Internet of Things 34 (2025) 101803

Contents lists available at ScienceDirect

Internet of Things

journal homepage: www.elsevier.com/locate/iot

A comparison of code quality metrics and best practices in non-IoT and IoT systems

Nour Khezemi[a], Sikandar Ejaz[b,*], Naouel Moha[a], Yann-Gaël Guéhéneuc[c]

[a] *Département de génie logiciel et des TI, École de technologie supérieure, H3C 1K3, Montreal, QC, Canada*
[b] *Concordia Institute for Information Systems Engineering (CIISE), Concordia University, H3G 1M8, Montreal, QC, Canada*
[c] *Department of Computer Science and Software Engineering (CSSE), Concordia University, H3G 1M8, Montreal, QC, Canada*

ARTICLE INFO

Keywords:
IoT
Code quality
Metrics
Comparison

ABSTRACT

IoT systems are a network of connected devices powered by software, requiring the study of software quality for maintenance. Despite extensive studies on non-IoT systems' software quality, research on IoT systems' software quality is lacking. It is uncertain whether non-IoT and IoT systems' software are comparable, limiting the application of results and best practices from non-IoT to IoT systems. Therefore, we compare the code quality of two equivalent sets of non-IoT and IoT systems to determine whether there are similarities and differences between the two kinds of software systems. We design and apply a systematic method to select two sets of 94 non-IoT and IoT system software from GitHub with comparable characteristics. We compute quality metrics on the systems in these two sets and then analyse and compare the metric values. We conduct an in-depth analysis and provide specific examples of the IoT systems' complexity and how it manifests in their source code. We conclude that software for IoT systems is more complex, coupled, larger, less maintainable, and cohesive than non-IoT systems. Several factors, such as integrating multiple hardware and software components and managing data communication between them, contribute to these differences. After the comparison, we systematically select and present a list of best practices to address the observed differences between non-IoT and IoT code. We present a list of revisited best practices with approaches, tools, or techniques for developing IoT systems. For example, applying modularity and refactoring are best practices for lowering complexity. Based on our work, researchers can now make informed decisions using existing studies on the quality of non-IoT systems for IoT systems. Developers can use the list of best practices to minimise disparities in complexity, size, and cohesion and enhance maintainability and code readability.

1. Introduction

The adoption of IoT systems is growing rapidly. These systems often operate in critical environments (e.g., healthcare, transport, and infrastructure management), and any bug in the code could lead to failures, posing significant risks. As the use of IoT systems continues to expand, it is crucial to assess the quality of the source code of the software running (on) these systems [1]. The source code quality of IoT systems significantly impacts their functionality, security, and reliability, making code assessment a critical component in their development lifecycle.

* Corresponding author.
E-mail addresses: nour.khezemi.1@etsmtl.ca (N. Khezemi), sikandar.ejaz@encs.concordia.ca (S. Ejaz), naouel.moha@etsmtl.ca (N. Moha), yann-gael.gueheneuc@concordia.ca (Y.-G. Guéhéneuc).

https://doi.org/10.1016/j.iot.2025.101803
Received 29 May 2025; Received in revised form 23 August 2025; Accepted 16 October 2025
Available online 23 October 2025

Before we delve into the details, we would like to define the terms non-IoT and IoT systems and clarify their meanings within the context of this research. IoT systems differ from traditional software systems in that they comprise multiple components (referred to as layers), whereas traditional or non-IoT systems typically emphasise a single component (i.e., the application layer). These components must function cohesively to enable physical devices to interact with the physical world, generating and processing data in real time [2]. Existing research, while examining non-IoT and IoT systems' software quality [3–5], primarily focuses on assessing non-IoT software code quality. Overall, the literature focused less on IoT systems, creating a gap in understanding their software quality and raising a critical idea to compare the software quality of IoT and non-IoT systems. Without this comparison, we cannot confidently apply the best practices from non-IoT system software to the IoT system software because it lacks the necessary foundation and specificity. As a result, the efficacy and reliability of such practices are not guaranteed when applied to IoT system software.

IoT systems are inherently resource-constrained and heterogeneous; they frequently require higher standards of software engineering and code quality than traditional non-IoT systems [6]. To integrate distributed communication protocols, real-time data acquisition, and embedded devices, developers must follow strict guidelines, including fault tolerance, concurrency management, and modular architecture. An IoT middleware, for instance, needs to have clear abstractions between application logic and hardware interfaces to guarantee maintainability across various sensors and communication protocols. Because of this requirement, design patterns, defined APIs, and standardised interfaces are frequently applied more systematically than in many standalone non-IoT systems, which may accept ad hoc coding techniques more.

Additionally, the safety-critical nature of many IoT applications, such as smart grids, industrial automation, and healthcare monitoring, forces a stronger adherence to software engineering concepts like continuous integration, automated testing pipelines, and static code analysis. Non-IoT systems, on the other hand, that function in less restricted or low-risk settings might not be under as much pressure to implement these procedures consistently. IoT development environments are often characterised by a culture of greater code robustness, testability, and long-term maintainability, positioning them as examples of disciplined engineering in modern software ecosystems.

Comparing code quality between non-IoT and IoT software systems is essential for establishing the best system development and maintenance practices. IoT systems' unique constraints, such as limited resources and distributed architectures, require a thorough analysis of code quality [7]. For instance, in smart cities, poor coding techniques in IoT systems might introduce various security flaws leading to data exposition, & security concerns. IoT devices work in a dynamic environment, necessitating specific quality characteristics such as scalability and adaptability [8].

Larrucea et al. [9] emphasised the lack of established software engineering best practices for IoT systems and highlighted the need for effective guidance in developing IoT systems. Many existing studies discuss the software quality of non-IoT systems, but very few discuss the software quality of IoT systems. Particularly, there is a lack of information on whether the software for non-IoT systems is comparable to IoT systems software. Without this knowledge, the results and best practices proven suitable for non-IoT systems can not be applied in IoT systems.

Our research question is therefore: *Do metric-based best practices for traditional software systems apply to IoT software systems?* The result will directly help developers better understand the differences between non-IoT and IoT systems by doing a comparative analysis via metrics and best practices. They can adjust best practices from non-IoT systems, considering the unique requirements of IoT systems. The comparison between non-IoT and IoT system software thus serves as a first step toward understanding the difference between the types of systems, tailoring existing practices according to IoT system software specificities, and highlighting the need to develop new strategies suited for IoT system software.

Recognising the importance of this comparison, we developed a systematic methodology. This method allowed us to collect, analyse, compare, and evaluate 94 comparable IoT and non-IoT systems. In the coming sections, we provide a descriptive and in-depth analysis of the two types of software systems. We also present specific examples of in-depth analysis of IoT systems to illustrate how our code metrics values manifest in IoT system codebases. Leveraging our findings, we propose an updated, systematically selected list of best practices to address the observed difference between the code of IoT and non-IoT systems.

The following are our research questions and the contributions of this work:

1. RQ1: How to select comparable non-IoT and IoT systems using a systematic method? This question arises from the problem of obtaining unbiased and comparable systems of both types. Our first contribution is a method for selecting equivalent 94 non-IoT and IoT systems software from GitHub to ensure the integrity and validity of our comparative analysis, minimising potential biases of our research. The selection process ensures that the chosen IoT and non-IoT systems are comparable, based on the number of stars and forks, forming a solid foundation for our evaluations.
2. RQ2: How significant are the values of non-IoT/general systems metrics computed on comparable non-IoT and IoT systems? We analyse the results of computing various code metrics, such as Lines of Code (LOC), Depth of Inheritance Tree (DIT), and Comment Percentage (CP), on non-IoT and IoT systems, like Apache/Druid and Samsung/TizenRT. These metric values enable detailed evaluations, offering a granular analysis of IoT system codebases, which we illustrate with specific examples to demonstrate how these metrics manifest in practice.
3. RQ3: How do occurrences of non-IoT/general systems best practices applied on comparable non-IoT and IoT systems correlate? We systematically select, analyse, and present an updated list of best practices for IoT systems, drawn from the literature for each category of code metrics studied. By incorporating practices such as code optimisation techniques, modularity, and design patterns, our study offers targeted solutions to tackle challenges like high code complexity, low maintainability, and poor readability, fostering the development of efficient and sustainable IoT codebases.

The results of the study have shown key differences between the two systems, such as complexity, cohesion, code size, and maintainability. We discuss their implications for IoT systems development. We found that developing software for IoT systems presents greater complexity than non-IoT systems, affecting the overall code quality. Considering these differences, we provide a revised list of best practices for developing IoT systems as a target solution. Our work demonstrates that future work is needed to implement the identified best practices list on IoT systems, and evaluating its effect is necessary to address issues such as complexity, size, and coupling.

The rest of this paper is organised as follows: Section 2 provides an overview and discussion of related work. Section 3 presents the research methodology of the study. Section 4 presents a quantitative analysis and discusses our comparison results. Section 5 presents an in-depth analysis of some IoT systems regarding the qualitative comparison results. Section 6 contains specific examples that illustrate the complexity of IoT systems and how they manifest in IoT codebases. Section 7 discusses results on best practices, while Section 8 validates the impact by surveying IoT developers. Section 9 discusses the practical challenges and implications of our comparison results for IoT systems development, and threats to validity. Finally, Section 10 presents conclusions and future work directions.

2. Related work

To the best of our knowledge, the literature on code-quality best practices only pertains to non-IoT systems. We could only find two previous works that studied code-quality best practices on IoT systems in digital libraries, including *IEEE Xplore* and *ACM Digital Library*. We discuss below the few studies, [10,11], that are related to our work. For IoT systems, research on best practices focused on security [12,13], agile software development [14], and data quality [15], which diverges from the focus of this study.

Klima et al. [10] summarised relevant code quality metrics from IoT systems and assessed their impact on general systems quality based on ISO/IEC standards. They categorise those metrics into size (Lines of Code), complexity (Cyclomatic Complexity), coupling (Response For Class), etc. These metrics offer an accurate evaluation of IoT systems' code quality, and we will use and present them in our comparison, enabling us to systematically assess and juxtapose the code quality between these IoT and non-IoT systems' software.

While our study shares a similar approach in utilising these established IoT systems' code quality metrics, our focus extends beyond the evaluation of metrics. We undertake a comprehensive and comparative analysis between non-IoT and IoT systems, leveraging these metrics to explore the nuanced differences and shared traits between these two types of software systems.

Corno et al. [11] investigated open-source software development in IoT and non-IoT systems, analysing 60 projects. They found significant differences in development processes, developer specialisation, and code reusability between these two types of systems. Their study also examined developer contributions, file modifications, specialisation, and project maturity by analysing project dependencies.

Although Corno et al.'s approach differs in research objectives and methods, it complements our work in understanding IoT systems. Our study focuses on and measures quality metrics to compare the code quality of non-IoT and IoT systems. We further build on this by also examining non-IoT systems.

Barrera et al. [12] discuss IoT security best practices and find that there isn't enough agreement or clarity on practical rules. They highlight the need for more precise, unambiguous procedures to improve IoT security at the device development stage and present a novel approach for assessing the actionability of security suggestions. Similarly, Bellman et al. [13] examine "best practices" in IoT security, pointing out that it lacks a precise definition and that behaviours and intended results are not often the same. The authors provide a more explicit vocabulary for the IoT security community by classifying over a thousand security standards as either actionable "practices" or ambiguous "outcomes," and they suggest a mechanism to encourage manufacturers to embrace improved security practices.

Bolhuis et al. [14] investigate how agile software development best practices might enhance the success of IoT initiatives, emphasising time-to-market, productivity, and cost reduction. They highlight the significance of adjusting agile methodologies to organisational contexts and developing technical and soft skills within teams. It is based on surveys and interviews with agile IoT practitioners and identifies essential practices and team skills that improve project success.

The systematic literature review [15], investigates data quality challenges in data-centric CPS/IoT applications within Industry 4.0, identifying common issues, sources, best practices, and engineering solutions. It offers a comprehensive synthesis of current techniques and highlights future research directions to enhance data quality management.

The previous work emphasises the complexity, differences, and lack of evidence regarding the applicability of best practices from non-IoT to IoT systems. To address these gaps, we identify, examine, and propose a set of metrics and best practices from the literature to support IoT system development. To the best of our knowledge, no prior work has compared the code quality of non-IoT and IoT systems. Our study aims to identify similarities or differences between non-IoT and IoT systems by analysing specific software metrics, offering a straightforward and effective approach for selecting comparable systems. We also provide a quantitative comparison to assess this complexity relative to non-IoT systems, as existing works [16,17] noted the complexity of the IoT systems.

3. Method

Finding comparable non-IoT and IoT systems is challenging, given the need for matching criteria like stars, forks, size, programming language, classes, and files. The approach of matching all those criteria did not yield a significant number of systems on which we could base our comparison, so we adapted our selection process to focus on similar numbers of stars and forks. The popularity of a GitHub repository, as indicated by stars and forks, reflects its relevance within a certain domain or for a particular use case

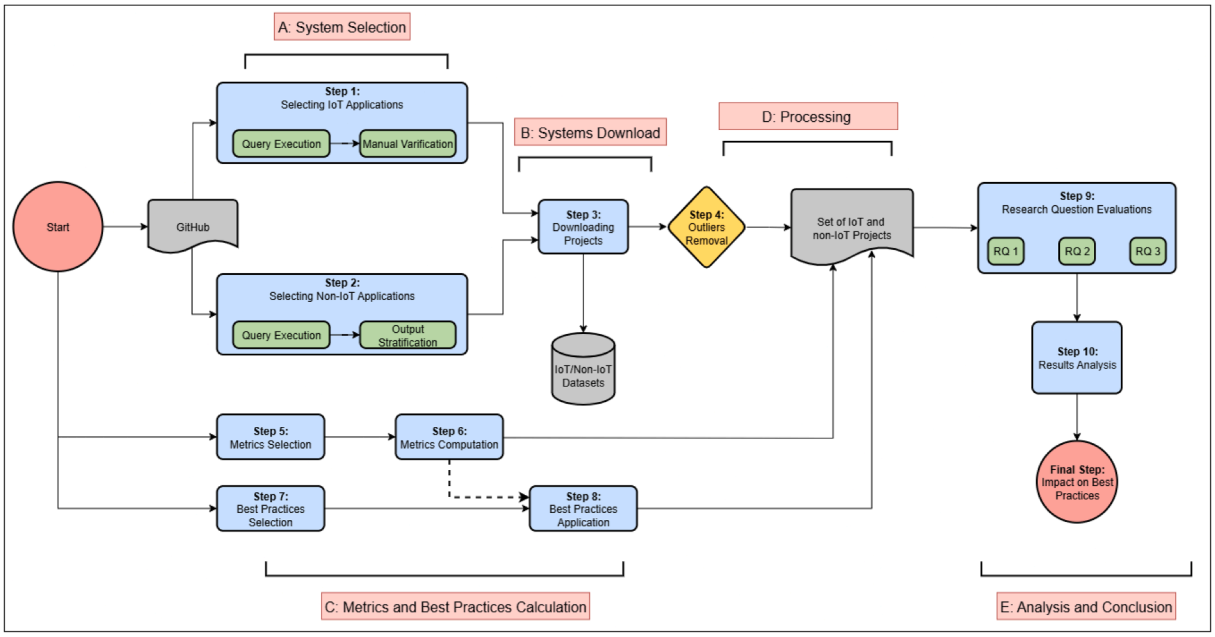
A: System Selection
Step 1: Selecting IoT Applications
Query Execution
Manual Varification
Step 2: Selecting Non-IoT Applications
Query Execution
Output Stratification
Start
GitHub
B: Systems Download
Step 3: Downloading Projects
IoT/Non-IoT Datasets
D: Processing
Step 4: Outliers Removal
Set of IoT and non-IoT Projects
Step 9: Research Question Evaluations
RQ 1
RQ 2
RQ 3
Step 10: Results Analysis
Final Step: Impact on Best Practices
Step 5: Metrics Selection
Step 6: Metrics Computation
Step 7: Best Practices Selection
Step 8: Best Practices Application
C: Metrics and Best Practices Calculation
E: Analysis and Conclusion

Fig. 1. System design for selecting non-IoT and IoT systems.

[18]. GitHub stars offer users a way to convey their appreciation for repositories [11]. When two repositories share similar popularity, it implies they are valued within their respective categories. This approach ensures attention and recognition from the GitHub community.

Our ten-step methodology, based on Politowski et al. [19] and Corno et al. [11], is shown in Fig. 1.

Steps 1–2: Repository Selection and Filtering: We initiated our study by curating a diverse sample of open-source software systems from GitHub to compare IoT and non-IoT codebases. To identify IoT systems, we filtered repositories using GitHub's topic tags–a user-assigned taxonomy used to classify repositories based on their subject area or intended functionality. We focused on repositories labelled with keywords such as IoT, Internet-of-things, EIoT (Enterprise IoT), IIoT (Industrial IoT), Internet of Everything, and Industrial Internet of Things. These topic labels served as a heuristic proxy for domain categorisation, enabling us to automate a large-scale initial filtering of candidate repositories.

Next, we ranked the filtered repositories based on their number of GitHub stars, using this metric to indicate popularity, community engagement, and perceived project quality. We selected the top-ranked, publicly accessible repositories for inclusion. Recognising that not all repositories are actual software implementations–many are instructional materials, API wrappers, or documentation-only projects–we conducted a manual verification process. This involved reviewing README files, project structures, and commit histories to confirm whether the repository represented a runnable and maintainable software system. Our inclusion criteria, which are detailed in a later section, were designed to ensure the relevance, completeness, and suitability of the selected repositories for quantitative software analysis.

Step 3: Data Collection and Archival: Once the relevant repositories were identified, we cloned them into a local database to ensure consistent and versioned access for subsequent analysis. This local archive allowed us to manage dependencies, extract relevant metadata, and apply static analysis tools without risk of upstream changes affecting the replicability of our experiments. For each project, we recorded metadata such as commit history, contributor count, file structure, and dependency configurations.

Step 4: Outlier Detection and Elimination: To maintain internal validity, we examined the dataset for outliers–repositories that either deviated significantly in terms of size, structure, or purpose, or failed to conform to our predefined inclusion rules. This included repositories with unusually high file counts due to generated files or those that were technically categorised as IoT but contained minimal or no actual source code. The identification process involved both automated heuristics (e.g., file extension ratios, LOC thresholds) and manual inspection. Outliers were excluded from further analysis to avoid biasing the metric distributions.

Step 5: Metric Selection: With the refined dataset, we curated a comprehensive set of software engineering metrics to capture structural and quality-related characteristics of each codebase. These metrics span multiple dimensions, including code complexity (e.g., cyclomatic complexity), modularity (e.g., number of classes and packages), maintainability (e.g., code duplication), and repository activity (e.g., number of commits, issue closure rate). We selected metrics based on relevance to prior empirical studies on software quality and their applicability to both non-IoT and IoT systems. We ensured that all metrics were consistently extractable using standardised tooling and scripts.

Step 6: Metric Computation and Comparative Analysis: We computed the selected metrics across all repositories using automated static analysis tools. This quantitative data formed the basis for our comparative study. We conducted statistical analyses (e.g., mean comparison, distribution analysis) to identify significant differences in metric profiles between IoT and non-IoT systems. Particular attention was given to identifying structural or quality-related trends that might distinguish IoT systems, such as elevated coupling or lower comment density. The goal was to establish an empirical baseline for understanding whether IoT systems exhibit unique software characteristics when compared to more traditional software systems.

Step 7: Best Practices Extraction from Non-IoT Systems: Drawing on the metric results and prior literature, we identified a set of best practices that were consistently observed in high-quality non-IoT systems. These practices were not arbitrarily selected; instead, they were empirically grounded in metric patterns associated with better modularity, readability, and maintainability. We prioritised tool-supported practices, which are widely recommended by the software engineering community (e.g., through coding standards) and are practical to implement. These included strategies such as improved modular decomposition, higher test coverage, better documentation, and reduced code complexity.

Step 8: Application of Best Practices to IoT Systems: We then applied a subset of these best practices to the selected IoT systems. The selection was guided by relevance and feasibility–only practices that could be meaningfully implemented in the context of an existing codebase were chosen.

Step 9: Evaluation of Adaptability and Research Question Mapping: Following the intervention, we systematically evaluated how well the selected best practices translated into the IoT context. This involved re-running the metric computations and comparing pre- and post-intervention values. We also qualitatively assessed whether the changes integrated well with the existing codebase or introduced maintainability burdens. These results were mapped to our research questions, which aimed to explore the generalizability of non-IoT practices to IoT systems and assess whether such cross-domain transfer yields tangible improvements.

Step 10: Final Analysis and Conclusion: In the final step, we synthesised our findings across all study phases. We analysed the observed metric changes, assessed whether the best practices led to consistent improvements, and reflected on the limitations of such interventions in IoT systems. Based on this analysis, we concluded the effectiveness and transferability of software engineering best practices across domains and outlined implications for practitioners and future researchers. Our results offer empirical insights into the software engineering characteristics of IoT systems and the potential benefits of importing practices from more mature non-IoT development contexts.

We present our methodology by asking a series of questions that guided our selection process for non-IoT and IoT systems, metrics, and tools. In the following subsections, we present each of them in detail and explain our choices.

3.1. *Which artefacts will we use to base our comparison?*

We could use various software artefacts, including documentation, code, bug reports, chat logs, or execution logs [20], in the comparison process. We choose to focus on source code because it is the common basis for any software system describing its behaviour and functionality.

3.2. *How will we compare the two sets of systems?*

We compare the two sets of systems using metrics because they offer a quantitative and objective way to assess quality. Metrics provide numerical values that allow for direct comparisons, reducing subjectivity and offering a clear basis for evaluating strengths and weaknesses.

We recognise that while metrics are valuable, they may not provide a complete picture of system quality. To conduct a thorough assessment, robust quality models that consider various dimensions and factors are essential. Our work is an initial step in gathering vital insights by measuring and comparing quality metrics. This contributes to the development of more comprehensive quality models.

3.3. *What category of metrics?*

Code metrics are categorised by properties such as size, redundancy, complexity, coupling, unit test coverage, cohesion, code readability, security, code heterogeneity, and maintainability [10]. In our study, when statically comparing code, we selected metrics from different categories: size, complexity, cohesion, coupling, code readability, and maintainability. We did not explore security aspects, which have received extensive attention in both systems [21]. We chose to exclude unit test coverage and effectiveness categories because we focused on static code aspects. The redundancy category was also excluded as we consider it closely related to code readability and maintainability.

3.4. *Which tools we use to compute metrics*

There are various tools available for computing metrics. We opted for two tools due to their popularity [22] and because they can measure the maximum of the list of metrics that we presented in Table 1. They provide comprehensive insights into system complexity, maintainability, and size, aligning perfectly with our research objectives.

Scitools Understand [23] is designed to assist in understanding, evaluating, and verifying source code. It supports a variety of languages and offers the possibility of measuring a variety of code metrics.

Multimetric is a Python library for creating multiple metrics [24]. It is designed to make it easy to build complex and multidimensional metrics that can be used in a variety of applications. The library provides a comprehensive set of APIs and utilities; we are

Table 1
Metrics and best practices categories.

Categories	Metrics	Best Practices
Size	Lines Of Code (LOC), Estimated rebuild value (ERV), Unit Interface Size (UIS), Average Unit Size (US), Number of Non-Architectural Components (NAC), Number of Classes and Files	Code Optimisation Techniques, Identification and Consolidation of Similar Functions, and Use of Run-Time Decomposition.
Complexity	Cyclomatic Complexity (CC), Halstead Volume (HV), WMC-McCabe, Number of Children (NOC), Number of Thing Interconnections (NTI), Depth of Inheritance Tree (DIT)	Refactoring, Modularity, and Packaged software components
Coupling	Response For Class (RFC), Coupling Between Objects (CBO), Number of Incoming calls per module (INC)	Design Principles & Patterns and Refactoring
Cohesion	Lack of Cohesion of Methods (LCOM), Conceptual Cohesion of Classes (C3), Ratio of Cohesive Interactions (NRCI)	Refactoring, and Modularity
Readability	Comment Percentage (CP) and Comment to Code Ratio (CCR)	Textual Features and Code Entropy
Maintainability	Maintainability Index (MI)	Source Code Conventions and Standards, Model-Driven Architecture (MDA), Design Patterns, Refactoring, and Continuous Integration and Continuous Deployment CI/CD)

using one of the APIs to measure our metrics. With Multimetric, we can quickly create, combine, and analyse multiple metrics in a single codebase.

In conclusion, Scitools Understand and Multimetric were selected due to their ability to handle multi-language support and provide a comprehensive analysis of a large list of metrics, ensuring accuracy in our measurements.

3.5. *Which metrics are we using?*

Size, complexity, coupling, cohesion, maintainability, and reliability are significant for software code quality because they collectively influence the system's robustness, ease of evolution, and long-term performance. These metrics provide a comprehensive view of software quality, guiding development and improvement efforts [25]. Table 1 provides an extensive list of code metrics under the categories presented in [10].

For each category, the selection of metrics was strongly guided by their computability within the two systems we analysed, *Understand* and *Multimetric*. Rather than being arbitrary, the choice of metrics was constrained by the capabilities and characteristics of these specific systems. We also focused on choosing metrics that are not exclusively applicable to IoT systems but are more general, allowing us to effectively analyse and compare both types of systems.

Table 2 presents the metrics that we compute using the chosen tools and their formulas. The motivation behind choosing these metrics lies in their collective ability to provide multifaceted insights into various aspects of code quality, ranging from system size and complexity to maintainability and readability. The selection aims to capture diverse dimensions that collectively contribute to software quality assessment.

3.6. *Which code quality best practices are we using?*

To study these best practices, we select some best practices among the many existing ones, such as Code Optimisation Techniques, Refactoring, Design Principles & Patterns, and Code Entropy. Among all these best practices, we select the ones that correspond to the categories of interest, such as Size, Complexity, Coupling, etc. Table 1 summarises the code-quality best practices that we selected for study in non-IoT and IoT systems, categorised by the quality categories chosen above.

3.7. *Which systems are we choosing for the comparison?*

We obtained the sets of systems from GitHub. GitHub provides a wide variety of applications that can be used to gain insights into software development trends, project management, and best practices. Here, we are presenting a selection of IoT and non-IoT systems.

3.7.1. *How can we obtain an IoT dataset?*

We followed the method presented above to select the IoT dataset. In Step 1 in the process presented in Fig. 1, we proceed to a manual selection based on a set of criteria. This allowed us to select systems that are relevant and mature to our research objectives to provide meaningful insights using those selection criteria:

1. The repositories under IoT tags have different variants of syntax (Internet of Things, IoT, EIoT, IIoT, Industrial Internet of Things, Internet of Everything).

Table 2
Metrics used.

Categories	Metrics	Definitions	Formulas	Tools
Size	Line of Code	Counts the number of lines of source code in the system, reflecting its size. In this work, LOC is calculated per file, and we sum all file values to have a value that represents the system.	LOC = Number of non-blank, non-comment lines in the code	Understand
	#Classes	Number of classes of each system	#Classes = Count of class declarations in the source code	Understand
	#Files	Number of files of each system	#Files = Total count of source code files in the project	Understand
Complexity	Cyclomatic Complexity	Calculates the number of linearly independent pathways in system modules [26]. We computed the cumulative CC values by summing up the CC values of all classes within each application.	$CC(m) = E - N + 2P$ Where: $CC(m)$ is the cyclomatic complexity of control flow graph m, E is the number of edges (transfers of control), N is the number of nodes (a sequential group of statements containing only one transfer of control), and P is the number of connected components.	Understand
	Halstead Volume	Measures the software complexity used to assess the program size. The HV is used to measure the amount of code written.	$HV = N.log_2(n)$ Where: Total operators $(N1)$ and total operands $(N2)$, N: Program length calculated as N = N1 + N2, n: The vocabulary of your program is the sum of unique operators and unique operands.	Multimetric
	Weighted Method Count	Measures the sum of the complexity of the methods in a class. This value is calculated per class; in this work, we sum up the WMC values of classes of each system.	$WMC = \sum_{i=1}^{N} CC_i$ Where: CC_i McCabe's Cyclomatic Complexity of local method i, N Total number of local methods in the class	Multimetric
Coupling	Response For Class	Measures the number of different methods and constructors that are called by a specific class. This value is calculated per class; in this work, we sum up the RFC values of classes.	$RFC = Fan - In + Fan - Out$	Understand
	Coupling Between Objects	Assess the coupling between classes based on their usage. CBO metric measures the extent of coupling between two classes by examining the interactions between their methods and instances. The low value of CBO indicates low coupling [27]. This value is calculated per class; in this work, we sum up the CBO values of classes of the system.	$CBO = \|C_{coup}\|$ Where: C_{coup} set of classes	Understand
Cohesion	Lack of Cohesion and Methods	Measures the count of separate sets formed by the local methods of a class, determined by their interaction with class variables [28]. High cohesion indicates good class subdivision [29]. We calculate the sum of LCOM values for each class in the system, and then we divide this sum by the number of systems to obtain the mean value of each.	$LCOM(C) = \frac{1}{a}\frac{\sum_{j-1}^{a} \mu(Aj) - m}{1 - m}$ Where: a stands for the number of variables in a class C. $\mu(Aj)$ is the number of methods of C accessing the variable Aj. m stands for the number of methods in m.	Understand
Readability	Comment Percentage	Quantifies the documentation level by measuring the proportion of code lines dedicated to comments. An appropriate documentation level is considered to be achieved when CP falls within the range of 20 % to 30 % [30]. CP is calculated per file; we sum all file values to have a value that represents the whole system.	$CP = \frac{N_{commet}}{LOC} 100\,\%$ Where: $N_{commet}$is the total number of comments in the source code	Multimetric
Maintainability	Maintainability Index	Measures the ease of maintaining a piece of software. Calculated based on metrics for a software system such as HV, CC, LOC, and the percentage of comment lines per module [31]. The higher the maintainability index, the easier it is to maintain the code.	$MI = 171 - 5.2ln(HV) - 0.23CC$ $-16.2ln(LOC) + 50\sin\sqrt{246.COM}$ Where: COM represents the percentage of comment lines per module.	Multimetric

2. Languages of the repository are supported by both used analysis tools (Java, JavaScript, C, C++, C#, Python).
3. The number of stars is greater than 200 (ensuring that the system is well-rated and of good quality).
4. The number of forks is greater than 20.
5. An active repository with the last push being at least 6 months ago (Date of last push greater than 04–2022).
6. Mature repositories created between 2012 and 2022.

3.7.2. *How can we obtain a non-IoT dataset?*

We used the same set of criteria used to select IoT systems to select the set of non-IoT systems. We built a query to choose a set of non-IoT systems with the same criteria as the selected IoT systems. This query was constructed to identify repositories on GitHub that met certain temporal, popularity, and technological criteria based on our IoT systems selection. The aim was to ensure that the selected repositories were recent, popular, actively maintained, and developed in languages relevant to our study, allowing us to analyse new, popular projects in the non-IoT domain.

For this study, we intentionally selected projects proposed between 2012 and 2022. This approach ensured that each project was mature enough to validate the study, as we focused on projects at least three years old-providing sufficient time for thorough community testing. Our query is:

```
created: > 2012-10-01 created: < 2022-10-01
stars: > 200 stars: < 6500
forks: > 20 forks:< 20216
pushed: > 2022-04-01
language: C, C#, C++, Java, JavaScript, and Python
```

To ensure reproducibility, we have saved the code and the selection process in a replication package on Zenodo[1].

Stratification of the Query Output: The execution of the query returns a large output. From this output, we select a representative dataset regarding the IoT dataset using stratification. Stratification is the process of dividing a dataset into homogeneous subgroups based on certain criteria. This approach allows for a more in-depth analysis within each subgroup and helps ensure that the datasets used for comparison (IoT and non-IoT systems) are as comparable as possible. To stratify the resulting dataset of non-IoT based on the criteria represented by the IoT dataset, we followed these steps.

1. We extracted pertinent details from the IoT repositories to serve as stratification criteria. We chose the composition of the programming language, the number of stars, and of forks. By considering these factors, we aim to create a subset of non-IoT systems that closely resembles the characteristics of the original set.
2. We create a mapping of criteria and repository names with each stratification criterion. For example, a dictionary maps programming languages to lists of repositories that use that language. We do the same thing with the stratification criteria.
3. We divide the non-IoT repositories into strata based on the relevant information.
4. For each subgroup, we select the repositories that most closely match the criteria represented by the IoT GitHub repositories. We use the Pareto principle [32] to select the top repositories in each subgroup.
5. We combine the selected repositories in each subgroup to form a representative subset of the data having the same number and characteristics of IoT systems.

3.7.3. *How to analyse and verify the two sets?*

In this step, we manually analyse the two selected sets to ensure they have an equal number of stars and forks and use the same programming language, thus eliminating external factors that could affect the results. To examine data distribution, we perform statistical tests, including the Shapiro-Wilk test introduced by Hanusz et al. [33].

The Shapiro-Wilk test assesses the normality of data distribution. Checking if the data follows a normal distribution helps ensure the appropriateness of parametric statistical tests.

We compare the number of stars and forks in both datasets and use the non-parametric statistical Mann-Whitney *U* test [34] to determine if they are significantly different. The Mann-Whitney U test is a non-parametric test used to compare two independent groups when assumptions for parametric tests are not met (such as normal distribution). It is employed to determine if there are significant differences between the two datasets in terms of stars and forks. A U-statistic value lower than $\approx$0.05 indicates significant differences, while a higher U-statistic suggests comparability between the datasets.

3.7.4. *How to identify outliers?*

Outliers are data points that are significantly different from the majority of the data [35]. They can impact the results of statistical analyses. We remove outliers from our dataset to improve the accuracy of our results analysis.

We employed a meticulous approach to detect outliers within our dataset. Our methodology prioritised visual inspection, a recognised technique for outlier identification. Through visual representation, specifically by plotting the data, we aimed to pinpoint observations that deviated notably from the expected range. By systematically evaluating outliers and their potential impact on our analysis, we aimed to maintain the integrity and accuracy of our dataset. The removal of these influential outliers allowed for more reliable and precise statistical analyses moving forward.

The identification and removal of outliers are discussed in detail in Section 4.3.

[1] https://zenodo.org/records/10564976

Table 3
Best practices for various code categories.

Category	Best Practices	How they can be Identified?	Application Tools	Available	Appliable	Reason
Size	Code Optimisation Techniques	Cppcheck	Eliminate unused, duplicate code, and replaceable instances and detects opportunities to optimise loops	Yes	Yes	Cppcheck can be used for code optimisation
	Identification and consolidation of similar functions	NA	NA	No	No	Algorithms are available, but not any software
	Use of Run-Time Decompression	IBM CodePack	NA	No	Yes	IBM's CodePack is not open source
Complexity	Applying refactoring	Eclipse, SonarQube, Checkstyle	Pulling up methods, extracting methods, and inlining methods	Yes	Yes	Tools are available and can be used
	Applying modularity	Docker, Virtual Box	Encapsulation and abstraction principles	Yes	Yes	Tools can be used
	Use of packaged software components	ThingSpeak, Microsoft Azure IoT Suite, Google Cloud IoT	NA	No	Yes	Available tools are not open source and will increase the code complexity of IoT systems
Coupling and Cohesion	Application of design principles and patterns	NA	DI pattern for IoC, Patterns like Single Responsibility, and Singleton	Yes	Yes	Multiple studies have shown that design patterns can be applied to IoT systems
	Applying refactoring	Eclipse, SonarQube, Checkstyle	NA	Yes	Yes	Refactoring for IoT is possible by utilising Eclipse
	Applying modularity	Docker, Virtual Box	Cluster Analysis Techniques [36]	Yes	Yes	Containerisation tools are available like Docker
Readability	Use of textual features	ESLint, Pylint, Checkstyle	Use shorter lines of code and consistent indentation, comments, blank lines, meaningful and descriptive variables, etc.	Yes	Yes	Textual features must be integrated while implementing the system
	Improve code entropy	NA	Enhance overall organisation, structure and variability of code	Yes	Yes	Developers must watch the value of entropy while implementing the system
Maintainability	Use of source code conventions	FindBugs1, Checkstyle2, Jtest3	NA	Yes	Yes	Best practice that must be integrated while implementing the system
	Use of model-driven architecture (MDA)	ThingML, Papyrus	NA	Yes	Yes	Applicable when designing the system
	Use design pattern	Eclipse	Factory Method, Singleton, and Decorator	Yes	Yes	Patterns can be used in the design phase
	Applying refactoring	Eclipse, SonarQube, Checkstyle	Encapsulation, limiting the length of code units to 15 lines of code, limiting the number of branch points per unit to 4, etc.	Yes	Yes	Refactoring of IoT is possible
	Continuous Integration and Continuous Deployment (CI/CD)	Azure IoT Edge application, CircleCI, Jenkins as IoT CI/CD Manager	NA	Yes	Yes	To integrate in the development phase

3.8. *How did we obtain non-IoT best practices?*

Best practices refer to a set of recommended guidelines, approaches, methods, tools, or techniques that are considered optimal for reducing issues or enhancing the overall quality of the software. We approached the identification of these best practices systematically, initiating the process by formulating research queries tailored to each metric category and incorporating pertinent keywords. The query structure was designed as follows: *(X or Y) AND (software) AND (best practices)*, where X represents the category name, and Y relates to the specific practice associated with category reduction or improvement. For instance, in the context of the code size category, the query took the form of *((code size) OR (code reduction)) AND (software) AND (best practices)*.

RQ1: How to select comparable IoT and non-IoT systems using a systematic method?
Our Findings: We conclude that no systematic method exists for selecting IoT and non-IoT projects, so we developed our own. This method involves multiple steps, including query execution, visual verification, and result stratification. We have described these steps in detail in Section 3.

We executed these queries on Google Scholar, yielding varying numbers of papers for each category. We followed a systematic selection process involving the filtration of the ten most highly cited articles that provided best practice insights for each category. We studied and extracted pertinent best practices from these articles. We assessed and categorised these extracted practices into three groups: *directly applicable*, *partially applicable with necessary adaptations*, or *not applicable to IoT*. IoT-specific requirements guided this categorisation, prioritising practices based on their relevance and potential impact on the metric categories. Table 3 presents best practices per category for non-IoT systems.

3.9. *How did we ensure the reproducibility of our selection and the generalizability of our results?*

We compared IoT and non-IoT systems, choosing the largest possible subset based on our criteria, believing it represents both types well. Our selection process considered various programming languages and system types to ensure diversity. Our dataset's relevance comes from methodically selecting diverse systems and using stringent criteria to ensure credibility and generalizability. By carefully choosing systems from GitHub and using stratification techniques, we ensured similarity and representativeness between IoT and non-IoT sets. Statistical analyses strengthened the comparability and credibility of our findings. While we could not cover every system, our careful process allows for reasonable generalisations to broader contexts for open-source systems available on GitHub. Also, we extended our systematic process to choose the best practices systematically, guaranteeing reproducibility and validity.

4. Descriptive statistics

4.1. *Output of queries*

We executed our query and applied the process of selection presented in Section 3.7.1. The execution of our query on IoT systems yielded 323 repositories. We removed 10 duplicate repositories. Next, based on manual verification of our selection criteria, we selected 94 repositories. We selected 94 comparable repositories using the stratification technique, presented in Section 3.7.2, of 1972 repositories found when running the non-IoT search query.

4.2. *Statistical analysis of two datasets*

4.2.1. *Nature of distribution*

To examine whether the distributions of stars and forks in our IoT and non-IoT datasets follow a normal distribution, we applied the Shapiro–Wilk test. The test yielded p-values below the significance threshold of 0.05 for both variables in both datasets, leading us to reject the null hypothesis of normality. This result suggests that the distributions of stars and forks are significantly non-normal, motivating the use of non-parametric statistical methods for further comparison.

4.2.2. *Mann–Whitney U test between the two datasets for stars and forks values*

Before doing the comparative analysis, we assessed the normality of the distributions for stars and forks in both IoT (n = 94) and non-IoT (n = 94) groups using the Shapiro-Wilk test. Results indicated strong non-normality for all: IoT stars (W = 0.473, $p < .001$), non-IoT stars (W = 0.273, $p < .001$), IoT forks (W = 0.422, $p < .001$), and non-IoT forks (W = 0.345, $p < .001$).

Given the non-normal distributions, we employed the two-sided Mann-Whitney U test to evaluate differences between IoT and non-IoT repositories. The null hypothesis H_o suggests no difference in distributions, while the alternative H_1 suggests a difference exists.

For stars, the test yielded U = 4409.0, p = 0.969 (> 0.05), with a negligible effect size (Cliff's delta ≈ -0.00045). For forks, U = 4401.0, p = 0.948 (> 0.05), with Cliff's delta ≈ -0.0045. These results fail to reject H_0, indicating no statistically significant differences. Fig. 2 further illustrates the similar medians and spreads between groups. Thus, in terms of stars and forks, proxies for community interest and reuse, the IoT and non-IoT repositories exhibit comparable popularity and engagement.

4.2.3. *Stars and forks distribution in the two datasets*

Fig. 3 presents a scatter plot illustrating the relationship between the number of stars and forks for functions across two datasets. Each point represents a function, with its position determined by its corresponding star and fork counts. The dispersion of points in the plot reflects the variability in popularity and reuse, measured through stars and forks, of these functions. Most data points are densely clustered, indicating that most functions receive a similar number of stars and forks, suggesting consistent patterns of community engagement and reuse across the datasets.

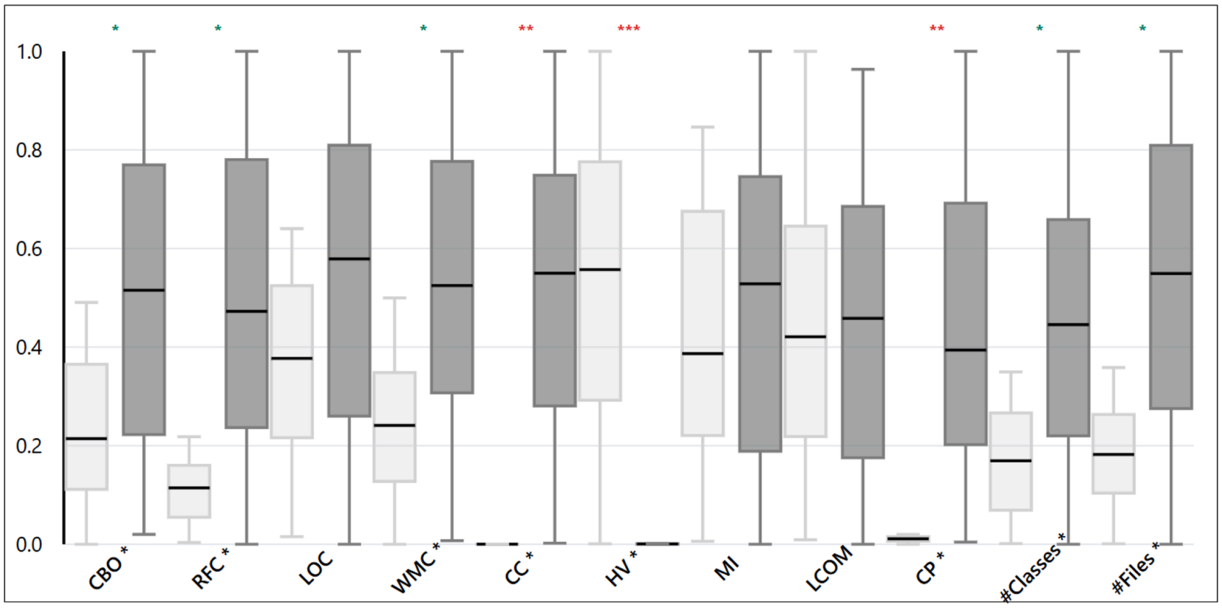

Fig. 2. Software Metrics against Normalised Values (* p < 0.05, ** medium effect, *** large effect Each metric pair shows Non-IoT (left, lighter) vs IoT (right, darker)).

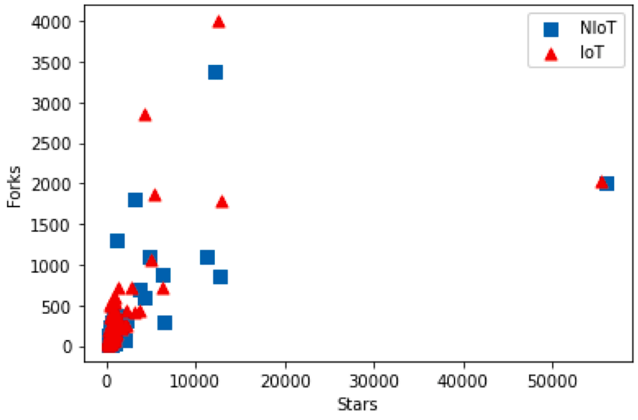

Fig. 3. Stars and forks distribution in the two datasets.

4.2.4. Languages distribution

We selected 14 Java, 11 C+ +, 16 JavaScript, 24 C, 25 Python, and 4 C# systems.

4.3. Outliers

There are several methods for identifying and removing outliers. We use visual inspection to eliminate outliers by plotting the data and then identifying outliers. Any observations that fall outside the expected range are potential outliers. In Fig. 3, visual inspection revealed outliers within our dataset. Notable instances included thingsboard-/thingsboard and home-assistant/core for IoT systems and Apache/Druid and ansible/ansible for non-IoT systems.

To assess these outliers' impact on our analysis, we associated them with top metric values shown in Table 4. 'Apache/Druid' emerged as a system significantly affecting the results, which is why we removed it from further analysis. To ensure consistency, we also excluded its counterpart 'thingsboard/thingsboard'.

4.4. Top values of each metric

We computed our metrics (CBO, RFC, LOC, WMC, CC, HV, MI, CP, LCOM and the number of Classes and Files). Table 5 presents the highest software metric values for IoT and non-IoT projects after removing outliers. IoT projects exhibit higher values in metrics

Table 4
Top before deleting outliers.

	IoT		**non-IoT**	
	Project Name	**Value**	**Project Name**	**Value**
RFC	Samsung/TizenRT [37]	475,185	Apache/Druid [38]	191,718
CBO	eclipse-ditto/ditto [39]	83,212	Apache/Druid	22,423
CC	espressif/esp-mqtt [40]	167	quarnster/SublimeGDB [41]	339
HV	Samsung/TizenRT	7539487.092	quarnster/SublimeGDB	137964.39
MI	eclipse-ditto/ditto	368880.47	Apache/Druid	475261.25
LOC	Samsung/TizenRT	2,009,696	Apache/Druid	1,003,619
WMC	project-chip/connectedhomeip [42]	83,347	Apache/Druid	16,057
LCOM	rwaldron/johnny-five [43]	0.95	rthenica/ffmpeg-kit [44]	0.93
CP	ARMmbed/mbed-os [45]	70251.58	Apache/Druid	4705.6

Table 5
Top after deleting outliers.

	IoT		**non-IoT**	
	Project Name	**Value**	**Project Name**	**Value**
RFC	Samsung/TizenRT	475,185	DarthFubuMVC/fubumvc [46]	64,016
CBO	eclipse-ditto/ditto	83,212	DarthFubuMVC/fubumvc	22,423
CC	flomesh-io/pipy [47]	75256385.40	mgba-emu/mgba [48]	26,505
HV	greghesp/assistant-relay [49]	86,796	dachev/node-cld [50]	240076813.4
MI	eclipse-ditto/ditto	368880.47	wmira/react-icons-kit [51]	811,513
LOC	Samsung/TizenRT	2,009,696	dachev/node-cld	551,449
WMC	project-chip/connectedhomeip	83,347	rthenica/ffmpeg-kit	78,624
LCOM	rwaldron/johnny-five	0.95	rthenica/ffmpeg-kit	0.93
CP	ARMmbed/mbed-os	70251.58	UnknownShadow200/ClassiCube [52]	1413.95

Table 6
Comparison of metrics between the two datasets.

		Median	Mean	Mode	U-statistics	p-value	effect size
CBO	IoT	**188**	4252.51	0	5220.5	0.03062	0.156891
	non-IoT	40	715.72	0			
RFC	IoT	**505**	22334.82	0	5825	0.000149	0.275072
	non-IoT		2956.8	0			
LOC	IoT	**4574**	62960.13	0	4434.5	0.965788	0.003226
	non-IoT		31374.58	0			
WMC	IoT	**217**	4550.46	0	5708.5	0.000848	0.240549
	non-IoT		1667.51	0			
CC	IoT	**462032.02**	26953.31	2	6189	0	0.386057
	non-IoT		2358.08	0			
HV	IoT	2016	7283.76	0	490.5	0	0.765686
	non-IoT		5334.77	441940.37			
MI	IoT	3645.21	24584.31	0	4308.5	0.932151	0.00636
	non-IoT		36403.86	987.39			
LCOM	IoT	**0.72**	0.70	0.83	4040	0.311436	0.0739
	non-IoT		0.74	0.84			
CP	IoT	**44.87**	2200.13	0	5945.5	0.000005	0.33665
	non-IoT		168.49	0			
#Classes	IoT	**116.5**	975.89	0	6688	0.00003	0.295284
	non-IoT		373.76	0			
#Files	IoT	**115**	984	0	5656	0.000909	0.242032
	non-IoT		356.24	2			

like CC, RFC, LOC, WMC, and CBO compared to non-IoT projects. We analysed these projects and found that IoT projects involve more complex hardware and software interactions driven by real-time processing needs. Non-IoT projects generally have higher HV and MI metric values (Table 5). This difference is due to non-IoT projects typically being less complex, influenced by distinct design and coding practices in non-IoT software development.

4.5. Statistical computation on metrics

Metrics reveal that IoT systems feature more extensive and complex code than non-IoT systems due to their hardware constraints, necessitating larger codebases. This highlights IoT's unique characteristics and the necessity to consider them in research and analysis.

Table 6 indicates that IoT systems exhibit greater interdependence than non-IoT systems. IoT's mean CBO is 4252.41, compared to non-IoT's 715.72, illustrating the higher interconnectedness in IoT systems. This interdependence makes IoT systems more challenging to maintain and modify, reflected in the MI values, with a median of 24854.31 for IoT and 36403.86 for non-IoT systems.

Furthermore, IoT systems have more classes and files compared to non-IoT systems. For example, IoT systems have a median of 116.5 classes, while non-IoT systems have 33. This is due to IoT's distributed nature, integrating advanced technologies and adapting to diverse device standards, which require a larger codebase.

5. Quantitative analysis

We conducted an in-depth study of systems in pairs, one for non-IoT and the other for IoT, written in the same language; the comparison is given in Table 6.

The statistical analysis of IoT and non-IoT systems was done using the Mann-Whitney *U* test, a non-parametric alternative to the *t*-test, which is appropriate for comparing two independent groups when the data may not follow a normal distribution. The size was 94 for both IoT systems and non-IoT systems, with a significance level set at $\alpha = 0.05$. Effect sizes were interpreted as small ($|r| = 0.1–0.3$), medium ($|r| = 0.3–0.5$), and large ($|r| > 0.5$).

Among the highly significant differences ($p < 0.001$), Halstead Volume (HV) showed the most unusual disparity, with non-IoT systems exhibiting much higher values than IoT systems ($U = 490.5$, $r = 0.77$). This suggests that non-IoT codebases are substantially larger and more monolithic, whereas IoT code tends to be modular and resource-efficient. Cyclomatic Complexity (CC) was significantly higher in IoT systems ($U = 6{,}189$, $r = 0.39$), reflecting more decision points, loops, and branching logic due to complex sensor data processing, multi-protocol handling, error management, and real-time processing requirements. IoT systems also demonstrated higher use of code patterns ($U = 5{,}945.5$, $r = 0.34$), likely relying on architectural patterns such as Observer, Publisher-Subscriber, and Factory.

Moderately significant differences ($p < 0.01$) included the number of classes, response for class (RFC), weighted methods per class (WMC), and the number of files. IoT systems had more classes ($U = 6{,}688$, $r = 0.30$), more method invocations per class (RFC: $U = 5{,}825$, $r = 0.28$), more complex methods (WMC: $U = 5{,}708.5$, $r = 0.24$), and more files per project ($U = 5{,}656$, $r = 0.24$), indicating greater architectural and functional complexity. Marginally significant differences ($p < 0.05$) were observed in coupling between objects (CBO: $U = 5{,}220.5$, $r = 0.16$), with IoT systems showing higher interdependence among components, suggesting potential integration and maintenance challenges.

Non-significant differences ($p > 0.05$) were found for lines of code (LOC), maintainability index (MI), and lack of cohesion of methods (LCOM), indicating that IoT and non-IoT systems have comparable code volumes, maintainability scores, and method cohesion despite structural and complexity differences. These findings suggest that the observed distinctions between IoT and non-IoT systems are driven more by architectural and functional complexity than by overall size.

The systems we analysed are presented in Table 7. As we obtained similar results for each programming language, we decided to showcase only the two Java-developed systems in the study of metrics.

5.1. *Definition of analysed repositories*

The two Java systems we present are kymjs/CJFrameForAndroid for non-IoT and eclipseditto/ditto for IoT.

Eclipse-ditto/ditto [39] is a framework for managing digital twins. A digital twin is a virtual representation of a physical object or system, and Ditto provides a way to manage the data associated with these virtual representations. The Ditto repository can support device connectivity, data modelling, access control, event processing and analytics.

kymjs/CJFrameForAndroid [53] is an open-source repository for Android developers, providing a framework for building Android apps. The framework is designed to simplify and accelerate Android app development. The framework provides an architecture for building Android apps.

5.2. *Comparison of classes and files*

When comparing the most complex classes of non-IoT and IoT systems, we found that IoT systems have an increased class complexity of 23 compared to 7 in the non-IoT systems.

Furthermore, the largest file in the IoT system exceeds 2500 lines of code, while the largest file in the non-IoT system surpasses 600 lines. Additionally, the largest function in the IoT system contains 290 lines of code, larger than the largest function in the non-IoT system, which has 65 lines. These differences arise from the distinctive nature of IoT systems, characterised by complex hardware compatibility, sensor integration, real-time data processing, diverse communication protocols, extensive data management, and customised business logic.

5.3. *Comparison of other metrics*

By analysing the code, we found that high RFC in the eclipse-ditto/ditto repository is due to many tightly coupled classes. The project handles complex IoT data and protocols, with one class having 55 complex imports for gateway-service connections. The difference between the measured metrics is given in Table 8.

Table 7
Selected systems for each language.

Language	IoT	non-IoT
Java	eclipse-ditto/ditto	kymjs/CJFrameFor-Android
JavaScriprt	fabaff/mqtt-panel	deboyblog/vue-wechat-title
C	timmbogner/Farm-Data-Relay-System	unbit/spockfs
C + +	project-chip/connectedhomeip	zeek/zeek
C#	renode/renode	madskristensen/MiniBlog
Python	DT42/BerryNet	JohnHammond/msdt-follina

Table 8
Comparing measured metrics.

	IoT: eclipse-ditto/ditto	Non-IoT: kymjs/ CJFrameForAndroid
#Stars	414	412
#Forks	147	157
#Classes	7573	32
#Files	4917	25
RFC	130,215	328
CBO	83,212	254
CC	17056558.06	94
HV	13,423	72196.47
MI	368880.47	2043
LOC	363,467	2040
WMC	41,499	238
LCOM	0.62	0.67
CP	5702.18	17.23

RQ2: How are the metrics values of non-IoT/general code computed on comparable non-IoT and IoT systems?
Our Findings: IoT projects generally have larger, more complex codebases than non-IoT systems with higher metrics due to the integration of hardware, software, and data communication components, resulting in more classes, files, and lines of code. In contrast, non-IoT projects exhibit a higher HV, indicating the use of more distinct operators and operands. The metric values are provided in Table 6.
We thus showed that these metrics enable detailed evaluations, providing a granular analysis of IoT system codebases.

The code includes modules that handle the processing of data collected from various IoT sensors. There are functions to parse, filter, aggregate, and transform sensor readings. The repository also provides implementations of communication protocols commonly used in IoT systems.

We examined a system with extensive class inheritance, leading to high coupling and an elevated CBO value. The code also featured intricate logic and business rules, necessitating extensive interaction between objects, further increasing CBO and indicating low cohesion (as evident from the LCOM value).

Eclipse-ditto/ditto exhibited a high CC metric due to its complex algorithms, workflow management, and diverse APIs for device interaction, device protocols, and communication patterns like AMQP, MQTT, and Apache Kafka. It showed a highly modular structure, contributing to a high WMC value.

Contrastingly, Kymjs/CJFrameForAndroid have a low CBO by employing techniques like the Model-View-Presenter architecture for code decoupling. Its focus is to minimise code volume while maintaining functionality.

6. Qualitative analysis

In the previous sections, we argued that IoT projects typically involve more complex interactions between hardware and software components, which can lead to increased code complexity compared to traditional software systems. In this section, we present concrete examples from selected IoT repositories that we analysed in depth. These examples illustrate the multifaceted nature of IoT system architectures, including the integration of sensor data acquisition, communication protocols (e.g., MQTT, CoAP), real-time data processing, and device control logic. By highlighting these elements within actual codebases, we aim to demonstrate how such complexity is reflected in both the structural and behavioural aspects of IoT software.

6.1. Java IoT system

As discussed above, the Java system *Eclipse-ditto/ditto* is highly complex. To further analyse the extent of complexity, we selected one class named **ImplicitThingCreationMessageMapper**, which belongs to the package: org.eclipse.ditto.connectivity.service.mapping.

```
thingTemplate = configuration.findProperty;
(THING_TEMPLATE).orElseThrow(()->
MessageMapperConfigurationInvalidException.
newBuilder(THING_TEMPLATE).build());
commandHeaders=configuration.findProperty(
COMMAND_HEADERS,JsonValue::isObject,JsonValue::
asObject).filter(configuredHeaders>!
configuredHeaders.isEmpty()).map(
configuredHeaders->{ /* ... */}).orElseGet(()->DittoHeaders.newBuilder()
/* ... */);
```

Listing 1. Code for msg transformation in java-based IoT system.

```
privateSignal<CreateThing>getCreateThingSignal(
finalExternalMessage message, final String template){}
The logic for creating a Thing is based on the message and template there are other similar methods of
↪  handling IoT entities
```

Listing 2. Code for IoT Entities Creation and Manipulation in Java-based IoT System.

```
var scanner = new five.Servo({ pin: 12, range: [0, 170]});
var ping = new five.Ping(7);
```

Listing 3. JavaScript code for interactions between hardware components in IoT system.

```
var socket = require("socket.io");
var io = socket.listen(app);
```

Listing 4. JavaScript Code to Integrate Real-time Data Handling with WebSocket Connections in IoT Systems.

This class is responsible for integrating new IoT devices into the Eclipse Ditto IoT platform, handling necessary configurations, policies, and message transformations for seamless device integration and management, making the codebase more complex.

Listing 1 contains configurations and logic for message transformations. The code uses a lambda expression in Java in a nested manner to set configurations for the message mapper. Then, it is used with the *method reference operator* to filter and map Header Configuration, increasing code complexity.

The code also deals with the creation and manipulation of IoT entities such as Thing, Policy, ThingId, etc., as seen in methods like **getCreateThingSignal**, **createInlinePolicyJson**, and **validateThingEntityId**. We present some details of those methods in Listing 2. It contains multiple imports, several interfaces, and methods specific to the *Eclipse Ditto IoT platform*, which results in a complex codebase. As discussed in the previous example, the nested environment incorporates various expressions, increasing overall code complexity.

While Java is commonly used for developing front-end and backend services in IoT systems, JavaScript plays a central role in enabling client-side interactions and lightweight event-driven applications, particularly in web-based IoT dashboards and interfaces. The next section discusses IoT systems developed in JavaScript.

6.2. *JavaScript IoT systems*

rwaldron/johnny-five is a protocol-based IoT and Robotics programming framework. We analysed https://github.com/rwaldron/johnny-five/blob/main/eg/nodeconf-radar.js**eg/nodeconf-radar.js** file. The code is responsible for a radar-like display with simulated scanning motion and distance detection using hardware components and real-time data transmission to a web interface.

The code given in Listing 3 is complex and needs experts to understand it, as it contains interactions and initialisation of hardware components, which requires understanding the pin configuration and range specifications for the servo motor and the Ping sensor from the **'johnnyfive'** library. The code contains some fictitious numbers, for example, "pin", which has a value of 12. Also, the range is defined between 0 to 170, but it is not clear what functions these numbers are performing.

The file also contains real-time data handling with Web-Socket Connections (Socket.io), as shown in Listing 4. This code snippet is used to set up Socket.io for real-time communication. The use of Socket.io indicates the implementation of real-time data transmission, allowing communication between hardware and the web interface via *WebSocket* connections, which makes the codebase more complex compared to non-IoT systems. A complex codebase is less efficient in terms of resource utilisation and is difficult to reproduce.

The inclusion of concurrent operations further contributes to the code's complexity. As demonstrated in Listing 5, which ensures the simultaneous management of servo scanning and Ping sensor data within event-based callbacks, presenting concurrent operations within the 'board.on("ready", function()...)' callback.

```
this.loop(100, function(){}
Logic for scanning servo motion concurrently);
io.sockets.on("connection", function(socket){}
Event-driven handling of Ping sensor data while serving socket connections
ping.on("data",function(){}
Handling Ping sensor data concurrently);});
```

Listing 5. JavaScript code to demonstrate simultaneous operations in IoT systems.

```
#define GLOBAL_LORA_FREQUENCY 915
Setting the LoRa frequency
#define GLOBAL_LORA_SF 12
Configuring the spreading factor
#define GLOBAL_LORA_TXPWR 17
Setting LoRa transmission power
...
...
(other configuration constants)
```

Listing 6. C code for defining configurations for LoRa.

```
crcResult transmitLoRa(uint16_t *destMac,DataReading *packet,uint8_t len){}
Logic for constructing and transmitting LoRa packets includes CRC calculation, packet construction, and
↪  transmission
```

Listing 7. C code to ensure LoRa communication.

```
volatile bool enableInterrupt=true;
Flag to control the interrupt
volatile bool operationDone = false;
Flag indicating packet sent/received
#if defined(ESP8266)||defined(ESP32)
ICACHE_RAM_ATTR
#endif
void setFlag(void)
Handling an interrupt by setting the operationDone flag
Enable/disable based on the enableInterrupt flag
```

Listing 8. C code for handling asynchronous communication in LoRa.

In contrast to JavaScript's high-level abstraction and event-driven model, C is often employed in IoT firmware development where direct hardware control, minimal memory usage, and real-time constraints are critical.

6.3. C IoT systems

The system *timmbogner/Farm-Data-Relay-System* uses ESP-NOW, LoRa, and other protocols to transport sensor data in remote areas without relying on WiFi. It is used for scenarios where there is a need for low power. The code is highly complex due to several factors. The following code examples are extracted from the file https://github.com/timmbogner/Farm-Data-Relay-System/blob/main/src/fdrs_gateway.h**fdrs_gateway_lora.h**. The code in the file handles LoRa communication that involves multiple aspects such as frequency, spreading factor, power levels, ACK timeout, and retries, all of which contribute to configuring the radio for communication.

Listing 6 overviews a code example to define constants for LoRa configuration parameters like frequency, spreading factor, and transmission power. Configuring these parameters is crucial for effective communication, but adds complexity due to their variety and specific values.

Also, functions responsible for LoRa Communication ultimately add to the complexity of the code. In Listing 7, a code snippet is provided where the '**transmitLoRa**' function handles the construction and transmission of LoRa packets. It involves CRC calculation, packet assembly, and finally, transmitting the packet. This increases the complexity due to the detailed packet handling requirements.

In addition, asynchronous handling of LoRa transmission and reception introduces complexity, managing interruptions, flags, and different states for handling data transmission and reception simultaneously. The code snippet of the '**setFlag**' function, in Listing 8, manages interrupts and flags ('enableInterrupt', 'operation-Done') to handle asynchronous communication. Complexity arises from managing interrupts and ensuring correct flag states for proper communication flow.

```
EmberAfStatus emberAfExternalAttributeReadCallback
(EndpointId endpoint,ClusterId clusterId,const EmberAfAttributeMetadata* attributeMetadata,uint8_t*
 ↪  buffer,uint16_t maxReadLength){}
uint16_t endpointIndex= emberAfGetDynamicIndexFromEndpoint(endpoint);
ChipLogDetail(DeviceLayer, "emberAfExternalAttributeReadCallback endpoint%d",endpointIndex);
EmberAfStatus ret=EMBER_ZCL_STATUS_FAILURE;
ContentApp*app=ContentAppPlatform::
GetInstance().GetContentApp(endpoint);
if(app!=nullptr){}
Handle attribute read based on the dynamic endpoint
ret=app->HandleReadAttribute(clusterId, attributeMetadata->attributeId,buffer, maxReadLength);
else
If the app is not found for the dynamic endpoint, use a generic handler
ret=AppPlatformExternalAttributeReadCallback( endpoint,rclusterId,attributeMetadata,
 ↪  buffer,maxReadLength);
return ret;
```

Listing 9. C++ code for determining correspondence between dynamic endpoints.

```
// Example of managing access control with ACLs and bindings
CHIP_ERROR ContentAppPlatform::ManageClientAccess( Messaging::ExchangeManager&exchangeMgr,
 ↪  SessionHandle&sessionHandle,uint16_t, targetVendorId,uint16_t targetProductId, NodeId
 ↪  localNodeId,std::vector<Binding:: Structs::TargetStruct::Type>bindings,
 ↪  Controller::WriteResponseSuccessCallback
successCb,Controller::WriteResponseFailure CallbackfailureCb){}
// Logic for managing ACLs and bindings
// Creation and handling of ACL entries and bindings
// based on vendor and product IDs
...
...
return CHIP_NO_ERROR;
```

Listing 10. Access Control with ACLs and Bindings using C++.

Building on C's low-level capabilities, C++ introduces object-oriented abstractions and is frequently used in IoT projects requiring more structured software design without compromising performance or hardware proximity.

6.4. C++ IoT systems

Project-chip/connectedhomeip is a repository for a unified, open-source application-layer connectivity standard built to enable developers and device manufacturers to connect and build reliable and secure ecosystems and increase compatibility among connected home devices. The code examined in the file https://github.com/project-chip/connectedhomeip/blob/c58f0624887746e6dfa67fb1846a6c04420e6867/src/app/app-platform/ContentAppPlatform.cpp#L4**ContentAppPlatform.cpp.** deals with dynamic endpoints and their associated attributes.

External callbacks for attributes read and write, as shown in Listing 9, through the method **emberAfExternalAttributeReadCallback** that handles attribute read operations, respectively, for dynamic endpoints. In the same file, there was a similar function **emberAfExternalAttributeWriteCallback**, which writes operations. The code checks whether the dynamic endpoint corresponds to a known content app. If found, it calls the app-specific handler; otherwise, it falls back to a generic handler. This demonstrates the complexity of managing different attribute operations based on dynamic endpoints and handling scenarios where the app is not available for a given endpoint, which results in a complex codebase.

Managing access control for endpoints of IoT systems introduces complexity. Listing 10 overviews code, which deals with setting and revoking permissions for various devices. It presents a function that manages access control by creating ACL entries and bindings for specific vendor and product IDs.

Unlike the native execution model of C and C++, C# is typically found in IoT systems built on the .NET ecosystem, where managed code, cross-platform development (via .NET Core), and integration with cloud services are prominent.

6.5. C# IoT systems

We studied *renode/renode*, which is an open-source simulation and virtual development framework for complex IoT embedded systems. We present a class named **ArduinoLoader** within the **Antmicro.Renode.Integrations** namespace from the file https://github.com/renode/renode/blob/b254f5d2f593e612da80dbb2337fb6394028eca8/src/Renode/Integrations/ArduinoLoader.cs#L27**ArduinoLoader.cs**.

The class sets up USB devices, configurations, and functional descriptors. It involves configuring multiple USB interfaces, endpoints, and descriptors, which make the codebase complex, as presented in Listing 11.

```
USBEndpoint interruptEndpoint = null;
// ... (USB configuration)
USBCore = new USBDeviceCore(this,classCode: USBClassCode.CommunicationsCDCControl,
↪  maximalPacketSize:PacketSize.Size16,vendorId: 0x2341,productId:0x805a,deviceReleaseNumber
↪  :0x0100).WithConfiguration(configure: c =>c.WithEndpoint(Direction.DeviceToHost,
↪  EndpointTransferType.Interrupt, maximumPacketSize:0x08, interval: 0x0a, createdEndpoint: out
↪  interruptEndpoint))
// ... (configuring USB interfaces, endpoints, and descriptors)
// ...
```

Listing 11. C# Code for Setting Up USB Endpoints, Configurations and Functional Descriptions.

```
private void Decode(byte[]d){}
this.Log(LogLevel.Noisy, "Decoding input:{0}", System.Text.ASCIIEncoding.ASCII.GetString(d));
uint value = 0;
uint savedValue = 0;
var command = Command.None;
for(var i=0;i<d.Length;i++){}
if(d[i]>='0'&&d[i]<='9'){}
AppendNibble(ref value,(byte)(d[i]-'0'));
else if(d[i] >= 'a' && d[i] <= 'f'){}
AppendNibble(ref value,(byte)(d[i]-'a'));
else if(d[i]>='A'&&d[i]<='F'){}
AppendNibble(ref value,(byte)(d[i]-'A'));
else{}
switch((char)d[i]){}
// ... (handling various cases)
}}}}
```

Listing 12. C# code for encoding and decoding incoming commands.

```
if mode=='inference':self.disable_engine=False
self.engine=PipelineEngine(...)
else:
self.disable_engine=True
self.engine=PipelineDummyEngine()
```

Listing 13. Python code to ensure dynamic engine switching.

Also, as we are dealing with an IoT system, there is data transfer. The Decode method processes incoming data as shown in Listing 12. It iterates over the input data, interpreting ASCII characters. Depending on the character type, it appends nibbles to form numerical values. Switch statements handle special characters, indicating different types of commands. The code handles various cases, increasing the complexity.

Python, with its simplicity and extensive ecosystem, is increasingly used in IoT for rapid prototyping, data processing, and machine learning integration, especially when high performance is not the primary constraint. We have discussed Python-based IoT systems in the next section.

6.6. *Python IoT systems*

DT42/BerryNet is an AI/IoT system that connects independent components. Component types include but are not limited to AI engines, I/O processors, data processors (algorithms), or data collectors. We studied the code in file https://github.com/DT42/BerryNet/blob/2f13f5b559ee22d1c0e325834677b10a504fd117/berrynet/bndyda/bnpipeline.py#L4**bnpipeline.py**, which defines classes related to a pipeline engine for processing data in an AI/IoT context.

The complexity of the studied codebase arises from its dynamic behaviour, extensive configuration options, communication handling, and the need to manage different modes and engines based on external messages. While these features provide flexibility to IoT, they also increase the overall complexity of the codebase.

In Listing 13, we present code ensuring dynamic engine switching between a real pipeline engine *PipelineEngine* and a dummy engine *PipelineDummyEngine* based on MQTT messages indicating the service mode (inference, idle, or learning). This dynamic switching adds complexity to the code.

The code ensures communication with an MQTT broker, handling various topics and messages. This includes sending results, deploying newly retrained models, and switching between inference and non-inference modes. Using MQTT for communication increases code complexity. Listing 14 is an example of communication handling.

```python
self.comm.send('berrynet/engine/pipeline/result' , tools.dump_json(generalized_result))
```

Listing 14. Python code to ensure communication with an MQTT broker.

7. Impact on best practices

Our study reveals differences between non-IoT and IoT systems. Therefore, we use these insights to enhance best practices from non-IoT systems to provide specific guidelines for addressing IoT-specific challenges, which we have discovered through our comparison. These challenges include high coupling, low cohesion, high complexity, low maintainability, code size reduction, and readability. In Section 3, we have presented best practices per category, shown in Table 3. Some of the best practices found, such as modularity and refactoring, can solve multiple problems, which is why they are repeated under different categories. A replication package with a detailed version of Table 3, including additional information, is accessible on Zenodo[2].

7.1. Size

We observed that code size is bigger in IoT systems based on high values of the measured metric LOC and the increased number of classes and files in IoT systems. The execution of the query yielded 530,000 papers from which we selected the ten most highly cited papers. To solve the previously demonstrated size issues in Sections 4, 5, and 6, we found these best practices.

7.1.1. Code optimisation techniques

Multiple studies introduced techniques to reduce the size of code [54,55]. Most of these techniques could be used in IoT systems, such as loop unrolling, strength reduction by replacing costly operations with less resource-intensive alternatives, function inlining to minimise function call overhead, strength reduction of arrays, eliminating redundant computations, removing duplicated code, and optimising function libraries by selecting lightweight dependencies and unused code removal.

Static analysis tools can help to implement these techniques, such as Cppcheck, which is used in embedded systems and IoT development for C/C++ code [56]. It can also detect opportunities to optimise loops or suggest better ways to handle iterations, indirectly impacting code size by reducing the number of instructions executed.

7.1.2. Identification and consolidation of similar functions

Reducing code size is possible through the identification and consolidation of similar functions.

One of the selected papers is the work of Edler et al. [55], which proposes a platform-independent code optimisation technique to reduce code size by merging structurally similar functions. The Function Merging algorithm compares function signatures and control flow graphs to detect equivalence.

The platform-independent nature of the algorithm, operating at the intermediate representation level within a Low-Level Virtual Machine (LLVM), makes it adaptable to the diverse range of IoT devices with varying architectures by abstracting away hardware-specific details and allowing for the generation of code suitable for different target environments. The algorithm parameters, including minimum instruction count and similarity thresholds, contribute to its adaptability, ensuring that the merging process caters to the specific constraints of IoT environments.

7.1.3. Use of run-time decompression

Run-time decompression techniques provide code size reduction. This run-time decompression involves employing techniques such as dictionary-based software decompression and selective compression. Lefurgy et al. [57] proposed a dictionary-based software decompression, a software decompressor based on IBM CodePack, and a technique of selective compression for controlling performance degradation due to decompression, using software-managed caches to support code decompression at the granularity of a cache line.

Techniques that we can adapt for IoT development from run-time decompression include selective decompression, dynamic decompression thresholds tailored to resource availability, and conditional decompression. However, whole-program decompression, real-time decompression of large codebases, and data compression techniques may be less practical for many IoT devices and components with limited resources.

7.2. Complexity

We observed that complexity is high in IoT systems based on high values of measured metrics such as CC and WMC. The query resulted in 366,000 papers, and we examined the ten most highly cited ones. To solve the previously demonstrated complexity issues in Sections 4–6, we found these best practices.

[2] https://zenodo.org/records/10564976

7.2.1. Applying refactoring

Refactoring methods [58] provide an array of strategies for reducing complexity [59]. Refactoring involves redistributing variables and methods across the class hierarchy to simplify the software system structure, with highlighted techniques such as pulling up, extracting, and inlining methods. Mayer Christian [60] underscores the importance of regular code refactoring in development for breaking down complex functions. While refactoring applies to IoT systems, it may introduce concurrency bugs and behaviour changes [61], requiring post-refactoring detection and evaluation for corrections.

For Java-based IoT systems, the refactoring process can be seamlessly executed in the Eclipse IDE, utilising its integrated refactoring tools. Static code analysis tools like SonarQube or Checkstyle can identify potential areas for refactoring. Integrating them into the development pipeline for continuous static code analysis and improvement suggestions aids in reducing code complexity.

7.2.2. Applying modularity

Modularity of the code is a prominent technique for complexity reduction, highlighted in Baldwin and Clark’s theory of modularity [62] and in the work of Kearney et al. [63]. This technique emphasises the advantages of decomposing complex systems into smaller, manageable modules, a concept that finds resonance in IoT development. We believe that this principle can be applied to IoT systems by breaking down an IoT system into modular components. Containerisation tools like Docker are available to facilitate encapsulation and abstraction principles, which are techniques that contribute to the better modularity of the code. Docker uses containerisation to encapsulate applications and their dependencies, creating isolated environments. In IoT systems, we can create Docker containers for different components or services and package each component with its dependencies into a separate Docker image.

7.2.3. Use of packaged software components

Packaged software components are pre-built, ready-to-use software modules or frameworks that can be integrated into a larger software system and are known for decreasing software complexity [64]. In IoT development, where this concept, like IoT platforms, is common, these findings bear significance. Examples of packaged software components include IoT platforms, which offer tools and services for building and managing IoT applications using tools such as *ThingSpeak*, *Microsoft Azure IoT Suite*, *Google Cloud IoT*, and *IBM Watson IoT Platform*.

7.3. Coupling and cohesion

IoT systems have high coupling (high RFC and CBO) and low cohesion (low LCOM) compared to non-IoT systems. We found 26,400 papers and selected the top ten based on citations. From these ten papers, we extract best practices to solve the previously demonstrated coupling and cohesion issues in Sections 4, 5, and 6.

7.3.1. Application of design principles and patterns

Walls and Breidenbach [65] showed that Dependency Injection (DI) achieves Inversion of Control (IoC), leading to reduced coupling and enhanced code cohesion.

In resource-constrained IoT environments, we do not have the luxury of using full-fledged DI frameworks. However, we believe that using lightweight DI Frameworks through a lightweight DI library such as TinyIoC or MicroDI, which are designed for embedded and IoT systems, is useful. These frameworks provide basic DI functionality without the overhead of larger frameworks.

Singleton and Factory [66,67] ensure individual class responsibilities, enhancing cohesion and reducing coupling. For IoT, the application of these patterns is straightforward and facilitates the decoupling of modules, simplifying role separation and mitigating device heterogeneity [68]. When applying these patterns, there are minimal differences compared to non-IoT contexts that must be considered, such as optimising custom design pattern implementations for IoT systems operating in resource-constrained environments.

7.3.2. Applying refactoring

Same as for complexity, Du Bois et al. [69] offered a guideline for refactoring to improve code coupling and cohesion. It is crucial to organise code with related functionality grouped and separate different concerns into distinct modules or classes [66]. Refactoring enhances coupling by reducing interconnections between modules through minimising method calls and shared variables [66].

We find that these refactoring principles apply to IoT systems based on our study of refactoring steps. There are tools and IDE features available for developers to automatically identify and suggest refactoring for IoT systems [67], such as Eclipse, SonarQube, and Checkstyle.

7.3.3. Applying modularity

Modularity is a well-known best practice to enhance coupling and cohesion. Utilising cluster analysis techniques can evaluate and improve modularisation [62]. In the IoT context, semantic categorisation can be employed to group IoT components based on their roles (e.g., sensors, actuators, controllers), and combining structural and semantic criteria enhances modularisation comprehensively.

To enhance modularity, cluster analysis techniques can be applied [36]. In the IoT system, this involves analysing relationships and dependencies between different components or modules. By identifying interrelationships among various IoT devices or components, we can create more cohesive and loosely coupled modules.

7.4. Readability

IoT systems exhibit higher code readability, as indicated by their higher CP values compared to non-IoT systems. The research query produced 66,700 papers and, from them, selected the ten most highly cited ones.

One significant challenge in readability studies is the complexity of experimentally substantiating what essentially constitutes a subjective perception. Obtaining measures of subjective perception is challenging, necessitating human studies and inherently involving variability. To derive useful measures, large-scale surveys that include multiple human raters and careful statistical analysis of inter-rater agreement are essential [70]. We report the best practices that proved useful for improving code readability.

7.4.1. Use of textual features

Using simple textual features enhances code readability, emphasising the importance of shorter lines, consistent indentation, and judicious use of comments [71,72]. While comments may not uniformly indicate high readability, they directly communicate intent, making their use preferable. Blank lines are positively correlated with readability [71,73]. Xiaoran et al. propose SEGMENT [74], a heuristic solution for automatic blank line insertion based on program structure and naming information.

Adapting *SEGMENT's* heuristics to IoT code by considering structural elements like event handlers, data processing, and communication tasks allows for inserting blank lines between logically related code segments, enhancing readability. Meaningful variable names and descriptive method names are important for clarity [73,75]. In IoT development, employing clear and descriptive names for variables representing sensors and actuators improves code readability, especially when methods interact with sensors or perform specific tasks.

To implement these techniques, manual code reviews focusing on the mentioned textual features or developing custom scripts tailored to IoT programming languages are viable options. Similarly, existing static code analysis tools supporting readability metrics, such as *ESLint*, *Pylint*, or *Checkstyle*, can be adapted or extended to address the outlined requirements.

7.4.2. Improve code entropy

The concept of entropy measures the amount of information content in the source code. It is often viewed as the complexity, the degree of disorder, or the amount of information in a signal or data set. Entropy is calculated from the counts of terms (tokens or bytes) as well as the number of unique terms and bytes. Posnett et al. [70] suggest that snippets with higher entropy are more readable. This implies that code with more varied elements (operators and operands) is easier to understand.

In IoT systems, it is very common to deal with a variety of sensors, actuators, and communication protocols. While enhancing the overall variability of code, developers must consistently monitor the entropy value across diverse elements (operators and operands) in the code to enhance its overall entropy using static code analysis tools while creating IoT systems.

7.5. Maintainability

The maintainability of IoT systems is low compared to non-IoT systems; this is proved by the low value of MI, high code complexity, and high interdependence between different modules within a system. The research resulted in 55,300 papers, and we examined the ten most highly cited ones. To solve the previously found maintainability issues in Sections 4, 5, and 6, we found the following best practices.

7.5.1. Use of source code conventions and standards

Source code conventions and programming languages have evolved together, and adhering to uniform conventions, such as naming conventions, inlined documentation, and syntactic structure, enhances code readability. Barry et al. outlined crucial code conventions for maintainability, particularly relevant to Java [76]. These conventions include recommendations for If, For, and Try statements, suggesting at most one additional nested statement, advocating the design of extensible classes without code in public methods, and more.

This convention list is widely applicable to IoT system codes. Implementing these conventions can be facilitated by employing tools like FindBugs, Checkstyle, and Jtest.

7.5.2. Use of model-Driven architecture (MDA)

MDA involves expressing system requirements in a modelling language (e.g., UML) to generate a Platform Independent Model (PIM). This PIM is then transformed into a Platform Specific Model (PSM) for a particular technology and then into the actual code. MDA improves system maintenance by facilitating changes at the requirements level, automatically propagating them to affected modules [77].

In IoT systems, MDA can be leveraged to create high-level models to capture system requirements like sensor integration, data processing, and communication protocols. MDA in IoT ensures code generation based on these models, enhancing code maintainability and reducing errors [78].

7.5.3. Use of design patterns

In addition to coupling and cohesion, design patterns positively impact code maintainability [79]. Jun et al. [80] empirically demonstrated that effective use of design patterns enhances software maintainability through an evaluation of a system without design patterns against its refined version after applying appropriate design patterns.

The use of design patterns in IoT systems is straightforward. For instance, the Factory Method pattern eases object creation without specifying concrete classes, facilitating the integration of new device types or functionalities in an IoT context. The Decorator pattern allows dynamic addition of responsibilities to objects, enabling flexible enhancement of IoT device capabilities without altering their core structure. Tools like Eclipse for Java systems can assist in implementing these patterns.

7.5.4. Applying refactoring

Like complexity, coupling, and cohesion, refactoring techniques positively impact software maintainability [79] and reduce technical debt [81].

For C# code, Visser et al. provided guidelines for maintainability improvement through refactoring [82]. This includes limiting the length of code units (methods or constructors) to 15 lines, restricting the number of branch points per unit to 4 (splitting complex units into simpler ones), and balancing the relative size of top-level components.

Refactoring code in IoT solutions requires an understanding of the system architecture and its implications on data flow and communication protocols, facilitating code restructuring for improved maintainability without altering the external behaviour of IoT systems. While refactoring, we can implement encapsulation, which, as advocated by Anda and Bente [83], improves maintainability by hiding system details. In IoT, encapsulation involves concealing the internal details of IoT devices and their communication protocols.

7.5.5. Continuous integration and continuous deployment (CI/CD)

Implementing CI/CD pipelines to automate testing and deployment processes improves maintainability [81,82].

For IoT systems, CI/CD facilitates rapid and reliable updates to IoT devices. However, there are specific considerations to consider before applying it to IoT, such as creating realistic IoT device simulations for testing. Tools and frameworks like Eclipse Kapua, IoTivity, and IoT-LAB can simulate IoT device behaviour and interactions. Update mechanisms are essential for remotely deploying firmware updates to IoT devices. The whole CI/CD pipeline can be done through Azure IoT Edge application, CircleCI and Jenkins as IoT CI/CD Manager.

8. Validation of the impact

To validate the study, a survey was conducted to capture the demographic profile, educational background, regional distribution, and levels of expertise and experience of participants engaged in IoT systems development. The survey collected 13 responses. Most participants were male (77 %), with females comprising 15 % and one respondent selecting “Other” (8 %). In terms of education, the majority held a master’s degree (46 %), followed by PhDs (38 %), while a smaller proportion reported bachelor’s or other qualifications (8 % each). Geographically, respondents were concentrated in the Americas (69 %), with the remainder from Asia (8 %) and other regions (23 %). Regarding IoT systems development expertise, most self-identified as beginners (38 %) or intermediates (46 %), with only one expert (8 %). Experience levels showed a bimodal distribution: a large share reported 0–2 years (38 %) or 3–5 years (31 %), while fewer reported 5–10 years (15 %) or more than 10 years (15 %).

We asked the developers two questions regarding best practices related to Code Size, Complexity, Coupling and Cohesion, Readability, and Maintainability: Are they currently following these best practices? And if not, would they consider adopting them?

The responses are summarised and discussed in this section as well in the Table 9 below:

8.1. Size

Code optimisation techniques: Based on the response, we conclude that the adoption is low (23.1 %). The overwhelming majority (76.9 %) do not currently apply these techniques, yet nearly all of them (92.3 %) would consider using them. This suggests that developers recognise the importance of optimisation but may be discouraged by the additional effort, tooling complexity, or the perception that optimisation is not as important in early development stages.

Identification and consolidation of similar functions: The survey responses show moderate adoption (53.8 %), which indicates a fair level of recognition of its value in reducing redundancy. Among non-adopters, 76.9 % expressed willingness to adopt, though 15.4 % preferred not to answer, which may signal ambiguity about the practice’s applicability in IoT contexts where function consolidation could conflict with device-specific requirements.

Use of run-time decompression: The responses show very low adoption (23.1 %) and very high non-use (76.9 %). Encouragingly, 84.6 % of those not practising it would consider doing so, which highlights awareness of memory and storage constraints in IoT devices. However, the presence of 13.1 % preferring not to answer indicates uncertainty about the performance trade-offs inherent to run-time decompression.

Size-related best practices are weakly adopted overall, with significant room for improvement. The high willingness to adopt suggests that barriers are likely technical or contextual rather than conceptual.

8.2. Complexity

Applying refactoring: The survey shows that the adoption is strong (76.9 %), showing widespread recognition of its role in managing complexity. Among non-adopters, 69.2 % are open to adopting it, reinforcing its general acceptance as a fundamental software engineering practice in IoT.

Table 9
Developers responses.

		Do you follow the best practice?			If not, would you consider it?		
Category	Best Practice	Yes	No	Prefer not to answer	Yes	No	Prefer not to answer
Size	Code optimization techniques	23.1 % (3)	76.9 % (10)	0	92.3 % (12)	7.7 % (1)	0
	Identification and consolidation of similar functions	53.8 % (7)	38.5 % (5)	7.7 % (1)	76.9 % (10)	7.7 % (1)	15.4 % (2)
	Use of run-time decompression	23.1 % (3)	76.9 % (10)	0	84.6 % (11)	15.4 % (2)	0
Complexity	Applying refactoring	76.9 % (10)	23.1 % (3)	0	69.2 % (9)	7.7 % (1)	23.1 % (3)
	Applying modularity	69.2 % (9)	30.8 % (4)	0	53.8 % (7)	15.4 % (2)	30.8 % (4)
	Use of packaged software components	30.8 % (4)	69.2 % (9)	0	69.2 % (9)	15.4 % (2)	15.4 % (2)
Coupling and Cohesion	Application of design principles and patterns	69.2 % (9)	30.8 % (4)	0	84.6 % (11)	0	15.4 % (2)
	Applying refactoring	69.2 % (9)	30.8 % (4)	0	76.9 % (10)	23.1 % (3)	0
	Applying modularity	53.8 % (7)	38.5 % (5)	7.7 % (1)	46.2 % (6)	30.8 % (4)	23.1 % (3)
Readability	Use of textual features	53.8 % (7)	46.2 % (6)	0	92.3 % (12)	7.7 % (1)	0
	Improve code entropy	46.2 % (6)	46.2 % (6)	4.4 % (1)	84.6 % (11)	7.7 % (1)	7.7 % (1)
Maintainability	Use of source code conventions	53.8 % (7)	46.2 % (6)	0	84.6 % (11)	7.7 % (1)	7.7 % (1)
	Use of model-driven architecture (MDA)	46.2 % (6)	53.8 % (7)	0	76.9 % (10)	23.1 % (3)	0
	Using design patterns	53.8 % (7)	46.2 % (6)	0	84.6 % (11)	7.7 % (1)	7.7 % (1)
	Applying refactoring	53.8 % (7)	46.2 % (6)	0	76.9 % (10)	15.4 % (2)	7.7 % (1)
	Continuous Integration and Continuous Deployment (CI/CD)	46.2 % (6)	53.8 % (7)	0	92.3 % (12)	7.7 % (1)	0

Applying modularity: The adoption is also relatively high (69.2 %). However, a substantial 30.8 % of non-adopters are resistant, possibly reflecting challenges in applying modularity within tightly integrated IoT systems where hardware-software coupling limits design flexibility.

Use of packaged software components: The survey shows that the adoption is low (30.8 %), suggesting hesitation in reusing pre-built components. Despite this, 69.2 % of non-users would consider using it, although 15.4 % preferred not to answer. We believe these responses reflect concerns about security, licensing, and compatibility of third-party components in IoT ecosystems.

Complexity-management best practices are fairly well integrated, with the exception of packaged software components, which developers are still hesitant to adopt.

8.3. Coupling and cohesion

Application of design principles and patterns: The survey concludes that the adoption is relatively high (69.2 %). Notably, 76.9 % of non-adopters expressed willingness to adopt, suggesting a clear recognition of their utility in improving maintainability and scalability, even if actual usage lags.

Applying refactoring: The survey shows moderate-to-high adoption (69.2 %), which indicates recognition of its benefits for reducing unnecessary coupling. Most non-adopters (76.9 %) are open to adoption, demonstrating broad alignment with its value.

Applying modularity: The adoption is lower (53.8 %), with a notable 23.1 % of non-adopters outright rejecting it. This resistance likely arises from practical limitations of modular design in resource-constrained or highly specialised IoT systems.

RQ3: How do occurrences of non-IoT/general best practices applied on comparable non-IoT and IoT systems correlate?
Our Findings: In this study, we have adapted best practices from non-IoT/general systems and applied them to IoT systems, targeting challenges like code complexity, low maintainability, and poor readability. We conclude that best coding practices, such as modularity, code optimisation, and design patterns from non-IoT systems, are effective in addressing the above-mentioned issues in IoT systems.

Practices aimed at improving coupling and cohesion show mixed levels of adoption. While refactoring is relatively well-accepted, modularity faces stronger resistance in IoT, reflecting a gap between theoretical best practices and practical feasibility.

8.4. Readability

Use of textual features: The survey shows that the adoption is modest (53.8 %), but nearly all non-users (92.3 %) would consider it. This suggests that textual features are seen as a lightweight and practical means to improve clarity, though not yet consistently prioritised.

Improve code entropy: The survey concludes that the adoption is divided (46.2 % yes, 46.2 % no). However, 84.6 % of non-users would consider using it, indicating recognition of the importance of code readability, though possibly not prioritised against functional concerns in IoT development.

Readability-related practices enjoy strong conceptual support, but adoption by developers lags. The gap between willingness and practice suggests institutional or technical barriers rather than scepticism about their value.

8.5. Maintainability

Use of source code conventions: The adoption is moderate (53.8 %), but significantly, 100 % of non-users expressed willingness to adopt. This reveals strong inherent acceptance and suggests that standardising conventions could be a relatively easy win for improving readability.

Use of model-driven architecture (MDA): The survey shows that the adoption is modest (46.2 %), while the majority of non-users (84.6 %) are willing to adopt. This indicates conceptual appreciation of MDA's value, but possible practical constraints such as tool maturity, steep learning curves, or integration challenges.

Using design patterns: The survey concludes that the adoption is also modest (53.8 %). Similar to MDA, a strong majority of non-users (84.6 %) would consider adoption, suggesting underutilization of proven practices rather than rejection of their usefulness.

Applying refactoring: The adoption is moderate (53.8 %), with 76.9 % of non-users willing to adopt. This indicates strong recognition of its role in enhancing maintainability, though consistency of application remains a challenge.

Continuous Integration and Continuous Deployment (CI/CD): The adoption is low (46.2 %), but willingness is exceptionally high (92.3 %). This disparity suggests that while IoT developers recognise the value of CI/CD, practical adoption is hampered by integration complexity, particularly in environments where embedded devices are difficult to test and deploy continuously.

Maintainability practices are seen as crucial, especially CI/CD. The willingness-to-adoption gap highlights infrastructural and methodological barriers rather than a lack of interest.

9. Discussions

9.1. Challenges and implications

We now discuss the implications of the observed values of the measured quality metrics, our in-depth analysis, and their results on practical IoT development. We link those observations to real-world challenges, and we provide implications for IoT developers and practitioners.

Real-world challenge and observation 1: IoT development involves intricate hardware-software interactions, intricate data communication, and complex algorithms. Metrics such as LOC, #Classes, #Files, CC, HV, and WMC highlight the extensive and complex nature of code in IoT systems. The in-depth example of code analysis also illustrates the difficulty of understanding IoT systems' code.

Implication 1: Developers engaged in IoT projects should cultivate specialised skills, such as developing a deep understanding of both hardware and software aspects and expertise in efficient data communication protocols to navigate challenges posed by hardware-software interactions, data communication intricacies and complex algorithms.

Real-world challenge and Observation 2: Higher interdependence between different modules within a system and low maintainability in IoT systems, reflected in metrics like RFC, CBO, and MI, pose challenges in making modifications and maintaining the codebase.

Implication 2: Emphasising modular design and efficient code organisation allows for the encapsulation of functionality into distinct, manageable units, which is essential to effectively manage interdependence, maintainability, and extensive codebases in IoT systems.

Real-world challenge and Observation 3: The increased number of classes and files in IoT systems is driven by their distributed nature. The statistical computation on metrics reveals that IoT systems exhibit more extensive code compared to non-IoT systems. Metrics showcase a significant difference in the number of classes and files as presented in Table 6.

Implication 3: Developers engaged in IoT projects should prioritise the implementation of efficient organisational and structural practices, such as eliminating redundant code segments and optimising function libraries. These practices involve code size reduction.

Real-world challenge and Observation 4: To understand and maintain code quality in evolving IoT projects. Understanding the trade-off between metrics like RFC, CBO, LCOM, CC, and WMC in IoT systems provides developers with actionable insights, leading them to make informed decisions and specific actions in the development to enhance metric values.

Implication 4: Continuous monitoring of code metrics, coupled with a willingness to adapt coding practices based on the results of these metrics, is essential for ensuring the quality of the IoT systems.

9.2. *Threats to validity*

9.2.1. *Internal validity*

Using a limited set of quality metrics may not comprehensively represent software systems. We selected various categories of popular metrics to address this limitation, ensuring a more holistic perspective on static code analysis.

We acknowledge that metrics alone are insufficient for a comprehensive assessment of software quality; robust quality models grounded in well-defined metrics are essential for this purpose. Furthermore, our selection of metrics was constrained by the availability and capabilities of existing measurement tools. Nonetheless, our work represents an initial step in this direction by systematically measuring and comparing relevant metrics, thereby laying the groundwork for the development of more comprehensive and context-aware quality models in future research.

The choice of tools for measuring quality metrics may not align perfectly with the specific characteristics of IoT and non-IoT systems. To mitigate this concern, we employed two popular analysis tools instead of relying on a single tool, enhancing the accuracy of our results.

Our study focuses on various heterogeneous non-IoT systems, such as programming libraries, frameworks, databases, IDEs, games, scientific programs, etc. However, we acknowledge that more characteristics of these non-IoT projects could be integrated to select those systems, avoiding introducing biases or limitations due to inherent differences among these project types. We hope that future research will build upon our pioneering work, utilising more selection criteria to enhance the comprehensiveness and robustness of such analyses.

9.2.2. *External validity*

Discrepancies in the experience levels of developers working on IoT and non-IoT systems can impact the differences in software quality. To address this potential bias, we conducted manual analyses to ensure the quality of selected systems. We also identified and addressed outliers and anomalies that could affect the validity of our results.

Our work may be susceptible to overlooking external factors influencing code metrics, such as environmental changes, specific hardware configurations, or external dependencies. We acknowledge that ignoring such factors might limit the accuracy and applicability of our findings. Also, as noted in the related work section, the lack of prior studies limits the extent to which our work can be compared to others.

We also acknowledge that while our study includes projects spanning multiple domains and varying sizes, we acknowledge that the selected projects may not fully represent all IoT or non-IoT systems, which constitutes a potential threat to external validity. Similarly, our analysis focuses on a selected set of software metrics commonly used in prior research; the exclusion of other metrics may limit the generality of our findings, representing a potential threat to construct validity. Future work could address these limitations by including a broader and more diverse set of projects and metrics to further strengthen the generalizability and robustness of the results.

9.2.3. *Conclusion validity*

Comparing non-IoT and IoT systems is a complex task due to the differences in their code and structures. Despite these challenges, our work serves as an initial step in this comparative analysis and as a foundation for future research.

We study a limited subset of open-source systems from GitHub that could threaten the generalizability and representativeness of the findings. These systems might not encompass the full breadth of diversity present in IoT and non-IoT systems, potentially limiting the applicability of the conclusions. Recognising this limitation, we tried to mitigate this issue by considering the most extensive possible dataset available within the scope of our study to capture a diverse representation across different programming languages, frameworks, and project scales to achieve a more comprehensive understanding.

10. Conclusion

The increasing prevalence of IoT systems highlights the need for developers and researchers to assess the quality of their source code, given the challenges of resource-constrained environments and the complexity of specific hardware requirements. Since IoT systems often operate in critical domains such as healthcare and infrastructure management, their code directly influences functionality and reliability, highlighting the importance of thorough code assessment throughout the development lifecycle.

This study addresses the existing gap in research on IoT systems software quality by conducting a comparative analysis with non-IoT systems software, acknowledging the unique challenges posed by IoT's limited resources and distributed architectures. The study and findings highlight key differences in metrics such as complexity, cohesion, code size, and maintainability, indicating that developing IoT systems demands tailored best practices.

Given these disparities, we systematically compiled a set of best practices commonly used in non-IoT systems and customised a list specifically designed for IoT system development to address these distinctions. Also, we systematically selected and analysed 94 comparable IoT and non-IoT systems, providing comprehensive insights into their respective codebases. Our contributions include a method for choosing equivalent systems, computation and analysis of various metrics, an in-depth analysis of some IoT systems' code to show the complexity compared to non-IoT systems, and a revised list of software engineering best practices for IoT development, addressing observed challenges such as high complexity, low maintainability, and readability issues.

We acknowledge that recognising those metrics alone is insufficient for a complete quality assessment; this work sets the stage for future research, emphasising the implementation and evaluation of quality models to evaluate the quality of IoT systems. Overall, this study enhances our understanding of the software quality of IoT systems and provides insights for developing more resilient and efficient IoT systems across various domains.

Future research can further enrich our findings on software quality in IoT systems. Currently, the focus is on open-source systems available on GitHub; further research can extend the scope beyond that to include more systems and have a more diverse range of systems. While our study compared code quality between IoT and non-IoT systems via metrics, further investigations can build quality models and repeat the comparison process. Also, our study relies on GitHub stars and forks as primary criteria to match IoT and non-IoT systems, future work could incorporate additional matching dimensions for selection systems from GitHub, such as functional domain, code complexity, or number of dependencies, to further enhance the representativeness and robustness of the analysis. For outlier detection, future work could incorporate standardised statistical techniques to complement the current visual inspection approach and further enhance the robustness of the analysis. Other aspects to explore in IoT systems include usability, security, and performance. Also, implementing identified best practices for non-IoT on IoT systems and evaluating their effects is necessary to address identified issues, such as complexity, size, coupling, etc., on IoT systems. Evaluating their effects is needed to address complexity, size, and coupling.

Data availability

We have shared the link to the data/code in the manuscript.

CRediT authorship contribution statement

Nour Khezemi: Writing – original draft, Methodology, Investigation, Data curation, Conceptualization; **Sikandar Ejaz:** Writing – review & editing, Writing – original draft, Validation, Methodology, Conceptualization; **Naouel Moha:** Writing – review & editing, Validation, Supervision, Conceptualization; **Yann-Gaël Guéhéneuc:** Writing – review & editing, Validation, Supervision, Conceptualization.

Declaration of interests

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] S.Y.Y. Tun, S. Madanian, F. Mirza, Internet of things (IoT) applications for elderly care: a reflective review, Aging Clin. Exp. Res. 33 (4) (2021) 855–867. https://doi.org/10.1007/s40520-020-01545-9
[2] J.B. Minani, F. Sabir, N. Moha, Y.-G. Guéhéneuc, A systematic review of IoT systems testing: objectives, approaches, tools, and challenges, IEEE Trans. Software Eng. 50 (4) (2024) 785–815. https://doi.org/10.1109/TSE.2024.3363611
[3] F. Ihirwe, D. Di Ruscio, S. Gianfranceschi, A. Pierantonio, Assessing the quality of low-code and model-driven engineering platforms for engineering IoT systems, in: 2022 IEEE 22Nd International Conference on Software Quality, Reliability and Security (QRS), 2022, pp. 583–594. https://doi.org/10.1109/QRS57517.2022.00065
[4] I.M. Gorbachenko, E.V. Gorshkov, T.N. Filipkina, Application of various metrics to assess the program code quality, J. Phys. Conf. Ser. 1679 (3) (2020) 032087. https://doi.org/10.1088/1742-6596/1679/3/032087
[5] G. Giray, B. Tekinerdogan, E. Tüzün, IoT system development methods, in: Internet of Things, Chapman and Hall/CRC, 2017, pp. 141–159.
[6] M. Fahmideh, A. Ahmad, A. Behnaz, J. Grundy, W. Susilo, Software engineering for internet of things: the Practitioners' perspective, IEEE Trans. Software Eng. 48 (8) (2022) 2857–2878. https://doi.org/10.1109/TSE.2021.3070692
[7] L.D. Xu, W. He, S. Li, Internet of things in industries: a survey, IEEE Trans. Ind. Inf. 10 (4) (2014) 2233–2243. https://doi.org/10.1109/TII.2014.2300753
[8] F. Dahlqvist, M. Patel, A. Rajko, J. Shulman, Growing opportunities in the internet of things, McKinsey & Company 22 (2019).
[9] X. Larrucea, A. Combelles, J. Favaro, K. Taneja, Software engineering for the internet of things, IEEE Softw. 34 (1) (2017) 24–28. https://doi.org/10.1109/MS.2017.28
[10] M. Klima, M. Bures, K. Frajtak, V. Rechtberger, M. Trnka, X. Bellekens, T. Cerny, B.S. Ahmed, Selected code-quality characteristics and metrics for internet of things systems, IEEE Access 10 (2022) 46144–46161. https://doi.org/10.1109/ACCESS.2022.3170475
[11] F. Corno, L. De Russis, J.P. Sáenz, How is open source software development different in popular IoT projects?, IEEE Access 8 (2020) 28337–28348. https://doi.org/10.1109/ACCESS.2020.2972364
[12] D. Barrera, C. Bellman, P. Van Oorschot, Security best practices: a critical analysis using IoT as a case study, ACM Trans. Priv. Secur. 26 (2) (2023). https://doi.org/10.1145/3563392

[13] C. Bellman, P.C. van Oorschot, Best Practices for IoT Security: What Does That Even Mean?, 2020, arXiv:2004.12179
[14] W. Moedt van Bolhuis, R. Bernsteiner, M. Hall, A. Fruhling, Enhancing IoT project success through agile best practices, ACM Trans. Internet Things 4 (1) (2023). https://doi.org/10.1145/3568170
[15] A. Goknil, P. Nguyen, S. Sagar, D. Politaki, H. Niavis, K.J. Pedersen, A. Suyuthi, A. Anand, A. Ziegenbein, A systematic review of data quality in CPS and IoT for industry 4.0, ACM Comput. Surv. 55 (14s) (2023). https://doi.org/10.1145/3593043
[16] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami, Internet of things (IoT): a vision, architectural elements, and future directions, Future Gen. Comput. Syst. 29 (7) (2013) 1645–1660. Including Special sections: Cyber-enabled Distributed Computing for Ubiquitous Cloud and Network Services & Cloud Computing and Scientific Applications - Big Data, Scalable Analytics, and Beyond, https://doi.org/10.1016/j.future.2013.01.010
[17] A. Taivalsaari, T. Mikkonen, A roadmap to the programmable world: software challenges in the IoT era, IEEE Softw. 34 (1) (2017) 72–80. https://doi.org/10.1109/MS.2017.26
[18] H. Borges, M. Tulio Valente, What's in a github star? understanding repository starring practices in a social coding platform, J. Syst. Softw. 146 (2018) 112–129. https://doi.org/10.1016/j.jss.2018.09.016
[19] C. Politowski, F. Petrillo, J.E. Montandon, M.T. Valente, Y.-G. Guéhéneuc, Are game engines software frameworks? a three-perspective study, J. Syst. Softw. 171 (2021) 110846. https://doi.org/10.1016/j.jss.2020.110846
[20] N. Nazar, Y. Hu, H. Jiang, Summarizing software artifacts: a literature review, J. Comput. Sci. Technol. 31 (5) (2016) 883–909. https://doi.org/10.1007/s11390-016-1671-1
[21] G. Baldini, A. Skarmeta, E. Fourneret, R. Neisse, B. Legeard, F. Le Gall, Security certification and labelling in internet of things, in: 2016 IEEE 3rd World Forum on Internet of Things (WF-IoT), 2016, pp. 627–632. https://doi.org/10.1109/WF-IoT.2016.7845514
[22] M. Alenezi, K. Almustafa, Empirical analysis of the complexity evolution in open-source software systems, Intern. J. Hybrid Inf. Technol. 8 (2) (2015) 257–266.
[23] Understand, https://www.scitools.com/.
[24] Multimetric, https://github.com/priv-kweihmann/multimetric.
[25] L. Ardito, R. Coppola, L. Barbato, D. Verga, A tool-Based perspective on software code maintainability metrics: a systematic literature review, Sci. Program. 2020 (1) (2020) 8840389. https://doi.org/10.1155/2020/8840389
[26] C. Ebert, J. Cain, G. Antoniol, S. Counsell, P. Laplante, Cyclomatic complexity, IEEE Softw. 33 (6) (2016) 27–29.
[27] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Trans. Softw. Eng. 20 (6) (1994) 476–493. https://doi.org/10.1109/32.295895
[28] W. Li, Another metric suite for object-oriented programming, J. Syst. Softw. 44 (2) (1998) 155–162. https://doi.org/10.1016/S0164-1212(98)10052-3
[29] Lcom, http://www.virtualmachinery.com/jhawkmetricsclass.htm.
[30] G. Lajios, D. Schmedding, F. Volmering, Supporting language conversion by metric based reports, in: 2008 12th European Conference on Software Maintenance and Reengineering, 2008, pp. 314–316. https://doi.org/10.1109/CSMR.2008.4493336
[31] I. Heitlager, T. Kuipers, J. Visser, A practical model for measuring maintainability, in: 6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007), 2007, pp. 30–39. https://doi.org/10.1109/QUATIC.2007.8
[32] R. Dunford, Q. Su, E. Tamang, The pareto principle, The Plymouth Student Scientist 7 (2014) 140–148.
[33] Z. Hanusz, J. Tarasinska, W. Zielinski, Shapiro-wilk test with known mean, REVSTAT-Stat. J. 14 (1) (2016) 89-100. https://doi.org/10.57805/revstat.v14i1.180
[34] T.W. MacFarland, J.M. Yates, Mann–Whitney U Test, Springer International Publishing, Cham, 2016, pp. 103–132. https://doi.org/10.1007/978-3-319-30634-6_4
[35] D.M. Hawkins, Identification of Outliers, 11, Springer, 1980.
[36] F. Brito e Abreu, M. Goulao, Coupling and cohesion as modularization drivers: are we being over-persuaded?, in: Proceedings Fifth European Conference on Software Maintenance and Reengineering, 2001, pp. 47–57. https://doi.org/10.1109/CSMR.2001.914968
[37] GitHub - Samsung/TizenRT: TizenRT is a lightweight RTOS-based platform to support low-end IoT devices — github.com, (https://github.com/Samsung/TizenRT.git) [Accessed 13-05-2025].
[38] GitHub - apache/druid: Apache Druid: a high performance real-time analytics database. — github.com, (https://github.com/apache/druid.git) [Accessed 13-05-2025].
[39] eclipse-ditto/ditto, https://github.com/eclipse-ditto/ditto.
[40] GitHub - espressif/esp-mqtt: ESP32 mqtt component — github.com, (https://github.com/espressif/esp-mqtt.git) [Accessed 13-05-2025].
[41] GitHub - quarnster/SublimeGDB: GDB integration with Sublime Text 2 — github.com, (https://github.com/quarnster/SublimeGDB.git) [Accessed 13-05-2025].
[42] GitHub - project-chip/connectedhomeip: Matter (formerly Project CHIP) creates more connections between more objects, simplifying development for manufacturers and increasing compatibility for consumers, guided by the Connectivity Standards Alliance. — github.com, (https://github.com/project-chip/connectedhomeip.git) [Accessed 13-05-2025].
[43] GitHub - rwaldron/johnny-five: JavaScript Robotics and IoT programming framework, developed at Bocoup. — github.com, (https://github.com/rwaldron/johnny-five.git) [Accessed 13-05-2025].
[44] GitHub - arthenica/ffmpeg-kit: FFmpeg Kit for applications. Supports Android, Flutter, iOS, Linux, macOS, React Native and tvOS. Supersedes MobileFFmpeg, flutter_ffmpeg and react-native-ffmpeg. — github.com, (https://github.com/arthenica/ffmpeg-kit.git) [Accessed 13-05-2025].
[45] GitHub - ARMmbed/mbed-os: Arm Mbed OS is a platform operating system designed for the internet of things — github.com, (https://github.com/ARMmbed/mbed-os.git) [Accessed 13-05-2025].
[46] GitHub - DarthFubuMVC/fubumvc: A front-controller style MVC framework for .NET — github.com, (https://github.com/DarthFubuMVC/fubumvc.git) [Accessed 13-05-2025].
[47] GitHub - flomesh-io/pipy: Pipy is a programmable proxy for the cloud, edge and IoT. — github.com, (https://github.com/flomesh-io/pipy.git) [Accessed 13-05-2025].
[48] GitHub - mgba-emu/mgba: mGBA Game Boy Advance Emulator — github.com, (https://github.com/mgba-emu/mgba.git) [Accessed 13-05-2025].
[49] GitHub - greghesp/assistant-relay: A Node.js server that allows for sending commands to Google Home/Assistant from endpoints — github.com, (https://github.com/greghesp/assistant-relay.git) [Accessed 13-05-2025].
[50] GitHub - dachev/node-cld: Language detection for Javascript (Node). Based on the CLD2 (Compact Language Detector) library from Google. — github.com, (https://github.com/dachev/node-cld.git). [Accessed 13-05-2025].
[51] GitHub - wmira/react-icons-kit: React Svg Icons — github.com, (https://github.com/wmira/react-icons-kit.git) [Accessed 13-05-2025].
[52] GitHub - ClassiCube/ClassiCube — github.com, (https://github.com/ClassiCube/ClassiCube.git) [Accessed 13-05-2025].
[53] kymjs/CJFrameForAndroid, https://github.com/kymjs/CJFrameForAndroid/blob/master/cjframe/src/org/kymjs/cjframe/bean/AndroidPackage.java.
[54] D.R. Hanson, Simple code optimizations, Softw. Prac. Exper. 13 (8) (1983) 745–763.
[55] T.J.K. Edler von Koch, B. Franke, P. Bhandarkar, A. Dasgupta, Exploiting function similarity for code size reduction, SIGPLAN Not. 49 (5) (2014) 85-94. https://doi.org/10.1145/2666357.2597811
[56] Cppcheck, 2018, (https://github.com/danmar/cppcheck).
[57] C. Lefurgy, E. Piccininni, T. Mudge, Reducing code size with run-time decompression, in: Proceedings Sixth International Symposium on High-Performance Computer Architecture. HPCA-6 (Cat. No.PR00550), 2000, pp. 218–228. https://doi.org/10.1109/HPCA.2000.824352
[58] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.
[59] Q.D. Soetens, S. Demeyer, Studying the effect of refactorings: a complexity metrics perspective, in: 2010 Seventh International Conference on the Quality of Information and Communications Technology, 2010, pp. 313–318. https://doi.org/10.1109/QUATIC.2010.58
[60] C. Mayer, The art of clean code: best practices to eliminate complexity and simplify your life, No Starch Press, 2022.
[61] Y. Zhang, S. Sun, D. Zhang, J. Qiu, Z. Tian, A consistency-guaranteed approach for internet of things software refactoring, Int. J. Distrib. Sens. Netw. 16 (1) (2020) 1550147720901680. https://doi.org/10.1177/1550147720901680

[62] E. Tempero, K. Blincoe, D. Lottridge, An experiment on the effects of modularity on code modification and understanding, in: Proceedings of the 25Th Australasian Computing Education Conference, ACE '23, Association for Computing Machinery, New York, NY, USA, 2023, p. 105-112. https://doi.org/10.1145/3576123.3576138
[63] J.P. Kearney, R.L. Sedlmeyer, W.B. Thompson, M.A. Gray, M.A. Adler, Software complexity measurement, Commun. ACM 29 (11) (1986) 1044-1050. https://doi.org/10.1145/7538.7540
[64] R.D. Banker, G.B. Davis, S.A. Slaughter, Software development practices, software complexity, and software maintenance performance: a field study, Manage. Sci. 44 (4) (1998) 433–450. https://doi.org/10.1287/mnsc.44.4.433
[65] C. Walls, R. Breidenbach, Spring in Action: Updated for Spring 2.0, Dreamtech Press, 2007.
[66] H. Alrubaye, D. Alshoaibi, E. Alomar, M.W. Mkaouer, A. Ouni, How does library migration impact software quality and comprehension? an empirical study, in: S. Ben Sassi, S. Ducasse, H. Mili (Eds.), Reuse in Emerging Software Engineering Practices, Springer International Publishing, Cham, 2020, pp. 245–260.
[67] E.A. AlOmar, H. AlRubaye, M.W. Mkaouer, A. Ouni, M. Kessentini, Refactoring practices in the context of modern code review: an industrial case study at xerox, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 348–357. https://doi.org/10.1109/ICSE-SEIP52600.2021.00044
[68] E. Jung, I. Cho, S.M. Kang, An agent modeling for overcoming the heterogeneity in the IoT with design patterns, in: J.J. J.H. Park, H. Adeli, N. Park, I. Woungang (Eds.), Mobile, Ubiquitous, and Intelligent Computing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 69–74.
[69] B. Du Bois, S. Demeyer, J. Verelst, Refactoring - improving coupling and cohesion of existing code, in: 11th Working Conference on Reverse Engineering, 2004, pp. 144–151. https://doi.org/10.1109/WCRE.2004.33
[70] D. Posnett, A. Hindle, P. Devanbu, et al., A simpler model of software readability, in: Proceedings of the 8th Working Conference on Mining Software Repositories, MSR '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 73-82. https://doi.org/10.1145/1985441.1985454
[71] R.P.L. Buse, W.R. Weimer, Learning a metric for code readability, IEEE Trans. Softw. Eng. 36 (4) (2010) 546–558. https://doi.org/10.1109/TSE.2009.70
[72] V. Piantadosi, F. Fierro, S. Scalabrino, A. Serebrenik, R. Oliveto, How does code readability change during software evolution?, Empir. Softw. Eng. 25 (6) (2020) 5374–5412. https://doi.org/10.1007/s10664-020-09886-9
[73] I.B. Sampaio, L. Barbosa, Software readability practices and the importance of their teaching, in: 2016 7th International Conference on Information and Communication Systems (ICICS), 2016, pp. 304–309. https://doi.org/10.1109/IACS.2016.7476069
[74] X. Wang, L. Pollock, K. Vijay-Shanker, Automatic segmentation of method code into meaningful blocks to improve readability, in: 2011 18th Working Conference on Reverse Engineering, 2011, pp. 35–44. https://doi.org/10.1109/WCRE.2011.15
[75] T. Sedano, Code readability testing, an empirical study, in: 2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET), 2016, pp. 111–117. https://doi.org/10.1109/CSEET.2016.36
[76] B. Gergel, E. Stroulia, M. Smit, H.J. Hoover, Maintainability and source code conventions: an analysis of open source projects (2011).
[77] J.B. Bowles, Code from requirements: new productivity tools improve the reliability and maintainability of software systems, in: Annual Symposium Reliability and Maintainability, 2004 - RAMS, 2004, pp. 68–72. https://doi.org/10.1109/RAMS.2004.1285425
[78] A. Moin, M. Challenger, A. Badii, S. Günnemann, A model-driven approach to machine learning and software modeling for the IoT, Softw. Syst. Model. 21 (3) (2022) 987–1014. https://doi.org/10.1007/s10270-021-00967-x
[79] P. Hegedus, Revealing the effect of coding practices on software maintainability, in: 2013 IEEE International Conference on Software Maintenance, 2013, pp. 578–581. https://doi.org/10.1109/ICSM.2013.99
[80] H.K. Jun, M.E. Rana, Evaluating the impact of design patterns on software maintainability: an empirical evaluation, in: 2021 Third International Sustainability and Resilience Conference: Climate Change, 2021, pp. 539–548. https://doi.org/10.1109/IEEECONF53624.2021.9668025
[81] I. Samoladas, I. Stamelos, L. Angelis, A. Oikonomou, Open source software development should strive for even greater code maintainability, Commun. ACM 47 (10) (2004) 83-87. https://doi.org/10.1145/1022594.1022598
[82] J. Visser, S. Rigal, G. Wijnholds, P. Van Eck, R. van der Leek, Building Maintainable Software: Ten Guidelines For Future-Proof Code, " O'Reilly Media, Inc.", 2016.
[83] B. Anda, Assessing software system maintainability using structural measures and expert assessments, in: 2007 IEEE International Conference on Software Maintenance, 2007, pp. 204–213. https://doi.org/10.1109/ICSM.2007.4362633