Empirical Software Engineering (2024) 29:57
https://doi.org/10.1007/s10664-024-10440-0

ROBUST: 221 bugs in the Robot Operating System

Christopher S. Timperley[1] · Gijs van der Hoorn[2] · André Santos[3] · Harshavardhan Deshpande[4] · Andrzej Wąsowski[5]

Accepted: 3 January 2024

Abstract
As robotic systems such as autonomous cars and delivery drones assume greater roles and responsibilities within society, the likelihood and impact of catastrophic software failure within those systems is increased. To aid researchers in the development of new methods to measure and assure the safety and quality of robotics software, we systematically curated a dataset of 221 bugs across 7 popular and diverse software systems implemented via the Robot Operating System (ROS). We produce historically accurate recreations of each of the 221 defective software versions in the form of Docker images, and use a grounded theory approach to examine and categorize their corresponding faults, failures, and fixes. Finally, we reflect on the implications of our findings and outline future research directions for the community.

Keywords Robotics · Software bugs · Dataset · Robot Operating System · ROS · BugZoo

Communicated by: Mehrdad Sabetzadeh

✉ Christopher S. Timperley
ctimperley@cmu.edu

Gijs van der Hoorn
g.a.vanderhoorn@tudelft.nl

André Santos
andre.santos@vortex-colab.com

Harshavardhan Deshpande
harshavardhan.deshpande@ipa.fraunhofer.de

Andrzej Wąsowski
wasowski@itu.dk

1 School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA

2 Department of Cognitive Robotics, faculty of Mechanical, Maritime and Materials Engineering, Delft University of Technology, Delft, The Netherlands

3 VORTEX CoLab, Vila Nova de Gaia, Portugal

4 Fraunhofer Institute for Manufacturing Engineering and Automation IPA, Stuttgart, Germany

5 Department of Computer Science, IT University of Copenhagen, Copenhagen, Denmark

Published online: 23 March 2024

1 Introduction

From assembling and manufacturing goods to driving us from place to place, robotic systems constitute an increasingly large part of the computing ecosystem. However, these systems, and the software that controls them, present new opportunities for cyberattacks and catastrophic failures with the potential for enormous economic and human damage (O'Kane 2020; Wall 2017; Charette 2014; McCausland 2019). To fully realize the benefits of robotic systems, we need effective quality assurance (QA) techniques for robotics software that allow developers to build advanced applications without compromise to safety. In order to catalyze the development of these QA techniques for robotics, it is important that we better understand the nature of bugs within robotics software. By better understanding the nature of software bugs in robotic systems, we can identify key challenges and promising avenues of research. To that end, we have endeavoured to paint a detailed picture of challenges in the largest software ecosystem for robotics, the Robot Operating System (ROS). We take the perspective of bugs, previously documented and, mostly, fixed in the open source repositories of ROS code.

ROS, colloquially known as the "Linux of Robotics," is a highly modular and distributed open-source platform for building robotics systems with a rich ecosystem of thousands of reusable software packages (Wyrobek 2017; Kolak et al. 2020). ROS runs on top of Linux typically. It is widely used in teaching, research and engineering, and attracts major industrial players including Amazon, Intel, and Microsoft: The ROS-Industrial Consortium counts close to a hundred organizations including Bosch, Siemens, and Boeing (ROSIn 2022). There is a growing belief that a shared open source platform will allow the industry to exploit the economy of scope in robotics. High-quality hardware drivers, control modules, AI components, and development tools shall benefit the entire ecosystem while the cost of building them is carried by multiple organizations. Such benefits shall also be extended to testing and quality assurance tools and methods, which are of paramount importance for professional development of software in industry. This paper, devoted to the software quality challenges in the ROS ecosystem, seeks to identify opportunities for research and development that will benefit software development for robotics in general and for ROS in particular.

We report the results of a collaboration between academic and industrial partners to document, reproduce, and understand software bugs that occur in ROS software. Our method is artifact-driven: we create a data set of ROS bugs and then study it. Thus the main outcome of the work is the artifact. We systematically identified 221 bugs across seven popular and diverse ROS subject systems, representing a variety of domains and layers of the ROS application stack, by studying their respective version control histories and artifacts. We methodically studied each bug to produce a structured forensic description of its causes, symptoms, and fix, amongst other details. We then used a grounded theory to categorize and understand the nature of faults and failures within ROS systems. To bolster research into effective quality assurance techniques, we package each bug into a historically accurate Docker container image, allowing it to be accurately studied by others for a variety of purposes (e.g., program repair, testing, debugging).

Our analysis of the ROBUST dataset demonstrates the inherent difficulties of building high-quality reusable robotics software components and the unintended consequences of framework design tradeoffs, and identifies opportunities for the development of new languages and analysis tools to mitigate these difficulties. We find that ROS developers make similar kinds of mistakes as other developers, and that certain bugs within the dataset could have been detected by existing tools and practices (e.g., type checking, fuzzing, continuous

integration). In line with previous research (Afzal et al. 2020), we find that very few bugs are accompanied by regression tests, testing is generally lacking across each of the studied systems, and, consequently, developers tend to rely on manual testing efforts. Finally, we observe that, while failures typically span across multiple components, bug fixes are comparatively simple and confined to a small number of lines within a single file.

The main contributions of this paper include:

- A dataset of ROS bugs, ROBUST, containing 221 bugs across seven popular ROS systems. ROBUST is open source and free to use, and can be found at: https://github.io/robust-rosin/robust,
- An analysis of the faults and failures of bugs represented within the ROBUST dataset (Sections 3 and 4),
- A discussion of implications for practitioners and the research community (Section 5),
- A method and a toolchain for building a bug repository and accurately replicating bugs in ROS systems and similar ecosystems (Section 6.)

Section 2 provides a basic characterization of ROS as a subject of study and discusses the selection of ROS systems to harvest the bugs from.

Our intended audience are software engineering researchers that build new tools and methods for robotics software developers. The `ROBUST` dataset aims to lower the barrier of entry into research in software engineering for robotics, supporting, among others, further work on testing, fuzzing, architectural analysis techniques, verification, program repair, etc. For example, a *fuzzing tool* for robotics would have to work with multiple executables, multiple input streams, multiple programming languages, and network communication. `ROBUST` bugs that manifest themselves in crashes can be used to test effectiveness of such a fuzzer as each of them can be easily re-established in a Docker container. Building a cross-language *program-repair tool* for Python and C++ requires understanding any cross-language bugs in such systems and of the API binding mechanisms in these languages. `ROBUST` can save a lot of time in this process, as it contains more than twenty bugs on the Python/C++ boundary that are ready to be reproduced and used in the tool design, evaluation, or regression testing. `ROBUST` can also support building tools for *domain-specific languages* as it contains more than a hundred documented bugs that have been repaired by developers in files written in domain specific languages. These are just three examples of applications of `ROBUST` as a research tool. We genuinely hope that it can also help in work on build systems, software configuration, concurrency, and evolution.

2 Subject Ecosystem and Subject Systems

The Subject Platform: Robot Operating System The Robot Operating System originates from Stanford University (Quigley et al. 2009). In 2007, the project had transferred to a robotics start-up, Willow Garage, which in turn founded the Open Source Robotics Foundation, a non-profit dedicated to promoting open source robotics and the present steward of the community. ROS started as a communication middleware allowing developers to create robot systems using established architectures for distributed systems. It has quickly expanded with numerous tools, hardware drivers, and software modules for robotics-specific tasks such as planning, navigation, and perception. Today, the core modules are still developed by a tight group of developers, however, the majority of components in the vast ecosystem are maintained by a large community of companies and research groups in a highly decentralized

manner. Due to its size and proliferation, ROS is presently the most representative robot software development framework.

ROS software is organized in *packages*, the basic build and release units. Packages can be added to an official index and released in *distributions*, which are updated following a well-defined life cycle pattern similar to that of Ubuntu Linux. For instance, ROS Indigo Igloo, released in May 2014, was a *Long Term Support* (LTS) distribution, targeting Linux Ubuntu Saucy (13.10) and Linux Ubuntu Trusty (14.04 LTS). Support for Indigo ended in April 2019, at the same time that Ubuntu Trusty reached its end-of-life. Current distributions have already surpassed the mark of 4,000 packages. Besides these, there are many community packages that are not part of the official distributions.

A ROS-based system follows a distributed architecture, with independent runtime resources connecting to each other. Every resource is named, using a hierarchical naming structure. Names can be *remapped* during system initialization without changing source code. This mechanisms allows developers to compose systems from different packages without modifying them.

ROS supports four types of resources: *parameters*, *nodes*, *topics*, and *services*. Parameters are variables holding shared data, stored in a central Parameter Server. Nodes are programs that consume and produce data, and communicate with each other via message passing channels—asynchronous topics and synchronous services. Topics and services are both typed. All nodes are expected to obey types handled by services and topics, which is enforced at runtime. Users can define custom messages adding to the available predefined types representing primitive data, laser scans, 3D poses, etc.

Topics are the most common message-passing mechanism. They follow an asynchronous publisher-subscriber model, with many-to-many connections. Publishers can send messages at any time, regardless of the number of active subscribers, and subscribers are notified via a callback function whenever a new message arrives. Services implement synchronous one-to-one communication, using remote procedure calls. The node that provides a service is called the server, and any nodes using the service are called clients. Messages are exchanged in

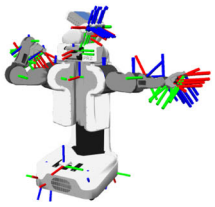
(a) TurtleBot with a Kobuki base

(b) Care-O-bot 4

(c) Universal Robot

(d) Motoman

(e) MavRos*



(f) Robot model with `tf2` frames

Fig. 1 Subject systems, not to scale. *Images: DinosoftLabs, photo3idea_studio

request-response transactions. Clients block while waiting for a response. Services shall be used for fast tasks, such as querying the current state of a node.

Nodes should be specific and modular, rather than large monolithic components. A single robot consists of many nodes, each accomplishing a task, such as localization, navigation, or perception. Each sensor and actuator might have its individual node, too. Nodes are implemented using a ROS client library. Many programming languages are supported but C++ and Python are most used.

Subject Systems Conducting a meaningful analysis of software bugs requires considerable domain expertise. Therefore, we deliberately set out to examine a diverse set of subject systems for which we had expertise in our research team. In total, we examined seven qualitatively different subject systems consisting of three robotics systems, two robot drivers, and

Table 1 Descriptions of subject systems

TurtleBot is an iconic ROS robot (Fig. 1a), a low-cost entry-level personal robot kit with open-source software built with the original authors of ROS. The software includes example applications for automatic docking, charging, navigation, and dynamic leader-follower behaviors, written mostly in Python.	**Kobuki** is the mobile base on top of which TurtleBot built, the black disk at the bottom of the robot in Fig. 1a. The Kobuki software repositories provide low-level packages that integrate the base hardware (servomotors, power systems) with ROS, as well as a safety controller and a velocity multiplexer, mostly written in C++.
MavRos implements a bridge between ROS and the MAVLink protocol for communicating with the autopilot of unmanned vehicles (air, ground, and water). The MAVROS repository provides tools and plugins that allow almost transparent bridging between a MAVLink-enabled autopilot, such as ArduPilot, and a ROS application, making as much of the data gathered and processed by these systems available to the ROS node graph. The repository contains only the bridging nodes (in C++/Python), and explicitly leaves the modelling of vehicle geometry, kinematics, and dynamics to other packages.	**Universal Robot** and **Motoman** provide packages for the integration of industrial robots with ROS (Fig. 1c and d, both robots are industrial manipulators). Packages, implemented mostly in C++, Python, and C, include low-level interfaces to the motion controllers of the robot, sensors and I/O interfaces, as well as higher-level declarative description packages that provide information on robot geometry, kinematics, dynamics, and motion planning configurations. Both repositories also include programs that are to be executed on the industrial robot controller itself, and which will collaborate with their ROS counterparts.
Care-O-bot is an autonomous service robot, created by Fraunhofer IPA and commercialized by Mojin Robotics (Fig. 1b). Nearly all its behavior is implemented as ROS nodes (a significant part open-source) including low-level interfaces to motor controllers and sensors, human-machine interfaces (speech, face, emotion and gesture recognition), collision-free path planning, manipulation planners, object recognition, and localization. Packages for simulation of the entire system are also included, as well descriptions of its geometry, kinematics, and dynamics. Most of the code is written in C++ and Python. A rather large amount of XML is used mostly to define the many configurations enabled off-the-shelf.	**Geometry2**, also known as `tf2`, offers a set of services for keeping track of multiple coordinate frames over time and for transforming points and other geometric objects between frames. The geometry2 libraries are implemented in C++ and interfaced to Python. They can be used to determine the global position of the robot relative to the world or the position of a gripper relative to the robot base. Applications range from localization and visualization to mapping and multi-robot cooperation. Figure 1f shows a 3D robot model annotated with coordinate frames tracked by `tf2` on the robot's base, head, arms and grippers (visualized as three colored axes).

See also Table 2 and Fig. 1

Table 2 Descriptive statistics for the subject systems

Subject	# Bugs	# Issues	Category	C++	C	Python	XML
Kobuki	57	325	application	23,555	18,073	4,207	2,325
TurtleBot	11	170	application	799	42	4,438	1,129
Care-O-bot	11	182	application	31,084	9,430	9,248	23,814
Universal Robot	25	158	driver	1,071	331	1,741	738
Motoman	22	78	driver	4,129	5,337	0	1,272
MavRos	40	623	middleware	12,807	1,611	1,013	330
geometry2	42	264	library	6,267	4,311	1,074	273
Total	**208**	**1800**		**79,712**	**39,135**	**21,721**	**29,881**

The second and third columns show the number of identified bugs and issues that were studied, respectively. The last four columns show the number of lines in main languages, counted with CLOC v1.60

two very different specialized libraries. Figure 1 shows photographs and visualizations of the subject systems, Table 1 summarizes their key descriptive properties, and Table 2 provides a quantitative characterization in the last four columns.

3 Method

Subject Bugs and Data Gathering To identify historical bugs in each subject system, we examined its issue tracker, pull requests, and commit history. For an initial screening, we prioritized issues labeled as a *Bug* (or similar) and commit messages including keywords such as *fix*. Issues clearly unrelated to bugs by their title or labels were discarded. All non-obvious issues and pull requests were inspected in consensus meetings to determine whether they describe additional bugs. In the meetings, we have asked whether the problem discussed is a result of a deliberate prior design decision, or whether it is a result of an omission, a mistake, a change in another system, etc. In any case, whenever developers used the term bug, error, or mistake in the discussion, we assumed a bug is being discussed. Bad smells and style issues were classified as not-bugs. In total, we identified 221 issues and pull requests that qualified as bugs across the subject systems.

Figure 2 exemplifies the data available about the bugs. The issue *creation date* (1) determines the versions of ROS and other dependencies that might have been used by the reporter. The *community status of the reporter* (2) distinguishes between issues found internally and by the downstream users. The *problem description* (3) is the key source regarding whether an issue is a bug and what is its nature. The *labels* (4) provide a diversity of information. Here the issue has been labelled as software-related, which makes it a potential *software* bug report. The existence of *commits* (5, 8) referencing the issue show that it is either fixed or being worked on. Inspecting the *commit* (6, 8) we can understand the bug from the perspective of its fix. The *referencing pull requests* (7) provide similar context. If the bug is fixed, we note down the *closing date* (9) of the issue. This allows to estimate how long it took to resolve the problem (14 days here).

Inspecting commits and pull requests that do not reference any issue requires additional work, as they tend to describe *what* changes are introduced, rather than the problem addressed. Despite this, from the commit in Fig. 3, we can sill harvest four relevant items: The *commit message* (1) includes the keyword *"fix"*, implying there was an underlying issue. In this case,

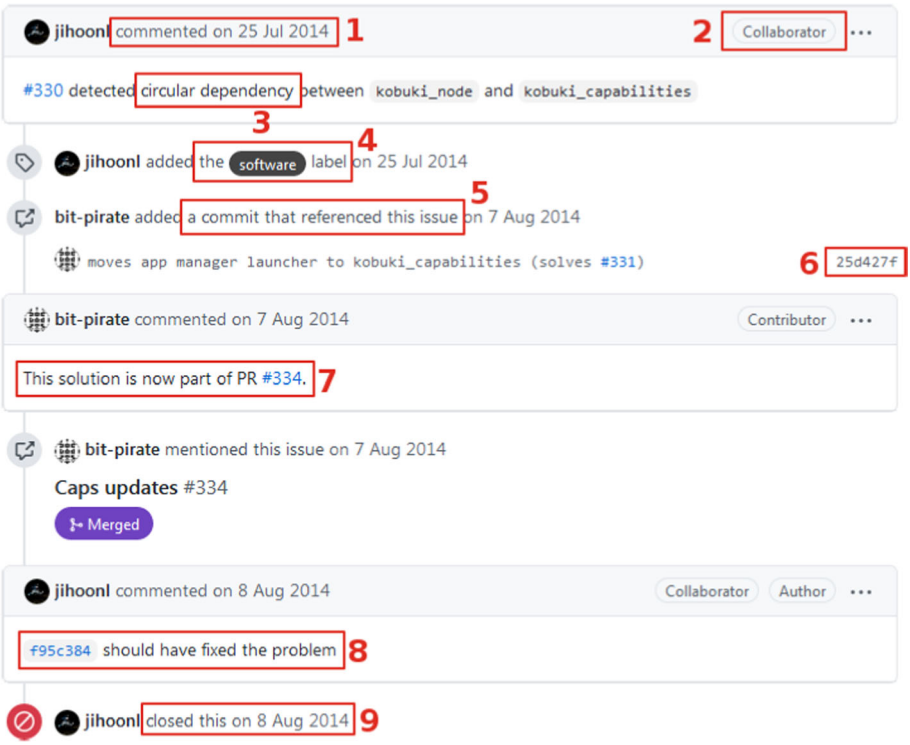


Fig. 2 Issue 331 in Kobuki, https://github.com/yujinrobot/kobuki/issues/331

the commit fixes warnings from the CPPCHECK code analyzer. The first *release* (2) of the repository, which includes the commit. Together with the *commit date* (3) this determines the versions of the involved software. The *parent commit* (4) is the last version of the code that still contains the bug fixed.

Data Analysis For each of the bugs in the data set we produced a *forensic description* by manually analyzing the available information. Each description follows a common schema. The initial list of attributes in the schema has been identified in a discussion of the authors based on their expertise in bug studies and in robotics software engineering. The list has remained stable for most of the data collection period, but several fields have been added in an exploratory fashion. These were usually derived either from the initial fields (using thematic coding) or by automatically querying GitHub repositories. Each description has

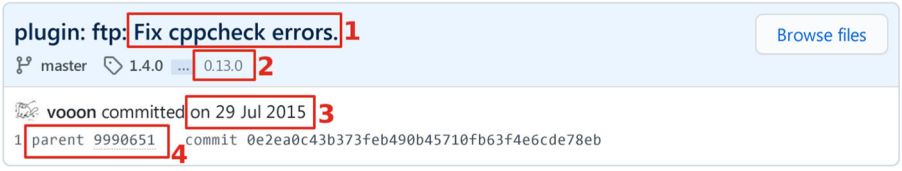


Fig. 3 Commit `0e2ea0c4` of https://github.com/mavlink/mavros

been initially written by a team member familiar with the associated subject system, before being discussed extensively and cross-checked by multiple members of the research team. We include all these descriptions as YAML documents, along with the schema in `ROBUST` repository on GitHub.

Figure 4 shows an example of a description for a bug in the Kobuki project. It opens with a unique identifier, a prefix of the hash of its fixing commit in a Git repository (e.g., `e964bbb`). The `title` summarizes the bug in general terms, and the `description` elaborates on the bug itself, the software components affected, and the context in which the bug occurred. We wrote the descriptions aiming to be as accessible as possible, without presupposing deep training in robotics. The `keywords` aid the search and retrieval of relevant bug reports. (Unlike the codes discussed below, the keywords are not derived systematically.) The `system` field records the name of the project in which the bug has been found.

```yaml
id: e964bbb
title: Calculation Error Inverts Turning Direction
description: "Due to an error in velocity calculations, when Kobuki was issued a
  very low negative linear speed (slow backwards movement), it would also
  inadvertently invert its turning direction. That is, if it was supposed to move
  backwards while turning left, it would move backwards and turn right instead."
CWE:
  id: 242
  description: Incorrect Calculation
keywords: ['differential drive', 'velocity', 'driver', 'movement']
system: kobuki
failure-codes: [ 'PROGRAMMING:CALCULATIONS' ]
fault-codes: [ 'SYSTEM:UNINTENDED-BEHAVIOUR', 'SYSTEM:MOTION' ]
bug:
  phase: runtime
  architectural-location: application-specific code
  task: locomotion
  package: yujinrobot/kobuki/kobuki_driver
  detected-by: developer
  reported-by: contributor
  issue: https://github.com/yujinrobot/kobuki/issues/227
  time-reported: 2013-02-22T09:09:21Z
fix:
  commits:
    - repo: https://github.com/yujinrobot/kobuki
      hash: e964bbb8700fb1a9b95c0cfe5a44d43321294d4f
  pull-request: null
  fix-in: ['kobuki_driver/src/driver/diff_drive.cpp']
  languages: ['C++']
  time: 2013-02-25 (09:31)
```

Fig. 4 An example report for a bug (kobuki:e964bbb)

We initially attempted to classify bugs using *Common Weakness Enumeration*, an established taxonomy of software weaknesses independent of us.[1] However, as CWE is predominantly concerned with security, we were unable to adequately classify most of the dataset. Motivated by this inadequacy, rather than re-using an existing taxonomy (e.g., IEEE 1044–2009; Seaman et al. 2008; Thung et al. 2012; Garcia et al. 2020; Wang et al. 2021; Zampetti et al. 2022), we elected to use open coding and grounded theory building as an established mechanism for structuring qualitative data, when no prior taxonomy is pre-supposed. This allows us to better represent and fully describe the nature of software bugs in ROS without being constrained by an existing categorization. Moreover, by allowing the taxonomy to emerge from the data, our study provides a conceptual replication of prior work.

We systematically analyzed all bugs through a process of thematic coding by establishing codes in two groups: *failure* descriptions and *fault* descriptions. We define failure as inability of software to perform its function, with a special focus on the observable manifestation of this inability (sometimes also referred to as *error*). The *fault* is the cause, or the reason for the failure, within the software (so if the fault is repaired, the failure is eliminated). The results of this analysis are stored under `failure-codes` and `fault-codes` respectively.

The thematic coding has been split among the five coauthors randomly; two coders per each bug description. They performed the initial coding independently, introducing new codes as necessary. After the initial coding has been obtained, we held a consistency meeting which produced a unified codebook. Afterwards all bug descriptions have been recoded according to the codebook. Finally, for all code assignments where the two coders disagreed we held a series of consensus meeting with all five coders—an agreement was achieved by a joint discussion, analysis of the source material, and any necessary context information about ROS. Two of the coders involved had extensive robotics engineering experience, and three had extensive software quality engineering experience.

The rest of the record is broadly split into two sections: the *bug description* that elaborates on the fault and failure, and the *fix description* that collects information on how the bug has been fixed. The bug description specifies: the *stage* at which failure occurs (e.g., build, deployment, runtime); the relationship of the person that *reported* the bug to the affected system (e.g., guest user, contributor, maintainer, automatic, unreported); the URL of the associated GitHub *issue*, and at what time the issue was reported; the *task* of the robot that is directly affected by the bug (e.g., perception, localization, planning); a determination of how the bug was *detected* (e.g., build system, static analysis, assertions, runtime detection, test failure, developer); and whether the failure occurred in the application or in the ROS/ROSIn platform itself (*architectural-location*). The fix description provides: a list of commits that constitute the bug fix; the URL of the associated pull request, the date and time at which the bug was fixed; the files that were changed as part of the fix, and the language of those files. We only include the subset of files that were changed and which relate to the bug fix itself. We do not include coincidental changes (e.g., refactorings).

In Section 4, we give clickable links to bugs in the repository. These lists of links are not exhaustive, in the sense that they show a small number of examples, not all the examples from the dataset in the given category.

Descriptive Statistics of the Obtained Dataset Table 2 lists how many bugs we collected for each of the subject systems, and out of how many issues they have been selected (the remaining issues did not report bugs, so the statistics paint a valid picture of the bug population for this systems at the collection time).

[1] https://cwe.mitre.org

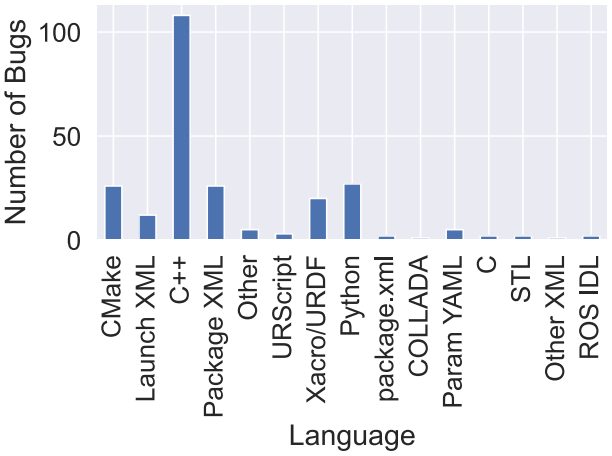


Fig. 5 The languages and file formats involved in bug fixes

Figure 5 breaks down the bugs by languages used in the fixed files, for the bugs that have been fixed. Over half of the bug fixes involve C++ (112 of 219). The remaining 107 fixes use a diversity of languages, many of which are domain-specific (e.g., Package XML, Launch XML, URScript, etc.), which typically lack associated analysis tools. Figure 6 shows the number of languages involved in each fix. We find that 200 fixes (91%) are limited to a single language. That is, while failures may span components written in different languages, fixes are usually restricted to a single language.

In the dataset, 118 failures occur at run-time and 38 at start-up time, so in total 156 bugs that have been fixed also have execution-time failures. Only 15 of these are accompanied by a test case. (We automatically identified bug fixes that add or modify tests by checking the paths of the changed files. We consider that a fix commit is accompanied by a test case if it adds or changes a file that contains the word `test` in its path, e.g., `test/foo.py`,

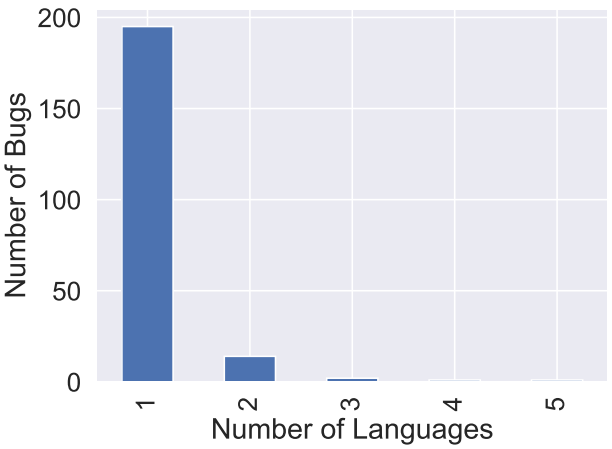


Fig. 6 The number of languages involved in bug fixes

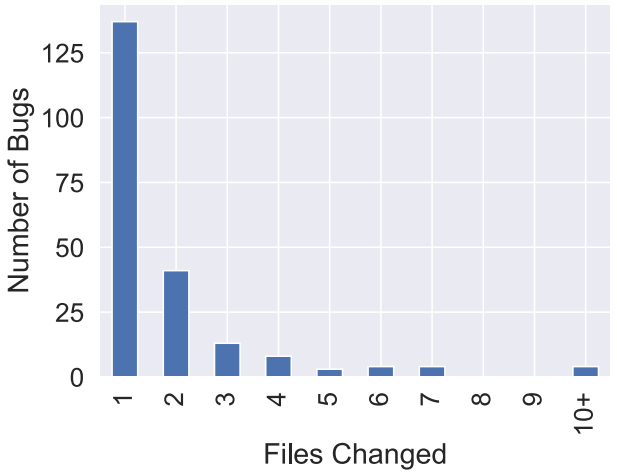


Fig. 7 The number of files involved in a bug fix

`test_foo.py`. We manually inspected the remaining fixes to confirm that they did not include a test case.)

Figure 7 shows an overview of the number of files that were fixed for each bug. The number is based on a manual removal of unrelated changes from bug-fixing commits. Almost two thirds of bug fixes affect only a single file. Specifically, 64% of bug fixes are confined to a single file (141 of 219), 19% span two files (41 of 219), and 17% (37 of 219) change three or more files.

In order to approximate the size of each bug fix, we measure the number of lines in the change differences across their fixing commits; see Fig. 8. As fixing commits may contain unrelated changes (e.g., opportunistic refactoring), the size of the change difference is greater than or equal to the size of the bug fix, and therefore represents a conservative upper bound. Four bug fixes consist solely of file renamings and have an associated diff size of zero. We

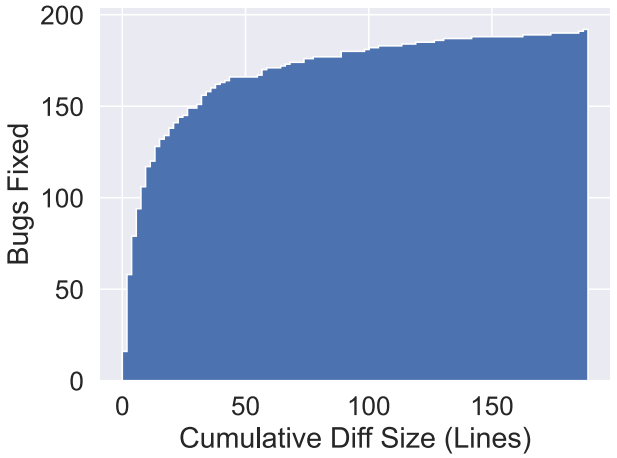


Fig. 8 The diff size of a bugfix; we truncated 11 bugs with size above 200 lines

find that more than 50% of bug fixes have diffs that consist of 12 lines or fewer, and 75% have a change difference that is 50 lines or smaller.

Limitations The *Credibility* (Shenton 2004; Sikolia et al. 2013) of this study has been ensured by careful selection of the systems to be analyzed (Section 2), by depending on qualified researchers for bug selection and analysis, by the use of established methods for both archival and coding of bugs (Section 3), by employing peer scrutiny, and by grounding the study in existing work (Section 7). All authors were involved in bug gathering, selection, and analysis. One author is a domain expert on ROS with considerable experience in this FOSS community, the others have prior experience with ROS and with the kind of studies presented in this paper. All bugs were analyzed by at least two authors and the results were cross-referenced and discussed among all. Corrections were made by consensus. The coding was done according to established practices (Saldaña 2015; Linneberg and Korsgaard 2019) and by all authors in parallel, in multiple sessions. The final classification of bugs and the resulting codebook have been extensively discussed and checked for consistency by all authors. Finally, the preliminary results of the study were presented at ROSCon, the main conference of the ROS community (Timperley and Wasowski 2019). Received comments were taken into account while further refining the study.

We used purposive sampling, so some negative impact on the *Transferability* is expected. The subject repositories were selected based on the role the packages have in ROS-based products. Care was taken to select a qualitatively diverse set. We do not claim that this set of packages is representative of the entire ROS landscape, or of the wider robotics software. The quantitative results are not directly generalizable to these wider contexts—they have a been presented as descriptive statistics of the dataset, not as general conclusions. Even with these restrictions however, the set of bugs described in `ROBUST` is diverse enough to be representative of the types of systems that were analyzed. Conclusions can be made qualitatively about the presence of particular *kinds of bugs* in other ROS packages. Furthermore, while the selection of repositories was purposeful, the identification of bugs and fixes was not: bugs and fixes were always reported and contributed by either developers, maintainers or users of the systems under analysis, not by the authors. The developers were treated as historical oracles and their assessments whether something is a bug, a fix or neither were taken at face value.

To increase *Dependability*, we detail how we gathered the dataset (Sections 2 and 3), how the historical images were build (Section 6), and how bug reports are structured, analyzed (Section 3) and stored (Section 6). All bug reports link back to the source data for each bug: affected source code repositories, the original issues, code contributions fixing the bug, and the state of the involved repositories *before* and *after* merging the fix. Bug reports also include timestamps for these events, whenever they were present and identifiable, and the full analysis as performed by the authors. All of the source material is made available open-source, on-line. Such traceability increases dependability, facilitating evaluation of the research methods and results (Shenton 2004).

To warrant *Confirmability*, the dataset was built from issue reports written by developers, maintainers, and users. Some of these reports are unambiguous and leave no margin for researcher bias or interpretation, mostly because of the detail of the report and the language used by the reporter. For instance, bugs related to the build phase of the software can hardly be mistaken for runtime issues. Others contain little description of the fault or the manifested failures. To minimize bias, we involved all authors in the analysis of such cases, relying on the buggy source code and the fix, until a consensus was reached, often over several meetings.

The commit history of the ROBUST repository, can be used to reconstruct how our bug reports changed over time, as a result of repeated analysis and discussion. Other aspects of our method can also be audited, as it is fully explained and all source code is available online. Despite this, we admit that a minimal set of bugs could be classified differently by another party for two reasons. First, we did not interview issue reporters to confirm whether our judgement matches theirs. Second, relying on the source code, especially on the changes introduced by a bug fix, might not tell a clear story because some commits might affect code that is not pertinent to fixing a particular bug. After completing our analysis, we estimate the accuracy of our labels by sampling 30 bugs from the dataset and recoding them according to our final taxonomy. Comparing the differences between the labels, we find that the original taxonomy had missing labels for three bugs and incorrect labels for a further three bugs, yielding a bug-level accuracy of 80% across our sample. Note that, only one label was incorrect or missing for each of those bugs (out of an average of 3.47 labels), giving us a label-level accuracy of 97%.

4 Analysis

We analyze the collected data to understand what bugs developers experience in practice in robotics systems. The discussion is organized along two questions:

RQ1: What software *faults* occur in robotics systems?
RQ2: What *failures* occur in robotics systems?

We ask RQ1 to determine the extent to which the faults in the data are specific to ROS and robotics, identify common ROS development pitfalls, and provide valuable insights to guide the construction of effective QA tools. We ask RQ2 to gauge the actual and potential consequences of software failures in open-source robotics software, and to understand what QA tools are needed to automatically detect such failures. By assessing the extent of potential failures we hope to motivate further constructive research on robotics software quality. The analysis follows the open thematic coding method described in Section 3. It gives a good qualitative description of the contents of the ROBUST dataset.

4.1 RQ1: What Software Faults Occur in Robotics Systems?

Table 3 outlines the top-level themes that emerged from the thematic analysis of the software faults in our dataset. We discuss them in detail below.

Build, Deployment, and Orchestration ROS software is structured in *packages* distributed with a package manager (e.g., `apt`, `pacman`, `dnf`) and in a source form. Packages are built using one of ROS's generic build tools, such as `catkin`. Under the hood, packages with C++ code are built with `CMake` using an accompanying `CMakeLists.txt` script obeying several ROS-specific conventions. Faults can appear in all specifications and scripts in the set up of packages, installation manifests, interface description languages (IDLs), and build. We include static compilation and linking errors under this theme as part of build [mavros:282c9be, universal_robot:39eb24f, motoman:ddc6f36]. In total, the ROBUST dataset contains 77 entries of this kind. Interestingly, this is the second largest theme within the dataset, suggesting that these robotics systems suffer from modularity, dependency, and

Table 3 Top-level themes identified in fault analysis of ROBUST bugs

Theme	Description	N
Build, deployment, orchestration	Faults in source code and build infrastructure that ultimately lead to failures at build, deployment and composition-time. This theme covers syntax errors, bad imports, broken dependencies, etc.	77
Run-time configuration	Faults that occur in configuration files that lead to a misconfiguration of the system. Includes incorrect parameters, arguments, constants, topics, and namespaces.	28
Concurrency	Faults related to the use of shared resources, including missing or flawed synchronisation, mistimings, and incorrect signal handling.	20
Evolution	Faults that are introduced by the change of a component, whether that be internal or external, that is not handled by the rest of the code. This include changes in programming language, directory structure, internal and external APIs, and underlying hardware and firmware.	37
General programming	Faults that typically occur during programming that are not exclusive to ROS or robotics. This includes incorrect logic and calculations, syntax errors, broken contracts (e.g., API and protocol misuses), and missing features.	123
Models	Faults affecting the accuracy and consistency of the robot's model of the world. This includes inaccurate robot and world models, missing or faulty transformations (e.g., coordinate frame transformations) and conversions (e.g., degrees to radians), and incorrect physical units.	20
Systems	Faults that either occur outside of the source code and configuration files of a system, or due to the interaction between the software and the system that it runs on. These faults may come from hardware devices and firmware, the system environment, or platform incompatibilities.	8

Column **N** shows the number of bugs labeled with a given theme. Bugs may be labeled with more than one theme

```
<package format="2">
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>This package provides foo capability.</description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>
  <url>http://ros.org/wiki/foo_core</url>
  <author>Ivana Bildbotz</author>
  <buildtool_depend>catkin</buildtool_depend>
  <depend>roscpp</depend>
  <depend>std_msgs</depend>
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
  <exec_depend>rospy</exec_depend>
  <test_depend>python-mock</test_depend>
  <doc_depend>doxygen</doc_depend>
</package>
```

Fig. 9 A Package XML file provides basic package metadata, including name, version, and dependencies (source: wiki.ros.org/catkin/package.xml, 2020-08-10)

distribution issues significantly. In fact, these issues are more prevalent in these systems than the robotics-specific issues. Below, we discuss the sub-categories within this theme.

- *Problems with code generation:* ROS automatically generates language bindings for clients using custom messages, services, and actions, from so called `.msg`, `.srv`, and `.action` files. The generators or the generated code need to be available for the client and server code to build. Typically the errors relate to missing dependency of the build on the generator or to missing imports of the generated files [kobuki:17560e9, careobot:105dc16].
- *Dependency problems:* A *manifest* `package.xml`[2] specifies dependencies for build-, test-, run-time, and documentation of a ROS package (Fig. 9). A dependency is either another ROS package (e.g., `std_msgs`, `roscpp`) or a *system dependency* installed via a package manager (`apt` on Ubuntu). The dataset contains numerous dependency faults both for package [motoman:9df36cb, geometry:e12e723] and system dependencies [careobot:c8091b6, careobot:ac6a181] that cause failures at build and execution time. A *missing run-time dependency* on `eigen_conversions` in the `depth_image_proc` package crashes the TurtleBot's image processing node, causing no images to be received from the camera [turtlebot:f01d952]. *Transitive* dependency faults are when a package wrongly relies on its dependency to provide another one. Kobuki developers expected that `libusb-dev` includes a required header file `stdint.h`, which is only packaged with `libusb-dev` on some systems [kobuki:0027b57]. *Circular dependencies*, where two packages depend on one another, can also cause inconsistent build outcomes and failures [kobuki:f95c384, turtlebot:3390789].
- *Problems with meta-packages:* Meta-packages simplify simultaneous installation of multiple related packages. They solely contain dependency specifications. ROS packaging standards do not allow proper packages to depend on meta-packages. The build tool,

[2] http://wiki.ros.org/catkin/package.xml

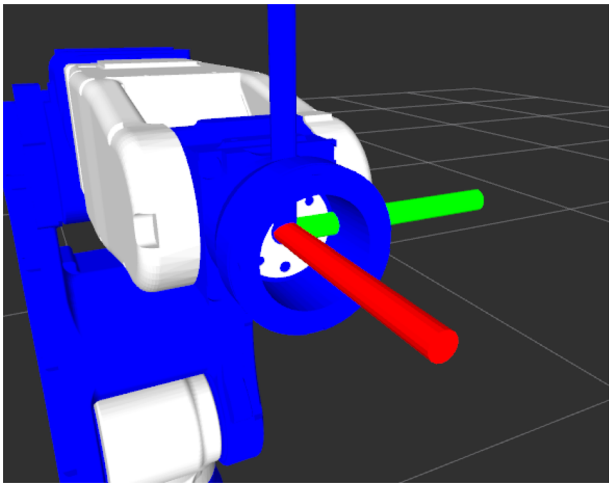


Fig. 10 A subtle fault in the specification of a joint led to an inaccurate robot model being used for planning, visualization, and simulation [motoman:1ec8ca1]

`catkin`, enforces this rule by reporting an error[3] [universal_robot:3a48064, universal_robot:a58f4b5].

– *Problems at the ecosystem level:* In rare cases, dependency problems appear outside the artifacts of the individual packages. In the case of kobuki:e3c9bbc, the fault was placed in the index of the ROS Distribution. A `source` version of a package was missing from the index, preventing others from downloading and building the package from source (Fig. 10).

Run-Time Configuration ROS supports developers to build complex robot systems by composing reusable software components (nodes). To reduce spatial coupling (Eugster et al. 2003), and to facilitate rapid prototyping, the nodes API is unaware of the nodes responsible for providing, reading from, or writing to a particular topic. Instead, nodes interact via topics and their associated names. The use of strings to identify communication channels allows nodes to be spatially decoupled, thereby facilitating rapid prototyping and dynamic architectural changes. However, such "stringly types" (Atwood 2012) prevent, for the most part, misnamings and misconfigurations from being discovered until run-time [mavros:263650d]. The dynamic, stringly-typed nature of ROS's run-time architecture forces nodes to rely heavily on conventions and assumptions, which are typically neither enforced nor checked.

To allow nodes to be used in a variety of contexts, configuration files often use name remapping and namespaces. Remappings (e.g., of topic names) are used to change names for a particular node, allowing nodes with different naming assumptions to interoperate. Unfortunately, incorrect remappings are easily introduced and sometimes difficult to identify [turtlebot:3e32933]. Topic remappings are often used together with namespaces to provide access to a group of related resources. Namespaces are typically used to safely manage multiple instances of a particular component (e.g., a robot, sensor, algorithm) by creating a scope. The use of hard-coded global names prevents multiple instances for a component and is considered an anti-pattern [turtlebot:a482f82].

[3] https://ros.org/reps/rep-0127.html

Parameters are also used to tailor the behavior of a component for a particular application or purpose. Failure can occur when incorrect values are supplied to parameters [kobuki:d9aa656] or when the wrong parameter name is used [kobuki:8a729db]. If a node attempts to read from an undefined parameter (e.g., due to a typographical error), ROS may quietly use a default value, leading to unexpected and difficult-to-debug behavior [mavros:e1a8005].

Concurrency ROS nodes can be implemented as either individual processes on the same or different machines that intercommunicate via network protocols (e.g., XMLRPC, TCPROS, UDPROS), or as threads within a single process that intercommunicate via zero-copy messaging. Naturally, ROS's distributed architecture leads to various concurrency and timing-related issues.

The most common concurrency-related fault within our dataset is a lack of synchronization [kobuki:62a38a9, geometry:15b2e3c, mavros:1f01916]. In some cases, synchronization primitives are present but are either misused, leading to liveness failures [geometry:74f0c66], or are incomplete, failing to provide synchronization generally [roscomm:ca23e58]. We also observe a small number of timing-related issues within the dataset [kobuki:f548cc7, kobuki:5a44ead]. For example, when reporting the states of the robot's joints to the `joint_states` topic, mavros:753226d failed to assign a timestamp to the `sensor_msgs/JointState` message, leading recipients to ignore the message as stale. Finally, faults may involve missing or incorrectly implemented signal handlers [mavros:29af3a3]. For example, in kobuki:f548cc7, a lack of appropriate signal handlers prevented the robot from safely and gracefully terminating its software processes upon receiving a `SIGTERM` signal.

Evolution Failures may suddenly appear as a result of changes to the environment in which the package is built and deployed, without any modification to its source code. Similarly, internal changes in one part of a package may not be reflected in other areas of that package leading to a variety of build issues and run-time failures [geometry:0481047, mavros:bdda1fa, kobuki:8a729db]. Potentially disruptive changes may occur through the introduction of newer programming language versions and compilers [geometry:7677ca7], operating system distributions [turtlebot:928306b], and ROS distributions [universal_robot:56cf07f], which lead to downstream issues in packages that rely on the older behavior. Changes to the robot's underlying hardware and its associated firmware that are not reflected in its software may also lead to issues [kobuki:b18f559, mavros:de2cc36].

Most commonly, problems arise as a result of changes to a dependency. While ROS packages may state their dependencies, there is no first-class mechanism for pinning those dependencies down to a particular version or set of versions, cf. PyPI (Coghlan and Stufft 2013). A library may alter, remove or deprecate parts of its API or ABI, leading to build failures [kobuki:5abe7d4, kobuki:9c8abeb] or unexpected behavior at runtime [turtlebot:61a75df, kobuki:55e84a6]. Issues are especially likely to occur when such changes are not reflected in the documentation [kobuki:9682b9a]. Existing source code, configuration, and data files may be changed [careobot:b826eae], moved to a different location [motoman:6a7a506, mavros:101c09b, kobuki:5abe7d4], or disappear unexpectedly from a dependency that provides them [kobuki:8c30446]. ROS names (e.g., topics, services, action servers, parameters) and namespaces may be inconsistently changed between source code and configuration files [universal_robot:778c1ac, mavros:263650d, kobuki:8a729db], and particular publishers, subscribers, and services may be changed or removed [mavros:bdda1fa].

General Programming Unsurprisingly, many faults within the ROS systems are general programming mistakes that could occur in any software, including typo mistakes, using the wrong logical or arithmetic operator [mavros:c172409], "copy-paste" or "clone-and-own" mistakes [geometry:439e235], code smells [confidential:86cb680,], and issues stemming from "stringly types" [mavros:dab1b8a].

We observe a variety of mathematical, logical, and control-flow issues. Several bugs stem from incorrect loop invariants [mavros:86255ba], mishandled loop variables [mavros:215010d], and missing loop break conditions [kobuki:e34428d]. A number of bugs occur as a result of missing or incorrect input validation [geometry:d12b890] and robustness measures [confidential:2688e7a], fail to adequately account for certain corner cases and boundary conditions [kobuki:af7946f], or lack important features [motoman:b1b6fcb]. Faults may also occur in mathematical calculations [geometry:860b866]. Bugs also occur as a result of API misuse and contract violation [universal_robot:b3c2c21], or, conversely, when an implementation does not satify its API specification [mavros:599c588].

We observe a number of issues specific to C++, for example: string length [mavros:2998e9f], string formatting [geometry:164cfa3], namespaces [geometry:b206807], multiple definitions [geometry:f19569c], and zero-copy messaging [kobuki:dbcdb12]. Uninitialized variables [mavros:fcf9cd9] and incorrect type casts and conversions [motoman:292b5cc] can lead to surprising failures at run-time. Numerous issues occur as a result of poor resource and memory management including off-by-one errors [universal_robot:cda133d, geometry:729a653], overreading buffers [universal_robot:359a2e9], and incorrectly calculated indices [kobuki:9397c6b]. Similarly, we also notice issues related to the misuse of Python features including missing or incorrect type conversions and checking [geometry:cec6208, geometry:e4466f0, geometry:d12b890], and bad imports [turtlebot:61a75df].

Models All of these faults lead to the robot forming an inaccurate model of reality. They may occur as semantic errors in the robot's URDF, Xacro, STL, and DAE files, which provide a physical and visual description of the robot for motion planning, visualization, and simulation. For example, the 3D meshes used to describe the robot may be missing, malformed, or incorrectly handled [motoman:0829607, motoman:6a7a506, kobuki:4ea5ea7]. Alternatively, the robot description files may incorrectly specify physical dimensions, mass, and inertia of the robot [kobuki:493e3f9, universal_robot:21b86f6, motoman:1ec8ca1].

Inaccurate models of reality can also come about as a result of missing or incorrect transformations and conversions [mavros:b96bf67]. For example, other:22e4e4f sees the visual and depth data from a Kinect V2 camera and its associated Freenect2 driver presented in a non-standard format, flipped around its vertical axis ("mirrored"), to an application that is unaware of the transformation. Similarly, an error in a parameter name in mavros:ff581a0 sees the robot incorrectly report its (x, y, z) coordinates as (x, x, x). In bug mavros:b96bf67, MAVROS, which operates in a different coordinate frame to MAVLink, fails to correctly handle rotation when converting and sending coordinates to MAVLink.

Systems Faults occur as a result of an interaction between the software and the system to which it is deployed. This includes the use of operating system and distribution-specific code or file formatting, preventing the software from being deployed to some platforms [kobuki:95b24e8, mavros:31ad11d, turtlebot:a4f35ee].

Alternatively, faults may occur due to interaction between the ROS software and an underlying (faulty) hardware device and its associated firmware (e.g., kobuki:841720a, kobuki:b18f559, mavros:de2cc36). For example, in kobuki:606b8b9, when the USB serial

cable is disconnected from the robot, the ROS driver node will, following the `udev` rules for the robot, attempt to read data from an unsupported bluetooth interface, leading the node to crash under certain conditions.

ROS packages can also provide shell scripts known as environment hooks which are typically used to set up environment variables. Issues may arise from the misuse of environmment hooks. For example, turtlebot:3ea2c30, introduced an environment hook that depended on a package that was not stated as a dependency in the package manifest. In cases where the other package was not coincidentally installed, error messages would be printed to the terminal.

4.2 RQ2: What Failures Occur in Robotics Systems?

We refine the research question RQ2 into the following sub-questions:

RQ2.1 What *immediate failures* occur in robotics systems?
RQ2.2 What *ultimate failures* occur in robotics systems?

Both questions address failures, but do so at different levels. Immediate failures, or the *software-level failures*, are failures that are immediately noticeable when testing or using software; for example, a ROS node crashes at startup or sends messages at a slower rate than expected. The propagation of the immediate failure throughout the entire robotic system might however result in a different kind of externally observable failure—the ultimate failure manifestation, or a *system-level* failure. For instance, a node that publishes incorrect velocity might make the robot turn to the wrong direction or become unresponsive, if other components detect and reject the incorrect values. Software-level failures rarely have no observable effect at the system level.

RQ2.1: Software-Level Failures

Table 4 summarizes the high-level themes that resulted from the thematic analysis of immediate software-level failures in `ROBUST`. For each theme, we list the relative frequency in the dataset. We provide further details on each theme below.

Build Bugs that manifest at build- or install-time are most often a direct consequence of faults under the *Build, deployment and orchestration* theme. Faults from other categories can also manifest at build time, when detected by code scanning tools or compilers. As they occur prior the system being able to run, they cannot result in a system-level failure.

Two main types of build bugs exist in this category: (i) bugs which cause immediate build failures, and (ii) bugs which cause failures when packages are consumed as dependencies by developers or users. Examples of the former include not linking against used libraries [kobuki:ddc6f36], missing dependencies on code generator targets [careobot:105dc16], violating packaging policies [universal_robot:0c34123], exporting incorrect package metadata [kobuki:45ee84a] and incorrectly specifying install targets [geometry:a723ecb]. Examples of the latter include missing declarations of *transitive* dependencies [geometry:4c160d3] and an incomplete build environment setup [kobuki:fd6b589]. Examples of deployment related bugs include kobuki:9de9690 and motoman:259b468, which both cause files expected by users to be absent from the runtime environment. The failures manifesting in dependent projects are particularly hard to diagnose, as they are observed by users in a completely different context than the one containing the fault.

Table 4 Top-level themes of software-level failures identified within the dataset

Code	Theme description	N
Build	Failures that prevent the successful completion of the build or installation process, such as compilation and linking errors.	54
User Experience	Failures that exclusively affect the user experience and do not result in software crashes or undesired behavior, nor affect the functional or non-functional performance of the robot, unless one considers the operator providing inputs to the robot via a user interface.	20
Performance	Failures that manifest in the degradation of non-functional performance of one or more software components, but do not necessarily result in any observable degradation of functional performance at the system level.	2
Crashing	Failures that lead to a software crash in an individual node at startup or runtime, such as failing to locate runtime dependencies, memory corruption or simple type errors in dynamically-typed programming languages.	
Liveness	Failures that cause an individual software component to become unresponsive due to, for example, a deadlock or an infinite loop.	15
Network	Failures that affect messaging behavior (between components) of individual software components, resulting in undesired functional behavior in either the sender, the receiver or both.	7
Behavioral	Failures that result in undesired functional behavior in one or more software components, and which do not lead to software crashes or liveness issues. Behavioral failures may be caused by, for example, miscalculations, logical errors, and violating contracts, or under certain unpredictable circumstances, memory-related errors (e.g., stack and heap corruption).	68

Column **N** shows the number of bugs labeled with a given theme. Bugs may be labeled with at most one theme; no label is applied to bugs that do not result in software-level failure

User Experience Bugs in this category influence the efficiency and efficacy of a robot operator affecting the way in which an operator either receives information from or provides commands to the robot.

Examples include misreporting the status of hardware subsystems, such as battery state [kobuki:841720a, kobuki:bb3c7ec], not showing information due to misconfiguration of the interface [kobuki:8a729db], incorrect positioning of 3D data in the UI due to parameter lookup falling back to defaults [mavros:84264f0], or because of updates to visualization tools which are incompatible with used 3D models [turtlebot:9299530]. Other example bugs include faulty data transformation causing wrong information to be shown to users [mavros:a67d81d] and presenting the same data twice under different names due to use of an outdated communication protocol [kobuki:b18f559].

Performance Failures observed as a reduction of performance of individual components, but which do not influence the perceived performance of a complete system. Examples include

not using language or framework features for efficiently passing data [mavros:0e2ea0c, kobuki:dbcdb12] and misconfiguration of framework features resulting in less efficient passing of data [kobuki:dd40270, kobuki:5ee65b0].

Crashing Failures observed as a crash of an individual node at either startup or run-time. The faults causing these failures include missing runtime dependencies (ROS nodes [turtlebot:891cb68], Python modules [careobot:c8091b6], shared libraries [turtlebot:f01d952]), incorrect linking of libraries [kobuki:3e88010], use of non-existent files [kobuki:9682b9a, geometry:fc854e0], use of malformed files [motoman:90a9464, careobot:b826eae], incompatible firmware on external devices [kobuki:c04eae5], management of resources [geometry:001fca6, geometry:6c13c78, mavros:215010d], out-of-bounds access [geometry:729a653, mavros:4fb6e7e], incorrect or unsafe concurrency [roscomm:ca23e58, geometry:12605ab] and hard-coding expectations about OS or runtime environment [turtlebot:a4f35ee].

Liveness These failures render an individual software component unresponsive due to, for example, a deadlock or an infinite loop. Liveness failures are a form of denial-of-service. They disturb the continued operation of a component or prevent the safe termination of the component or its service. Deadlocks occur both internally to a component [mavros:1f01916, geometry:74f0c66] and between an external system [universal_robot:bd1fce5]. Infinite loops can be caused by incorrect handling of termination conditions [kobuki:38dce2a, mavros:86255ba] or absence of such conditions [kobuki:6e748c1]. Other faults causing liveness failures include incorrect use of network functionality [roscomm:eab0d3c].

Network Network failures manifest by affecting messaging between individual components, resulting in undesired functional behavior in either the sender, the receiver, or both. Examples include incorrect handling of communication failures and protocol mismatches [confidential:2688e7a, confidential:332f09f, confidential:c5dc9de], failing to set message fields which may lead to subsequent misinterpretation [mavros:594978d], missing publisher queue sizes leading to inefficient and unintended blocking [kobuki:dd40270], and messages being lost due to an incorrect implementation of IPv6 [roscomm:eab0d3c].

Behavioral This theme covers run-time failures that result in undesired functional behavior other than software crashes or liveness issues. Behavioral failures are caused by, for example, miscalculations [kobuki:af7946f], incorrect implicit assumptions about input [confidential:96e2c6c], logic errors [geometry:566092b, motoman:292b5cc, roscomm:ca23e58], contract violations [mavros:599c588, mavros:de2cc36], misconfigured networking [kobuki:dd40270, roscomm:eab0d3c], incorrect processing of network data [mavros:594978d], or memory-related errors (e.g., stack and heap corruption). Some of these are robotics-specific, but most are usual specification violations known from regular software development.

RQ2.2: System-Level Failures

Many faults initially manifest in a component and then propagate to other components, possibly affecting the system as a whole, giving rise to *system-level failures*. We analyzed the repositories for drivers, controllers, and other components that are, ultimately, building blocks for concrete robotic applications. But, given that there is no well-defined concept

of *system* or *application* in ROS, what looks like a fully functional system (e.g., a robot patrolling a known map) can either be the complete system, as envisioned by the users, or simply a complex component of a larger system (e.g., multiple robots working in concert). Thus for bug reports devoid of information at the system level, such as what was the robot's mission and how it went wrong, we resort to to our experience and make an educated guess at how software-level failures manifest from a system perspective.

For RQ2.2, we performed a thematic analysis of the externally observable symptoms of failure at the system level. Table 5 lists the five high-level themes that resulted from this analysis. As we can see in the table, system-level failures are different from software-level failures. First, the definitions are more open-ended, in the sense that many types of observable symptoms fit under a given theme. Second, while the themes are distinct, there are no clear boundaries between themes, or a logical sequence over them. It is possible for a software failure to propagate and manifest in such a way that it fits multiple themes at the system level. For example, consider a robot driver that receives velocity messages, updates its estimated pose, and converts the velocity message into actuator commands for the hardware. If the conversion is miscalculated (e.g., wrong sign), the driver would report a correct pose estimation, as per the received messages, but would move in unintended ways (e.g., turning right instead of left). While this is clearly a behavioral failure (the robot does not do what it is told), it could also be perceived as a communication failure (the robot reporting a pose estimate that diverges from its observed movement).

Not all faults propagate all the way up. It is possible for software-level failures to be *harmless* at the system level, or to have no manifestation at all. For an obvious example, a

Table 5 Top-level themes of system-level failures identified within the dataset

Code	Theme description	N
Loss of Functionality	Failures that make the functionality offered by a group of system components unavailable. User's perspective: *"the camera stopped working!"*	42
Unresponsive	Failures that make the system become unresponsive to a number of operator commands. User's perspective: *"it does not move!"*	26
Degraded Performance	Failures that cause the system to miss deadlines or perform one or more tasks with reduced timeliness. User's perspective: *"it moves too slowly!"*	7
Behaving Incorrectly	Failures that make the robot perform unintended movements or actions, or perform the intended commands but with unexpected outcomes or side effects. User's perspective: *"it turned left instead of turning right!"*	44
Monitoring	Failures that result in inaccurate observation and diagnosis of the system's current state, via user interfaces of various types. User's perspective: *"I cannot see the battery levels!"*	11

Column **N** shows the number of bugs labeled with a given theme. Bugs can be labeled with more than one theme; no label is applied to bugs that do not result in system-level failure

failure during the build process does not affect the system as a whole, because the system is not yet deployed at this stage. The ROBUST dataset contains 106 bugs that had no meaningful manifestation at the system level. An example of this is kobuki:8163705, in which developers handled a raised exception within the code, but reported the handled exception in the error logging stream instead of the debug stream. This misled users into believing that an actual error was occurring. Despite the faulty component, the system as a whole worked correctly. We now provide further details on each of the identified failure themes.

Loss of Functionality This theme covers failures that bring down parts of the system. Partial loss of functionality prevents the use of certain non-critical features under particular conditions. This is typically something the system can recover from even though it might affect its overall performance. Substantial losses, on the other hand, are more likely to result in mission failure. For example, if a robot's safety controller is unable to respond to bumper events it would not hinder the robot's ability to perform its mission, if it never bumps into an obstacle, or if a redundant sensor is present. But losing the safety controller in its entirety, would likely be considered too dangerous to keep the robot in operation.

The software-level failures that most likely escalate to a loss of functionality are crashing and liveness failures. If a component, or group of components, crashes or becomes otherwise unresponsive, the functionality it provides is lost to the system. Behavioral failures can also result in a loss of functionality. When a component behaves incorrectly due to logical mistakes, a functionality may be locked away due to the program never entering the appropriate execution path. User experience failures propagate in a similar fashion; a poor user interface might omit features by missing buttons or using unclear language. Network failures are more varied; loss of functionality, in this case, could be due to dropped messages in a misconfigured system, where the channel that enables the functionality becomes unusable. Performance failures are less likely to lead to loss of functionality, unless features are lost when their response times become too slow.

In the ROBUST dataset, most software-level failures that escalate to a system-level loss of functionality are crashes (20 out of 42). The remaining bugs are split between liveness (2), network (8) and behavioral failures (12). There are examples of both partial and substantial functionality loss. In mavros:86255ba an infinite loop prevents users from accessing and manipulating parameter values at runtime, while in roscomm:eab0d3c the robot is unable to communicate with a ground control station (both partial losses). Crashes in the robot driver prevent interaction with the robot base [motoman:377d7be, universal_robot:359a2e9], and a bad topic remapping causes velocity commands to be ignored, preventing all movement [kobuki:35682ec] (critical functionality losses). Software crashes in the vision pipeline, an essential part of localization and navigation, can also prevent autonomous movement of the robot [turtlebot:f01d952, turtlebot:3e32933].

Unresponsive These dangerous failures make the system unresponsive from the user perspective. The human operator cannot control the robot, despite it being turned on and working. The operator might be unable to make the robot perform its mission and to react to unintended and potentially dangerous behaviors.

Any software-level runtime failure can escalate to unresponsiveness. A crash of a node can prevent operator inputs from reaching the robot. A node that behaves incorrectly, say a velocity multiplexer, can prevent user input from reaching the robot due to logical errors. The ROBUST dataset contains examples of unresponsiveness that occur deterministically at startup [kobuki:fbe70c7, mavros:263650d], unexpectedly at runtime [uni-

versal_robot:359a2e9, universal_robot:bd1fce5], or when the operator attempts to safely stop the robot [kobuki:38dce2a, mavros:29af3a3]. In terms of the software-level failures, 20 out of 26 records fall under one of two sources: *crashing* and *liveness*. In the former, robots may become unresponsive due to missing runtime dependencies that crash core components [universal_robot:58790ba]. As for the latter, robots may become unresponsive due to, e.g., infinite loops [mavros:86255ba]. In another example, the program ignores the termination signal, preventing the user from safely terminating the robot with `SIGTERM` if the robot begins to behave dangerously [kobuki:054c753].

Degraded Performance Safe and effective robot operation often relies on the system behaving in a timely manner. Some computations are expected to complete periodically, with a given frequency; others are expected as a timely reaction to a given stimulus. A system fails due to *lag* (reduced timeliness) if it does not meet expected deadlines. For many robotic tasks, such as detecting and avoiding obstacles or picking up parts from a conveyor belt, a delay means failure.

Any software-level runtime failure may reduce system timeliness. A crash of a node that is set to `respawn` (i.e., start again after sudden termination) is a failure from which the system can recover, but likely with noticeable delays. Intermittent liveness failures can easily lead to reduced timeliness, too. Still, performance-related software failures (e.g., CPU and GPU load) are likely the main culprit of degraded non-functional performance for the system as a whole.

The seven software failures in `ROBUST` that may affect the timeliness of the system are of types *crashing*, *liveness*, and *performance*. For example, geometry:1b5fa94 leaks memory at the component level, at a rate of about 15 megabytes per second. When running for long enough, systems tend to slow down, due to the lack of resources, until they eventually crash.

Behaving Incorrectly A robot's observable behaviour is described not only in terms of digital effects as in other software systems, but also in terms of physical effects, such as moving from one place to another. When the software produces unintended outcomes, such as moving in the wrong direction or at a wrong velocity, we face a functional failure. This type of failure can easily pose physical danger to humans, living beings, the robot itself, and surrounding valuable objects.

Any software-level failure could end up escalating to incorrect functional behavior. Crashes or liveness failures in runtime monitoring components, such as safety controllers, could make the robot move into objects. Performance failures in components that issue movement commands can prevent the robot from acting as intended. User experience failures can also lead to unintended movement. If a user interface assigns the wrong label to a command button, the user would observe a robot movement that does not match their expectations. Finally, behavioral failures (especially those stemming from miscalculations or logical errors) are among the most likely causes for unintended movement.

Many functional failures in the `ROBUST` dataset originate from behavioral software failures (36 out of 44). Failures may be caused by incorrect or imprecise calculations [geometry:439e235, kobuki:f7946f], missing frame conversions [mavros:248cb38], and the use of absolute rather than relative time [universal_robot:b3c2c21]. In kobuki:af7946f, a calculation error causes the robot to move slowly and turn in wrong direction when dealing with low negative linear velocities and rotation commands. Unintended movement may also occur as a result of rounding errors and numerical imprecision [motoman:9bf25ea, kobuki:1c141a5]. For example, in kobuki:1c141a5, a loss of precision while converting a `float` to a `short`

during a repeated calculation caused the robot to move in the opposite direction in special cases. Missing or incorrectly implemented safety features can also lead to unintended motion, such as in motoman:b1b6fcb, where the robot is able to immediately resume motion from a paused state. In kobuki:ad906f0, a lack of acceleration smoothing causes the robot to abruptly perform wheelies when instructed to move from an idle position. Lastly, hard-coded speeds in kobuki:0416c81 prevent heavier-than-anticipated robots from being able to move, and inaccurate joint limits in universal_robot:89145c4 and motoman:2d42582 cause the robot arm to collide with itself or attempt to reach physically unreachable positions.

Monitoring These failures hide or misrepresent the robot's internal state from a user perspective. They prevent the human operator from accurately observing or diagnosing the robot. Being able to observe the state of the robot is crucial to ensure an additional layer of safety and emergency responses from the human operator (explainability). Accurate monitoring may help prevent a robot from hitting an obstacle or overloading hardware components.

Almost any failure can lead to divergence between the actual state of the robot and what the operator observes. Incorrect calculations in a sensor node can manifest in incorrect behavior (publishing wrong data), which could cause the system to build and display an incorrect model of the world. Node crashes, node liveness issues, or degraded non-functional performance likely result in stale data and a model that lags behind the actual state of the robot and its environment.

Our dataset contains mostly examples of incorrect *behaviour* that escalate to monitoring failures (9 out of 11). For instance, in geometry:860b866 incorrect vector translations end up producing wrong pose values, which ends up producing incorrect visualizations. In mavros:fcf9cd9 and mavros:753226d issues with visualization are caused by uninitialized variables carrying wrong default values.

5 Findings

In this section, we interpret the results of our study in light of the existing literature and discuss the implications for both practitioners and researchers. Table 6 summarizes the key findings and their impact on future work.

Building high-quality reusable robotics software components is difficult ROS's modular design, rich package ecosystem, and spatially decoupled run-time architecture allow its users to quickly and easily build diverse robots using off-the-shelf hardware and software components without being an expert in all areas of robotics (Kolak et al. 2020). *However, those same characteristics make it difficult or impossible to anticipate incorrect or unintended interactions between that ROS module and its application, firmware, operating system, and hardware context.* Sources of variability include: hardware components, models, and firmware; operating system distributions; language and compiler versions; ROS parameters and launch arguments; and package versions. Even if the source code for a ROS node remains unchanged, its environment will continue to evolve, creating opportunity for unexpected failures.

Exhaustively testing all possible variations to find potential failures is both technically challenging and prohibitively expensive. Instead, the risk of an unexpected failure due to unexpected interactions can be mitigated by identifying, minimizing, and testing sources of variability within a process of continuous integration and deployment. Most software-related sources of variability (e.g., OS distributions, language and compiler versions, package ver-

Table 6 A summary of the key findings from building the ROBUST dataset

Finding	Impact / Action / Future Work
Many late-stage failures are caused by misconfiguration, feature interactions, and co-evolution of components, platforms, and hardware.	There is need for better tools to detect and address potential incompatibilities between hardware and software components ahead of deployment. Our dataset provides many case studies for researchers willing to address this challenge.
Many of the problems contained in the ROBUST are similar to those experienced by other software engineers.	This calls for prioritization of research and innovation. Consider whether it is more beneficial to focus on general software technology or on specific tools aiming at narrower problems.
Many of the general programming mistakes within ROBUST could, in theory, be caught by existing, general-purpose QA tools and practices.	More action research combining software engineering researchers and robot programmers is needed to better understand this gap, to increase awareness of QA methods & tools within robotics, and to make software engineering methods more accessible to programmers from non-computing backgrounds.
Detecting system-level failures relies *heavily* on human oracles; Specifying and monitoring intended behaviour is fundamentally difficult.	ROBUST provides 115 examples of such bugs that can be used to drive research into oracle specification and inference and automated testing for robotics software more broadly.
Many bugs are detected during systems integration. ROBUST bugs were reported against software components, thus, in principle, they should be detectable in earlier life-cycle stages.	ROBUSTprovides a dataset to experimentally assess whether, indeed, unit testing, fuzzing, combinatorial testing, or other automated techniques could have detected these faults early, and if so, at what cost.
Failures are complex but fixes are often simple. However, they are heavily domain- and project-specific.	We recommend developing generic methods for building lightweight program analysis and program fixing tools, in the spirit of language workbenches (lightweight tools for developing languages).

sions) can be controlled effectively by using containers (e.g., Docker) to package, distribute, and deploy specific versions of individual nodes. Indeed, the use of Docker containers has gained popularity within the ROS community (White 2019). Hardware-related sources of variability (e.g., differing models, revisions, firmware) are harder to minimize and must be tested via the slow, expensive, and complicated process of software–hardware (SW/HW) integration testing (Afzal et al. 2020). This testing is often performed in an ad-hoc manner by taking a joystick and manually running the robot through several scenarios, chosen based on intuition. However, more sophisticated setups have been used to perform continuous integration of hardware and software by deploying new software releases to a fleet of robots within a controlled testing facility outfitted extensively with a variety of sensors (Henning 2016).

Given the inherent costs and associated challenges of dynamically detecting errors due to misconfiguration, there is a need for tools that can reliably detect the presence of such errors ahead of testing and deployment. One approach is to require or allow additional specification from users (Bardaro et al. 2018; Martinez et al. 2021; Bozhinoski et al. 2021). Indeed, ROS 2 is moving in this direction. For example, ROS 2 requires that parameters be declared and allows users to specify expected types and value ranges that are checked at run-time (ROS2 2021; The Robotics Back-End 2021; Foote 2021). Care needs to be taken however, as increasing the specification burden upon users can be both heavyweight (i.e., writing programs takes longer) and constraining (i.e., not all programs are covered). As an alternative to relying on increased specification, a number of papers have proposed the greater use of inference (Witte and Tichy 2018; Santos et al. 2016, 2021, 2019; Timperley et al. 2022). For example, tools such as HAROS and ROSDiscover combine the coarse-grained architectural information provided in ROS configuration files with static analysis of source code to approximate run-time architectures. Whether inference tools can be sufficiently powerful to produce accurate results and eliminate the specification burden remains an open question. Instead, the most effective approach may be in building inference techniques that exploit lightweight specification that is quick and easy for developers to provide, such as that required by ROS 2 parameters.

ROS developers make the same mistakes as other developers Most ROS software is written in C++ and Python. Naturally, it inherits the same set of general programming mistakes that are observed in those languages. Given the potentially catastrophic consequences of failure within the domain of robotic systems, it is vital that we reduce the space of possible failures by eliminating the possibility for common programmer mistakes (e.g., memory management, unhandled exceptions, missing synchronization). This may be achieved either through the use of safe languages (e.g., Rust (Matsakis and Klock 2014), Erlang (Virding et al. 1996)), the introduction of new safety features and programming abstractions into existing languages (e.g., RxROS (Larsen et al. 2021)), or the development of easy-to-use analysis tools that reliably detect certain classes of error (e.g., PhrikyUnits (Ore et al. 2017), Phys (Kate et al. 2018), PhysFrame (Kate et al. 2021)). Solutions need to be accessible to ROS's broad demographic of users, without compromising run-time performance or entirely sacrificing the lightweight, prototypical aspects of ROS. Advanced computer science knowledge should not be assumed (Alami et al. 2018). *It is a crucial finding of this study that, in fact, most software development problems faced by robotics developers, as documented in issues and pull requests in* `ROBUST`*, are similar to those experienced by other software engineers. The robotics-specific issues do not dominate the landscape.*

Many problems are in classes addressed by existing tools Just as ROS developers make the same mistakes as other developers, *we observe that many of those mistakes could be caught by existing, general-purpose QA tools and practices.* For example, incorrect or missing build and run-time dependencies can be quickly detected by automatically building and fuzzing as part of a continuous integration pipeline. Certain runtime errors bugs caused by, e.g., out-of-bounds memory access, data races, and resource mismanagement, may also be detected statically via analysis tools (e.g., Infer (Facebook 2022), Footpatch (van Tonder and Le Goues 2018)) or at runtime via monitoring instrumentation (e.g., AddressSanitizer (Serebryany et al. 2012), MemorySanitizer (Stepanov and Serebryany 2015), ThreadSanitizer (Serebryany and Iskhodzhanov 2009)). *A key challenge lies in identifying and eliminating the barriers to adoption for these tools and techniques within the open source robotics community.*

Bug detection relies heavily on human-in-the-loop testing Many software failures in the dataset (106 of 221) have no noticeable effect at the system level. As many as 63 of the 106 occur at build time, are easily detected, and, by definition, have no effect on system behavior. The remaining 43 failures, however, are difficult to detect, lacking immediately noticeable effects on system-level behavior.

The 115 bugs that lead to externally observable symptoms vary considerably in terms consequences, ranging from mild annoyance to potential catastrophe depending on the operational environment. *A common theme among these bugs is that their detection relies* heavily *on human oracles: It is hard to specify expected behaviour at the system level, especially in terms of observable physical effects, and it is harder still to automatically observe and analyze such behaviour.* The 115 bugs mentioned above are by themselves a research agenda for the testing community.

Indeed, the challenges of writing tests are reflected in ROBUST: fewer than 10% of relevant bug fixes (15 of 156) are accompanied by a test case (Section 3). Other studies have made a similar observation. In a sociocultural study of the ROS community, Alami et al. (2018) found that the ROS community values the creation of new features and functionality, QA tasks are perceived as difficult and time consuming, are often undervalued and neglected. In a series of semi-structured interviews of robotics practitioners, Afzal et al. (2020) find that field testing is the predominant means of QA within robotics, and that the challenges prevent or dissuade developers from writing automated tests.

Given the importance and difficulty of writing tests for robotics software, there is an urgent and growing need for new tools, techniques, and infrastructure to allow developers to easily and effectively test their robotics software. To that end, researchers have proposed test input generation techniques for individual components (e.g., Santos et al. 2022) and entire systems (e.g., Kim et al. 2019), methods for inferring and monitoring intended robot behavior (e.g., Aliabadi et al. 2017; He et al. 2019; Inoue et al. 2017; Afzal et al. 2021b), and domain-specific languages for describing simulated testing environments (e.g., Fremont et al. 2019; Majumdar et al. 2019; Klück et al. 2018; Afzal et al. 2021a).

From a software quality point of view, employing a variety of techniques (possibly in a dependability case argument) is likely the best approach. In general, it might be easier to simply detect the underlying software faults with automatic or semi-automatic tools, than it is to detect the failure at the system level. At the very least, all the bugs collected in ROBUST have been reported to issue trackers of software components, which indicates that a regression tests for them might be possible. These further reinforces our view that failures should be shifted *to the left* as far as possible, closer to build time, and ideally contained in single components.

Failures are complex; fixes are often simple While failures can be complex, unpredictable, and affect multiple components in several languages, over 90% of bug fixes are limited to a single language, almost two thirds of bugs are fixed in a single file, and most bug fixes are relatively small in terms of their number of lines added, modified, and deleted (Section 3). Program repair techniques are most effective when the necessary syntactic changes are both small (i.e., require changes to few lines of code) and isolated to a particular region of the program (e.g., a single file or method). *Thus, our findings are encouraging for existing APR techniques whose success is likely to be bound by the lack of sufficient tests rather than the complexity of the necessary repairs.*

That most bugs are fixed in a single language is encouraging for development of analysis and automated repair tools, which are typically limited to a single language. However, many

bug fixes occur in domain-specific languages or dialects that lack analysis tools, or else involve interactions with external components that are difficult to deal with using existing techniques. *While research on domain-specific languages resulted in many efficient and generic techniques for their implementation* (Lämmel 2018; Wąsowski and Berger 2023)*, static analysis tools and testing methods for domain-specific models have hardly received the same attention, even though, as our data indicates, they would be potentially useful in robotics software engineering practice.*

We list ROS-specific analysis tools and techniques that help to tackle some of these challenges. Techniques have been proposed for both statically and dynamically determining and verifying interactions (e.g., over topics and services) between ROS components (Witte and Tichy 2018; Santos et al. 2019). Purandare et al. (2012) implement an analysis that helps to explain changes in conditional message flows (i.e., the conditions under which messages are published) between program versions to aid in debugging. Fischer-Nielsen et al. (2020) provide an approach for automatically identifying and fixing missing ROS package dependencies. HAROS (Santos et al. 2016) provides a graphical front-end for presenting and integrating the results of various analyses on ROS systems.

6 Benchmark Infrastructure

To allow our dataset to be used to assess new QA tools and techniques, it is vital that we provide access to accurate recreations of the build and run-time environments in which each of the identified bugs took place. Reproducing historical software environments is never trivial, but ROS, its tools, and the operating systems it runs on proved to be an exceptionally volatile mix in this project. The interdependence between tools and the required versions of software libraries complicated building up the infrastructure needed to allow restoration of the bugs in `ROBUST`. While we recognise this infrastructure is specific to the context of ROS, the challenges it addresses are not specific to ROS, and would be experienced when restoring historical snapshots in any ecosystem depending on heterogeneous, independently evolving components.

Identification and Retrieval of Historical Source Code A historical bug is embedded in source code with many static and dynamic dependencies. In order to build the system with the bug, one needs the exact version of the buggy source code with all its build and run-time dependencies as they were at the point in time where the bug was identified—a *contextual restoration snapshot*. The following properties of historical ROS bugs necessitate the recreation of a snapshot of more than just the buggy component:

1. The bug has likely been fixed since it was reported which means it can no longer be found in up-to-date versions of the source code.
2. The code containing the bug, along with its dependencies and dependants, have evolved, which precludes compilation of that code on current systems.
3. ROS is an evolving platform, which prevents reproduction with new versions; even properties other than source-code-level compatibility change, e.g., run-time behavior of components and semantics of message exchanges.

The possibility of creating a contextual restoration snapshot depends entirely on our ability to identify the exact version of the source code which caused the bug. For `ROBUST`, this information has been extracted from issue reports, pull requests, and by inspection of the revision log of the version control systems used by the authors and maintainers of the analyzed

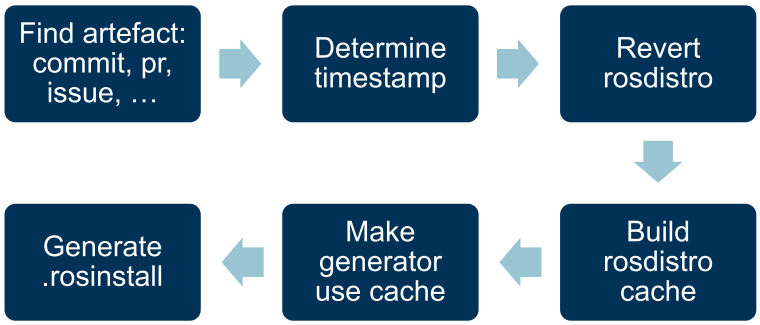


Fig. 11 Diagram of the rosinstall generator time machine workflow

components. For each bug, we establish the contextual reproduction snapshot as either: (a) the *parent commit* of the bug-fixing commit, if the bug has been fixed, or (b) the last commit *preceding* the date and time at which the bug was reported. The snapshot does not contain the fix, but the buggy code.

Recreation of the ROS Distribution Figure 11 shows a visual representation of our approach. Once we have identified the contextual reproduction snapshot and its associated time, we reproduce the ROS distribution as it was "back then," by computing the transitive closure of all dependencies of the subject package. For a typical bug, this set can range from three to a hundred packages, both ROS and standard OS packages (called *system dependencies* in ROS). We gratefully make use of the index of all *released packages* in a distribution, centrally maintained in a GitHub repository known as `rosdistro`.[4] A file called `distribution.yaml` in this repository lists all packages released into a specific ROS distribution including their source repositories and the commit hashes and branches used for a particular release. Figure 12 shows an entry for the `kobuki_core` package for a particular ROS distribution (Kinetic) at a specific point in time. We revert the `rosdistro` database back to the right point in time and then use a ROS tool called the `rosinstall_generator` to compute the previously mentioned closure of dependencies. The result of this, a `rosinstall` file, describes the locations and versions of all required dependencies along with a pointer to the repository containing the reproduction snapshot of the package which contains the bug. Figure 13 shows an excerpt from such a `rosinstall` file for the kobuki:45ee84a bug.

We then use this file to clone all the required repositories at the right commit and build them using their original (CMake) build scripts. (Of course, for non-ROS packages this procedure will be slightly different as they may use a different build-system, however the process largely remains the same.)

Admittedly, this restoration procedure does not capture the versions of all dependencies as they were exactly at the snapshot time on the particular computer when and where the bug was identified and potentially fixed, but rather, it selects the versions that were *released* at that point in time. We assumed this to be sufficient for most cases as developers typically work with the latest available releases of packages. Of course, this procedure also will not capture cases where a developer had either a really old, really new, or unreleased version of a particular dependency installed and the bug depends intimately on that particular version. For the bugs described in `ROBUST`, however, we determine that this is seldom the case, and

[4] https://github.com/ros/rosdistro

```
kobuki_core:
  doc:
    type: git
    url: https://github.com/yujinrobot/kobuki_core.git
    version: kinetic
  release:
    packages:
    - kobuki_core
    - kobuki_dock_drive
    - kobuki_driver
    - kobuki_ftdi
    tags:
      release: release/kinetic/{package}/{version}
    url: https://github.com/yujinrobot-release/kobuki_core-release.git
    version: 0.7.8-1
  source:
    type: git
    url: https://github.com/yujinrobot/kobuki_core.git
    version: kinetic
  status: maintained
```

Fig. 12 `ros/rosdistro` entry for Kobuki, showing several released packages and the location of the repository containing the source code (ROS Kinetic, `ros/rosdistro@cdfe60d2`)

where it was discovered this played a role in reproduction of the bug, manual changes to the output of the `rosinstall_generator` were made to account for this.

Reproducing the historical version of a ROS distribution is one of the hardest steps in the process. The existence of tools in ROS, such as the generator and the dependency modelling and introspection tools, has facilitated this greatly. In projects where seeding of development environments is less structured and automated or is not using explicit package management and metadata, along with the corresponding tools, this would have been much harder.

Restoration of the Non-ROS Environment Even if we have successfully identified and copied the historical versions of the source code and computed and cloned all the correct dependencies of packages, compilation often fails on machines with up-to-date build environments.

```
- tar:
    local-name: catkin
    uri: https://github.com/ros-gbp/catkin-release/.../catkin/0.6.11-0.tar.gz
    version: catkin-release-release-indigo-catkin-0.6.11-0
- tar:
    local-name: ecl_command_line
    uri: https://github.com/yujinrobot-release/.../ecl_command_line/0.61.0-0.tar.gz
    version: ecl_core-release-release-indigo-ecl_command_line-0.61.0-0
- tar:
    local-name: ecl_license
    uri: https://github.com/yujinrobot-release/.../ecl_license/0.61.0-0.tar.gz
    version: ecl_tools-release-release-indigo-ecl_license-0.61.0-0
```

Fig. 13 `rosinstall` file listing the dependencies of `kobuki_ftdi` in the ROS Indigo distribution (bug: kobuki:45ee84a). Note: `uri` entries have been shortened

This is caused by the fact that operating systems, system libraries, run-time environments, compilers, interpreters and build tools are continuously evolving and the subject package may depend on particular versions of any of these. Such dependencies may even be only implicitly embedded in the package, as explicitly stating versions of system dependencies is not often done in ROS packages. Furthermore, dependencies may become unavailable on newer systems. For example, code repositories could disappear, operating system distros could become unavailable, or the compiler or other toolchain elements could have been discontinued entirely.

In the period covered by `ROBUST`, between 2010 and 2018, Canonical released 10 different Ubuntu versions (and still supported three older LTS releases, GCC saw 36 releases (including patch releases), Boost (a main dependency of ROS and thus all ROS packages) released 20 versions and Python supported a range of both 2.x and 3.x versions at the same time, spanning from version 2.6 on Ubuntu Lucid to version 3.5 on Ubuntu Bionic. ROS itself saw 12 releases in the same period, with multiple ROS versions supporting multiple Ubuntu versions (Table 7).

We use containerization to manage the historical contexts required by the bugs in `ROBUST`. Each entry is accompanied by a Docker image containing the appropriate historical code snapshot accompanied by the correct versions of tools and the right build and run-time environment needed for reproduction. Containers also allow for relatively easy and efficient distribution of pre-built environments, which reduce the time it takes researchers to start working with these bugs, as sessions can be started after a download of the pre-built context, instead of having to wait for one to be built on demand.

To avoid having to handcraft images for each bug in `ROBUST`, we make use of *BugZoo* (Timperley et al. 2018b), which simplifies the specification of parameters and build file generation. BugZoo takes care of building and launching Docker containers for specific bugs by using information from the `ROBUST` bug description file combined with the `rosinstall` files. As many bugs share the same basic container structure, a template `Dockerfile` is used by BugZoo to generate the recipes for specific bugs.

Extensibility By following the process described in Section 2 more bugs could be added, and not just for the packages already analysed by the authors. The same procedure could be used for any ROS package, the only requirement being that the package has been released at some point in its lifetime. This is due to limitations in the tooling used to compute the transitive closure of dependencies for a specific package, which depends on `rosdistro`

Table 7 Versions of ROS supported on various Ubuntu versions (Long Term Support releases only) and the versions of core dependencies of ROS on those platforms

Ubuntu	ROS	GCC	Boost	Python 2	Python 3
Lucid (10.04)	C, D, E, F	4.4	1.40	2.6.5	3.1.2
Precise (12.04)	F, G, H	4.6	1.48	2.7.3	3.2.3
Trusty (14.04)	I, J	4.8	1.54	2.7.5	3.4
Xenial (16.04)	K, L	5.3	1.58	2.7.12	3.5.1
Bionic (18.04)	M	7.4	1.65	2.7.15	3.6.7
Focal (20.04)	N	9.3	1.71	2.7.17	3.8.2

Ubuntu versions listed by their codenames. ROS versions listed only by the first character of the codename: *C-Turtle*, *Diamondback*, *Electric*, *Fuerte*, *Groovy*, *Hydro*, *Indigo*, *Jade*, *Kinetic*, *Lunar*, *Melodic* and *Noetic* respectively

for information about those dependencies and the state of other ROS packages part of the historical snapshot.

Researchers from both the software repair and the ROS community could submit new bugs for inclusion into the database by submitting pull requests against the ROBUST GitHub repository.

Durability While using container technology allows us to safeguard reproduction snapshots against decay and becoming incompatible with ever evolving OS and run-time environments, by itself, this technology does not solve the problem of persistence of the dataset. Container technology itself will most likely evolve, and the implementation which today enjoys the greatest popularity may not exist in the future, which would reduce all our containers to static data archives instead of packaged, executable environments. While we cannot prevent this from happening, we have employed several different tactics to mitigate the impact of these types of bitrot, to the point where recreation of the snapshot itself should at least be possible if direct reuse of the containers themselves turns out to be difficult.

First, the process of container creation is as much as possible captured in code, and documented in natural language. These plain-text artifacts are all committed to an open-source Git repository.

Second, all technology used for the creation of the current containers is open-source, without relying on any proprietary tools. Proprietary tools have a clear disadvantage when it comes to reproducibility: without source code available, rebuilding them on future platforms will be impossible and run-time support limited to platforms which are compatible today. By comparison, open-source tools, while perhaps also not immediately usable, offer at least the possibility of fixing any limitations which might prevent them from being used in newer run-time environments.

Third, even though the source repositories which host the subject systems are all open-source and available online, we create forks of those repositories and host them in the robust-rosin organization on GitHub. Those forks are then used to build Docker container images. This organization is under our control, and forking (i.e., copying with preservation of provenance) creates a duplicate of the forked repository which prevents changes to the source repository from tampering with the code history or structure.

Finally, every container includes a copy of the source code repositories which make up the reproduction snapshot. These repositories were identified using the process and tools described above and include both the subject package (containing the buggy code) as well as its (transitive) dependencies. Should a container no longer be functioning, these sources could be extracted from it and used to repopulate a new reproduction environment.

7 Related Work

Bug Datasets A variety of bug datasets have been proposed as benchmarks for evaluating fault localization, fuzzing, test prioritization, bug diagnosis, and program repair techniques. Datasets have been created for a variety of languages and platforms including JVM-based languages (Benton et al. 2019), Java (Just et al. 2014; Madeiral et al. 2019; Saha et al. 2018; Lin et al. 2017; Tomassi et al. 2019), Android Java (Tan et al. 2018), NodeJS (Gyimesi et al. 2019), Python (Hu et al. 2019; Lin et al. 2017; Tomassi et al. 2019; Widyasari et al. 2020), and C (Böhme and Roychoudhury 2014; Le Goues et al. 2015; Böhme et al. 2017; Tan et al. 2017; Do et al. 2005).

Some of the buggy programs within these datasets are provided exclusively as source code, and do not come with a self-contained environment for accurately reproducing their behavior. Over time, this can lead to programs that are unusable or that produce different results, often due to unstated dependencies (e.g., libraries and compilers). Other datasets, such as ManyBugs and IntroClass (Le Goues et al. 2015), are provided as a single virtual machine (VM) image. While VMs provide a stable environment for conducting experiments, they often lack transparency (i.e., it is unclear how they were produced), make it difficult to use new software and tools inside old VMs, and have considerable performance overheads, making them unsuitable for time-sensitive and resource-intensive systems such as those used in robotics.

Most recently, datasets have moved toward using Docker containers as the preferred means of distribution. BugSwarm is a relatively new benchmark that currently consists of over 3000 bug fixes in open-source Java and Python projects. BugSwarm is continually populated by mining GitHub and TravisCI to find regression fixes. In our dataset, we observe that tests are rarely provided with bug fixes and that tests are lacking, generally, indicating that BugSwarm's approach would be unable to find most of the run-time bugs of our dataset.

We build on top of BugZoo, which packages bugs as individual Docker container images and a set of machine-readable instructions (e.g., for building and and testing the buggy program), to allow reproducible interactions. We significantly advance the state of the art in bug reproduction through the introduction of our "time machine," which obtains a historically accurate context for each bug, and a generic Docker setup that can be used to recreate historical versions of arbitrary ROS packages. Our machinery and methods can be reused by others to study a greater portion of software within the ROS ecosystem.

Bug Studies Our methodology for analyzing and documenting bugs is inspired by Abal et al. (2014, 2018) qualitative study of variability bugs in the Linux kernel. Their dataset provides a natural language description, execution trace, CWE classification, location, presence condition, and traceability information (e.g., repository URL, commit hash, associated issues) for each bug. Our approach also includes these elements, where relevant, and goes further by incorporating a qualitative analysis of faults and failures, and providing historically accurate Docker images for each bug.

Numerous empirical studies have investigated the characteristics of bugs in various contexts and of various nature: bugs in test code (Vahabzadeh et al. 2015), concurrency bugs (Fonseca et al. 2010; Wang et al. 2017; Asadollah et al. 2017), configuration bugs (Yin et al. 2011) performance-related issues (Jin et al. 2012; Han and Yu 2016; Yang et al. 2018), how developers diagnose, debug, and fix bugs (Böhme et al. 2017). Others have examined faults and failures within various domains including machine learning (Thung et al. 2012; Islam et al. 2019; Humbatova et al. 2020), blockchain (Wan et al. 2017), Android (Bhattacharya et al. 2013), and operating systems (Abal et al. 2014; Chou et al. 2001). In this study, we construct and analyze a dataset of bugs in popular ROS software, to provide (a) insight into the nature of software bugs in open-source robotics and (b) a testbed for building and evaluating QA tools and techniques for ROS.

A smaller number of papers have examined faults and failures within robotics systems software. Steinbauer (2012) conducts a quantitative survey of research groups that participated in RoboCup, an annual international robotics competition, to learn more about software, hardware, algorithmic, and human-interaction faults in robotic systems. Grottke et al. (2010) identify and characterize 520 software faults in the on-board software for 18 JPL/NASA space missions in terms of their ease of reproducibility by studying how they are triggered

and their resulting error propagation. Timperley et al. (2018a) perform an empirical study of 228 historical bugs in ArduPilot (ArduPilot 2023), a popular autopilot for a variety of autonomous vehicles, to determine whether those bugs are reproducible in simulation. In comparison to those studies, we restrict out attention to software bugs and curate, examine, and document individual bugs across a diversity of ROS software.

Comparison to Existing Taxonomies Below, we present a comparison of our the results of our study against those obtained from other, recent studies of bugs in similar domains: Garcia et al. (2020) study 499 bugs in two, popular autonomous vehicle software systems, Apollo (Baidu 2021) and Autoware (The Autoware Foundation 2021); Wang et al. (2021) examine 569 bugs across two well-known UAV software platforms: PX4 (Dronecode Foundation 2023) and ArduPilot (ArduPilot 2023); and Zampetti et al. (2022) analyze 655 bugs across 14 projects related to cyberphysical systems, including Autoware, ArduPilot, and PX4. All three approaches construct a dataset by identifying issues (Wang et al. 2021; Zampetti et al. 2022) or merged pull requests (Garcia et al. 2020) in GitHub projects.

In terms of faults, Garcia et al. use an open coding to adapt an existing taxonomy of root causes Thung et al. and Zampetti et al. follow a similar approach to extend Garcia et al.'s taxonomy. After identifying 569 bugs in their subject systems, Wang et al. use open coding to devise a taxonomy for 168 of those bugs (29.5%) that they determine to be UAV-specific.[5]

Table 8 provides a mapping from those existing taxonomies to our own.[6] Despite being specific to UAVs, all of the bugs within each of Wang et al.'s taxonomy fit into one or more of our categories. Most of Zampetti et al.'s taxonomy can be mapped onto our own with the exception of "Documentation", which we consider to be out of scope, "Network", which we consider as a software-level failure caused by an underlying fault (e.g., general programming), and parts of their "Hardware" category (e.g., energy, hardware failure) since we restrict our study to software issues.

Table 9 provides a mapping from our taxonomy to theirs. None of the taxonomies identify "Evolution" as a fault, nor do they fully capture our "Models" category. While ArduPilot, Autoware and PX4 are mostly self contained, our subjects exhibit the norms of ROS development and represent either (a) reusable components that are used to build a larger system (e.g., Geometry2, MAVROS) or (b) robot systems that are built from third-party components (e.g., Care-o-Bot, Kobuki, TurtleBot). ROS's component-based architecture and emphasis on reusability with minimal assumptions makes ROS software more prone to "Evolution" issues. Similarly, as components are often designed to be agnostic to the specifics of any particular robot (e.g., hardware, environment, application), inconsistencies are relatively easy to introduce but difficult to check.

In terms of failures, neither Wang et al. nor Zampetti et al. study the failures associated with the bugs within their datasets. Garcia et al. derive a new taxonomy of bug symptoms (i.e., failures) through of a combination of examining previous taxonomies of bugs in machine learning (Thung et al. 2012), deep learning (Zhang et al. 2018; Islam et al. 2019), and numerical software (Di Franco et al. 2017) and following an open coding approach on their dataset. They make no explicit distinction between software and system-level failures, but both levels appear within their taxonomy. In terms of software-level failures, our "Performance" and

[5] Note that they make a similar observation to our own study (see Section 5) that the majority of bugs that they studied were not specific to UAVs.

[6] We do not compare directly to Garcia et al. (2020) since Zampetti et al. (2022)'s taxonomy subsumes it.

Table 8 A comparison of the root causes presented in Wang et al. (2021) and Zampetti et al. (2022) against our fault taxonomy

Zampetti et al.	Ours	Wang et al.	Ours
Algorithm	● Programming	Limit	● Config, Models
Config	● BDO, Config	Math	● Programming
Data	● Programming	Inconsistency	● Models, Systems
Documentation	○	Priority	● Concurrency
Hardware	◐ Systems	Parameter	● Config
Interface	● Programming	H/W Support	● Systems
Network	○	Correction	● Models, Programming
		Initialization	● Programming

● indicates that all bugs in their category fall into one or more of our categories. ◐ indicates that some of the bugs in their category fall into one or more of our categories. ○ indicates that none of the bugs in their category are represented by any of our categories

"Network" categories are not covered. Garcia et al.'s system-level failures are focused on the outputs of AV subsystems (e.g., incorrect trajectory prediction, localization, lane positioning and navigation). In constrast, our system-level failures provide a different perspective and are focused on *how* the failure manifests (e.g., loss of functionality, unresponsiveness, degraded functionality) as opposed to *where* it manifests.

In comparison to the prior studies, our dataset is application and architecture agnostic, and we examine a diverse set of reusable libraries, components, and platforms that are used to build robot applications. Our results highlight interesting differences in the bugs that affect ROS software and provide further support for the findings of previous studies by performing a conceptual replication (i.e., devising a taxonomy *de novo*). Crucially, we uniquely provide detailed reports for every bug within our dataset (c.f. a URL and a list of labels), along with Docker images and associated tooling, all of which allow researchers to study robotics bugs in more depth and to use our dataset as a benchmark for assessing QA techniques.

Table 9 A comparison of the coverage of existing taxonomies vs. our own

Ours	Zampetti et al.	Wang et al.
BDO	● Config	○
Config	● Config	◐ Limit, Parameter
Concurrency	● Concurrency	◐ Priority
Evolution	○	○
Programming	◐ Algorithm, Data, Interface	◐ Correction, Initialization, Limit, Math
Models	○	◐ Correction
Systems	◐ Hardware	◐ H/W Support

● indicates that our label is fully covered by a set of their labels. ◐ indicates partial coverage, and ○ indicates no coverage

8 Conclusion

In this paper, we presented ROBUST, a study and accompanying dataset of 221 bugs in Robot Operating System software. We systematically collected, documented, and analyzed bugs across 7 popular ROS projects, and produced Docker images that allow researchers building QA tools and techniques for robotics to interact those bugs. We classified faults and failures within a taxonomy we constructed for this purpose, based on a qualitative analysis of our dataset, highlighted findings of particular interest to the software engineering research community, and discussed the ramifications of our results. The ROBUST dataset and more information about the project can be found at our companion website: https://github.com/robust-rosin/robust.

Acknowledgements We thank Claus Brabrand, Zhoulai Fu, Jon Azpiazu, Jon Tjerngren, Jonathan Hechtbauer, Jam Marcos Hernandez, and Claire Le Goues for discussions about the study design, motivation, and for contributing bug analyses or technical knowledge in the process. This work was partially supported by the European Union's Horizon 2020 research and innovation programme, grant agreement No. 73228 (ROSIN), Marie Skłodowska-Curie Actions grant agreement No 956200 (REMARO), by DARPA (#A8750-16-2-0042), and by AFRL (#FA8750-15-2-0075). The authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government or the European Union.

Funding Open Access funding provided by Carnegie Mellon University. This work was partially supported by the ROSIN project under the European Union's Horizon 2020 research and innovation programme, grant agreement No. 73228, and by DARPA (#A8750-16-2-0042) and AFRL (#FA8750-15-2-0075). The authors are grateful for their support. Any opinions, findings, or recommendations expressed are those of the authors and do not necessarily reflect those of the US Government or the European Union.

Availability of data and material The ROBUST dataset together with the Jupyter notebook used to produce figures shown in this paper, is publicly available at the following location: https://github.com/robust-rosin/robust.

Code availability Not applicable.

Declarations

Conflicts of interest The authors have no competing interests to declare that are relevant to the content of this article.

References

Abal I, Brabrand C, Wąsowski A (2014) 42 Variability Bugs in the Linux Kernel: A Qualitative Analysis. In: International conference on automated software engineering, ASE '14, pp 421–432

Abal I, Melo J, Stănciulescu c, Brabrand C, Ribeiro M, Wąsowski A (2018) Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. ACM Trans Softw Eng Methodol 26(3)

Afzal A, Le Goues C, Hilton M, Timperley CS (2020) A study on challenges of testing robotic systems. In: International conference on software testing, validation and verification, ICST '20, pp 96–107
Afzal A, Le Goues C, Timperley CS (2021a) GzScenic: Automatic Scene Generation for Gazebo Simulator. 2104.08625
Afzal A, Le Goues C, Timperley CS (2021) Mithra: Anomaly detection as an oracle for cyberphysical systems. IEEE Trans Softw Eng 1–1. https://doi.org/10.1109/TSE.2021.3120680
Alami A, Dittrich Y, Wasowski A (2018) Influencers of quality assurance in an open source community. In: International workshop on cooperative and human aspects of software engineering, CHASE '18, pp 61–68
Aliabadi MR, Kamath AA, Gascon-Samson J, Pattabiraman K (2017) ARTINALI: dynamic invariant detection for cyber-physical system security. In: Joint meeting on European software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '17, pp 349–361
ArduPilot (2023) ArduPilot. https://ardupilot.org
Asadollah SA, Sundmark D, Eldh S, Hansson H (2017) Concurrency bugs in open source software: a case study. J Internet Serv Appl 8(1):1–15
Atwood J (2012) New programming jargon. https://blog.codinghorror.com/new-programming-jargon
Baidu (2021) Apollo. https://github.com/ApolloAuto/apollo
Bardaro G, Semprebon A, Matteucci M (2018) A Use Case in Model-Based Robot Development Using AADL and ROS. In: International workshop on robotics software engineering, RoSE '18, pp 9–16
Benton S, Ghanbari A, Zhang L (2019) Defexts: A curated dataset of reproducible real-world bugs for modern jvm languages. In: International conference on software engineering: companion proceedings, ICSE '19, pp 47–50
Bhattacharya P, Ulanova L, Neamtiu I, Koduru SC (2013) An empirical analysis of bug reports and bug fixing in open source Android apps. In: European conference on software maintenance and reengineering, IEEE, pp 133–143
Böhme M, Roychoudhury A (2014) Corebench: Studying complexity of regression errors. In: International symposium on software testing and analysis, ISSTA '14, pp 105–115
Böhme M, Soremekun EO, Chattopadhyay S, Ugherughe E, Zeller A (2017) Where is the bug and how is it fixed? an experiment with practitioners. In: Joint meeting of the european software engineering conference and the symposium on the foundations of software engineering, ESEC/FSE '17, pp 1–11
Bozhinoski D, Aguado E, Oviedo MG, Hernandez C, Sanz R, Wąsowski A (2021) A Modeling Tool for Reconfigurable Skills in ROS. In: International workshop on robotics software engineering, RoSE '21, pp 25–28
Charette RN (2014) Nissan recalls nearly 1 million cars for air bag software fix. IEEE Spectrum. https://spectrum.ieee.org/riskfactor/transportation/safety/nissan-recalls-nearly-1-million-cars-for-airbag-software-fix
Chou A, Yang J, Chelf B, Hallem S, Engler D (2001) An empirical study of operating systems errors. In: Symposium on operating systems principles, pp 73–88
Coghlan N, Stufft D (2013) PEP 440 – Version Identification and Dependency Specification. https://www.python.org/dev/peps/pep-0440. Accessed 16 June 2021
Di Franco A, Guo H, Rubio-González C (2017) A comprehensive study of real-world numerical bug characteristics. In: International conference on automated software engineering, ASE '17, pp 509–519
Do H, Elbaum S, Rothermel G (2005) Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. Empir Softw Eng 10(4):405–435
Dronecode Foundation (2023) PX4. https://px4.io
Eugster PT, Felber PA, Guerraoui R, Kermarrec AM (2003) The many faces of publish/subscribe. ACM Comput Surv 35(2):114–131
Facebook (2022) Infer. https://fbinfer.com. Accessed 22 July 2022
Fischer-Nielsen A, Fu Z, Su T, Wąsowski A (2020) The forgotten case of the dependency bugs: on the example of the Robot Operating System. In: International conference on software engineering, software engineering in practice, ICSE-SEIP '20, pp 21–30
Fonseca P, Li C, Singhal V, Rodrigues R (2010) A study of the internal and external effects of concurrency bugs. In: International conference on dependable systems & networks, DSN '10, pp 221–230
Foote T (2021) Parameter api design in ros. http://design.ros2.org/articles/ros_parameters.html. Accessed 26 May 2021
Fremont DJ, Dreossi T, Ghosh S, Yue X, Sangiovanni-Vincentelli AL, Seshia SA (2019) Scenic: a language for scenario specification and scene generation. In: Conference on programming language design and implementation, PLDI '19, pp 63–78
Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen QA (2020) A comprehensive study of autonomous vehicle bugs. In: International conference on software engineering, pp 385–396

Grottke M, Nikora AP, Trivedi KS (2010) An empirical investigation of fault types in space mission system software. In: International conference on dependable systems & networks, DSN '10, pp 447–456
Gyimesi P, Vancsics B, Stocco A, Mazinanian D, Beszédes A, Ferenc R, Mesbah A (2019) BugsJS: a benchmark of JavaScript bugs. In: International conference on software testing, validation and verification, ICST '19, pp 90–101
Han X, Yu T (2016) An empirical study on performance bugs for highly configurable software systems. In: International symposium on empirical software engineering and measurement, ESEM '16, pp 1–10
He Z, Chen Y, Huang E, Wang Q, Pei Y, Yuan H (2019) A system identification based oracle for control-cps software fault localization. In: International conference on software engineering, ICSE '19, pp 116–127
Henning A (2016) Physical continuous integration — CI on real robots! In: ROSCon Seoul 2016, Open Robotics. https://doi.org/10.36288/ROSCon2016-900758. https://doi.org/10.36288/ROSCon2016-900758
Hu Y, Ahmed UZ, Mechtaev S, Leong B, Roychoudhury A (2019) Re-factoring based Program Repair applied to Programming Assignments. In: International conference on automated software engineering, ASE '19, pp 388–398
Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: International conference on software engineering, ICSE '20, pp 1110–1121
Inoue J, Yamagata Y, Chen Y, Poskitt CM, Sun J (2017) Anomaly detection for a water treatment system using unsupervised machine learning. In: International conference on data mining workshops, ICDMW '17, pp 1058–1065
Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: Joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '19, pp 510–520
Jin G, Song L, Shi X, Scherpelz J, Lu S (2012) Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices 47(6):77–88
Just R, Jalali D, Ernst MD (2014) Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: International symposium on software testing and analysis, ISSTA '14, pp 437–440
Kate S, Ore JP, Zhang X, Elbaum S, Xu Z (2018) Phys: probabilistic physical unit assignment and inconsistency detection. In: Joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '18, pp 563–573
Kate S, Chinn M, Choi H, Zhang X, Elbaum S (2021) PHYSFRAME: Type Checking Physical Frames of Reference for Robotic Systems. In: Joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '21, pp 45–56
Kim T, Kim CH, Rhee J, Fei F, Tu Z, Walkup G, Zhang X, Deng X, Xu D (2019) RVFUZZER: Finding input validation bugs in robotic vehicles through control-guided testing. In: USENIX Security Symposium, pp 425-442
Klück F, Li Y, Nica M, Tao J, Wotawa F (2018) Using ontologies for test suites generation for automated and autonomous driving functions. In: International Symposium on Software Reliability Engineering Workshops, ISSREW '18, pp 118–123
Kolak S, Afzal A, Le Goues C, Hilton MC, Timperley CS (2020) It takes a village to build a robot: An empirical study of the ros ecosystem. Int Conf Softw Maint Evol 430–440
Larsen H, Hoorn Gvd, Wąsowski A (2021) Reactive Programming of Robots with RxROS. In: Robot Operating System (ROS), Springer, pp 55–83
Le Goues C, Holtschulte N, Smith EK, Brun Y, Devanbu P, Forrest S, Weimer W (2015) The ManyBugs and IntroClass benchmarks for automated repair of C programs. Trans Softw Eng 41(12):1236–1256
Lin D, Koppel J, Chen A, Solar-Lezama A (2017) QuixBugs: A multi-lingual program repair benchmark set based on the Quixey Challenge. In: International conference on systems, programming, languages, and applications: software for humanity, pp 55–56
Linneberg MS, Korsgaard S (2019) Coding qualitative data: A synthesis guiding the novice. Qual Res J
Lämmel R (2018) Software languages: Syntax, semantics, and metaprogramming. Springer
Madeiral F, Urli S, Maia M, Monperrus M (2019) Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In: International Conference on Software Analysis, Evolution and Reengineering, SANER '19, pp 468–478
Majumdar R, Mathur A, Pirron M, Stegner L, Zufferey D (2019) Paracosm: A language and tool for testing autonomous driving systems. arXiv:1902.01084
Martinez J, Ruiz A, Garzo A, Keller T, Radermacher A, Tonetta S (2021) Modelling the Component-based Architecture and Safety Contracts of ArmAssist in Papyrus for Robotics. In: International Workshop on Robotics Software Engineering, RoSE '21, pp 13–18
Matsakis ND, Klock FS II (2014) The Rust language. ACM SIGAda Ada Letters 34:103–104

McCausland P (2019) Self-driving uber car that hit and killed woman did not recognize that pedestrians jaywalk. NBC News. https://www.nbcnews.com/tech/tech-news/self-driving-uber-car-hit-killed-woman-did-not-recognize-n1079281
O'Kane S (2020) Boeing finds another software problem on the 737 max. The Verge. https://www.theverge.com/2020/2/6/21126364/boeing-737-max-software-glitch-flaw-problem
Ore JP, Detweiler C, Elbaum S (2017) Phriky-units: a lightweight, annotationfree physical unit inconsistency detection tool. In: International Symposium on Software Testing and Analysis, ISSTA '17, pp 352–355
Purandare R, Darsie J, Elbaum S, Dwyer MB (2012) Extracting conditional component dependence for distributed robotic systems. In: International Conference on Intelligent Robots and Systems, RSJ '12, pp 1533–1540
Quigley M, Conley K, Gerkey BP, Faust J, Foote T, Leibs J, Wheeler R, Ng AY (2009) Ros: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software
ROS2 (2021) rclcpp: C++ ros client library api: Node class reference. https://docs.ros2.org/latest/api/rclcpp/classrclcpp_1_1Node.html#a12d535bced9f26b65c0a450e6f40aff8. Accessed 26 May 2021
ROSIn (2022) ROS-Industrial: Current Members. https://web.archive.org/web/20220723013322/https://rosindustrial.org/ric/current-members. Accessed 28 July 2022
Saha RK, Lyu Y, Lam W, Yoshida H, Prasad MR (2018) Bugs.jar: a largescale, diverse dataset of real-world java bugs. In: International Conference on Mining Software Repositories, MSR '18, pp 10–13
Saldaña J (2015) The Coding Manual for Qualitative Researchers. Sage, Thousand Oaks, CA
Santos A, Cunha A, Macedo N, Lourenço C (2016) A framework for quality assessment of ROS repositories. In: International conference on intelligent robots and systems, IROS '16, pp 4491–4496
Santos A, Cunha A, Macedo N (2019) Static-time extraction and analysis of the ROS computation graph. In: International Conference on Robotic Computing, IRC '19, pp 62–69
Santos A, Cunha A, Macedo N (2021) The high-assurance ROS framework. In: International Workshop on Robotics Software Engineering, RoSE '21, p In press
Santos A, Cunha A, Macedo N (2022) Schema-guided testing of message-oriented systems. In: International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE), SCITEPRESS, pp 26–37. https://doi.org/10.5220/0010976100003176
Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008) Defect categorization: Making use of a decade of widely varying historical data. In: International Symposium on Empirical Software Engineering and Measurement, ESEM '08, pp 149–157
Serebryany K, Iskhodzhanov T (2009) Threadsanitizer: data race detection in practice. In: Workshop on Binary Instrumentation and Applications, pp 62–71
Serebryany K, Bruening D, Potapenko A, Vyukov D (2012) AddressSanitizer: A fast address sanity checker. In: USENIX Annual Technical Conference, pp 309–318
Shenton A (2004) Strategies for ensuring trustworthiness in qualitative research projects. Educ Inf 22:63–75. https://doi.org/10.3233/EFI-2004-22201
Sikolia D, Biros DP, Mason M, Weiser M (2013) Trustworthiness of grounded theory methodology research in information systems. MWAIS 2013 Proceedings, vol 16. https://aisel.aisnet.org/mwais2013/16
Steinbauer G (2012) A survey about faults of robots used in RoboCup. In: Robot Soccer World Cup, Springer, pp 344–355
Stepanov E, Serebryany K (2015) MemorySanitizer: fast detector of uninitialized memory use in C++. In: International Symposium on Code Generation and Optimization, pp 46–55
Tan SH, Yi J, Yulis, Mechtaev S, Roychoudhury A (2017) Codeflaws: A Programming Competition Benchmark for Evaluating Automated Program Repair Tools. In: International Conference on Software Engineering, ICSE '17, pp 180–182
Tan SH, Dong Z, Gao X, Roychoudhury A (2018) Repairing crashes in android apps. In: International Conference on Software Engineering, ICSE '18, pp 187–198
The Autoware Foundation (2021) Autoware.ai. https://github.com/Autoware-AI/autoware.ai
The Robotics Back-End (2021) rclcpp Params Tutorial - Get and Set ROS2 Params with Cpp. https://roboticsbackend.com/rclcpp-params-tutorial-get-set-ros2-params-with-cpp/#rclcpp_parameter_callback. Accessed 26 May 2021
Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: International Symposium on Software Reliability Engineering, ISSRE '12, pp 271–280
Timperley C, Wasowski A (2019) 188 ros bugs later: Where do we go from here? In: ROSCon Macau 2019, Open Robotics. https://doi.org/10.36288/ROSCon2019-900898. https://doi.org/10.36288/ROSCon2019-900898
Timperley CS, Afzal A, Katz DS, Hernandez JM, Le Goues C (2018a) Crashing simulated planes is cheap: Can simulation detect robotics bugs early? In: International Conference on Software Testing, Verification and Validation, IEEE, ICST '18, pp 331–342

Timperley CS, Stepney S, Le Goues C (2018b) Bugzoo: a platform for studying software bugs. In: International Conference on Software Engineering: Companion Proceeedings, ICST '18, pp 446–447
Timperley CS, Dürschmid T, Schmerl B, Garlan D, Le Goues C (2022) ROSDiscover: Statically Detecting Run-Time Architecture Misconfigurations in Robotics Systems. In: International Conference on Software Architecture, ICSA '22, pp 112–123
Tomassi DA, Dmeiri N, Wang Y, Bhowmick A, Liu YC, Devanbu PT, Vasilescu B, Rubio-González C (2019) Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In: International Conference on Software Engineering, ICSE '19, pp 339–349
Vahabzadeh A, Fard AM, Mesbah A (2015) An empirical study of bugs in test code. In: International Conference on Software Maintenance and Evolution, ICSME '15, pp 101–110
van Tonder R, Le Goues C (2018) Static automated program repair for heap properties. In: International Conference on Software Engineering, pp 151–162
Virding R, Wikström C, Williams M (1996) Concurrent programming in ERLANG. Prentice Hall International (UK) Ltd
Wall M (2017) European mars lander crashed due to data glitch, esa concludes. Space. https://www.space.com/37015-schiaparelli-mars-lander-crash-investigation-complete.html
Wan Z, Lo D, Xia X, Cai L (2017) Bug characteristics in blockchain systems: a large-scale empirical study. In: International Conference on Mining Software Repositories (MSR), MSR '17, pp 413–424
Wang D, Li S, Xiao G, Liu Y, Sui Y (2021) An Exploratory Study of Autopilot Software Bugs in Unmanned Aerial Vehicles. In: Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '21, pp 20–31
Wang J, Dou W, Gao Y, Gao C, Qin F, Yin K, Wei J (2017) A comprehensive study on real world concurrency bugs in node.js. In: International conference on automated software engineering, ASE '17, pp 520–531
White R (2019) Announcing Official Docker Images for ROS2. https://discourse.ros.org/t/announcing-official-docker-images-for-ros2/7381. Accessed 9 June 2021
Widyasari R, Sim SQ, Lok C, Qi H, Phan J, Tay Q, Tan C, Wee F, Tan JE, Yieh Y, Goh B, Thung F, Kang HJ, Hoang T, Lo D, Ouh EL (2020) BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies. In: Joint meeting on european software engineering conference and symposium on the foundations of software engineering, ESEC/FSE '20, pp 1556–1560
Witte T, Tichy M (2018) Checking consistency of robot software architectures in ros. In: International workshop on robotics software engineering, RoSE '18, pp 1–8
Wyrobek K (2017) The Origin Story of ROS, the Linux of Robotics. IEEE Spectrum. https://spectrum.ieee.org/the-origin-story-of-ros-the-linux-of-robotics
Wąsowski A, Berger T (2023) Domain-Specific Languages: Effective Modeling, Automation, and Reuse. Springer
Yang J, Yan C, Subramaniam P, Lu S, Cheung A (2018) How not to structure your database-backed web applications: a study of performance bugs in the wild. In: International conference on software engineering, ICSE '18, pp 800–810
Yin Z, Ma X, Zheng J, Zhou Y, Bairavasundaram LN, Pasupathy S (2011) An empirical study on configuration errors in commercial and open source systems. In: Symposium on operating systems principles, pp 159–172
Zampetti F, Kapur R, Di Penta M, Panichella S (2022) An empirical characterization of software bugs in open-source cyber-physical systems. J Syst Softw 192
Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on tensorflow program bugs. In: International symposium on software testing and analysis, ISSTA '18, pp 129–140