# How does ROS-based Robot Software Evolve? An Empirical Study on Architectural Evolution

Paulo Canelas[1,2][0000−0002−0154−8989],
Bradley Schmerl[2][0000−0001−7828−622X],
Alcides Fonseca[1][0000−0002−0879−4015], and
Christopher S. Timperley[2][0000−0002−9785−324X]

[1] Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
{pacsantos,amfonseca}@ciencias.ulisboa.pt
[2] Carnegie Mellon University, Pittsburgh, USA
{pasantos,schmerl,ctimperl}@andrew.cmu.edu

**Abstract.** The Robot Operating System (ROS) is the de facto framework for building open-source robotic software systems. Developers architect their system by using reusable components, connecting them via configuration files and source code. These architectures evolve as developers add, remove, and modify components, connections, and configurations. These changes can lead to architectural misconfigurations detectable by specification languages and analysis tools. However, these focus on snapshots in time without considering how architectures evolve. By understanding this evolution, we can help specification writers estimate the maintenance burden of their descriptions and promote tools toward the most change-prone elements. To that end, we conduct an empirical study of architectural evolution in open-source ROS systems. We develop a differential, cross-language architectural recovery tool that extracts components, connections, and configurations from source code and apply it to 294 repositories, with 1728 packages across 9914 releases, spanning ROS 1 and ROS 2 in C/C++, Python, and XML. As ROS repositories aggregate multiple packages at different maturity stages, we analyze evolution at the repository and package levels. We find that evolution diverges across levels, with repositories remaining addition-dominant while mature packages consolidate by removing connections and configurations. Furthermore, changes co-occur across elements: component removals are associated with the removal of their connections and configurations.

**Keywords:** Robot Operating System · Empirical Study · Architectural Evolution

## 1 Introduction

The Robot Operating System (ROS) is the de facto open-source framework for building robotic software systems [24,27]. Its ecosystem provides reusable components for common robot functions [21] (e.g., localization to determine the robot's position [26], and planning to decide how to accomplish tasks [7]).

Developers define their system's architecture via configuration files and source code API calls by specifying components, connections, and runtime configurations.

Nevertheless, ROS architectures evolve as developers add, remove, and modify these architectural elements. Even small code-level changes can reshape the system's architecture. This makes architectural evolution a key maintenance challenge, as undocumented changes may lead to unintended misconfigurations [6].

For instance, consider mgonzs13/yolo_ros (Figure 1), a package for object detection. Between releases, its maintainers refactored component names to remove version references (e.g., `yolov8_node`), added new connections, and removed deprecated ones, yet no documentation captured this evolution. Without tooling to detect these changes, developers integrating these components risk introducing silent architectural mismatches [6,17] when upgrading to a new version.

Architectural specification languages for ROS, such as AADL for ROS [3] and ROSpec [5], can detect these misconfigurations but require developers to manually describe the architecture. Measuring how much these elements change over time is essential to assess the maintenance burden of keeping such specifications. For instance, if component definitions change frequently, the cost of keeping their specifications updated may become a significant burden.

Furthermore, existing tools for recovering and analyzing ROS architectures, such as HAROS [29], ROSDiscover [33], and ROSInfer [12], recover and analyze the architecture at a single snapshot in time. Without understanding which elements change most frequently, these tools cannot prioritize change-prone elements nor anticipate co-occurring architectural modifications, potentially missing changes like those in yolo_ros that silently break downstream integrators. Despite this, no prior work has studied how ROS architectures evolve across packages [1].

In this work, we conduct an empirical study of architectural evolution in open-source ROS-based systems. We develop a differential, cross-language architectural recovery tool that extracts components, connections, and configurations from source code. We apply this tool to all 294 ROS repositories on GitHub with at least 10 tagged releases, totaling 1728 packages across 9914 releases, spanning ROS 1 and ROS 2 projects[3] in C/C++, Python, and XML. As ROS repositories often contain multiple packages, we analyze evolution at the repository and package levels. We study how architectural elements change over time, how changes co-occur, and how evolution differs across ROS versions and packages.

In this work, we make the following contributions:

- **A quantitative study of architectural evolution in ROS systems** (Section 4.2). We study the frequency of architectural changes over time, comparing evolution between ROS versions and types of packages;
- **A lightweight differential cross-version architectural recovery tool** (Section 3). We introduce a differential, cross-language tool that trades full architecture recovery for lightweight detection of architectural changes;
- **A dataset of extracted architectures and their evolution** (Section 4.1). We provide a dataset of 238 847 architectural changes and their evolution, allowing replication and extension in future work.

[3] ROS transitioned from ROS 1 to ROS 2 with architectural and API improvements.

## 2 Background and Motivating Example

In this section, we introduce ROS as a middleware for building robot software and illustrate how architectural elements evolve through a real-world example.

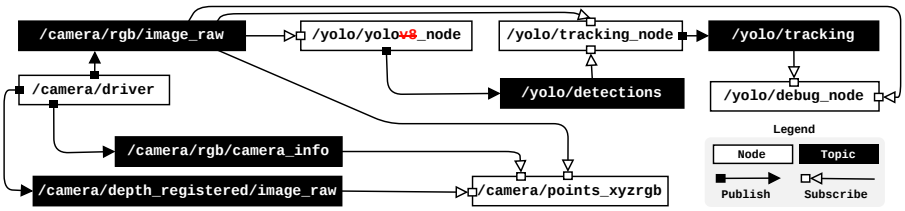


Figure 1: Component-and-Connector architectural view of the publisher-subscriber graph in mgonzs13/yolo_ros. Package refactoring between versions removed references to "v8" from node and topic names. Systems integrating newer versions of this package must ensure naming consistency to prevent misconfigurations.

ROS provides a rich ecosystem of reusable *packages* that allows developers to focus on configuration and integration rather than implementation from scratch [21,23]. A *distro package* is included in a versioned ROS distribution (e.g., Noetic, Humble), such as Nav2 [36], MoveIt! [10], and ROS Control [9], while a *community package* is maintained independently. Repositories often contain multiple packages, each serving a different role. We classify packages that provide reusable functionality (e.g., drivers) as *utility* packages, while *system* packages integrate multiple components into a deployable application using *launch* files (XML or Python) and parameter files (YAML) to configure their behavior. As configurations[4] are spread across source code and parameter files, developers must manually trace how components integrate into the overall architecture [14].

At its core, systems in ROS are built by integrating *components* (i.e., *nodes*), each responsible for a specific function (e.g., perception, planning, and control). ROS provides the communication infrastructure for message exchange between these distributed components. ROS components communicate primarily via anonymous publish-subscribe topics [30], with services and actions providing synchronous and asynchronous call-return communication, respectively.

Components publish messages to named *topics* (e.g., `/yolo/detections`), while other components subscribe to those same topics to receive messages [15]. Communications are defined at run time via ROS API calls, with strings specifying topic names dynamically. Components are also parameterized at launch through *parameters* and *remappings* that rebind topic and node names at startup, and allowing the same component to have different topic names across deployments. For instance, Figure 1 depicts an example of a publish-subscribe system in ROS for mgonzs13/yolo_ros. This package integrates YOLO-based object detection

[4] In this paper, we use configurations to refer to runtime parameterization.

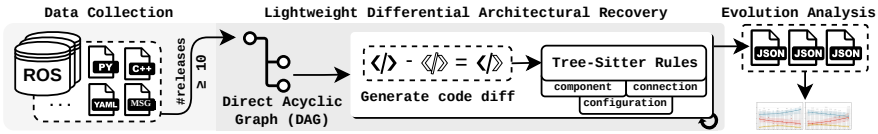


Figure 2: Overview of the methodology. First, we collect ROS repositories and filter them by release count. Then, we model their release history as a directed acyclic graph (DAG). For each release pair, we compute code diffs and use Tree-Sitter to recover architectural elements. Finally, we process and analyze the data.

where the `/camera/driver` node publishes camera data to multiple topics, which `/yolo/yolov8_node` receives for detection. As each topic is strongly typed, publishers and subscribers must agree on the message format to prevent mismatches [6].

We classify these architectural elements into three categories: *components* (nodes), *connections* (publishers, subscribers, services, actions), and *configurations* (e.g., parameters, arguments, QoS profiles). Each element contributes to the system's architecture, and any change corresponds to *architectural evolution*.

## 3 Architectural Evolution Study

It remains unclear which elements change most, whether changes propagate across architectural elements, and how evolution differs across the ecosystem. With this understanding, recovery tools could prioritize which elements to re-extract after a release, and specification writers could focus revision effort on the most change-prone correlated elements. To address these gaps, we pose three main research questions:

**RQ1** **To what extent do architectures evolve at the repository and package level?** Understanding how configurations, connections, and components evolve and how these patterns differ across scales can guide which elements recovery tools should prioritize and how frequently specifications need updating.

**RQ2** **How do architectural changes co-occur across architectural elements?** Architectural changes are rarely isolated, as adding a component often requires new connections and configurations. Understanding which changes co-occur can show if modifications propagate predictably, allowing analysis tools to warn developers about potentially incomplete modifications.

**RQ3** **How does architectural evolution differ across projects?** Understanding how evolution differs across versions, release status, and package types can inform migration and maintenance efforts, and determine to what extent tools developed for ROS 1 architectures transfer to ROS 2.

To answer our research questions, we conduct an empirical study of open-source ROS projects. Figure 2 depicts our three-step methodology: we collect and filter repositories, recover architectural elements from source code, and analyze their evolution. We describe each step as follows.

*Data Collection.* We use the ROS Anatomy dataset [2], a large-scale dataset of over 100,000 ROS open-source repositories from GitHub. To focus on projects with meaningful history, we filter for repositories with at least 10 releases. Fewer releases provide too few data points to distinguish evolution from initial prototyping. This filtering yields 294 repositories for analysis. Since each repository may contain multiple ROS packages in different languages and ROS versions, we collect architectural elements at the package level. We classify each package by ROS version, inspecting build files (`package.xml` or `manifest.xml`) for `catkin` (ROS 1) or `ament` (ROS 2) build systems. Despite ROS 1 reaching end-of-life in 2025, it remains in active use and provide the longest available evolution histories. We further classify each package by language (C/C++ or Python) based on the elements recovered, and as a distro or community package using ROS Anatomy metadata. We manually classified all 294 repositories by inspecting their documentation and source code into four categories: *system*, *utility*, *simulation*, or *other* (e.g., visualization tools). In total, we consider 1728 packages and 9914 releases.

*Constructing the Release Evolution Graph.* As repositories may maintain multiple parallel release branches (e.g., for different ROS distributions), we construct a directed acyclic graph (DAG) representing the parent-child relationships. By constructing the full DAG, we can analyze all evolution paths. For each edge in the DAG, we diff consecutive releases to identify all changed files.

*Differential Cross-Language Architectural Recovery.* To study how ROS architectures evolve, we need an architectural representation at each release. Prior architectural recovery tools [29,33] leverage source code analysis of API calls to recover the full architecture. However, each recovery requires resolving cross-package dependencies and often compilation, making extraction across thousands of releases impractical. Our key insight is that **studying architectural evolution does not require recovering the full architecture, but only detecting changes to architectural API calls**. This makes our approach lightweight and scalable across thousands of releases as it only detects API changes in diffs.

As ROS spans two major versions with distinct APIs, each supporting multiple distributions (e.g., ROS 1 Noetic, ROS 2 Humble), and most projects are written in C/C++ or Python [30], this creates four language-version combinations to support. To support this diversity, we use Tree-sitter [34], a parser generator that provides query rules to capture ROS API calls that build, configure, and connect components. We created 202 rules by systematically covering the ROS API for each language and version, validating each against real-world examples and tutorials. Each query rule is labeled by category, with component (node), configuration (arguments, parameters, environment variables), or connection (e.g., publishers and subscribers), and by change type, either addition, removal, or modification (e.g., a renamed connection). We classify changes by overlaying a line-level diff on the extracted elements: overlapping regions are modifications, while elements exclusive to the old or new version yield removals or additions.

Table 1: Distribution of the 1728 packages by language, ROS version, and status.

| Language | ROS 1 | | ROS 2 | | Both | | Neither | |
|---|---|---|---|---|---|---|---|---|
| | Distro | Comm. | Distro | Comm. | Distro | Comm. | Distro | Comm. |
| Python | 13 | 175 | 25 | 207 | 5 | 10 | 4 | 23 |
| C/C++ | 42 | 335 | 18 | 185 | 15 | 25 | 7 | 10 |
| Both | 12 | 54 | 8 | 115 | 25 | 49 | 2 | 9 |
| Messages | 6 | 91 | 11 | 82 | 26 | 11 | 1 | 2 |
| Launch | 10 | 98 | | 10 | 1 | 3 | | 3 |
| **Total** | **83** | **753** | **62** | **599** | **72** | **98** | **14** | **47** |

*Evolution Analysis.* For each release, we recover all architectural elements and their changes relative to the parent release(s), along with release metadata (e.g., date, authors). We use this data to answer our research questions.

For the lifetime analysis (RQ1), we normalize each project's lifetime (first to last tagged release) into 10 equal deciles to compare projects of different lengths, deduplicating repeated edges when parallel release branches exist. Since types of change are shares of a fixed total, we fit a multinomial logistic regression with a quadratic lifetime term to capture non-linear patterns such as early growth followed by late consolidation. We report early-vs-late effects as the modeled share difference between the first and last deciles, with 95% bootstrap confidence intervals [18] and Benjamini-Hochberg (BH) $q$-values [4].

For the association analysis (RQ2), we study how change types co-occur within package releases using two models: pairwise Pearson correlations on log-transformed counts across 9 event variables (3 categories $\times$ 3 change types), with clustered bootstrap inference and BH correction, and logistic regressions predicting each event while controlling for all others to isolate direct associations.

For the subgroup comparison (RQ3), we stratify the lifetime model by ROS version (ROS 1 vs. ROS 2) and release status (community vs. distro), and separately compare utility vs. system packages against the aggregated baseline.

## 4 Results

In this section, we provide an overview of the dataset, including package distribution and the number of architectural changes (Section 4.1), report our findings for each research question (Section 4.2), and threats to validity (Section 4.3).

### 4.1 Dataset Overview

Table 1 presents the distribution of 1728 packages in our dataset, classified by ROS version and release status for each programming language.

Table 2: Architectural changes by ROS version, and language for each sub-category. Each sub-category is composed of at least one detection rule.

| Sub-category | ROS 1 | | | ROS 2 | | | Both | | |
|---|---|---|---|---|---|---|---|---|---|
| | Python | C/C++ | Launch | Python | C/C++ | Launch | Python | C/C++ | Launch |
| argument | 22 | 7 | 102,251 | 10,472 | | 6,295 | 532 | | 4,659 |
| parameter | 2,716 | 7,789 | 17,795 | 1,470 | 20,204 | 1,720 | 56 | 726 | 4,445 |
| environment | | | 420 | | | 31 | | | 50 |
| publisher | 1,518 | 4,801 | | 705 | 4,195 | | 28 | 305 | |
| subscriber | 1,581 | 4,602 | 9 | 771 | 3,849 | | 45 | 252 | |
| service-client | 901 | 1,171 | | 373 | 441 | | 84 | 68 | |
| service-server | 864 | 1,645 | | 54 | 1,293 | | 23 | 101 | |
| action-client | 122 | 6 | | 196 | 212 | | 1 | | |
| action-server | 50 | 13 | | 35 | 84 | | | | |
| node | 1,071 | 4,437 | 2,456 | 7,502 | 3,376 | 380 | 612 | 218 | 63 |
| include | | | 6,014 | | | 328 | | | 332 |
| **Total** | **8,845** | **24,471** | **128,945** | **21,578** | **33,654** | **8,754** | **1,381** | **1,670** | **9,549** |

ROS 1 packages account for 836 (48%), ROS 2 for 661 (38%), and 170 (10%) for packages containing elements from *Both* versions, through dual build files (49/170) or version transitions (121/170). We exclude the remaining 61 packages with no ROS build file. Community packages represent the majority (1497), with 231 belonging to official ROS distributions, in line with prior work [2]. C/C++ is the most common language (637), followed by Python (462) [13,30].

Table 2 presents the 238 847 architectural changes detected across sub-categories, ROS versions, and languages. ROS 1 accounts for 68% of all changes, ROS 2 for 27%, and packages with *Both* versions for 5%, consistent with ROS 1's longer presence in the ecosystem (first stable release in 2010 vs. 2017 for ROS 2). The distribution across languages differs between versions: in ROS 1, launch files account for 79% of changes, while in ROS 2, source code accounts for 86%, shifting from XML to Python-based launch files.

### 4.2 Findings

We analyze evolution at the repository level, which shows how the overall system architecture grows as packages are added and integrated, and the package level, which captures how individual components mature over time. Figure 3 presents the evolution of change types across the lifetime at both levels.

***Repository level.*** Configurations show the strongest pattern, with additions increasing (+17.2%) and removals decreasing (−15.9%), indicating that repositories become increasingly configurable over time. Components remain additive throughout the repository lifetime. Connections shift from additions (-10.6%) toward modifications (+11.0%), suggesting that as repositories mature, existing communications are maintained rather than replaced by new ones.

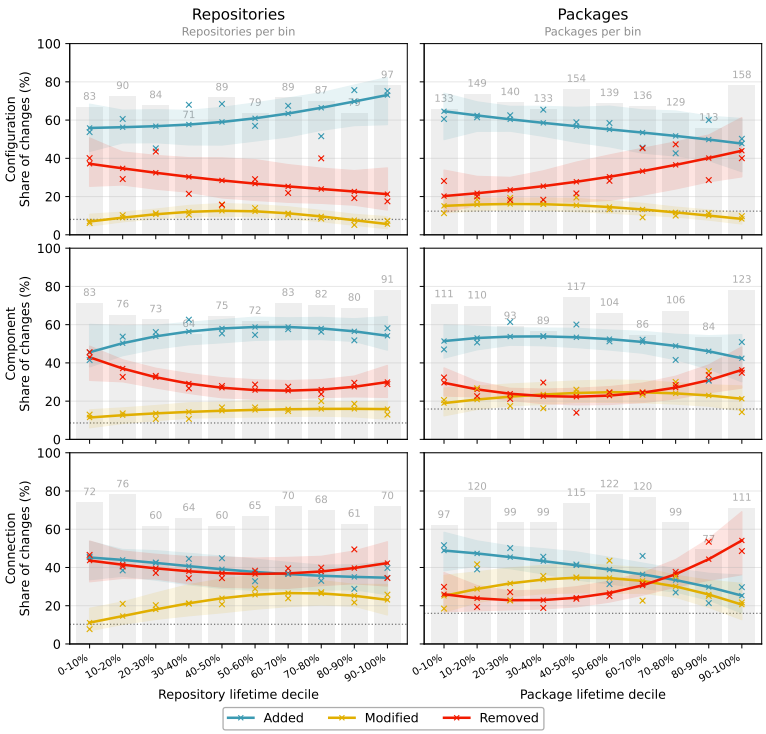


Figure 3: Evolution of additions, modifications, and removals for each architectural element, comparing repository and package-level lifetimes.

***Package level.*** In contrast, packages show a clear shift from additions to removals in later stages. Connections show the strongest package-level lifetime effect. Additions decrease ($-23.6\%$) while removals increase ($+28.1\%$) ($q = 0.02$). Configuration follows a similar pattern, with additions decreasing ($-16.9\%$) and removals increasing ($+23.7\%$) ($q = 0.05$). Components are more stable, also drifting toward a maintenance pattern. The package and repository-level results provide two scales of architectural evolution. The clearest contrast appears in configuration. Package-level configurations are removal-heavy in late stages, while repository-level configurations become more additive. This reveals that packages prune configurations, while repositories keep adding them. At the repository level, the architecture continues to grow as new packages are added. At the package level, individual packages consolidate by pruning connections and configurations.

**RQ1: Multi-level Analysis.**

Repositories become increasingly configurable as developers add new packages over their lifetime. In contrast, individual packages tend to mature and consolidate their architecture. This suggests architectural growth is driven by integrating new packages rather than expanding existing ones.

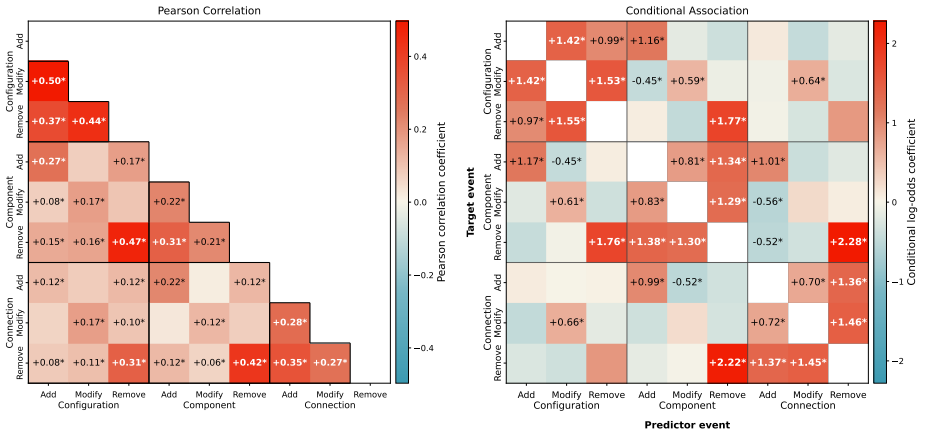


Figure 4: Association between architectural change types at the package level. Releases bundle changes across elements, like component removals with connection and configuration removals, while expansion and cleanup rarely overlap (29/36 pairwise and 35/72 conditional coefficients significant, $q < 0.05$).

Figure 4 presents the association between architectural change types at the package level. The pairwise Pearson correlations show that 29/36 pairs of change events are significant and positively correlated ($q < 0.05$). This shows that releases typically bundle architectural changes. The stronger links match the domain expectation: when a component is removed, its connections and configurations also tend to be removed, and when configurations are added, they are often changed in the same release. The conditional model controls for other change types and shows that component removals remain strongly associated with connection and configuration removals. However, adding new connections is negatively associated with removing or modifying components, suggesting that releases tend to either grow the architecture or consolidate it, but rarely both.

**RQ2: Co-occurrence of architectural changes.**

Within a release, developers tend to either add new architectural elements or remove existing ones, but seldom both, suggesting that expansion and cleanup are treated as separate tasks.

***ROS 1 vs. ROS 2.*** Figure 5 compares the modeled share of change types across ROS versions and release status. At the repository level, ROS 1 shows a stronger additive configuration pattern late in the lifetime (+18%). This is driven by large integration repositories (e.g., start-jsk/jsk_apc and dvrk-ros) that add launch and configuration files to their packages. On ROS 2 repositories, connections become more additive than the baseline, as newer repositories are still actively building system integrations while reworking their configurations.

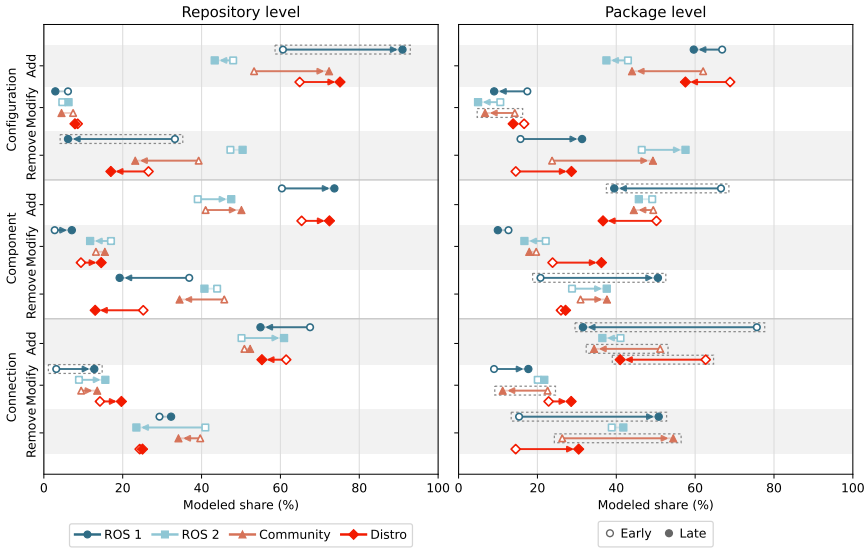


Figure 5: Share of change types for each category, comparing ROS versions and distro status at the repository and package level. Arrows indicate the direction of change from early to late lifetime. Dashed boxes highlight significant shifts.

At the package level, ROS 1 shows stronger late-life cleanup than the baseline, particularly for connections and components. Older ROS 1 packages (e.g., nasa/astrobee and grvc-ual) have accumulated more legacy topics, services, and nodes. In contrast, ROS 2 packages show less late connection teardown, as they have not yet accumulated the same share of communications. Moreover, ROS 2 configuration shifts toward removals, driven by system startup packages.

***Community vs. Distro.*** As community packages are more representative in our dataset, the baseline reflects community behavior. These include more experimental integration stacks and autonomy systems, which produce stronger expansion and cleanups as they accumulate and later prune communications.

Distro packages are more curated and maintenance-oriented as they are intended to be reused in many ROS projects. Their late-life behavior is often shaped by large vendors or infrastructure repositories (e.g., IntelRealSense/realsense-ros and SICKAG/sick_scan_xd), which tend to be hardware wrappers and drivers that evolve more incrementally. At the repository level, distro repositories are more additive for configuration late in their lifetime.

***Utility vs. System.*** To understand how late-life cleanup differs by package type, we also analyze system and utility repositories at the package level. Figure 6 presents the evolution of change types across lifetime deciles for each architectural element. We further divide components into *component definition*, source-level implementation, and *component use*, i.e., instantiation in launch files.

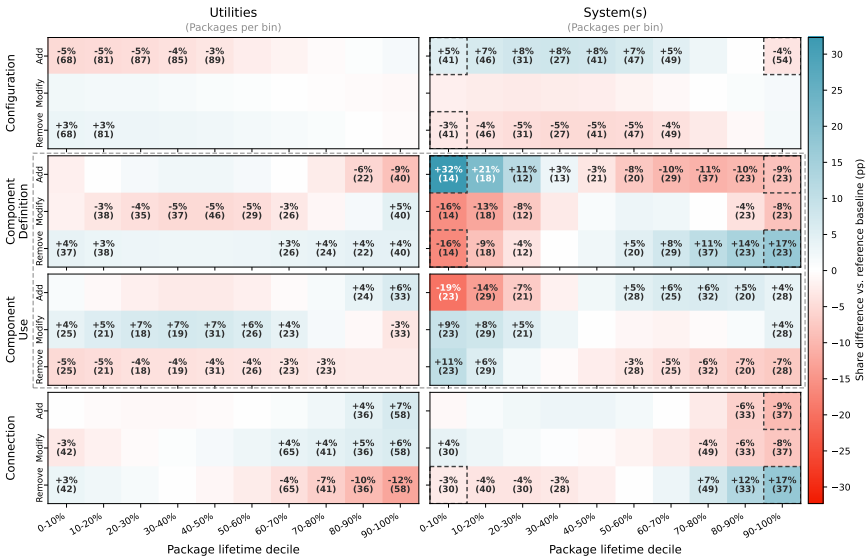


Figure 6: Evolution of additions, modifications, and removals across lifetime deciles for packages in system and utility repositories. Components are decomposed into node definitions (source-level implementation) and node uses (in launch files).

The strongest lifetime effects appear in system repositories at the component definition level. Component definition additions shift from strongly overrepresented early in the lifetime (+32.3%) to underrepresented by the final decile (−8.5%), while removals follow the inverse pattern, from underrepresented (−16.4%) to overrepresented (+17.0%) ($q = 0.048$ for both). Systems actively expand their component definitions early in their lifetime, while integrating other components into the architecture. As these mature, they gradually replace custom components with off-the-shelf reusable packages and prune existing connections.

In contrast, utility packages evolve more incrementally, as they may be required to maintain more stable, reusable interfaces. Component definition additions slightly decrease over time while removals increase, suggesting that these packages stabilize their implementations by pruning unused node definitions. However, component uses and connection additions grow, consistent with expanding adoption and broader interface support for downstream integrators.

**RQ3: Evolution across the ecosystems.**

ROS 1 packages show stronger late-life cleanup than ROS 2, as they accumulate more legacy interfaces. System component definitions transition from custom-built components early on to reusing off-the-shelf packages, while utility repositories maintain stable interfaces with growing downstream adoption.

### 4.3 Threats to Validity

*External Validity.* Our findings may not generalize to all ROS systems, as we focus on open-source projects with at least 10 releases, excluding early-stage or closed-source systems. Furthermore, our study is specific to ROS, and the evolution patterns we observe may be shaped by ROS-specific practices. To mitigate the risk of conflating ROS 1 and ROS 2 evolution, we report cross-version comparisons.

*Internal Validity.* Our architectural recovery tool may produce false negatives as rules for topic remappings, dynamically loaded components, and conditional launch includes are not captured, as they require runtime or data-flow resolution beyond source-level pattern matching. Furthermore, our study is scoped to component-and-connector architectural evolution. Future directions can explore the evolution of behavioral properties in these systems. As no ground truth currently exists, evaluating the tool's accuracy would require manually analyzing hundreds of release paths and thousands of file diffs across different systems, which is impractical. To mitigate these limitations, we designed our rules to recognize both current and deprecated API calls exclusively on ROS packages, across ROS 1 and ROS 2, using the official ROS API documentation for each language and version. To support reproducibility, the tool is publicly available in the replication package, allowing independent verification and extension.

## 5 Related Work

Prior work on ROS architecture spans recovery tools, specification languages, and broader software architecture evolution research.

Several approaches recover ROS architectures from source code and configuration files. HAROS [29] and ROSDiscover [33] extract the component-and-connector architecture, ROSInfer [12] infers behavioral information, and Witte et al. [35] combine static launch-file analysis with sandboxed execution. Runtime approaches such as ROS Snapshot [11] and rosbag-based graph extraction [8] capture the architecture during execution. These approaches recover the architecture at a single point in time and require full dependency resolution, making repeated recovery across thousands of releases impractical; our differential approach complements them by detecting changes directly from release diffs.

Prior work also introduced explicit architectural models for ROS. ROSpec [5] proposes a DSL for specifying component configuration, Bardaro et al. [3] adapt AADL for component connections, and related work proposes translating architectures into robotics artifacts [16,22]. Our findings quantify how frequently the elements these languages describe change, directly informing the maintenance burden they impose.

Within the ROS ecosystem, prior work has studied dependency evolution [21], launch-file duplication [13] and variability [19,31], and how roboticists architect ROS systems [20,25]. These works establish that configuration is a known source of complexity, but none studies architectural evolution at the granularity of components, connections, and configurations across releases.

## 6 Discussion

Our results show that evolution follows different patterns depending on level and ecosystem. In this section, we discuss the future directions for architectural analysis tools, specification languages, and the study of software evolution.

***Differential architectural change analysis for monitoring and improving analysis tools.*** Tools such as HAROS [29] and ROSDiscover [33] extract the architecture at a single point in time, treating all elements equally. Our differential architectural recovery tool can complement these in three ways.

First, our co-occurrence analysis (Figure 4) reveals that architectural changes follow specific patterns within releases. For instance, developers rarely add and remove elements in the same release, and component additions seldom occur alongside configuration changes. Tools can leverage these patterns to identify incomplete modifications or flag unusual combinations of changes.

Second, architectural changes are points where misconfigurations are likely to be introduced [6]. Prior tools require full architecture recovery to detect *errors*, but by focusing on changes, our approach may *warn* about potential misconfigurations. For instance, as developers may hardcode component and connection names, detecting that these changed between releases can flag potential mismatches.

Finally, the differential approach is lightweight enough to integrate into continuous integration (CI) pipelines. Architectural change detection tools could automatically warn developers about potential misconfigurations before deployment. Furthermore, monitoring architectural evolution could help downstream integrators assess the stability of packages they reuse, and tracing how upstream changes propagate to integrators is a natural extension.

***Maintenance and design of specification languages.*** Specification languages such as AADL for ROS [3] and ROSpec [5] can verify correct component integration, but maintaining them may become a burden [14]. Our findings inform the maintenance and design of these languages in three ways.

First, the maintenance burden is highest early in a package's lifetime, when developers must write full definitions, and decreases as packages mature toward cleanup (Figure 3). As the activity shifts from additions to removals, updating specifications becomes lighter than writing them from scratch. This suggests a lower long-term burden of maintaining languages like ROSpec.

Second, our findings can inform specification language design. Specification languages such as ROSpec [5] and AADL for ROS [3] describe structural composition: which components exist and how they are parameterized and connected. Our results (Table 2) show that these elements change at different rates. Arguments are the most volatile, followed by publishers and subscribers, while services, actions, and environment variables are comparatively stable. For instance, languages could treat volatile elements as first-class citizens, grouping them together so that the most frequently revised parts of a specification are easy to update.

Finally, the differential approach can be integrated into developer workflows to detect stale specifications, as it detects that the architecture *did* change even with-

out recovering precisely *how*. For example, when a component definition is removed or modified (Figure 6), downstream integrators whose specifications reference that component can be automatically warned about potential mismatches [28], preventing their architecture from drifting from the specification [32].

***Integrating software evolution into developer workflows.*** Our differential approach leverages release diffs to infer architectural changes rather than recovering the full architecture, detecting changes across thousands of releases. This approach is not restricted to ROS and could be applied to any component-based framework with architectures defined through API calls. By integrating this into the developer lifecycle, developers can continuously monitor their architecture, receive warnings when specifications become stale, and assess the stability of packages they depend on. Furthermore, architectural evolution patterns could serve as a project health metric, where a package with frequent changes late in its lifetime may signal instability for downstream integrators.

## 7 Conclusion

In this work, we conduct an empirical study of architectural evolution across 1728 ROS packages and 9914 releases, using a differential, cross-language recovery tool applied to releases. We find that evolution diverges across scales: repositories remain addition-dominant, while mature packages consolidate through late-life removal of connections and configurations. Architectural changes co-occur predictably, with component removals strongly associated with connection and configuration removals, and system repositories driving the strongest cleanup through source-level node definitions. These findings have direct implications for the ROS ecosystem. Analysis tools can leverage co-occurrence patterns to detect potential misconfigurations without full architecture recovery. Specification languages can group volatile elements for easier revision, and since maintenance burden decreases as packages mature, the long-term cost of adopting them may be lower than expected. Our study does not assess change severity, is correlational, and is limited to open-source projects. Future work could weight changes by impact, distinguish intentional from incidental modifications, and extend the methodology to industrial projects and other component-based frameworks.

## 8 Data Availability Statement

The recovery tool is publicly available along with our dataset, including the extracted architectures, computed changes across releases, and the analysis scripts used in this paper, at https://figshare.com/s/6843f1c2fd88b48b6ed6.

## 9 Acknowledgements

This work used AI-assisted tools during its preparation. OpenAI Codex 5.4 was used in the creation and generation of data analysis scripts and figures. Claude Opus 4.6 was used to detect misspellings and improve the fluency of the paper.

## References


1. Albonico, M., Dordevic, M., Hamer, E., Malavolta, I.: Software engineering research on the robot operating system: A systematic mapping study. Journal of Systems and Software **197**, 111574 (2023)
2. Anonymous Author(s): A large-scale study of reuse in ROS-based software on github. In: In Submission (2026)
3. Bardaro, G., Semprebon, A., Matteucci, M.: A use case in model-based robot development using AADL and ROS. In: International Workshop on Robotics Software Engineering. pp. 9–16 (2018)
4. Benjamini, Y., Hochberg, Y.: Controlling the false discovery rate: a practical and powerful approach to multiple testing. Journal of the Royal statistical society: series B (Methodological) **57**(1), 289–300 (1995)
5. Canelas, P., Schmerl, B., Fonesca, A., Timperley, C.S.: ROSpec: A domain-specific language for ROS-based robot software. In: Proceedings of the ACM on Programming Languages (2025)
6. Canelas, P., Schmerl, B., Fonseca, A., Timperley, C.S.: Understanding misconfigurations in ROS: An empirical study and current approaches. In: International Symposium on Software Testing and Analysis (2024)
7. Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtós, N., Carreras, M.: Rosplan: Planning in the robot operating system. In: International Conference on Automated Planning and Scheduling. pp. 333–341. AAAI Press (2015)
8. Chen, Z., Albonico, M., Malavolta, I.: Automatic extraction of time-windowed ROS computation graphs from ROS bag files. CoRR **abs/2305.16405** (2023)
9. Chitta, S., Marder-Eppstein, E., Meeussen, W., Pradeep, V., Rodríguez Tsouroukdissian, A., Bohren, J., Coleman, D., Magyar, B., Raiola, G., Lüdtke, M., Fernández Perdomo, E.: ros_control: A generic and simple control framework for ROS. The Journal of Open Source Software **2**, 456 (2017)
10. Chitta, S., Sucan, I.A., Cousins, S.: Moveit! [ROS topics]. Robotics & Automation Magazine **19**(1), 18–19 (2012)
11. Drumheller, W.R., Conner, D.C.: Online system modeling and documentation using ROS snapshot. Journal of Computing Sciences in Colleges **36**(3), 128–141 (2020)
12. Dürschmid, T., Timperley, C.S., Garlan, D., Le Goues, C.: ROSInfer: Statically inferring behavioral component models for ROS-based robotics systems. In: International Conference on Software Engineering (2024)
13. Estefo, P., Robbes, R., Fabry, J.: Code duplication in ROS launchfiles. In: International Conference of the Chilean Computer Science Society. pp. 1–6 (2015)
14. Estefo, P., Simmonds, J., Robbes, R., Fabry, J.: The robot operating system: Package reuse and community dynamics. Journal of Systems and Software pp. 226–242 (2019)
15. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. Computing Surveys **35**(2), 114–131 (jun 2003)
16. Garcia, N.H., Deshpande, H., Santos, A., Kahl, B., Bordignon, M.: Bootstrapping MDE development from ROS manual code: Part 2 - model generation and leveraging models at runtime. Softw. Syst. Model. **20**(6), 2047–2070 (2021)
17. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is still so hard. IEEE Software **26**(4), 66–69 (2009)
18. Horowitz, J.L.: Bootstrap methods in econometrics. Annual Review of Economics **11**(1), 193–224 (2019)

19. Jiang, G., Mao, X.: Exploring problems and solutions about launch files in ROS from Q&A community. In: Proceedings of the 2022 3rd International Conference on Big Data & Artificial Intelligence & Software Engineering (ICBASE 2022). vol. 3304, pp. 9–14. CEUR Workshop Proceedings (2022)
20. Kidwai, H., Bates, D.P., Suhi, S.I., Young, J., Chowdhury, S.: Robotics meets software engineering: A first look at the robotics discussions on stackoverflow. CoRR **abs/2410.04304** (2024)
21. Kolak, S., Afzal, A., Hilton, M., Le Goues, C., Timperley, C.: It takes a village to build a robot: An empirical study of the ros ecosystem. In: International Conference on Software Maintenance and Evolution. pp. 430–440 (2020)
22. Kumar, P.S., Emfinger, W., Kulkarni, A., Karsai, G., Watkins, D., Gasser, B., Ridgewell, C., Anilkumar, A.: ROSMOD: a toolsuite for modeling, generating, deploying, and managing distributed real-time component-based software using ROS. In: International Symposium on Rapid System Prototyping. pp. 39–45 (2015)
23. Macenski, S., Moore, T., Lu, D.V., Merzlyakov, A., Ferguson, M.: From the desks of ROS maintainers: A survey of modern & capable mobile robotics algorithms in the robot operating system 2. Robotics Autonomous Systems **168**, 104493 (2023)
24. Macenski, S., Foote, T., Gerkey, B., Lalancette, C., Woodall, W.: Robot operating system 2: Design, architecture, and uses in the wild. Science Robotics **7**(66) (2022)
25. Malavolta, I., Lewis, G.A., Schmerl, B.R., Lago, P., Garlan, D.: Mining guidelines for architecting robotics software. Journal of Systems and Software **178** (2021)
26. Moore, T., Stouch, D.: A Generalized Extended Kalman Filter Implementation for the Robot Operating System. In: International Conference on Intelligent Autonomous Systems. pp. 335–348 (2014)
27. Quigley, M., Conle, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.: ROS: an open-source robot operating system. ICRA Workshop on Open Source Software **3**(3.2), 1–6 (01 2009)
28. Rosik, J., Gear, A.L., Buckley, J., Babar, M.A., Connolly, D.: Assessing architectural drift in commercial software development: a case study. Software: Practice and Experience **41**(1), 63–86 (2011)
29. Santos, A., Cunha, A., Macedo, N.: The high-assurance ROS framework. In: International Workshop on Robotics Software Engineering. pp. 37–40 (2021)
30. Santos, A., Cunha, A., Macedo, N., Arrais, R., dos Santos, F.N.: Mining the usage patterns of ROS primitives. In: International Conference on Intelligent Robots and Systems. pp. 3855–3860 (2017)
31. Santos, A., Cunha, A., Macedo, N., Melo, S., Pereira, R.: Variability analysis for robot operating system applications. In: International Conference on Robotic Computing. pp. 111–118 (2022)
32. de Silva, L., Balasubramaniam, D.: Controlling software architecture erosion: A survey. J. Syst. Softw. **85**(1), 132–151 (2012)
33. Timperley, C.S., Dürschmid, T., Schmerl, B.R., Garlan, D., Le Goues, C.: ROSDiscover: Statically detecting run-time architecture misconfigurations in robotics systems. In: International Conference on Software Architecture. pp. 112–123 (2022)
34. Tree-sitter Contributors: Tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io/tree-sitter/ (2025), accessed: 2026-03-10
35. Witte, T., Tichy, M.: Checking consistency of robot software architectures in ROS. In: Workshop on Robotics Software Engineering. pp. 1–8 (2018)
36. Zheng, K.: ROS navigation tuning guide. CoRR **abs/1706.09068** (2017)