive and false negative cases occurred within Type B.1. Regarding the false positive, RosdepAuditor incorrectly identified the mapping from `lua5.1-0-dev` on Ubuntu to `lua` on Arch as defective because the version numbers in their names did not match, and instead considered `lua51` a better alternative. However, in this case, `lua` is the expected equivalent package. Regarding the false negative, RosdepAuditor missed a defect in $Dataset_{df}$ related to PR #47703 [46], which adds `libglib2.0-dev-bin` to the existing `libglib2.0-dev` entry on Ubuntu to make it more functionally comprehensive. RosdepAuditor failed to flag the omission of `libglib2.0-dev-bin` because it already identified `libglib2.0-dev` as the most likely equivalent package for Ubuntu owing to its higher priority ranking. Moreover, the maintainers noted that such cases could be handled by creating new dependency entries rather than by modifying existing ones [45].

Answer to RQ5: *Although RosdepAuditor might wrongly identify or miss mappings due to its conservative scoring mechanism, it achieves strong defect diagnosis performance, with 98.8% recall and 99.0% precision.*

5.4 RQ6: RosdepAuditor Usefulness

This section applies RosdepAuditor to diagnose defects in the official `rosdistro` central index.

Defect Diagnosis Results. We conducted a comprehensive scan over the full set of entries in the current central index using RosdepAuditor, which revealed a substantial number of latent defects. Specifically, RosdepAuditor identified 3,249 potential defects in total, associated with 2,233 affected entries, i.e., entries with at least one reported defect, covering 88.9% of the 2,513 entries in the total index. Table 4 details the distribution of these potential defects. The majority of

Table 4. Potential Defects Detected in the Central Index

Defect Type	Count
A.1 Missing Dependency Definition	-
A.2 Incomplete Platform Coverage	1,820
B.1 Invalid Package Specification	1,139
B.2 Suboptimal Repository Prioritization	290
Total Potential Defects	3,249
Entries Affected	2,233

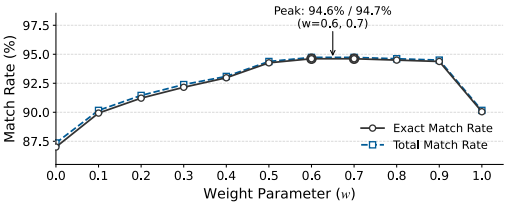


Fig. 3. Exact/Total Match Rates Under Varied w.

detected issues (1,820 cases) were Type A.2, in which an installation rule existed for some platforms, such as Ubuntu, but was missing for others, such as Fedora. The second most common issue was Type B.1, with 1,139 cases; in 818 of these cases, the referenced package did not exist in the target repository. Type A.1 defects are not reported because the scan covers only existing entries in the index and therefore cannot detect dependency keys that are entirely absent from the index.

To further demonstrate RosdepAuditor's usefulness, we actively engaged with the ROS community in two ways: first, by submitting a subset of the defects reported by RosdepAuditor, and second, by monitoring community activity to provide corroborating evidence of RosdepAuditor's capability.

Developer Feedback. Given that identifying and reporting all detected defects in a single repository would have imposed an untenable review workload on `rosdistro` maintainers, we prioritized a representative subset of 46 high-impact defects covering all detected defect types for verification. The first group concerns the recent 64-bit transition of time_t in Debian Trixie and Ubuntu Noble, which required widespread renaming of downstream libraries to preserve ABI compatibility. However, as the central index was not updated promptly to reflect this evolution, RosdepAuditor detected a cluster of index defects, with 5 Type A.2 defects and 25 Type B.1 defects reported [41]. The second group concerns the expansion of OS repositories to include packages that were previously exclusive to repositories such as PyPI, which can introduce cross-repository priority conflicts that violate the policy of preferring native OS packages. RosdepAuditor identified and reported 16 Type B.2 defects in which the central index prioritized ROS packages despite the availability of native system alternatives [42]. All 46 defects we reported were promptly verified and subsequently fixed [47, 49].

Prospective Validation. To validate the remaining findings without adding to the maintenance burden, we conducted a prospective study by monitoring independently submitted community PRs between September 2025 and January 2026. We identified 33 relevant PRs involving 35 defects—21 Type A.2 and 14 Type B.1—of which 32 (91.4%) were exactly identified by RosdepAuditor. In 21 of these PRs, developers later applied exactly the packages inferred by RosdepAuditor. We highlight the case of `python3-bson` [48], which has a misleadingly similar name to the incompatible `bson` package. The latter was originally specified in the central index, causing build failures in the downstream project `rosbridge-server` [30]. Although the issue was eventually reported and fixed by the developers, it could have been detected proactively by RosdepAuditor, which correctly inferred `python3Packages.pymongo` as the equivalent package.

Answer to RQ6: *RosdepAuditor detected 3,249 potential defects in the* `rosdistro` *central index. All 46 reported defects were confirmed and fixed, and the tool identified 91.4% of prospectively monitored defects. These results demonstrate the practical utility of RosdepAuditor in maintaining the central index.*

6 Threats to Validity

A validity threat stems from inconsistencies across repositories in metadata availability and naming conventions, which prevent exhaustive coverage of all variations. For example, ecosystems like Alpine offer limited metadata (e.g., no repository-level filelist index), while others follow entirely different packaging models. To address this, we integrate heuristics based on a broad survey of supported repositories and aggregate all available metadata. Because RosdepAuditor uses a multi-dimensional approach rather than exact comparisons of individual metadata fields, we believe that these threats can be mitigated. RosdepAuditor indeed showed satisfactory performance. Moreover, differences in packaging granularity further complicate identification, and the reference selection used by RosdepAuditor may influence outcomes. To mitigate this, we follow ROS's recommended repository ordering and, where possible, use Ubuntu as the reference.

Threats to internal validity may also arise from manual analysis and dataset construction. Our empirical study involves manual work, which carries a risk of subjectivity. To reduce this threat, we grounded the taxonomy in concrete maintenance artifacts rather than defining it a priori. After collecting candidate issues and pull requests, we extracted changes related to `rosdep` mapping rules and linked each case to its corresponding issue or PR discussion and CI feedback. Two co-authors independently conducted open coding over these artifacts, grouping cases by the observed failure mode of the central index. Disagreements were reviewed by a third co-author and resolved through discussion until consensus was reached. All involved authors have over three years of experience in open-source dependency-management research. While the resulting taxonomy focuses on mapping-rule maintenance defects observed in the current dataset, additional categories may emerge as the ecosystem evolves. Furthermore, bias in ground-truth labels could affect the evaluation. To mitigate this for RQ4, we selected only recently modified PRs that had been audited by official ROS reviewers and filtered out non-installable packages. For RQ5, a potential threat is data leakage, as the scarcity of Type B.2 defects necessitated supplementing the test set with a small number of cases from our empirical study. We minimized this threat by ensuring that the vast majority of the testing dataset remained independent of the empirical observation data, thereby preserving the fairness of the evaluation.

7 Related Work

Software Dependency Management. Managing software dependencies remains a challenging task. While significant research has focused on problems such as dependency bloat and missing dependencies in ecosystems like Java and Python [4, 11, 54–56, 67, 69, 72], similar issues persist in ROS [16, 59]. Previous studies [16, 59] have shown that robotic systems often encounter bugs due to missing dependencies, yet they generally assume a correctly functioning underlying infrastructure. We challenge this assumption by investigating the heterogeneity of ecosystem-level dependency management, demonstrating that the package management infrastructure itself is often the root cause of errors. Software Composition Analysis (SCA) is frequently adopted to identify third-party libraries by calculating artifact similarity across source and binary formats [12, 19, 20, 23, 26, 60, 74]. Existing approaches [58], however, mainly focus on single-language environments and functional code, failing to address the complexity of the ROS ecosystem. The ROS ecosystem is heterogeneous and multi-language, spanning source and binary packages whose contents may also include non-code artifacts such as images. As a result, current SCA tools are unable to effectively audit the cross-repository mappings maintained by the central index.

In dependency conflict resolution, most existing methods focus on single ecosystems like JavaScript, Java, and Python [25, 29, 32, 61, 63–66]. While Xie et al.'s Hera [70] advances this by exploring cross-repository compatibility between Python and Ubuntu, it remains focused on

resolving user-facing incompatibility issues during package installation. Recent robotics-specific efforts, such as RoboStack [14] and Pixi-Ros [15], further improve reproducibility by moving ROS software toward a unified package-management stack, but this essentially requires a substantial shift away from the existing `rosdep`/`rosdistro` framework and thus demands migration effort from projects and users. Our work addresses a different scope: instead of replacing the current framework, we study the defects that arise within it. The ROS ecosystem's integration of three heterogeneous repository types (ROS, OS, and PL) creates systemic inconsistencies, including divergent naming conventions and packaging policies, and we focus on auditing the existing central index to eliminate such defects at the source before they propagate to downstream users.

Software Engineering in ROS. The growing importance of robotics software has spurred significant research [1] into the ROS ecosystem. Studies have adapted software engineering techniques, such as fuzzing and runtime analysis, to address the unique challenges of robotic systems [21, 53, 68]. Others have created benchmarks of real-world bugs to establish baselines for evaluation [59]. While such work primarily targets code logic and runtime behavior, our research shifts focus to the foundational dependency infrastructure necessary for building and deploying these applications. Prior work has extensively examined how developers configure and use ROS. Researchers have analyzed common misconfigurations [5, 6], documented barriers like fragmented documentation [8], and studied inter-package relationships [16, 33] from the developer's perspective. Our work complements these studies by investigating the central index, a shared resource that governs dependency resolution for all. At the macro level, the ROS community's structure and evolution have been extensively analyzed. Studies have mapped package reuse and collaborative dynamics [13, 22, 33], while others have developed tools to facilitate ecosystem traversal, such as Knowledge Graph-based search engines [62, 75] and architectural guideline synthesizers [27]. Crucially, these prior works treat the ecosystem's infrastructure as a static environment to be navigated. In contrast, our work interrogates the infrastructure itself. We systematically audit the consistency of the underlying cross-repository linkages that sustain the ROS distribution network.

8 Conclusion

This paper addresses the critical challenge of dependency management in the fragmented ROS ecosystem. We make three contributions: (1) the first comprehensive study of the ROS central index, analyzing 863 real-world cases to identify practical defect types and key maintenance bottlenecks; (2) an automated auditing framework with cross-repository mapping and hybrid scoring for high-quality package equivalence mapping and defect diagnosis; and (3) RosdepAuditor, an effective tool that outperforms existing baselines and has uncovered thousands of potential defects, dozens of which have been verified and fixed. Looking ahead, we also recognize that LLMs' reasoning capabilities and knowledge bases could help interpret ambiguous corner cases that sometimes challenge our current approach. We will explore how to integrate LLMs into RosdepAuditor while mitigating their potential hallucination risks to further improve ROS dependency management.

Data-Availability Statement

All tools and data supporting this work are publicly available in the accompanying research artifact [57]. The artifact includes the implementation of RosdepAuditor, the datasets used in our empirical study and evaluation, and June 2026 snapshots of the package repositories examined in this work, which document the repository state at the time of the artifact release.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (Grant Nos. 92582201, 62522209, and 62302209), the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of

the Ministry of Education of China (No. JYB2025XDXM118), and the "111 Center" (No. B26023). The authors gratefully acknowledge the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. The authors thank the anonymous reviewers for their constructive feedback and Dr. Yanyan Jiang for the helpful guidance.

References

[1] Michel Albonico, Milica Đorđević, Engel Hamer, and Ivano Malavolta. 2023. Software Engineering Research on the Robot Operating System: A Systematic Mapping Study. *Journal of Systems and Software* 197 (March 2023), 111574. doi:10.1016/j.jss.2022.111574

[2] Anthropic. 2026. Claude Opus 4.5. https://www.anthropic.com/news/claude-opus-4-5.

[3] arena.ai. 2026. Arena Leaderboard | Compare & Benchmark the Best Frontier AI Models. https://arena.ai/zh/leaderboard.

[4] Cor-Paul Bezemer, Shane McIntosh, Bram Adams, Daniel M. German, and Ahmed E. Hassan. 2017. An Empirical Study of Unspecified Dependencies in Make-Based Build Systems. *Empirical Software Engineering* 22, 6 (Dec. 2017), 3117–3148. doi:10.1007/s10664-017-9510-8

[5] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. 2024. Understanding Misconfigurations in ROS: An Empirical Study and Current Approaches. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna Austria, 1161–1173. doi:10.1145/3650212.3680350

[6] Paulo Canelas, Bradley Schmerl, Alcides Fonseca, and Christopher S. Timperley. 2025. ROSpec: A Domain-Specific Language for ROS-Based Robot Software. *Proceedings of the ACM on Programming Languages* 9, OOPSLA2 (Oct. 2025), 3313–3341. doi:10.1145/3763169

[7] cottsay. 2026. rosdep_repo_check. https://github.com/ros/rosdistro/tree/master/test/rosdep_repo_check.

[8] Paulius Daubaris, Simo Linkola, Anna Kantosalo, and Niko Mäkitalo. 2023. Getting Started with ROS2 Development: A Case Study of Software Development Challenges. In *2023 IEEE/ACM 5th International Workshop on Robotics Software Engineering (RoSE)*. IEEE, Melbourne, Australia, 37–44. doi:10.1109/RoSE59155.2023.00011

[9] Debian Wiki. 2025. ReleaseGoals/64bit-Time - Debian Wiki. https://wiki.debian.org/ReleaseGoals/64bit-time.

[10] DeepSeek. 2026. DeepSeek-V3.2. https://api-docs.deepseek.com/news/news251201.

[11] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2584–2607. doi:10.1145/3660821

[12] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-Day Security Risk at Large Scale. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17)*. Association for Computing Machinery, New York, NY, USA, 2169–2185. doi:10.1145/3133956.3134048

[13] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. 2019. The Robot Operating System: Package Reuse and Community Dynamics. *J. Syst. Softw.* 151, C (May 2019), 226–242. doi:10.1016/j.jss.2019.02.024

[14] Tobias Fischer, Wolf Vollprecht, Silvio Traversaro, Sean Yen, Carlos Herrero, and Michael Milford. 2022. A RoboStack Tutorial: Using the Robot Operating System Alongside the Conda and Jupyter Data Science Ecosystems. *IEEE Robotics & Automation Magazine* 29, 2 (June 2022), 65–74. doi:10.1109/MRA.2021.3128367

[15] Tobias Fischer, Wolf Vollprecht, Bas Zalmstra, Ruben Arts, Tim de Jager, Alejandro Fontan, Adam D. Hines, Michael Milford, Silvio Traversaro, Daniel Claes, and Scarlett Raine. 2025. Pixi: Unified Software Development and Distribution for Robotics and AI. arXiv:2511.04827 [cs] doi:10.48550/arXiv.2511.04827

[16] Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wąsowski. 2020. The Forgotten Case of the Dependency Bugs: On the Example of the Robot Operating System. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. ACM, Seoul South Korea, 21–30. doi:10.1145/3377813.3381364

[17] Google. 2026. Gemini 3 Pro. https://deepmind.google/models/gemini/pro/.

[18] Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *J. ACM* 24, 4 (Oct. 1977), 664–675. doi:10.1145/322033.322044

[19] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*. Association for Computing Machinery, New York, NY, USA, 1–13. doi:10.1145/3597503.3639100

[20] Jong-Wouk Kim and Mi-Jung Choi. 2025. A Cross-Language and Cross-Binary Type Approach to Binary-Source Software Composition Analysis Using BM25. *International Journal of Information Security* 24, 6 (Nov. 2025), 231. doi:10.1007/s10207-025-01148-3

[21] Seulbae Kim and Taesoo Kim. 2022. RoboFuzz: Fuzzing Robotic Systems over Robot Operating System (ROS) for Finding Correctness Bugs. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Singapore Singapore, 447–458. doi:10.1145/3540250.3549164

[22] Sophia Kolak, Afsoon Afzal, Claire Le Goues, Michael Hilton, and Christopher Steven Timperley. 2020. It Takes a Village to Build a Robot: An Empirical Study of The ROS Ecosystem. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Adelaide, Australia, 430–440. doi:10.1109/ICSME46990.2020.00048

[23] Siyuan Li, Yongpan Wang, Chaopeng Dong, Shouguo Yang, Hong Li, Hao Sun, Zhe Lang, Zuxin Chen, Weijie Wang, Hongsong Zhu, and Limin Sun. 2023. LibAM: An Area Matching Framework for Detecting Third-Party Libraries in Binaries. *ACM Trans. Softw. Eng. Methodol.* 33, 2 (Dec. 2023), 52:1–52:35. doi:10.1145/3625294

[24] Zhong Li, Minxue Pan, Yu Pei, Tian Zhang, Linzhang Wang, and Xuandong Li. 2024. Empirically revisiting and enhancing automatic classification of bug and non-bug issues. *Frontiers of Computer Science* 18, 5 (2024), 185207. doi:10.1007/s11704-023-2771-z

[25] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: Escape from NuGet Dependency Maze. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh, PA, USA, 1545–1557. doi:10.1145/3510003.3510118

[26] Cristina V. Lopes, Petr Maj, Pedro Martins, Vaibhav Saini, Di Yang, Jakub Zitny, Hitesh Sajnani, and Jan Vitek. 2017. DéjàVu: A Map of Code Duplicates on GitHub. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017), 1–28. doi:10.1145/3133908

[27] Ivano Malavolta, Grace A. Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 2021. Mining Guidelines for Architecting Robotics Software. *Journal of Systems and Software* 178 (Aug. 2021), 110969. doi:10.1016/j.jss.2021.110969

[28] moveit2. 2026. Issue #2862 of moveit/moveit2. https://github.com/moveit/moveit2/issues/2862.

[29] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, Denmark) *(ISSTA 2021)*. Association for Computing Machinery, New York, NY, USA, 439–451. doi:10.1145/3460319.3464797

[30] nix-ros-overlay. 2026. Issue #728 of lopsided98/nix-ros-overlay. https://github.com/lopsided98/nix-ros-overlay/issues/728.

[31] Open Robotics Zulip. 2026. Massive Software-Driven Rosdep Update. https://openrobotics.zulipchat.com/#narrow/channel/526027-ROS-General/topic/Massive.20software-driven.20rosdep.20update.3F/with/564350407.

[32] 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*. ACM, Gothenburg, Sweden, 741–751. doi:10.1145/3180155.3180184

[33] Marc Pichler, Bernhard Dieber, and Martin Pinzger. 2019. Can I Depend on You? Mapping the Dependency and Quality Landscape of ROS Packages. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. IEEE, Naples, Italy, 78–85. doi:10.1109/IRC.2019.00020

[34] Repology. 2026. Repology. https://repology.org/.

[35] rosdistro. 2026. Issue #32002 of ros/rosdistro. https://github.com/ros/rosdistro/issues/32002.

[36] rosdistro. 2026. Issue #34332 of ros/rosdistro. https://github.com/ros/rosdistro/issues/34332.

[37] rosdistro. 2026. Issue #36442 of ros/rosdistro. https://github.com/ros/rosdistro/issues/36442.

[38] rosdistro. 2026. Issue #41622 of ros/rosdistro. https://github.com/ros/rosdistro/issues/41622.

[39] rosdistro. 2026. Issue #42698 of ros/rosdistro. https://github.com/ros/rosdistro/issues/42698.

[40] rosdistro. 2026. Issue #44826 of ros/rosdistro. https://github.com/ros/rosdistro/issues/44826.

[41] rosdistro. 2026. Issue #47577 of ros/rosdistro. https://github.com/ros/rosdistro/issues/47577.

[42] rosdistro. 2026. Issue #48237 of ros/rosdistro. https://github.com/ros/rosdistro/issues/48237.

[43] rosdistro. 2026. Issue #49424 of ros/rosdistro. https://github.com/ros/rosdistro/issues/49424.

[44] rosdistro. 2026. PR #38540 of ros/rosdistro. https://github.com/ros/rosdistro/pull/38540.

[45] rosdistro. 2026. PR #41413 of ros/rosdistro. https://github.com/ros/rosdistro/pull/41413.

[46] rosdistro. 2026. PR #47703 of ros/rosdistro. https://github.com/ros/rosdistro/pull/47703.

[47] rosdistro. 2026. PR #48183 of ros/rosdistro. https://github.com/ros/rosdistro/pull/48183.

[48] rosdistro. 2026. PR #48361 of ros/rosdistro. https://github.com/ros/rosdistro/pull/48361.

[49] rosdistro. 2026. PR #48659 of ros/rosdistro. https://github.com/ros/rosdistro/pull/48659.

[50] rosdistro. 2026. ros/rosdistro. https://github.com/ros/rosdistro.

[51] rosdistro contributors. 2026. Guidelines for Rosdep Rules. https://github.com/ros/rosdistro/blob/master/CONTRIBUTING.md#guidelines-for-rosdep-rules. Accessed 2026.

[52] rosdistro-reviewer. 2026. Issue #30 of ros-infrastructure/rosdistro-reviewer. https://github.com/ros-infrastructure/rosdistro-reviewer/issues/30.

[53] Yuheng Shen, Jianzhong Liu, Yiru Xu, Hao Sun, Mingzhe Wang, Nan Guan, Heyuan Shi, and Yu Jiang. 2024. Enhancing ROS System Fuzzing through Callback Tracing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Vienna Austria, 76–87. doi:10.1145/3650212.3652111

[54] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A Longitudinal Analysis of Bloated Java Dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Athens, Greece, 1021–1031. doi:10.1145/3468264.3468589

[55] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *Empirical Software Engineering* 26, 3 (March 2021), 45. doi:10.1007/s10664-020-09914-8

[56] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. 2023. Automatic Specialization of Third-Party Java Dependencies. *IEEE Transactions on Software Engineering* 49, 11 (November 2023), 5027–5045. doi:10.1109/TSE.2023.3324950

[57] Weijie Sun, Huiyan Wang, Ying Wang, and Chang Xu. 2026. Replication Package for Article 'Guarding the Lifeline: A First Look and Automated Defect Diagnosis for ROS Central Index'. Zenodo. doi:10.5281/zenodo.21280924

[58] Wei Tang, Ping Luo, Jialiang Fu, and Dan Zhang. 2020. LibDX: A Cross-Platform and Accurate System to Detect Third-Party Libraries in Binary Code. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, London, ON, Canada, 104–115. doi:10.1109/SANER48275.2020.9054845

[59] Christopher S. Timperley, Gijs van der Hoorn, André Santos, Harshavardhan Deshpande, and Andrzej Wąsowski. 2024. ROBUST: 221 Bugs in the Robot Operating System. *Empirical Software Engineering* 29, 3 (March 2024), 57. doi:10.1007/s10664-024-10440-0

[60] Hao Wang, Zeyu Gao, Chao Zhang, Mingyang Sun, Yuchen Zhou, Han Qiu, and Xi Xiao. 2024. CEBin: A Cost-Effective Framework for Large-Scale Binary Code Similarity Detection. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2024)*. Association for Computing Machinery, New York, NY, USA, 149–161. doi:10.1145/3650212.3652117

[61] Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco, CA, USA, 198–210. doi:10.1145/3611643.3616264

[62] Shuo Wang, Xinjun Mao, Shuo Yang, Menghan Wu, and Zhang Zhang. 2024. ROS Package Search for Robot Software Development: A Knowledge Graph-Based Approach. *Frontiers of Computer Science* 19, 6 (Dec. 2024), 196320. doi:10.1007/s11704-024-3660-9

[63] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: Monitoring Dependency Conflicts for Python Library Ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 125–135. doi:10.1145/3377811.3380426

[64] 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 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, Lake Buena Vista, FL, USA, 319–330. doi:10.1145/3236024.3236056

[65] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing Chi Cheung. 2019. Could I Have a Stack Trace to Examine the Dependency Conflict Issue?. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*. IEEE, Montréal, QC, Canada, 572–583. doi:10.1109/ICSE.2019.00068

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

[67] Nimmi Rashinika Weeraddana, Mahmoud Alfadel, and Shane McIntosh. 2024. Dependency-Induced Waste in Continuous Integration: An Empirical Study of Unused Dependencies in the Npm Ecosystem. *Proceedings of the ACM on Software Engineering* 1, FSE (July 2024), 2632–2655. doi:10.1145/3660823

[68] Trey Woodlief, Sebastian Elbaum, and Kevin Sullivan. 2021. Fuzzing Mobile Robot Environments for Fast Automated Crash Detection. In *2021 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE Press, Xi'an, China, 5417–5423. doi:10.1109/ICRA48506.2021.9561627

[69] Rongxin Wu, Zhiling Huang, Zige Tian, Chengpeng Wang, and Xiangyu Zhang. 2025. PackHunter: Recovering Missing Packages for C/C++ Projects. *IEEE Transactions on Software Engineering* 51, 1 (2025), 206–219. doi:10.1109/TSE.2024.3506629

[70] Yifan Xie, Zhouyang Jia, Shanshan Li, Ying Wang, Jun Ma, Xiaoling Li, Haoran Liu, Ying Fu, and Xiangke Liao. 2024. How to Pet a Two-Headed Snake? Solving Cross-Repository Compatibility Issues with Hera. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Sacramento CA USA, 694–705. doi:10.1145/3691620.3695064

[71] Sheng-Bin Xu, Si-Yu Chen, Yuan Yao, and Feng Xu. 2025. Detecting and Untangling Composite Commits via Attributed Graph Modeling. *Journal of Computer Science and Technology* 40, 1 (2025), 119–137. doi:10.1007/s11390-024-2943-9

[72] Zhengmin Yu, Yuan Zhang, Ming Wen, Yinan Nie, Wenhui Zhang, and Min Yang. 2025. CXXCrafter: An LLM-Based Agent for Automated C/C++ Open Source Software Building. *Proceedings of the ACM on Software Engineering* 2, FSE (June 2025), 2618–2640. doi:10.1145/3729386

[73] z.ai. 2026. GLM-4.7. https://z.ai/blog/glm-4.7.

[74] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. 2023. Software Composition Analysis for Vulnerability Detection: An Empirical Study on Java Projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, San Francisco, CA, USA, 960–972. doi:10.1145/3611643.3616299

[75] Yuxin Zhao, Xinjun Mao, and Yi Yang. 2023. An Effective Method for Constructing a Robot Operating System Node Knowledge Graph Based on Open-Source Robotics Repositories. *Electronics* 12, 19 (Jan. 2023), 4022. doi:10.3390/electronics12194022

[76] Yu-Qian Zhuang, Liang Wang, Ke-Xin Sun, Hong-Yu Kuang, and Xian-Ping Tao. 2026. Understanding Users' Affective States During Issue Resolution in Open Source Software Projects. *Journal of Computer Science and Technology* 41, 2 (2026), 724–741. doi:10.1007/s11390-025-4478-0

Received 2026-01-30; accepted 2026-04-16