How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems

Ivano Malavolta
Vrije Universiteit Amsterdam, The Netherlands
i.malavolta@vu.nl

Grace A. Lewis
Software Engineering Institute, Carnegie Mellon University, USA
glewis@sei.cmu.edu

Bradley Schmerl
Institute for Software Research, Carnegie Mellon University, USA
schmerl@cs.cmu.edu

Patricia Lago
Vrije Universiteit Amsterdam, The Netherlands
p.lago@vu.nl

David Garlan
Institute for Software Research, Carnegie Mellon University, USA
garlan@cs.cmu.edu

ABSTRACT

The Robot Operating System (ROS) is *the* de-facto standard for robotic software. If on one hand ROS is helping roboticists, *e.g.,* by providing a standardized communication platform, on the other hand ROS-based systems are getting larger and more complex and could benefit from good software architecture practices. This paper presents an observational study aimed at (i) unveiling the state of the practice for architecture of ROS-based systems and (ii) providing guidance to roboticists about how to properly architect ROS-based systems. To achieve these goals, we (i) build a dataset of 335 GitHub repositories containing real open-source ROS-based systems, (ii) mine the repositories for extracting the state of the practice about how roboticists are architecting them, and (iii) synthesize a catalog of 49 *evidence-based guidelines* for architecting ROS-based systems. The guidelines have been validated by 77 roboticists working on real-world open-source ROS-based systems.

ACM Reference Format:
Ivano Malavolta, Grace A. Lewis, Bradley Schmerl, Patricia Lago, and David Garlan. 2020. How do you Architect your Robots? State of the Practice and Guidelines for ROS-based Systems. In *Proceedings of ACM Conference (ICSE'20).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Robots and the software that controls them are becoming more prevalent as society automates more industries. From autonomous vehicles and factories, to healthcare, services, and commerce, these robots are playing an increasingly important role in many companies' growth and the growing demands of society. As robots become important in more facets of our world, and their tasks become more complex, *engineering* their software to meet quality requirements such as safety and reliability becomes more critical.

One emerging standard framework for developing robotic software is the Robot Operating System (ROS) [28], a set of open source libraries and tools for developing modular robotic functions that communicate with each other in a loosely-coupled, multi-process, distributed environment. ROS also has a set of mature tools for managing software builds and deployment, simulating production environments for testing and development, and sharing common packages. Though it is hard to estimate the current adoption of ROS, as of writing there are 5941 papers that cite the seminal paper on ROS [28] and more and it officially supports more than 140 different types of robots (http://ros.org).

ROS systems are becoming large and complex. However, while there are many open source packages, documents, and examples on how to use ROS, engineering robots with particular properties is still mostly an art and a matter of trial and error. Moving robot software development from an art to an engineering discipline as robots become further commoditized, means that developers need to have ways to more systematically and quickly produce software that meets the quality demands of the domains in which they are used. For example, robots need to be engineered to be safe when they are required to interact with humans; engineers need to know that sensor data about the environment can be consumed and processed in a timely manner to achieve interaction with the world, and robots need to be reliable and secure when they are performing functions critical to industry and infrastructure.

To understand how to achieve well-architected robots, we must first look at the body of knowledge that has already been developed around ROS. We can understand how roboticists have architected existing systems, the guiding principles can be derived from this experience, and the quality requirements concerning them the most.Fortunately, because ROS is open source, it has encouraged the open source development of many robots and robotic components that we can study to find answers to these questions.

This paper addresses two research questions: *What is the state of the practice for architecting ROS based systems?* (**RQ1**), and *How to guide roboticists when architecting ROS-based systems?* (**RQ2**). Our approach consists of two main parts: (1) mining ROS repositories on GitHub, GitLab, and BitBucket to uncover architecture documentation and issues related to architecture design or quality requirements, and (2) surveying developers who actively contributed to those repositories to determine the usefulness of these guidelines, as well as elicit additional guidelines directly.

ICSE'20, May 2020, Seoul, South Korea

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

The main **contributions** of this study are: (1) A *characterization of the state of the practice* with respect to the architecture of ROS-based systems; (2) A set of *49 guidelines* (39 from mining repositories, 10 directly from roboticists) and the *quality requirements* that these guidelines are concerned with; (3) A *validation of these guidelines* from 77 roboticists who were active committers to these projects, ranking their usefulness; (d) The *replication package* for the study.

There are two main targets for the results of this study: (1) roboticists who want to apply good design principles to develop robots that meet quality requirements, and (2) architecture researchers who can use them as evidence-based indications about how real-world ROS systems should be architected, thus inspiring future research contributions such as automated architectural analysis.

The remainder of this paper is organized as follows. We discuss the general software architecture style for ROS-based systems in Section 2. Section 3 illustrates the design of the study used to elicit the guidelines, including the selection criteria for repository and survey participants. We then present results and guidelines (in Section 4), followed by the related discussion and implications for developers and researchers (in Section 5). Section 6 discusses the threats to validity of our study. We close with the related work (in Section 7), and our conclusions and future work (in Section 8).

2 ROS-BASED SYSTEMS

We define a ROS-based system as a system that contains robotics capabilities built using the ROS framework. ROS 1 was developed in 2007 as the development environment for the Willow Garage PR2 robot, but has proven useful for a wide variety of robots [13]. ROS 1 is currently evolving into ROS 2 to address this broader ROS community. Some of the new features in ROS 2 include better support for teams of multiple robots and real-time requirements, as well as improved APIs.

For both ROS 1 and ROS 2, from a software perspective, a ROS-based system is composed of *Nodes*, which are processes that perform computation [29]. Nodes communicate with each other using a publish/subscribe model based on *Topics*, or using a request/reply model based on *Services*, as shown in Figure 1.[1]

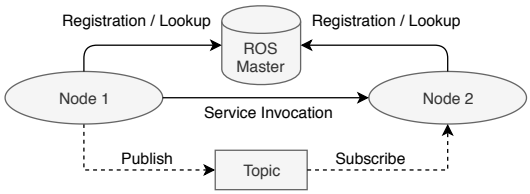


Figure 1: ROS Node Communication

The *ROS Master* provides topic and service registration for Nodes, as well as lookup capabilities. In addition, the ROS Master contains a *Parameter Server* that centrally stores parameters as key/pair values that can be accessed at run-time. There are other concepts in ROS-based systems that will not be covered due to page limitations, such as nodelets and namespaces [28], but they are not necessary as background for this paper.

ROS-based systems are typically started using a launch file. The launch file is used by the roslaunch tool [29] to start the ROS Master and all system Nodes, set parameters in the Parameter Server, and perform any other initialization required by the system. One of the main differences between ROS 1 and ROS 2 is that launch files are written in XML in ROS 1 and in Python in ROS 2.

Figure 2 shows the architecture for a computer vision subsystem built by Team IGVC from the Indian Institute of Technology Kanpur, available on GitHub. It shows nodes as ellipses and topics as rectangles. As an example of communication between nodes, the bottom right of the figure shows a node called *CLASSIFIER NODE* publishing a message to the */prediction* topic, which is subscribed to by the *MASKER NODE* node. For systems with documented architectures, the "ellipses for nodes and rectangles for topics" is a common convention. More sample architectures can be found in the replication package for this study (see Section 3).

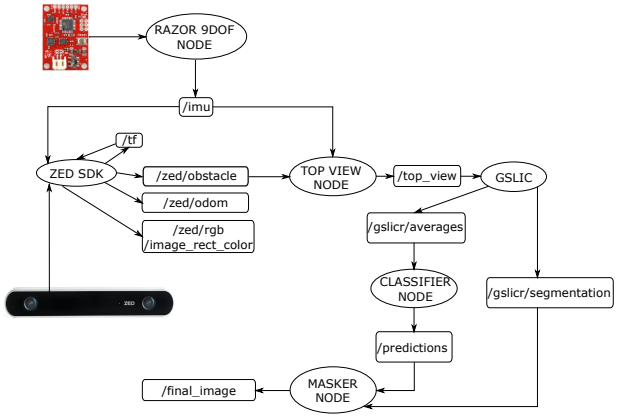


Figure 2: Example of a ROS software architecture – adapted from the IGVC-IITK/computer_vision project

3 STUDY DESIGN

This study has two main goals: (i) to characterize the state of the practice of ROS-based systems and (ii) to empirically identify a set of evidence-based guidelines for architecting ROS-based systems. These goals drive the design of the study and lead us to the following research questions.

RQ1 – *What is the state of the practice for architecting ROS-based systems?* We refined this RQ into the following sub-questions:

RQ1.1 – *What types of systems are implemented using ROS?* This question aims at setting the context for the study by quantitatively assessing the types of systems implemented using ROS, *e.g.,* ground robots, manipulation robots, self-driving vehicles.

RQ1.2 – *What capabilities are provided by ROS-based systems?* This question aims at characterizing (i) whether software repositories contain the full software stack of a robotic system or only a single component (*e.g.,* a computer vision component such as the one in Figure 2); (ii) what robotic capabilities are implemented in ROS. Answering this question will help roboticists and researchers get an overview of the landscape of robotics software implemented in ROS, as well as subsystems that are readily available in the ROS ecosystem and thus become opportunities for software reuse.

RQ1.3 – *To what extent do roboticists document the software architecture of ROS-based systems?* The benefits of architecture documentation have been largely discussed in the past [9]. However, given the wide adoption of agile methods [16], the extent to which

[1] Communication can also use *actions*, which are implemented as a combination of topics and services.

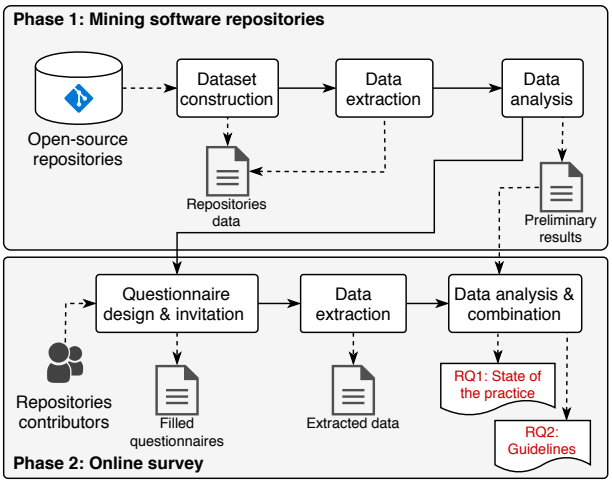


Figure 3: Overview of the Study Design

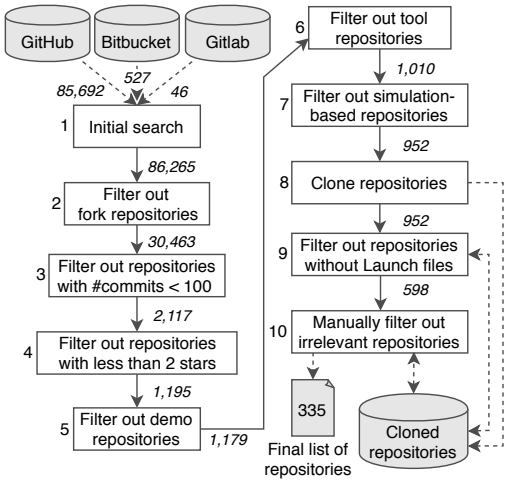


Figure 4: Repository Collection and Filtering

roboticists are documenting their architecture (and how) is not known. This research question aims at filling this gap. A goal is for roboticists to use the collected example architectures as inspiration for how to document the architecture of their own systems.

RQ1.4 – *What quality requirements are considered when architecting ROS-based systems?* With this question we aim at supporting roboticists' understanding of the quality requirements that are potentially impacted the most by their architectural decisions. Even though modern robotics systems have to cope with a large number of quality requirements [2], it appears that the software engineering community tends to primarily focus on performance measurement and functional issues of robotic systems, while neglecting other crucial quality requirements, such as maintainability, energy efficiency, safety [6, 8]. With this research question we aim at objectively assessing this phenomenon in the context of real robotic projects.

RQ2 – *How to guide roboticists when architecting ROS-based systems?* The goal is to provide a catalog of guidelines for architecting ROS-based systems. Roboticists can use the catalog as a source of actionable guidance for architecting their next system. Researchers can use it as a solid foundation for developing new scientific contributions, *e.g.,* techniques to detect violations of the guidelines.

Figure 3 presents the overview of the study design. It follows a *mixed method research methodology* involving two sequential phases. Phase 1 produces preliminary results via a quantitative and qualitative analysis of data mined from open-source repositories (see Section 3.1), whereas Phase 2 complements the results of Phase 1 via an online survey, which targets roboticists contributing to real ROS-based projects (see Section 3.2).

To allow independent replication and verification of the study, we provide a full *replication package*[2] including the details of the research protocol, sample architectures, raw data, mining scripts in Python, and data analysis scripts in R.

3.1 Phase 1: Mining Software Repositories

3.1.1 Dataset Construction. Because the community around ROS has always encouraged open-source development in the form of publicly available packages [12], we can consider open-source software repositories as good data sources for characterizing the practices of roboticists in the context of real-world projects.

Figure 4 shows the steps for building the dataset of our study. The initial search (step 1) is based on (i) *rosmap*, a ROS dependency analysis tool that can also identify ROS-based repositories in GitHub, Bitbucket, and Gitlab [26] and (ii) *ghtorrent*, a widely-used queryable mirror of GitHub [14]. Then we filtered out fork repositories to avoid duplication (step 2). We excluded repositories with less than 100 commits (step 3) and less than 2 stars (step 4) to avoid inactive or non-maintained projects [20]. Since we are targeting real-world projects involving ROS systems used in real contexts, in steps 5, 6, and 7 we filtered out all repositories containing demos, development tools, simulators (or software runnable only on simulators), respectively; in these steps a repository was discarded if either its name, description, or unique identifier matched a black-list regular expression. In order to avoid false positives, we manually checked all the repositories discarded in each of the three steps and refined its black-list regular expressions until no relevant repository was discarded. In step 8 we locally cloned all the 952 repositories, and in step 9 we discarded all repositories not having at least one ROS launch file, either in XML (ROS 1) or Python (ROS 2). Step 9 is necessary because (i) we are looking for repositories containing ROS-based *systems*, rather than simple isolated ROS nodes, and (ii) launch files are predominantly used for system-related activities, *e.g.,* configuring and starting multiple nodes.

Finally, in step 10 we performed an in-depth quality assessment of all the 598 repositories. Guided by the systematic literature review methodology [35], we *manually analyzed* each potentially relevant repository and selected it according to a set of inclusion and exclusion criteria. Two examples of representative exclusion criteria are (the full set is available in the replication package): (1) *The repository contains only collections of snippets of code, examples, or templates*, (2) *The repository contains only testing artifacts (e.g., test cases).* A repository was selected if it satisfied *all* inclusion criteria and *none* of the exclusion criteria. Four researchers were involved in step 10 and conflicts were resolved by a fourth one.

The final dataset is composed of 335 GitHub repositories. As shown in Table 1, the dataset is quite heterogeneous, *e.g.,* in terms

[2] http://github.com/S2-group/icse-seip-2020-replication-package

Table 1: Descriptive statistics of the repository dataset (SD = standard deviation, CV = coefficient of variation)

	Min.	Max.	Median	Mean	SD	CV
Commits	100	7611	272	621.13	991.62	1.59
Contributors	1	233	12	20.86	27.42	1.31
Branches	2	483	6	11.3	29.78	2.63
Issues	0	983	17	60.8	117.48	1.93
Pull requests	0	2165	20	79.87	194.67	2.44
Launch files	1	579	14	30.18	55.22	1.83

of commits, contributors, and launch files. This, in addition to our manual quality assessment, makes us reasonably confident that the repositories considered in this study are of good quality and adequately representative of real-world projects.

3.1.2 Data Extraction and Analysis. For RQ1.1, RQ1.2, and RQ1.3 we inspected the following data sources of each of the 335 repositories: (i) all the markdown files stored in the repository (*e.g.,* readme and change log file of the project), (ii) all external web resources and documents linked in the repository, and (iii) all documents directly stored in the repository. For each data source we conducted iterative *content analysis* sessions with open coding [22] which eventually led to a classification framework composed of three facets, one for each sub-question. Three researchers were involved in this phase over three splits of the dataset: in the first iteration we considered 50 random repositories, then the next 150, and finally the remaining ones. At each iteration we cross-checked the resulting categories and refined the classification framework. The resulting categories are described in Section 4.

For 115 of 335 repositories, full or partial architecture documentation is present (see Section 4 for details). For RQ1.4 and RQ2, two researchers manually inspected all documentation artifacts in those 115 repositories and collaboratively collected 142 unique text fragments where architecturally-relevant concerns are discussed (*e.g.,* presence of integrator nodes, system layers). For answering RQ1.4 we conducted content analysis sessions again, but in this case (i) we targeted architecture documentation fragments and (ii) we used the quality requirements of the ISO/IEC 25010 standard [1] as the initial set of codes. For RQ2 we applied *thematic analysis* [10] to synthesize architecting guidelines (*i.e.,* the themes) from the architecture documentation fragments. We chose thematic analysis because architectural information can be strongly dependent on project- and system-specific characteristics and thematic analysis copes well with context-dependent data [10]. Four researchers were involved in this phase, which led to the definition of a preliminary set of 39 guidelines. The guidelines Contextually to the thematic analysis, we organized the guidelines into 7 families of architectural concerns manifested by developers working on ROS systems (*e.g.,* communication and networking, nodes responsibility, etc.).

3.2 Phase 2: Online Survey

The target population of the online survey is roboticists who worked on at least one real ROS system with some level of architectural design. We extracted all contributors of the 115 repositories containing architecture documentation. Then, in order to increase the quality of the provided answers and to be sure that the context of the project is still fresh in participants' minds, we discarded all contributors who did not make any commits in the considered repositories in the last 12 months. The resulting sample is composed of *520 roboticists distributed over 81 unique GitHub repositories.*

3.2.1 Questionnaire design and invitation. We designed the questionnaire to complement the results we of Phase 1. In the questionnaire we target exclusively RQ1.4 and RQ2. In this phase we followed well-established guidelines for questionnaire design [31].

The questionnaire is composed of 8 questions organized in 4 groups. Group 1 introduces the questionnaire and asks about demographics. Group 2 asks the participant to rate the *usefulness* of each of the 39 preliminary guidelines in their last ROS-based project; answers are provided on a four-point Likert scale and a *Don't know* option. Additional guidelines can be proposed by roboticists in a following open-ended question. Group 3 asks about the top-3 quality requirements considered when working on the last ROS system. Group 4 is optional and asks general feedback about the study. The questionnaire was created as an online form and its complete transcript is available in our replication package.

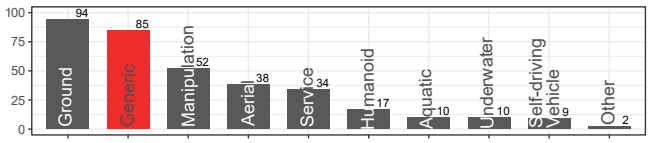
The questionnaire was completed by 77 participants, yielding a 14.8% response rate, which is in line with other studies in software engineering [31]. Participants tend to have multiple years of *experience with ROS* (min=1, max=10, mean=4.5, median=4, SD=2.55) and multiple *contributions to ROS packages* (more than 10 packages=26/77, between 6 and 10=12/77, between 2 and 5=34/77, and 1 package=5/77), and their *primary motivation for using ROS* is mostly academic (39/77), professional (28/77), and others (10/77).

3.2.2 Data extraction, analysis, and combination. With regards to RQ1.4 (*i.e.,* quality requirements), two researchers performed content analysis sessions using the same codes defined in Phase 1 and we report them as an additional perspective to assess the level of agreement between the quality requirements *mentioned in architecture documentation* and the ones *considered important* by roboticists.

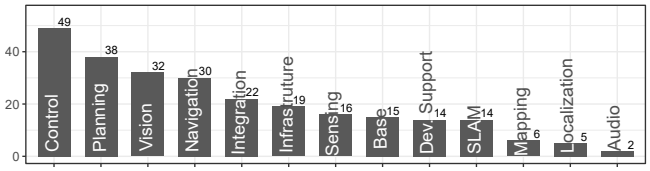
Concerning RQ2, we first collected the usefulness values for each of the preliminary 39 guidelines, and reported them in the form of stacked bar chart. They provide an indication of the applicability of the guidelines in future ROS projects. Second, 21 participants discussed additional guidelines in the follow-up open question; four researchers performed thematic analysis and merged the resulting guidelines with the ones emerged in Phase 1. This additional analysis lead to the identification of 10 new guidelines, leading to the final set of 49 guidelines for architecting ROS-based systems.

4 RESULTS

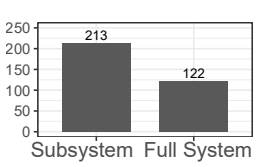
Types of Implemented Systems (RQ1.1). Figure 5a shows the ten different types of systems that were found in the repositories (some repositories were classified in more than one category). *Ground* robots are the prevalent system type (28.1%), followed by a category that we have termed *Generic* (25.4%), to mean that the systems can be used independently of any specific application domain (*e.g.,* object tracker, task manager, parameter wrapper). Categories that follow are *Manipulation* (*e.g.,* industrial robotic arm) (15.6%), *Aerial* (11.3%), *Service* (*e.g.,* mobile robot assistant) (10.1%), *Humanoid* (5.1%), *Aquatic* (3.0%), *Underwater* (3.0%), *Self-Driving Vehicle* (2.7%), and *Other* (a specialized software stack and a research kit) (0.6%).



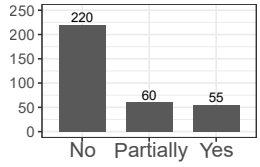
(a) Types of Implemented Systems



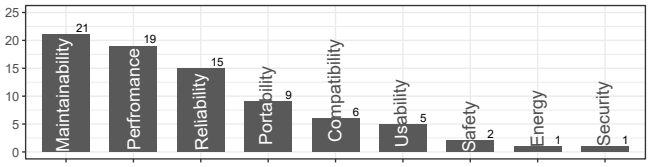
(b) Provided Capabilities



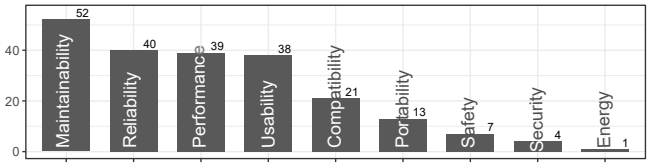
(c) System Scope



(d) Architecture Documentation



(e) Quality Reqts Mentioned in the Documentation



(f) Most Considered Quality Reqts when Working on ROS Systems

Figure 5: Results for RQ1

System Capabilities (RQ1.2). We studied two aspects of the systems: scope and provided capabilities. As far as scope, as shown in Figure 5c, we distinguished between *Subsystems* and *Full Systems*. Subsystems refer to repositories that contain the implementation of a component that is meant to be used in the context of a larger system, and have the larger representation in the set (63.6%). Full Systems refer to repositories that contain complete systems, such as a robot or a group of robots, and account for 36.4% of the set.

Figure 5b shows the distribution of capabilities provided by the subsystems (some subsystems implement more than one capability). *Control* (40.2%), *Planning* (31.1%), *Vision* (26.2%), and *Navigation* (24.6%) are the largest categories, which is not surprising given that these are common capabilities in robotics systems. Categories that follow are *Integration* (*i.e.,* integration with external systems such as the cloud, a web page, and apps) (18.0%), *Infrastructure* (*e.g.,* self-healing, monitoring and logging) (15.6%), *Sensing* (13.1%), *Base* (*e.g.,* component startup, basic configurations, hardware-specific nodes) (12.3%), *Development Support* (*e.g.,* exporters, visualizers, GUIs) (11.5%), *Simultaneous Localization and Mapping (SLAM)* (11.5%), *Mapping* (4.9%), *Localization* (4.1%), and *Audio* (1.6%).

Architecture Documentation (RQ1.3). The repositories were examined to see if we could find architecture documentation for the systems, as explained in Section 3.1.2. Most projects (65.7%) do not have architecture documentation. Some projects have partial documentation (17.9%), meaning that the architecture was either informally described (*e.g.,* via a simplified box-and-line or layered diagram) or presented as a list of topics, services, or nodes, but not how all of them were connected (*i.e.,* no configuration information). Finally, only (16.4%) of the projects have a documented architecture (*i.e.,* the system is described in terms of nodes, services, topics, and their configuration).

Quality Requirements (RQ1.4). For those repositories that contained full or partial architecture documentation (115 repositories), Figure 5e shows the distribution of the quality requirements that were explicitly mentioned in these repositories (details on how these were obtained are in Section 3.1.2. The most frequently mentioned quality requirement is *Maintainability* (26.6%), followed by *Performance* (24.1%), and *Reliability* (19.0%). Fewer mentions were found for *Portability* (11.4%), *Compatibility* (7.6%), *Usability* (6.3%), *Safety* (2.5%), *Energy* (1.3%), and *Security* (1.3%).

Figure 5f shows the results for the question in the questionnaire regarding the most considered quality requirements when working on ROS-based systems. Similar to the results in Figure 5e, the top three quality requirements are *Maintainability* (67.6%), *Reliability* (51.2%), and *Performance* (50.1%). However, according to the survey responses, reliability is considered by roboticists more frequently than performance. The next considered quality requirement is *Usability* (49.4%), which appears in a much higher position than in Figure 5e. This result is not surprising, given that the goal of open-source repositories is precisely to get the community to use them. *Compatibility* (27.3%), *Portability* (16.9%), *Safety* (9.1%), *Security* (5.2%), and *Energy* (1.3%) follow, in positions similar to those reported in Figure 5e. The fact that security and energy are the lowest in both figures, however, is surprising, given that these are robotics systems (see Section 5).

4.1 Guidelines (RQ2)

The guidelines and architectural concerns are shown in Table 2. Within each architectural concern, guidelines are sorted by their level of *usefulness* ("U", column 5 in Table 2), defined as the number of responses in the online survey where a guideline was considered either as `absolutely useful` or `useful`. As shown in Figure 6, the vast majority of survey participants assessed the guidelines as (absolutely) useful. It is important to note that here we are using the usefulness of the guidelines just as a sorting mechanism (not a ranking), so roboticists will need to evaluate the applicability of each guideline depending on the characteristics of their specific system. The last column of Table 2 presents the number of repositories where a guideline has been mentioned across the whole dataset. We notice that (i) more than half of the guidelines (29/49) are mentioned in more than one repository, (ii) only 8 guidelines have been mentioned in only one repository, and (iii) some guidelines are mentioned in a large number of repositories (*e.g.,* I5 and H2). This is a confirmation of the general applicability of the guidelines for roboticists working on different types of robotic systems. Due to space limitations, we only describe the most useful guideline for

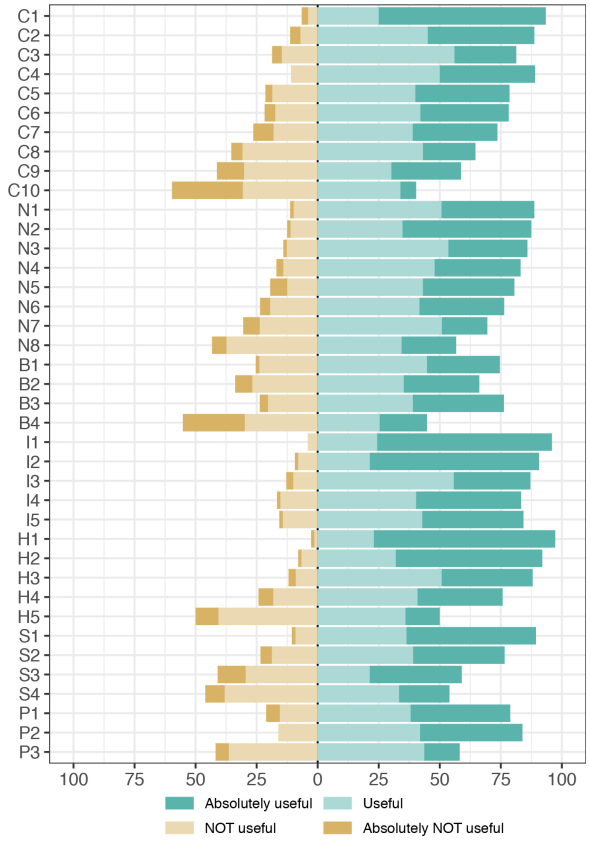


Figure 6: Usefulness of the Guidelines

each architectural concern.

C1 – Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions. The ROS community provides standardized message formats for representing both primitive and widely used data types, *e.g.,* geometric transformations, point clouds, battery levels, etc. These packages are collected into two well-known ROS stacks called `common_msgs` and `std_msgs`.When possible, roboticists should adopt these standardized message formats in order to increase the opportunities for *reuse* for ROS nodes. If commonly used message formats are used within a system, then compatible nodes can be replaced or upgraded with less effort, for example by simply remapping the topics expected by the new nodes to the topics already available in the system.

Standardized message formats also allows roboticists to save effort by using *already-available development tools* such as visualizers for sensor data (e.g., `rqt_plot`) and SLAM algorithms (*e.g.,* `gmapping`). C1 also improves the overall *testability* of ROS nodes because they can be tested in isolation by replaying events and data stored in compatible ROS bags.[5]

Using standardized message formats also eases the integration of new sensors and hardware devices in the system. Indeed, many manufacturers of robotics hardware are supporting ROS out of the box and the ROS nodes running on top of their shipped devices typically publish/subscribe to topics conforming to message formats belonging to the `common_msgs` ROS stack.

Finally, as any other software, the definitions of standardized message formats are also subject to change (*e.g.,* the GitHub repository for `common_msgs` has more than 700 commits). It is important to make the nodes of the system under development as independent as possible from (possibly evolving) message formats, especially when working on long-lived or difficult-to-update systems.

N1 – Group nodes and interfaces into cohesive sets, each with their own responsibilities and well-defined dependencies. The software architecture of ROS-based systems is getting more and more complex and can easily result in an intricate network of hundreds of interdependent nodes [30]. Such complexity can lead to technical debt, lock-in to specific ROS packages and difficulty in extending the system.

Developers should take special care of the dependencies between packages and how ROS nodes communicate with each other. This task can be carried out with different degrees of rigor, depending on the criticality of the system. Having a clear overview of the ROS nodes and their relationships allows developers to assign clear responsibilities to ROS nodes and make well informed decisions when the system will need to evolve. As an example, the architecture of the ROS stack for the Niryo One[6] manipulator arm is designed around the following five layers.

- External communication: high-level interfaces, both for other ROS software and non-ROS software;
- Command: receives raw commands, validates them, and calls the layers below for executing them;
- Motion planning: finds inverse kinematics and builds a path for the robot, based on the `Moveit` package;
- Control: joint trajectory controller and other controllers based on the `ros_control` package;
- Hardware: all the drivers interacting with the arm hardware.

Having a well defined organization of node responsibilities allows developers to (i) concisely reason about the workflow of the system and (ii) avoid reinventing the wheel by reusing well-maintained and tested packages such `Moveit` and `ros_control`.

B1 – The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at runtime. ROS considers nodes as black-box components and does not prescribe any specific behavior to nodes *per se.* On the one hand this level of flexibility provides great freedom to developers, but on the other hand it impacts node testability, reliability, and maintainability. When dealing with stateful ROS nodes, developers should treat their internal lifecycle as a first-class concern of the system. As a base case, node lifecycles can be included in the documentation of a ROS package so that third-party developers and users know in advance the behavior they can expect from the nodes and properly interact with them (e.g., by sending messages in the correct order). Having a well-defined internal lifecycle of a ROS node can also help in terms of *testability*, *e.g.,* by guiding the development or even the automatic generation of test cases.

Some steps towards a precise definition of the lifecycle of ROS nodes are being taken by different players of the ROS ecosystem.

[3]ROS 2 supports C12 via *intra-process communication*, ROS 1 supports it via *nodelets.*
[4]ROS 2 partially supports guideline B1 via *managed nodes.*
[5]A bag in ROS is the file format used to record messages exchanged by ROS nodes and replay them for future inspection of the system.

[6]https://github.com/NiryoRobotics/niryo_one_ros

Table 2: Guidelines for Architecting ROS-based Systems. P=Provenance (GitHub|Questionnaire), U=Usefulness, M=Mentions.

ID	Architectural guideline	Quality requirements	P	U	M
Communications and networking (C)					
C1	Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.	MAINT, COMP, PORT	G, Q	71	9
C2	ROS nodes should be agnostic of the underlying communication mechanisms (*e.g.,* network protocols, deployment topology, etc.).	MAINT, PORT	G	63	3
C3	Include health information about both nodes and data in messages containing critical data. (*e.g.,* strength of GPS signal)	REL	G	61	8
C4	If the system is remotely distributed, constantly observe the status of the communication channels, hosts, and machines on the network.	REL	G	57	4
C5	Nodes that potentially produce/consume large amounts of messages should be configurable in terms of their publish/subscribe rates.	PERF, MAINT, REL	G	55	2
C6	Selectively limit the data exchanged between nodes to provide only the information that is strictly necessary for completing tasks.	PERF, SEC	G	54	3
C7	If different types of data are always sent/received together and must be synchronized, then either package them into a single message or subscribe to them using a time synchronizer filter.	REL	G	53	2
C8	Develop adapter components when data exchanged between nodes is not be fully compatible (semantically), incorrect, out-of-order, or redundant.	COMP, REL	G	42	9
C9	Use services when starting up robots (instead of publishing to topics) so that the status of the system can be checked before operation.	REL	G	37	1
C10	Publish empty messages when triggering atomic actions.	PERF	G	25	1
C11	Frequent messages should be exchanged either via services with persistent connections or via topic-based communication.	PERF	Q	-	-
C12	Run multiple nodes in a single process when the overhead due to inter-process communication is too high both in terms of frequency of messages and payload.[3]	PERF	Q	-	-
C13	Do not subscribe to (or unsubscribe from) topics where no messages will be published to (even temporarily).	PERF	Q	-	-
C14	Nodes should unsubscribe from topics when messages are not needed (even temporarily).	PERF	Q	-	-
C15	Publish messages (and perform the computation necessary to generate them) to a topic only when there is at least one subscriber to it.	PERF	Q	-	-
Node responsibilities within the system (N)					
N1	Group nodes and interfaces into cohesive sets, each with its own responsibilities and well-defined dependencies.	MAINT, PORT	G, Q	63	3
N2	Each ROS package should be responsible for one and only one feature of the system or robot capability and provide a well-defined interface.	MAINT	G	63	9
N3	Decouple nodes with responsibilities that naturally work at different rates and use different rates for different purposes.	PERF, REL	G	61	7
N4	By design, limit unnecessary computationally-heavy operations by carefully analyzing the execution scenarios across ROS nodes.	PERF, EN	G, Q	59	6
N5	Transform data only when it is used, for efficiency in terms of computation and bandwidth.	PERF, MAINT	G	58	1
N6	Each single node should also be runnable in isolation.	MAINT	G	55	2
N7	Provide dedicated nodes for doing introspection and querying the lower levels of the system.	PORT	G	41	3
N8	Use a dedicated node to store and represent globally-relevant data (*e.g.,* the physical environment where the system operates) and use it as the single source of truth for all the other nodes in the system.	PERF, REL, SAFE	G	38	5
N9	Keep the number of nodes as low as possible to support the basic execution scenarios and then extend the architecture for managing corner cases.	MAINT	Q	-	-
N10	Take full advantage of existing packages in the ROS ecosystem and create your own package only when it is strictly needed.	MAINT	Q	-	-
Internal behavior of the nodes (B)					
B1	The behavior of each node should follow a well-defined lifecyle, which should be queryable and updatable at run-time.[4]	REL, MAINT, PORT	G	50	4
B2	The spinning rate for nodes should be configurable so that they can operate according to available computational resources.	PERF	G	47	4
B3	If a node is stateful and its behavior strongly depends on time and message arrival order, specify the message protocol expected by the node.	REL, MAINT, USAB	G	45	2
B4	ROS nodes should be stateless and their behavior should not depend on previous operations or received messages.	REL, MAINT, USAB	G	30	1
B5	Nodes with configuration errors should fail explicitly at launch time.	REL	Q	-	-
B6	If a node is computationally expensive, then ensure that it only executes when other nodes are subscribed to a topic that it publishes to (unless topic latching is enabled).	PERF	Q	-	-
Interface to external users and third-party developers (I)					
I1	Assign meaningful names to architectural elements (*e.g.,* nodes, topics, messages, services) and group them by adopting standard prefixes/suffixes.	MAINT, USAB	G, Q	71	4
I2	When possible, core algorithms, libraries, and other generic software components should be ROS-agnostic.	PORT, MAINT	G, Q	68	1
I3	Expose a single ROS node with interfaces for third-party users for the most common use cases.	USAB, MAINT	G	61	7
I4	Systems interacting with other non-ROS systems should provide two types of interfaces: a ROS-independent interface for the external systems and a ROS-based interface for ROS tools such as Rviz, Qt, etc.	USAB, COMP	G	60	4
I5	Identify variation points of the system in advance, and design the system to be extended by third-parties without modifying its core nodes.	MAINT	G	59	24
I6	Logging should be standardized across the project and follow well-defined guidelines.	MAINT	Q	-	-
Interaction with hardware and other lower-level entities (H)					
H1	Nodes interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system.	PORT	G	72	3
H2	Design ROS nodes to be as hardware-independent as possible.	MAINT, PORT, COMP	G	69	15
H3	Decouple ROS nodes from variations in the execution environment.	MAINT, PORT, COMP	G, Q	59	9
H4	The interface of nodes responsible for state estimation should (i) support an arbitrary number and different types of sensors and (ii) be able to combine the information provided by the sensors.	COMP	G	50	1
H5	If context-specific configuration is needed at run-time (*e.g.,* available hardware capabilities), then persist this configuration in a dedicated node to avoid having to recalculate it at run-time.	PERF	G	32	4
Safety-critical concerns (S)					
S1	ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.	REL	G	59	6
S2	Use different communication channels and different hardware depending on the criticality and real-time needs of the nodes.	REL	G, Q	49	1
S3	For real-time requirements, collect timestamps from as many sources as possible (*i.e.,* do not rely on ROS-based timestamps only).[5]	PERF	G	36	2
S4	Provide at least one globally-reachable node capable of receiving run-stop messages and stopping/resetting the whole system	REL, SAFE	G	34	3
Data persistence (P)					
P1	Avoid persisting raw data (*e.g.,* a full resolution video) if only part of it will be used.	PERF	G	56	1
P2	Pay special attention to race conditions when persisting data received from other ROS nodes within the system.	MAINT, PERF	G	52	1
P3	Use a dedicated node for persisting and querying long-term data and short-term data (*e.g.,* in the order of seconds).	MAINT, PERF	G	32	2

Firstly, ROS 2 is providing support for *managed nodes*,[7] *i.e.,* nodes whose lifecycle follows a known state machine containing four states, namely *Unconfigured*, *Inactive*, *Active*, and *Finalized*. The state machine of a ROS 2 managed node can be inspected and controlled by other nodes and launch files. Secondly, some ROS packages are already explicitly considering the lifecycle of their nodes in their APIs, even without ROS 2 managed nodes; for example, quoting the documentation of `ros-controls/ros_control`, *the lifecycle of controllers is not static. It can be queried and modified at run-time through standard ROS services provided by the controller_manager. Such services allow to start, stop and configure controllers at run-time.* This example also highlights that defining and enforcing the lifecycle of ROS nodes can enhance the system in terms of run-time configurability and reflection, which can be exploited for providing autonomous capabilities [2].

I1 – Assign meaningful names to components (*e.g.,* nodes, topics, services) and group them by adopting standard prefixes/suffixes. Similarly to how bad variable names impact readability, maintenance, and understandability of source code [7, 17], bad practices for naming ROS entities can also negatively impact the architecture of ROS-based systems. This problem is especially

[7] https://design.ros2.org/articles/node_lifecycle

severe in ROS-based systems because in ROS, topics and services are created programmatically by the nodes at run-time and their identifiers are just strings. In addition to the obvious issue related to understandability, bad naming practices can lead to run-time errors which are hard to detect, such as mistakenly unconnected nodes or unintentional and unwanted connections, incompatible interfaces, node misconfigurations, and bugs in launch files [34].

H1 – Nodes directly interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system. ROS provides a full ecosystem of robotic simulators supporting a wide variety of physics engines, robot models, etc. One of the advantages of using ROS-based simulators is *software in the loop simulation* (SITL) in which the ROS nodes executed during the simulation are exactly the same as the ones executed when using real robots.

The key point for developers is that having identical messaging interfaces from the hardware/simulation levels to the rest of the system localizes by design *the impact for switching between various simulators and hardware devices*, thus reducing future modification costs. Following H1 can also lead to a superior level of portability and testability of the system by design. For example, different simulation engines can be used (even in combination) for better exercising different aspects of the system.

A recurrent strategy to achieve H1 is to have a hardware abstraction layer. For example, in `ros-controls/ros_control` *the backbone of the framework is the Hardware Abstraction Layer, which serves as a bridge to different simulated and real robots. ... It also allows for integrating heterogeneous hardware or swapping out components transparently whether it is a real or simulated robot. ... Through these typed interfaces, this abstraction enables easy introspection, increased maintainability and controllers to be hardware-agnostic.*

S1 – ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors. Developers should be aware that hardware devices like sensors tend to produce data in bursts (*e.g.,* due to changes in the environment), degrade, and become less accurate over time [19]. Depending on the tasks being performed, developers should address this variability by design, *e.g.,* by setting up load balancing nodes for managing sudden bursts of sensor data or by making nodes resilient to gaps in sensor data. The latter is especially important for the reliability of state estimation nodes because faults in these nodes can lead to severe failures at the system level. The contributors of `hasauino/rrt_exploration` document their own solution for continuous estimation as follows: *each state estimation node in robot_localization begins estimating the vehicle's state as soon as it receives a single measurement. If there is a holiday in the sensor data ... the filter will continue to estimate the robot's state via an internal motion model.*

Being resilient with respect to the amount and frequency of sensor data can make the robotic system more scalable, even at runtime. For example, in `cra-ros-pkg/robot_localization` *each robot runs an instance of the local_rrt_frontier_detector ... Running additional instances of the local frontier detector can enhance the speed of frontier points detection, if needed.*

P1 – Avoid persisting raw data (*e.g.,* a full resolution video) if only part of it will be used. Depending on the provided capabilities, ROS systems can produce large amounts of data. For example, the ROS bag of a demo mapping session of the `rtabmap_ros` package can take up to 1.1Gb[8]. As informally confirmed by the ROS community[9], data persistence in ROS can lead to severe performance overheads at run-time.

Even though it can be tempting to persist all raw data produced by the system (*e.g.,* for subsequent inspection or replay), developers should identify the subset of relevant data according to their future needs and selectively persist only that data. When recording ROS bag files, developers should avoid to record *all* topics within the system, rather only the most relevant ones for subsequent replays should be recorded. The same principle also holds for audio/video data. A clever mechanism dealing with audio/video data is implemented in `florisvb/multi_tracker`, where a "buffering node" *listens to the camera topic ... and saves the pixels and values for any pixels that change more than the specified threshold. This results in a dramatically compressed filesize relative to a full resolution video.*

5 DISCUSSION

In this study we mined ROS open source software repositories and surveyed active developers to elicit quality requirements and guidelines that impact the software architecture of robotics systems. We found discussions relating to ten different quality requirements. The top three quality requirements were maintainability, performance, and reliability. While we cannot say for certain, we suspect that maintainability is important because the aim of the ROS community (and open source software more generally) is to build software that is general and can be used in many contexts. This is further supported by the observation that many survey participants highlighted the importance of documentation and ease of source code understanding of the packages. Performance is important because many of the computations done by robots (e.g., computer vision, planning, navigation) are computation- and data-intensive and so challenge the computing resources typically available on robots (especially mobile autonomous robots). Reliability is important because robots are cyber-physical systems that need to be reliable when interacting with physical environments.

More surprising is the sporadic discussion surrounding safety and security – in fact, security is a major factor in spawning variants of ROS, such as SROS[10] and ROS-M.[11] We suspect that these qualities will gain importance as robots become more common and guidelines emerge for them. We also suspect that safety and security are currently of more concern in commercial robotics, an area that was outside the scope of this study. We were also surprised that energy was not a major concern, especially when robots are autonomous and have restricted energy supplies.

Because the guidelines have been sourced from an examination of open source projects and surveys to participants of those projects, we do not make any claims that we have elicited all guidelines associated with developing ROS-based systems. From Table 2, out of the 49 guidelines, 32 came uniquely from open-source repositories, 10 came uniquely from surveys, and 7 came from both sources. Therefore, we can say that the GitHub is a good source for guidelines. Interestingly, repositories on GitLab and BitBucket were filtered out in Phase 1 of the study - after the first nine steps we had 0/46

[8] http://wiki.ros.org/rtabmap_ros
[9] https://answers.ros.org/question/266095/rosbag-record-performance-issues/
[10] http://wiki.ros.org/SROS
[11] https://rosmilitary.org/

GitlLab repositories included and 23/527 BitBucket repositories. These final ones were discarded after manual selection because they did not satisfy the inclusion criteria.

We believe that these guidelines are generally applicable, but they should be followed in context. For example, C2 and C4 appear to be in conflict – if a developer wants to observe the status of hosts on the network per C4, the ROS node responsible for doing this must be aware of the network topology, contradicting C2. An architect should be aware of this and resolve it, for example, by making C4 generally applicable and isolating the functionality for C2 into a separate node. Similarly, data may need to be distributed if network connections are unreliable, counter to N2. As with all architectural guidelines, architects must understand the trade-offs that must be made in context.

6 THREATS TO VALIDITY

This section discusses the main threats to the validity of our study and the countermeasures we applied for mitigating them.

External validity. ROS-based projects hosted on GitHub, Gitlab, and Bitbucket may not be representative of the state of the practice of ROS-based development. From an inspection of the obtained dataset, the projects are highly heterogeneous in terms of number of contributors, number of commits, etc. Also, we performed a strict search and selection process when building the dataset of repositories, making us reasonably confident that irrelevant projects were not considered in subsequent phases (*e.g.,* toy or demo projects). Therefore, due to the high heterogeneity of the dataset and the strict quality assessment we performed, we do not deem this as a major threat to external validity. Even though we contacted *all* contributors to the repositories in our dataset, they still might not be representative of the whole roboticists population. This potential bias is reasonably avoided because participants exhibited a good level of heterogeneity in terms of type of experience, number of contributed ROS packages, and primary motivation for using ROS.

Internal validity. This study has been conducted by adhering to well-established guidelines in software engineering [20, 31, 35]. The replicability of the study and independent verification of its findings are ensured by documenting each phase of the study in a publicly available research protocol.

Conclusion validity. The qualitative analysis for answering both RQ1 and RQ2 is based on the manual inspection and categorization of several repositories and text fragments, potentially leading to subjective results. We mitigated this potential threat to validity by carefully following the content and thematic analysis methodologies and involving at least three researchers, with jointly discussed disagreements and conflicts managed by a fourth. Moreover, the 39 guidelines emerging from Phase 1 have been scrutinized by 77 independent roboticists and the additional 10 emerging from Phase 2 were directly proposed by roboticists working on real projects. In order to be sure about the quality of our guidelines, we collected feedback about the final set of 49 guidelines from two volunteers working on robotic systems for several years.

Construct validity. It is important that the mining pipeline for searching and filtering ROS-based projects from the code hosting platforms is implemented and configured correctly. We mitigated this potential threat to validity by carefully designing the whole pipeline (see Section 3.1.1), by testing each component of the pipeline in isolation via subsets of data for which we knew already the expected outcomes, and by making the implementation of the whole pipeline publicly available in the replication package.

7 RELATED WORK

Studies on ROS. Because ROS is under active development, is a de facto standard for development of robots, and comprises many open-source components, software engineering researchers are increasingly using it as a source for research. A number of researchers have tools and analyses that extract architectural structures from ROS systems through static analysis of source files and configurations. For example, HAROS [30] and other work described in [27, 34] can generate the architecture and perform architectural consistency checks such as detecting communication errors (*e.g.,* nodes subscribing to unpublished topics). This body of research focuses on reconstructing architectures of existing systems rather than trying to understand what architectural patterns or guidelines could be used to achieve particular quality requirements.

To design good quality robotic software in ROS, the work in [15, 18] focuses on formally modeling ROS systems for real-time analysis and V&V. These do not really focus on the architecture of ROS systems, but could be used to generate or check the correctness of ROS systems. In terms of code quality, Pichler *et al.* examine open source projects with tools such as cpplint and XML validation to analyze the quality of packages that many other ROS systems depend on [25]. They caution that many projects that ROS systems depend on are not engineered with code quality in mind, but are developed just to demonstrate particular functionality. While not addressing quality requirements specifically, this result confirms the importance of having guidelines for designing ROS systems with safety, reliability, or availability in mind. These quality requirements are not much discussed in the existing repositories (see Section 4), which supports this observation. Similarly, [11] provides a visualization tool with run-time usage metrics that can be used after-the-fact to assess performance.

While this growing body of research examines the ROS ecosystem and provides information on its code quality and reuse, our approach goes a step further in eliciting (i) the state of the art about the architectural design and decisions of ROS systems, (ii) the quality requirements that are currently most concerning to ROS developers, and (iii) evidence-based architectural guidelines for achieving those qualities by both looking at code artifacts, targeting their official documentation and, more importantly, collecting the perspectives of roboticists working on real ROS projects.

Mining Architectural Information. Mining information related to software architecture from open source software repositories is not new. Other researchers have shown that such information can be derived from commit logs (either manually [33] or by applying machine learning [5]), issue trackers [21] as well as developer discussions in community groups like StackOverflow [32] or in chat groups [3]. In this paper, we have confirmed this by surveying developers to verify the usefulness of the architectural guidelines.

Identifying areas of the source code that implement specific architectural tactics (like heartbeat, audit, or checkpoint) has also been investigated [24]. In fact, these can be the areas of code that

change the most [23]. In future work we will use these techniques to extend our work to discover how existing tactics are used in robots and if there are any new domain-specific tactics that can also be used in other software domains.

8 CONCLUSIONS AND FUTURE WORK

In this paper we described a study to elicit evidence-based architectural guidelines for open-source ROS-based software for robots. We were interested in studying two research questions: *What is the state of the practice for architecting ROS based systems?* (RQ1), and *How to guide roboticists when architecting ROS-based systems?* (RQ2). For RQ1, we found that of the relevant repositories, 16.4% of them documented their architecture. In these cases, the most discussed qualities were maintenance, performance, and reliability. For RQ2 we derived 39 guidelines derived from the repositories, and 10 more from the survey. By surveying roboticists actively involved in these projects, we are confident that these guidelines are generally useful. These results can be used by roboticists to architect their systems to achieve particular quality requirements. Furthermore, architecture researchers can use this study as a baseline for understanding the architectural principles and practices of roboticists.

These results are a major foundation on which to base our own future work on determining architecture tactics for high-quality robot software, and thereby provide a solid engineering basis for developing robots in a future where they are ubiquitous. To achieve this, we plan to (i) explore how existing architecture tactics found in [4] and others could be applied in this domain, (ii) mine robot-specific tactics from other robotics sources and semi-automatically from source code using approaches based on [30, 34], and (iii) provide more comprehensive guidance for architecting high-quality robot software.

ACKNOWLEDGMENTS

This research is partially supported by the Dutch Research Council (NWO) through the OCENW.XS2.038 research grant. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center (DM19-0986), and on research sponsored by AFRL and DARPA under agreement number FA8750-16-2-0042. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the AFRL, DARPA or the U.S. Government. We would also like to thank Jarrett Holtz and Selva Samuel for their comments on parts of this paper.

REFERENCES

[1] ISO/IEC 25010:2011. 2011. Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. (2011). https://www.iso.org/standard/35733.html

[2] Jonathan Aldrich, David Garlan, Christian Kästner, Claire Le Goues, Anahita Mohseni-Kabir, Ivan Ruchkin, Selva Samuel, Bradley Schmerl, Christopher Steven Timperley, Manuela Veloso, et al. 2019. Model-based adaptation for robotics software. *IEEE Software* 36, 2 (2019), 83–90.

[3] Rana Alkadhi, Manuel Nonnenmacher, Emitza Guzman, and Bernd Bruegge. 2018. How do developers discuss rationale?. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. 357–369.

[4] Len Bass, Paul Clements, and Rick Kazman. 2012. *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional.

[5] Manoj Bhat, Klym Shumaiev, Andreas Biesdorf, Uwe Hohenstein, and Florian Matthes. 2017. Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach. In *Software Architecture*, Antónia Lopes and Rogério de Lemos (Eds.). Springer International Publishing.

[6] Davide Brugali and Erwin Prassler. 2009. Software engineering for robotics [From the Guest Editors]. *IEEE Robotics & Automation Magazine* 16, 1 (2009), 9–15.

[7] Simon Butler, Michel Wermelinger, Yijun Yu, and Helen Sharp. 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering (CSMR '10)*. IEEE Computer Society, Washington, DC, USA, 156–165.

[8] Federico Ciccozzi, Davide Di Ruscio, Ivano Malavolta, Patrizio Pelliccione, and Jana Tumova. 2017. Engineering the software of robotic systems. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 507–508.

[9] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. 2002. *Documenting software architectures: views and beyond*. Pearson Education.

[10] D. S. Cruzes and T. Dyba. 2011. Recommended Steps for Thematic Synthesis in Software Engineering. In *2011 International Symposium on Empirical Software Engineering and Measurement*. 275–284.

[11] W. Curran, T. Thornton, B. Arvey, and W. D. Smart. 2015. Evaluating impact in the ROS ecosystem. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 6213–6219.

[12] Pablo Estefo, Jocelyn Simmonds, Romain Robbes, and Johan Fabry. 2019. The Robot Operating System: Package reuse and community dynamics. *Journal of Systems and Software* 151 (2019), 226–242.

[13] Brian Gerkey. 2019. Why ROS 2? (2019). Retrieved 9/29/2019 from https://design.ros2.org/articles/why_ros2.html

[14] Georgios Gousios. 2013. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR '13)*. IEEE Press, Piscataway, NJ, USA, 233–236.

[15] R. Halder, J. Proença, N. Macedo, and A. Santos. 2017. Formal Verification of ROS-Based Robotic Applications Using Timed-Automata. In *2017 IEEE/ACM 5th International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*. 44–50.

[16] William Harrison, Anthony Downs, and Craig Schlenoff. 2018. The Agile Robotics for Industrial Automation Competition. *AI Magazine* 39, 4 (2018), 77.

[17] Johannes Hofmeister, Janet Siegmund, and Daniel V Holt. 2017. Shorter identifier names take longer to comprehend. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 217–227.

[18] F. Ingrand. 2019. Recent Trends in Formal Validation and Verification of Autonomous Robots Software. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 321–328.

[19] Pooyan Jamshidi, Javier Cámara, Bradley Schmerl, Christian Kästner, and David Garlan. 2019. Machine Learning Meets Quantitative Planning: Enabling Self-Adaptation in Autonomous Robots. In *2019 IEEE/ACM 14th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*.

[20] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21, 5 (2016), 2035–2071.

[21] D. M. Le, D. Link, A. Shahbazian, and N. Medvidovic. 2018. An Empirical Study of Architectural Decay in Open-Source Software. In *2018 IEEE International Conference on Software Architecture (ICSA)*. 176–17609.

[22] William Lidwell, Kritina Holden, and Jill Butler. 2010. *Universal principles of design*. Rockport Pub.

[23] M. Mirakhorli and J. Cleland-Huang. 2015. Modifications, Tweaks, and Bug Fixes in Architectural Tactics. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. 377–380.

[24] Mehdi Mirakhorli, Ahmed Fakhry, Artem Grechko, Matteusz Wieloch, and Jane Cleland-Huang. 2014. Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 739–742.

[25] M. Pichler, B. Dieber, and M. 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)*. 78–85.

[26] Marc Pichler, Bernhard Dieber, and Martin Pinzger. 2019. Can i depend on you? Mapping the dependency and quality landscape or ROS packages. In *Proceedings of the 3rd International Conference on Robotic Computing*. IEEE.

[27] R. Purandare, J. Darsie, S. Elbaum, and M. B. Dwyer. 2012. Extracting conditional component dependence for distributed robotic systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 1533–1540.

[28] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3. Kobe, Japan, 5.

[29] ROS.org. 2019. ROS Wiki Documentation. (2019). Retrieved 9/14/2019 from http://wiki.ros.org
[30] A. Santos, A. Cunha, and N. Macedo. 2019. Static-Time Extraction and Analysis of the ROS Computation Graph. In *2019 Third IEEE International Conference on Robotic Computing (IRC)*. 62–69.
[31] Forrest Shull, Janice Singer, and Dag IK Sjøberg. 2007. *Guide to advanced empirical software engineering*. Springer.
[32] M. Soliman, A. Rekaby Salama, M. Galster, O. Zimmermann, and M. Riebisch. 2018. Improving the Search for Architecture Knowledge in Online Developer Communities. In *2018 IEEE International Conference on Software Architecture (ICSA)*. 186–18609.
[33] Jan Salvador van der Ven and Jan Bosch. 2013. Making the Right Decision: Supporting Architects with Design Decision Data. In *Software Architecture*, Khalil Drira (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–183.
[34] T. Witte and M. Tichy. 2018. Checking Consistency of Robot Software Architectures in ROS. In *2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE)*. 1–8.
[35] C. Wohlin, P. Runeson, M. Höst, M.C. Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in Software Engineering*. Springer.