Mining Guidelines for Architecting Robotics Software

Ivano Malavolta[a], Grace A. Lewis[b], Bradley Schmerl[c], Patricia Lago[a,d] and David Garlan[c]

[a]Vrije Universiteit Amsterdam, The Netherlands
[b]Software Engineering Institute, Carnegie Mellon University, Pennsylvania, USA
[c]Institute for Software Research, Carnegie Mellon University, Pennsylvania, USA
[d]Chalmers University of Technology, Sweden

ARTICLE INFO

Keywords:
Software Architecture
Robotics
ROS

ABSTRACT

Context. The Robot Operating System (ROS) is *the* de-facto standard for robotics software. However, ROS-based systems are getting larger and more complex and could benefit from good software architecture practices.
Goal. We aim at (i) unveiling the state-of-the-practice in terms of targeted quality attributes and architecture documentation in ROS-based systems, and (ii) providing empirically-grounded guidance to roboticists about how to properly architect ROS-based systems.
Method. We designed and conducted an observational study where we (i) built a dataset of 335 GitHub repositories containing real open-source ROS-based systems, and (ii) mined the repositories to extract and synthesize quantitative and qualitative findings about how roboticists are architecting ROS-based systems.
Results. First, we extracted an empirically-grounded overview of the *state of the practice* for architecting and documenting ROS-based systems. Second, we synthesized a catalog of 47 *architecting guidelines* for ROS-based systems. Third, the extracted guidelines were validated by 119 roboticists working on real-world open-source ROS-based systems.
Conclusions. Roboticists can use our architecting guidelines for applying good design principles to develop robots that meet quality requirements, and researchers can use our results as evidence-based indications about how real-world ROS systems are architected today, thus inspiring future research contributions.

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 lives, 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 robotics software is the Robot Operating System (ROS) [75], a set of open source libraries and tools for developing modular robotics functions that communicate with each other in a loosely-coupled, multi-process, distributed environment. This promotes many ways of structuring robotics software, *e.g.,* offloading computation-intensive tasks to the cloud, or coordinating multiple robots in teams. 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 7286 papers that cite the seminal paper on ROS on the Google Scholar indexing system. Furthermore, a 2018 report [33] by the ROS community lists approximately 140 different types of robots available to the community on `http://ros.org`, including aerial, ground, industrial, and aquatic robots.

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 robotics software development from an art to an engineering discipline as robots become further commoditized requires developers to 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 proper interaction with the world; and robots need to be reliable, safe, and secure when they are performing functions critical to industry and infrastructure.

ORCID(s): 0000-0001-5773-8346 (I. Malavolta); 0000-0001-9128-9863 (G.A. Lewis); 0000-0001-7828-622X (B. Schmerl); 0000-0002-2234-0845 (P. Lago); 0000-0002-6735-8301 (D. Garlan)

In general, software engineering practices can provide great benefit to the engineering of robotics systems, similar to the benefits they provide to software-intensive systems in other domains such as automotive [124], or domains that intrinsically need diverse domain expertise such as scientific software [5] or astronomy. While for such domain-specific systems the importance of sound software engineering practices is well-understood, for robotics systems it is still emerging [44]. Yet, as confirmed by the IEEE Technical Committee for Software Engineering for Robotics and Automation[1], the synergy between Robotics and Software Engineering is strategic in both areas, where opportunities for building new systems, reducing their development cost, and enhancing their quality are emerging for both researchers and practitioners.

In particular, to understand how to achieve well-architected robots, we must first look at the body of knowledge that has already been developed around ROS. Examining how roboticists have engineered existing systems, we can derive guiding principles for architecting robotics software, as well as the quality requirements that concern them the most. Such guidelines could lead to a set of architectural tactics [7] for robotics software. Fortunately, because ROS is open source, it has encouraged the open source development of many robots and robotics components that we can study to find answers to these questions.

This paper addresses three research questions: *What quality requirements are considered when architecting ROS-based systems?* (**RQ1**), *How do roboticists document the software architecture of ROS-based systems* (**RQ2**), and *What guidelines are followed by roboticists when architecting ROS-based systems?* (**RQ3**). Our approach consists of two main parts: (1) mining publicly-available repositories on GitHub, GitLab, and BitBucket to uncover architecture documentation and guidelines related to architecture design or quality requirements of ROS-based systems, and (2) surveying developers who actively contributed to those repositories to determine the usefulness of these guidelines, as well as elicit additional guidelines directly.

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 *47 guidelines* and the *quality requirements* that these guidelines are concerned with; (3) A *validation of these guidelines* from 119 roboticists who were active committers to these projects; and (d) The *replication package* for the study. In [55] we reported on an initial set of guidelines. In this paper, we extend the contributions of that paper by (a) providing a more thorough description of the experimental process of the study, (b) including an analysis of the architectural documentation provided by these ROS projects, (c) including an analysis of the main reasons a repository of a ROS project does not contain architecture documentation, and (d) providing a comprehensive description of all guidelines elicited by the study, including examples of how they are applied in the context of real ROS-based systems, and discussing their main alternative implementations. Furthermore, these examples have been enriched through engagement with the ROS community, particularly the ROS Quality Working Group, as a result of publishing the initial study.

The **target audience** of this study includes (1) roboticists who want to apply good design principles to develop robots that meet quality requirements, and (2) architecture researchers who can use the guidelines as evidence-based indications about how real-world ROS systems should be architected, thus inspiring future research contributions such as automated architectural analysis.

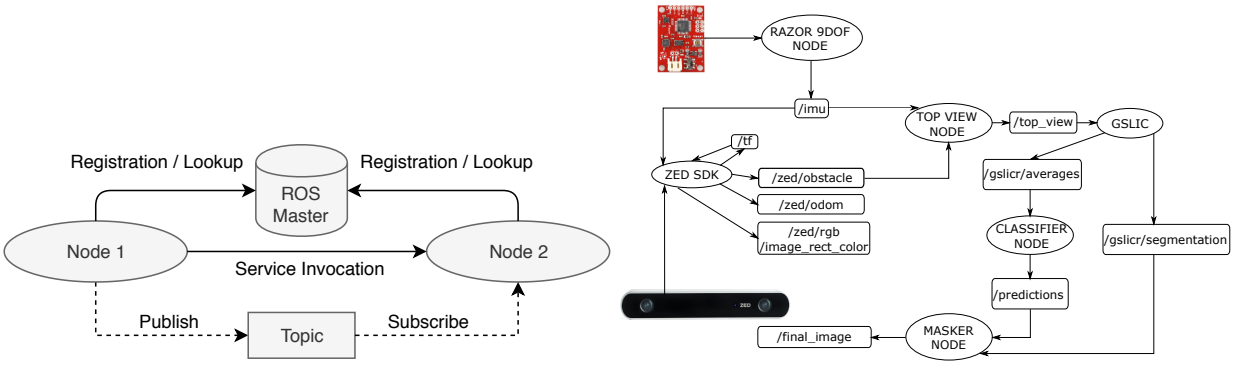
The remainder of this paper is organized as follows. We discuss the general software architecture style for ROS-based systems in Section 2. Section 3 illustrates the design of the study used to elicit the guidelines, including the selection criteria for repository and questionnaire participants. Repository demographics are shown in Section 4. We then present results, guidelines, and discussion for each research question in Sections 5, 6, and 7. Section 8 discusses the threats to validity of our study. We close with the related work in Section 9, and our conclusions and future work in Section 10.

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 [35]. ROS 1 is currently evolving into ROS 2 to address the 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 [83]. Nodes communicate with each other using a publish/subscribe model based

[1] https://www.ieee-ras.org/software-engineering-for-robotics-and-automation



(a) ROS Node Communication

(b) Example of a ROS Software Architecture - Adapted from the `IGVC-IITK/computer_vision` Project

Figure 1: ROS Architecture Communication Pattern with Example

on *Topics*, or a request/reply model based on *Service Invocations*, as shown in Figure 1a. Communication can also use *actions*, which are implemented as a combination of topics and services.

For ROS 1, 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 for the sake of brevity, such as nodelets and namespaces [75], but these 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 to start the ROS Master and all system Nodes, set parameters in the Parameter Server, and perform any other initialization required by the system [83].

Figure 1b shows the architecture for a computer vision subsystem available in our dataset (see Section 3.1). 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, likely due to the visualization provided by the commonly-used (`rqt_graph`) ROS package.

From a code perspective, elements of a ROS-based system are implemented inside *ROS packages*, *i.e.,* a set of ROS nodes, configuration files, and their metadata and build dependencies [75]. Some systems use layered diagrams to show the relationship between ROS packages. Each layer represents a grouping of packages that offers a cohesive set of functionality, and each layer is only allowed to use functionality from the layer below, thereby promoting portability and modifiability [17]. Figure 2 shows an example for the Niryo One ROS stack [66]. The legend in the diagram indicates which packages are specific to the Niryo One system, which are third-party ROS packages, and which are hardware or external components. A further discussion of the current state of architecture documentation for the analyzed ROS-based systems is presented in Section 6, along with a recommended set of best practices. Additional sample architectures can be found in the replication package for this study (see Section 3).

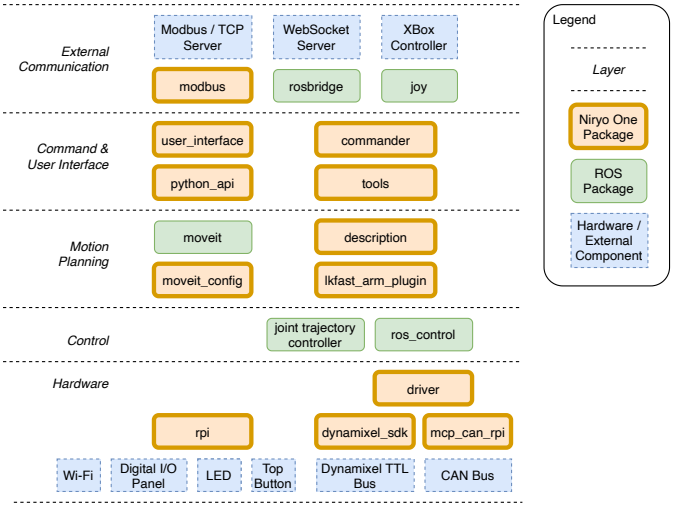


Figure 2: Example of a Layered Software Architecture for a ROS System – Enhanced from the `NiryoRobotics/niryo_one_ros` Project

The Niryo One example also highlights one of the key challenges of robotics software. Robotics is becoming a

software-intensive industry with each system growing massively towards having *several* software components, many of which are developed by independent development teams with different backgrounds and skills (*e.g.,* experts in control systems and automation, electronic engineers, mechanical engineers). Also, as stated in the H2020 Multi-Annual Roadmap for Robotics in Europe, in robotics "usually there are no system development processes (highlighted by a lack of overall architectural models and methods)" [40]. This results in increasing costs, longer time-to-market, and lack of project control. The software engineering community has faced variations of these challenges in other domains such as mobile embedded systems [57] and service-based systems [22]. One of the most critical success factors for the design and development of these systems is to raise the level of abstraction by focusing on their software architecture [110, 7].

3. Study Design

This study has two main goals: (i) to characterize the state-of-the-practice in terms of targeted quality attributes and architecture documentation in 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 to the following research questions.

RQ1 – *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 and functional issues of robotics systems, while neglecting other crucial quality requirements, such as maintainability, energy efficiency, and safety [15, 12]. With this research question we aim at objectively assessing this phenomenon in the context of real robotics projects.

RQ2 – *How do roboticists document the software architecture of ROS-based systems?* The benefits of architecture documentation have been largely discussed in the past [17]. However, given the wide adoption of agile methods [42], the extent to which 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.

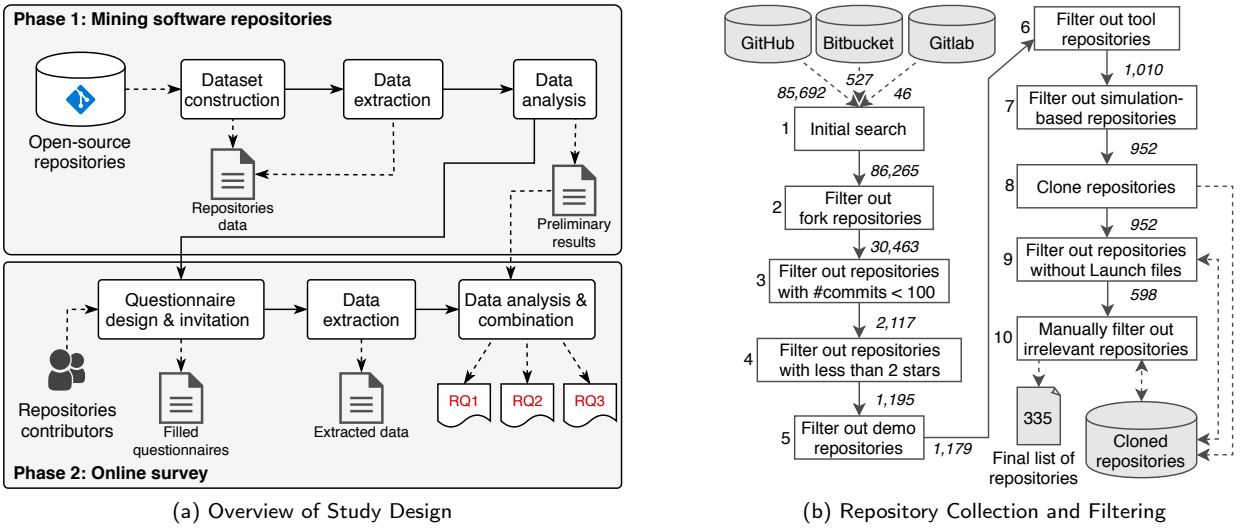
RQ3 – *What guidelines are followed by roboticists when architecting ROS-based systems?* To answer this question we provide a catalog of guidelines for architecting ROS-based systems. Roboticists can use the catalog as a source of actionable guidance for architecting their next robotics system. Researchers can use it as a solid foundation for developing new scientific contributions, *e.g.,* techniques to detect violations of the guidelines, or even to automatically enforce them in new or existing ROS-based systems.

Figure 3a presents the overview of the study design. It follows a *mixed method research methodology* involving two sequential phases. In the first phase we conduct a quantitative and qualitative investigation targeting data mined from repositories of open-source ROS systems; this phase aims at answering RQ1 and RQ2 and at building a preliminary set of architecting guidelines (RQ3). In the second phase we conduct an online questionnaire with roboticists contributing to open-source ROS systems and aim at (i) assessing the usefulness of the preliminary guidelines extracted from the software repositories, (ii) complementing the set of guidelines with additional ones elicited directly from practitioners, (iii) collecting further information about the quality requirements they consider when working on a ROS-based system, and (iv) understanding the main reasons why some repositories do not contain architectural documentation. To allow independent replication and verification of the study, we provide a full *replication package* [56] 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 [29], 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 3b shows the steps we carried out in building the dataset for our study, with the number before each step indicating the number of repositories that were filtered/produced by the prior step. For the initial search (step 1) we used (i) *rosmap*, a dependency analysis tool for ROS that can also identify ROS-based repositories in GitHub, BitBucket, and Gitlab [71] and (ii) *ghtorrent*, a widely-used queryable



(a) Overview of Study Design

(b) Repository Collection and Filtering

Figure 3: Study Processes

mirror of GitHub [38]. From the resulting 86,265 results we filtered out fork repositories in order to avoid duplication (step 2). Then, we excluded repositories with less than 100 commits (step 3) and less than 2 stars (step 4) to avoid inactive or non-maintained projects [49]. Because in this study we target 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; for each of these steps we defined a set of regular expressions (*e.g.,* `*[demo | course | thesis | exam]*` for step 7 in Figure 3b) and a repository was discarded if either its name, description, or unique identifier matched any regular expression. In order to avoid false positives, we manually checked all the repositories discarded in each of the three steps and refined its set of regular expressions until no relevant repository was discarded. In step 8 we locally cloned all 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. This is also confirmed by the fact that launch files are the main input of existing architecture extraction approaches for ROS systems [122, 108].

Table 1
Selection Criteria for the Repositories

ID	Description
I1	Contains a ROS-based system defined as either a ROS application or a ROS application framework
I2	Contains code which can be physically deployed either on a robot or on a general purpose host
E1	Contains only a student project, submission to a robotics competition, or support materials for tutorials, workshops, university courses, exams, competitions, etc.
E2	Contains only data, model definitions, simulators, or plugins for simulation (*e.g.,* Gazebo)
E3	It is clearly or admittedly incomplete or deprecated
E4	Contains only collections of potentially useful snippets of code, examples, or templates for other projects
E5	It is clearly or admittedly only about experimenting with ROS or a demo
E6	Contains only testing artifacts (*e.g.,* test cases)
E7	Contains only a ROS driver or an interface layer between ROS and hardware
E8	Contains only a software development tool (*e.g.,* an inspector for ROS nodes)
E9	Contains only a wrapper around an existing library for using it within a ROS-based system
E10	It is clearly or admittedly a duplicate of an already considered repository
E11	Its readme file is not written in English

Finally, in step 10 we performed an in-depth quality assessment of the 598 remaining repositories. Guided by the systematic literature review methodology [123], we *manually analyzed* each potentially relevant repository and selected it according to a set of well-defined inclusion and exclusion criteria (see Table 1). A repository was selected if it satisfied *all* inclusion criteria and it was discarded if it satisfied *any* of the exclusion criteria. Three researchers were involved in this step and emerging conflicts were resolved by a fourth researcher.

Table 2: Descriptive Statistics of the Repository Dataset

	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

* SD = standard deviation, CV = coefficient of variation

The final repository dataset is composed of 335 GitHub repositories. Their descriptive statistics are reported in Table 2. The dataset is quite heterogeneous in terms of commits, contributors, launch files, and other metrics. 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

In this phase we manually analyzed each of the 335 repositories to extract repository demographics and the information needed for answering our RQs. We inspected the following data sources: (i) all the markdown files stored in the repository (*e.g.,* readme and change log file of the project), (ii) all the external web resources and documents linked in the repository (*e.g.,* links in the description of the repository or in its readme), and (iii) all documents directly stored in the repository. For each identified data source we conducted iterative content analysis sessions with open coding [53]. 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 accordingly (see Section 4 for details). As part of this initial analysis we identified that 115 of the 335 repositories had full or partial architecture documentation, which were the ones that were further analyzed to obtain the preliminary results for Phase 1.

For answering RQ1 we conducted collaborative content analysis sessions that targeted architecture documentation fragments. We used the quality requirements of the ISO/IEC 25010 standard [46] as the initial set of codes. The resulting quality requirements are presented in Section 5.

For answering RQ2 we conducted iterative open coding sessions to categorize view types and elements used in the 115 repositories with architecture documentation. The resulting architecture documentation details are described in Section 6.

Finally, for RQ3 we relied on *thematic analysis* [20] to synthesize architecting guidelines (*i.e.,* the themes) from the 115 repositories with architecture documentation. We chose thematic analysis because the architectural information we extracted in the fragments was strongly dependent on project- and system-specific characteristics, and thematic analysis copes well with context-dependent data points [117, 20]. We organized our thematic analysis according to the process recommended by Cruzes and Dyba [20], which is divided into five main steps:

1. *Extract data*: two researchers manually inspected *all* documentation artifacts in the 115 repositories with architecture documentation and collaboratively collected 142 unique text fragments where architecturally-relevant concerns are discussed (*e.g.,* presence of integrator nodes, organization of the system into layers, etc.). When needed, unique text fragments where also enriched with free-text information about their context (*e.g.,* whether a fragment refers to a standard, if the whole project is oriented towards video streaming, etc.). Data extraction was carried out as a paired activity to (i) mitigate the risk of missing potentially-relevant text fragments and (ii) foster discussion and reflection about the identified fragments from the beginning of the thematic analysis.

2. *Code data*: two researchers systematically identified and coded relevant concepts, discussions, and solutions from all text fragments extracted in the first step. In this activity the researchers followed an inductive approach, *i.e.,* new and already-established codes were under constant comparison where codes were refined and new codes were added iteratively, as they emerged from the text fragments. This step resulted in a total of 190 codes.

3. *Translate codes into themes*: two researchers examined the codes and organized them into meaningful subsets. Then, for each subset a precise guideline was formulated. We focused on making the phrasing of the guidelines (i) representative of its corresponding codes, (ii) not overlapping, and (iii) actionable to be readily usable by roboticists in the field. The phrasing (and semantics) of each guideline underwent a rigorous scrutiny of two additional researchers, leading to a further refinement of the set of the guidelines. This activity resulted in a total of 39 guidelines.

4. *Create a model of higher-order themes*: four researchers discussed the obtained guidelines (and their corresponding codes) and explored relationships between them in order to identify families of guidelines related to the same concern about the architecture of ROS-based systems. As a result, 7 families of guidelines were identified, namely: Communications and networking (C), Node responsibilities within the system (N), Internal behavior of the nodes (B), Interface to external users and third-party developers (I), Interaction with hardware and other lower-level entities (H), Safety-critical concerns (S), and Data persistence (P).

5. *Assess the trustworthiness of the synthesis*: each of the identified guidelines underwent the scrutiny of 119 roboticists who contributed to the 335 repositories of our dataset. Section 3.2 reports how we integrated the results of this step. Moreover, as a result of distributing our previous study on the architecting guidelines [55], we discussed the extracted guidelines with (i) the ROS community via a post on the ROS Discourse platform[2] and (ii) the ROS Quality Working Group[3]. The discussions with roboticists allowed us to (i) assess and further refine the phrasing of the guidelines to correctly capture the essence of architecting ROS-based systems and (ii) expand the description of some of the guidelines with additional insights provided by roboticists.

In total, four researchers were involved in the thematic analysis, which led to the definition of a preliminary set of 39 architecting guidelines, organized over a set of 7 families (see Section 7).

3.2. Phase 2: Online Survey

We administered an online questionnaire to validate the preliminary set of guidelines and obtain additional information related to our initial findings. The target population of the online questionnaire is composed of roboticists who worked on at least one real ROS-based system. We extracted all contributors of the 335 repositories. Then, in order to increase the quality of the provided answers and ensure that the context of the project was still fresh in participants' minds, we discarded all contributors who did not make any commits to the repositories in the last 12 months. The resulting sample is composed of a total of 808 roboticists distributed over 165 unique repositories. Among those, 520 developers contributed to 81 unique repositories with architectural documentation, whereas the rest contributed to 83 unique repositories which do not contain (or directly link) to architectural documentation.

3.2.1. Questionnaire design and invitation

In this phase of the study we followed well-established guidelines for questionnaire design [112]. As already stated, we designed the questionnaire to complement the results we obtained in Phase 1. In the questionnaire we target primarily RQ1 and RQ3 because they are the most context- and project-dependent information we extracted in Phase 1, so we decided to further explore them directly with roboticists.

The questionnaire is composed of 9 questions organized into 5 sections. The first section provides brief guidance about how to fill out the questionnaire and asks about demographics. The second section 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 ranging from *Absolutely useful* to *Absolutely not useful* and a *Don't know* option. Additional guidelines can be proposed by roboticists in a following open-ended question. The third section of the questionnaire asks about the top 3 quality requirements the participant considered when working on their last ROS-based system. The fourth section is shown only to participants who contributed to repositories that do not contain (or directly link) to architectural documentation and it asks about why architecture documentation is not present in the repository. Finally, the last section is optional and asks general comments and suggestions 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 119 participants, yielding a 14.7% response rate. Participants tend to have multiple years of *experience with ROS* (min=1, max=12, mean=4.7, median=4, SD=2.63) and multiple *contributions*

[2] https://discourse.ros.org/t/guidelines-on-how-to-architect-ros-based-systems
[3] https://discourse.ros.org/c/quality

to ROS packages (more than 10 packages=45/119, between 6 and 10=14/119, between 2 and 5=52/119, and 1 package=8/119), and their *primary motivation for using ROS* is academic (56/119), professional (49/119), others/hobby (6/119), or a combination (7/119).

3.2.2. Data extraction, analysis, and combination

With regards to RQ1 (*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. The goal is to assess the level of agreement between the quality requirements *mentioned in architecture documentation* and the ones *considered important* by roboticists.

For RQ2, four researchers performed thematic analysis on the answers provided by participants when asked about why their repositories do not seem to contain architectural documentation. This activity led to the identification of 9 main reasons for practitioners not documenting the software architecture in the context of ROS-based systems.

Concerning RQ3 (*i.e.,* the guidelines), we first collected the usefulness values for each of the preliminary 39 guidelines, and reported them in the form of a stacked bar chart. The collected usefulness values provide an indication of the applicability of the guidelines in future ROS projects. Second, 32 participants discussed additional guidelines in the follow-up open question; four researchers performed thematic analysis on those 32 answers and merged the resulting guidelines with the ones that emerged in Phase 1. The analysis was carried out by following the same steps described in Section 3.1.2, with the only difference being that data is extracted from participants' answers instead of documentation fragments. This additional analysis led to the identification of 8 new guidelines, thus leading to the final set of 47 guidelines for architecting ROS-based systems.

In our replication package, all originating data items, codes, extracted data, categories, and analysis results are fully reported and traceable, to ease third-party verification and replication.

4. Repository Demographics

4.1. Types of Implemented Systems

Figure 4a shows the ten 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 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%).

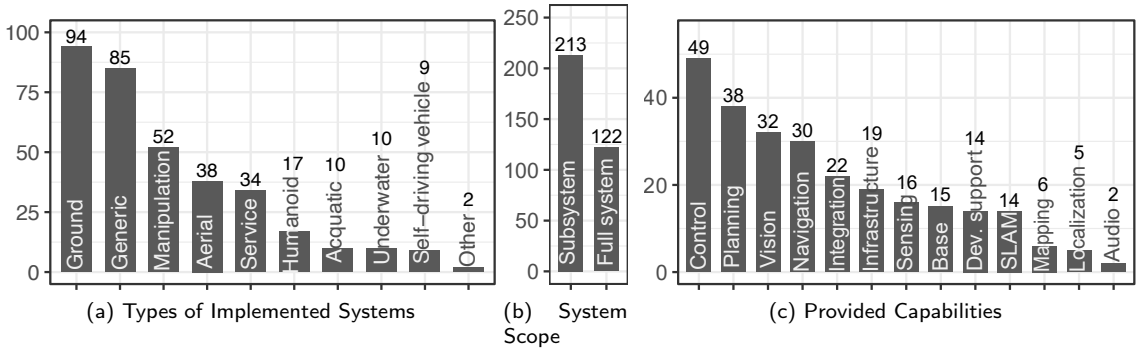


Figure 4: Categorized Architecture-Relevant Demographics

4.2. System Capabilities

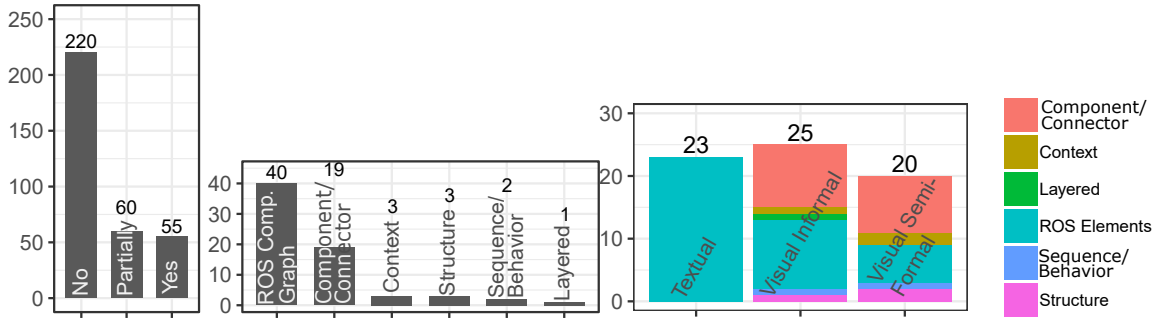
We studied two aspects of the systems: scope and provided capabilities. For scope, as shown in Figure 4b, we distinguished between *Subsystems* and *Full Systems*. Subsystems refer to repositories that contain the implementation of a component that is meant to be used in the context of a larger system, 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 4c 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%).

4.3. Presence of Architecture Documentation

We examined the repositories to find architecture documentation, as explained in Section 3.1.2. As shown in Figure 5a, 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 some (16.4%) of the projects have a documented architecture (*i.e.,* the system is described in terms of nodes, services, topics, and their configuration). The 55 projects with a documented architecture were further analyzed to answer RQ2 (details in Section 6).

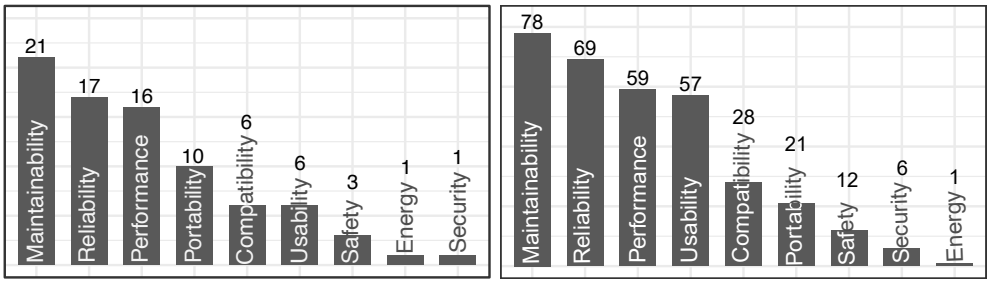


(a) Documentation Present (b) View Types (c) View Depiction vs. View Types

Figure 5: Architecture Documentation

5. Quality Requirements (RQ1)

For the 115 repositories containing full or partial architecture documentation, Figure 6a shows the number of guidelines where a specific quality requirement was mentioned (details on how these were obtained are in Section 3.1.2). The most frequently mentioned quality requirement is *Maintainability* (in 21 guidelines). This phenomenon can be explained by the fact that robotics is becoming a software-intensive industry; developing even simple robotics applications requires integrating several components with intricate interdependencies, making it extremely challenging for a single group of roboticists to effectively maintain and design the software stack of a robotics system. *Reliability* (in 17 guidelines) and *Performance* (16) are also mentioned relatively frequently by practitioners, whereas fewer mentions were found for *Portability* (10), *Compatibility* (6), *Usability* (6), *Safety* (3), *Energy* (1), and *Security* (1).



(a) Number of Guidelines Mentioning Each Quality Requirement (b) Number of Participants that Mentioned Each Quality Requirement in the Top-3

Figure 6: Quality Requirements Mentioned

Figure 6b shows the number of participants that indicated the reported quality requirement as the one considered the most when working on ROS-based systems. Similar to the results in Figure 6a, the top three quality requirements are *Maintainability* (78 participants), *Reliability* (69), and *Performance* (59). The next considered quality requirement is *Usability* (57), which appears in a much higher position than in Figure 6a. This result is not surprising, given that the goal of open-source repositories is to get the community to use them. *Compatibility* (28), *Portability* (21), *Safety* (12), *Security* (6), and *Energy* (1) follow. The fact that security and energy are the lowest in both figures, however, is surprising, given that it is expected that in the future robots will operate in open and heterogeneous environments, intermixed with humans, and perform a broad spectrum of tasks in autonomy [11].

Discussion. 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 in general) is to build software that can be used in many contexts. This is further supported by the observation that many questionnaire participants highlighted the importance of documentation and ease of source code understanding. A well-known strategy for managing the complexity of the source code of software systems is to start from a minimum viable product and then to improve/expand it as needed; this point is clearly summarised by a questionnaire participant as follows: *"scale the complexity of the system accordingly to the knowledge and skills of the team. E.g. Don't start with a fully featured ROS package with everything abstracted as plugins configured on load via the parameter server via launch files that import other launch files. Scale the complexity as the team gets on the same page and understands the parts and why it is good (if it is) to implement it so."*

Performance is important because many of the computations performed by robots (*e.g.,* computer vision, planning, navigation) are computation- and data-intensive and therefore challenge the computing resources typically available on robots (especially mobile autonomous robots). Reliability is important because robots tend to interact with the physical environment and should provide assurances about their expected behaviour.

More surprising is the sporadic discussion surrounding security and safety. We expect that these qualities will gain importance as robots become more common and guidelines emerge for them. Related to security, it does not seem to be taken seriously enough in robotics practice, despite the urgency of the topic [18] and existing evidence of the ease with which ROS1 can be sabotaged by an attacker [23]. At the time of writing, there are some initiatives for improving the security of ROS-based systems, such as emerging security-oriented variants of ROS such as SROS [101] and ROS-M [105]. Moreover, the Robot Security Framework (RSF) has been recently released [119]; RSF is a methodology to perform systematic security assessments in robots, independent of whether the robots are based on ROS or not. Finally, it is important to note that ROS2 has been designed with security as a first-class priority; specifically, the new DDS-based communication middleware of ROS2 provides authentication, enforces access control restrictions on authenticated nodes, and supports encrypted communication. With respect to safety, roboticists can refer to a recent systematic literature study on safety for mobile robotics systems [11] which provides an evidence-based description of existing safety-related approaches that can be transferred to robotics projects. For example, the authors of [48] propose an approach that, starting from a training set defined a priori, automatically performs inference and monitoring of specialized invariants during the lifetime of ROS-based robotics systems. A second example is HAZOP-UML [39], a method that supports safety analysts in specifying dynamic UML models of the robot and in identifying hazards, recommendations, and hypotheses of possible deviations of the system from the specified models.

We are also surprised that energy is not perceived as a major concern, especially when robots are autonomous and have restricted energy supplies. Roboticists interested in energy efficiency can refer to a recent systematic literature study [115] providing an overview of the major sources of energy consumption for robotics software and a set of state-of-the-art techniques for improving its energy efficiency.

6. Architecture Documentation (RQ2)

To answer RQ2, we further analyzed the 115 projects that have a documented architecture, in order to understand how this documentation is currently provided by roboticists. We found a huge variation in what is considered architecture documentation and how it is represented. With respect to *View Depiction*, as shown in Figure 5c, we found 68 instances of architecture views across the projects, of which 45 (66.2%) are visual representations. Of these 45 views, 25 (55.6%) are *Informal*, meaning that there is no definition of the notation used, and 20 (44.4%) are *Semi-Formal*, meaning that there is some explanation of the notation used in the view or an accompanying legend. The remaining 23 (33.8%) views are *Textual* and describe only the ROS Elements of the system and their configuration. This minimal architecture

documentation is what we expected to find in open-source projects sharing ROS implementations, but as stated earlier, only 16.4% of total systems had at least this minimal documentation. An interesting finding is that 9 out of the 55 projects contained multiple views (between 2 and 4 views each). Even though not shown explicitly in the figures, these projects contained a total of 20 visual views and 3 textual views, which is 33.8% of the total views. Having multiple views is an architecture documentation best practice that will be further discussed in the discussion for this section.

With respect to *View Types*, as shown in Figure 5b, most views (40 (58.8%)) contain elements of the *ROS Computation Graph* (*i.e.,* ROS nodes, topics, services, messages, parameters), which is expected. The Component and Connector (C&C) view is the second most prevalent in the analyzed architectures, with 19 (27.9%) in total. A C&C view contains additional components that are not ROS Elements. Of these 19 C&C views, 10 have a *Visual Informal* depiction, which means that they have a legend to understand the different components and connectors (Figure 5c). Including legends with architecture views is another best practice that will be further elaborated in our discussion below. Other view types found, in much smaller numbers, are *Context* views (3) that show relationships of the system with external entities, *Structure* views (3) that show how code is organized in the system (*e.g.,* packages, modules), *Sequence/Behavior* views (2) that show control flow or data flow through the system, and finally a *Layered* view (1) in which each layer represents a group of system elements that offer a cohesive set of functionality (see Section 2 for an example of a layered view).

With respect to *View Elements*, as shown in Figure 7, the most-frequently documented elements are ROS elements: *Nodes* (57), *Topics* (56), and *Parameters* (26). Other elements included in the architecture views are *Other Software Components* (23) such as non-ROS software, and third-party libraries and drivers; *External System Elements* (13) such as hardware and third-party systems; *Packages* (12); *ROS Messages* (10); and *ROS Libraries* (4).

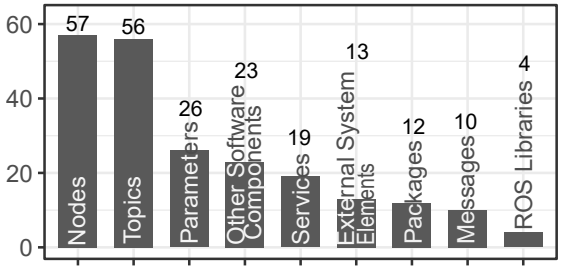


Figure 7: Documented View Elements

Finally, in our online questionnaire we have an additional question for the 288 unique contributors of the 220 repositories without a documented architecture. The question aims at understanding why those repositories do not contain a documented architecture. Out of the 42 received answers, 24 participants elaborated on their rationale for not documenting the architecture of their ROS-based system; the qualitative analysis of their answers allowed us to synthesize 9 main rationale statements, which are reported in Table 3. The remaining 18 participants stated that the architecture is either (i) documented externally, mostly without referring to any specific location (9 occurrences), (ii) documented in the official ROS Wiki, but the repository does not link to the corresponding Wiki page (6 occurrences), or (iii) kept private, *e.g., "the original architecture is kept in personal notes, but should really be uploaded alongside the repository"* (3 occurrences).

Discussion. The advantages of architecture documentation have been widely documented [17]. In particular for open-source ROS-based systems, a benefit of having a documented architecture is describing how the system works not only to potential users of the system, but also to other project contributors. In spite of that, we observe that software documentation is minimalist and often informal (hence prone to misinterpretation). Recommendations on architecture documentation for roboticists, based on the analyzed repositories, include:

- Visual is always better: When it comes to software architecture, the old English language adage "A picture is worth a thousand words" also applies. At a minimum, a graphical representation that includes system nodes and how these communicate with other nodes using topics and services is a much better communication form than a long list of nodes with a long list of published and subscribed topics, for example. Diagrams should be accompanied by text to explain aspects of the view that are not visually conveyed, such as message formats.

- Use legends or keys: Every architecture view should include a legend or key that explains the notation used. Elements and relationships of different types should be represented with different visual representations. For example, the enhanced architecture view in Figure 2 contains a legend that shows which elements are ROS packages, which elements are Niryo One packages, and which elements are external or hardware components. As a counter-example, the architecture in Figure 1b does not have a legend. One could infer that the ellipses are nodes, the rectangles are topics, and the lines between nodes and topics represent publish/subscribe relationships, because that is a commonly used notation. However, whether the line between the external elements and nodes is

Table 3
Rationale Provided by Participants for Not Documenting the Software Architecture

Rationale	Description	Representative answers (fragments)	Count
Lack of resources	The participant knows that documenting the architecture is a valuable activity, but they claim not to have time (or resources in general) for properly doing it.	*"[The repository] does not contain a correct documentation that is right, which there must be. Unfortunately I did not have much time to write down those documentations, which I'll do as soon as I get available time for it"*	5
Small-scale system	The participant considers the system to be not large/complex enough for justifying the effort of documenting its architecture.	*"Software architecture documentation should certainly exist in big and complex software systems or in systems where multiple people work on the same code, but in smaller systems with a single or a few developers trying to maintain it could be an overkill"*	5
The source code is informative enough	The participant states that the architecture of the system (i) is embodied in its source code, (ii) is self-explanatory by properly structuring the source code and strictly following code conventions, and (iii) good-enough documentation can be generated from annotated source code by using automatic documentation generators.	*"In many cases well-structured and documented code (i.e. API documentation and inline comments) is the best way to document open source software."* *"I would say most of the time the "software architecture" docs are in the source code (Doxygen comments, etc.)"*	4
The architecture is unstable	The participant does not see the added value of investing time and resources for properly documenting the architecture if it is continuously changing.	*"No architecture documentation because the repository is under heavy development and changes frequently"*	2
Not enough external contributors	The participant perceives the architecture documentation as a communication-only instrument, so if there are no/few external contributors there is no need to invest in producing architectural documentation.	*"I believe that few contributors expected others outside a small group to use the code, and thus saw documentation as less important"*	2
No clear advantage	The participant (and other stakeholders of the system) do not perceive the advantages of having a documented architecture for their system.	*"Haven't had time. Customer doesn't care about docs."*	1
Lack of knowledge	The participant does not know how to document the architecture of the system.	*"I am not sure how to create this. I documented what I knew how to in the Readme."*	1
The system has a standard structure	The system follows a standard structure, so the participant sees architectural documentation as a redundant activity with respect to known architectures in the robotics domain.	*"The ROS packages I worked [on] are nothing special rather than standard ROS nodes regarding software architecture"*	1
The architecture is not clear	The system is developed by contributors who are not trained in software engineering skills; the participant states that no explicit architectural reasoning is in place for the project.	*"Having such documentation is valuable for more complex packages and repositories, especially for developers and contributors. Unfortunately many ROS packages have been contributed by domain experts, without having a clear architecture in mind, let alone documented it."*	1

also to be interpreted as a publish/subscribe relationship is questionable. If these are different, the lines should be of a different type and the legend should indicate the relationship type expressed by the different line types.

- Limit views to one aspect of the system: Views should represent a single aspect of the system, whether that is interaction/integration with external system elements, mapping between packages and nodes, control/data flow, or any other aspect that is important to convey to the person that is going to use or extend the system. Include multiple views to show different aspects of the system instead of trying to convey everything in a single view.
- Ensure traceability between views: If using multiple views, it is important for elements to have the same names across views and for there to be consistency between views. For example, if a *Layered* view of a system is providing a structural view of packages and their dependencies, a *ROS Elements* view of the system should map elements to packages (when applicable) and follow the "allowed to use" constraints conveyed by the layered view.

Despite the relatively low sample size (N=24), the rationale provided by participants for not documenting the software architecture raises some interesting points for discussion. A non-negligible portion of participants acknowledge that documenting the architecture of their ROS-based software is a valuable activity, but either (i) they do not have the resources to do it properly or (ii) the scale/complexity of the system is not enough to require explicit architectural documentation. *Automation* is a possible solution for documenting the architecture of ROS-based systems with relatively low effort. The need for automation is also mentioned by one participant of the questionnaire, who reported that *"it would be nice to have a more automated system to create the software architecture diagram."* Approaches for the automatic production of architectural views of ROS-based systems are already available to the community. For example,

the `rqt_graph` package (http://wiki.ros.org/rqt_graph) provides a QT plugin for monitoring and visualizing the ROS computation graph of any ROS-based system; `rqt_graph` also supports the automatic annotation of each node/arc of the graph with metrics collected at runtime. The computation graph of a ROS-based system can be also extracted statically, *e.g.,* using the HAROS framework [108]. In this context, the usual tradeoffs between static and dynamic analysis of software systems apply, *e.g.,* static extraction can be easily integrated in CI/CD pipelines but it may be inaccurate and may miss dynamically-emerging topologies, whereas dynamic approaches require the system to be up and running to work, might require historical data, or to instrument the system under analysis.

Moreover, four participants stated that, with the proper coding conventions and documentation-oriented annotations in place, the source code of the system is informative enough for documenting the system under development. While we agree that indeed the architecture of a system is embodied in its source code, this claim is in contrast with available evidence about how architecture documentation is used in software projects, *e.g.,* architecture visualization is one of the most popular ways of making better design decisions and externalizing their rationale [117]. The latter observation is also confirmed by the fact that some participants stated that they did not document the architecture of their system because there is not a clear advantage to doing so, no known procedures for properly capturing the architecture of the system, or even because the architecture itself is not clear. As also confirmed in other independent studies on architecture visualization [8], software architecture researchers have the opportunity and duty to make practitioners aware of the advantages of architecture documentation. This objective can be attained by providing objective evidence about the effectiveness of architecture visualization (*e.g.,* via controlled experiments or industrial case studies), or at least to design and conduct such validation when proposing new visualization approaches for the software architecture of ROS-based systems.

7. Guidelines (RQ3)

Starting from the 115 repositories containing a documented software architecture and the 119 questionnaire responses, our thematic analysis synthesized a total of 47 architecting guidelines. Guidelines are organized around 7 families of architectural concerns manifested by developers working on ROS-based systems. The guidelines and architectural concerns are presented in detail in Sections 7.1 through 7.7.

We summarise the guidelines belonging to each family of architectural concerns as in Table 4, with six columns representing: (i) unique ID, (ii) description, (iii) *quality requirements* mentioned by roboticists in the context of the guideline, (iv) whether the guideline comes from fragments extracted from GitHub or the online questionnaire ("P" stands for Provenance), (v) level of *usefulness* (U_g), which is defined below, and (vi) how specific to ROS systems we consider the guideline ("S" stands for Specificity). As shown in Figure 8, the vast majority of questionnaire participants assessed the guidelines as (absolutely) useful. We define the level of usefulness U_g of a guideline g as

$$U_g = \sum_{p=1}^{119} uScore(g, p) \quad (1)$$

where $uScore(g, p)$ represents the assessment provided by a questionnaire participant p for guideline g according to the $[-2, +2]$ range (*i.e., absolutely not useful* = -2, *not useful* = -1, *don't know* = 0, *useful* = 1, *absolutely useful* = 2).

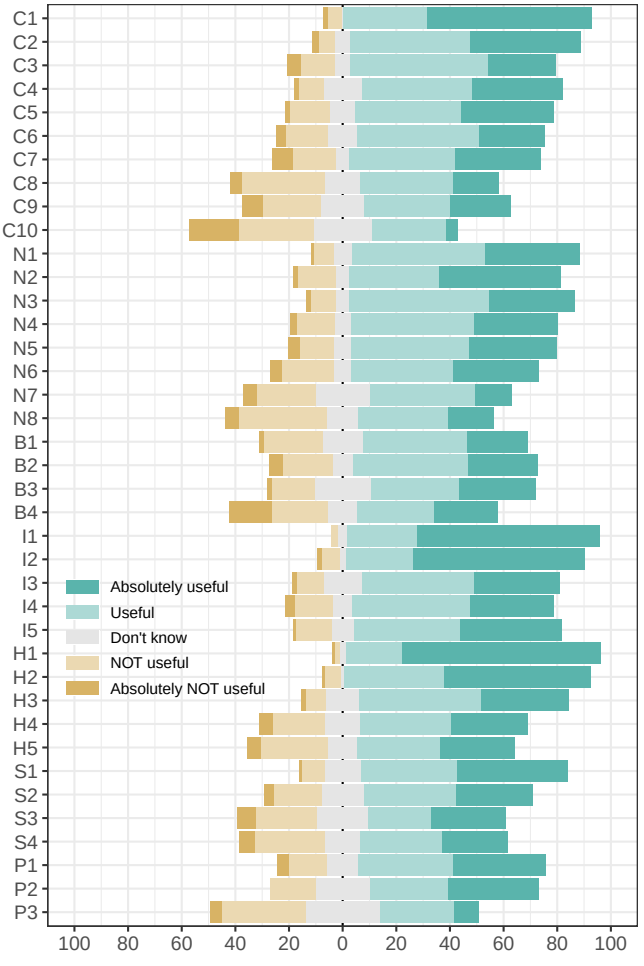


Figure 8: Usefulness of the Identified Guidelines (percentage)

The provenance column (P) can help in discerning whether the guidelines appear in the repositories, or whether they are guidelines that were deemed by experts to be good practice. In the specificity column (S), we categorize guidelines into one of the following three categories: *General* - the guideline is a general architecture principle that applies to many software systems, also outside the ROS realm; *Specialized* - the guideline specializes a general principle to ROS-based systems; and *ROS-specific* - the guideline is specific and applies only to ROS-based systems.

It is important to note that we are using the usefulness of the guidelines just as an overall indication of their applicability in practice (not a ranking) and for assessing the trustworthiness of our synthesis of the guidelines; we suggest to roboticists to carefully evaluate the applicability of each guideline depending on the characteristics of their specific system. Also, some guidelines (*e.g.,* C10, B4) tended to be considered not useful, but we decided to keep them because (i) we did not want to lose any potentially useful guideline, and (ii) a non-negligible number of questionnaire participants still assessed them as `useful` or `absolutely useful`.

We also collected the number of occurrences in which each guideline has been mentioned across the whole dataset – *Provenance (P)* column in Table 4. We notice that more than 4/5 of the guidelines (38/47) are mentioned more than once; among those 38 guidelines, 14 guidelines are mentioned at least in 5 cases, with some of them mentioned several times (*e.g.,* I5, H2). This is a good indication of the applicability of many of the guidelines for roboticists working on different types of robotics systems in terms of application domain, criticality, and provided capabilities. In the remainder of this section we will describe in detail each family of extracted guidelines.

7.1. Communication and Networking

In this family of concerns we group all guidelines related to the communication and networking aspects of a ROS-based system, such as types and content of messages sent/received by ROS nodes, network topology of the robotics system, frequency and rate of exchanged messages, when and how to publish/subscribe to ROS topics, etc. Table 4 presents the guidelines belonging to this family.

Table 4
Guidelines for Communication and Networking (P = Provenance: GitHub | Questionnaire; U_g = Usefulness score, S=Specificity: General | Specialized) | ROS-specific.

ID	Guideline	Quality Requirements	P	U_g	S
C1	Use standardized ROS message formats, possibly supporting also their legacy versions.	Maintainability, Portability, Compatibility	G(9) Q(2)	173	R
C2	ROS nodes should be agnostic of underlying communication mechanisms (*e.g.,* network protocols, deployment topology, etc.).	Maintainability, Portability	G(4)	138	S
C3	Include health information about both nodes and data in messages containing critical data. (*e.g.,* strength of GPS signal)	Reliability	G(8) Q(1)	94	S
C4	If the system is remotely distributed, constantly observe the status of the communication channels, hosts, and machines on the network.	Reliability	G(4)	114	G
C5	Nodes that potentially produce/consume large amounts of messages should be configurable in terms of their publish/subscribe rates.	Maintainability, Reliability, Performance	G(3)	107	S
C6	Selectively limit the data exchanged between nodes to provide only the information that is strictly necessary for completing tasks.	Performance, Security	G(3)	85	R
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.	Reliability	G(2)	86	S
C8	Develop adapter components when data exchanged between nodes is not compatible (semantically), incorrect, out-of-order, or redundant.	Compatibility, Reliability	G(10)	34	G
C9	Use services when starting up robots (instead of publishing to topics) so that the status of the system can be checked before operation.	Reliability	G(1)	48	R
C10	Use empty messages when triggering atomic actions.	Performance	G(1)	- 34	R
C11	Frequent messages should be exchanged either via services with persistent connections or via topic-based communication.	Performance	Q(2)	-	R
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.	Performance	Q(2)	-	R
C13	Manage topics to avoid unnecessary publishing and subscribing.	Performance, Usability	Q(3)	-	S

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 primitive and widely used data types, *e.g.,* geometric transformations, point clouds, battery levels. These packages are collected in two well-known ROS stacks

called `common_msgs` [86] and `std_msgs` [102]. When possible, roboticists should adopt these standardized message formats in order to increase the opportunities for *reuse* of ROS nodes. If commonly used message formats are exploited 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 allow 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`). For example, the documentation of the SPENCER multi-modal people tracking framework for mobile robots explicitly mentions the advantages of using standardized message formats: *"we highly encourage reuse of these [standard ROS] messages to benefit from our rich infrastructure of detection, tracking, filtering and visualization components! Existing detection and tracking algorithms can often easily be integrated by publishing additional messages in our format, or by writing a simple C++ or Python node that converts between message formats."* SPENCER is a project funded by the European Union in 2011-2016 targeting social situation-aware perception and action for cognitive robots (`spencer-project/spencer_people_tracking`) (In this paper, denotes `http://github.com`.)

Standardized message formats improve the overall *testability* of ROS nodes because they can be tested in isolation by replaying events and data stored in ROS bags that contain exchanged data (a bag in ROS is the file format used to record messages exchanged by ROS nodes that can be replayed for future inspection of the system). Indeed, if the system is using standard message formats (rather than ad-hoc messages), then it is more likely to find and reuse *compatible* ROS bags (*i.e.,* ROS bags containing data exchanged using standard message formats). As an example, the recent dataset provided by Lamarre *et al.* [51] is about planetary exploration emulated by a Clearpath Husky UGV across 6 independent runs over varied terrain and provides raw exchanged ROS messages about a colour omnidirectional stereo camera, a monocular camera, an IMU, a pyranometer, drive power consumption, wheel encoders, and a GPS receiver. Additionally, standardized message formats ease integration of new sensors and hardware devices into 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 with 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. Specifically, we suggest implementing ROS nodes so that (i) they use standard message formats for communicating with other nodes within the system, but also (ii) they internally define their own data abstractions and use those independently of the externally-used message formats. This proposed solution allows roboticists to maintain the business logic of their ROS nodes independently of the externally-used message formats, which can be subject to changes due to other factors outside their direct control.

C2 – ROS nodes should be agnostic of underlying communication mechanisms (*e.g.,* network protocols, deployment topology, etc.). The ROS communication model is one of its most acknowledged advantages because (i) it is flexible enough for nodes to communicate with each other independently of the specific host/platform they are deployed in, and (ii) it masks the complexity of the underlying communication infrastructure to roboticists.

However, because ROS does not restrict the internal behavior of nodes, roboticists may be tempted to circumvent the ROS communication abstractions and use other communication mechanisms, such as OS-level sockets or shared memory. In order to take advantage of the benefits of the ROS communication model, roboticists should avoid these workarounds as much as possible and follow the communication abstractions provided by ROS.

Similarly, roboticists should not make any strong assumptions about the network topology of the robotics system, *e.g.,* by assuming the specific host where the ROS master node is running or expecting that the nodes they are communicating with are deployed either remotely or locally. This improves the *maintainability* of ROS nodes because they can be tested and debugged in isolation, without the need to bring up the full networking infrastructure of the system. It also improves *portability*, because nodes can be deployed within the system without the constraint of having communication-related parameters hard-coded in their business logic. As an example, the documentation of the *m-explore* system (`hrnr/m-explore`) states that it builds on the `multirobot_map_merge` package (http://wiki.ros.org/multirobot_map_merge) because *"although the communication in the ROS network is always peer-to-peer, roscore is required for advertisement, enumerating topics and establishing communication. This might not be acceptable for exploration robots communicating over [an] unreliable link.* `multirobot map merge` *can work transparently with both configurations as it is not tied to any particular communication between robots, allowing a*

great flexibility."

C3 - Include health information about both nodes and data in messages containing critical data. (*e.g.*, strength of GPS signal). This guideline is mostly related to systems heavily relying on (i) the *reliability* of (some portions of) the system and (ii) the quality of the data exchanged between nodes. Examples include systems whose malfunction may lead to severe consequences in terms of user safety (*e.g.,* UAVs) or high-precision systems (*e.g.,* industrial manufacturing robots). The main point of this guideline is to allow ROS nodes to collect and check (i) the status of other nodes within the system and (ii) the quality level of received data.

This guideline can be implemented using different strategies. For example, the documentation of the ROS interface to the onboard SDK for DJI flying drones suggests to *"use* `gps_position` *for control only if* `gps_health >= 3`*"* (`dji-sdk/Onboard-SDK-ROS`). The COLA2 system for underwater vehicles takes it further and explicitly establishes a fixed protocol for checking the health of safety-related data such as battery level and navigation sensor status. The protocol involves the collection of diagnostic messages from various nodes of the system, their combination by a dedicated aggregator node, and the execution of user-defined rules which may trigger user-defined recovery actions in case the health of the system is not acceptable (`udg_cirs/cola2_core`). (In this paper, denotes http://bitbucket.com.)

The application of this guideline allows roboticists to take different actions based on collected information, thus providing opportunities for the application of architectural tactics for improving the overall reliability/availability of the system. For example, node health information can be used as input for applying the *Removal from service* architectural tactic for fault prevention; in this case the system can temporarily place a ROS node in an *out-of-service* state (based on its health status) for the purpose of mitigating potential system-level failures. In this context, roboticists needing to implement this tactic could use and adapt the *circuit breaker* pattern for cloud-based software systems [62].

C4 – If the system is remotely distributed, constantly observe the status of the communication channels, hosts, and machines on the network. Some robotics systems are remotely distributed due to their nature, *e.g.,* swarm robotics and UAVs controlled via a ground control and mission planning station such as QGroundControl[4]. In this context it is important for the communication link between ROS nodes across remotely distributed hosts to provide enough bandwidth and speed for ensuring that the system functions as expected (*e.g.,* the communication between the flight control station and its controlled drones should be as reliable as possible). The documentation of the BLUEsat Off World Robotics Rover (https://www.github.com/bluesat/owr_software) provides a simple but effective strategy for implementing this guideline, which is to have a dedicated *"node which pings the ground station to check whether the network connection has dropped out and publishes the result"*; the ping node is used across many parts of the project, but most notably it is used by *"the drive controls to autostop the rover if the network drops out."*

If advanced assessment of network conditions is needed, the Advanced ROS Network Introspection (ARNI) package can be used, which provides additional statistics related to measurements about the hosts and ROS nodes within the system. By providing detailed network information such as message rates and available bandwidth at the host, node, and topic levels, ARNI can help in achieving a higher level of *reliability* when the system is heavily relying on network conditions (`/ROS-PSE/arni`).

Finally, it is important to note that this guideline may conflict with guideline C2. In particular, if the developer wants to observe the status of hosts on the network per C4, the ROS node responsible for doing this must we aware of the network topology. Developers must be aware of the specific context of their project and should apply one of those two guidelines depending on their project-specific needs. For example, the documentation of the `ros_geometry` project states that *"*`tf` *messages [*i.e., *ROS messages representing 3D coordinate frames] do not deal with low bandwidth networks well. Within a single well connected network* `tf` *works fine, but as you transition to wireless and lossy networks with low bandwidth the* `tf` *messages start to take up a large fraction of the available bandwidth. This is partially due to the many to many nature and partially due to the fact that there is no way to choose the desired data rate for each consumer"* (`ros/geometry2`). Observing the status of the communication channels (either at bringup or startup time) and switching to a different configuration and node behavior accordingly is one of the possible solutions to this problem.

C5 – Nodes that potentially produce/consume large amounts of messages should be configurable in terms of their publish/subscribe rates. One of the most relevant benefits of using ROS is the possibility of reusing third-party ROS packages providing various robotics capabilities (*e.g.,* SLAM algorithms, control stacks, drivers for hardware

[4] `http://qgroundcontrol.com`

components). As a result, a typical ROS-based system is composed of various nodes, each potentially (i) producing and consuming messages at different rates and (ii) running on physical nodes with different computational power. When architecting a ROS-based system, the differences in terms of publish/subscribe rates and computational power may lead to unexpected results. For example, developers may experience loss of data if a node is publishing messages faster than the subscribing nodes are able to process.

Roboticists should (i) pay special attention to those nodes that may potentially produce/consume large amounts of messages and (ii) make them configurable in terms of their publish/subscribe rates. The first benefit is in terms of *performance* because pub/sub rates can be adjusted according to the needed level of performance of the system. Second, *maintainability* is also improved because ROS nodes that could have been incompatible due to high differences in pub/sub rates, can be configured to align their rates for compatibility. Third, having configurable pub/sub rates can also positively impact the *reliability* of the system because the risk of negative effects due to loss of data can be better controlled, both at design time and run time.

Different mechanisms can be put in place to implement this guideline. For example, the developers of the ROS driver for the Parrot Bebop drones (https://www.github.com/AutonomyLab/bebop_autonomy) statically limit the publication rate of the node publishing the state of the drone and make it configurable via the Parameter Server; specifically: *"[the state node] does not constantly transmit all onboard data back to the host device with high frequency. Each state variable is sent only when its value is changed. In addition, the publication rate is currently limited to 5 Hz. The driver publishes these states selectively and when explicitly enabled through a ROS parameter."* In another project, the developers of the ROS driver for Parrot AR-Drone quadcopters (`AutonomyLab/ardrone_autonomy`) apply a different strategy consisting of multimodal ROS nodes: *"the driver [of the quadcopter] can operate in two modes: real-time or fixed rate. When the* `realtime_navdata` *parameter is set to* `True`*, the driver publishes any received information instantly. When it is set to* `False`*, the driver caches the received data first, then sends them at a fixed rate. This rate is configured via the* `looprate` *parameter."*

C6 – Selectively limit the data exchanged between nodes to provide only the information that is strictly necessary for completing tasks. This guideline is complementary to C5 and it concerns the content of the exchanged messages instead of their frequency. Specifically, if *performance* is a concern, in addition to configuring the publishing rate of nodes, developers can also control (and limit, if needed) the *size* of published messages by providing only the information that is strictly necessary for completing the task at hand.

A concrete example of the application of this guideline is in the *ros_localization* project, a collection of ROS nodes implementing nonlinear state estimation algorithms for robots moving in 3D space. By quoting the project's documentation: *"if a given sensor message contains data that you [the developer] don't want to include in your state estimate, the state estimation nodes in* `robot_localization` *allow you to exclude that data on a per-sensor basis"* (`cra-ros-pkg/robot_localization`). Technically, this feature is implemented via a configuration vector, which contains a boolean value for each possible value of interest (*e.g.,* velocity); then, depending on the specific requirements of the system, only the values in the configuration vector which are set to *true* are produced by the node, whereas all the others are discarded.

Having fewer data types exchanged between ROS nodes also reduces the attack surface of the system, thus contributing to its *security*. This point is especially important for nodes spawning a dedicated ROS topic for each produced data type. For example, the previously-mentioned ROS driver for Parrot AR-Drone quadcopters provides a feature called "selective Navdata" which allows developers to access a subset of the sensor readings, debug values, and status reports sent from the quadcopter by setting a data structure similar to the *robot_localization*'s configuration vector, with the difference that each selected data type is published to a separate topic. In this case, the lower the number of selected data types, the smaller the number of created topics, and therefore the smaller the attack surface of the system (*e.g.,* malicious attackers have fewer entry points for sniffing the status of the quadcopter during the execution of a mission).

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. For many robotics systems data synchronization is of paramount importance, *e.g.,* autonomous rovers collecting real-time information about the environment via both cameras and laser scans. In order to improve *reliability* in these cases, it is suggested to (i) identify which types of data tend to be often sent/received together, (ii) check if they must be synchronized, and (iii) either package them into a single message or subscribe to them using a *time synchronizer filter*. According to our analysis, data synchronization can be realized either **internally** or **externally** to ROS nodes.

Internal synchronization is implemented via ROS message filters, *i.e.,* special C++/Python components that ROS nodes can use internally for receiving messages and then, based on specific conditions, filtering them out if the conditions are not met. Time synchronizer filters are special types of message filters which "synchronize incoming channels by the timestamps contained in their headers, and outputs them in the form of a single callback that takes the same number of channels" (http://wiki.ros.org/message_filters#Time_Synchronizer).

External synchronization consists of externalizing the business logic of data synchronization into a dedicated ROS node. In this context, a ROS node acts as a *mutiplexer* for ROS messages; it (i) subscribes to multiple ROS topics (or exposes multiple services) and (ii) publishes the synchronized data to one topic (or exposes only one service for providing synchronized data). The business logic for establishing when two or more messages should be synchronized is not limited to timestamps, but rather it can be refined according to the specifics of the system (*e.g.,* two messages can be synchronized when the difference between their timestamp is lower than 10ms and their session id field is the same).

Data synchronization is particularly important when dealing with multiple sensors whose data must be aggregated and aligned in real-time. As an example, the *rtabmap_ros* system (`introlab/rtabmap_ros`) provides a SLAM algorithm with real-time constraints and implements external data synchronization. Specifically, this system has a nodelet called `rgbd_sync` which *"synchronize[s]* `RGB`*,* `depth` *and* `camera_info` *messages into a single message. [The developer] can then use* `subscribe_rgbd` *to make rtabmap or odometry nodes subscribing to this message instead. This is useful when, for example, rtabmap is subscribed also to a laser scan or odometry topic published at a different rate than the image topics. We can then make sure that images are correctly synchronized together. If [the developer] has a camera publishing on the network, this can be also a good format to synchronize images before sending them on the network, to avoid synchronization issues when the network is lagging."*

C8 – Develop adapter components when data exchanged between nodes is not compatible (semantically), incorrect, out-of-order, or redundant. The fact that ROS-based systems can be composed of multiple independently-developed nodes may lead to incompatibilities between them, which may prevent the *reuse* of already-available functionality (*e.g.,* nodes using a custom representation for point clouds) or lead to *malfunction* of the whole system (*e.g.,* nodes providing duplicate or out-of-bounds sensor readings). Having intermediate nodes acting as adapter components can allow developers to overcome those potential issues. In this context, we define an *adapter component* based on the definition of the adapter pattern by Gamma *et al.* [34]: an adapter component is an intermediate entity which converts the interface of a component into another interface expected by its client. In this specific context, components are ROS nodes, interfaces are either the shared ROS topics or ROS services, and interface incompatibilities may arise when the data exchanged between them is semantically different, incorrect, out-of-order, or redundant.

During our analysis we saw several occurrences of adapter components. The well-known MoveIt project for manipulation-based motion planning makes extensive use of this design pattern (`AcutronicRobotics/moveit2`). For example, *"the fix start state bounds adapter fixes the start state to be within the joint limits specified in the URDF [the physical model of the robotics arm]. The need for this adapter arises in situations where the joint limits for the physical robot are not properly configured. The robot may then end up in a configuration where one or more of its joints is slightly outside its joint limits. In this case, the motion planner is unable to plan since it will think that the starting state is outside joint limits. The "FixStartStateBounds" planning request adapter will "fix" the start state by moving it to the joint limit. [...] A parameter for the adapter specifies how much the joint can be outside its limits for it to be "fixable"."* The latter case extends the notion of adapter by making the adaptation logic extensible via external parameters provided by the interface clients.

An implementation of an adapter component for incompatible and redundant data is provided in the *rrt_exploration* project for multi-robot map exploration. The *"filter nodes receive the detected frontier points from all the detectors, filter the [received] points, and pass them to the [...] node to command the robots. Filtration includes the deletion of old and invalid points, and it also discards redundant points."*

Adapters can also be used for improving the *reliability* of the system. For example, the previously mentioned COLA2 autonomous underwater robot has a node in charge of checking if the values passing through it fall within pre-established safety boundaries and, by quoting the project's documentation, *"if some of these values are not inside a user predefined threshold, the vehicle can abort a mission and/or surface [...] automatically."*

C9 – Use services when starting up robots (instead of publishing to topics) so that the status of the system can be checked before operation. ROS services allow ROS nodes to communicate synchronously, *i.e.,* caller nodes wait until they have received a response from the called nodes before continuing their execution. The initial phase of the execution of a robotics system typically includes a phase in which each robot within the system is started up – roboticists

refer to this phase as the *bringup time*. At bringup time, a series of key operations for the overall functioning of the whole system are performed, such as starting the ROS nodes managing the hardware components of each robot (*e.g.,* drivers for sensors, LIDARs, cameras), checking the battery levels of battery-operated robots, checking and publishing the state of the robots to the rest of the system, performing preliminary diagnostics checks, running the control nodes of the robots and estimating their initial pose, and launching monitoring and logging nodes.

The specific operations performed at bringup time are system-specific, but some characteristics are common across many ROS-based systems: (i) the duration of the bringup phase is relatively long and may take seconds or even minutes, (ii) in this phase nodes send diagnostics messages and acknowledgement of (mal-)functions, and (iii) a failure in this phase may heavily compromise the reliability of the whole system for the duration of the entire run. Therefore, developers should use ROS services for exchanging messages at bringup time in order to naturally exploit the blocking nature of service calls, thus ensuring that all messages are delivered and all bringup operations are performed successfully and in the correct order. Given the typical long duration of the bringup phase, the overhead induced by the blocking nature of service calls can be generally afforded for most robotics systems. In addition, ROS services can also be used to exchange data that is mandatory for starting up the system. For example, the ROS interface to the onboard SDK for DJI flying drones (introduced in the description of C3), has a dedicated service *"to activate the drone with app ID and key pair"*, thus ensuring that the drone is activated only if the correct <app ID, key> pair is provided. After checking the credentials provided by the developer, it checks the drone firmware, performs diagnostics operations, sets up the baud rate for the transfer of telemetry data, and activates stereo vision for collision avoidance and onboard image processing (if stereo cameras are present).

C10 – Use empty messages when triggering atomic actions. Some robotics systems have very tight constraints in terms of available network bandwidth and latency. Similarly, some robotics systems also have very limited computational power. For example, the controller of the robot presented in [77] is powered by an Arduino UNO [4] board, which has only 2Kb of memory. In these types of robotics systems it is very important to optimize the usage of available (network and computational) resources as much as possible. If a ROS node is expecting atomic messages whose objective is simply to activate a function without any additional parameters, developers should use *empty messages* when triggering those functions. In this way, the developer can keep to a minimum both (i) network usage because sent messages have a lighter payload and (ii) computational resources for marshalling and unmarshalling the messages. The ROS `std_msgs` package provides the `Empty` primitive type for sending empty messages.

This guideline is especially useful when the sent messages are simply needed to activate a specific function in the ROS node receiving them. For example, the ROS driver for Parrot AR-Drone quadcopters mentioned in C5 takes advantage of this strategy when atomic commands must be sent to the drone, specifically *"the drone will takeoff, land or emergency stop/reset if a ROS* `std_msgs/Empty` *message is published to* `ardrone/takeoff`*,* `ardrone/land` *and* `ardrone/reset` *topics, respectively"*. Clearly, the added value brought by this guideline is very project- and configuration-specific and its application may come at the cost of greatly restricting the interface provided by ROS nodes (*e.g.,* the ROS driver for Parrot AR-Drone quadcopters does not allow the target altitude to be specified when the drone takes off). This guideline could be viewed as a specialization of C6. We decided to separate them because they are semantically very different, and the use cases in which they are applied can be quite different (*i.e.,* C6 is about minimizing data exchange, whereas C10 is about sending acknowledgements or switching on/off specific functionalities).

We suggest to developers to first check if the available network and resource constraints are constrained enough to justify the application of this guideline. A good starting point is to benchmark the network of the system under development by using standard tools for network analysis, such as Wireshark [121]. In addition, the ROS ecosystem provides the `rostopic` command-line tool [97], which allows developers to inspect a given ROS topic in terms of, among others, its publishing rate, bandwidth, and published messages. Finally, since its Indigo release (July 2014), the ROS platform provides the `enable_statistics` parameter, which supports the collection of runtime statistics about topics, such as message age, number of dropped messages, and traffic volume in bytes.

C11 – Frequent messages should be exchanged either via services with persistent connections or via topic-based communication. As mentioned in Section 2, in ROS, there are two primary communication types: topics and services. Producing frequent messages means that the underlying network must be able to deliver them efficiently and that receiving nodes must be able to consume the received data at least with the same rate of the sending nodes. Both benchmarking the network as described in C10 and measuring the overall performance of the nodes within the system are key to understanding if the frequency of exchanged messages negatively impacts the overall performance of the system.

When a ROS node receives messages at too high a frequency, there are two main solutions for improving its performance: (i) using services with persistent connections and (ii) using topic-based communication. In normal conditions, when a ROS node implements a ROS service, internally the service is registered in the ROS Master node, which acts as a service broker for all ROS nodes. When a client node makes a call to a service, the ROS Master has to lookup which nodes are implementing the called service and allow the client and server nodes to communicate in a peer-to-peer fashion (typically via TCP in ROS 1 or DDS in ROS 2). This happens every time any node within the system calls a ROS service. On the other hand, a *service with a persistent connection* keeps the client-server TCP connection open. This solution considerably improves the performance for repeated calls to the same service because it avoids the overhead of service lookup and reconnection for each service call. It is important to note that services with persistent connections may greatly impact the reliability of the system because (i) developers of client nodes must manually implement the reconnection logic in case the server node goes down [100] and (ii) if a new server node appears in the system, the persistent connection will still target the original server node, potentially losing an opportunity for load balancing.

The second part of the guideline refers to the use of topic-based communication. This solution is advantageous from the point of view of performance because (i) nodes subscribing to a topic (*i.e.,* the client nodes) receive published messages in a reactive and non-blocking manner, (ii) lookup operations are not performed on a per-request basis, but rather publisher (subscriber) nodes receive updates about available subscribed (published) nodes on-the-fly by the Master node, and (iii) once the pairs of publisher and subscriber nodes are established, the rest of the communication is generally based on persistent and stateful TCP/IP socket connections. However, as mentioned by a questionnaire respondent, it is important to a declare a node as a *"publisher"* as soon as it is started up (not only just before messages are published); otherwise subscribers might not be ready to receive the published messages, leading to potential loss of the first messages. For more details about the networking aspects of ROS, we refer the reader to the Technical Overview of ROS [95].

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. As discussed in C10 and C11, exchanging messages between nodes may result in considerable performance overhead. If the developer can safely assume that a group of nodes will always run in the same physical host (*e.g.,* a laptop acting as ground control station of a UAV), then those ROS nodes can run in a single process to reduce the overhead due to inter-process communication and message (de-)serialization. In a sense, this guideline is complementary to C4 because it deals with co-localized ROS nodes instead of remotely distributed nodes.

This guideline can be implemented in different ways, depending on the version of the ROS platform used. ROS 1 supports this guideline via *nodelets*. Nodelets can be considered objects living inside the same ROS node (thus inside the same process within the host OS) but having separate namespaces [88]. Intuitively, nodelets are within the same process at the OS level, but they act like separate nodes. On one hand this allows developers to still use the topic-based communication primitives provided by ROS, but on the other hand their communication is intra-process. Communication between nodelets is based on zero copy pointer passing (*i.e.,* when a nodelet publishes to a topic it is actually just sharing a pointer to the published message to all subscribed nodelets), thus skipping completely the serialize/deserialize steps of ROS. This mechanism is very fragile and adds additional complexity to the internal code of the nodelets. Developers must manually control the modifications of the memory locations referred to by the shared pointers in order to ensure the consistency of the messages exchanged between nodelets. In general, given that standard ROS nodes are easier to use, the application of this guideline should be driven by a real need, *i.e.,* cases where network benchmarking results in the identification of performance bottlenecks, for example due to high frequency and payload size of exchanged messages.

ROS 2 supports C12 via *intra-process communication* (IPC). IPC is conceptually similar to nodelet-based communication in ROS 1, but it is better integrated with the implementation of standard ROS nodes; it can be enabled on any ROS node at run time by simply programmatically changing a configuration option of the node. This enables developers to change the communication style (*i.e.,* inter- vs intra-process) of ROS nodes depending on the performance level required by the system according to the current operational context. Similarly to ROS 1, the ROS 2 implementation of IPC aims at skipping the cost of message serialization, however it avoids the additional complexity of managing shared memory locations by relying on a pair of independent buffers for each ROS topic. For further details on ROS 2 IPC we point the reader to the design documentation [80].

C13 – Manage topics to avoid unnecessary publishing and subscribing. As stated in C11, the ROS Master node

acts as a service broker for all ROS nodes. The ROS Master node contains two registries for topic-based communication: one for topic publishers and one for topic subscribers [95]. Every time a node subscribes (publishes) to a new topic, a new entry in the subscribers (publishers) registry is created for the <node, topic> pair. These pairs are continuously kept up-to-date by the ROS Master node by exchanging advertisement messages with all the other nodes in the system via XML-RPC. Therefore, from a very abstract point of view, every time a node publishes to a topic, the Master node looks up in the topic subscribers registry all nodes subscribing to that topic and negotiates a direct (*e.g.,* TCP) connection between the publisher and every subscriber. The ROS messages are exchanged only when publisher-subscriber direct connections are established.

Questionnaire participants suggested to (i) *"unsubscribe from topics where no messages will be published to (even temporarily)"*, (ii) *"unsubscribe from topics when the subscribing node will not need the messages published to them (even temporarily)"*, and (iii) *"publish messages (and perform the computation necessary to generate them [this part is related to guideline B6]) on a topic only when there is at least one subscriber to it."* The general message of these suggestions is that developers should keep the registries of the Master node as clean as possible, so that they do not contain unneeded entries. Lighter registries in the ROS Master node can lead to a *performance* improvement of the whole system because (i) almost all publish/subscribe operations involve a negotiation phase passing through the ROS Master node, making it a potential system bottleneck, and (ii) having fewer entries in the registries can result in faster insert and lookup operations. Moreover, the application of C13 also helps in terms of architecture *understandability*: having fewer publish/subscribe relationships within the system leads to a simpler and more cognitively manageable computation graph and helps roboticists to reason about the overall architecture of the system.

Even though no participant mentioned it, we highlight that this guideline also holds true for *service providers* and for subscribers to parameters in the *parameter server* [95]. In this context, possible implementations of C13 include: (i) to `shutdown` provided services when they are not needed (even temporarily) and (ii) to `delete` parameters that are no longer needed from the parameter server.

7.2. Node Responsibilities within the System

This family of concerns represents the guidelines related to the responsibilities of ROS nodes within the system, how to achieve low coupling and high cohesion between nodes, independent deployment and testing, and so on. Table 5 presents the guidelines belonging to this family.

Table 5
Guidelines for Node Responsibilities within the System

ID	Guideline	Quality Requirements	P	U_g	S
N1	Group nodes and interfaces into cohesive sets, each with its own responsibilities and well-defined dependencies.	Maintainability, Portability	G(3) Q(2)	132	G
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.	Maintainability, Portability	G(9)	127	G
N3	Decouple nodes with responsibilities that naturally work at different rates and use different rates for different purposes.	Performance, Reliability, Maintainability	G(7)	123	S
N4	By design, limit unnecessary computationally-heavy operations by carefully analyzing the execution scenarios across ROS nodes.	Performance, Energy	G(6) Q(1)	106	G
N5	Transform data only when it is used, for efficiency in terms of computation and bandwidth.	Performance, Maintainability	G(1)	105	G
N6	Design each single node so that it is runnable (and testable) in isolation.	Maintainability	G(2) Q(2)	88	G
N7	Provide dedicated nodes for doing introspection and querying the lower levels of the system.	Portability	G(3) Q(1)	41	R
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.	Reliability, Safety, Performance	G(5)	29	S
N9	Keep the number of nodes as low as possible to support the basic execution scenarios and extend the architecture for managing corner cases.	Maintainability	Q(3)	-	S
N10	Take full advantage of existing packages in the ROS ecosystem and create your own package only when it is strictly needed.	Maintainability, Reliability	Q(4)	-	S

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 [108]. Such complexity can lead to technical debt, lock-in to specific ROS packages (which in turn can have other hardware/software dependencies), and difficulty extending the system with new or better nodes. Developers need to carefully manage 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 evolving the system. For example, the architecture of the ROS stack for the Niryo One manipulator arm is designed around the five layers presented in Figure 2 (NiryoRobotics/niryo_one_ros).

Having a well-defined organization of node responsibilities allows developers to (i) concisely reason about the workflow of the system and (ii) reuse well-maintained and tested packages such as `MoveIt` and `ros_control`.

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. Our analysis revealed that robotics systems are composed of a number of features or robot capabilities, such as planning, control, vision, etc. (see Figure 4c). Typically, in ROS each of those features is realized by a dedicated ROS package (or stack in case of large features such as arm control) and interacts with the rest of the system via the ROS communication infrastructure. For example, the ROS-TMS service robot system [73] *"consists of 73 packages categorized into 11 groups and 151 processing nodes. Re-configuration of structures, for instance adding or removing modules such as sensors, actuators, and robots, is simple and straightforward owing to the high flexibility of the ROS architecture."* (irvs/ros_tms)

In order to improve the overall technical sustainability of the system, each ROS package should be responsible for one and only one feature of the system (or robot capability) and provide a well-defined interface for the other nodes within the system. In other words, it is advantageous to architect the system around the main capabilities of the robot in order to ease the future maintenance, evolution, and operation of the robotics system. For example, the developers of the `Free Gait` controller for legged robots achieved this by *"separating the motion generation [from] the motion execution [. . .] enabling to easily port motion generation methods and algorithms to a different robot. As Free Gait is also independent of the underlying whole body controller, implementing different control methods for the same control tasks requires only to write a new adapter without additional overhead. Furthermore, it is often useful to simulate or preview the motion plan, which can be accomplished by writing adapters for each and viewing the Free Gait commands before the execution on the real robot."* (leggedrobotics/free_gait).

Architecting the system around its features/capabilities also enables independent testing and deployment of the involved stacks/nodes, thus strongly supporting collaborative development. For example, the SRS project we mentioned in C2 *"involves dozens of researchers and developers from different organisations, disciplines, and backgrounds. Since it targets to build large systems contributing to a sizable code base, it is of high importance to enforce modularity and interoperability between the software components and organise them in a systematic way allowing concurrent work on the system from all the partners in a fashion that components can be developed and verified separately, and integrated efficiently in the future."*

N3 – Decouple nodes with responsibilities that naturally work at different rates and use different rates for different purposes. The hardware and software of a ROS-based system can produce data at very different rates. For example, the RPLIDAR low-cost LIDAR sensor produces field scans with a frequency of 10Hz [98], whereas the SenseComp Series 7000 ultrasonic sensor is able to produce readings at 50kHz [109]. The ROS nodes implementing the drivers of those two sensors internally operate at different rates, but more importantly they also publish messages for other nodes at very different rates.

First, we suggest to roboticists to decouple nodes with responsibilities that work at different rates. Coming back to the example of the two sensors, having a single ROS node subscribing to both ROS drivers will result in a single ROS node receiving data at completely different rates (*i.e.,* 10Hz on the topic published by the LIDAR and 50Khz on the topic published by the ultrasonic sensor). This can result in a number of drawbacks, specifically: (i) testing and debugging the node receiving the sensor data is cumbersome because two totally different data types and data frequencies need to be inspected by the developer, (ii) the physical device where the node is running may become a performance bottleneck due to the high frequency of the ultrasonic data, thus lowering also the performance of the (potentially unrelated) portion of the system managing LIDAR data, and (iii) the performance bottleneck discussed in the previous point may lead to message loss, potentially causing reliability- and safety-related issues. A solution to this issue is to keep as independent as possible all control and data flows that have different frequencies. As an example, the ROS node for obstacle detection and avoidance of the PX4 open-source autopilot has different nodes for its local,

global, and safe landing planners (PX4/avoidance).

Second, this guideline also suggests to use different rates for different purposes, depending on the specific nature of the produced data. By doing this, the code of ROS nodes will better match the domain of the produced data, semantic mismatches between ROS nodes will be less likely, and computational resources will be used efficiently. As an example, in the `neonavigation` package for autonomous vehicle navigation (at-wat/neonavigation) *"the motion control that controls the vehicle to follow the generated path [...] and collision prevention are separated from the planner; then the motion control and collision prevention can have the faster control frequency. For example, global/local planner frequency is around 5 Hz, and the motion control and collision prevention can be 50 Hz [...] The main advantage of this meta-package is that the collision avoidance frequency can be faster than the conventional, planned path, always taking the global goal into account."*

N4 – By design, limit unnecessary computationally-heavy operations by carefully analyzing the execution scenarios across ROS nodes. Given that in many cases robots operate under strict hardware and networking constraints, it is suggested to design the system in a way that limits unnecessary computationally-heavy operations.

One way to apply this guideline is to identify portions of the system that do not need to always be up and running and to bring them up only when they are needed. As stated in the documentation of the `Capabilities` package of the Robotics-in-Concert (ROCON) project: *"only run those capabilities that are required by currently running applications (e.g., to save power and/or processing)."* (osrf/capabilities). In the context of the ROCON project, capabilities required by a robotics application include the ability to navigate in the physical environment, access the video stream produced by an RGB camera, etc.

Another strategy is to exploit (expected or known) characteristics of the physical environment where the robots operate, and apply heuristics or use pre-computed information instead of performing unnecessary computationally-heavy operations at run time. For example, the Google Cartographer system provides a suite of real-time SLAM algorithms. Among many, it provides a ROS node for building an occupancy grid representing the physical obstacles and objects in the environment. This node applies the following strategy for improving its performance: *"generating the map is expensive and slow, so map updates are in the order of seconds. [The developer] can selectively include/exclude submaps from frozen (static) or active trajectories with a command line option."* (googlecartographer/cartographer_ros).

Also the characteristics of the physical body of the robot can be exploited to reduce computationally-heavy operations. For example, the contributors of the `MoveIt2` framework are aware that collision checking is one of the most expensive operations when planning the movements of an industrial arm. They mitigate this potential performance issue by (i) building a matrix representing all possible pairs of physical bodies (either in the arm or in the environment where it operates) and (ii) checking the collisions only for pairs of bodies that may come into contact, thus avoiding the need to check collisions when *"two bodies are always so far way that they would never collide with each other."* (ros-planning/moveit2)

N5 – Transform data only when it is used, for efficiency in terms of computation and bandwidth. This guideline can be considered as a corollary of N4, with the difference that here the focus is on the computation and transformation of the messages exchanged between nodes. The underlying intuition behind this guideline is to take into consideration the current state of the system and to avoid unnecessary operations.

This guideline is particularly relevant in the context of ROS nodes publishing messages at very high rates. For example, in the `tf2` project, coordinate transformations are used to represent the orientation of robots, sensors, or other objects in the physical environment among the various frames; typically, coordinate transformations are computed, organized into a buffered tree structure, and broadcast with a frequency ranging from 10Hz (default) to 100Hz (real-time). Those operations are not computationally expensive individually, but their execution at high frequencies may impact the performance and energy consumption of the system, so the developers of `tf2` agreed on the design goal of *"only transform[ing] data between coordinate frames at the time of use."*

More in general, there are situations where many high-frequency messages are produced by a ROS node (*e.g.,* the ROS driver of a sensor) and the data contained in those messages may undergo different manipulations by other nodes. Developers should pay attention to those situations and limit as much as possible the frequency in which this data is produced.

N6 – Design each single node so that it is runnable (and testable) in isolation. ROS nodes are meant to interact with each other in order to realize the features provided by the system as a whole. However, in order to support the maintainability and testability of the system, it is suggested to design each single node so that it is runnable and testable

in isolation.

The first advantage of following this guideline is the promotion of the *continuous integration testing* practice within the project. Indeed, having independently deployable and runnable ROS nodes allows developers to run unit tests against them and to automatically detect regressions. The ROS community promotes the systematic use of automated tests for a number of reasons, including faster incremental updates, higher confidence when refactoring code, fewer bug regressions, promotion of contract-based development, and easier onboarding process into open-source projects [90].

The ROS community also suggests using different testing frameworks for different testing levels. First, for testing nodes at the library level, *i.e.,* independent of the ROS communication infrastructure, the `unittest` [74] and `gtest` [37] frameworks are suggested for Python code and C++ code, respectively. Second, for testing node functionality which is strictly dependent on the ROS communication mechanisms, the `rostest` [96] framework allows developers to launch the node under test, provide fixtures for mocking the rest of the system, and programmatically exercise the interface of the nodes via their published and subscribed topics, services, needed parameters, etc. Finally, `rostest` can also be used in the context of multiple nodes, thus enabling the execution of *integration testing*, *i.e.,* to verify that (independently-developed) nodes behave as expected when they are connected to each other [65]. Automated testing also requires the presence of test input data, which can be serialized as ROS bag files and used by the continuous integration pipeline. For example, the SLAM Constructor Framework *"provides the utility* `lslam2d_bag_runner` *to launch algorithms in the offline mode to process datasets in BAG format."* (`OSLL/slam-constructor`)

Moreover, designing ROS nodes that are runnable in isolation also helps developers to carefully reason about the specific responsibilities and provided/required interfaces of each node under development, eventually promoting separation of concerns, node decoupling, and independence from contextual factors of the system (*e.g.,* connectivity). For example, the ROS client for UAVs developed at McGill University *"can also run in an offline mode which reads mission and obstacle information from a [JSON] file instead of the server."* (`mcgill-robotics/ros-interop`).

N7 – Provide dedicated nodes for doing introspection and querying the lower levels of the system. Conditions monitoring is a well-known architectural tactic for improving system reliability by detecting faults in a timely manner or validating assumptions made at design time [7]. It is suggested to clearly separate the core functionalities of the system from the monitoring logic, so as not to increase the complexity of these two aspects of the system, to make them independently verifiable and testable, and to achieve higher flexibility (*e.g.,* monitoring nodes may perform different degrees of introspection depending on the current status of the robots).

Having clearly separated monitoring nodes also enables developers to save considerable effort by using off-the-shelf ROS packages for doing introspection and querying the lower levels of the system. The ARNI package (discussed in C4) can also be used for monitoring other aspects of the system in addition to networking, such as read and write I/O operations, CPU and GPU utilization, and memory consumption. ARNI can also be used for fault recovery, indeed *"all statistics or metadata [collected by ARNI] can be compared against a set of reference values [. . .] allowing to run optional countermeasures when a deviation from the reference is detected."* (`ROS-PSE/arni`).

The level of introspection and the type of monitored aspects can also be system- or domain-specific. In those cases, the monitoring logic must be implemented by the developers because they know exactly the type of information to be monitored. For example, the ROS interface to the onboard SDK for DJI flying drones publishes a `query_drone_version` topic for querying the version of the firmware running on the drone. This allows developers to adapt the behavior of the ground controller to the specific characteristics of the drone being used for the current mission. Similarly, the command line tools of the Care-o-bot service robot (`ipa320/cob_command_tools`) allow developers to inspect the status of each single mechanical part of the robot in addition to standard data points such the battery level of the robot, CPU usage, etc.

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. In many robotics systems data consistency can become an issue, potentially leading to safety-related disruptions. For example, when running a swarm of UAVs for performing a mission in an unknown environment (*e.g.,* surveillance or environmental monitoring), it is essential that the world model be consistent across UAVs in order to avoid collisions [16]. In order to avoid such issues, developers can use a dedicated ROS 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 ROS nodes within the system.

This guideline is also followed in SLAM-based systems, where the operations of the robots are often split into two concurrent activities: (i) building a global map of the environment to provide high-level information about the status

of the mission and (ii) building local submaps by exploiting the onboard sensors of each robot in order to facilitate trajectory planning and obstacle avoidance. For example, in the `ROS Navigation stack` *"one costmap is used for global planning, meaning creating long-term plans over the entire environment, and the other is used for local planning and obstacle avoidance."* (`ros-planning/navigation`).

The `MoveIt` project also advocates the centralization of the description of the physical environment where the robots operate to efficiently check collisions of industrial arms against the physical environment. Specifically, the developer documentation for MoveIt states that *"[having a global world representation] has advantages for multi-threaded collision detection. In motion planning, it is often necessary to check a large amount of robot poses against the same current world environment. [...] Keeping a single* `CollisionWorld` *gives memory advantages especially in the case of complicated point cloud world environments."* The Environment Description (ED) project may be useful for representing the global environment of a ROS-based system. ED is developed by TU Delft and provides a dedicated ROS package for representing, storing, and querying 3D physical environments. In this way, roboticists can dedicate their effort towards the core functionalities of their system, instead of dealing with different environment representations (`tue-robotics/ed`).

Depending on the characteristics of the system under development, centralized data sources can also be used for representing information from different physical worlds. For example, the ROCON ROS multimaster system provides the `rocon_hub` package which *"acts as a kind of name server assisting ROS subsystems to find each other."* `rocon_hub` is based on a key-value store implemented in Redis [78] and can be used as an efficient centralised data source for any type of globally-relevant data within a ROS-based system.

N9 – Keep the number of nodes as low as possible to support the basic execution scenarios and extend the architecture for managing corner cases. Similarly to what happened in other industrial sectors some years ago (*e.g.,* automotive [68]), robotics is becoming a software-intensive industry. Nowadays, deploying even simple applications requires integrating a myriad of ROS nodes with intricate interdependencies and all running at the same time [76]. This makes it extremely challenging for roboticists to fully control the development and design of a robotics system today. For example, simply bringing up a PR2 robot by Willow Garage [120] without navigation and perception packages launches 50 nodes and 500 topics.

One way to reduce complexity is to keep the number of ROS nodes as low as possible during all stages of development, specially in the initial stages. As suggested by one of the questionnaire participants: *"most ROS systems have way too many independent nodes with messages going back and forth in order to support theoretical use cases you never see in practice. Writing simple code to solve your actual problem is better than writing complicated code with complicated configuration to try and solve all problems (which won't work anyway)."*

It is suggested to design the architecture of the system in an incremental manner, *i.e.,* to start with the minimal number of nodes for the most basic usage scenarios (and covered by test cases), and only in the following stages to incrementally implement optional features and cover corner cases. In this context, software-in-the-loop (SITL) simulations may be used to speed up development without incurring safety issues. The main characteristic of SITL simulations is that the used software stack is exactly the same as the one used in real operations, with the only difference that hardware drivers are simulated via dedicated software. [107]

This guideline also allows developers to follow agile principles (if needed), thus collecting feedback about the core features of the system early during its life cycle, and having the chance to iteratively refine the design of the system. Incremental development also helps in unveiling performance and reliability regressions. If the system under development is benchmarked at each development cycle, then it will be easier for developers to detect architectural hotspots (*e.g.,* performance bottlenecks or recurrently failing nodes) because the search of their root cause can be scoped to the changes performed in the last iteration.

N10 – Take full advantage of existing packages in the ROS ecosystem and create your own package only when it is strictly needed. This guideline is about not "reinventing the wheel." Indeed, ROS has been designed to facilitate collaborative robotics software development and the ROS community has always encouraged the open-source development of reusable robotics capabilities in the form of publicly available ROS packages [106]. As summarised by one of the questionnaire participants: *"[developers should] look for already existing ROS packages that could be used instead of doing redundant work."*

Reusing third-party ROS nodes and packages also promotes *design by contract* [58]. Design by contract can be advantageous in the ROS ecosystem because the developers of third-party ROS packages do not know, and do not control, how client nodes will interact with them. This means roboticists should carefully design and precisely document

(i) the provided/required topics and services in terms of acceptable and unacceptable input values, error conditions exceptions, etc., (ii) the needed configuration for correctly interacting with other ROS nodes, (iii) the results and side effects produced when they are called, and (iv) performance and reliability guarantees, *e.g.,* in terms of response time, throughput, and memory consumption.

Moreover, reusing third-party ROS nodes and packages also advocates for a higher *separation of concerns*, where robotics capabilities tend to be more localized, without tangling the rest of the system with intricate dependencies. For example, the the Niryo One system described in Section 2 makes heavy use of the `MoveIt` framework, which lives in a dedicated Motion Planning layer. In principle, if a different motion planning framework is used in the future, the impact of this important architectural change will be limited only to the layers directly interacting with `MoveIt`.

Finally, the most commonly used ROS packages tend to be developed with high standards in terms of process quality. For example, at the time of writing the `MoveIt` GitHub repository has 200 contributors who (i) work on a dedicated branch for each of the main ROS distributions, (ii) automatically build the package via a Travis-based continuous integration pipeline, (iii) make extensive use of the issue/pull request mechanism in GitHub for discussing proposed changes, problems, and design decisions, and (iv) automatically run 1,174 distinct test cases every time the package is either built or released [64].

7.3. Internal Behavior of Nodes

The following guidelines are meant to help roboticists make architecturally-relevant decisions about internal behavioral aspects of their ROS nodes, such as spin rates, when to manage configuration errors, having stateless or stateful behavior, etc. Table 6 presents the guidelines belonging to this family.

Table 6
Guidelines for Internal Behavior of Nodes

ID	Guideline	Quality Requirements	P	U_g	S
B1	The behavior of each node should follow a well-defined life cycle, which should be queryable and updatable at run time.	Reliability, Maintainability, Portability	G(3) Q(1)	70	R
B2	Nodes with high-frequency operations should be configurable so that they can operate according to available computational resources.	Performance, Reliability, Safety	G(2)	79	R
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.	Reliability, Usability	G(2)	84	G
B4	When possible, ROS nodes should be stateless and their behavior should not depend on previous operations or received messages.	Reliability, Usability, Maintainability, Safety	G(1)	27	S
B5	Nodes with configuration errors should fail explicitly at bringup time.	Reliability	Q(1)	-	R
B6	If a node is computationally expensive, then ensure that it only executes when it is strictly needed.	Performance, Energy	Q(2)	-	G

B1 – The behavior of each node should follow a well-defined life cycle, which should be queryable and updatable at run time. 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 life cycle as a first-class concern of the system. As a base case, node life cycles 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 life cycle 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, possibly covering all possible states of the nodes within the system.

Some steps towards a precise definition of the life cycle of ROS nodes are being taken by different players in the ROS ecosystem. First, ROS 2 provides support for *managed nodes* [81], *i.e.,* nodes whose life cycle 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. Second, some ROS packages already explicitly consider the life cycle of their nodes in their APIs, even without ROS 2 managed nodes; for example, quoting the documentation of `ros-controls/ros_control`, *"the life cycle of controllers is not static. It can be queried and modified at run-time through standard ROS services provided by the controller_manager."* This example also highlights that defining and enforcing the life cycle of ROS nodes can enhance the system in terms of runtime configurability and reflection, which can be exploited for providing autonomous capabilities [2].

B2 – Nodes with high-frequency operations should be configurable so that they can operate according to available computational resources. This guideline complements C5 by focusing on the internal logic of nodes with high-frequency operations; differently, C5 focuses on the communication aspect among nodes with high-frequency message exchanges.

In a ROS-based system there may be nodes that operate at very different frequencies. For example, the node running the real-time control loop of the Franka robotics arm operates at a frequency of 1Khz [27], whereas the diagnostics node of the PR2 robot mentioned in N9 runs at 1Hz only [120]. Depending on the characteristics of the system or specific responsibilities, a ROS node may require to operate at different frequencies in order to properly function. Operations at too high (or low) frequencies may lead to hardware malfunctions, data loss, or system failures.

In order to take into account the design concerns mentioned above, nodes should be configurable so that they can operate according to available computational resources, specially when high-frequency operations are needed. Our analysis revealed three main strategies for implementing this guideline.

First, the frequency of operation can be provided directly to the nodes to be configured. For example, in the `ROSplane` system [26] (`byu-magicc/rosplane`) *"all nodes can be throttled to operate at lower rates as processing limitations require"*. The throttling rate is provided via the ROS parameter server and is internally used by a timer for triggering the computation at the required rate. This guideline is specially useful for the state estimation node of `ROSplane`, which operates on a fixed-wing aircraft under tight timing constraints. This strategy has the benefit of being easy to implement and use by developers because it involves a simple lookup in the parameter server; however, it may (i) lead to reliability issues if the parameter server becomes (accidentally or maliciously) compromised and (ii) complicate the implementation of the nodes being throttled because the management of their execution rate is intermingled with the execution of their core business logic.

The second strategy involves a dedicated throttler node which acts as a man-in-the-middle among other nodes within the system and adjusts the message frequency of publishing nodes to the computational capabilities of the subscribing nodes. This is the strategy applied by the *rtabmap_ros* system introduced when discussing C7, where a `data_throttle` node (i) subscribes to three input topics for the RGB image stream, depth image stream, and RGB camera metadata, (ii) checks the throttling configuration in the parameter server, namely the maximum frame rate in Hz and approximate synchronization for the input streams, and (iii) publishes to three corresponding output topics according to the specific throttling configuration. This strategy allows to develop the mediated nodes independently of each other, without needing to take into consideration the computational constraints of the other nodes within the system. However, potential drawbacks of this strategy include a potential performance overhead due to the additional serialization and deserialization of the messages passing through the intermediate node and the increase of the overall complexity of the architecture of the system if a large number of throttling nodes are employed.

Third, the frequency of operation can be determined by the ROS platform itself. When a node subscribes to a given topic, it `spins`, *i.e.,* it blocks until the node is shut down and periodically processes the received messages in a callback queue. At the time of writing, the spinning rate for nodes is fixed. The spinning rate for nodes should be either configurable by node developers or dynamically set by the standard ROS clients (*e.g., rospy*, *roscpp*) so that nodes can always operate according to available computational resources.

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. Some nodes within a ROS-based system may be designed to be naturally stateful. For example, the ROS driver for a Parrot AR-Drone quadcopter (see C5) is composed of a single ROS node subscribing to various topics where it receives commands such as `takeoff`, `land`, and the generic `cmd_vel` for flying the quadcopter towards a specified direction. Clearly, `cmd_vel` messages are meaningful only when the quadcopter has already taken off. During our analysis, we noticed that roboticists tend to not specify these kinds of dependencies between ROS messages. This underspecification may lead to *reliability* and *usability* issues because messages exchanged in the wrong order may cause (i) malfunctions or failures in the server nodes (*e.g.,* the quadcopter may lose its balance while still on the ground) or (ii) unexpected behavior from the perspective of client nodes (*e.g.,* the ground control station sends flying commands but the quadcopter is still on the ground).

As a solution, if a ROS node is stateful and its behavior strongly depends on time and message arrival order, it is important to specify its expected message protocol. The level of detail and rigor of such a specification depends on the domain of the robot and project requirements. Moreover, this guideline may be used in combination with B1, where the description of the behavioral life cycle followed by the node also represents the types of messages which can be sent/received by the node for each state. However, it is important to note that ROS 2 managed nodes are not of great

use here because in this context the states of a node are application-specific (*e.g.,*, the quadcopter flying), rather than standard states (*e.g.,* active, inactive).

Moreover, specifying the protocol of expected messages is especially important for nodes subscribing to multiple frequently published topics. Indeed, messages published to different topics internally trigger different independent callbacks in the receiving node. Callback-based programming is generally challenging because the order in which different code fragments are executed is unpredictable; in the literature, this problem is called callback hell [25].

A first step towards the satisfaction of this guideline is to lower the chances of wrongly ordered message exchanges by (i) clearly identifying stateful nodes in the documentation of the system and (ii) precisely documenting the expected order of the messages they can receive. In case a server node performs long-term operations, another helpful strategy is to use ROS actions [84], so that server nodes can inform client nodes about the status of their internal operations via progress updates.

B4 – When possible, ROS nodes should be stateless and their behavior should not depend on previous operations or received messages. Stateless components have a several advantages in terms of reliability, usability, and maintainability. In the context of ROS-based systems, (i) redundant stateless nodes can subscribe to the same topic (*e.g.,* for load balancing) and any of them can handle incoming messages, (ii) if a node fails or is unexpectedly shut down, messages can quickly be routed to another node without data loss, (iii) test cases do not need to rely on complex fixtures for reconstructing the internal state of the node under test, (iv) nodes can be redeployed to different hosts at run time according to available computational resources, and (v) because there is no state management, the code of stateless nodes tends to be easier to understand, debug, and extend.

The implementation of this strategy is highly application- and project-specific. As a starting point we suggest to developers to identify the nodes that are good candidates for being stateless and to implement their interfaces in such a way that their behavior does not depend on previous operations or received messages. Hints leading to stateless candidates include ROS nodes acting as data pipes, data filters, or providing capabilities whose behavior depends only on the information stored in the currently sent message. A concrete example of a system with stateless nodes is the `Geometry 2` package. It is designed to be a highly distributed system with no bottlenecks and minimal latency. The developers of this package achieve this design goal by building on stateless nodes where *"everything [the data] is broadcast and reassembled at end consumer points. There can be multiple data sources for* `tf` *information. Data is not required to be synchronized by using interpolation. Data can arrive out of order."*

Finally, it is important to note that implementing this guideline may not be feasible in some cases and it may be subject to application-specific constraints (*e.g.,* an object tracker is naturally stateful). Nonetheless, we suggest to developers to strive as much as possible towards stateless nodes in order to benefit from the advantages we mentioned above.

B5 – Nodes with configuration errors should fail explicitly at bringup time. The majority of ROS packages can be configured for a number of reasons, *e.g.,* to align their behavior to the rest of the system (*e.g.,* topic remapping), to make them properly run on the available hosts (*e.g.,* defining the joints managed by a controller), etc. Configuration errors can happen in any project. However, the impact of configuration errors highly depends on *when* they happen. For robotics systems, discovering a configuration error while the system is fully operational may lead to considerable reliability and safety issues; for example, imagine discovering that the node for receiving velocity commands on a drone is misconfigured while the drone is already mid-air.

The purpose of this guideline is to discover misconfigurations as early as possible, possibly before the system is engaging in its operations. The life cycle of the the majority of ROS-based systems can be roughly divided into three main phases: (i) bringup, where the system is configured and brought to an operational state, (ii) operations, where the system carries out its main tasks, and (iii) shutdown, where the system is halted. In light of this, it is suggested to (i) *check* node configurations and (ii) let misconfigured nodes *fail explicitly* **at bringup time**. The bringup phase is the best candidate for checking node configurations because it is the time where the majority of the nodes of the system are launched, either locally or on several machines, global parameters are set, hardware and low-level diagnostics procedure are carried out, and the connectivity between the core nodes of the system can be safely checked. Typically, all those activities are implemented in ROS *launch files* [28]. In ROS 2, launch files are implemented in Python, making them good candidates for containing checks for configuration errors, both within individual ROS nodes and globally within the whole system (*e.g.,* for checking that the launched nodes follow the expected communication topology).

Clearly, checking configurations at bringup time does not mean that system diagnostics can be completely avoided during the operations and shutdown phases. However, having explicit failures at bringup time is a way to localize the

majority of configuration problems in the life cycle phase of the system where their consequences are potentially far less severe. Also, having nodes with configuration errors failing explicitly at bringup time makes the system more maintainable and usable. Indeed, after the bringup phase, developers can relatively safely assume that the system is properly configured (at least for development purposes), thus speeding up the development and testing of its core capabilities.

B6 – If a node is computationally expensive, then ensure that it only executes when it is strictly needed. As we already discussed, some ROS nodes may be very expensive from a computational perspective and at the same time are required to run on computationally-restricted hosts. For example, the image processing node by Dunkley *et al.* [24] runs on a 25g Crazyflie nano-quadrotor [45] with only 3.5m of flight time. In those cases it is important to ensure that computationally expensive nodes carry out their operations only when it is strictly needed. In the example of the Crazyflie quadrotor, if the current task of the drone is to hover for a certain amount of time and by design there are no dynamic obstacles that may hit it, then its image processing node can be safely paused, thus leading to considerable energy and performance savings.

This guideline can be implemented at different levels, depending on how the responsibility of computation management can be allocated to nodes. For example, the system may follow an *orchestration-based design* where a managing node collects the state of all other nodes and sends them suitable start/stop/resume messages depending on the current status of the mission. In this case each managed node must provide an interface for starting/pausing/resuming its own computation. Differently, the system may follow a *choreography-based design* where the responsibility to manage computation is left to the single nodes, which autonomously decide when to pause themselves. In this case, each managed node is independent from the others and has to have introspection capabilities. The decision on which strategy to follow is highly application-specific and has been widely discussed in the areas of web services [69], wireless sensor networks [14], and scientific workflows [32].

7.4. Interfaces to External Users and Third-Party Developers

This family of guidelines refers to the interaction of ROS-based systems with external third-party stakeholders, mainly users and developers. Table 7 presents the guidelines belonging to this family.

Table 7
Guidelines for Interfaces to External Users and Third-Party Developers Guidelines

ID	Guideline	Quality Requirements	P	U_g	S
I1	Assign meaningful names to architectural elements (*e.g.,* nodes, topics, messages, services) and group them by adopting standard prefixes/suffixes.	Maintainability Usability	G(4) Q(3)	190	G
I2	When possible, core algorithms, libraries, and other generic software components should be ROS-agnostic.	Portability Maintainability	G(1) Q(1)	170	G
I3	Expose a single ROS node with interfaces for third-party users for the most common use cases.	Usability Maintainability	G(7)	110	S
I4	Systems interacting with other non-ROS systems should provide two types of interfaces: a ROS-independent interface for non-ROS systems and a ROS-based interface for ROS tools (*e.g.,* Rviz, Qt).	Usability Compatibility	G(3)	101	S
I5	Identify variation points of the system in advance, and design the system to be extended by third-parties without modifying its core nodes.	Maintainability	G(24) Q(2)	119	G
I6	Logging should be standardized across the project and follow well-defined guidelines.	Maintainability	Q(1)	-	G

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 [43, 13], bad practices for naming ROS entities negatively impact the architecture of ROS-based systems. This problem is especially severe in ROS-based systems because ROS topics and services are created programmatically by the nodes at run time and their identifiers are simply strings. In addition to the obvious issue related to understandability, bad naming practices can lead to runtime 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 [122].

Developers should adhere to standard naming conventions for ROS architectural entities such as nodes, topics, and services. The name of each ROS node, service and, most importantly, topic should clearly describe their responsibilities within the system, content, and other key characteristics of architectural relevance (*e.g.,* whether a topic is transient

or permanent, the fact that a node is a singleton within the architecture, whether a topic belongs to a family of topics providing the same type of data). A good starting point for making informed decisions about how to name ROS entities is the ROS conventions page in the official ROS Wiki [93].

Moreover, adopting naming conventions can also help to avoid name clashes at run time. For example, the contributors of the `ros/geometry2` package support multiple robots with the same or similar configuration by using *"a tf_prefix similar to a namespace for each robot: if the robot gets unplugged and replugged in, this value may change. It is recommended to setup a udev rule for the robot that will always name it something like* `/dev/rover` *and to set this parameter to the same thing so that if it gets unplugged and replugged in things will [still] work."*

I2 – When possible, core algorithms, libraries, and other generic software components should be ROS-agnostic. Technically, a ROS node can be considered a piece of software (usually implemented in C++ or Python [30]) with the additional capability of being able to communicate with other nodes through the ROS communication middleware. Therefore, the source code of a ROS node is composed of code conforming to a standard programming language which makes calls to the communication API provided by ROS (this API is provided by ROS clients such as *rospy* and *roscpp*). Indeed, in the ROS ecosystem there is a large number of packages that simply provide a ROS-oriented wrapper around libraries widely used in contexts other than robotics, such as OpenCV for real-time computer vision [104]. Conceptually, the source code of a ROS node can be always divided into ROS-specific code and ROS-independent code.

This guideline suggests to (i) minimize as much as possible the amount of ROS-specific code and (ii) clearly separate ROS-specific code from the ROS-independent code implementing core algorithms, libraries, and other generic software components. This design decision helps developers in several ways: (i) unit testing for ROS-independent code is faster and more straightforward because there is no need to bring up all the (potentially hardware-dependent) ROS nodes needed by the source code under test, (ii) it is more portable and reusable in other contexts because it can also be deployed in systems that do not use ROS, (iii) if new versions of the ROS-independent code are released (*e.g.,* a new version of the OpenCV library), then they can be easily integrated into the system without the risk of having to modify other nodes within the system, (iv) the cumulative knowledge of the communities around the wrapped libraries can still be exploited independently of the fact that they are used in a ROS-based system; for example, at the time of writing, Stack Overflow contains 56,138 technical discussions tagged with the OpenCV topic, and among them only 198 are specific to ROS.

I3 – Expose a single ROS node with interfaces for third-party users for the most common use cases. The ROS ecosystem includes a large number of systems that are meant to be used in the context of larger systems (in Section 4 we called them *subsystems*); *e.g.,* the `Moveit` project provides planning and control capabilities, which are typically used by other ROS nodes to provide the full functionalities of the system. Subsystems are composed of multiple nodes and their interaction with the rest of the system may not be trivial. In this context, it is of paramount importance to design the "interface" or "edge" between a subsystem and the rest of the system in order to (i) improve the degree to which the subsystem can be used by system developers to achieve their goals with effectiveness, efficiency and satisfaction (*usability*) and (ii) clearly separate the responsibilities between the subsystem and the system using it in order to ease future modifications (*maintainability*).

This guideline proposes to achieve these goals through *simplicity*. The intuition is that having a single ROS node at the interface/edge of a subsystem helps in limiting by design the possible interaction points with the other nodes of the system and reduces the cognitive burden of developers during development and maintenance. This can be seen as a way to implement *information hiding* (or promote its practices) in an environment where it is possible to see what is happening beyond the interface because all topics/nodes are public in ROS. Also, information hiding provides the developer of the subsystem the freedom to change the design behind the interface nodes, while maintaining compatibility with clients. For example, the `FlyingROS` system exposes a single ROS node towards a web interface for controlling, configuring and monitoring multicopters and it *"sends every useful information on the web though ROSBridge (to avoid to subscribe to 10 topics from the web app)"* (`AlexisTM/flyingros`).

If the subsystem provides several capabilities, the interface node may become a God node, *i.e.,* a node with an overly complex interface (*i.e.,* its published topics and registered services) and too many responsibilities. Therefore, it is important to design the interface node and its provided interface so that it covers only the most common use cases. In this way, the amount of masked complexity is maximized and the actual complexity of the interface node will be just the complexity of the most commonly used capabilities. This is the strategy applied in the `Moveit` system, where "*the simplest user interface is through the* `MoveGroupInterface` *class. It provides easy to use functionality for most operations that a user may want to carry out, specifically setting joint or pose goals, creating motion plans, moving the*

robot" (AcutronicRobotics/moveit2).

A notable implementation of I3 is the multimaster subsystem of the ROCON project that was introduced in N4. This subsystem contains a *gateway* node which is *"the public frontend for a ROS master and is intended to act much like a gateway on a local area network controlling what is exposed and what is forwarded between the local ROS master and the outside world (other ROS systems). The intention is to generalise this kind of interface beyond tools that previously existed [...] and also make their configuration/usage as simple as possible"* (robotics-in-concert/rocon_multimaster).

I4 – Systems interacting with other non-ROS systems should provide two types of interfaces: a ROS-independent interface for non-ROS systems and a ROS-based interface for ROS tools (*e.g.*, Rviz, Qt). Many ROS-based systems do not live in isolation; they may be running within an ecosystem including non-ROS components which need to interact with the robots. Those external components may include services running in the cloud, mobile apps, non-ROS embedded devices, etc. Despite the flexibility and the various advantages of ROS as communication middleware, developers cannot simply assume that all non-ROS components will adopt ROS-based communication.

This guideline suggests providing a ROS-independent interface for non-ROS systems interacting with the robots. This guideline can be implemented by identifying the ROS nodes living at the interface/edge of the ROS-based subsystem and have them behave like *adapters* [34] between the ROS world and the external world. As an illustrative example, the ROS stack for the Niryo One manipulator (see N1) has *"the external/user layer [on top of the ROS command level]"* (NiryoRobotics/niryo_one_ros). The reader can get insights on which types of high-level interfaces can be implemented by looking at the ones provided by the Niryo One system, which include: (i) a Python interface, (ii) a Blocky [36] interface for graphical programming, (iii) a Modbus [10] interface, (iv) a raw TCP-based interface, and (v) a Web Socket [31] interface based on Rosbridge [19]. We want to highlight that developing such a high number of possible interfaces to non-ROS systems implies additional effort and we are not advocating that a ROS-based system should always provide all of them; rather, we suggest reasoning in advance about the possible external usage scenarios of the system and designing those interfaces accordingly.

The second part of this guideline states that the system should still have a ROS-based interface, even when alternative non-ROS interfaces are available. Indeed, the ROS-based interface is useful when using the development tools provided by the ROS ecosystem, such as simulators, diagnostics frameworks, visualizers, etc. For example, the ROS/IOP Bridge system provides two interfaces: *"one for interfacing with external systems (ROS is masked) and one purely ROS-based for interfacing with ROS tools like Rviz, Qt, etc."* (fkie/iop_core).

Finally, it is important to note that the ROS-based and non-ROS interfaces are not mutually exclusive and can even build on top of each other. The Capabilities package of the ROCON project takes advantage of this design decision and it has *"a Python "client" API that provides an easy-to-use Python interface for interacting with the capability server. Under the hood this client API uses the ROS API"* (osrf/capabilities).

I5 – Identify variation points of the system in advance, and design the system to be extended by third-parties without modifying its core nodes. As with any software-intensive system, ROS-based systems are subject to changes over their lifetime. In this context, it is important to localize as much as possible the parts of system that are more likely to change and to design the system so that it is easy to update those parts, while limiting as much as possible the impact of the changes. Our analysis revealed that this guideline can be implemented at different levels.

The first solution is to have *statically-defined configurations* for allowing developers to easily specify the information changing across executions without modifying the source code or recompiling it. Typically, those configurations follow the YAML, JSON, or XML syntax, are statically edited by developers before bringing up the system, and are either provided as configuration files or as entries in the ROS parameter server.

The second solution we encountered consists of adopting a plugin-based architectural style. In this case, the system has a core which does not change and provides extension points – all features or new functionalities are added to the system via external plugins; plugins can be separately developed and tested by independent third-party developers. An example of this solution is the VIGIR footstep planning framework, where *"the entire footstep planning pipeline has been "pluginized", thus the system can be easily modified by adding or replacing plugins in the pipeline. This allows for adaptive code execution even during run time which is superior to classic configuration-based systems [*i.e., *the first solution described above]."* (team-vigir/vigir_footstep_planning_core). Developers can implement the plugin-based solution using the *pluginlib* ROS package, which provides tools for defining, implementing, and dynamically loading plugins using the ROS build infrastructure [89].

I6 – Logging should be standardized across the project and follow well-defined guidelines. In the context of this study, logging refers to the act of keeping a record of all the relevant events that occur during the execution of a ROS-based system. Typically, logs are either sent to the standard or error output of the running processes, or persisted as files. For the latter, the ROS ecosystem provides a standard file format called *bag* for storing ROS message data exchanged by nodes at run time [85]. A bag file consists of a sequence of records conforming to the same representation used in the network transport layer of ROS. This technical decision makes the recording and playback of bag files very efficient. The ROS ecosystem provides a dedicated package called `rosbag` for managing bag files [94]. This package allows developers to record and playback ROS bag files, summarize, compress their content, decompress their content, etc. When possible, it is suggested to use the bag format for logging information about the exchanged messages because of its efficiency, tool support, and also because it is supported by many packages within the ROS ecosystem, such as visualizers, graphical user interfaces, and simulators.

Another decision point about logging is about the actual content of the logs. We already saw that bag files can store exchanged messages, but logging other types of information might be used for better understanding what is happening at run time, such as relevant events, stack traces, error messages, CPU and memory utilization, and energy consumption. As suggested by a questionnaire participant, *"logging should be standardized across the project and follow well-defined guidelines."* The specific content of the logs can be either application-specific (*e.g.,* an industrial arm grasping a specific object) or generic (*e.g.,* memory consumption of a node). For the latter, developers can reuse already existing monitoring and logging tools such as the `top` or `vmstat` Unix utilities and the ROS Diagnostics package (`ros/diagnostics`).

7.5. Interaction with Hardware and Other Lower-Level Entities

This family of guidelines is about designing how ROS-nodes interact with the hardware platform (*e.g.,* sensors, actuators) in order to improve the overall quality of the system in terms of portability, maintainability, compatibility, and performance. Table 8 presents the guidelines belonging to this family.

Table 8
Guidelines for Interaction with Hardware and Other Lower-Level Entities

ID	Guideline	Quality Requirements	P	U_g	S
H1	Nodes interacting with simulators and hardware devices should provide identical ROS messaging interfaces to the rest of the system.	Portability	G(3) Q(1)	197	R
H2	When possible, design the system to be hardware-independent.	Maintainability Portability Compatibility	G(15)	165	S
H3	Decouple ROS nodes from variations in the execution environment.	Maintainability Portability Compatibility	G(9) Q(1)	119	S
H4	State estimation nodes should support an arbitrary number and different types of sensors.	Compatibility Maintainability	G(3)	73	R
H5	If context-specific (hardware) configuration is needed, then load it at bringup time.	Performance	G(3)	61	R

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 robotics simulators supporting a wide variety of physics engines, robot models, etc. ROS-based simulators enable roboticists to run *software in the loop* (SITL) simulations, where the ROS nodes executed during the simulation are exactly the same as the ones executed when using real robots; the only difference is that events and data coming from the hardware and the physical environment are fed to the system by the simulator (*e.g.,* Gazebo [67]).

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. Quoting the contributors of the AutoRally system, developers can achieve a *"seamless software migration between hardware and simulation"* when following H1. 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.

Also the concept of *time* can be abstracted (*i.e.,* by using the ROS `Time` package) so that either a simulator could be used to report the (simulated) time or the ROS master could provide the (real-world) time. This solution allows ROS Nodes to be agnostic of the (potentially-simulated) environment in which they are executing or even being used while replaying previously-monitored scenarios.

A recurrent strategy to achieve H1 is to have a hardware abstraction layer. For example, in the `ros_control` project

"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 … Through these typed interfaces, this abstraction enables easy introspection, increased maintainability and controllers to be hardware-agnostic."

H2 – Design the system to be as hardware-independent as possible. Whether a ground rover, flying drone, or industrial arm, the majority of ROS-based systems are meant to run onboard a hardware device or at least to interact with it. However, the hardware platform is a prominent source of uncertainty in the design of ROS-based systems, mainly for two reasons. First, today's hardware is typically co-designed with its software counterpart and has a custom design. Second, many ROS-based systems have been conceived to be used in the context of any robotics system, and therefore the hardware and low-level characteristics of the robot cannot be known a priori; for example, this is the case of the suite of real-time SLAM algorithms provided by Google Cartographer (`googlecartographer/cartographer_ros`).

This guideline suggests isolating the hardware-dependent aspects of the system into distinct parts of the system. In this way the uncertainty introduced by the hardware platform is mitigated, thus providing advantages in terms of system maintainability, portability, and compatibility. Roboticists should strive for (i) clearly separating the responsibilities of the software and hardware counterparts of their ROS-based systems, (ii) identifying the edge/layer up to where the hardware characteristics are relevant for the whole system, and (iii) having only hardware-independent nodes and interfaces (*e.g.,* topics and services) above that layer. In this manner, the hardware-specific and hardware-independent subsystems can be developed and tested in parallel, new features can be added to both without interference, and the hardware-independent subsystem could ported to other compatible hardware platforms with relatively low effort.

In our analysis we identified several solutions for achieving hardware independence. We grouped these solutions into three clusters: configuration-based, man-in-the-middle, and layering. *Configuration-based* solutions require the roboticist to provide the hardware characteristics of the robot in a configuration file, and then the hardware-dependent parts of the system are designed to be parametric with respect to the information included in the configuration file. For example, the `MoveIt` system *"can be configured using the ROS parameter server from where it will also get the URDF and SRDF [models] for the robot"* (`ros-planning/moveit2`), where the two models include (among others) the physical geometries, joints, and collision checking information of the robot [103] .

The *man-in-the-middle* solution is used when the hardware-specific part of the system is minimal and can be easily localized. In this case, the solution involves an intermediate ROS node acting as an adapter between the hardware-specific nodes and the rest of the system, while abstracting the hardware-specific information produced/consumed by the hardware-specific nodes. The `BLUEsat` project uses this solution by having a man-in-the-middle node which *"takes the input from two XBox controllers and turns it into a nice standardised format […], this means [developers] can add new controllers by implementing a different version of this node (and not editing the board control stuff)"* (`bluesat/owr_software`). In more complex projects with many hardware-specific components, the *layering* solution may be applied. In this case, a layered architectural style is adopted, where a hardware abstraction layer runs on top of all hardware-specific nodes. The ROS stack for the Niryo One arm described in Section 2 is an example of hardware independence achieved via layering.

H3 – Decouple ROS nodes from variations in the execution environment. In ROS-based systems variability is not confined to the hardware platform, but it can involve many other aspects of the system, such as the operating system running on the hosts, the reachability of cloud-based services, and the availability and configuration of third-party components.

It is suggested to decouple ROS nodes from possible variations in the execution environment so that they can be (i) properly developed and tested without the complexity added by the variation points, (ii) easily ported to different execution environments, and (iii) integrated with other existing software components and services.

Our analysis revealed that a recurrent implementation of this guideline is to make nodes parametric and to resolve the variation points at run time; this solution is similar to the configuration-based solution discussed in H2, but does not deal with hardware only. For example, the VIGIR footstep planning framework uses an extended ROS parameter server that provides additional features such as *"a flexible structure for parameter sets that can be shared via ROS messages and an [efficient] parameter set management"* (`team-vigir/vigir_footstep_planning_core`). While describing H2 we mentioned that the VIGIR footstep planning framework has a plugin-based architecture, and the usage of their extended parameter server allows the system to quickly switch plugins at run time depending on the set of available plugins and the type of terrain encountered by the robot.

It is important to note that as robots often operate in dynamic and partially observable environments (*e.g.,* underwater

unmanned vehicles), the execution environment may not be known *a priori* and may emerge only at run time. The main sources of runtime changes include variations in the type and number of available resources (both hardware and software), connectivity, capabilities provided by external services, etc. It is important to continuously monitor the execution environment of the system to ensure that runtime changes will not lead to unwanted consequences. The COLA2 system for autonomous underwater robots has a dedicated node that checks safety- and system-related parameters at run time and *"if some of [the values of these parameters] are not inside a user-predefined threshold, the system can abort the mission and/or surface the vehicle automatically"* (udg_cirs/cola2_core).

H4 – State estimation nodes should support an arbitrary number and different types of sensors. State estimation refers to the reconstruction of the underlying state of a system given a sequence of measurements and a prior model of the system [6]. In the context of this study, the system is the robot (*e.g.,* a flying vehicle); its state is typically its position, velocity, orientation, etc.; the measurements are the readings of onboard sensors (*e.g.,* accelerometers, gyroscopes, GPS receivers); and the model is a set of mathematical procedures for extracting (probabilistic) estimates based on sensor readings (*e.g.,* a Kalman filter).

Intuitively, the best state estimation algorithms make use of a combination of multiple different sensors [50]. From an architectural perspective, the ROS nodes implementing state estimation algorithms should support an arbitrary number and different types of sensors. This will allow roboticists to achieve more precise state estimations, mainly because of the higher number and diversity of input readings. This is what happens in the RoboJackets IGVC system developed at the Georgia Institute of Technology, where the node implementing a Kalman filter for localizing the robot receives data from an IMU and a GPS receiver; by doing this, in the event of significant errors in the orientation produced by the IMU, the node can *"compensate [the error] with accurate positions [produced by the GPS receiver] and [relatively accurate] accelerations [still produced by the IMU]"* (RoboJackets/igvc-software).

Moreover, if the state estimation node is independent from the number and type of sensors, then the robot can be easily upgraded by adding additional sensors, without needing to spend effort modifying the source code of the node. Thanks to the pub/sub communication style of ROS, this strategy can be implemented in a straightforward manner by (i) fixing the names and message types of the topics subscribed by the state estimation node and (ii) publishing compatible sensor data to the input topics of the state estimation node. This is the solution implemented in the ros_localization project where *"the nodes do not restrict the number of input sources. If, for example, the robot has multiple IMUs or multiple sources of odometry information, the state estimation nodes within robot_localization can support all of them"* (cra-ros-pkg/robot_localization). Moreover, this solution allows developers to also improve the flexibility of their state estimation nodes; for example, the ros_localization project allows developers to control which message fields are fused with the state estimate on a per-sensor basis [63].

H5 – If context-specific (hardware) configuration is needed, then perform it at bringup time. Being configurable greatly improves system compatibility and flexibility. However, roboticists should pay special attention to *when* the operations for configuring the system (*e.g.,* loading configuration files, checking and bringing up registered plugins) are performed. Checking and enacting available configurations while the robots are operational may negatively impact the runtime performance of the system. With this guideline we suggest to roboticists to check the (hardware) context in which the system is running and perform its configuration at bringup time. For example, in the Autorally system *"on startup the chassis interface loads a priority list of controllers from a configuration file"* (AutoRally/autorally). Note that this guideline is different from B5; H5 deals solely with hardware configuration, whereas B5 deals with software configuration.

Even though this guideline emerged from discussions prominently about hardware configuration, it can be generally applied at other abstraction levels. For example, the VIGIR footstep planning framework we discussed in H3 applies this guideline when managing the plugins for the planning algorithms, where *"it is clearly inefficient to request needed plugins from the manager for each single iteration during planning. For this reason, the planner retrieves all plugins once at the beginning of each planning request and initializes them with the given parameters for this request"* (team-vigir/vigir_footstep_planning_core).

7.6. Safety-Critical Concerns

In this family we report the guidelines that are strictly related to safety aspects of a ROS system, such as the management of data produced by high-frequency sensors and real-time communication. Table 9 presents the guidelines belonging to this family.

S1 – ROS nodes should be resilient with respect to the amount and frequency of data received by sensors. De-

Table 9
Guidelines for Safety-Critical Concerns

ID	Guideline	Quality Requirements	P	U_g	S
S1	ROS nodes should be resilient with respect to the amount and frequency of data received by sensors.	Reliability	G(5)	129	R
S2	Use different communication channels and different (hardware and software) platforms depending on the criticality and real-time requirements of the nodes.	Reliability	G(1) Q(1)	80	G
S3	For real-time requirements, collect timestamps from multiple sources (*i.e.,* do not rely on ROS-based timestamps only).	Performance	G(2)	51	S
S4	Provide at least one globally-reachable node capable of receiving run-stop messages and stopping/resetting the whole system.	Reliability Safety	G(3)	49	R

velopers should be aware that hardware devices such as sensors tend to produce data in bursts (*e.g.,* due to changes in the environment), degrade, and become less accurate over time [47]. 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 robotics system more scalable, even at run time. For example, in the `robot_localization` project *"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. All detectors will be publishing detected frontier points on the same topic (*`/detected_points`*)"* (`cra-ros-pkg/robot_localization`).

S2 – Use different communication channels and different (hardware and software) platforms depending on the criticality and real-time requirements of the nodes. This guideline is specific to ROS nodes with real-time requirements [111] and is rooted in two main observations. First, given that many robotics systems involve wireless communication, networking is one of the most important concerns for real-time robotics systems, mainly due to variations in available bandwidth, latency, packet loss, etc. [116]. Second, it is very difficult to meet real-time guarantees over a non-real-time general-purpose platform (*e.g.,* a Raspberry Pi running a standard Ubuntu Linux distribution) [114].

We suggest to roboticists to (i) critically reflect and identify the nodes of the system that have real-time responsibilities (*e.g.,* state estimation, control, etc.), (ii) if possible, isolate real-time nodes from nodes with non-real-time responsibilities (*e.g.,* speech recognition, logging), and (iii) deploy real-time and non-real-time nodes on different platforms and have them communicate using different independent communication channels. This guideline is specially important for mobile robots because safety is a first-class concern in these types of robots and all safety requirements have to be satisfied, while other concerns (*e.g.,* mission completion) can be partially satisfied (or even traded off for safety) [11]. For example, the AutoRally system discussed in H5 uses three different communication channels: (i) a 2.4GHz RC signal for controlling the car, (ii) a standard 802.11 Wifi signal for non-real-time data such as images and diagnostics, and (iii) a 900 MHz XBee Pro signal for sending GPS corrections to the car at 2Hz, the position and velocity of other cars in proximity at 10Hz, and a global run-stop message at 5Hz.

The same separation principle for the communication channels can be applied to the platforms on which the nodes are running. For example, a robot may have (i) a dedicated board running the nodes responsible for the real-time control of the robot, possibly on top of a real-time operating system and (ii) a more affordable Raspberry Pi microcomputer for performing non-real-time tasks on top of a standard operating system. By doing this, real-time control of the robot is guaranteed, while limiting the (generally higher) effort for developing on top of a dedicated platform to only a part of the system.

S3 – For real-time requirements, collect timestamps from as many sources as possible (*i.e.,* do not rely on ROS-based timestamps only). Time synchronization is central in ROS-based systems with real-time requirements. However, just relying on host system clocks is not enough because ROS-based systems are often distributed over a network of

multiple hosts with different characteristics (*e.g.,* a laptop controlling a flying drone) and can manage data produced by high-frequency hardware devices (*e.g.,* an IMU device running onboard the drone).

This guideline suggests that roboticists working on systems with real-time requirements should achieve higher precision in their time-dependent nodes by collecting timestamps from as many sources as possible and suitably synchronizing them according to well-defined policies. In this guideline we do not discuss how to implement time synchronization between nodes because it depends on available hardware, operating systems, and the level of precision required in the project. For example, the nodes running in the DJI Onboard SDK ROS package always consider two different timestamps when receiving a message (`auengagroactu_sens/dji_matrice100_onboard_sdk_ros`): the first timestamp is produced by the ROS communication middleware and is present in the header of the message itself and the second timestamp is produced by the clock of the board running on the drone.

Another interesting example of implementation of S3 is present in the AutoRally system, where GPS timestamps are collected by a time server running in the ground control station and then used onboard the racing cars to synchronize the clock of their IMU sensor (`AutoRally/autorally`).

Finally, time synchronization is also important for non-real-time systems, specially when their executions must be recorded and played back for testing purposes. In those cases, it is important to have an abstract representation of time, to allow simulations to have slower or faster playback, depending on the needs of the developer. ROS 2 supports time synchronization for those use cases via its `ROSTime` abstraction [79].

S4 – Provide at least one globally-reachable node capable of receiving run-stop messages and stopping/resetting the whole system. When one or more robots are operating in the physical environment, a common safety mechanism is to have an emergency button for blocking any hazardous operations such as robot movements, rotation of blades or rotors, or constraining joint motion [61]. We suggest to roboticists to facilitate the implementation of this safety mechanism by deploying at least one globally-reachable ROS node capable of receiving run-stop messages.

We noted in our analysis that several robotics systems in our dataset are using a similar safety mechanism, both via a physical button reachable by the human operator and a logical mechanism triggered via software. For example, the AutoRally system discussed in S3 has an emergency button that allows the developer to simultaneously stop *all* racing cars within radio range. This safety mechanism is implemented by defining a lightweight ROS message for *run-stop* signals and enabling any program in the whole system to publish a run-stop ROS message in case of hazards. In the specific case of the AutoRally project, run-stop signals are also used as heartbeat messages for keeping track of all racing cars within radio range.

Run-stop messages can also be used for pausing system operation, *e.g.,* to allow operators to manually put the system in an acceptable state, and then bring the system back to a safe state. For example, the Franka robotics arm has a mechanism that ensures that *"if a reflex or error occurs on any of the robots, the control loop of all robots will stop until they are recovered"* (`frankaemika/franka_ros`). The recovery of the robots is realized via an additional action server in the same namespace of the ROS node responsible for controlling the robotics arm.

7.7. Data Persistence

This family contains guidelines related to data persistence in a ROS system, including how to persist large raw data such as full resolution videos, how to avoid race conditions when persisting data, and others. Table 10 presents the guidelines belonging to this family.

Table 10
Guidelines for Data Persistence

ID	Guideline	Quality Requirements	P	U_g	S
P1	Avoid persisting raw data (*e.g.,* a full resolution video) if only part of it will be used.	Performance	G(1)	97	R
P2	Avoid race conditions when persisting data received from other ROS nodes within the system.	Maintainability	G(1)	95	G
P3	Use a dedicated node for persisting and querying long-term data.	Maintainability Performance	G(2)	8	S

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 [99]. As informally confirmed by the ROS community [82], data persistence in ROS systems 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 based on future needs and selectively persist only that data. When recording ROS bag files, developers should avoid recording *all* topics within the system, rather only the ones required for subsequent replays. The same principle holds for audio/video data. A clever mechanism for dealing with audio/video data is implemented in the `Multi tracker` project (`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 file size relative to a full resolution video."*

P2 – Avoid race conditions when persisting data received from other ROS nodes within the system. This guideline applies to ROS-based systems where data is persisted to the same data storage in a multi-threaded fashion. In this context, having race conditions may lead to data loss due to undesired sequences of write operations. A typical example of race condition in a ROS-based system is when two independently-running nodes read and write the same parameter in the ROS parameter server; in this case, one of the nodes may overwrite or even delete a valid value in the parameter server before the other node can use it. The same problem applies when independently-running ROS nodes read/write to the same file in the file system, database, shared memory location, etc.

The implementation of this guideline is highly dependent on the technologies used in the persistence layer. However, a generic solution at the application layer is provided by the developers of the `ros_geometry` project, who achieve thread safety by using a set of mutex around the data storage of each frame of the robot. Specifically: (i) the current pose of each frame of the robot is stored in a shared map, (ii) each entry of the map has an associated unique mutex, and (iii) when a ROS node needs to access a specific frame in the map, its mutex is first locked, then the node reads the current pose of the frame, and finally the mutex is unlocked (`ros/geometry2`). It is important to keep in mind that this solution achieves the goal of avoiding race conditions, but it may incur in (i) lower performance due to the cost of frequently locking and unlocking multiple mutexes and (ii) lower maintainability because the logic for managing the mutexes must also be developed, tested, and debugged.

P3 – Use a dedicated node for persisting and querying long-term data. ROS-based systems can produce various types of long-term data, such as a description of the physical constraints of the robot extracted from an initial introspective phase, the global map reconstructed in a SLAM-based mission, historical data used to build predictive models about how the robot will move in the environment, etc. Common characteristics of this data are that (i) it is meant to be persisted in the long term and (ii) it should be queryable by the system at any point in time for future use. It is important to note that in P3 we are not referring to ROS bags because they have been designed for logging system execution in record-and-play scenarios; instead, we are referring to application-level data that is produced or needed by the system for achieving its objectives (*e.g.,* building a map and storing it for future use).

This guideline suggests to roboticists to have dedicated ROS nodes for persisting long-term data. This design decision improves the *maintainability* of the system because the responsibility of persistent data management is highly localized, thus facilitating its independent development with respect to the rest of the system, and it becomes easier to debug and (unit) test in isolation. Moreover, because a file system or database management system is involved, data storage may become a *performance* issue, and therefore centralizing long-term persistence in dedicated ROS nodes helps in isolating performance regressions and, if needed, refactor the system in such a way that data storage nodes are deployed in hosts with high computational power (*e.g.,* in the cloud).

An important decision point for this guideline is how to design the interface of data storage nodes. In this context, it is worth to mention several alternatives, depending on the usage scenarios of the data storage node: (i) publish a dedicated topic if the node performs infrequent write operations, (ii) register a service if the node performs infrequent read operations of small-sized data (recall that service calls are blocking for caller nodes), (iii) register a service with a persistent connection if the node performs frequent read/write operations involving high amounts of data, (iv) use ROS actions if the node tends to return high amounts of data (in this manner, the progress of the query can be observed from the calling node).

7.8. Discussion

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. Indeed, there may be other sources for guidelines, such as robotics textbooks, interviews with robotics experts working on commercial products, etc. Moreover, out of the 47 guidelines, 26 came uniquely from open-source repositories, 7 came uniquely from surveys, and 14 came from both sources. Therefore, we can say that versioning systems can be considered as a good source for architecting guidelines. Interestingly, repositories on GitLab and

BitBucket were filtered out in Phase 1 of the study. More specifically, from the 46 GitLab repositories in the initial search (step 1), we had 0/46 GitLab repositories included after the first six steps, with 3 discarded because they were fork repositories (step 2), 28 because they had less than 100 commits (step 3), 12 because they had less than 2 stars (step 4), and 3 because they contained only tools (step 6). From the initial 527 BitBucket repositories (step 1), we had to discard 42 because they were fork repositories (step 2), 221 because they had less than 100 commits (step 3), 145 because they had less then 2 stars (step 4), 15 because they contained only tools (step 6), 9 because they were simulation-based repositories (step 7), 72 because they did not contain at least one launch file (step 9), and finally 23 repositories were discarded after manual selection because they did not satisfy the selection criteria reported in Table 1.

The results also show a positive correlation between the ranked usefulness of a guideline and the ranked number of mentions in GitHub. To show this, we calculated the Spearman Rank Coefficient. Ranking all guidelines together, we get a coefficient of $Rs = 0.0.422$ with p-value of 0.01 (99% confidence), which is a moderate positive correlation. This suggests that the most useful guidelines are also followed in many projects.

We posit that these guidelines are generally applicable, but they should be used in context, *e.g.,* C2 and C4 are potentially in conflict – if a developer wants to observe the status of hosts on the network (C4), the ROS node responsible for doing this must be aware of the network topology, therefore 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 context-sensitive trade-offs.

For the most part, we have not discriminated between guidelines relating to ROS 1 and those relating to ROS 2. At the time of writing, ROS 2 is still under active development, but we believe that many of the guidelines apply to both versions. ROS 2 emerged from a need to better support more robot use cases than ROS 1, which was primarily designed for single robot use case scenarios (although ROS 1 has been used in many contexts), including real-time and embedded systems, non-ideal networks, and production environments [35]. When applicable (*e.g.,* C11, C12, B1), we have illustrated how the application of some guidelines might be different in ROS 2.

Furthermore, we have assessed each guideline with respect to its specificity to ROS. We categorized this in three ways: (1) General guidelines that would apply to any software project (e.g., I1 which talks about meaningful naming), (2) General guidelines that are specialized to ROS systems (e.g., C2 about hiding the implemented communication infrastructure), and (3) Guidelines that only apply to ROS systems (e.g., C10 recommending empty messages when triggering atomic actions). Of the 47 guidelines, we categorized 16 as general, 16 as general guidelines specialized for ROS, and 15 as ROS-specific guidelines.

8. Threats to Validity

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. We also acknowledge that the repositories in our dataset may not cover all possible types of robots or capabilities. 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* recent contributors to the repositories in our dataset, they still might not be representative of the entire roboticist 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 [123, 49, 112]. The replicability of the study and independent verification of its findings are ensured by documenting each phase of the study in a publicly available replication package. The qualitative analysis for answering our RQs 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. The 39 guidelines that emerged from Phase 1 were scrutinized by 119 independent roboticists and the additional 8 emerging from Phase 2 were directly proposed by roboticists working on real projects. It should be noted that the online questionnaire was filled out by those roboticists who had contributed to the repositories from which the guidelines were extracted; this might have induced an acceptance bias. To increase confidence in the quality of our final

set of guidelines, we collected feedback on all the 47 guidelines from two volunteers working on robotics systems for several years. We further recognize that there are other sources of developer guidelines being produced, formally and informally, by the ROS community [87, 91, 92] and therefore our guidelines may provide only a partial view. However, we note that most of these developer guidelines are much more lower-level in nature and would be a complement rather than a replacement to our architecture guidelines by providing implementation details or examples. Finally, during our analysis we might have missed some external data sources because they are not linked in the repositories within our dataset, *e.g.,* the default page of a ROS package in the official ROS Wiki page or generated API documentation hosted on third-party web servers. Similarly, those data sources can be seen as complementary to our findings.

Conclusion validity. Third-party researchers can verify and check the obtained results by replicating our study independent from our results. This is possible thanks to the complete replication package, which is publicly available. Another potential threat to conclusion validity may lie in our definition of the guidelines and architectural concerns. Indeed, other researchers may identify different text fragments from the system documentation and different concerns, thus potentially leading to totally different results. We mitigated this potential threat to validity by (i) carefully documenting the process we followed for building the guidelines (see Sections 3.1.2 and 3.2.2), (ii) having the data extraction process conducted by multiple researchers in collaboration, and (iii) making the raw extracted data available for independent verification.

Construct validity. It is important for the mining pipeline for searching and filtering ROS-based projects from the code hosting platforms to be 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 already knew the expected outcomes, and by making the implementation of the complete pipeline publicly available in the replication package.

9. Related Work

Studies on ROS. Because ROS (i) is under active development, (ii) is a de facto standard for development of robots, and (iii) 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 [108] and other research results [122, 72] 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.

In a recent study we mined the ROS ecosystem for quantifying and characterizing the main causes, solutions, and possible trade-offs of *energy-related issues* in ROS-based systems [1]. The study targets (i) the source code, documentation, commit messages, issues, and pull requests of the 335 Git repositories considered in this study and (ii) discussions on StackOverflow and official technical forums used by ROS developers (*e.g.,* ROS Answers and ROS Discourse).

To design good quality robotics software in ROS, the work by Halder [41] and Curran [21] 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.* [70] 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. 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 functionalities. 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 5), which supports this observation. Similarly, Curran *et al.* [21] provide a visualization tool with runtime 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 looking at code artifacts, targeting their official documentation and, more importantly, collecting the perspectives of roboticists working on real ROS projects.

Mining Architectural Knowledge. 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 (manually [118] or by applying machine learning techniques [9]), issue trackers [52], as well as developer discussions in community groups such as StackOverflow [113] or 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 (*e.g.,* heartbeat, audit, checkpoint) has also been investigated. In fact, these can be the areas of code that change the most [59] [60]. As 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. As an initial step towards this direction, in a recent study [54] we built a similar dataset as the one reported in [1], and we (i) mined four architectural tactics for the energy efficiency of ROS-based systems, (ii) implemented each tactic in a real robot running a common ROS stack, and (iii) performed an empirical evaluation of the runtime impact of each tactic in terms of energy consumption. The obtained results are promising for this line of research since we observed that the tactics significantly help to improve the energy efficiency of the robot.

10. Conclusions and Future Work

This paper described a study to elicit evidence-based architectural guidelines for open-source ROS-based software for robots. We were interested in studying three research questions: *What quality requirements are considered when architecting ROS-based systems?* (**RQ1**), *How do roboticists document the software architecture of ROS-based systems?* (**RQ2**), and *What guidelines are followed by roboticists when architecting ROS-based systems?* (**RQ3**). We found that the most frequently discussed quality requirements were maintenance, performance, and reliability. We also found that of the relevant repositories, only 16.4% documented their architecture. We elicited 39 guidelines derived from the repositories, and 8 more from the online questionnaire. By surveying 119 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 for our future work on determining architectural tactics for high-quality robotics software, and thereby providing a solid engineering basis for developing robots in a future in which they are ubiquitous. To achieve this, we plan to (i) explore how existing architectural tactics found in Bass *et al.* [7] 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 [108, 122], and (iii) provide more comprehensive guidance for architecting high-quality robotics software.

Acknowledgments

This research is partially supported by the Dutch Research Council (NWO) through the OCENW.XS2.038 grant. This research 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] Albonico, M., Malavolta, I., Pinto, G., Guzmán, E., Chinnappan, K., Lago, P., 2021. Mining Energy-Related Practices in Robotics Software, in: Proceedings of the 18th International Conference on Mining Software Repositories, MSR, IEEE/ACM, New York, NY.

[2] Aldrich, J., Garlan, D., Kastner, C., Le Goues, C., Mohseni-Kabir, A., Ruchkin, I., Samuel, S., Schmerl, B., Timperley, C.S., Veloso, M., et al., 2019. Model-based adaptation for robotics software. IEEE Software 36, 83–90.

[3] Alkadhi, R., Nonnenmacher, M., Guzman, E., Bruegge, B., 2018. How do developers discuss rationale?, in: International Conference on Software Analysis, Evolution and Reengineering, ACM. pp. 357–369.

[4] Arduino, 2020. Arduino - ArduinoUno. URL: https://www.arduino.cc/en/Guide/ArduinoUno. [Online; accessed 30. Mar. 2020].

[5] Arvanitou, E.M., Ampatzoglou, A., Chatzigeorgiou, A., Carver, J.C., 2021. Software engineering practices for scientific software development: A systematic mapping study. The Journal of systems and software 172, 110848.

[6] Barfoot, T., 2017. State estimation for robotics. Cambridge University Press.

[7] Bass, L., Clements, P., Kazman, R., 2012. Software Architecture in Practice. 3rd ed., Addison-Wesley Professional.

[8] Bedu, L., Tinh, O., Petrillo, F., 2019. A tertiary systematic literature review on Software Visualization, in: 2019 Working Conference on Software Visualization (VISSOFT), IEEE. pp. 33–44.

[9] Bhat, M., Shumaiev, K., Biesdorf, A., Hohenstein, U., Matthes, F., 2017. Automatic Extraction of Design Decisions from Issue Management Systems: A Machine Learning Based Approach, in: Lopes, A., de Lemos, R. (Eds.), Software Architecture, Springer. Springer International Publishing.

[10] Boyer, S.A., 2009. SCADA: supervisory control and data acquisition. International Society of Automation.

[11] Bozhinoski, D., Di Ruscio, D., Malavolta, I., Pelliccione, P., Crnkovic, I., 2019. Safety for mobile robotic systems: A systematic mapping study from a software engineering perspective. Journal of Systems and Software 151, 150–179.

[12] Brugali, D., Prassler, E., 2009. Software engineering for robotics [From the Guest Editors]. IEEE Robotics & Automation Magazine 16, 9–15.

[13] Butler, S., Wermelinger, M., Yu, Y., Sharp, H., 2010. Exploring the Influence of Identifier Names on Code Quality: An Empirical Study, in: European Conference on Software Maintenance and Reengineering, IEEE. IEEE CS. pp. 156–165.

[14] Cherrier, S., Ghamri-Doudane, Y.M., Lohier, S., Roussel, G., 2012. Services collaboration in wireless sensor and actuator networks: orchestration versus choreography, in: 2012 IEEE Symposium on Computers and Communications (ISCC), IEEE. pp. 000411–000418.

[15] Ciccozzi, F., Di Ruscio, D., Malavolta, I., Pelliccione, P., Tumova, J., 2017. Engineering the software of robotic systems, in: International Conference on Software Engineering Companion (ICSE-C), IEEE. pp. 507–508.

[16] Cimino, M.G., Lega, M., Monaco, M., Vaglini, G., 2019. Adaptive exploration of a UAVs swarm for distributed targets detection and tracking, in: The 8th International Conference on Pattern Recognition Applications and Methods (ICPRAM 2019), IEEE. pp. 1–8.

[17] Clements, P., Garlan, D., Bass, L., Stafford, J., Nord, R., Ivers, J., Little, R., 2002. Documenting software architectures: views and beyond. Pearson Education.

[18] Consortium, C.C., 2020. From Internet to robotics: A roadmap for US robotics: 2020 Edition. URL: http://www.hichristensen.com/pdf/roadmap-2020.pdf. [Online; accessed 29. Oct. 2020].

[19] Crick, C., Jay, G., Osentoski, S., Pitzer, B., Jenkins, O.C., 2017. Rosbridge: Ros for non-ros users, in: Robotics Research. Springer, pp. 493–504.

[20] Cruzes, D.S., Dyba, T., 2011. Recommended Steps for Thematic Synthesis in Software Engineering, in: 2011 International Symposium on Empirical Software Engineering and Measurement, ACM. pp. 275–284.

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

[22] Di Nitto, E., Di Penta, M., Gambi, A., Ripa, G., Villani, M.L., 2007. Negotiation of service level agreements: An architecture and a search-based approach, in: International Conference on Service-Oriented Computing, Springer. pp. 295–306.

[23] Dieber, B., White, R., Taurer, S., Breiling, B., Caiazza, G., Christensen, H., Cortesi, A., 2020. Penetration Testing ROS. Springer International Publishing, Cham. pp. 183–225. URL: https://doi.org/10.1007/978-3-030-20190-6%5F8, doi:10.1007/978-3-030-20190-6_8.

[24] Dunkley, O., Engel, J., Sturm, J., Cremers, D., 2014. Visual-inertial navigation for a camera-equipped 25g nano-quadrotor, in: IROS2014 aerial open source robotics workshop, IEEE. p. 2.

[25] Edwards, J., 2009. Coherent reaction, in: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications, ACM. pp. 925–932.

[26] Ellingson, G., McLain, T., 2017. ROSplane: Fixed-wing autopilot for education and research, in: 2017 International Conference on Unmanned Aircraft Systems (ICUAS), IEEE. pp. 1503–1507.

[27] Emika, F., 2020. FRANKA EMIKA - Support. URL: https://frankaemika.github.io. [Online; accessed 6. Apr. 2020].

[28] Estefo, P., Robbes, R., Fabry, J., 2015. Code duplication in ROS launchfiles, in: 2015 34th International Conference of the Chilean Computer Science Society (SCCC), ACM. pp. 1–6.

[29] Estefo, P., Simmonds, J., Robbes, R., Fabry, J., 2019a. The Robot Operating System: Package reuse and community dynamics. Journal of Systems and Software 151, 226–242.

[30] Estefo, P., Simmonds, J., Robbes, R., Fabry, J., 2019b. The Robot Operating System: Package reuse and community dynamics. Journal of Systems and Software 151, 226–242.

[31] Fette, I., 2020. The WebSocket Protocol. URL: https://tools.ietf.org/html/rfc6455. [Online; accessed 9. Apr. 2020].

[32] Fleuren, T., Gotze, J., Muller, P., 2011. Workflow skeletons: increasing scalability of scientific workflows by combining orchestration and choreography, in: 2011 IEEE Ninth European Conference on Web Services, IEEE. pp. 99–106.

[33] Foote, T., 2018. ROS Community Metrics Report. URL: http://download.ros.org/downloads/metrics/metrics-report-2018-07.pdf.

[34] Gamma, E., Johnson, R., Vlissides, J., Helm, R., 1995. Design patterns: elements of reusable object-oriented software .

[35] Gerkey, B., 2019. Why ROS 2? URL: https://design.ros2.org/articles/why%5Fros2.html.

[36] Google, 2020a. Blockly - Google Developers. URL: https://developers.google.com/blockly. [Online; accessed 9. Apr. 2020].

[37] Google, 2020b. googletest. URL: https://github.com/google/googletest. [Online; accessed 4. Apr. 2020].

[38] Gousios, G., 2013. The GHTorrent dataset and tool suite, in: Proceedings of the 10th Working Conference on Mining Software Repositories, IEEE. IEEE Press, Piscataway, NJ, USA. pp. 233–236.

[39] Guiochet, J., 2016. Hazard analysis of human–robot interactions with HAZOP–UML. Safety science 84, 225–237.

[40] H2020, E., 2016. Robotics 2020 Multi-Annual Roadmap For Robotics in Europe. URL: http://sparc-robotics.eu/wp-content/uploads/2014/05/H2020-Robotics-Multi-Annual-Roadmap-ICT-2016.pdf.

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

[42] Harrison, W., Downs, A., Schlenoff, C., 2018. The Agile Robotics for Industrial Automation Competition. AI Magazine 39, 77.

[43] Hofmeister, J., Siegmund, J., Holt, D.V., 2017. Shorter identifier names take longer to comprehend, in: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE. pp. 217–227.

[44] Holzmann, 2013. Landing a Spacecraft on Mars. {IEEE} Software 30, 83–86.

[45] Honig, W., Ayanian, N., 2017. Flying Multiple UAVs Using ROS. Robot Operating System (ROS): The Complete Reference 2, 83.

[46] ISO/IEC, 2010. ISO/IEC 25010 System and software quality models. Technical Report.

[47] Jamshidi, P., Cámara, J., Schmerl, B., Kastner, C., Garlan, D., 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), IEEE.

[48] Jiang, H., Elbaum, S., Detweiler, C., 2017. Inferring and monitoring invariants in robotic systems. Autonomous Robots 41, 1027–1046.

[49] Kalliamvakou, E., Gousios, G., Blincoe, K., Singer, L., German, D.M., Damian, D., 2016. An in-depth study of the promises and perils of mining GitHub. Empirical Software Engineering 21, 2035–2071.

[50] Khaleghi, B., Khamis, A., Karray, F.O., Razavi, S.N., 2013. Multisensor data fusion: A review of the state-of-the-art. Information fusion 14, 28–44.

[51] Lamarre, O., Limoyo, O., Marić, F., Kelly, J., 2020. The Canadian Planetary Emulation Terrain Energy-Aware Rover Navigation Dataset. The International Journal of Robotics Research 39, 641–650.

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

[53] Lidwell, W., Holden, K., Butler, J., 2010. Universal principles of design. Rockport Pub.

[54] Malavolta, I., Chinnappan, K., Swanborn, S., Lewis, G., Lago, P., 2021a. Mining the ROS ecosystem for Green Architectural Tactics in Robotics and an Empirical Evaluation, in: Proceedings of the 18th International Conference on Mining Software Repositories, MSR, ACM, New York, NY.

[55] Malavolta, I., Lewis, G.A., Schmerl, B., Lago, P., Garlan, D., 2020. How do you Architect your Robots? State of the Practice and Guidelines for ROS-based System, in: Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice, ACM/IEEE.

[56] Malavolta, I., Lewis, G.A., Schmerl, B., Lago, P., Garlan, D., 2021b. Mining guidelines for architecting robotics software – replication package. URL: https://github.com/S2-group/jss-2021-replication-package. [Online; accessed 14. Apr. 2021].

[57] Malek, S., Mikic-Rakic, M., Medvidovic, N., 2005. A style-aware architectural middleware for resource-constrained, distributed systems. IEEE Transactions on Software Engineering 31, 256–272.

[58] Meyer, B., 1992. Applying'design by contract'. Computer 25, 40–51.

[59] Mirakhorli, M., Cleland-Huang, J., 2015. Modifications, Tweaks, and Bug Fixes in Architectural Tactics, in: IEEE/ACM Working Conference on Mining Software Repositories, IEEE. pp. 377–380.

[60] Mirakhorli, M., Fakhry, A., Grechko, A., Wieloch, M., Cleland-Huang, J., 2014. Archie: A Tool for Detecting, Monitoring, and Preserving Architecturally Significant Code, in: ACM SIGSOFT International Symposium on Foundations of Software Engineering, ACM. ACM. pp. 739–742.

[61] Mitka, E., Gasteratos, A., Kyriakoulis, N., Mouroutsos, S.G., 2012. Safety certification requirements for domestic robots. Saf. Sci. 50, 1888–1897. doi:10.1016/j.ssci.2012.05.009.

[62] Montesi, F., Weber, J., 2016. Circuit breakers, discovery, and API gateways in microservices. Preprint arXiv:1609.05830 .

[63] Moore, T., Stouch, D., 2016. A generalized extended kalman filter implementation for the robot operating system, in: Intelligent autonomous systems 13. Springer, pp. 335–348.

[64] MoveIt.org, 2020. moveit - ROS Planning. URL: https://github.com/ros-planning/moveit. [Online; accessed 6. Apr. 2020].

[65] Myers, G.J., Sandler, C., Badgett, T., 2011. The art of software testing. John Wiley & Sons.

[66] Niryo, 2020. Get started with the Niryo One ROS stack - Niryo. URL: https://niryo.com/docs/niryo-one/developer-tutorials/get-started-with-the-niryo-one-ros-stack. [Online; accessed 15. Apr. 2020].

[67] OSRF, 2020. Gazebo. URL: http://gazebosim.org. [Online; accessed 30. Mar. 2020].

[68] Pelliccione, P., Knauss, E., Heldal, R., Ågren, S.M., Mallozzi, P., Alminger, A., Borgentun, D., 2017. Automotive architecture framework: The experience of volvo cars. Journal of systems architecture 77, 83–100.

[69] Peltz, C., 2003. Web services orchestration and choreography. Computer 36, 46–52.

[70] Pichler, M., Dieber, B., Pinzger, M., 2019. Can I Depend on you? Mapping the Dependency and Quality Landscape of ROS Packages, in: IEEE International Conference on Robotic Computing (IRC), IEEE. pp. 78–85.

[71] Pichler, M., Dieber, B., Pinzger, M., 2020. rosmap: ROS package dependency scanner. URL: https://github.com/jr-robotics/rosmap.

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

[73] Pyo, Y., Nakashima, K., Kuwahata, S., Kurazume, R., Tsuji, T., Morooka, K., Hasegawa, T., 2015. Service robot system with an informationally structured environment. Rob. Auton. Syst. 74, 148–165. doi:10.1016/j.robot.2015.07.010.

[74] Python, 2020. unittest — Unit testing framework — Python 3.8.2 documentation. URL: https://docs.python.org/3/library/unittest.html. [Online; accessed 4. Apr. 2020].

[75] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y., 2009. ROS: an open-source Robot Operating System, in: ICRA workshop on open source software, Kobe, Japan. p. 5.

[76] Quigley, M., Gerkey, B., Smart, W.D., 2015. Programming Robots with ROS: a practical introduction to the Robot Operating System. O'Reilly

Media, Inc.
[77] Raza, S.J.A., Gupta, N.A., Chitaliya, N., Sukthankar, G.R., 2018. Real-World Modeling of a Pathfinding Robot Using Robot Operating System (ROS). arXiv preprint arXiv:1802.10138 .
[78] Redis, 2020. Redis. URL: https://redis.io. [Online; accessed 5. Apr. 2020].
[79] ROS 2 Design, 2020a. Clock and Time. URL: https://design.ros2.org/articles/clock_and_time.html. [Online; accessed 14. Apr. 2020].
[80] ROS 2 Design, 2020b. Intra-process Communications in ROS 2. URL: http://design.ros2.org/articles/intraprocess_communications.html. [Online; accessed 31. Mar. 2020].
[81] ROS 2 Design, 2020c. Managed nodes. URL: https://design.ros2.org/articles/node_lifecycle. [Online; accessed 30. Mar. 2020].
[82] ROS Answers, 2020. rosbag record performance issues - ROS Answers: Open Source Q&A Forum. URL: https://answers.ros.org/question/266095/rosbag-record-performance-issues. [Online; accessed 30. Mar. 2020].
[83] ROS Wiki, 2019. ROS Wiki Documentation. URL: http://wiki.ros.org.
[84] ROS Wiki, 2020a. Actions. URL: http://design.ros2.org/articles/actions.html. [Online; accessed 7. Apr. 2020].
[85] ROS Wiki, 2020b. Bags. URL: http://wiki.ros.org/Bags. [Online; accessed 9. Apr. 2020].
[86] ROS Wiki, 2020c. common_msgs. URL: https://wiki.ros.org/common_msgs. [Online; accessed 30. Mar. 2020].
[87] ROS Wiki, 2020d. Index of ROS Enhancement Proposals. URL: https://ros.org/reps/rep-0000.html. [Online; accessed 19. Jul. 2020].
[88] ROS Wiki, 2020e. nodelet. URL: http://wiki.ros.org/nodelet. [Online; accessed 31. Mar. 2020].
[89] ROS Wiki, 2020f. pluginlib. URL: http://wiki.ros.org/pluginlib. [Online; accessed 9. Apr. 2020].
[90] ROS Wiki, 2020g. Quality/Tutorials/UnitTesting. URL: http://wiki.ros.org/Quality/Tutorials/UnitTesting. [Online; accessed 4. Apr. 2020].
[91] ROS Wiki, 2020h. ROS Answers. URL: https://answers.ros.org/questions/. [Online; accessed 19. Jul. 2020].
[92] ROS Wiki, 2020i. ROS Best Practices. URL: http://wiki.ros.org/BestPractices. [Online; accessed 19. Jul. 2020].
[93] ROS Wiki, 2020j. ROS Conventions. URL: http://wiki.ros.org/ROS/Patterns/Conventions. [Online; accessed 20. Jul. 2020].
[94] ROS Wiki, 2020k. rosbag. URL: http://wiki.ros.org/rosbag. [Online; accessed 9. Apr. 2020].
[95] ROS Wiki, 2020l. ROS/Technical Overview. URL: http://wiki.ros.org/ROS/Technical%20Overview. [Online; accessed 30. Mar. 2020].
[96] ROS Wiki, 2020m. rostest. URL: http://wiki.ros.org/rostest. [Online; accessed 4. Apr. 2020].
[97] ROS Wiki, 2020n. rostopic. URL: http://wiki.ros.org/rostopic. [Online; accessed 30. Mar. 2020].
[98] ROS Wiki, 2020o. rplidar. URL: http://wiki.ros.org/rplidar. [Online; accessed 2. Apr. 2020].
[99] ROS Wiki, 2020p. rtabmap_ros. URL: http://wiki.ros.org/rtabmap_ros. [Online; accessed 30. Mar. 2020].
[100] ROS Wiki, 2020q. Services. URL: http://wiki.ros.org/Services. [Online; accessed 30. Mar. 2020].
[101] ROS Wiki, 2020r. SROS. URL: http://wiki.ros.org/action/show/SROS.
[102] ROS Wiki, 2020s. std_msgs. URL: https://wiki.ros.org/std_msgs. [Online; accessed 30. Mar. 2020].
[103] ROS Wiki, 2020t. URDF. URL: http://wiki.ros.org/urdf. [Online; accessed 10. Apr. 2020].
[104] ROS Wiki, 2020u. vision_opencv. URL: http://wiki.ros.org/vision_opencv. [Online; accessed 8. Apr. 2020].
[105] rosmilitary.org, 2020. ROS-M. URL: https://rosmilitary.org/.
[106] ROS.org, 2020. About ROS. URL: https://www.ros.org/about-ros. [Online; accessed 6. Apr. 2020].
[107] Ruscio, D.D., Malavolta, I., Pelliccione, P., Tivoli, M., 2016. Automatic Generation of detailed Flight Plans from High-level Mission Descriptions, in: ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MODELS), ACM. pp. 45–55.
[108] Santos, A., Cunha, A., Macedo, N., 2019. Static-Time Extraction and Analysis of the ROS Computation Graph, in: 2019 Third IEEE International Conference on Robotic Computing (IRC), IEEE. pp. 62–69.
[109] SensComp, Inc., 2020. Series 7000 Ultrasonic Sensors - SensComp. URL: http://www.senscomp.com/ultrasonic-sensors/series-7000-sensors.php. [Online; accessed 2. Apr. 2020].
[110] Shaw, M., Garlan, D., 1996. Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.
[111] Shin, K.G., Ramanathan, P., 1994. Real-time computing: A new discipline of computer science and engineering. Proceedings of the IEEE 82, 6–24.
[112] Shull, F., Singer, J., Sjøberg, D.I., 2007. Guide to advanced empirical software engineering. Springer.
[113] Soliman, M., Rekaby Salama, A., Galster, M., Zimmermann, O., Riebisch, M., 2018. Improving the Search for Architecture Knowledge in Online Developer Communities, in: IEEE International Conference on Software Architecture (ICSA), IEEE.
[114] Stankovic, J.A., Ramamritham, K., 1989. The Spring kernel: a new paradigm for real-time operating systems. ACM SIGOPS Operating Systems Review 23, 54–71.
[115] Swanborn, S., Malavolta, I., 2020. Energy Efficiency in Robotics Software: A Systematic Literature Review, in: 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW '20), ACM. pp. 137–144.
[116] Thiele, L., Chakraborty, S., Naedele, M., 2000. Real-time calculus for scheduling hard real-time systems, in: 2000 IEEE International Symposium on Circuits and Systems. Emerging Technologies for the 21st Century. Proceedings (IEEE Cat No. 00CH36353), IEEE. pp. 101–104.
[117] Vaismoradi, M., Turunen, H., Bondas, T., 2013. Content analysis and thematic analysis: Implications for conducting a qualitative descriptive study. Nursing & health sciences 15, 398–405.
[118] van der Ven, J.S., Bosch, J., 2013. Making the Right Decision: Supporting Architects with Design Decision Data, in: Drira, K. (Ed.), Software Architecture, Springer. Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 176–183.

[119] Vilches, V.M., Kirschgens, L.A., Calvo, A.B., Cordero, A.H., Pisón, R.I., Vilches, D.M., Rosas, A.M., Mendia, G.O., Juan, L.U.S., Ugarte, I.Z., et al., 2018. Introducing the robot security framework (RSF), a standardized methodology to perform security assessments in robotics, in: Proceedings of the Symposium on Blockchain for Robotic Systems 2018. URL: https://arxiv.org/abs/1806.04042.

[120] Willow Garage, 2016. Overview - Willow Garage. URL: http://www.willowgarage.com/pages/pr2/overview. [Online; accessed 6. Apr. 2020].

[121] Wireshark, 2020. Wireshark · Go Deep. URL: https://www.wireshark.org. [Online; accessed 30. Mar. 2020].

[122] Witte, T., Tichy, M., 2018. Checking Consistency of Robot Software Architectures in ROS, in: 2018 IEEE/ACM 1st International Workshop on Robotics Software Engineering (RoSE), IEEE. pp. 1–8.

[123] Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., Wesslen, A., 2012. Experimentation in Software Engineering. Computer Science, Springer.

[124] Wohlrab, R., Knauss, E., Pelliccione, P., 2020. Why and how to balance alignment and diversity of requirements engineering practices in automotive. The Journal of systems and software 162, 110516.