Check for updates

Research article

INTERNATIONAL JOURNAL OF ADVANCED ROBOTIC SYSTEMS

International Journal of Advanced Robotic Systems
July–August 2025: 1–21
© The Author(s) 2025
Article reuse guidelines:
sagepub.com/journals-permissions
DOI: 10.1177/17298806251363648
journals.sagepub.com/home/arx

Sage

Evaluation of a model-driven approach for the integration of robot operating system-based complex robot systems

Nadia Hammoudeh García[1], Yuzhang Chen[1], David Lieb[1] and Andreas Wortmann[2]

Abstract

In system development, integration is crucial—especially in domains like robotics, where the complexity of the applications makes it a particularly challenging task. The rapid growth in the sector can lead to the obsolescence of existing tools and procedures. Although other technologies have emerged in scientific research, they face limited acceptance within the robotics community due to high entry barriers and a lack of evaluative studies, with real use cases, demonstrating their effectiveness. In this effort, we have used RosTooling, a model-driven tool to facilitate the integration of robot systems, as a target and have conducted several evaluative studies, both quantitative and qualitative. The conclusions of these studies allow us to identify the types of systems and lifecycle stages where these tools are most effective. Additionally, we have conducted expert interviews to provide insights into how to foster their acceptance and improve usability.

Keywords

Robotics, model-driven development, evaluation

Received: 17 January 2025; accepted: 1 July 2025

Topic: Robotics Software Design and Engineering
Topic Editor: Sunita Bansal
Associate Editor: Sunita Bansal

Introduction

Model-driven development (MDD) has proven its efficiency in the integration of systems in different domains. Several comparative studies have shown that MDD significantly reduces development time, with some experiments[1] even concluding that a model-based approach could be executed in only 11% of the time required by code-centric methods. However, in robotics software development, there is a prevailing preference for a hand-written approach. To demonstrate how MDD can be used in a code-centric domain,[2] such as robotics, by reusing existing code we have created a set of tools, called RosTooling that thanks to reverse engineering processes are able to combine hand-written code with model-driven techniques.

This paper focuses on the evaluation of RosTooling (https://github.com/ipa320/RosTooling). In order to illustrate and assess its effectiveness, we conducted several experiments on robots and real applications that serve as demonstration exercises exemplifying the use and the performance benefits of our technical approach. In all case studies, the objective was the integration of complete robotic applications from existing packages from the robot operating system (ROS) ecosystem. In the first instance, the starting point is a collection of scenarios that the robot solution must fulfill, targeting two essential applications of robotics, a manipulation setup and a mobile service robot platform. For the cases presented here and as a comparative study, the design and implementation phases of the system were performed by using RosTooling and by using a traditional approach in parallel.

[1]Robot and Assistive Systems Department, Fraunhofer Institute for Manufacturing Engineering and Automation (IPA), Stuttgart, Germany
[2]Institute for Control Engineering of Machine Tools and Manufacturing Units (ISW), University of Stuttgart, Stuttgart, Germany

Corresponding author:
Nadia Hammoudeh García, Robot and Assistive Systems Department, Fraunhofer Institute for Manufacturing Engineering and Automation (IPA), Nobelstr. 12, 70569 Stuttgart, Germany.
Email: nadia.hammoudeh.garcia@ipa.fraunhofer.de

Data Availability Statement included at the end of the article

Creative Commons CC BY: This article is distributed under the terms of the Creative Commons Attribution 4.0 License (https://creativecommons.org/licenses/by/4.0/) which permits any use, reproduction and distribution of the work without further permission provided the original work is attributed as specified on the SAGE and Open Access page (https://us.sagepub.com/en-us/nam/open-access-at-sage).

To complement the studies, we also conducted a qualitative study, interviewing experts to establish the usability and ease of use degree of the developed tools.

This paper is structured as follows: Firstly reviews related work on empirical and qualitative evaluations of software solutions, providing a foundation for our study. The next section outlines the background necessary to contextualize this work. Then the core of the publication focus on the quantitative and qualitative evaluations, respectively. Finally, it concludes by summarizing our findings.

Related work

Empirical MDD software evaluation studies

As an example of the comparison between MDD and traditional software development there studies based on the Palladio Editor[3] that show the MDD's benefits in improving design accuracy and system analysis, on the other side limitations like tooling constraints and a steep learning curve are evident. The findings from a survey through practitioners to evaluate MDD's impact[4] indicate that MDD increases productivity and maintainability while reducing errors, particularly in complex systems. The key metrics are development speed and quality, though initial setup complexity and costs are highlighted barriers. Practitioners recommended MDD primarily for large-scale projects, where its advantages are more apparent. A comparative case study[5] points out that the traditional software development processes are faster for the initial progress, while MDD provides better scalability and maintainability. For this empirical study, the considered metrics include time to delivery, scalability, and defect rates. It concluded that MDD offered more sustainable benefits for long-term projects. Another empirical effort[6] analyzed MDD in four industrial case studies with a focus on usability, compatibility, and performance, and using as metrics adoption rates, ease of integration, and the impact on development time and quality. The study concludes that domain-specific contexts significantly influence platform effectiveness. Together, these studies highlight that while MDD may require higher initial investments, it often outperforms traditional development in long-term scalability, maintainability, and quality, particularly in complex, large-scale projects. Also, the domain must be taken into account and unfortunately for robotics, there is no previous work evaluating the impact of an MDD solution.

Methodologies for qualitative evaluations

The goal question metric (GQM)[7] is an approach to measure software quality. It begins by defining clear, goal-oriented objectives, breaking these goals down into specific questions, and then developing metrics to evaluate the results. This structured inquiry ensures that the evaluation is focused and aligned with the overarching objectives of the software engineering approach, covering a broad range of factors such as performance, reliability, maintainability, and usability. In addition, System Usability Scale (SUS)[8] is a standardized tool used to measure the usability of a system, providing a quick and reliable way to assess user satisfaction and usability through a simple 10-item questionnaire. The 10 questions alternates positive and negative questions and the participant must score it using a 5-point Likert scale. SUS is a standardized, widely recognized tool that is both quick and easy to administer. Similarly, Post-Study System Usability Questionnaire (PSSUQ)[9] focuses on three core dimensions: System usefulness, information, and interface quality. Participants complete the PSSUQ after interacting with a system, rating their experience on a 7-point Likert scale.

Background: The RosTooling

The ROS ecosystem encompasses an array of libraries and packages catering to functionalities, such as perception, planning, control, and simulation. Supporting programing languages, like C++ and Python, ROS boasts an active community contributing to its extensive range of resources.

ROS has a widespread impact on the robotics community.[10] With a federative development model, i.e., new packages can be added to the ecosystem without modifying the core, ROS allows easy sharing of components across developers' teams.

The target of this evaluation, RosTooling (https://ipa320.github.io/RosTooling.github.io/),[11] is a specific instance of a domain-specific language (DSL)[12] used to model ROS components. It is built from a collection of open-source tools designed to assist in the creation and development of robotic systems. Its primary focus is on supporting components and systems developed using the ROS framework. RosTooling includes various MDD tools and methods that facilitate the development process and enhance the quality of the resulting systems. This approach brings several benefits to the ROS ecosystem, such as improved system quality, increased productivity, and enhanced maintainability and extensibility.

From a technical perspective, RosTooling enables the production of simple models that encapsulate the core concepts of existing ROS code. It provides two main models: one for defining components (or ROS packages) and another for defining systems. These models can then undergo transformations to evolve into more generalized and complex models.

The main novelty of RosTooling in comparison with other approaches is that it provides reverse engineering tools so that the existing hand-written code can be automatically transformed into models supported by the MDD tools, this takes advantage of re-using the thousands of available software packages available on the ROS

ecosystem. The goal is to enable software engineers to concentrate on core development tasks by leveraging code generators to automate repetitive and framework-related coding. This approach enhances productivity and optimizes the overall software development process.

Developer perspective

The big picture of the of the RosTooling is presented in Figure 1. The core of the backend is a family of DSLs, and their corresponding metamodels. This combination of metamodels, implemented using the eclipse modeling framework, and DSLs enables more robust and extensible solutions. The ROS DSL (Xtext) serves as a foundational module containing both the grammar implementation, defining the language's syntax and structure, and a validator to enforce correctness rules for language constructs. Meanwhile, the ROSSYSTEM DSL (Xtext) module represents a system-level DSL for describing entire ROS-based systems. It includes its own grammar, validator, and generator implementations and depends on both the ROS 1 DSL and ROS 2 DSL, referencing elements from both ROS versions. For the implementation, the RosTooling uses Xtext,[13] a popular framework that eases the implementation of programing languages and even supports the development of tools like editors, syntax highlighting, and code completion. To reuse all the existing code, the RosTooling integrates an API implemented in Python to connect the Xtext implementation with static code analysis frameworks, like HAROS[14] or introspection at runtime tools.

The RosTooling enables a federative development, by the easy extension through plugins without altering the core itself. Plugin types include:

- Code generator extensions: Simplest and most useful, these add new file generators to the compiler.
- Validation extensions: Add new compiler validation rules via plugins.
- Extraction and conversion extensions: Provide reusable mechanisms (like a Python API and model-to-model techniques with ECORE) to generate and map models.
- Metamodels and DSLs extensions: Allow adding and referencing new metamodels and DSLs.

User perspective

From the user's perspective, there are two key aspects of the RosTooling: The integrated development environment (IDE) based on Eclipse, provided for user interaction with the tooling, and the support it offers throughout the system development lifecycle. The frontend IDE plays a crucial role in making these capabilities accessible. Some frontend modules are available to lower the entry barrier for end-users, as shown in Figure 1. Firstly, textual editors based on Xtext that offer features like syntax highlighting, code completion, and validation. Secondly, a graphical editor using Sirius is integrated, making system modeling more intuitive as shown in Figure 2, although it has limited support for editing model properties. The backend is designed to be extensible beyond Eclipse, so it allows compatibility for Theia, a modern cloud and desktop IDE. Other visualization tools are also provided, enabling users to view system components and their relationships either through a graphical representation in Eclipse or via automatically generated PlantUML diagrams. Lastly, acknowledging the preferences of the ROS developer community, which often favors minimal tooling, Python-based utilities allow model parsing, generation, and interaction directly with ROS environments.

Limitations

The main limitations of this kind of approach, and specifically of the RosTooling are:[2]

- Ambiguity and lack of structure. ROS lacks well-defined architectural abstractions like components, modules, or subsystems. This makes it hard to define meaningful, reusable models. Also, the bottom-up modeling approach results in complex, hard-to-read system models that require specialized graphical tooling to manage.
- Limitations of model extraction technologies. The static code analysis limitations due to the high variability in ROS coding styles lead to incomplete model extraction (66% coverage). While runtime extraction is unreliable, as it can't capture all needed data (e.g., service calls, file-level parameters), it may be risky or incomplete.
- Inconsistency between design-time and runtime information due to dynamic naming and the set of interfaces. RosTooling is a tool for the design of the system; however, in ROS, topics or services are often dynamically managed during runtime.

Also, it must be taken into account that every bottom-up MDD approach presents several inherent limitations. This type of approach typically exhibits weak or missing support for version control systems, which complicates collaborative development and model evolution. Additionally, there is a steep learning curve for developers, especially within the ROS community, which is accustomed to rapid prototyping using familiar programing languages. The introduction of new tools can thus raise the entry barrier and slow down adoption.

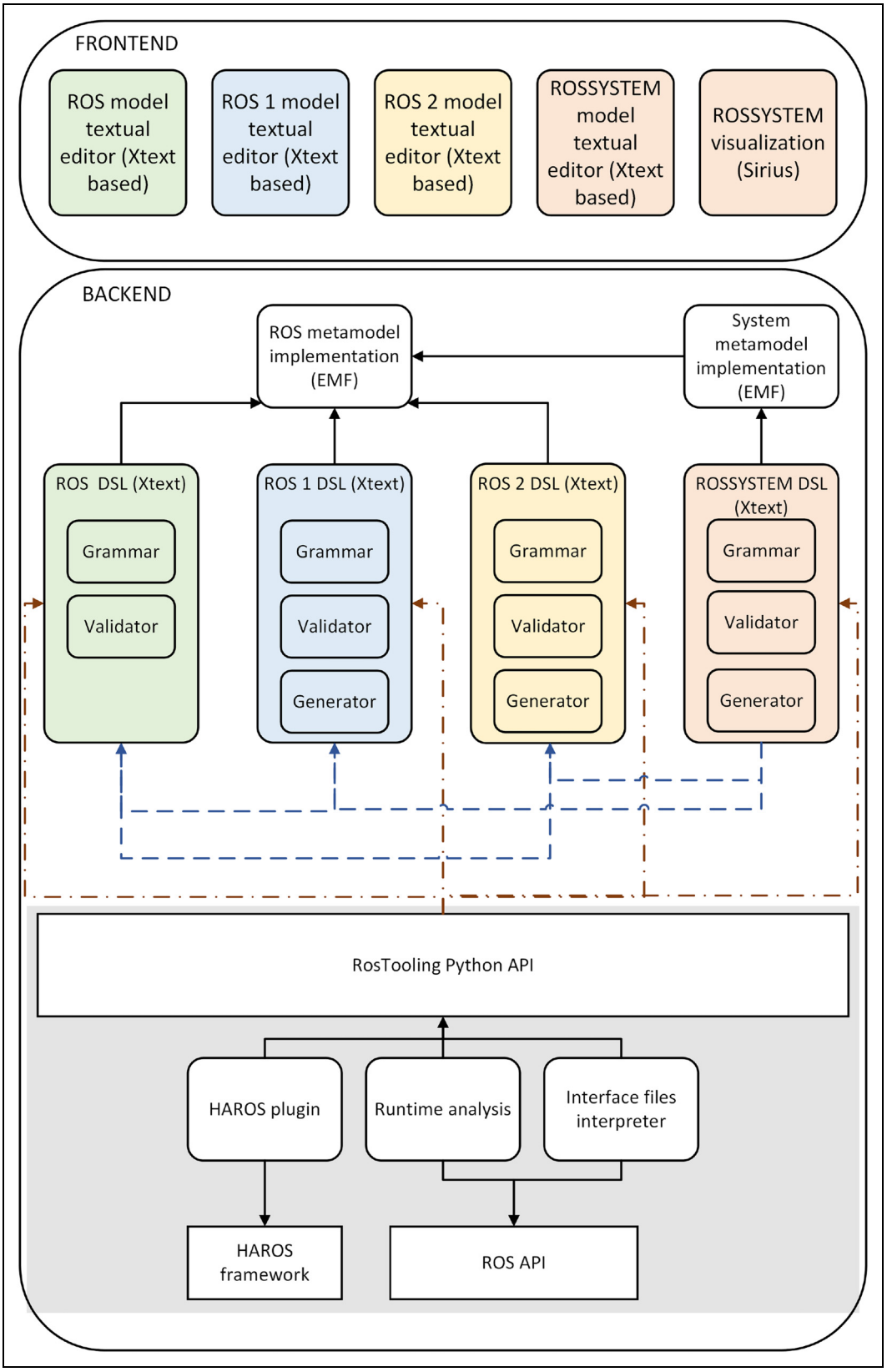

Figure 1. Diagram representing the RosTooling modules.

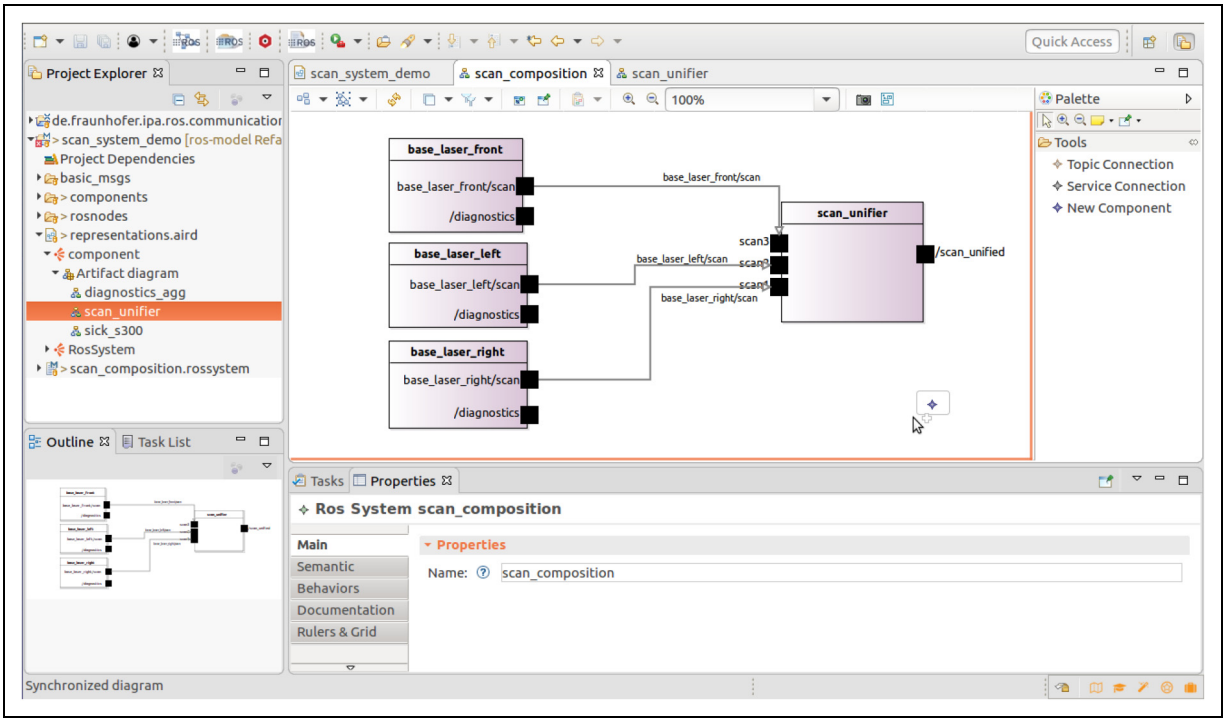

Figure 2. Screenshot from the RosTooling of the Sirius tool implementation for the system architecture development.

Table 1. List of user stories for the manipulation use case, the number of subsystems (S), the total number of components (C), and parameters (P).

User Story ID	Description	S	C	P
MANI-01 URe	Sample movements	1	19	0
MANI-01 PRBT	Sample movements	1	18	0
MANI-02 URe	QR detection and decision	1	28	56
MANI-02 PRBT	QR detection and decision	1	27	50
MANI-03 URe	Pick and place ball	1	23	32
MANI-03 PRBT	Pick and place ball	1	22	32
MANI-04 URe	Pick box from shelf with GUI	1	23	32
MANI-04 PRBT	Pick box from shelf with GUI	1	22	32

Quantitative evaluation

We conducted 14 experiments on real robots Tables 1 and 2. In all case studies the objective was the integration of complete robotic applications from existing ROS packages starting with the description of a collection of user stories to fulfill.

For the cases presented here and as a comparative study, the design and implementation phases of the system were performed by using RosTooling and by using a traditional approach of the ROS community, to collect then data to be compared.

The experiments were carried out by people outside of the RosTooling development team. The manipulation case was developed by a software engineering student, while the mobile assistant robot was developed by a mechanical engineering student. Both with full-time dedication for 6 months, and had the freedom to choose the setup of the experiments and the architecture of the system. Also, they had no previous experience with either ROS or MDD tools.

Study design

The following four evaluation criteria,[15,16] and their respective metrics, have been carried out for the evaluation.

Development time. Its corresponding metric is the total hours actively working on development tasks. Hours are categorized by development phase (design, implementation, testing, and deployment) to reveal the phase-specific impacts of each approach. The phases are defined as:

- Design: The system architecture and detailed component designs are defined, including software choices and interface design. In our study, for both cases, we already have the set of components to be integrated; for the case of the traditional approach, the list of repositories that contain the code of the different components, and for modeling, thanks to the extraction tools, the models are obtained automatically.
- Implementation: Involves building and integrating the components of the system by developing any

Table 2. List of user stories for the mobile assistant robot use case, the number of subsystems (S), the total number of components (C) and parameters (P).

User story ID	Description	S	C	P
MOBI-01 COB	Sample movements using teleoperation	1	79	19
MOBI-01 SIM	Sample movements using teleoperation	1	14	21
MOBI-02 COB	SLAM and self-navigation	3	98	71
MOBI-02 SIM	SLAM and self-navigation	2	30	130
MOBI-03 COB	Medicine delivery robot through hospitals	3	105	75
MOBI-03 SIM	Medicine delivery robot through hospitals	2	32	134

Figure 3. Picture of both robot cells together performing the same task. Universal Robot UR5e on the left side and the Pilz PRBT on the right side.

necessary software. For the MDD solution, thanks to code generators, this is partly done automatically.

- Testing: The system is tested to ensure that it meets its requirements and functions as expected, especially important for our experiments is the integration testing of the complete system. Testing for our experiments consider also the debugging and fixing issues time, so we can then compare the advantages of one approach over the other.
- Deployment: The system is installed and made available for use by its intended users.

Additionally, time is recorded for each user story individually. All the data have been collected by calendar working day in a detailed Excel sheet.

Code generation efficiency. Code generation efficiency measures how effectively a development approach produces functional code with minimal resources, an important factor in evaluating MDD methods.[17] Frameworks that require less code to describe a system are more efficient than those needing extensive custom code.

The metric in this case is the Lines of Code (LOC). Although fewer lines can improve maintainability, LOC alone does not reflect code quality or functionality. Instead, this metric focuses on the quantity of code generated, not its quality. A tool named cloc is used, to count the LOC. It contains language support for a wide range of programing languages and therefore is able to detect blank lines and commented-out lines. In the evaluation we only considered the lines that contain actual code.

Error traceability. Error traceability refers to the ability to identify, trace, and link errors or defects in the software back to their root cause.

To measure the error traceability, the selected metric is the resolution time as the average time to fix a runtime error.[18] Once a system is operational and a misbehavior is detected, error traceability refers to the ability to identify the source of the error. To support this process, a

standardized form is used to document each error. This form categorizes errors into four main types: misspellings, incorrect system behavior, misconfigured parameters, and interface mismatches. Additionally, the time taken to resolve each error is recorded. This approach enables tracking of the effort required to address issues in terms of the working time invested.

Portability. Portability is the ability to run the same piece of software on different hardware.

To measure portability, we analyze the relationship between the *cost* or *effort* of porting software and rewriting individual components.[19]

$$M_{port}(p) = 1 - \frac{C_{port}(p)}{C_{new}(p)} \quad (1)$$

Where $M_{port}(p)$ measures the portability of the process p, where $C_{port}(p)$ is the cost to modify p for another platform, and $C_{new}(p)$ is the cost to rewrite it for a new platform.

In our case, Equation 1 sums the number of lines for all user stories, showing the average portability value among them. The process definition p is the sum of all experiments within either the traditional ROS approach or the MDD approach.

Use cases

Manipulation use case. For implementation, two different robotic cells were used, as shown in Figure 3. Both cells contain a robotic arm with a gripper mounted at the flange position. In addition, a RealSense 3D camera is mounted to perform perception tasks. They only differ on the robot arm manipulator which comes from different vendors, using on one hand a Universal Robot UR5e robot and on the other hand, a PILZ PRBT.

In this study, we tackle the topic of portability and how the use of models can streamline the process. Selecting manipulation as a use case is also a great challenge since it requires a high degree of configuration and parameterization. In order to have a better collection of data to guide us to a conclusion, we have defined different user stories, ranged from simple to complex.

In total, we conducted the development of four different user stories (Full description of the user stories attached as supplemental material), all of which were built for the two target hardware, the Universal Robot UR5e arm and the Pilz PRBT, which means a total of eight different application designs. For all the cases the robot low-level drivers, together with the path planning software are implemented as a subsystem that will be re-used among the different cases. To obtain the models corresponding to the component drivers, all of them already available on ROS 2, static code analysis tools have been used as a reverse engineering method. Concretely, we made use of the framework HAROS.[14] This framework involves extracting information from the source code without executing it. Thanks to it, from code written in C++ or Python, we are able to automatically extract its related model compatible with the RosTooling.

For the case of the UR5e robot, the subsystem is composed of 18 components, and 17 for the PRBT, in both cases this subsystem cover the low-level functionalities. Table 1 details the number of extra components added for every application and the definition of the parameters.

Figure 4 shows a simplified view of the implementation of the user story MANI-02 as it is sketched by the RosTooling.

```
# .rossystem file for the MANI-02 user story with
    the the Universal Robot UR5e arm
qr_detection_and_decision_of_picking_robot:
subSystems:
  ur_robot
nodes:
  realsense_camera_driver:
    from: "realsense2_camera.camera"
    interfaces:
      - "/camera_info": pub-> "
          realsense2_camera_node::color/
          camera_info"
      - "/image_raw": pub-> "
          realsense2_camera_node::color/
          image_raw"
  ....
  bt_skills_client:
    from: "man2_bt_skill_clients.bt_skill_client
        "
    interfaces:
        - compute_path_to_pose_: ac-> "
            bt_skill_client::
            compute_path_to_pose"
        - compute_path_to_state_: ac-> "
            bt_skill_client::
            compute_path_to_state"
        - compute_path_to_point_: ac-> "
            bt_skill_client::
            compute_path_to_point"
        - execute_trajectory_: ac-> "
            bt_skill_client::execute_trajectory"
        - get_current_ik_frame_pose_: ac-> "
            bt_skill_client::
            get_current_ik_frame_pose"
        - detect_aruco_marker_: ac-> "
            bt_skill_client::detect_aruco_marker
            "
        - control_gripper_via_io_: ac-> "
            bt_skill_client::
            control_gripper_via_io"
connections:
  - ["/image_raw", "image"]
  ...
  - [set_io, set_io_command]
```

Listing 1. Extract from the ROS System DSL for the MANI-02 with the Universal Robot UR5e robot arm.

Listing 1 shows an extract from the ROSSYSTEM model, the design of the system developed using the RosTooling DSL. The language in YAML format is relatively intuitive to write and understand. First of all, the name of the system must be set, in this case "qr_detection_and_decision_of_picking_robot." Then the user can define three different types of attributes: (1) *subSystems* to include other exiting systems as complete modules; (2) *nodes* as components based on existing ROS nodes,

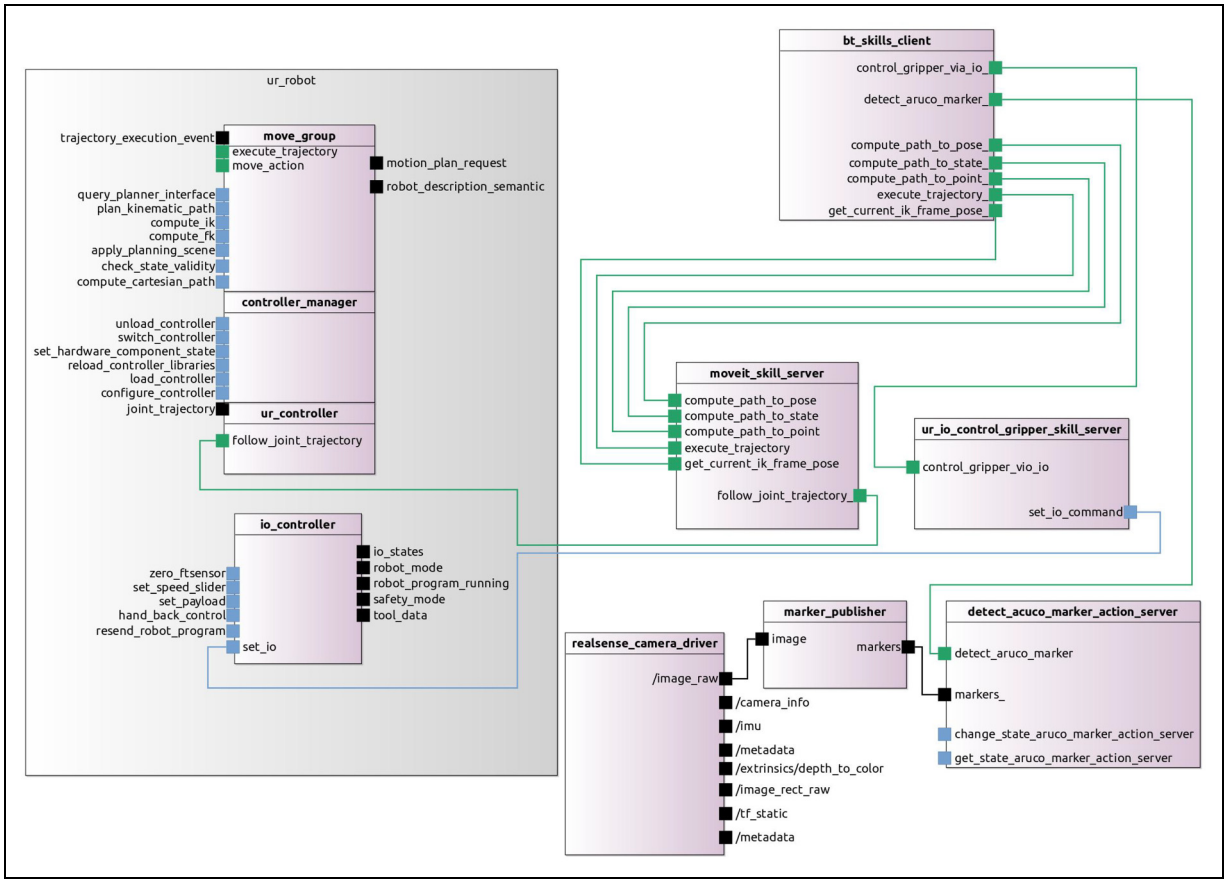

Figure 4. Screenshot from the visual interface of the RosTooling showing a simplified overview of the system for the MANI-02 user story for the Universal Robot UR5e.

Figure 5. Care-O-bot setup used for the experiments. The real hardware robot on the left side, and the simulation environment in the right.

like the known example "realsense_camera_driver"; and (3) connections among the nodes and/or the subsystems. In case an unexpected connection, like interfaces with a different type, are connected, the user interface of the RosTooling will pop up the issue and will not proceed with the code generation. It should be emphasized in this design how this composition facilitates the modularity and resusability of parts of the design. In this case, changing the *ur_robot* subsystem for the one corresponding to the PRBT arm, we obtain our ported application. In terms of the model, this means the modification of line 5 in Lst. 1. Once the model is completed, by saving it inside the RosTooling it will be automatically validated and it generates a ROS package containing ready-to-execute code for the designed system, the generator needs just a few seconds for this.

Mobile assistant robot use case. For this study, we used the Care-O-bot 4,[20] this robot features an advanced and completely modular hardware setup as shown in Figure 5. Its core is an omnidirectional mobile base that includes three powerful laser scanners. The torso and head are outfitted with multiple sensors, by integrating a total of five RGB-D cameras. This robot enables, among other, navigation, perception, manipulation, and human robot interaction functionalities. For the experiment, two setups have been used, the real robot, which is running on ROS 1, and a simulation environment created in ROS 2.

With this study, we addressed several topics. First of all, the interoperability between middlewares. Secondly, a design based on an existing and deployed system. This is a typical scenario in robotics, many mobile platforms can be bought with a pre-defined installation. Lastly, this type of robot is very complex as it enables the development of a large number of functionalities.

For this scenario, we consider two variants, the real robot in ROS 1 with bridges to ROS 2 software and the simulation in ROS 2. By defining three different user stories (Full description of the user stories attached as supplemental material), we developed six different application variations. All six were developed by using the two methods, the traditional approach, and the RosTooling proposal.

The first step to using the RosTooling was the analysis of the running robot by using introspection tools. This reverse engineering method, introspect the information about the system being executed on the robot and outputs a system model that encloses the list of all the running nodes, their configurations, and the connections among them. Also, it lists all the open interfaces to connect further functionalities. On the RosTooling, this model can be imported as a big block that will act as a subsystem of the complete system and entails 75 components. This simulation environment is defined as a single subsystem that contains 10 components to enable a replicated version of the most relevant hardware drivers. Also, a subsystem collects all the navigation framework, composed of 13 components. Table 2 summarizes the implemented cases.

```
# .rossystem file for the MOBI-02 user story with
    the real robot cob4 as subsystem
cob_navi_robot:
3 subSystems:
4   cob4_bringup
5   cob_nav2
6   launch_visual
7 nodes:
8   joy_node:
9     from: "joy.joy_node"
10    interfaces:
11      - joy_pub: pub-> "joy_node::joy"
12    parameters:
13      - autorepeat_rate: "joy_node::
            autorepeat_rate"
14        value: 20.0
15      - deadzone: "joy_node::deadzone"
16        value: 0.3
17      - joy_dev: "joy_node::joy_dev"
18        value: "/dev/input/js0"
19  twist_mux:
20    from: "twist_mux.twist_mux"
21    interfaces:
22      - '/base/twist_controller/command': pub->
            "twist_mux::cmd_vel_out"
23      - cmd_vel_sub: sub-> "twist_mux::cmd_vel"
24  .....
25 connections:
26      - [ joy_pub , joy ]
27      ....
28      - [ map , map_sub ]
```

Listing 2. Extract from the ROSSYSTEM model for the MOBI-02 user story where the real robot Care-O-bot 4-25 is integrated as subsystem.

The ROSSYSTEM model extract in Listing 2 shows a part of the design with the RosTooling of the user story MOBI-02, whose system name is "cob_navi_robot" for the real robot, where three modules are imported as subsystems, including *cob4_bringup*, which includes ROS 1 implementation. The rest of the components are listed under the category *nodes*. Also, as for the previous example (1), the connections expected among the parts of the system are specifically listed for their validation. The model compiler will check and automatically detect the nature of the connected interfaces and add the bridges to the system. This means that by generating the code, the compilers will also generate the executable code for the bridges. The generated bridges utilize the standard ros1_bridge (https://github.com/ros2/ros1_bridge) library provided by the official ROS 2 distribution. Although potential latency issues may arise from its use, these are not relevant for this study, as the same issues appear for the two environments of this study, MDD and the traditional approach.

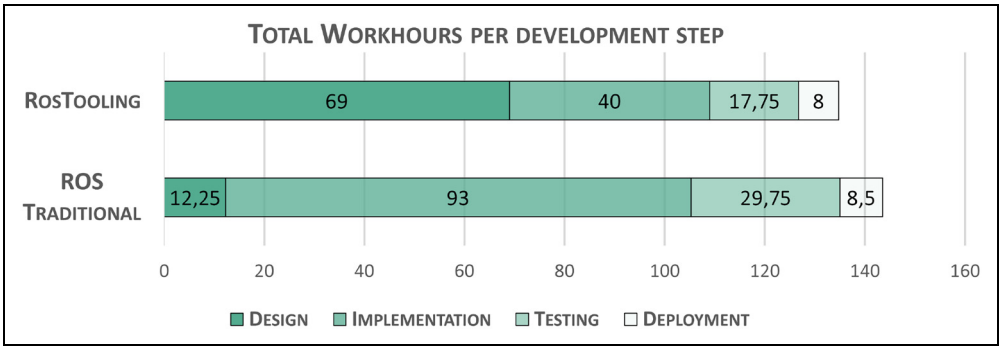

Figure 6. Total work hours taking into account all the user stories for both use cases and splitting them by development steps.

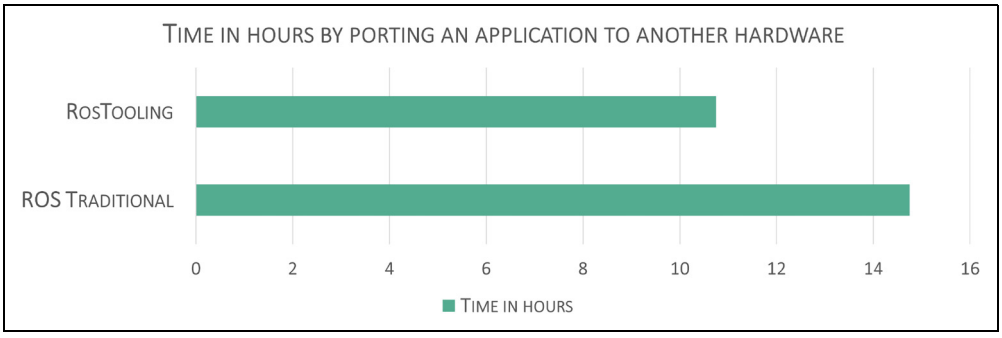

Figure 7. Work hours data by splitting the number of hours required to port an application to a new hardware, the target of the manipulation use case.

Collected data

Development time. Based on Figure 6, the key findings are: (1) Modeling tools make design the most time-intensive phase, (2) code generation significantly reduces implementation time, cutting it from 93 hours to 40 hours (a 57% reduction), and (3) testing and debugging time is also reduced, from 29.75 to 17.75 hours (a 40.3% reduction). Overall, modeling tools shift development focus from implementation to design. Design in this case means the formal description of the system architecture. The actual task performed by the developer, starting from the basis of component models, is either with a graphical interface or with the use of modeling languages (or DSLs) to describe the configurable components that form the system and how they interact with each other. In our study, that means that the design consists of the creation of ROSSYSTEM files as the examples in Listing 1 and Listing 2. Obviously, the more advanced the tooling user becomes, the faster and more intuitive it will be to use, as well as the more reusable model patterns will be constructed.

Additionally, using RosTooling generates code documentation, installation and execution guides, component lists, and system diagrams. These benefits, though not visible in this study, would be valuable during system maintenance, upgrades, or hardware changes. Unfortunately, this study was limited to the development phase of the system lifecycle due to constraints in human resources. Analyzing data from the operation and maintenance phases in future work would provide valuable insights. Furthermore, extending RosTooling to support runtime modeling will be essential for a better demonstration of the advantages offered by MDD approaches for system maintenance.

Lastly, for the manipulation case, RosTooling was tested by porting an application from one hardware manipulator (UR5e) to another (PRBT arm). Figure 7 shows the comparison of the total number of hours. The reduction of the needed time is 27%, this is a considerable amount of time and resources.

Tables 3 and 4 illustrates the development effort, measured in hours, required for various use cases across different development approaches. In the case of a simple use case, the effort is slightly higher when using modeling techniques (18 hours) compared to traditional development (17 hours). The creation of models and the associated entry barriers of the technology do not justify their use for integrating simple components. However, this scenario changes for medium-complexity systems, where the modeling approach, RosTooling, demonstrates a reduction in development effort. The advantage of using modeling becomes even more evident for more complex systems, where the

Table 3. Workhours development time by complexity of the system using the traditional approach.

User Story ID	Design	Implementation	Testing	Deployment	Total
MOBI-01 COB (simple)	3	9	2.5	2.5	17
MOBI-02 COB (medium)	2	12.5	7.5	1.5	23.5
MOBI-03 COB (complex)	4	8	7.5	1	20.5

Table 4. Workhours development time by complexity of the system using the RosTooling.

User story ID	Design	Implementation	Testing	Deployment	Total
MOBI-01 COB (simple)	11.5	2	3	1.5	18
MOBI-02 COB (medium)	13.5	2.5	4.5	2	22.5
MOBI-03 COB (complex)	11.5	3.5	2.5	1	18.5

traditional approach requires 20.5 hours, whereas the modeling solution reduces this to 18.5 hours. It is important to note that, in our case -the mobile platform- the components are reused from one user story to the next, in other words, it is a progressive development of the same application until the MOBI-03 case is achieved, which is the most complex. Consequently, there is no progressive increase in development hours with complexity, as the effort invested in the simpler and medium-complexity cases is reutilized in the more complex one. Figure 8 summarizes the reduction of time (in percentage) and clearly shows the tendency, the more complex the system is, the greater the percentage of time is reduced.

Lines of code. Figure 9 shows the total LOC written manually by adding the data from both use cases and all the user stories implemented. We can see that by the implementation of the ROSSYSTEM models and thanks to the generators we reduced from 5,616 LOC to 1,776 LOC this means 31% of the total number of LOC. Also, by including subsystems in the models, we reduce from 1,283 LOC to 290 LOC.

In total, with the traditional approach 6,854 lines must be manually written, this includes Python code, build, manifest, and configuration files in YAML format, while with the tooling only the ROSSYSTEM model files must be written, all in YAML format, and the total of lines is 2,066, the achieved reduction is 69.86%.

Runtime error-fixing time. Figure 10 represents the sum of all the counted minutes by all the studied user stories together.

The most evident reductions are seen in misspelling issues from 30 to 10 minutes and in mismatched interfaces, it reduces from 134 to 40 minutes. This is because the models include spelling checkers and cross-validation of the properties while designing.

Regarding the system performance, there is no difference, it is the same for both approaches since it depends on the hardware and the quality of the components used.

In the case of fixing errors of values given to the parameters, we see that with the RosTooling we need more time, from 350 and 380 minutes, this is probably because with the traditional ROS approach only a concrete file of the YAML is modified for a component. This is a relatively quick and localized change. In contrast, when using RosTooling, parameter corrections must be made within the system model itself. After modifying the model, the corresponding code and configuration files must be regenerated to reflect the changes. This additional modeling and generation step introduces overhead, which explains the longer resolution time. This is related to the difference between a design-centric approach and an implementation-centric approach. In case of the use of runtime models with self-adaptation through the adjustment of parameters, this will be optimized.

Portability. Table 5 presents the development costs, measured in LOC, for creating a new user story from scratch compared to porting components from a previous development. In the mobile platform use case, the traditional approach requires 1299 LOC to develop a new user story. By porting components from a similar application, this number decreases to 283 LOC. In contrast, the MDD approach necessitates 531 LOC for a new user story, which can be reduced to 76 LOC when reusing components. For the manipulation use case, which emphasizes portability, the traditional approach entails developing 1.859 LOC for a new application. Porting components reduces this to 759 LOC. Using the RosTooling approach, developing a new application requires 448 LOC, and porting components further decreases this to 42 LOC.

By using the equation previously presented as shown in Figure 11, for the manipulation use case Figure 11, the score with the RosTooling is 0.9, while with the traditional approach down to 0.59.

For the case of the mobile assistant robot, by evaluating the portability between the simulation and the real robot

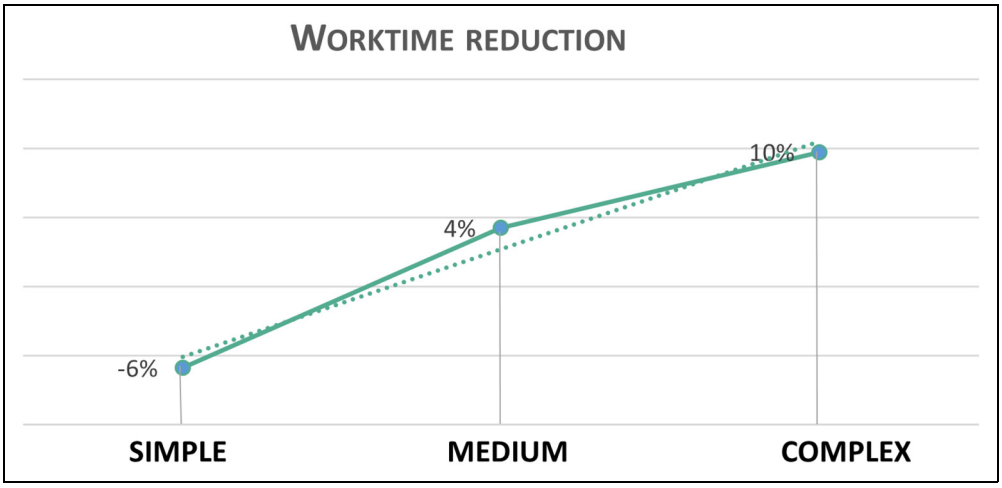

Figure 8. Difference of the work hours metric system complexity for the mobile assistant robot use case.

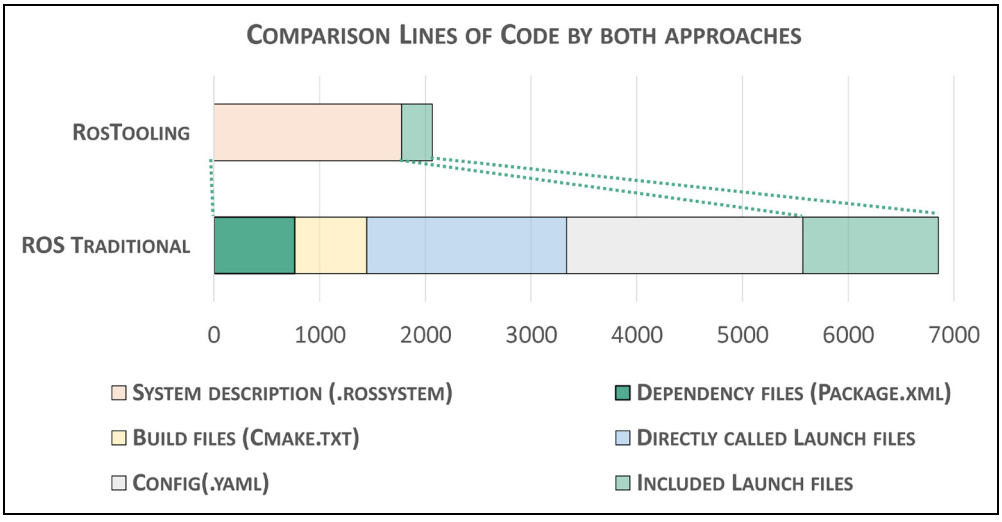

Figure 9. Total amount of lines of code manually written by both approaches by taking into account all the use cases and user stories.

environment, the resulting value by using the RosTooling is 0.85, while we get 0.78 for the traditional approach.

In terms of portability, the use of models has a high performance. If the design of the system is modular, porting modules from one application to another using models is very efficient, as the system model can reuse complete modules from other applications.

Observations

MDD is characterized as a design-centric approach. Then in our case study, as expected, design also gets higher importance, shown in Figure 6, this is the step that requires more time, while in a code-centered development, the implementation is the step that requires more effort. But like every MDD approach, a better design reduces time in the rest of the steps, especially we should highlight the reduction of the testing time, as well as the reduction of the time to fix errors, as shown in Figure 10. Different measures to improve the efficiency of the design could be a more intuitive graphical interface or the extension of the catalog of components and subsystems, in order to reduce the time of analysis of the available elements for their composition.

Two factors increase the time in the development of systems, on the one hand, the learning of a new tool and a new language, on the other hand, for this type of solution it is essential to have a catalog, a set of models of components that can be imported and composed to create systems. This can be clearly seen in Figure 8, where the RosTooling does not give better results than the traditional method for simple systems, however, once learned and supported by a base of models, the time is considerably reduced, this is evident if we analyze the case of portability of Figure 7.

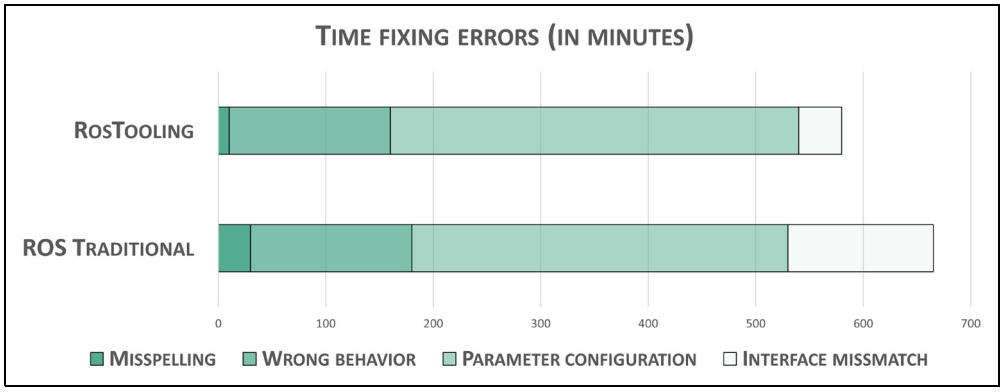

Figure 10. Time counted in minutes to fix the errors found during the testing step.

Table 5. Comparison of the lines of code (LOC) written to implement a new use story from scratch versus porting parts of the code for both approaches.

Use case	New LOC by rewriting	New LOC by porting
MOBI - Traditional	1299	283
MOBI - RosTooling	531	76
MANI - Traditional	1859	759
MANI - RosTooling	448	42

In code generation efficiency, it is evident that the use of the models brings a clear advantage (Figure 9). It reduces the amount of manually written code, which is also error-prone. It must also be noted that with the modeling solution, the user only has to learn one specific language or format, while the manually created code involves the ability to develop code in various languages.

This code generation efficiency is also delivering good results while analyzing the portability as shown in Figure 11. This is a very notable advantage for modular robots or serial production.

Threads to validity

In empirical evaluations, construct, internal, external, and conclusion validity are critical considerations, as established by scientific studies.[21] Construct validity concerns arise from the flexibility in operationalizing evaluation criteria, such as code generation efficiency and portability, which could be measured in terms of time but would still support the conclusion that RosTooling minimizes refactoring effort for porting applications to different hardware. For internal validity, the primary limitations stem from subjective metrics, or "soft data." To mitigate this, we introduced randomization through varied robot use and altered the sequence of applying the MDD and traditional ROS approaches. Regarding external validity, this study's results are contextualized by two participant profiles: one with a computer science background focused on software engineering and the other, a mechanical engineer specializing in robotics, both lacking prior experience with the target technologies. Since programer skill heavily influences results, some strategies were applied to mitigate skill-related biases:

1. The development was carried out by people not involved in the development of RosTooling, and without previous experience with ROS.
2. In both cases, the developers were fully dedicated to the experiments, free from the distraction of parallel tasks.
3. The order in which the approach was performed first (with RosTooling or with the traditional method) was interchanged at each developed user story.
4. If there is a development effort is used for both, then the time is added to both approaches.

Testing on physical hardware further aimed to emulate real-world conditions. Finally, conclusion validity, which addresses risks of type I (false-positive) or type II (false-negative) errors, was safeguarded by using multiple metrics to inform a reliable statement in the observations.

Qualitative evaluation

For this study, we encouraged people who are not involved in the development of the tool to experiment and use it.

We created material on how to test all relevant features and conducted hands-on experiments with two different groups of developers. In both cases, the target was professionals working on research projects related to the development of software for robotic applications.

Both groups were given a brief theoretical introduction to model-driven programing. Then participants had two hours to experiment and carry out different examples and tutorials. Finally, the attendees filled out a survey, the results of which are detailed here.

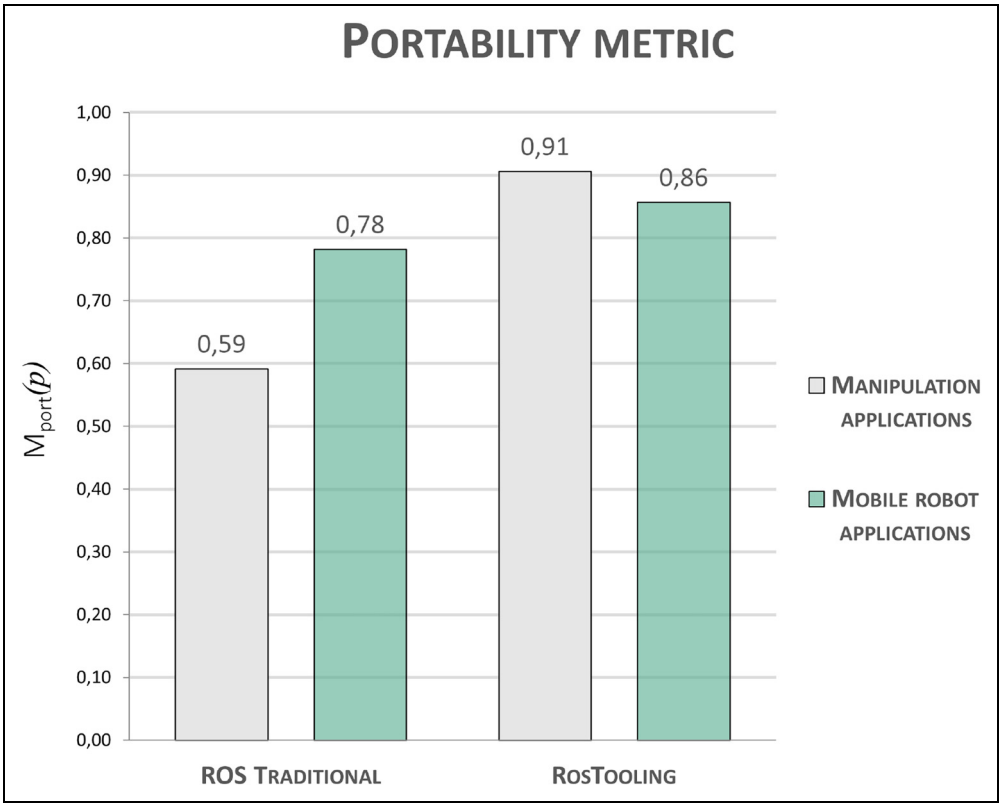

Figure 11. Portability score for both use cases.

Design

We combined GQM and SUS approaches. GQM and SUS cover broad and deep aspects of software evaluation. GQM's structured approach ensures that all relevant aspects of the software engineering process are considered, while SUS provides detailed usability data that can highlight specific areas for improvement.

The goals we aim to cover with our study are:

G1Identify the benefits of using RosTooling during the development of robotic software systems. In order to disseminate the use of MDD over handwritten code, we must determine the type of cases for which it is most profitable.
G2Identify where RosTooling is most valuable in the development and lifecycle of a system. The development of a system is divided into several phases; it is crucial to know in which of them the MDD method is more appropriate than the traditional approach.
G3Evaluate the usefulness of ROS modeling. Analyzing the usefulness of a software tool ensures that it effectively meets user needs and justifies the investment of time and resources in its adoption or development.
G4Evaluate the ease of use of models and the RosTooling. Analyzing the ease of use of a software tool is essential to ensure that users can efficiently and effectively interact with it without unnecessary training or frustration.
G5Evaluating the acceptance of model-based techniques over ROS existing code. Analyzing the acceptance of a software tool by a user group is crucial to determine whether it will be adopted in practice and integrated into users' workflows.

For clarity and consistency, the questions (Q1–Q34) (Questionnaire attached as supplemental material) have been organized into different sections:

S1(Q1-Q6) Profiling questions
S2(Q5-Q7) Models evaluation
S3(Q8-Q14) Ease to use
S4(Q15-Q25) Usability
S5(Q26-Q29) Comparative evaluation with existing methods
S6(Q30-Q31) Future usage intentions
S7(Q32-Q34) Open feedback

The questions can be mapped to the goals in the following way:

G1Q22, Q23, Q24, Q26, Q27, Q28, Q29
G2Q21, Q22, Q23, Q25

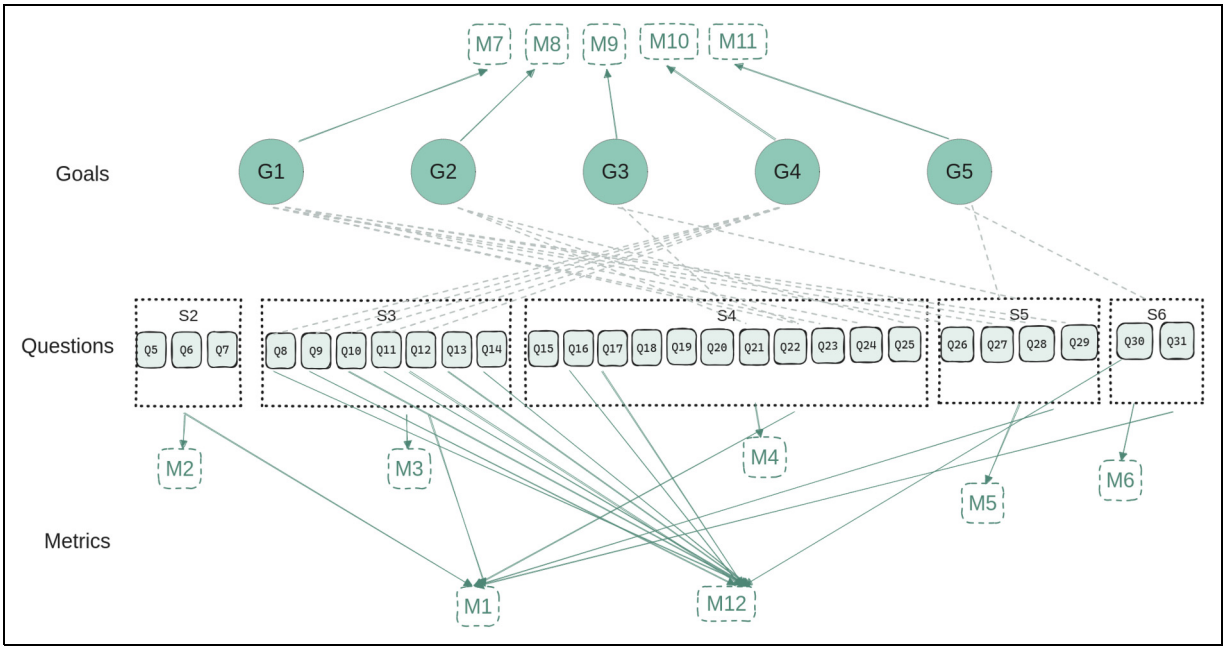

Figure 12. Design of the goal question metric (GQM) methods. The diagram shows the relations between the selected goals, questions and metrics.

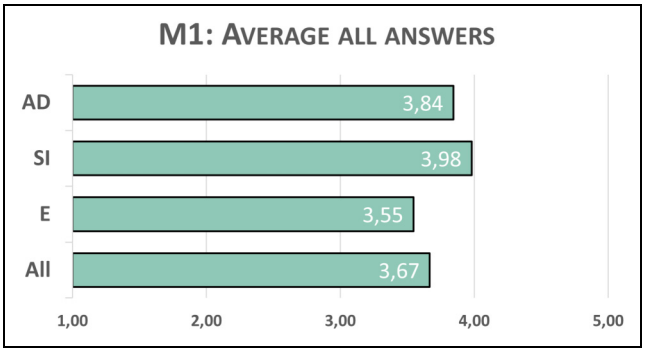

Figure 13. Results obtained for the M1, i.e., considering the answers for all the questions, and all with the same weight.

G3Q15-Q25, Q26-Q29
G4Q8, Q9, Q10, Q11, Q12
G5Q26-Q29, Q30-Q31

To simplify the collection of the data and also to get manageable data, all the questions, excluding the profiling questions and the open feedback ones, can be answered with a 5-point Likert scale, where "1" strongly disagree and "5" strongly agree.

To complete the GQM method, the last aspect is the definition of the metrics:

M1General average all the questions. This means the use of all the answers Q5-Q31.

M2Average of all the questions in S2, evaluation of the models
M3Average of all the questions in S3, ease to use.
M4Average of all the questions in S4, usability.
M5Average of all the questions in S5, comparative with traditional methods.
M6Average of all the questions in S6, future use intentions
M7Average of all the questions related to G1
M8Average of all the questions related to G2
M9Average of all the questions related to G3
M10 Average of all the questions related to G4
M11 Average of all the questions related to G5
M12 Score formula from the method SUS, applied to its related questions.

As a representation of how they are related to each other, Figure 12 shows the mapping between the goals, the questions, and the metrics.

Collected data

With this experiment, we collected a total of 15 responses. For the analysis of the data, all the answers are divided into the profiles ROS experts people with more than five years of experience with ROS), systems integrators and architecture designers (AD).

Considering all the data obtained and giving all the questions of the survey the same weight, a broad overview of the data obtained is presented in Figure 13.

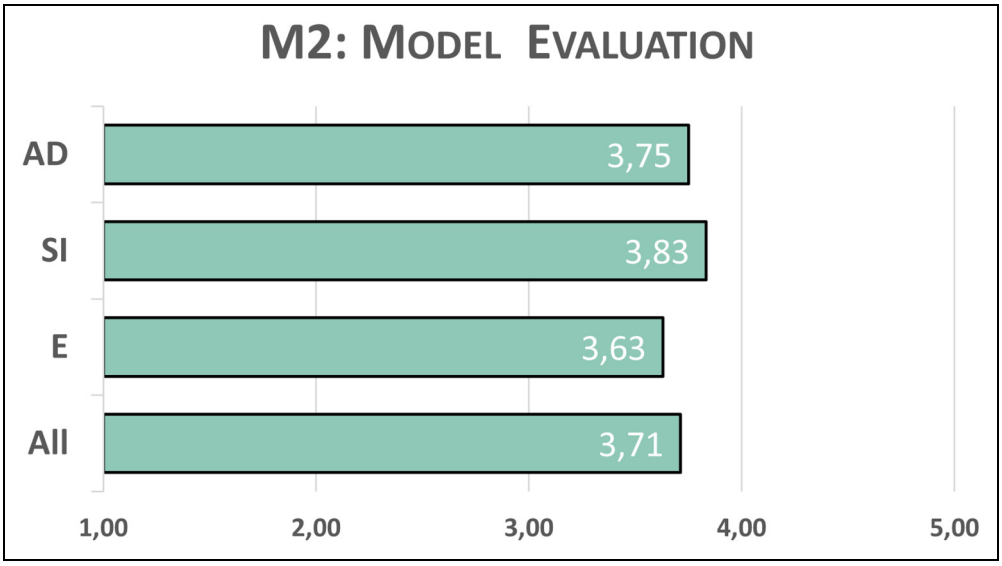

Figure 14. Results obtained for the M2. Answers related to the completeness, consistency, and clarity of the models.

In general terms, we can see that the profile with the highest degree of satisfaction with the solution we provide is the system integrators (SIs). This certainly makes sense, since they are the target audience of the RosTooling.

Next are the AD, this profile is very interesting because they are the people who really understand the need to make a well-documented design from the beginning of the development process.

For the evaluation of the models, we take into account factors like model completeness, which measures how well models capture the entire system, model consistency, to check for coherence, preventing misunderstandings and errors in the development process, and model clarity, to assess readability.

Using a 1–5 scale, 1 means that models are unclear, difficult to interpret, and fail to accurately or precisely represent the structure and behavior of a ROS system. Important properties and nuances may be missing or misrepresented. 5 implies that the models are clear, well-structured, and easy to interpret. They accurately and precisely capture the key properties of ROS systems.

In this aspect, the results are very similar for all the profiles they are between 3.63 and 3.83, as shown in Figure 14. The clarity of the models and their accuracy in capturing the properties is highlighted by the responses. The worst ranking is found in the accuracy in capturing nuances of the software.

The next two metrics evaluate the ease of use and the usability of the technical solution. For the evaluation of ease of use, we have two metrics, **M3** that correspond to all the questions related to the section **S2**, and the **M10** mapped to **G4**. The main difference between both metrics is that **M3** includes how easy is to learn the tooling while for the **M10** we want to know, once the tooling is established, how easy it is to use, we get this by omitting all the answers about learnability.

Using the scale 1–5, we define score 1 as the tooling is very difficult to use, with a steep learning curve and high complexity. Users need assistance, find it cumbersome or unmanageable, and lack confidence when using it. The tool's interface and workflow are unintuitive, making even basic tasks frustrating and inefficient. While 5 means the tooling is very easy to use, it has an intuitive design. Users feel confident and independent. The tooling is well-organized, manageable, and does not overwhelm users with unnecessary complexity or excessive learning requirements.

Looking at the comparison in Figure 15, it is clear that the ROS experts are the ones who give the lowest score for ease of use, we obtain here the most negative metric of our whole survey (3.33 for **M10** and 3.41 for **M3**). SIs are those who did not have great difficulties in using the RosTooling.

M4 and **M9** evaluates usability. **M4** considers the typical questions for the usability evaluation of a software tool. For **M9**, we have also accounted for usefulness compared to the way of integrating systems currently using ROS (**S5**).

For the usability scale, from 1 to 5, 1 means that the tooling negatively affects development as it restricts how the developer wants to work, feels hard to integrate, or lacks coherence. Features may feel disconnected or unhelpful, offering little value during software design or coding. Conversely, 5 means it integrates smoothly, fits the developer workflow, and adds clear value. Features are consistent, intuitive, and helpful for design, implementation, and collaboration.

In Figure 15, we observe that in general, more evident in the case of ROS experts, we score higher in usability than in

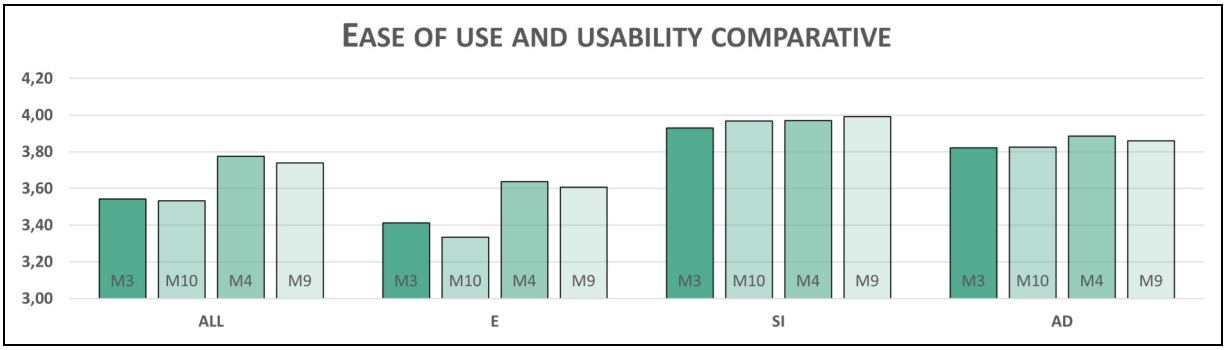

Figure 15. Comparison of M3 and M10, easy of use considering learnability and without it, and of M4 and M9, usability in terms of the typical aspects of software engineering tools and taking into account its benefits on top of robot operating system (ROS).

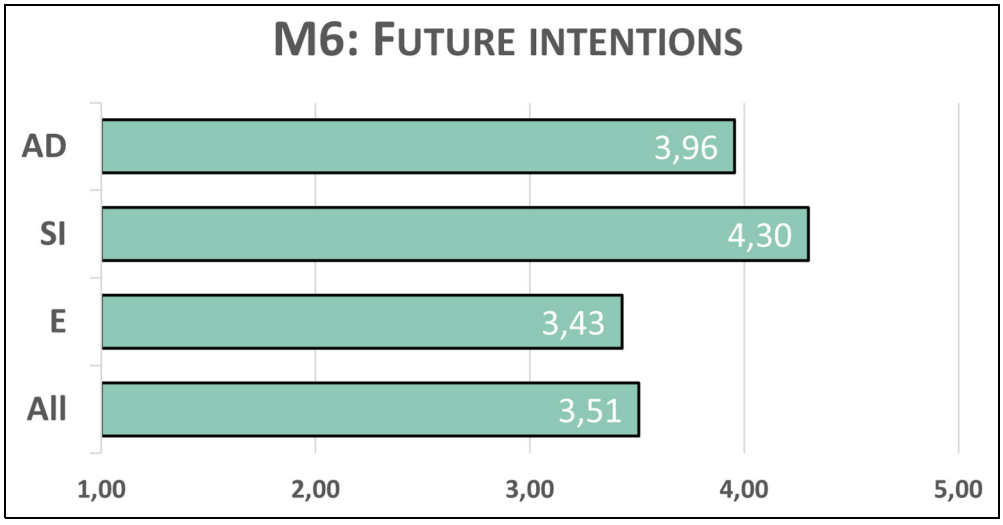

Figure 16. Results obtained for the M6, i.e., the future use intentions.

ease of use. For SIs, the average is slightly higher if we consider the comparison with the usability compared to the current process in ROS (3.99 vs. 3.96), while the trend for the rest of the profiles is the opposite. In either case, the differences are not very significant.

The **M6**, whose results are shown in Figure 16, attempts to evaluate the intentions of the use of the proposed solution in the future. The results are really satisfactory for the system integrator profile. The average of all the answers is 4.3, the highest value of this study.

With the **M7** we aimed to get data that helps us to identify the benefits of using a MDD approach. From highest to lowest score, these are the items in which the participants have answered us:

- The models can be better understood compared to manually written packages (4/5 for all, and 4.8/5 for system integrator).
- The communication between developers is reduced by using models (3.73/5 for all, and 4.33/5 for SIs).
- The system software is better documented than typical manually written launch packages (3.72/5 for all, and 3.8/5 for system integrator).
- When configuring the system, the amount of changes inside the code is reduced (3.54/5 for all, and 3.8/5 for the system integrator).
- The code generator makes implementation and deployment easier (3.4/5 for all, and 3.83/5 for SIs).
- The amount of system configuration is reduced compared to manual development (3.27/5 for all, and 3.8/5 for SIs).
- The approach reduces the validation and testing efforts (3.26/5 for all, and 4.16/5 for SIs).

The next metric, **M8**, targets the stages of the development lifecycle. Table 6 shows the obtained results separated by profiles.

Figure 17 shows the average values for **M11**, which comprises the answers related to the comparison with the ROS traditional approach and the future usage

Table 6. Answers related to the M8 metric, the relation between the RosTooling and the SDLC.

Inquiry "The RosTooling· · ·	All	E	SI	AD
..helps by encouraging the user to design before implementing."	3.93	3.66	4.5	4.37
..helps on design thanks to the visualization tools."	4.33	4.22	4.16	4.25
..helps with the implementation and deployment by using code generators."	3.4	3.22	3.83	3.62
..helps on testing, by using the validation tools."	3.26	3	4.16	3.75

AD: architecture designers; SI: system integrators.

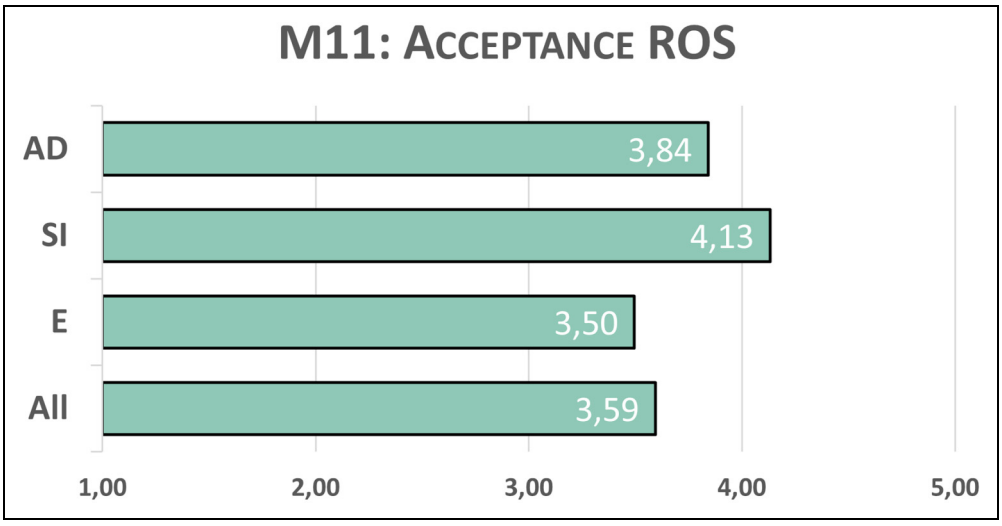

Figure 17. Results obtained for the M11. The acceptance from the robot operating system (ROS) community.

intentions. The joined data should be helpful in determining the acceptance of the proposed solution over ROS existing code. In this sense, we consider 1 that the use of the RosTooling does not reduce the amount of system configuration or code changes compared to manual development. The models add complexity, and the resulting software is poorly documented, offering little improvement over manually written packages. While 5 means that RosTooling significantly reduces the amount of system configuration and code changes needed. The models generate well-documented software, greatly improving system transparency and maintainability over traditional manual methods. According to the same trend that we have observed in all the results, the acceptance is higher for SIs. And we obtained the lowest value in the ROS experts sector.

Lastly, we calculate the SUS score by applying its standardized equation.[8]

The result is a number between 0 and 100, where if the score is between 50 and 70, the acceptance of the surveyed solution is marginal. For values over 70, it is considered acceptable. Table 7 shows the resulting scores for the RosTooling. For all the profiles, the obtained SUS score is over 50, while for SIs and AD, it is even over 70. We can conclude that the RosTooling passes the threshold with a slightly accepted marginal.

Observations

G1: Identify the benefits of using RosTooling during the development of robotic software systems. (M7). The three most voted options (see M7 results) are related to understandability, the reduction of the gap between the different members of the team, and documentation. All these points are somehow related to process optimization.

Understandability and the improvement of the communication between the members of the team ensures a more efficient workflow and fosters collaboration. Effective documentation serves as a reference that standardizes processes and provides consistency, which reduces time spent on clarifications and retraining, and also eases the maintenance of the system.

The lower scores are assigned to the items related to technical tools that reduce the programer's effort. It is well known that the ROS community, used to programing code by hand, generally tends not to appreciate code generators. Therefore, the answers are not a big surprise. Likewise, everything related to technical tools is evaluated previous section, which the quantitative evaluation.

Nevertheless, the SIs appreciate very positively the validation during the design of the composition of the systems.

G2: Identify where RosTooling is most valuable in the development and lifecycle of a system. (M8). The obtained

Table 7. SUS results for the evaluated profiles and the meaning for the obtained values.

Profile	Score	Acceptability	Rank
All participants	64.87	Marginal	OK
ROS experts	61.25	Marginal	OK
System integrators	73.58	Acceptable	Good
Architecture designers	70.71	Acceptable	Good

SUS: System Usability Scale; ROS:robot operating system.

answers (see Table 6) clearly show the value of using models, and a modeling-based tool, for the design of the system. Modeling, thanks to its visual representation, improves the overview of the system, its analysis, and documentation. But also, assets consistency, by providing a blueprint to be followed by the developers, and to some extent, reliability of the composition, by checking the validness of the connections and interfaces.

G3: Evaluate the usefulness of ROS modeling. (M9). The fact that the SIs gave a pretty high score on usability (3.99/5) as shows M9 in Figure 15, even for the SUS scale (73.58/100) as shown in Table 7, indicates that the current development of the RosTooling is targeting the correct users, and they understand the needs and benefit of using an MDD solution for the integration of robotics systems.

G4: Evaluate the ease of use of models and RosTooling. (M10). In this aspect we must make a disclaimer, RosTooling is created as a proof of concept for the use of models for ROS in the context of research projects. Therefore, the frontend part and the user interface are mostly leveraged on tools available in Eclipse. So the ease of use depends on the strengths and limitations that Eclipse offers us. In this aspect, for RosTooling to be a highly user-friendly solution, requires further development of the user perspective. Showing the data the rating given by the SIs is not too bad, see M10 in Figure 15, (3.99/5), but we can conclude that the current development is not ROS-user-friendly.

G5: Evaluating the acceptance of model-based techniques over ROS existing code. (M11). About acceptance (Figure 17) we see that there is a great acceptance to use models during the design step. Using models as an artifact to improve the understanding between the members of the team, as well as to serve as documentation of the system.

Acceptance is lower in the implementation. This also fits the data divided by profiles, integrators and architects give more importance to the design and therefore see that the use of models adds value to their work.

The implementation part, in ROS, is mostly done by hand, the experts, used to hand-crafted code, do not favor code generators as they add rigidity to the implementation, being this the root of a lower acceptance.

Threads to validity

To address the threats to validity in our survey, we assess construct, internal, and external validity following standard classification guidelines.[21] For construct validity, we utilized the GQM methodology to ensure questions directly mapped to evaluation goals, acknowledging that our metrics relied on averaging responses. To enhance goal alignment, questions were crafted clearly and concisely, and we analyzed combinations of metrics to better capture goal-related aspects. Internal validity was maintained by limiting introductory materials to theoretical overviews, allowing participants to independently explore the tools and benefits without external influence. Although the 5-point Likert scale tends toward neutral responses, we adopted it as part of the SUS scale, widely validated in terms of comprehension and response rate. For external validity, recognizing that the solution targets robotics practitioners, we conducted the experiments within European robotics project teams, drawing participants from both academia and industry. Participants brought diverse profiles and expertise, representing fields such as drone, service, logistics, and industrial robotics, as well as developers of application-agnostic architectures.

Conclusion

Based on the observations from our studies, we can conclude that the proposed MDD solution, the RosTooling improves efficiency. Quantitative analysis shows that while design requires the most effort initially, this investment significantly reduces time spent in subsequent phases, such as testing and error correction. Code generation efficiency is a notable advantage, minimizing manual coding and accelerating implementation while reducing errors. Additionally, RosTooling proves especially valuable in enhancing portability for modular systems, making it easier to adapt software to diverse hardware platforms.

The tool also facilitates collaboration and standardization through its modeling approach. Visual models improve team communication, helping bridge the gap between members with varying expertise. They also serve as comprehensive documentation artifacts, streamlining processes and easing maintenance. However, the usability of RosTooling remains a challenge. The entry barrier and the reliance on platforms such as the Eclipse IDE limit user experience. For further development, the popular IDEs used by the target audience, the ROS community, must be taken into account.

Adoption of RosTooling changes across development phases. It is highly appreciated during the design phase, as it improves system understanding, validation, and

documentation. However, acceptance is lower in implementation, recalling a preference among ROS developers for the flexibility of hand-coded solutions. This emphasizes the cultural resistance to code generators within the ROS community. To achieve greater acceptance, especially within the ROS community, the entry barrier must be further lowered by facilitating a more user-friendly development environment. ROS developer favors development environments like VS Code over Eclipse. Besides not particularly enthusiastic about graphical editors, but rather more of a fan of powerful code editors. Moreover, in the open-source community, dissemination is essential, particularly in the form of hands-on sessions where the developers themselves can put the new tools into practice. Another way to gain users is to allow the community access to the development of the tool itself; for this, the RosTooling is extensible by third parties. It would be recommended the add of tutorials on how to add new plugins can be a great way to involve other developers in the project.

It is worth noting that conducting the same experiment using additional robotic system development tools would be beneficial to enable broader comparisons beyond just RosTooling. In our study, we intentionally positioned the MDD tool as an abstraction layer on top of the same middleware. This approach ensures cleaner and more consistent comparative data by avoiding deviation factors such as runtime errors caused by the middleware implementation itself. In other words, our primary objective is to assess how different tools assist developers in producing analogous code. Unfortunately, other MDD tools on top of ROS regenerate completely new code for each component to be integrated, so the data about the development time will also be considerably shifted. Recently, the MDD tool Papyrus for Robotics (https://gitlab.eclipse.org/eclipse/papyrus/org.eclipse.papyrusrobotics) introduced support for reverse engineering of ROS 2 code, making it also suitable for the study. Unfortunately, this feature was not available at the time our experiments were carried out, so it was not possible its insert.

In summary, the MDD solutions demonstrate a significant potential to enhance robotic software development through better design practices, improved collaboration, and increased portability. To fully realize its benefits, future efforts should focus on improving usability, expanding model libraries, and promoting cultural shifts within the ROS community to embrace model-based techniques.

ORCID iD

Nadia Hammoudeh García https://orcid.org/0000-0001-6847-5748

Funding

The authors disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This article is supported by the CORESENSE project with funding from the European Union's Horizon Europe research and innovation programe (Grant Agreement No. 101070254). The authors of the University of Stuttgart were partly funded by the Ministry of Science, Research and Arts of the Federal State of Baden-Württemberg within the Innovations Campus Future Mobility (ICM).

Declaration of conflicting interest

The authors declare that there are no conflicts of interest regarding the publication of this manuscript.

Data availability

The data that support the findings of this study are available from the corresponding author upon reasonable request.[22]

Supplemental material

Supplemental material for this article is available online.

References

1. Younse PJ, Cameron JE and Bradley TH. Comparative analysis of a model-based systems engineering approach to a traditional systems engineering approach for architecting a robotic space system through knowledge categorization. *Syst Eng* 2021; 24: 177–199.
2. Hammoudeh García N, Deshpande H, Wu R, et al. Lifting ROS to model-driven development: lessons learned from a bottom-up approach. In: *2023 IEEE/ACM 5th international workshop on robotics software engineering (RoSE)*, pp.31–36. DOI: 10.1109/RoSE59155.2023.00010.
3. Krogmann K and Becker S. A case study on model-driven and conventional software development: the palladio editor. In: *Software Engineering 2007—Beiträge zu den Workshops—Fachtagung des GI-Fachbereichs Softwaretechnik*. Bonn: Gesellschaft für Informatik e. V., 2007, pp.169–175.
4. Martínez Y, Cachero C and Meliá S. MDD vs. traditional software development: a practitioner's subjective perspective. *Inf Softw Technol* 2013; 55(2): 189–200.
5. Kapteijns T, Jansen S, Brinkkemper S, et al. A comparative case study of model driven development vs traditional development: the tortoise or the hare. In: *From code centric to model centric software engineering: practices, implications and ROI*. Print, pp.1–38. https://api.semanticscholar.org/CorpusID:38594162.
6. Farshidi S, Jansen S and Fortuin S. Model-driven development platform selection: four industry case studies. *Softw Syst Model* 2021; 20(5): 1525–1551.
7. Basili VR, Caldiera G and Rombach HD. *The goal question metric approach*. John Wiley & Sons, Inc., 1994.
8. Brooke J. *SUS: a "quick and dirty" usability scale*. CRC Press, 1996.
9. Vlachogianni P and Tselios N. Perceived usability evaluation of educational technology using the post-study system

usability questionnaire (PSSUQ): a systematic review. *Sustainability* 2023; 15(17): 12954.
10. Bonci A, Gaudeni F, Giannini MC, et al. Robot operating system 2 (ROS2)-based frameworks for increasing robot autonomy: a survey. *Appl Sci* 2023; 13(23): 12796.
11. Hammoudeh García N, Deshpande H, Santos A, et al. Bootstrapping MDE development from ROS manual code: part 2 model generation and leveraging models at runtime. *Softw Syst Model* 2021; 20: 1–24.
12. Fowler M. *Domain specific languages*. Addison-Wesley Professional, 2010.
13. Eysholdt M and Behrens H. Xtext: implement your language faster than the quick and dirty way. In: *Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion*. OOPSLA '10, New York, NY, USA: Association for Computing Machinery, pp.307–309. https://doi.org/10.1145/1869542.1869625.
14. Santos A, Cunha A and Macedo N. The high-assurance ROS framework. In: *IEEE/ACM international workshop on robotics software engineering, RoSE@ICSE*. IEEE, pp.37–40. DOI: 10.1109/RoSE52553.2021.00013.
15. Blundell JK, Hines ML and Stach J. The measurement of software design quality. *Ann Softw Eng* 1997; 4(1): 235–255.
16. Wen-Hong L and Xin W. The software quality evaluation method based on software testing. In: *2012 International conference on computer science and service system*, pp.1467–1471. DOI: 10.1109/CSSS.2012.369.
17. Papotti P, Prado A, Lopes de Souza W, et al. A quantitative analysis of model-driven code generation through software experimentation. In: *International conference on advanced information systems engineering*, volume 7908, pp.321–337. DOI: 10.1007/978-3-642-38709-8_21.
18. Huang CY and Lin CT. Software reliability analysis by considering fault dependency and debugging time lag. *IEEE Trans Reliab* 2006; 55(3): 436–450.
19. Lenhard J and Wirtz G. Measuring the portability of executable service-oriented processes. In: *2013 17th IEEE international enterprise distributed object computing conference*, pp.117–126. DOI: 10.1109/EDOC.2013.21.
20. Kittmann R, Fröhlich T, Schäfer J, et al. Let me introduce myself: I am Care-O-bot 4, a gentleman robot. In: *Mensch und Computer 2015—Proceedings*. Berlin: De Gruyter Oldenbourg, 2015, pp.223–232.
21. Shull F, Singer J and Sjøberg D. *Guide to advanced empirical software engineering*. Springer, 2008.
22. Hammoudeh Garcia N. https://ipa-nhg.github.io/MDD_Evaluation_Study.github.io/. 2025.