# Whispers and Watts: An Empirical Study of the Energy Consumption of ROS 2 Topic-Based Communication



**Michel Albonico**
Centre for Software Technology, University of Southern Denmark (SDU), Vejle, Denmark
Federal University of Technology, Paraná (UTFPR), Francisco Beltrão, Brazil


**Nuno Macedo**
Departamento de Informática, Universidade do Minho, Braga, Portugal



## Abstract

*Background*: The Robot Operating System (ROS) is a framework widely adopted for developing robotics applications. Its topic-based communication model allows distributed components to exchange data via message-passing, where suboptimal messaging configuration can lead to performance degradation and resource inefficiency.

*Aims*: This paper aims to investigate the energy consumption for ROS 2 topic-based communication, focusing on messaging configurations commonly used in real-world robotics projects.

*Method*: We systematically mine 112 active ROS 2 GitHub repositories to identify prevalent message types, frequencies, sizes, and number of subscribers. These insights guide the design of a controlled experiment in which we measure the power consumption of ROS 2 nodes across multiple configurations.

*Results*: Analysis reveals that message type and interval significantly affect subscriber-side energy usage, with complex messages also affecting the publisher side. This evidence helps guide the design of energy-aware robotics software based on publish/subscribe communication.

*Conclusions*: ROS 2 topic-based communication design decisions, especially message type and publishing interval, have a substantial impact on energy consumption.



**2012 ACM Subject Classification** Software and its engineering → Software design tradeoffs

**Keywords and phrases** ROS 2, Energy Consumption, Topic Communication, GitHub Mining

**Digital Object Identifier** 10.4230/LIPIcs.ESEM.2026.4

**Category** Technical Track Paper

**Supplementary Material** *DataPaper (The replication package with scripts and data can be found here:)*: `https://github.com/IntelAgir-Research-Group/ESEM-2026-ROS-Topic`

**Funding** Research launched under INESC TEC International Visiting Researcher Programme - 2023 Edition.


## 1 Introduction

Robots are increasingly being integrated into different sectors of society, such as industry, agriculture, and autonomous vehicles. Energy consumption is a critical concern in the development of robotics systems [35], affecting not only battery-dependent robots but also industrial applications where non-mobile robots account for a considerable portion of total energy usage. Software components have an increasingly central role in controlling robotics systems, and studies show that design decisions can considerably impact robotics software energy consumption [19]. Although relevant studies are scarce [21], previous work has identified software architectural patterns used to improve the energy efficiency of robots [29].

The Robot Operating System (ROS) [16,33] is a middleware that has become the *de facto* standard for robotics software development in both industry and research [20]. Its topic-based publish/subscribe communication model enables interoperability and modularity between components, fostering reuse and flexibility. However, this flexibility leads to a substantial design space, where developers must choose among hundreds of message types, determine appropriate publishing frequencies, and design message structures suited to their systems. These decisions can have significant energy implications since they result in different resource demands. Therefore, good practices in designing ROS component communication could directly benefit the energy efficiency of robotics software systems. As a result, primarily, battery-powered robotic missions could operate for longer durations without sacrificing other quality attributes [35]. Secondarily, this approach helps to minimize the overall software emissions footprint, since lower energy consumption typically translates into reduced greenhouse gas emissions.

The main **goal** of this work is to evaluate how design decisions in topic-based communication impact the software energy consumption of ROS-based systems. Such insights are particularly relevant in a domain where developers are often not trained software engineers and software is modularly built from existing components. In **this study**, we first systematically mine 112 active ROS 2 GitHub repositories, identifying the most frequently used topic communication configurations. Guided by those factors, we then design and execute controlled benchmarking experiments to measure the energy consumption of ROS 2 nodes across various combinations of *message types*, *sizes*, *intervals*, and *number of subscribers*, using the RAPL interface [27] for fine-grained power monitoring. Lastly, from the analysis of the results, we compile guidelines on how to design energy-efficient ROS 2 topic communication.

Experimental **results** reveal that *message type* and *interval* have a significant and statistically measurable impact on power consumption, especially on subscriber nodes. Complex sensor messages such as `Image`, `PointCloud2`, and `LaserScan` incur the highest energy cost, while lightweight types (*i.e.,* simpler data structures) like `String`, `Float32`, and `Int32` offer more efficient communication. We also observe that short *message intervals* have a greater energy impact than increased *message size*. The *number of subscribers* does not affect publisher's energy profile, confirming the delivery decoupling assumptions of the DDS layer over which ROS 2 is built.

The main **contribution** of this paper is a comprehensive empirical investigation into the energy efficiency of topic-based communication in ROS 2, which also delivers a replication package[1] that includes a valuable dataset and the implementation of the experiment. The 6 resulting guidelines inform practitioners on how to design ROS 2 topic communication in an energy-efficient manner. The **target audience** of this work is robotics researchers, ROS developers, and software engineers concerned with the sustainability and efficiency of robotic systems. By providing empirical evidence, our study assists practitioners in making informed decisions in the design of ROS 2 systems.

## 2 Related Work

Considerable research has been conducted on the impact of software on the energy consumption of robotics systems, but most focuses on how software affects the energy consumption of the whole system, including the hardware, rather than the consumption of the software

[1] `https://anonymous.4open.science/r/ESEM-ROS-Topic-rep-pkg-51A3`

itself [35]. For instance, for ROS-based systems, there is work on mining architectural "green" tactics from code repositories, which were subsequently evaluated for their impact on the energy consumption of the system [23, 29].

A few works have systematically analyzed the impact of ROS-level design and parametrization decisions on software energy consumption. Previous work has explored the issue at the programming language level [21], and architectural decisions such as communication paradigm, number of nodes, and frequency of publication [19]. This work goes a step further by focusing on the impact of finer design decisions, taking into consideration widely used configurations extracted from a rigorous repository mining process. Beyond robotics, a few works have studied the impact of architectural decisions on software energy consumption, namely on cloud patterns [28] and IoT security patterns [26]. These studies concluded that architectural decisions can have a considerable impact on energy consumption.

## 3 Robot Operating System

ROS consists of a middleware that promotes the development of modular robotics software systems. There are two main versions, ROS 1 and ROS 2, the former having reached end-of-life in May 2025. *Nodes* are the main components of a ROS system, computational units that provide a well-defined functionality. They are written in general-purpose programming languages that connect to the ROS infrastructure through the ROS Client Library (RCL) [18]. The most popular clients built over RCL are `roscpp` (C++) and `rospy` (Python).

*Nodes* communicate with each other through *topics* (an asynchronous publish/subscribe model), *services* (a synchronous client/server model), or *actions* (a client/server model with asynchronous feedback), among which *topics* are the most widely used [34]. Multiple nodes may publish and subscribe to the same *topic*. Figure 1 illustrates an example of topic-based communication abstracted as a *computation graph*, where a *node /publisher* publishes to a *topic /t*, and 2 others, */subscriber1* and */subscriber2*, listen to what is published. To publish

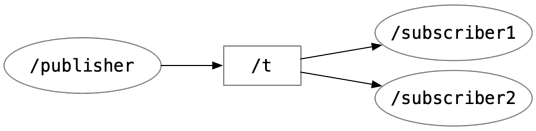



**Figure 1** Example of ROS topic-based computation graph.

to a *topic*, the *node* calls the primitive `create_publisher`, which requires a name of the *topic* and a message type. The publication of messages at a fixed rate is usually enforced through timers with callbacks (`create_timer`) or loops blocking in `Rate` objects, both provided by the ROS clients. To subscribe to a topic, the *node* calls the primitive `create_subscription`, which requires the topic name, a message type, and a callback procedure that processes incoming messages.

## 4 Research Design

In this section, we present how the research is planned after a Goal-Question-Metric (GQM) [36] definition. Then, we detail the data analysis of the experiment results. This study's main **goal** is to *analyze* the energy consumption of ROS 2 topic-based communication mechanisms *for the purpose of* understanding and improving their efficiency *with respect to*

message design, communication architecture, and deployment decisions *from the viewpoint* of ROS 2 developers and researchers in the context of robotic software systems.

This research goal is unfolded in the following **research questions**:

**RQ1:** *What is the nature of active ROS 2 projects on GitHub?*
This RQ helps map the current state of ROS 2 development by identifying what types of systems, domains, and technologies active projects represent. This also helps us to understand the real impact of the findings by considering real applications.

**RQ2:** *How are ROS 2 topic message types distributed across the studied projects?*
This RQ aims to understand which message types are most prevalent across distinct robotic domains.

**RQ3:** *What is the impact of different message types on the energy efficiency of ROS 2 nodes?*
This RQ investigates whether certain ROS 2 message types inherently demand more energy for being processed.

- **RQ3.1:** *How does energy consumption scale in topics with different message types?*
  This RQ focuses on the scaling behavior with the number of clients, publishing interval, and message size.
- **RQ3.2:** *How does the architecture of publisher nodes affect other ROS nodes?*
  This RQ explores the importance of designing efficient publishers, as subscribers directly depend on their behavior—particularly in message size and publishing interval.

Table 1 describes the **metrics** used to answer the RQs, which are categorized into 2 main groups: *resource usage*, measured during experiment execution (Section 6); and *repository characteristics*, mined with the GitHub API (Section 5, phase *i*).

**Table 1** Research metrics.

| Metric | Unit | Description | RQ |
|---|---|---|---|
| | | **Resource Usage** | |
| CPU usage | % | Average CPU usage during execution. | |
| RAM usage | KB | Average memory usage during execution. | *RQ 3* |
| Power | W | Average power consumption during execution. | |
| | | **Repository Characteristics** | |
| Creation | Date | The date the repository was created. | *All RQs* |
| Last update | Date | The date of the last commit to the repository. | |
| Size | KB | The total amount of data of the repository. | |
| Forks | # | The number of forks from the repository. | |
| P. language | Category | The language in which nodes are implemented. | |

*Resource usage* metrics focus on *CPU*, *memory usage*, and *power*. Among these, *CPU* is widely recognized as the primary factor influencing software energy consumption [19, 29, 32]. The impact of *memory usage* remains less clear [25, 32], but it is nonetheless a critical resource in algorithm execution. Lastly, *power* represents the energy required to run the ROS nodes.

*Repository characteristics* capture project activity, codebase size, popularity, and collaboration. *Qualitative attributes* describe the robot type, manipulation capabilities, system cardinality, project goal, and application field. *Topic information* is extracted from source code to characterize communication patterns.

## 4.1 Analysis of Experiment Results

The RQs are addressed through a careful analysis of the experimental results, supported by appropriate statistical tests. The process begins by verifying the assumptions required for parametric testing, namely normality and homogeneity of variances. When both assumptions are satisfied, parametric tests are applied; otherwise, non-parametric alternatives are used.

Finally, whenever statistically significant differences are identified, post-hoc analyses are conducted to determine which groups differ while controlling for multiple comparisons.

## 5 Research Approach

In this section, we present the workflow of the research, which follows an empirical methodology combining systematic repository mining, descriptive and qualitative analysis, and controlled experiments. Figure 2 illustrates the workflow, divided into 5 phases: *(i) initial search*, *(ii) qualitative analysis*, *(iii) message extraction*, *(iv) experimentation*, and *(v) profiling and guidelines*.

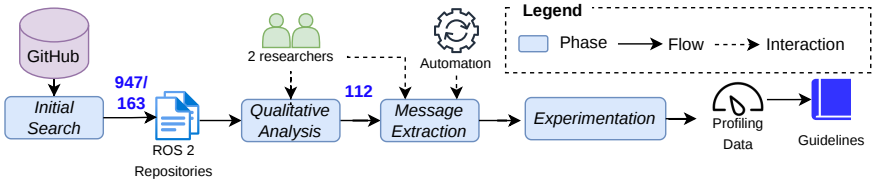



**Figure 2** Research workflow.

***(i)* Initial Search**: To create a representative dataset of ROS 2 projects, we crawl GitHub repositories using its REST API [9] with Query 1, where keyword `ros2` is used to select only repositories that contain it in their name, description, or topics. The resulting list is representative and captures repositories with notable engagement (up to 566 forks), even though it excludes those indexed under alternative keywords like `robotics`. This search yields 947 repositories, which are further filtered to retain only the most relevant ones in terms of *size*, *engagement*, and *activeness*. The `size:>4000` filter selects repositories with more than 4000 kilobytes of code and content, intended to exclude very small repositories that are unlikely to contain a complete robotics application. The `forks:>10` condition ensures that only repositories with at least 10 forks are retrieved, used as a conservative indicator of community reuse or interest. The `is:public` filter limits the search to public repositories. The `pushed:>2024-01-01` criterion ensures the projects have been updated recently (within the previous year of the mining), implying ongoing development or maintenance. Finally, `archived:false` excludes repositories that have been archived and are no longer actively maintained. This query resulted in 163 potential ROS 2-related repositories.

**Query 1** Query for ROS 2 robotics software repositories.

```
ros2 size:>4000 forks:>10 is:public pushed:>2024-01-01
    archived:false
```

***(ii)* Qualitative Analysis**: In this phase, two independent researchers qualitatively classify each repository based on categories that reflect the nature of the robotic projects:

- *Robot's domain*: Specifies the environment in which the robot is intended to operate, *i.e.*, *grounded*, *aerial*, or *aquatic*. Projects without a specific domain are marked as *generic*.
- *Manipulation*: Indicates whether the robot in the project is designed to use manipulators, such as robotic *arms* or *grippers*.
- *Cardinality*: Describes whether the project involves a *single* robot or a *multi-robot* system.
- *Main goal*: Represents the primary objective of the project, such as *education*, *simulation*, *navigation*, and so on.

• *Robotic field*: Refers to the broader application domain the robot is intended for, such as *industrial automation*, *autonomous vehicles*, or *agriculture*, if any.

During the classification process, each repository was thoroughly examined by analyzing the repositories' structure, description, and `README` files. In this phase, 51 repositories were discarded based on the following exclusion criteria, leaving a total of 112 repositories retained for further analysis:

• *No ROS package*: Projects that do not contain the minimal structure required to be recognized as a ROS package. • *Not in English*: Documentation in other languages prevents reproducibility and validation by the broader community. • *Documentation only*: Projects that consist exclusively of documentation, without a ROS codebase. • *Tool for ROS*: Tools for ROS development (*e.g.,* simulators), not associated with a specific robotics application. • *Educational projects*: Projects intended for teaching purposes that are not targeted at a specific robotics domain/field.

The process started with a random sample of 20 repositories, which both researchers classified independently and in parallel. The simple agreement rate for each classification category was then calculated. Across all categories, an agreement of at least 84% was achieved, demonstrating strong alignment between the researchers' analyses. After resolving any discrepancies, each researcher independently analyzed half of the remaining repositories. Finally, to mitigate potential bias, each researcher reviewed and validated the other's classifications.

***(iii)*** **Message Extraction**: In this phase, detailed information on how topic-based message exchanges are implemented is extracted for the selected repositories. This process is largely automated using custom Python algorithms, which identify and extract the *message types* and *publishers* and *subscribers* associated with each topic. The algorithm scans the repositories' codebase, locating files with calls to `create_publisher` and `create_subscription`. Next, each file translates to an Abstract Syntax Tree (AST), enabling the extraction of *message types*, topic names, and subscription points.

We mined a total of 354 distinct *message types* from various message description packages, the most frequent being `sensor_msgs`, `geometry_msgs`, and `std_msgs`. Regarding the number of *subscribers*, instances of 1, 2, 4, 5, 6, and 10 subscribers per topic were observed, where 1 subscriber is the most common configuration, with only 30 projects featuring multiple subscribers per topic (2 subscribers being the most popular among those). Additionally, 50 projects with *orphan topics* were identified; that is, topics with only publishers or only subscribers. This is expected due to the modular development approach of ROS.

Extracting the *message interval* poses additional challenges, as it can be specified in multiple ways, either directly in the source code or through external configuration files. Due to this variability, automated extraction is limited, and manual inspection is employed, searching for calls to ROS primitives related to timers and rates. The two researchers inspected a subset of 26 ROS 2 repositories, identifying 21 distinct message intervals, ranging from every 10 seconds to every 2 milliseconds. These are already distinct time intervals that can provide consistent scalability insights.

***(iv)*** **Experimentation**: We implemented *publisher* and *subscriber* nodes that can be executed with parameters from previous phases as arguments. In Sections 6.1 and 6.2, we detail their implementations. We then orchestrated an experiment that invokes these nodes with parameters corresponding to independent variables. Section 6 details this experiment's design and execution.

***(v)*** **Profiling and Guidelines**: During the *experimentation* phase, we profile resource usage (Table 1), which informs the research questions. We implement a customized profiler for

CPU and memory usage of each ROS process, using the *psutil* Python library [14]. For *power* profiling, we use PowerJoular [31], a RAPL-based tool that provides real-time energy monitoring for individual processes across hardware components (e.g., CPU, GPU, and memory), and has been employed in several recent works on software energy analysis [19, 30]. These are subsequently translated into 6 *guidelines* (Section 8.3), designed to make the results more accessible and actionable for researchers and practitioners.

## 6 Experiment Design and Execution

The experiments were conducted on a single Desktop Computer with the following specifications: Ubuntu Linux 24.04, 64 GB of RAM, and an AMD Ryzen 5 7535HS CPU, with the resources never saturated during the experiments. We implement three algorithms: *generator*, *publisher*, and *subscriber*. The *generator* produces messages according to the experimental parameters and stores them in shared memory, from which the *publisher* retrieves the messages for transmission. This design isolates the computational overhead of message generation in a separate process. The *publisher* and *subscriber* are implemented as ROS 2 nodes running in a local ROS Jazzy [15] environment, each isolated as independent processes.

For the experiment, from the data collected in Phase *iii*, we carefully select design decisions of ROS topic-based communication as **independent variables**: *message interval*, *message type*, *message size*, and *number of clients*. The *resource usage* metrics are already described in Table 1.

• The *message interval* represents the frequency at which messages are published. We focused on 4 intervals within a range from 0.1–1.0 seconds (avoiding stressing the system). The factor levels were deliberately derived from the most frequently observed configurations in the mined repositories, while constraining the factorial design to 360 configurations and 3.600 experimental runs. For message intervals, the selected values (0.1, 0.2, 0.5, and 1 s) capture the most frequently occurring short-to-moderate publication intervals observed in the analyzed projects.

• The *message type* variable captures a range of widely used ROS message types, grouped by their respective ROS packages. To select a representable but feasible subset of types from the 354 extracted, we ranked them by number of occurrences and selected the 50 most frequent types, which are implemented in 12 distinct ROS 2 packages. Then, we selected the 3 packages with the most message types: `std_msgs`, `geometry_msgs`, and `sensor_msgs`, with 13, 11, and 9 message types in the top 50. Those packages account for more than half of message types found in that subset, with all other message packages counting 3 or fewer message types each. Therefore, the three selected packages can already offer valuable insights into how different message types affect the energy efficiency, with a feasible experimentation time. We then included the 5 most common types from each packages to ensure balanced coverage across groups, resulting in the following:

- `std_msgs` = {`String`, `Int32`, `Float32`, `Float64`, `Float64MultiArray`}
- `geometry_msgs` = { `Vector3`, `Twist`, `PoseStamped`, `PoseWithCovariance`, `PoseWithCovarianceStamped`}
- `sensor_msgs` = {`Image`, `JointState`, `PointCloud2`, `Imu`, `LaserScan`}

• The *message size* was defined at 3 levels: *large* (the maximum payload for each message type), *medium* (half the maximum size) and *small* (one third the maximum size).

• The *number of clients* refers to how many subscriber nodes receive messages from the publisher. We used values of 1 and 2, which correspond to the most common subscriber counts observed in the mined repositories.

## 6.1 ROS 2 Message Publisher

We implemented a flexible ROS 2 *publisher* node in Python designed to generate synthetic messages of various types and sizes, allowing reproducible experiments. The *message type*, *message interval*, and *message size* are configurable at runtime via command-line arguments. Internally, the *publisher*, given the arguments, constructs a synthetic message and periodically publishes it to a ROS 2 topic for a fixed duration (*e.g.,* two minutes), also passed as an argument. The *publisher* (re-)generates the message every time with random data, mimicking a real application, where the message must be processed before publishing (*e.g.,* the *talker* file – `fogros2_examples/fogros2_examples /talker.py` – in repository *R35* [6]). The ROS 2 *message type* packages are loaded dynamically using Python's `importlib`, which avoids unnecessary overhead. Additionally, the *publisher* injects the header timestamp at each publication, avoiding identical messages and activating DDS features such as zero-copy [1], which prevents transmission of the same message in sequence.

Each message type in our *publisher* supports a configurable *message size* parameter that controls the message payload generated. For basic types like `String`, the content consists of repeated phrases, increasing linearly in length. Numeric types, such as `Float32` and `Int32`, use increasingly large values computed as fractions of their respective maximum representations. For structured types like `Imu` and `PoseStamped`, higher size levels incrementally fill in more fields (e.g., $x$ dimension only, then $xy$, and $xyz$ dimensions). For more complex *message types*, we carefully generate data with compatible length regarding the *message size* argument. `Image` messages are synthetic RGB images created using OpenCV [22], with increasing resolutions: 86×86 (∼22KB), 105×105 (∼33KB), and 148×148 (∼66KB). The `Float64MultiArray` message increases the number of double-precision elements up to a total byte size near 64KB. `PointCloud2` messages are 3D point arrays with random data, using up to 64KB in memory by adjusting the number of points. `LaserScan` messages increase the number of range and intensity samples proportionally.

## 6.2 ROS 2 Message Subscriber

We implemented a generic and configurable ROS 2 *subscriber* node that listens to the same topic where the *publisher* publishes messages. The *subscriber* runs until it does not receive any message for a preset timeout period of 5 seconds. It dynamically loads the specified *message type* based on a runtime parameter, allowing flexible support for different ROS 2 message structures and avoiding unnecessary overhead. Upon receiving a message, the node prints its content to the console, effectively mimicking the behavior of a process that unpacks or consumes the data. For example, in project *R653* [13], the *Rpicam_ai_node* node receives a `String` message with a command (*i.e.,* for the camera to take pictures) that must be read and interpreted.

## 6.3 Experiment Orchestrator

We designed a `bash` experiment *orchestrator script* to automate the execution of all experimental configurations. The number of *subscriber* nodes instantiated is determined by the *number of clients* factor, either 1 or 2, running in parallel. The full set of factor combinations

(i.e., the Cartesian product) is stored in a *run_table* Comma-Separated Values (CSV) file, with each configuration repeated 10 times to ensure statistical robustness. Given the identified parameters, there are 360 configurations: 15 *message types* × 4 *message intervals* × 3 *message sizes* × 1–2 *subscribers*. This results in 3600 rows representing distinct runs. The *orchestrator script* sequentially reads each row, launches both the *publisher* and *subscriber* components with the corresponding parameters, and ensures the cleanup of the experimental environment after each run.

# 7 Results

This section presents the demographics (Section 7.1) from our qualitative analysis of the repositories and the subsequent experiment results (Section 7.2).

## 7.1 Projects' Nature

Our search identified projects created between 2019, shortly after the release of modern ROS 2 distributions, and July 2024, when the dataset was built. Although the repositories span a relatively long period, the dataset remains representative, since two of the three actively supported distributions were released before that date (i.e., *Humble*, launched in 2022, and *Iron*, launched in 2023). Furthermore, these ROS 2 distributions already included modern message types. The dataset should still provide a representative view since the study focuses primarily on identifying frequent message types rather than analyzing ROS 2 projects.

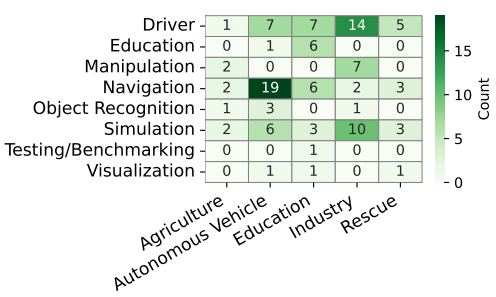
An analysis of the main ***programming language*** in the development of the selected projects indicates that the majority of the studied repositories are developed in C++ (50) and Python (42), which is already expected given that the ROS 2 documentation primarily focuses on these two languages. However, C (5), Elixir (2), and C# (1) are also used, even if they are not largely covered as the first two in the ROS documentation.

Considering projects across specific ***fields of application***, the ones focused on *autonomous vehicles* are the most prevalent. Like those in *rescue* and *agriculture*, these robots are mobile (*e.g., R159* [4] and *R292* [3]) and therefore reliant on battery power, which highlights the relevance of research aimed at reducing power consumption. The distribution also emphasizes the widespread adoption of ROS 2 in practice. We further analyzed application fields that depend on ***manipulators***, such as robotic arms or grippers. All 20 *industrial* projects and 3 out of 4 *agriculture* projects rely on this feature, as does one additional project classified under both *education* and *rescue*. Understanding this dependency is essential to assess the role of message exchange in energy efficiency for actuator devices. Lastly, regarding the ***robot type*** used across fields of application, the vast majority are *grounded robots*, which aligns with the findings by Garcia *et al.* [24]. Only 2 *aquatic* robots are used in *rescue* scenarios, and a single *aerial* robot appears in an *education* project.

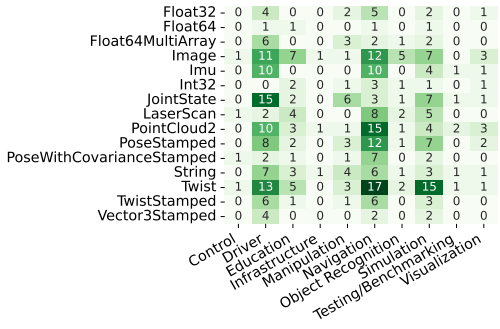
We also measure project ***goals*** popularity by their ***number of forks***. This metric is relevant for our analysis, as it reflects how widely adopted the design of message exchange is for each project type. Among the goals, *navigation* (closely tied to the field of *autonomous vehicles*) emerges as the most popular, followed by *driver* projects, which implement robot-specific drivers, and *simulation*, a key component in validating ROS-based software. Most of the highly forked projects are between 2 and 5 years old. A notable example is *R199* (*autoware.universe* [2]), a *navigation*-related project with over 500 forks and more than 250 contributors. Its popularity highlights the representativeness of our dataset in terms of project

goals. In total, the *navigation* goal accounts for 1816 forks, reinforcing its prominence. Other influential goals include *simulation* and *driver*, which are critical to the ROS development pipeline. The most forked *driver* project is *R9* (*Universal_Robots_ROS2_Driver* [17] – 186 forks), which provides ROS 2 support for the CB3 and e-Series robotic arms from Universal Robots. The leading *simulation* project is *R462* (*linorobot2* [11] – 149 forks), offering a complete Gazebo-based simulation pipeline for various robot configurations, including 2WD, 4WD, and Mecanum drive systems. Some goals like *infrastructure* are underrepresented, with a single repository.

Figure 3a presents a heatmap illustrating the relationship between robotic ***goals*** and ***fields of application***. Vertically, we observe that most of the heat points are concentrated in the *industry* and *autonomous vehicle* domains. Within the *industry* field, the most common project goals are *driver*, *simulation*, and *manipulation*. The *driver* goal reflects the frequent use of diverse robotic platforms, necessitating dedicated ROS packages. The *simulation* goal represents a key step in robotic software development, enabling virtual testing before deploying on physical robots. The emphasis on *manipulation* is tied to the nature of industrial robots, which typically rely on actuator systems. In the case of *autonomous vehicles*, the dominant goal is *navigation*, which aligns with the core functionality of such systems. There is also a strong association with the *driver* and *simulation* goals, with the same motivations found in the industrial domain. We can also observe that the goals *driver*, *navigation*, and *simulation* are connected to every field of application in at least one instance, highlighting their general relevance across robotics projects.




**(a)** Heatmap of the correlation between project goal ($y$ axis) and field of application ($x$ axis).




**(b)** Correlation of *message types* and *project goals*.

**Figure 3** Relationships between project goals, application fields, and message types.

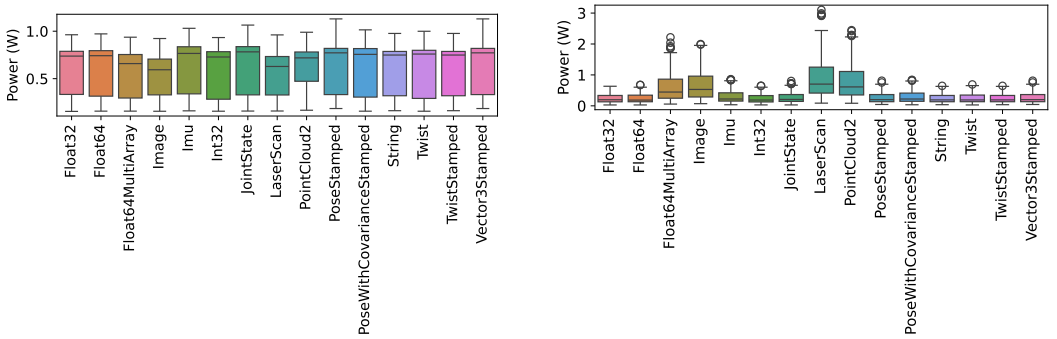
Figure 3b illustrates the relationship between the selected ***message types*** and the corresponding project ***goals***. The most frequent goals, *navigation*, *driver*, and *simulation*, are associated with all message types. The `Twist` message type stands out as widely used across all 3 goals, highlighting its relevance in various robotic applications. In *driver* projects, `JointState` is the most common message type, which aligns with the typical focus on manipulator devices such as *robotic arms* (e.g., *R31* – *franka_ros2* [7]). This observation also reinforces the trend shown in Figure 3a, where *driver* projects are strongly tied to the *manipulation* field. In *navigation* projects, `PointCloud2` is the second most common message type after `Twist`, followed by `Image` and `PoseStamped`. These messages are essential for tasks like mapping the environment and path planning. For *simulation* projects, `Twist` remains dominant, while message types `Float32`, `Float64`, and `Int32` appear less frequently, each with a single occurrence.

Finally, we analyze the ***cardinality*** of robotic systems across the studied projects, as it

can indicate the degree of inter-robot communication. Despite only 6 of the projects being explicitly designed for multi-robot setups, during message extraction (Section 5, phase *iii*), we spot a dependency on external components (given the orphan topics), which reflects a modular design approach, in which ROS 2 packages are intended to be integrated into larger systems. These findings underscore the importance of examining the impact of multiple *subscribers* on a single topic, as project dependency through topic-based communication appears to be a consistent pattern.

## 7.2 Experiment Results

The experiments ran for a total of ≈150 hours/6.25 days: 360 combinations × 2.5 minutes each execution × 10 repetitions. Here we analyze the results of those executions. We start by investigating the impact of message types on the energy efficiency of both *publisher* and *subscriber* nodes. We then show results regarding the scalability of energy consumption for each of those message types.




**(a)** *Publisher* node average power.

**(b)** *Subscriber* node average power.

**Figure 4** Average power by message type for *publisher* and *subscriber* nodes.

**Power Consumption by Message Types:** Figure 4 shows the power consumption distribution across multiple executions of each message type for both the *publisher* and the *subscriber*, considering different *message intervals* and *sizes*. For the publisher (Figure 4a), the power consumption shows high variability across message types (given the different *message intervals* and *sizes*), yet their median values are close, ranging from 0.6 W to 0.8 W, suggesting that the power required by each message type is comparable. The *subscriber* (Figure 4b) presents a different profile, with four message types standing out as more energy-intensive than the others: `Float64MultiArray`, `Image`, `LaserScan`, and `PointCloud2`. Those message types also produce the highest outlier values, meaning that, in some runs, they consumed exceptionally more energy (around 1.5 W more than the median) than typical. This is an expected behavior since they are larger and more complex messages that, as a consequence, may require more effort to be read. Additionally, the relative consumption of these message types differs from what is observed in the publisher, suggesting that the energy impact of publishers and subscribers should be considered independently during system design.

A similar pattern is observed for CPU utilization, as the power metric primarily reflects CPU power consumption. However, the CPU utilization varies far less than the power because these metrics capture fundamentally different aspects of execution. CPU utilization reflects how busy the processor is, but it does not indicate how intensive or costly that work is in terms of voltage, frequency, or memory activity. In contrast, power and energy measurements

capture the actual electrical demand required to execute that workload. Another point is that, although one might expect higher CPU utilization to correspond directly to higher energy consumption, our results show that power does not scale proportionally with CPU usage. Nonetheless, although the difference between the two metrics is present, it remains relatively small. Such a discrepancy arises from the behavior of modern processors, which rely on sophisticated power-management mechanisms, such as Precision Boost 2 performance enhancement[2], that break any simple linear relationship between processor activity and energy consumed. The experiment factors do not affect memory usage; every execution using around 2772KB on average.

• *Statistical tests*: The statistical analysis revealed a significant overall difference between the *publisher* and *subscriber* groups. Post-hoc test demonstrate that the groups differ substantially from one another, with an extremely small p-value ($p = 7.67 \times 10^{-75}$), far below conventional significance thresholds. This indicates a robust and highly significant difference between the two distributions.

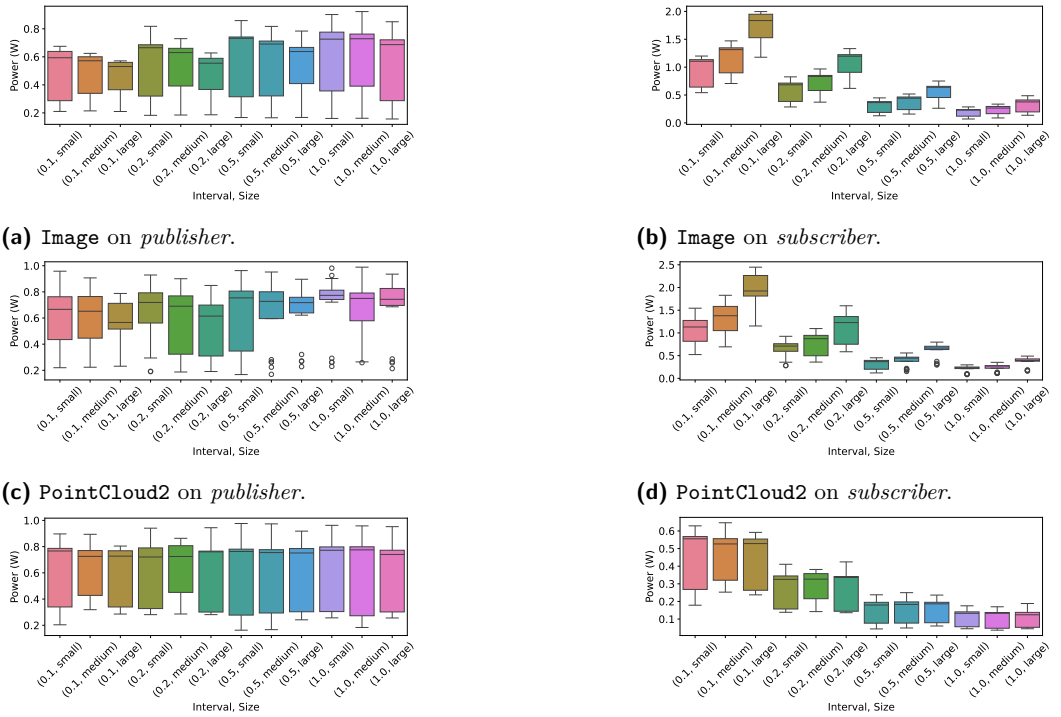
Grouping messages by their empirically observed energy demand further reveals strong within-group differentiation. On the *subscriber* side, the analysis of low energy–demanding messages showed no statistically significant differences across message types according to the Kruskal–Wallis test ($H = 12.83$, $p = 0.076$). Despite this, Welch's ANOVA indicated a significant effect ($F = 57.77$, $p = 2.22 \times 10^{-16}$), suggesting sensitivity to differences in group means under heteroscedasticity. In contrast, the high-energy– demanding messages exhibited clear differences across groups, as confirmed by the Kruskal–Wallis test ($H = 24.77$, $p = 1.72 \times 10^{-5}$). Post-hoc Dunn's tests showed that `LaserScan` differed significantly from both `Float64MultiArray` ($p = 1.9 \times 10^{-5}$) and `PointCloud2` ($p = 0.00135$), while other pairwise comparisons were not significant. These findings were further supported by Welch's ANOVA ($F = 29.47$, $p = 2.76 \times 10^{-10}$), confirming substantial performance variation among high-energy message types.

On the *publisher* side, we consider the same groups as in the *subscriber*, where low and high energy message types lead to differences across message categories, particularly pronounced effects for high energy message types. Kruskal–Wallis test for the high energy–demanding messages indicates a significant differences ($H = 28.65$, $p = 2.65 \times 10^{-6}$). Dunn's post-hoc analysis showed that `PointCloud2` displayed robust differences from several other message types, including `Float64MultiArray` ($p = 0.00157$), `Image` ($p = 3.5 \times 10^{-5}$), and `LaserScan` ($p = 1.09 \times 10^{-4}$). These strong differences were further confirmed by Welch's ANOVA ($F = 106.90$, $p \approx 0$).

**Scalability of Message Types:** Figure 5 presents the energy measurements for 3 message types under varying *message sizes* and *intervals*. For a matter of comparison, we illustrate `Image` (which shows high energy demand only on the *subscriber* side), `PointCloud2` (consistently demanding on the *publisher* side), and `Twist` (a widely used standard message type – see Figure 3b). For all message types, the power consumption on the *publisher* side is slightly influenced by *message interval*, while *message size* shows no consistent effect. Despite exhibiting high and discrepant energy demand for the *subscriber*, all *publisher* executions show a comparable power consumption level to `Twist`. This is explained by the fact that the *publisher* is only responsible for packing the message, while the *subscriber* needs to sequentially read it. For demanding types, i.e., `Image` (Figure 5b) and `PointCloud2` (Figure 5d), both *message size* and *interval* impact *subscriber* power. In contrast, for lightweight types such as `Twist` (Figure 5f), only *message frequency* appears to influence energy usage, possibly

[2] `https://www.amd.com/en/resources/support-articles/faqs/CPU-PB2.html`

because even large messages of those types are not very complex.




**(a)** `Image` on *publisher*.

**(b)** `Image` on *subscriber*.

**(c)** `PointCloud2` on *publisher*.

**(d)** `PointCloud2` on *subscriber*.

**(e)** `Twist` on *publisher*.

**(f)** `Twist` on *subscriber*.

**Figure 5** Scalability of different message types on *publisher* and *subscriber* nodes, with different *message sizes* and *intervals*.

We do not observe a difference in the *publisher* power consumption means when increasing the **number of clients** from 1 to 2, a pattern consistent across all other message types. This finding aligns with results from a previous study focused solely on `String` message types [19]. In that work, there was no significant increase in power consumption for a *publisher* implemented in Python at high message frequencies. This behavior is likely due to the ROS 2 architecture, in which messages are dispatched to the DDS layer that manages delivery [37], effectively decoupling the sending process and preventing additional load on the *publisher* node.

• *Statistical tests*: When analyzing the impact of different *message intervals*, none of the *message types* exhibit normally distributed data. Whelch's ANOVA test reveals statistically significant differences in the power consumption of complex messages, such as `Image` and `PointCloud`, on the *publisher* side: `Image` ($F = 34.59$, $p < 0.001$), and `PointCloud2` ($F = 31.41$, $p < 0.001$). For simple messages, such as `Twist`, Welch's ANOVA indicates no statistically significant difference in power consumption across message intervals ($F = 0.23$, $p = 0.113$). For the *subscriber*, the *message interval* is not normally distributed either. Welch's ANOVA also confirms significant differences across intervals, including for `Twist` ($F = 26.15$, $p < 0.001$), `Image` and `PointCloud2` consistently show non-normal distributions and significant interval-based effects ($F = 626.09$ and $F = 1{,}425.23$, respectively). These results indicate that *message interval* significantly impacts power consumption for both nodes, especially for complex message types.

Considering *message size*, *publisher*-side power consumption shows no significant variation: `Image` ($F = 6.00$, $p =< 0.001$), `PointCloud2` ($F = 8.04$, $p =< 0.001$), and `Twist` ($F = 0.21$,

$p < 0.001$). On the *subscriber* side, all message types also present non-normal distributions. Welch's ANOVA detects significant effects for `Image` ($F = 33.34$, $p < 0.001$) and `PointCloud2` ($F = 46.96$, $p < 0.001$), whereas `Twist` shows negligible practical impact despite statistical significance ($F = 0.65$, $p = 0.032$). Finally, the *number of clients* does not significantly affect the publisher's power consumption, confirming that this factor is not energy-relevant.

# 8 Discussion

In this section, we interpret the main findings of our study, answering the RQs from Section 4 and discussing implications considering the studied projects.

## 8.1 Answers to the Research Questions

• **ARQ1** – *What is the nature of active ROS 2 projects on GitHub?*

Our analysis of 112 GitHub repositories reveals a diverse and active ROS 2 ecosystem. Most projects are implemented in C++ (50) and Python (42), aligning with the most common client libraries in ROS 2 documentation. These projects cover a range of robotic domains, including autonomous vehicles, industrial automation, agriculture, rescue, and education, where grounded robots are the most common robot type. Popular project goals include *navigation*, *driver development*, and *simulation*. The dominance of projects such as *autoware.universe* [2] highlights the representativeness of our dataset and underscores the real-world relevance of our communication design analysis.

We further observe that project goals are strongly associated with specific message types. For example, `Twist` is widely used across navigation, simulation, and driver projects, while `JointState` is particularly common in driver projects associated with manipulators. Similarly, `PointCloud2`, `Image`, and `PoseStamped` are strongly tied to navigation projects, reflecting the importance of sensing and localization in autonomous systems. This indicates that the choice of message type and communication design factors is dependent on the application domain and deployment environment, especially in collaborative or highly dynamic scenarios.

• **ARQ2** – *How are ROS 2 message types distributed across the studied projects?*

We identified 354 unique message types across the analyzed repositories, with the majority belonging to 3 core ROS 2 packages: `sensor_msgs`, `geometry_msgs`, and `std_msgs`. Frequently used message types such as `Twist`, `Image`, `PointCloud2`, and `JointState` appear across several domains and project goals.

The high number of *message types* reinforces the relevance of studies such as ours, since developers must choose from a very large design space. Another important observation is that many commonly used *message types* are stamped variants, such as `PoseStamped`, `TwistStamped`, and `PoseWithCovarianceStamped`, suggesting a preference for timestamped data to support synchronization and sensor fusion. This adds an extra dimension to the *message type* selection process.

The presence of deprecated messages such as `PointCloud` in current projects also suggests technical debt and emphasizes the importance of energy profiling to guide message deprecation and replacement. Overall, these findings reinforce the need for careful message type selection to avoid both energy inefficiencies and long-term maintenance issues.

• **ARQ3** – *What is the impact of different message types on the energy efficiency of ROS 2 nodes?*

Our experiments show that *message type* significantly impacts energy consumption, particularly on the *subscriber* side, while *publishers* exhibit relatively stable power consumption.

This is attributed to the computational effort required to deserialize and process large and structured message payloads. Results suggest that the complexity and structure of message types are critical factors for energy-aware system design. Lightweight messages such as `String`, `Int32`, `Float32`, and `Float64` demonstrate minimal energy impact, making them suitable for low-power applications and resource-constrained deployments.

All *message types* associated with higher power consumption belong to the `sensor_msgs` package. In particular, `Image`, `PointCloud2`, `LaserScan`, and `Float64MultiArray` stand out as the most energy-demanding types on the subscriber side. This outcome is expected, as these messages are typically rich in data, enabling the robot to perceive and interact with its environment. Consequently, sensor messages play a critical role in the design of ROS 2 topic communication.

• **ARQ3.1** – *How does energy consumption scale with topics using different message types?*

We observed that energy usage scales differently depending on *message type*, *size*, and *interval* (frequency). For simple messages, variations in these parameters have negligible impact on power consumption, except for *message interval* on the *subscriber* side. However, consumption increases significantly with higher frequencies and larger message sizes for heavy types such as `PointCloud2` and `Image`. This effect is especially pronounced for subscriber nodes, while publisher nodes remain comparatively stable.

Our results further indicate that for complex message types, *message interval* has a stronger impact in the node power. For demanding types such as `Image` and `PointCloud2`, both interval and size influence subscriber-side energy consumption. In contrast, lightweight messages are affected mainly by frequency, while message size has only a limited effect.

Considering the studied projects, this is a relevant finding. An example is an *issue* opened by a follower of project *R284* [12] asking for increasing point cloud frequency from 2 to 10 hertz [10]. This suggests that some ROS 2 users may not be aware of the energy impact of such communication design decisions.

• **ARQ3.2** – *How does the architecture of publisher nodes affect other ROS nodes?*

The number of *clients* does not significantly influence the *publisher*'s power consumption, supporting the assumption that the DDS middleware offloads delivery overhead. This decoupling allows *publishers* to maintain a stable energy profile, even with multiple *subscribers*.

However, the architectural choice of allowing multiple *subscribers* to a single topic has implications for overall system energy consumption. In practice, each additional *subscriber* adds overhead to the system, particularly when processing complex messages, amplifying total energy consumption. Our mining identified 30 projects with multi-client topics, with some of them potentially running on the same machine. An example is *R101* [8], where the topic `diffbot/odom` has two subscribers, *OdometrySubscriber* and *RobotController*, which are part of the same package (`py_action_pkg`).

This enables different optimization strategies. For instance, instead of two nodes on the same machine reading a common topic, they could be designed to exchange data via memory sharing, a native feature of modern ROS 2 distributions via *component* composition [5].

## 8.2 Impact of the Findings Across the Studied Projects

The observed differences in energy consumption are particularly relevant when interpreted considering the nature of the studied projects. In the following, we discuss the main implications considering them.

• **Autonomous vehicles**: Projects centered on autonomous driving and navigation, such as *R199*, *R528*, *R59*, and *R296*, typically combine multiple sensing, perception, localization, planning, and control pipelines. These projects rely on complex message types such as `Image`,

**Table 2** Guidelines for Energy-Aware Design ROS 2 Topic-Based Communication

| Guideline | Description |
|---|---|
| **G1** – *Simplify/compress messages* | Complex message types should be simplified, compressed, or transmitted at lower resolution whenever possible. |
| **G2** – *Minimize publishing frequency* | Publish intervals should be tuned according to the robot application and context, avoiding frequent updates. |
| **G3** – *Limit unnecessary subscribers and topics* | Avoiding redundant subscriptions, disabling unused sensors, and reducing unnecessary debug topics. |
| **G4** – *Use intra-process communication* | When possible, use ROS 2 composition, shared memory, or zero-copy communication to reduce communication overhead. |
| **G5** – *Profile energy usage and adapt to deployment* | Integrate RAPL-based tools (e.g., PowerJoular) into CI pipelines to detect energy-efficient alternatives early. |
| **G6** – *Refactor deprecated or inefficient msg. types* | Replace legacy or overly complex custom message types with more efficient and well-maintained alternatives. |

`PointCloud2`, `Odometry`, and `PoseStamped`, which tend to increase communication overhead and consequently energy consumption. This is especially important in such applications, where multiple sensors operate simultaneously and require continuous real-time updates.

• **Manipulators/Cobots**: Manipulation-oriented projects such as *R9*, *R31*, *R27*, and *R69* mainly exchange compact control-oriented messages, including `JointState`, `JointTrajectory`, `TwistStamped`, and vendor-specific status messages. These projects usually involve fewer active publishers and subscribers and smaller payload sizes, which reduces communication costs and makes their energy profile more stable. Therefore, experimental results suggest that manipulation systems are less sensitive to message complexity than navigation ones.

• **Environment perception**: Projects involving perception and object recognition, such as *R50*, *R95*, and *R143*, are strongly influenced by image-based message types. In dynamic environments, it may result in substantial data movement between nodes. Even if the number of topics is relatively small, the large payload size of image streams can produce high energy consumption. This reinforces the importance of efficient serialization, compression, and zero-copy communication strategies for perception-heavy ROS 2 applications.

• **Aggressive update rates**: Our partial analysis of topic frequency configuration across the studied projects reinforces the relevance of the energy consumption experiments. Most repositories adopt communication rates between 1 and 50 Hz (five times the max rate we experimented). Examples include controller frequencies around 20 Hz, local costmap updates between 2 and 5 Hz, and waypoint followers operating at 20 or 30 Hz. We also identified examples of controllers, IMU pipelines, and telemetry systems operating at 100, 200, 400, or even 500 Hz. This suggests that some of the projects may unknowingly (or by design) introduce considerable energy overhead through aggressive update rates. Therefore, results make ROS developers aware of the critical aspects of high update rates in communication.

• **Widespread message types**: Messages such as `Twist`, `Odometry`, `Image`, `PointCloud2`, and `JointState` appear repeatedly across projects of different nature. Therefore, their widespread adoption suggests that any optimization applied to these common message types could have a broad impact on the energy efficiency of ROS 2 systems as a whole.

## 8.3 Towards Guidelines for Energy-Aware ROS 2 Communication

Based on the experimental results summarized in Table 2, we propose a preliminary practical *guidelines* for energy-aware ROS 2 communication design. All of them being previously discussed in the paper.

## 9 Threats to Validity

**Internal validity:** In our experiments, *subscriber* nodes print the received messages to simulate data consumption. This behavior may increase CPU usage, particularly for large and structured messages (e.g., vectorized `Image` types). However, this design choice reflects realistic usage patterns. For instance, when handling `Image` messages, a *subscriber* often needs to traverse the entire data vector and perform operations such as rendering or object recognition tasks, which can be more computationally intensive than printing. To minimize variability and isolate effects, we execute each configuration in separate Docker containers and repeat it 10 times. We also prevent message generation from interfering with the *publisher* measurements by separating its process into an isolated container.
**External validity:** Our experiment includes a subset of the 354 message types mined from repositories, selected from the 3 most frequently used ROS 2 packages, representing a diverse range of structural complexity and data payload. This selection ensures coverage of typical communication patterns found in real-world ROS 2 systems. Therefore, although not exhaustive, our sample is representative enough to deliver meaningful insights into the energy behavior of commonly used message types. Additionally, all experiment design choices were mined from real projects, which makes them representative. Another potential threat arises from using synthetic messages instead of real sensor data. For instance, `Image` messages were generated at a low resolution (∼64KB), underestimating the energy cost of high-resolution cameras. Likewise, some messages rely on NumPy-generated data, possibly introducing computational bias in packing the messages. However, our primary goal is to compare configurations in a controlled environment where relative trends remain valid and meaningful.
**Conclusion validity:** We use appropriate statistical tests depending on the distribution and variance assumptions, ensuring the reliability of our conclusions. To mitigate biased data interpretation, we provide a replication package with all scripts, datasets, and experiment configurations. This enables the replication or extension of our results with minimal effort.

## 10 Conclusion

This paper empirically investigated the energy implications of ROS 2 topic-based communication. Our results show that design choices in message exchange, particularly type and frequency, strongly affect energy consumption, with subscribers bearing most of the cost. Rich messages (e.g., from `sensor_msgs`) demand considerably more power, while lightweight messages are more efficient. Message interval proved the dominant factor, underscoring the importance of tuning this parameter. The proposed guidelines can support more energy-aware design in ROS 2 software. Future work focus on real sensor data, the DDS layer as a factor (experimenting with additional implementations), and different hardware architectures.

## Acknowledge

In this document, we only used AI (ChatGPT) to improve parts of the writing, especially relating to grammar corrections. All the AI changes were supervised by the authors.

## References


1. 15.8. zero-copy communication - 3.2.1. Accessed on: May 04, 2025. URL: https://fast-dds.docs.eprosima.com/en/latest/fastdds/use_cases/zero_copy/zero_copy.html.
2. autoware.universe. GitHub repository. URL: https://github.com/autowarefoundation/autoware.universe.
3. botanbot_sim. GitHub repository. URL: https://github.com/NMBURobotics/botanbot_sim.
4. clydemcqueen. GitHub repository. URL: https://github.com/clydemcqueen/orca4.
5. Composition — ROS 2 Documentation: Jazzy documentation. URL: https://docs.ros.org/en/jazzy/Concepts/Intermediate/About-Composition.html.
6. Fogros2. GitHub repository. URL: https://github.com/HaiderAbasi/ROS2-Path-Planning-and-Maze-Solving.
7. franka_ros2. GitHub repository. URL: https://github.com/frankaemika/franka_ros2.
8. gcamp_ros2_basic. GitHub repository. URL: https://github.com/Road-Balance/gcamp_ros2_basic.
9. GitHub REST API documentation. URL: https://docs-internal.github.com/en/rest?apiVersion=2022-11-28.
10. Increase frequency · Issue #7 · stm32f303ret6/livox_laser_simulation_ro2. URL: https://github.com/stm32f303ret6/livox_laser_simulation_RO2/issues/7.
11. linorobot2. GitHub repository. URL: https://github.com/linorobot/linorobot2.
12. livox_laser_simulation_ro2. GitHub repository. URL: https://github.com/stm32f303ret6/livox_laser_simulation_RO2.
13. Nindamani-the-weed-removal-robot. GitHub repository. URL: https://github.com/AutoRoboCulture/Nindamani-the-weed-removal-robot.
14. psutil: Cross-platform lib for process and system monitoring. URL: https://github.com/giampaolo/psutil.
15. Ros 2 documentation - ros 2 documentation: Jazzy documentation. Accessed on: May 04, 2025. URL: https://docs.ros.org/en/jazzy/index.html.
16. ROS: Home. URL: https://www.ros.org/.
17. Universal_robots_ros2_driver. GitHub repository. URL: https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver.
18. ros2/rcl, September 2025. original-date: 2015-02-21T00:06:40Z. URL: https://github.com/ros2/rcl.
19. Michel Albonico, Manuela Bechara Canizza, and Andreas Wortmann. Energy efficiency in ROS communication: A comparison across programming languages and workloads. *Frontiers in Robotics and AI*, 12:1548250.
20. Michel Albonico, Ivano Malavolta, Gustavo Pinto, Emitza Guzman, Katerina Chinnappan, and Patricia Lago. Mining energy-related practices in robotics software. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 483–494. IEEE, 2021.
21. Michel Albonico, Paulo J. Varela, Adair José Rohling, and Andreas Wortmann. Energy efficiency of ROS nodes in different languages: Publisher-subscriber case studies. In *RoSE*, pages 1–8. IEEE, 2024.
22. Gary Bradski, Adrian Kaehler, et al. Opencv. *Dr. Dobb's journal of software tools*, 3(2), 2000.
23. Katerina Chinnappan, Ivano Malavolta, Grace A. Lewis, Michel Albonico, and Patricia Lago. Architectural tactics for energy-aware robotics software: A preliminary study. In *ECSA*, volume 12857 of *LNCS*, pages 164–171. Springer, 2021.
24. Sergio García, Daniel Strüber, Davide Brugali, Thorsten Berger, and Patrizio Pelliccione. Robotics software engineering: a perspective from the service robotics domain. ESEC/FSE 2020, page 593–604, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3368089.3409743.

**25** Philipp Gschwandtner, Michael Knobloch, Bernd Mohr, Dirk Pleiter, and Thomas Fahringer. Modeling CPU energy consumption of HPC applications on the IBM POWER7. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 536–543, 2014. doi:10.1109/PDP.2014.112.

**26** Saeid Jamshidi, Amin Nikanjam, Kawser Wazed Nafi, and Foutse Khomh. Understanding the impact of iot security patterns on CPU usage and energy consumption: a dynamic approach for selecting patterns with deep reinforcement learning. *Int. J. Inf. Sec.*, 24(2):91, 2025.

**27** Kashif Nizam Khan, Mikael Hirki, Tapio Niemi, Jukka K Nurminen, and Zhonghong Ou. Rapl in action: Experiences in using rapl for power measurements. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)*, 3(2):1–26, 2018.

**28** Foutse Khomh and S. Amirhossein Abtahizadeh. Understanding the impact of cloud patterns on performance and energy consumption. *J. Syst. Softw.*, 141:151–170, 2018.

**29** Ivano Malavolta, Katerina Chinnappan, Stan Swanborn, Grace A. Lewis, and Patricia Lago. Mining the ROS ecosystem for green architectural tactics in robotics and an empirical evaluation. In *MSR*, pages 300–311. IEEE, 2021.

**30** Felix Nahrstedt, Mehdi Karmouche, Karolina Bargieł, Pouyeh Banijamali, Apoorva Nalini Pradeep Kumar, and Ivano Malavolta. An empirical study on the energy usage and performance of pandas and polars data analysis python libraries. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, page 58–68, New York, NY, USA, 2024. Association for Computing Machinery. doi:10.1145/3661167.3661203.

**31** Adel Noureddine. PowerJoular and JoularJX: Multi-platform software power monitoring tools. In *Intelligent Environments*, pages 1–4. IEEE, 2022.

**32** Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *SLE*, pages 256–267. ACM, 2017.

**33** Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: An open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

**34** André Santos, Alcino Cunha, Nuno Macedo, Rafael Arrais, and Filipe Neves dos Santos. Mining the usage patterns of ROS primitives. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2017, Vancouver, BC, Canada, September 24-28, 2017*, pages 3855–3860. IEEE, 2017. doi:10.1109/IROS.2017.8206237.

**35** Stan Swanborn and Ivano Malavolta. Energy efficiency in robotics software: a systematic literature review. In *ASE Workshops*, pages 144–151. ACM, 2020.

**36** Rini Van Solingen, Vic Basili, Gianluigi Caldiera, and H Dieter Rombach. Goal question metric (GQM) approach. *Encyclopedia of software engineering*, 2002.

**37** Yuqing Yang and Takuya Azumi. Exploring real-time executor on ROS 2. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2020. doi:10.1109/ICESS49830.2020.9301530.