2019 Third IEEE International Conference on Robotic Computing (IRC)

Can I depend on you? Mapping the dependency and quality landscape of ROS packages

Marc Pichler[1], Bernhard Dieber[1], and Martin Pinzger[2]

[1]Institute for Robotics and Mechatronics, JOANNEUM RESEARCH, Klagenfurt, Austria
[2]Software Engineering Research Group, Alpen-Adria-Universität Klagenfurt, Austria

***Abstract*— Since its beginnings ten years ago, the Robot Operating System (ROS) has created a huge community of developers and researchers and is now the most widespread open-source framework for robotics development. It is used in research, prototyping but also in commercial products and supports a wide range of robotic platforms, sensors and high-level data processing functions. While for a research platform, quality of the software developed with it is typically of lower importance, ROS is gradually moving towards industrial applications making software quality a premier topic. In this paper, we want to gain insights on how ROS is used in practice, how high the quality of the ROS packages and applications is, and where potential pitfalls in the use of ROS lie. To achieve this, we have analyzed several thousands of open-source ROS packages found on GitHub and Bitbucket for their quality and their inter-dependencies. Our results include packages on the `rosdistro` index and, more importantly, packages that are not. From our results we show common issues in ROS-applications, quality implications, and also which packages of what quality are particularly popular.**

I. Introduction

The Robot Operating System (ROS) [1] is the most popular software framework for building robotic applications to date. Despite its already broad use, it is expected to grow even stronger in the coming years along with the boom in intelligent robotics. This also requires the ROS-based software to be of high quality and sufficient robustness.

ROS has an avid community of students, researchers and - amongst others in the form of the ROS Industrial Consortium[1] - industry partners alike creating and publishing open-source software intended to control manipulators and build complex robotic-systems based on a publish/subscribe architecture.

Due to this community being comprised of a large number of developers from different backgrounds, researchers have questioned the code quality of community-created ROS packages for quite some time. The use of these packages in a high-risk environment, such as robotics, further complicates the issue. In such environments, lack of software-testing and other formal verification methods implies increased risk of property damage, or worse, injury and loss of life.

*The work reported in this article has been supported by the Austrian Research Promotion Agency in the program "ICT of the Future" (grant no. 861264) and by the Austrian Ministry for Transport, Innovation and Technology (BMVIT).

[1]`https://rosindustrial.org/`

Aside from potentially dangerous effects caused by a lack of tests, many community-developed ROS-based applications have suffered from unclear dependencies to third-party packages. While ROS tries to counteract this problem by offering a dependency manager - `rosdep` - we will show that many developers do not use it correctly; causing the experience of someone unfamiliar with the project compiling these packages to be tedious and time-intensive.

In order to get an overview on which ROS packages are predominantly in use, the ROS community constantly maintains a list of open-source packages. With this list, some efforts have been undertaken to provide an overview of the ROS landscape in form of the ROS-Wiki.[2] However, many open-source packages are not being indexed and documented, possibly skewing the big picture in terms of code-quality, making deficiencies in ROS applications as a whole look less apparent and therefore safer than they actually are.

While others have examined code-quality, and provided static-analysis methods to measure them [2], we want to provide a conservative estimation of how many ROS-based applications actually exist. To this end, we try to compile a list of repositories available to the public, with repositories that are on the index acting as a starting point. Taking off from that, we collect more data and set out to find non-indexed packages on various social coding sites, namely GitHub and Bitbucket.

As an example, consider `iiwa_stack`, a software stack developed by researchers at the TU Munich. It is currently not listed as a package in the ROS distribution, but available for use under BSD-License on GitHub.[3] In its original use-case, it has been proposed to be used to acquire ultrasound data on humans using a KUKA LBR iiwa lightweight collaborative robot [3]. At the time of writing, this repository had 97 forks, and has been starred by 105 distinct users. However, taking a closer look at the repository reveals that it does neither contain unit- nor integration-tests. Since the robot itself provides safety features specifically for use in human-robot-collaboration, in this case, the risk a human would be exposed to is relatively small; in other instances however, this might not be the case.

Furthermore, due to the fact that unlisted packages do not provide an easy way to check their dependencies the

[2]`http://wiki.ros.org/`
[3]`https://github.com/IFL-CAMP/iiwa_stack/`

978-1-5386-9245-5/19/$31.00 ©2019 IEEE
DOI 10.1109/IRC.2019.00020

same way listed packages do, the risk of having a lower quality package somewhere in the dependency graph grows as well. While some packages could be well tested, they might require the proper operation of an untested package, possibly rendering the testing efforts ineffective.

The contributions in this paper are threefold. First, we present an overview on the dependency structure of ROS packages and point out conclusions we draw from that. Second, we define quality metrics for ROS packages and categorize their code repositories as a first approximation to how quality propagates through the dependency graph. Finally, we provide a tool for developers to inspect the dependency graph and the repository quality of ROS packages.

This paper is structured as follows: Section II discusses related work on this issue. In Section III we present our approach to mapping the ROS-Landscape, as well as an overview on how the subsequently analyzed data has been collected. Section IV presents our analysis results and discusses their possible causes, and Section V wraps up the paper and provides an outlook on future work.

II. Related Work

Quality of robotic software has received some attention in research already. As argued by Reichardt et al. [4], robotic frameworks should foster good software design by exhibiting certain design elements themselves, which then tend to be adopted by developers using the frameworks. With the broader use of ROS in applied research and also product development, also the research interest in the quality of ROS and ROS-based software has recently increased. While some works are mainly concerned with ROS itself, e.g., concerning runtime verification [5] or security [6], [7], also work regarding the quality of software using ROS has been presented.

Recently, Santos et al. have presented a tool called HAROS [2], which is a custom code quality analysis framework for ROS packages. The authors draw interesting conclusions concerning the quality of important ROS packages. In a follow-up work, they present how the ROS primitives like publish/subscribe or services are used in real implementations [8]. HAROS is also the official tool of the ROS software quality working group. In [9], static source code analysis is used to generate the message flow and thus, analyze component dependencies based on this. From this data, a model of the application can be generated which describes the temporal context of communication between nodes. Also the work of Cortesi et al. [10] shows the use of static code analysis to improve software quality in robotics. Recently, HAROS has been extended by static analysis methods to extract the ROS graph and use this as basis for property-based testing [11].

Sharma et al. propose a method to determine the impact of code changes to the outgoing data rate of ROS nodes [12]. This helps developers to isolate the reason of regressions or changed behavior in components.

Ore et al. have demonstrated that the message definition and the use of those messages in source code sometimes exhibit inconsistencies which leads to problems in the application execution [13]. An approach to formal verification has been proposed in [14]. Here, timed-automata are used to identify inconsistent states, which during execution can result in hard to trace errors.

A very strict approach to implementing algorithms using the SPARK language is presented in [15]. The re-implementation of navigation algorithms in this language, which is specifically tailored for reliable software, showed that significant improvements in resulting quality can be achieved.

While existing work mainly focused on the small-scale analysis of a couple of ROS packages, we provide a large-scale analysis of several quality attributes of ROS packages available to the public.

III. A map of ROS packages

In this section, we discuss how ROS manages package dependencies followed by the five research questions that we aim to answer. Then, we present the approach we took to collect the data for our analysis as well as the approach to conduct the analysis.

A. ROS concepts for dependency management

A ROS application is typically composed of multiple packages. This increases the re-usability and scalability of applications. Each package can declare dependencies, which it needs to function. Dependencies can be either other ROS packages or system dependencies (i.e., libraries or tools which must be installed such as `boost` or `openCV`).

As an example, the package `openni_camera` has (amongst others) system dependencies to the logging framework `log4cxx` and the `libopenni-dev` library and depends on the ROS package `sensor_msgs`, which defines the message data types (information from the indigo-develop branch[4]).

ROS defines multiple types of dependencies like build and run dependencies describing what the current package requires to be built or executed. The various dependencies of different types are defined for each package separately either in the `package.xml` file used by the `catkin` build system or in the `manifest.xml` file of the legacy buildsystem `rosbuild`.

Further, `rosdep` is the dependency management tool for ROS. It reads the required dependencies for a package and installs them. A backend git repository contains descriptive files, which enable `rosdep` to find a specified dependency. `rosdep` can resolve dependencies in various ways like using the system's package management (e.g., apt or yum).

One of the official repositories, `rosdistro`[5], contains a community-maintained list of packages available for every distribution of ROS. Distribution lists may contain version information on every listed package, as well as links to the source code, release and documentation repositories.

[4] https://github.com/ros-drivers/openni_camera/
[5] https://github.com/ros/rosdistro/

These repositories are then in turn used to update the ROS-Wiki with information about dependencies, distribution-compatibility, maintenance status, and it provides the developers with documentation on the package, if available. In addition to the distribution lists, the repository also contains multiple YAML files used by `rosdep`. These files specify aliases for system dependencies that allow them to be independent from the target platform. For instance, apt-repositories on Ubuntu use a different naming scheme as yum-repositories on Fedora. This benefits the package's portability by eliminating the need to go through the packages documentation in hopes of finding a list of dependencies there.

Dependencies which are not listed in `rosdistro` must be manually installed to satisfy a dependency, or cloned from remote repositories by means of a `rosinstall` file. A `rosinstall` file contains links to remote repositories that need to be cloned into the current workspace and compiled on the developer's machine, in order to satisfy the dependency.

B. Research questions

With our analysis we aim to answer the following five research questions:

RQ1: *How extensive is ROS' index of packages?*
RQ2: *Does quality of indexed and non-indexed ROS-Packages significantly differ, and how do common errors look like?*
RQ3: *How many packages and system-dependencies are currently unaccounted for?*
RQ4: *Does the popularity and the quality of ROS packages correlate?*
RQ5: *Do high quality repositories tend to depend on packages of high quality?*

RQ1 aims to investigate, which portion of widely referenced packages are actually resolvable via `rosdep`. In addition, continuous monitoring of this number gives indications how active and large the ROS community actually is.

From answering RQ2, we can conclude if it is beneficial to the quality of a certain package to be listed in the official ROS index. While there are no guidelines or checks to fulfill to get a package on this list, it could be motivating for developers to increase the quality due to the increased visibility and more frequent use of such a package. In looking for answers to RQ2, we are able to assess the maturity of the ROS ecosystem. Additionally, in a large system like ROS, it is easy to use it in a way it was not intended to. By finding out common errors in the usage of ROS (and its tools), this can provide indications where it should be improved.

RQ3 complements RQ1 by concluding if all important dependencies can already be found or if there are valuable packages missing. This might lead to further insights on the workflow which brings a package into `rosdistro`.

RQs 4 and 5 deal with the quality of the repository that hosts a ROS package. This is a very important indication on the popularity and on the developers' working habits. A well maintained repository is assumed to contain well-defined and higher-quality code.

IV. Approach

In this section, we present the approach to answer our research questions, what metrics we use, and how we retrieve and analyze the representative data.

We first look at the total number of packages in the `rosdistro` index and compare it to the total number of packages, which can be found on popular social coding platforms, such as GitHub. The difference enables us to find out whether there is a large number of important packages not present in the official ROS package index. For classifying the importance of a package, we consider counting the number of packages that depend on it. We assume that the more packages depend on a package, the more important it is. Regarding the software quality, we consider two dimensions. First, we use standard static code analysis, namely Google's `cpplint` to check for defects in the code. Second, we consider several quality indicators of the source code repositories that host the code of each package. They are described in the following.

A. Code repository quality

To make the analysis of dependencies more meaningful, we want to approximate the quality of packages that ROS applications depend on. Our approach includes the collection of a number of properties and metrics from the code repositories of those packages. The broken window theory [16] tells us that a house that is already damaged is more likely to receive more damage than an intact house in the same neighbourhood. Transferred to software development, this means that a well-maintained repository is more likely to contain high-quality software than an ill-maintained one [17]. Furthermore, researchers in other areas have shown that the success of software also depends on the quality of the underlying libraries/frameworks that they use. For instance, Bavota et al. [18] and Linares Vasquez et al. [19] showed that the change and fault proneness of an API used by an Android app is related with the app rating, i.e., high-rated apps depend on more stable and reliable libraries. Therefore, we use the maintenance quality of a repository to indirectly conclude towards the work habits of developers and their resulting code quality.

Since we mainly look at GitHub and Bitbucket, we also use the information available on such social coding platforms. As an example, Github stars are an indication for popular repositories. In addition, modern software development–especially in open-source domains–tends, to follow commonly-accepted rules in what a repository should contain. For instance, it is common that a repository on GitHub or Bitbucket contains readme files and uses a pull-request and/or branches to structure the development. In the field of empirical software engineering, mining open source repositories is an accepted approach to generate statistical evidence on how software is developed [20]. In [21] a large study on the use of GitHub is performed where it is also

TABLE I

METRICS FOR ASSESSING THE QUALITY OF SOURCE CODE REPOSITORIES.

Metric	Type	Description
Number of stars	Integer	The number of times users have marked a repository with a star.
Readme	Boolean	Indicates that a readme file of a minimum length is present
Changelog	Boolean	Indicates that a changelog file of a minimum length is present
Contributors	Integer	The number of people contributing to this repository
Issue duration	Float	The average time between opening and closing an issue in seconds
Number of issues	Integer	The total number of open and closed issues
Number of pull requests	Integer	The total number of open and closed pull requests
Branches	Integer	The number of branches in the repository

shown that users judge the quality of a project from its community status (stars, forks ...).

Motivated by the existing work on open source repositories, we come with a set of metrics to assess the quality of repositories hosting the source code of ROS packages and a tool to mine them. The metrics are shown in Table I. We use the number of stars as indication for the repository popularity. Combined with the number of issues and pull requests, it indicates if a repository has an active community. The presence of readme and changelog files are indications of good open-source development culture. The number of contributors is used to categorize repositories into single-person and team efforts. The average duration of resolving issues can be used to assess the involvement of the repository contributors with their community. The existence of pull-requests and branches indicates that the developers adhere to modern collaborative development principles.

In addition based on these metrics, we classify each repository into one of the following four classes.

1) *one-person show*
2) *lone wolf*
3) *good team effort*
4) *improvable team effort*

In general, we distinguish single-person and team efforts. We assume that single repositories that are maintained by a single person suffer from a higher risk of being abandoned than repositories with contributions from multiple developers.

The *one-person show* categorizes repositories that are maintained by a single person but generally have a high quality and are popular with the community. Members of this class have at least 10 stars, readme and changelog files, multiple branches, and open or closed issues. The rationale for introducing this class is to find out if there are popular high-quality ROS packages that are maintained by a single person.

The *lone wolf* class categorizes repositories that are maintained by a single person but do not achieve the quality requirements for a one-person show. We use this class to find out if ROS packages depend on repositories that are maintained by a single person but are of low quality.

The *good team effort* is equal in its quality requirements to the one-person show but has two or more contributors. The comparison to the one-person show in terms of popularity and number of dependents will show if teams achieve a wider use of their packages than a single person.

The *improvable team effort* is the complement class to the good team effort and just as the lone wolf marks those repositories that do not achieve high quality and do not have a strong community. Also here, we want to find out whether many ROS packages depend on such repositories of potentially questionable quality.

B. Data retrieval

To answer the question how extensive ROS's index is, we start by examining the information that is provided by a standard ROS installation. The `rosdistro` repository provides a list of all packages that are on the ROS dependency index. To find the packages not listed on the index, we follow the hypothesis that the majority of such packages are shared on GitHub and Bitbucket. We considered also packages hosted on GitLab servers that are accessible to the public while we excluded packages on self-hosted Git- or SVN-Servers.

We developed a tool suite to mine the information on potential ROS packages on the aforementioned hosting platforms. Regarding GitHub, we use GitHub's search API to compile a list of packages associated with the ROS-Framework. This search returned 1102 repositories. To extract a list of repositories from Bitbucket, we used its online search feature (since Bitbucket does not provide a search API). This search resulted in 572 repositories. We also mined several public accessible GitLab servers, however we excluded these repositories since a manual check showed that the results contain a large number of false positives, i.e., repositories that did not contain any ROS-related content.

Then, we merged the list of repositories obtained from GitHub and Bitbucket with the list of repositories from the official ROS dependency index removing all duplicates. Counting this merged list, this resulted in a total of 4394 repositories that comprise 80.4 GB. In contrast, the repositories extracted from all distribution lists combined only made up 2795 repositories. Furthermore, some of these repositories were not reachable because their hosts did not exist anymore, or because they have been converted to private repositories.

Next, we used GitHub's and Bitbucket's APIs to mine for each repository the number of stars and the number of issues and pull-requests. Concerning the issues, we calculated the average duration from opening to closing an issue. Since Github's API considers pull-requests as issues, we differentiated between issues and pull-requests while retrieving the data. Furthermore, because the concept of stars does not exist on Bitbucket, we used Bitbucket's watchers as the closest compatible concept to the GitHub stars.

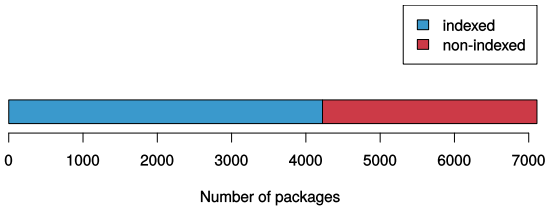

Fig. 1. Distribution of indexed and non-indexed packages in our analysis.

C. Data analysis

In general, ROS is used with two build systems: the legacy build system `rosbuild` and the current build system `catkin`. The latter handles package descriptions in form of `package.xml` files, they contain the package-name, and its different types of dependencies, such as build- or run-dependencies. The `manifest.xml` files used by `rosbuild` serve the same purpose, with the difference that the package-name is determined by the folder-name. Since every ROS-Package is mandated to have such a file, it serves as a starting point for our analysis, and allows us to ignore packages that do not use ROS in a classical sense. Our analysis therefore does not include applications based on `rosbridge`, a system used to communicate with ROS outside of it's usual scope, such as via Websocket. However, these systems are not considered pure-ROS programs since they do not specify dependencies, and only communicate with the system via its JSON API.

To generate a complete dependency graph (showing which package depends on which others), we extracted all `package.xml` and `manifest.xml` files, and analyzed them by matching package names to the names of all dependent packages specified in the files. Since all source code and version control information is available in the cloned repositories, we also conducted a search of how many packages exist that have not been updated in the last year, and can therefore be assumed to be not maintained anymore. This allows us to find maintained packages that depend on unmaintained packages, which might affect the quality of that package.

To answer, which errors are common in indexed and non-indexed packages, as well as in an effort to answer how quality of these packages differs, we analyzed the source code of the ROS packages with Google's `cpplint`[6] and modified rules to better fit the ROS-Styleguide.

V. EVALUATION

In this section, we present the results of our analysis of ROS packages and the answers to the five research questions.

[6]`https://github.com/google/styleguide/tree/gh-pages/cpplint`

A. Package dependencies and quality evaluation

Our analysis of the indexed and non-indexed ROS packages resulted in 4225 indexed and 2887 non-indexed packages, in total 7111 packages. 6670 packages are using `catkin` while 441 packages are using `rosbuild` that have not been upgraded to `catkin`, yet. An overview of the relation between indexed an non-indexed packages is given in Figure 1. With regards to RQ1, we conclude that while the ROS index is substantial, an extensive set of non-indexed packages, namely 2887, could be found on the social coding platforms GitHub and Bitbucket with reasonable effort.

Table II gives an overview of the top five most referenced packages overall, indexed packages, and non-indexed packages. By comparing the columns one and two, we observe that the ROS core packages are the most frequently referenced ones. Furthermore, the number of references is much higher for packages that are listed on the ROS index. This leads us to the conclusion that while non-indexed packages contribute to the overall ROS landscape, ROS packages primarily depend on indexed packages.

The dependency analysis of all gathered repositories resulted in 8293 packages and system-dependencies. This number originates from the 7111 packages that we extracted from the repositories, and the number of package-names found after scanning those packages for dependencies, which resulted in 1182 "other" packages that have not been analyzed by us. After cross-referencing these 1182 package-names with the available `rosdep`-rules, we found that 678 packages were missing - 325 of them stem from indexed packages and 353 stem from non-indexed packages. In summary and answering RQ3, by extracting the depedants to missing packages from the list of dependencies, eliminating duplicates and counting them we find that for indexed packages, 387 (9.1%) depend on missing packages, while the same is the case for 286 (a slightly higher 9.9%) of non-indexed packages.

Manually inspecting random samples of these packages, many missing dependencies originate from one of the following three errors:

1) The package cannot be found anymore via online search
2) Incorrect `rosdep` use: using system-dependencies that are not part of any `rosdep` rule
3) Typo in the specification of the `rosdep`-rule

While incorrect `rosdep` use and spelling mistakes were mostly prevalent in non-indexed packages, indexed packages suffered mainly from missing packages that were not available anymore, for instance because the service hosting the package, such as Google Code, was not available any more. With respect to answering RQ2, this indicates that a more correct use of `rosdep` rules might improve the overall quality of non-indexed ROS packages. Additionally a lower entry-barrier for adding `rosdep`-rules might allow more developers to add their suggestions to the list and increase the number of available `rosdep`-rules.

All packages combined show 49024 dependencies between each other. By dividing this number with the total number

TABLE II
TOP FIVE MOST USED ROS PACKAGES

	Overall		Indexed		Non-indexed	
#	Package name	Dependants	Package name	Dependants	Package name	Dependants
1	catkin	6616	catkin	3965	autoware_msgs	43
2	roscpp	3105	roscpp	1542	uchile_srvs	24
3	std_msgs	2366	std_msgs	1071	uchile_msgs	24
4	rospy	1849	geometry_msgs	846	matec_msgs	21
5	geometry_msgs	1585	rospy	820	matec_utils	21

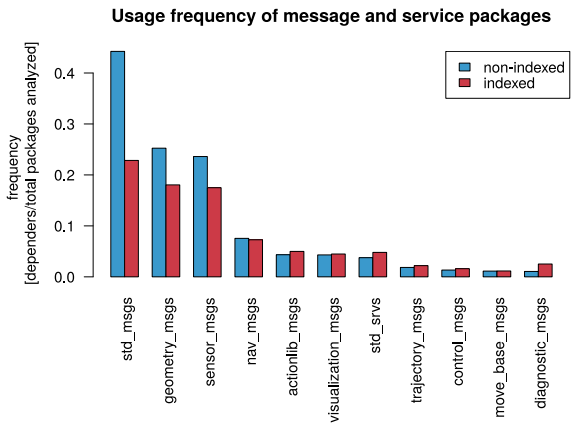

Fig. 2. Use of message or service packages relative to number of packages analyzed, grouped by indexed (red), and non-indexed (blue) packages

of packages analyzed, we arrive at an average number of 6.89 dependent packages per analyzed package. Non-indexed packages account for 18315 of these dependencies or an average of 6.34 dependencies per package. Indexed packages account for 30709 dependencies, or 7.26 dependencies per package. A manual investigation of a small sample of non-indexed packages revealed that, while their average dependency count is lower than the one of indexed packages, many packages define dependencies that they do not actually use in their manifest file. For example, packages define dependencies to both, `roscpp` and `rospy`, even though all of the package's code is written entirely in C++. Indexed packages did not show this kind of error. However, on average indexed packages declared more dependencies per package. One reason that we observed in our manual analysis seems to be that non-indexed packages are typically smaller often containing only one large source file and node.

Out of the 49024 dependencies, 10224 are to separate message and service packages. They account for 20.8% of all the dependencies. The number of message packages is only 578 and thus, 20.8% of all dependencies are to only 7.0% of the packages. This leads us to the positive conclusion, that the re-use of message packages works well in the ROS community and that developers rather reuse standard message packages instead of providing their own. Non-indexed packages seem to pursue this pattern slightly more aggressively, since 23.7% of dependencies originating from these packages are dependencies to messages or services. As seen in Figure 2, the relative use of message packages is much higher compared to other packages. The most frequently referenced message package is `std_msgs`. 22.8% of the indexed packages depend on it, while 44.2% of the non-indexed packages depend on it. A similar, though not so drastic ratio for indexed and non-index packages can be observed for the other two of the top three referenced packages, namely `geometry_msgs` and `sensor_msgs`. For the remaining packages, the differences between indexed and non-indexed packages is more balanced. One notable exception is the package `diagnostic_msgs`, for which the relative usage in indexed packages is almost twice as high as in non-indexed packages. This suggests that publishing of diagnostics for troubleshooting is more common within indexed packages, indicating higher overall quality (RQ2).

Other issues that have been found were malformed XML in manifest.xml or package.xml files. This is a severe error that prevents a package from building with `catkin`. With regards to RQ2, we found that while none of these errors were prevalent in indexed packages, they occurred in a small number of non-indexed packages.

Further regarding RQ2, we checked all retrieved repositories with `cpplint`. A check for style-guide-violations and bad practices revealed that C++ code from indexed repositories contained an average of 261 violations, while non-indexed repositories contained an average of 862 violations. This result was reached by ignoring include directories, since many repositories delivered libraries like `Eigen` with their source files in the project, making results appear worse than they are in practice, since these libraries are not expected to adhere to the ROS-style-guide. However, often header-only libraries that were added to the project, were also available as `rosdep`-rules. More common than violations of the ROS-style-guide were issues that are not included in the ROS-style-guide, but are generally regarded as bad-practice; these include long lines and whitespace-issues, as well as more serious issues, such as use of C-style casts in C++ source code. While white space issues were prevalent in both, indexed and non-indexed packages, the use of C-Style casts was more prevalent in indexed packages. Furthermore, we found that many packages contained source files of more than 2000 lines of code. This indicates quality issues with C++ code in both, indexed and non-indexed packages.

B. Repository quality evaluation

Regarding the quality of the source code repositories implementing the analyzed ROS packages, we classified them

TABLE III

CLASSIFICATION OF THE TOP 100 REPOSITORIES IN TERMS OF STARS AND DEPENDENCIES

Class	Top 100 (stars)	Top 100 (dependencies)
Lone-Wolf	2	0
One-person-Show	2	0
Improvable-Team Effort	4	17
Good-Team Effort	92	70
System dependencies	0	13

TABLE IV

CLASSIFICATION OF ALL REPOSITORIES

Class	Total number of repositories	Percentage
Lone-Wolf	1132	28.27%
One-person-Show	26	0.65%
Improvable-Team Effort	2376	59.34%
Good-Team Effort	470	11.74%

according to the four categories introduced in Section IV-A. In the following, we present the results of this evaluation.

First, we look at the top 100 repositories in terms of number of stars and number of dependencies as shown in Table III. For the ranking according to stars, clearly the most successful class is the good-team effort with 92 of the top 100 starred repositories. Four repositories are maintained by single developers. If ranked according to the number of dependencies, 70 repositories are of the class good-team effort while 17 repositories belong to the class improvable-team efforts.

In contrast to the top 100 starred classes, the overall numbers show a different picture as shown in Table IV. By far the biggest portion of repositories according to our metrics are assigned to the class improvable-team effort. Combined with the repositories classified as Lone-Wolf they account for more than 87% of all repositories implementing ROS packages. This might be attributed to the fact, that many ROS packages originate from research projects and hardly ever mature from the prototype status. In addition, many packages were almost never written with quality in mind but rather with the aim to complete a specific piece of research, such as demonstrating that a specific functionality can be implemented. We argue that especially this practice is dangerous to the quality of the overall ecosystem because a package found on Github or Bitbucket may promise to solve a specific problem but does that only partially and/or at low quality.

Since a repository can consist of multiple packages, we also want to look at the number of packages per class. In Table V we explicitly also show how many system dependencies there are (i.e., non-ROS packages available via the system's package manager) to exclude them since these packages were not part of our analysis. After subtracting the percentage of system dependencies and then normalizing the remaining to make up 100% we see that bad quality packages make up roughly 80% of all packages. This lower number is to be expected, since also inspecting samples of high and low quality repositories seems to indicate that higher quality repositories (e.g., the official `ros_comm` package) usually contain more than one package while bad repositories often only contain one single package. We explain the low number of one-person show packages with the fact, that a good and useful package will quickly attract a community and also potentially collaborators thus elevating the project to a good-team effort.

Further, we examined how many issues are created for packages of various classes. In addition, we collected data on the number of pull requests. Both numbers are strong indications on the community participation in a repository. The results are summarized in table VI. It can immediately be seen that the higher-quality repositories have more issues and pull requests. We interpret the average number of issues not as a direct measure for bad quality but argue, that a repository of higher popularity naturally gets more issues since more people are using it. On the contrary, a repository with a low number of issues might carry many undiscovered bugs and problems.

Looking at the multiple dependencies from good-quality packages, we can see that 94.49% of dependencies refer to good-quality packages as well, while 5.51% go to packages of lesser quality. Of packages with bad quality, only 15.69% depend on bad quality packages. This seems to indicate that bad quality packages tend to depend on bad packages three times more often than their good quality counterparts (RQ4).

Regarding RQ5, we investigated whether or not the quality of packages correlates with their usage. We found that these two values are mostly unrelated, this leads us to the conclusion, that packages from all parts of the spectrum might be used extensively by developers in private repositories.

VI. CONCLUSION AND OUTLOOK

In this paper, we have given an overview of the current ROS applications landscape. We have shown that packages on the ROS package index are typically more often referenced by other packages. We have further investigated if high-quality repositories are more popular and more often referenced than others.

We found that some often used packages like `moveit` or `rviz` still show many issues that suggest improvable coding practice. A high number of 261 average style guide violations or bad practices per indexed package also shows that improvements in quality are recommended for many ROS packages. The quality of the project does not seem to correlate with their usage according to our results. In order to build high-quality software however, the quality of the dependencies must be equally high as the quality of the own software.

On the positive side, at the example of message packages, it can be seen that the re-use of existing software is very well fostered by ROS thus.

Based on our result, we formulate the following suggestions to ROS developers and the ROS community:

1) Strengthen the use of quality-improving tools like `roslint` to improve code quality.
2) The correct use of `rosdep` should be made easier and tools to check the correctness and validity of references

TABLE V

THE OVERALL NUMBER OF PACKAGES OF A CERTAIN CLASS

Class	Total number of packages	Percentage	Percentage w/o system-dependencies
Lone-Wolf	1016	12.25%	15.39%
One-person-Show	34	0.41%	0.52%
Improvable-Team Effort	3627	43.74%	54.95%
Good-Team Effort	1923	23.19%	29.14%
System-Dependencies	1693	20.41%	

TABLE VI

TOTAL AND AVERAGE ISSUES AND PULL REQUESTS PER CLASS.

	One-person show		Good-team effort		Improvable-team effort		Lone wolf	
	Issues	Pull Requests	Issues	Pull requests	Issues	Pull request	Issues	Pull requests
Total	262	86	40717	48243	9887	14193	303	258
Average	10.08	3.31	86.63	102.65	4.16	5.97	0.27	0.23

should be developed.

3) Consolidate the ROS package index and remove or update packages where the sources are not referenced correctly.
4) Increase the quality and quality-oriented work habits in the maintenance of repositories.

We think that many of these stated issues could also be prevented by offering better tool-support for ROS in general. IDEs like RoboWare Studio that offer automatic management of package.xml files - adding both to convenience and quality - are a welcome step in the right direction. In addition, tools like HAROS, which report on code quality of ROS-packages are a big step towards the professionalization of this ecosystem.

We have released the tool for dependency scanning along with a visualization and an up-to-date data basis for the ROS community to use [7]. We will enhance it with user-selectable quality metrics and plan to integrate the dependency analysis into HAROS in future.

REFERENCES

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, no. 3.2, 2009, p. 5.

[2] A. Santos, A. Cunha, N. Macedo, and C. Loureno, "A framework for quality assessment of ros repositories," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Oct. 2016, pp. 4491–4496.

[3] C. Hennersperger, B. Fuerst, S. Virga, O. Zettinig, B. Frisch, T. Neff, and N. Navab, "Towards mri-based autonomous robotic us acquisitions: A first feasibility study," *IEEE transactions on medical imaging*, 2016.

[4] M. Reichardt, T. Föhst, and K. Berns, "On software quality-motivated design of a real-time framework for complex robot control systems," *Electronic Communications of the EASST*, vol. 60, 2013.

[5] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, *ROSRV: Runtime Verification for Robots*. Cham: Springer International Publishing, 2014, pp. 247–254.

[6] R. White, H. Christensen, and M. Quigley, "Sros: Securing ros over the wire, in the graph, and through the kernel," in *Proceedings of the IEEE-RAS International Conference on Humanoid Robots (HUMANOIDS)*., 2016.

[7] B. Dieber, B. Breiling, S. Taurer, S. Kacianka, S. Rass, and P. Schartner, "Security for the robot operating system," *Robotics and Autonomous Systems*, vol. 98, pp. 192–203, 2017.

[7] https://www.github.com/jr-robotics/rosmap

[8] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. N. dos Santos, "Mining the usage patterns of ros primitives," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Sept. 2017, pp. 3855–3860.

[9] R. Purandare, J. Darsie, S. Elbaum, and M. B. Dwyer, "Extracting conditional component dependence for distributed robotic systems," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Oct. 2012, pp. 1533–1540.

[10] A. Cortesi, P. Ferrara, and N. Chaki, "Static analysis techniques for robotics software verification," in *Proc. IEEE ISR 2013*, Oct. 2013, pp. 1–6.

[11] A. Santos, A. Cunha, and N. Macedo, "Property-based testing for the robot operating system," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, ser. A-TEST 2018. New York, NY, USA: ACM, 2018, pp. 56–62. [Online]. Available: http://doi.acm.org/10.1145/3278186.3278195

[12] N. Sharma, S. Elbaum, and C. Detweiler, "Rate impact analysis in robotic systems," in *Proc. IEEE Int. Conf. Robotics and Automation (ICRA)*, May 2017, pp. 2089–2096.

[13] J. P. Ore, S. Elbaum, and C. Detweiler, "Dimensional inconsistencies in code and ros messages: A study of 5.9m lines of code," in *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS)*, Sept. 2017, pp. 712–718.

[14] R. Halder, J. Proena, N. Macedo, and A. Santos, "Formal verification of ros-based robotic applications using timed-automata," in *Proc. IEEE/ACM 5th Int. FME Workshop Formal Methods in Software Engineering (FormaliSE)*, May 2017, pp. 44–50.

[15] P. Trojanek and K. Eder, "Verification and testing of mobile robot navigation algorithms: A case study in spark," in *Intelligent Robots and Systems (IROS 2014), 2014 IEEE/RSJ International Conference on*. IEEE, 2014, pp. 1489–1494.

[16] J. Q. Wilson and G. L. Kelling, "Broken windows," *Atlantic monthly*, vol. 249, no. 3, pp. 29–38, 1982.

[17] A. Hunt, *The pragmatic programmer*. Pearson Education India, 1999.

[18] G. Bavota, M. Linares-Vsquez, C. E. Bernal-Crdenas, M. D. Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, vol. 41, no. 4, pp. 384–407, April 2015.

[19] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 477–487. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491428

[20] A. E. Hassan, "The road ahead for mining software repositories," in *2008 Frontiers of Software Maintenance*, Sept 2008, pp. 48–57.

[21] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: Transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, ser. CSCW '12. New York, NY, USA: ACM, 2012, pp. 1277–1286. [Online]. Available: http://doi.acm.org/10.1145/2145204.2145396