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

Ivano Malavolta†, Grace Lewis‡, Bradley Schmerl§, Patricia Lago¶, David Garlan‖

†Vrije Universiteit Amsterdam, The Netherlands - i.malavolta | p.lago@vu.nl
‡ Software Engineering Institute, Carnegie Mellon University, USA - glewis@sei.cmu.edu
§ Institute for Software Research, Carnegie Mellon University, USA - {schmerl | garlan}@cs.cmu.edu

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

ACM Reference Format:

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

One emerging standard framework for developing robotic software is the Robot Operating System (ROS) [30], a set of open source libraries and tools for developing modular robotic functions that communicate with each other in a loosely-coupled, multi-process, distributed environment. ROS also has a set of mature tools for managing software builds and deployment, simulating production environments for testing and development, and sharing common packages. Though it is hard to estimate the current adoption of ROS, as of writing there are 5941 papers that cite the seminal paper on ROS [30], and ROS supports more than 140 different types of robots (http://ros.org).

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

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

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

The main contributions of this study are: (1) A characterization of the state of the practice with respect to the architecture of ROS-based systems; (2) A set of 49 guidelines and the quality requirements that these guidelines are concerned with; (3) A validation of these guidelines from 77 roboticists who were active committers to these projects; and (d) The replication package for the study.
There are two main targets for the results of this study: (1) roboticists who want to apply good design principles to develop robots that meet quality requirements, and (2) architecture researchers who can use them as evidence-based indications about how real-world ROS systems should be architected, thus inspiring future research contributions such as automated architectural analysis.

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 [15]. ROS 1 is currently evolving into ROS 2 to address this broader ROS community. Some of the new features in ROS 2 include better support for teams of multiple robots and real-time requirements, as well as improved APIs.

![Figure 1: ROS Node Communication](image)

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 [31]. Nodes communicate with each other using a publish/subscribe model based on Topics, or using a request/reply model based on Service Invocations, as shown in Figure 1.¹

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 due to page limitations, such as nodelets and namespaces [30], but these are not necessary as background for this paper.

![Figure 2: Example of a ROS software architecture](image)

ROS-based systems are typically started using a launch file. The launch file is used by the roslaunch tool [31] to start the ROS Master and all system Nodes, set parameters in the Parameter Server, and perform any other initialization required by the system. One of the main differences between ROS 1 and ROS 2 is that launch files are written in XML in ROS 1 and in Python in ROS 2.

Figure 2 shows the architecture for a computer vision subsystem, 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. More sample architectures can be found in the replication package for this study (see Section 3).

3 STUDY DESIGN

This study aims at answering three main research questions.

RQ1 – What are the architecture-relevant characteristics of open-source ROS-based systems?

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

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

RQ1.3 – To what extent do roboticists document the software architecture of ROS-based systems? The benefits of architecture documentation have been largely discussed [10]. However, given the wide adoption of agile methods [18], 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.

RQ2 – What quality requirements are considered when architecting ROS-based systems? With this question we aim at supporting roboticists’ understanding of the quality requirements that are potentially impacted the most by their architectural decisions. Even though modern robotics systems have to cope with a large number of quality requirements [2], it appears that the software engineering community tends to primarily focus on performance measurement and functional issues, while neglecting other crucial quality requirements, such as maintainability, energy efficiency, and safety [7, 9]. With this research question we aim at objectively assessing this phenomenon in the context of real-world robotic projects.

RQ3 – How to guide roboticists when architecting ROS-based systems? The goal is to provide a catalog of guidelines for architecting ROS-based systems. In this study a guideline is defined as the “information intended to advise [architects] on how something should be done or what something should be” [36]. Roboticists can use the catalog as a source of actionable guidance for architecting their systems.
systems. Researchers can use it as a solid foundation for developing new scientific contributions, e.g., techniques to detect violations of the guidelines.

Figure 3 presents the overview of the study design. It follows a mixed method research methodology involving two sequential phases. Phase 1 produces preliminary results via a quantitative and qualitative analysis of data mined from open-source repositories (see Section 3.1), and Phase 2 complements the results of Phase 1 via an online survey, which targets roboticists contributing to real ROS-based projects (see Section 3.2). To allow independent replication and verification of the study, we provide a full replication package\(^2\) including the details of the research protocol, sample architectures, raw data, mining scripts in Python, and data analysis scripts in R.

### 3.1 Phase 1: Mining Software Repositories

#### 3.1.1 Dataset Construction

Because the community around ROS has always encouraged open-source development in the form of publicly available packages [14], we can consider open-source software repositories as good data sources for characterizing the practices of roboticists in the context of real-world projects.

Figure 4 shows the steps for building the dataset of our study. The initial search (step 1) is based on (i) rosumap, a ROS dependency analysis tool that can also identify ROS-based repositories in GitHub, Bitbucket, and Gitlab [28] and (ii) gftorrent, a widely-used queryable mirror of GitHub [16]. Then we filtered out fork repositories to avoid duplication (step 2). As suggested in [23], we excluded repositories with less than 100 commits (step 3) and less than 2 stars (step 4) to avoid inactive or non-maintained projects. Since we are targeting real-world projects involving ROS systems used in real contexts, in steps 5, 6, and 7 we filtered out all repositories containing demos, development tools, simulators (or software runnable only on simulators), respectively; in these steps a repository was discarded if either its name, description, or unique identifier matched a black-list regular expression. In order to avoid false positives, we manually checked all the repositories discarded in each of the three steps and refined the black-list regular expression until no relevant repository was discarded. In step 8 we locally cloned the 952 repositories, and in step 9 we discarded all repositories not having at least one ROS launch file, either in XML (ROS 1) or Python (ROS 2). The rationale for Step 9 is (i) we are looking for repositories containing ROS-based systems, rather than simple isolated ROS nodes, (ii) launch files are predominantly used for system-related activities, e.g., configuring and starting multiple nodes, and (iii) launch files are the main input for existing architecture extraction approaches for ROS [32, 37].

Finally, in step 10 we performed an in-depth quality assessment of all the 598 repositories. Guided by the systematic literature review methodology [38], we manually analyzed each potentially relevant repository and selected it according to a set of inclusion and exclusion criteria. Two examples of representative exclusion criteria are (the full set is available in the replication package): (1) The repository contains only collections of snippets of code, examples, or templates, and (2) The repository contains only testing artifacts (e.g., test cases). A repository was selected if it satisfied all inclusion criteria and none of the exclusion criteria. Three researchers were involved in step 10 and conflicts were resolved by a fourth one.

The final dataset is composed of 335 GitHub repositories. As shown in Table 1, the dataset is quite heterogeneous, e.g., in terms of commits, contributors, and launch files. This, in addition to our manual quality assessment, makes us reasonably confident that the repositories considered in this study are of good quality and adequately representative of real-world projects.

#### 3.1.2 Data Extraction and Analysis

For RQ1.1, RQ1.2, and RQ1.3 we inspected the following data sources of each of the 335 repositories: (i) all the markdown files stored in the repository (e.g., readme and change log file of the project), (ii) all external web resources and documents linked in the repository, and (iii) all documents stored in the repository. For each data source we conducted iterative content

3.2.1 Questionnaire design and invitation. We designed the questionnaire to complement the results obtained from Phase 1. In the questionnaire we target exclusively RQ2 and RQ3. In this phase we followed well-established guidelines for questionnaire design [33].

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

4 RESULTS
4.1 Architecture-relevant Characteristics (RQ1)
Types of Implemented Systems (RQ1.1). Figure 5a 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 robotic arm) (15.6%), Aerial (11.3%), Service (e.g., mobile robot assistant) (10.1%), Humanoid (5.1%), Aquatic (3.0%), Underwater (3.0%), Self-Driving Vehicle (2.7%), and Other (a specialized software stack and a research kit) (0.6%).

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

Figure 5b shows the distribution of capabilities provided by the subsystems (some subsystems implement more than one capability). Control (40.2%), Planning (31.1%), Vision (26.2%), and Navigation (24.6%) are the largest categories, which is not surprising given that these are common capabilities in robotics systems. Categories that follow are Integration (i.e., integration with external systems such as the cloud, a web page, and apps) (18.0%), Infrastructure (e.g., self-healing, monitoring and logging) (15.6%), Sensing (13.1%), Base (e.g., component startup, basic configurations, hardware-specific nodes) (12.3%), Development Support (e.g., exporters, visualizers, GUIs) (11.5%), Simultaneous Localization and Mapping (SLAM) (11.5%), Mapping (4.9%), Localization (4.1%), and Audio (1.6%).
Architecture Documentation (RQ1.3). We examined the repositories to find architecture documentation, as explained in Section 3.1.2. Most projects (65.7%) do not have architecture documentation. Some projects have partial documentation (17.9%), meaning that the architecture was either informally described (e.g., via a simplified box-and-line or layered diagram) or presented as a list of topics, services, or nodes, but not how all of them were connected (i.e., no configuration information). Finally, only (16.4%) of the projects have a documented architecture (i.e., the system is described in terms of nodes, services, topics, and their configuration).

4.2 Quality Requirements (RQ2)

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

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

4.3 Guidelines (RQ3)

The guidelines and architectural concerns are shown in Table 2. Within each architectural concern, guidelines are sorted by their level of usefulness (“U”, column 5 in Table 2), defined as the number of references in the online survey where a guideline was considered either as absolutely useful or useful. As shown in Figure 6, the vast majority of survey participants assessed the guidelines as (absolutely) useful.
and tested packages such as MoveIt\(^8\) and ros\_control\(^9\).

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

Some steps towards a precise definition of the lifecycle of ROS nodes are being taken by different players of the ROS ecosystem. First, ROS 2 is providing support for managed nodes\(^10\) i.e., nodes whose lifecycle follows a known state machine containing four states, namely Unconfigured, Inactive, Active, and Finalized. The state machine of a ROS 2 managed node can be inspected and controlled by other nodes and launch files. Second, some ROS packages are already explicitly considering the lifecycle of their nodes in their APIs, even without ROS 2 managed nodes; for example, quoting the documentation of ros\_controls/\_ros\_control, "the lifecycle of controllers is not static. It can be queried and modified at run-time through standard ROS services provided by the controller\_manager. Such services allow to start, stop and configure controllers at run-time." This example also highlights that defining and enforcing the lifecycle of ROS nodes can enhance the system in terms of run-time configurability and reflection, which can be exploited for providing autonomous capabilities [2].

**I1 - Assign meaningful names to components (e.g., nodes, topics, services) and group them by adopting standard prefixes/suffixes.** Similarly to how bad variable names impact readability, maintenance, and understandability of source code [8,19], bad practices for naming ROS entities can also negatively impact the architecture of ROS-based systems. This problem is especially severe in ROS-based systems because in ROS, topics and services are created programmatically by the nodes at run-time and their identifiers are just strings. In addition to the obvious issue related to understandability, bad naming practices can lead to run-time errors which are hard to detect, such as mistakenly unconnected nodes or unintentional and unwanted connections, incompatible interfaces, node misconfigurations, and bugs in launch files [37].

**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 [32]. Such complexity can lead to technical debt, lock-in to specific ROS packages, and difficulty extending the system.

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 the system will need to evolve. As an example, the architecture of the ROS stack for the Niryo One\(^7\) manipulator arm is designed around five layers: external communication, command, motion planning, control, and hardware. 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

\(^{A}\) Bag in ROS is the file format used to record messages exchanged by ROS nodes and replay them for future inspection of the system.

\(^{B}\) https://moveit.ros.org/

\(^{C}\) https://moveit.ros.org/

\(^{D}\) https://design_ros2.org/articles/node_lifecycle

\(^{E}\) https://github.com/NiryoRobotics/niryo_one_ros

---

\(\text{\textcopyright} 2023\)
executed when using real robots.

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

*e.g., full resolution video) if only part of it will be used.

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Communication and networking (C)</th>
<th>Quality requirements</th>
<th>P</th>
<th>U</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>Use standardized ROS message formats as much as possible, possibly supporting also their legacy versions.</td>
<td>MAINT, PORT</td>
<td>G(6), (Q)1</td>
<td>71</td>
<td>9</td>
</tr>
<tr>
<td>C2</td>
<td>ROS nodes should be agnostic of the underlying communication mechanisms (e.g., network protocols, deployment topology, etc.), e.g., use AMQP or MQTT.</td>
<td>MAINT, PORT</td>
<td>G(3), (Q)3</td>
<td>63</td>
<td>3</td>
</tr>
<tr>
<td>C3</td>
<td>Include health information about both nodes and data in messages containing critical data, e.g., strength of GPS signal.</td>
<td>REL</td>
<td>G(8)</td>
<td>61</td>
<td>8</td>
</tr>
<tr>
<td>C4</td>
<td>If the system is remotely distributed, constantly observe the status of the communication channels, hosts, and machines on the network.</td>
<td>REL</td>
<td>G(4)</td>
<td>57</td>
<td>4</td>
</tr>
<tr>
<td>C5</td>
<td>Nodes that potentially produce/consume large amounts of messages should be configurable in terms of their publish/subscribe rates.</td>
<td>PERF, MAINT, REL</td>
<td>G(2)</td>
<td>55</td>
<td>2</td>
</tr>
<tr>
<td>C6</td>
<td>Use core rates to limit the data exchanged between nodes to provide only the information that is strictly necessary for completing tasks.</td>
<td>PERF, SEC</td>
<td>G(3)</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>C7</td>
<td>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.</td>
<td>REL</td>
<td>G(2)</td>
<td>53</td>
<td>2</td>
</tr>
<tr>
<td>C8</td>
<td>Developer components when data exchanged between nodes is not compatible (semantically), incorrect, out-of-order, or redundant.</td>
<td>MAINT, PORT</td>
<td>G(8)</td>
<td>42</td>
<td>9</td>
</tr>
<tr>
<td>C9</td>
<td>When starting up robots (instead of publishing to topics) so that the status of the system can be checked before operation.</td>
<td>REL</td>
<td>G(1)</td>
<td>37</td>
<td>1</td>
</tr>
<tr>
<td>C10</td>
<td>Publish empty messages when triggering actions.</td>
<td>PERF</td>
<td>G(1)</td>
<td>25</td>
<td>1</td>
</tr>
<tr>
<td>C11</td>
<td>Frequent messages should be exchanged either via services with persistent connections or via topic-based communication.</td>
<td>PERF, SEC</td>
<td>G(2)</td>
<td>54</td>
<td>3</td>
</tr>
<tr>
<td>C12</td>
<td>Use a dedicated node to store and represent globally-relevant data.</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C13</td>
<td>Do not subscribe to (or unsubscribe from) topics where no messages will be published (even temporarily).</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C14</td>
<td>Nodes should unsubscribe from topics when messages are not needed (even temporarily).</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>C15</td>
<td>Publish messages (and perform the computation necessary to generate them) to a topic only when there is at least one subscriber to it.</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Node re-organizations within the system (N)

<table>
<thead>
<tr>
<th>ID</th>
<th>Group nodes and interfaces into cohesive sets, each with its own responsibilities and well-defined dependencies.</th>
<th>MAINT, PORT</th>
<th>G(6), (Q)1</th>
<th>63</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>N1</td>
<td>Each ROS package should be responsible for one and only one feature of the system or robot capability and provide a well-defined interface.</td>
<td>MAINT, PORT</td>
<td>G(9)</td>
<td>63</td>
<td>9</td>
</tr>
<tr>
<td>N2</td>
<td>Decouple nodes with responsibilities that naturally work at different rates and use different rates for different purposes.</td>
<td>PERF, REL</td>
<td>G(7)</td>
<td>61</td>
<td>7</td>
</tr>
<tr>
<td>N3</td>
<td>By design, limit unnecessary computationally-heavy operations by carefully analyzing the execution scenarios across ROS nodes.</td>
<td>PERF, EN</td>
<td>G(6), (Q)1</td>
<td>59</td>
<td>6</td>
</tr>
<tr>
<td>N4</td>
<td>Transform data only when it is used, for efficiency in terms of computation and bandwidth.</td>
<td>PERF, MAINT</td>
<td>G(1)</td>
<td>58</td>
<td>1</td>
</tr>
<tr>
<td>N5</td>
<td>Do not subscribe to (or unsubscribe from) topics where no messages will be published (even temporarily).</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>N6</td>
<td>Publish messages (and perform the computation necessary to generate them) to a topic only when there is at least one subscriber to it.</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Interface behavior of the nodes (I)

<table>
<thead>
<tr>
<th>ID</th>
<th>The behavior of each node should follow a well-defined lifecycle, which should be queryable and updatable at runtime.</th>
<th>REL, MAINT, PORT</th>
<th>G(4)</th>
<th>50</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>The spinning rate for nodes should be configurable so that they can operate according to available computational resources.</td>
<td>PERF</td>
<td>G(4)</td>
<td>47</td>
<td>4</td>
</tr>
<tr>
<td>B2</td>
<td>If a node is stateful and its behavior strongly depends on time and message arrival order, specify the message protocol expected by the node.</td>
<td>REL, MAINT, USAB</td>
<td>G(2)</td>
<td>45</td>
<td>2</td>
</tr>
<tr>
<td>B3</td>
<td>ROS nodes should be stateless and their behavior should not depend on previous operations or received messages.</td>
<td>REL, MAINT, USAB</td>
<td>G(1)</td>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>B4</td>
<td>Nodes with configuration errors should fail explicitly at launch time.</td>
<td>REL</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>B5</td>
<td>If a node is computationally expensive, then ensure that it only executes when other nodes are subscribed to a topic that it publishes (unless topic latching is enabled).</td>
<td>PERF</td>
<td>Q(1)</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Interaction with hardware and other lower-level entities (H)

<table>
<thead>
<tr>
<th>ID</th>
<th>Use standardized naming to architectural elements (e.g., nodes, topics, messages, services) and group them by adopting standard prefixes/suffixes.</th>
<th>MAINT, USAB</th>
<th>G(6), (Q)2</th>
<th>71</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>H1</td>
<td>When possible, core algorithms, libraries, and other generic software components should be ROS-agnostic.</td>
<td>PORT, MAINT</td>
<td>G(1), (Q)3</td>
<td>68</td>
<td>1</td>
</tr>
<tr>
<td>H2</td>
<td>Design ROS nodes to be as hardware-independent as possible.</td>
<td>USAB, COMP</td>
<td>G(4)</td>
<td>60</td>
<td>4</td>
</tr>
<tr>
<td>H3</td>
<td>Decouple ROS nodes from variations in the execution environment.</td>
<td>REL, MAINT, USAB</td>
<td>G(1)</td>
<td>52</td>
<td>3</td>
</tr>
<tr>
<td>H4</td>
<td>The interface of nodes responsible for state estimation should (i) support an arbitrary number and different types of sensors and (ii) be able to combine the information provided by the sensors.</td>
<td>REL, MAINT</td>
<td>G(1)</td>
<td>50</td>
<td>1</td>
</tr>
<tr>
<td>H5</td>
<td>If context-specific configuration is needed at run-time (e.g., available hardware capabilities), then persist this configuration in a dedicated node to avoid having to recalculate it at run-time.</td>
<td>PERF</td>
<td>G(4)</td>
<td>32</td>
<td>4</td>
</tr>
</tbody>
</table>

Safety-critical concerns (S)

<table>
<thead>
<tr>
<th>ID</th>
<th>ROS nodes should be resilient with respect to the amount and frequency of the data received by sensors.</th>
<th>REL</th>
<th>G(6)</th>
<th>59</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Use core rates to limit the data exchanged between nodes depending on the criticality and real-time needs of the nodes.</td>
<td>REL</td>
<td>G(1), (Q)3</td>
<td>49</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>For real-time requirements, collect timestamps from as many sources as possible (i.e., do not rely on ROS-based timestamps only) 5.</td>
<td>PERF</td>
<td>G(2)</td>
<td>36</td>
<td>2</td>
</tr>
<tr>
<td>S3</td>
<td>Provide a dedicated node for managing temporary and short-term data (e.g., in the order of seconds).</td>
<td>REL, SAFE</td>
<td>G(3)</td>
<td>34</td>
<td>5</td>
</tr>
</tbody>
</table>

Resilience (P)

<table>
<thead>
<tr>
<th>ID</th>
<th>Avoid persisting raw data (e.g., a full resolution video) if only part of it will be used.</th>
<th>PERF</th>
<th>G(1)</th>
<th>56</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Avoid race conditions when persisting data received from other ROS nodes within the system.</td>
<td>PERF, MAINT</td>
<td>G(1)</td>
<td>52</td>
<td>1</td>
</tr>
<tr>
<td>P2</td>
<td>Use a dedicated node for persisting and querying long-term data and short-term data (e.g., in the order of seconds).</td>
<td>PERF, MAINT</td>
<td>G(2)</td>
<td>32</td>
<td>2</td>
</tr>
</tbody>
</table>

*ROS 2 supports C12 via intra-process communication. ROS 1 supports it via nodelets.

5ROS 2 partially supports guideline B1 via managed nodes.

A recurrent strategy to achieve H1 is to have a hardware abstraction layer. For example, in ros-controls/ros_control "the backbone of the framework is the Hardware Abstraction Layer, which serves as a bridge to different simulated and real robots … It also allows for integrating heterogeneous hardware or swapping out components transparently … Through these typed interfaces, this abstraction enables easy introspection, increased maintainability and controllers to be hardware-agnostic".
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 [22]. 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 hasauinao/rrt_exploration document their own solution for continuous estimation as follows: “each state estimation node in robot_localization begins estimating the vehicle’s state as soon as it receives a single measurement. If there is a holiday in the sensor data … the filter will continue to estimate the robot’s state via an internal motion model”.

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

**P1 – Avoid persisting raw data (e.g., a full resolution video) if only part of it will be used.** Depending on the provided capabilities, ROS systems can produce large amounts of data. For example, the ROS bag of a demo mapping session of the rtabmap_ros package can take up to 1.1Gb11. As informally confirmed by the ROS community12, data persistence in ROS can lead to severe performance overheads at run-time.

Even though it can be tempting to persist all raw data produced by the system (e.g., for subsequent inspection or replay), developers should identify the subset of relevant data 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 florisvb/multi_tracker, where a “buffering node” listens to the camera topic … and saves the pixels and values for any pixels that change more than the specified threshold. This results in a dramatically compressed filesize relative to a full resolution video.”

## 5 DISCUSSION

In this study we mined ROS open source software repositories and surveyed active developers to elicit quality requirements and guidelines that impact the software architecture of robotics systems. We found discussions relating to ten different quality requirements. The top three quality requirements were maintainability, performance, and reliability. While we cannot say for certain, we suspect that maintainability is important because the aim of the ROS community (and open source software in general) is to build software that can be used in many contexts. This is further supported by the observation that many survey participants highlighted the importance of documentation and ease of source code understanding of the packages. Performance is important because many of the computations 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 safety and security – in fact, security is a major factor in spawning variants of ROS, such as SROS and ROS-M.12 We suspect that these qualities will gain importance as robots become more common and guidelines emerge for them. We also suspect that safety and security are currently of more concern in commercial robotics, an area that was outside the scope of this study. We were also surprised that energy was not a major concern, especially when robots are autonomous and have restricted energy supplies.

Because the guidelines have been sourced from an examination of open source projects and surveys to participants of those projects, we do not make any claims that we have elicited all guidelines associated with developing ROS-based systems. Indeed, there may be other sources for guidelines too, such as robotics textbooks, interviews with robotic experts working on commercial products, etc. Moreover, as shown in Table 2, out of the 49 guidelines, 32 came uniquely from open-source repositories, 10 came uniquely from surveys, and 7 came from both sources. Therefore, we can say that GitHub is a good source for guidelines. Interestingly, repositories on GitLab and BitBucket were filtered out in Phase 1 of the study - after the first nine steps we had 0/46 GitLab repositories included and 23/527 BitBucket repositories. These final ones were discarded after manual selection as they did not satisfy the selection criteria.

We believe that these guidelines are generally applicable, but they should be used in context, e.g., C2 and C4 appear to be 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, contradicting C2. An architect should know 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.

Furthermore, while this paper reports only on guidelines related to ROS, we think that many of the guidelines will apply to other non-ROS robotic systems. Most robotic systems are implemented in some form of publish-subscribe style (e.g., [6, 11]), and many of the guidelines (e.g., C3-C15) may also apply to those systems.

## 6 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 whole roboticists population. This potential bias is reasonably avoided because participants exhibited a good level of heterogeneity in terms of type of experience, number of contributed ROS packages, and primary motivation for using ROS.

**Internal validity.** This study has been conducted by adhering to well-established guidelines in software engineering [23, 33, 38]. 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. Moreover, the 39 guidelines emerging from Phase 1 have been scrutinized by 77 independent roboticists and the additional 10 emerging from Phase 2 were directly proposed by roboticists working on real projects. In order to be confident about the quality of our guidelines, we collected feedback about the final set of 49 guidelines from two volunteers working on robotic systems for several years.

**Conclusion validity.** Third-party researchers can verify and check the obtained results by replicating our study independently 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 systems’ 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 that the mining pipeline for searching and filtering ROS-based projects from the code hosting platforms is implemented and configured correctly. We mitigated this potential threat to validity by carefully designing the whole pipeline (see Section 3.1.1), by testing each component of the pipeline in isolation via subsets of data for which we knew already the expected outcomes, and by making the implementation of the whole pipeline publicly available in the replication package.

7 RELATED WORK

**Studies on ROS.** Because ROS is under active development, is a de facto standard for development of robots, and comprises many open-source components, software engineering researchers are increasingly using it as a source for research. A number of researchers have tools and analyses that extract architectural structures from ROS systems through static analysis of source files and configurations. For example, HAROS [32] and other work described in [29, 37] can generate the architecture and perform architectural consistency checks such as detecting communication errors (e.g., nodes subscribing to unpublished topics). This body of research focuses on reconstructing architectures of existing systems rather than trying to understand what architectural patterns or guidelines could be used to achieve particular quality requirements.

To design good quality robotic software in ROS, the work in [17, 20] focuses on formally modeling ROS systems for real-time analysis and V&V. These do not really focus on the architecture of ROS systems, but could be used to generate or check the correctness of ROS systems. In terms of code quality, Pichler et al. examine open source projects with tools such as cpplint and XML validation to analyze the quality of packages that many other ROS systems depend on [28]. 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 4), which supports this observation. Similarly, [13] provides a visualization tool with run-time usage metrics that can be used after-the-fact to assess performance.

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

**Mining Architectural Information.** Mining information related to software architecture from open source software repositories is not new. Other researchers have shown that such information can be derived from commit logs (either manually [35] or by applying machine learning [5]), issue trackers [24], as well as developer discussions in community groups such as StackOverflow [34] 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 [27]. In fact, these can be the areas of code that change the most [26]. In future work we will use these techniques to extend our work to discover how existing tactics are used in robots and if there are any new domain-specific tactics that can also be used in other software domains.

8 CONCLUSIONS AND FUTURE WORK

This paper described a study to elicit evidence-based architectural guidelines for open-source ROS-based software for robots. Among other findings, we found that of the relevant repositories, 16.4% of them documented their architecture. In these cases, the most discussed qualities were maintenance, performance, and reliability. We elicited 39 guidelines derived from the repositories, and 10 more from the survey. By surveying roboticists actively involved in these projects, we are confident that these guidelines are generally useful. These results can be used by roboticists to architect their systems to
achieve particular quality requirements. Furthermore, architecture researchers can use this study as a baseline for understanding the architectural principles and practices of robotists. These results are a major foundation on which to base our future work on determining architecture tactics for high-quality robot software, and thereby provide a solid engineering basis for developing robots in a future where they are ubiquitous. To achieve this, we plan to (i) explore how existing architecture tactics found in [4] and others could be applied in this domain, (ii) mine robot-specific tactics from other robotics sources and semi-automatically from source code using approaches based on [32, 37], and (iii) provide guidance for architecting high-quality robot software.

ACKNOWLEDGMENTS

This research is partially supported by the Dutch Research Council (NWO) through the OCENW.XS2.038 grant. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center (DM19-0986), and on research sponsored by AFRL and DARPA under agreement nr. FA8750-16-0042. 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. Thanks to Jarrett Holtz and Selva Samuel for their comments on parts of the paper.

REFERENCES