Evaluating Microservices Maintainability: A Classification System Using Code Metrics and ISO/IEC 250xy Standards

Oğuzcan ÖZDEMİR
ozdemiro20@itu.edu.tr
Istanbul Technical University
Department of Computer Engineering
Istanbul, Turkey
ASELSAN Inc.
Ankara, Turkey

Feza BUZLUCA
buzluca@itu.edu.tr
Istanbul Technical University
Department of Computer Engineering
Istanbul, Turkey

ABSTRACT

In the rapidly evolving landscape of software engineering, Microservice Architecture (MSA) has emerged as a pivotal approach, renowned for its modular structure, operational efficiency, scalability, and flexibility. Despite the extensive research on MSA development, and numerous studies dedicated to evaluating the maintainability of object-oriented programs, the focus on the quality of microservice-based systems remains notably limited. This study introduces an innovative model for evaluating the maintainability of microservices, a key element in MSA. Our approach, grounded in code metrics analysis, aligns with the ISO/IEC 250xy standards SQuaRE (System and Software Quality Requirements and Evaluation). It specifically targets testability and modifiability, integral components of maintainability. We carefully chose essential code metrics that precisely encapsulate the varied characteristics of MSA. The model employs clustering algorithms to categorize the quality characteristics of MSA into three distinct groups: low, medium and high. Our project's primary goal is to identify microservices with low maintainability values. Our methodology was applied to a range of open-source MSA-designed applications, demonstrating its effectiveness and yielding promising outcomes. In our results, we achieved a recall of 83.33% and a precision of 71.43%. This research contributes a novel viewpoint in assessing microservice maintainability and offers a valuable resource for software architects and developers. It aims to improve the overall quality and longevity of software systems within the MSA.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**; *Software maintenance tools*; Software development techniques.

KEYWORDS

Microservice Architecture (MSA), Maintainability, Microservice Quality, Testability, Modifiability, Software Evaluation

ICSCA 2024, February 01–03, 2024, Bali Island, Indonesia

ACM ISBN 979-8-4007-0832-9/24/02
https://doi.org/10.1145/3651781.3651790

ACM Reference Format:
Oğuzcan ÖZDEMİR and Feza BUZLUCA. 2024. Evaluating Microservices Maintainability: A Classification System Using Code Metrics and ISO/IEC 250xy Standards. In *2024 13th International Conference on Software and Computer Applications (ICSCA 2024), February 01–03, 2024, Bali Island, Indonesia.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3651781.3651790

1 INTRODUCTION

The current landscape of software development has increasingly gravitated towards Microservice Architecture (MSA), favored for its scalability, adaptability, and accelerated delivery [9]. Despite its rising popularity, the maintainability of microservices software—a crucial factor for the long-term effectiveness and reliability of these systems—often receives insufficient attention [6][17]. Our study focuses on evaluating the maintainability of microservices software, adhering to the ISO/IEC 250xy standards SQuaRE (System and Software Quality Requirements and Evaluation), with an emphasis on testability and modifiability [1].

Our research employs code metrics to quantitatively assess microservices attributes, categorizing their maintainability as low, medium or high. In our study, we initially applied clustering algorithms to classify the modifiability and testability values, setting the stage for determining maintainability. We then computed the overall maintainability score of the applications using these metrics. To validate our model, we implemented it on various open-source, microservice-based projects. Our model proved highly effective in identifying projects with poor maintainability, pinpointing those requiring improvement. In the evaluation of our model, following a comparison with expert assessments, we achieved a recall of 83.33% and a precision of 71.43% in our performance metrics. This juxtaposition with expert evaluations highlights the effectiveness and reliability of our approach.

This approach not only highlights the crucial role of maintainability in the lifecycle of microservices but also provides insights into the factors influencing it. By proposing a standardized, metric-based framework, aligned with the ISO/IEC 250xy standards, our study enhances the understanding of microservices software maintainability through the analysis of testability and modifiability. These findings are crucial for developing strategies to improve the maintainability of microservices, thereby serving as a valuable guide for practitioners and researchers. Overall, this paper presents a succinct yet comprehensive evaluation of microservices software maintainability, emphasizing its importance in the sustainable and reliable

development and deployment of microservice systems, contributing significantly to the field of software engineering.

2 RELATED WORK

The shift towards microservice architectures in software engineering has prompted the need for new methodologies to assess software quality, a trend gaining traction in academic and industrial realms [8]. Microservice systems, despite being a relatively recent area of study compared to traditional service-oriented architectures, pose unique challenges for quality assessment due to their decentralized and scalable nature. This gap highlights the necessity for specialized, automated evaluation methods for these complex systems. Current research is crucial in developing suitable metrics and models for effectively assessing the quality of microservice architectures.

Bogner, Wagner, and Zimmermann (2017) conducted an in-depth analysis of maintainability quality models for service- and microservice-based systems, offering a comprehensive approach to examining maintainability within computer science, particularly focusing on service- and microservice-based systems. Their research intricately explores various dimensions of maintainability, shedding light on the complexities and challenges inherent in these systems and providing valuable insights into the intricate factors contributing to the maintainability of microservice architectures, thus offering a robust framework for further research in this area [6].

In the same vein, Bogner, Wagner, and Zimmermann's separate 2017 study emphasized the automatic measurement of maintainability in these systems. They presented a nuanced analysis of key factors influencing maintainability, with a special focus on methodologies for automatic measurement. This work is crucial for its contribution to understanding how maintainability in microservice systems can be quantitatively assessed and improved, providing a foundational perspective for evaluating and enhancing maintainability, making it an indispensable reference for studies in this domain [4].

In 2019, Bogner, Schlinger, Wagner, and Zimmermann explored a modular methodology to derive maintainability metrics from microservices' runtime data. Their research is pivotal in the field of microservices for presenting a unique approach to calculating maintainability metrics. By incorporating runtime data, this study breaks new ground in understanding and improving microservice maintainability, enriching the field by providing a comprehensive understanding of approaches used in deriving maintainability metrics, thereby augmenting the existing body of knowledge on microservice architectures [3].

Yilmaz and Buzluca's (2021) study into a fuzzy quality model for measuring microservice architecture maintainability presents a groundbreaking approach using fuzzy logic. This study is a significant contribution to the field, offering a novel perspective in evaluating microservice architectures. By integrating fuzzy logic into the quality assessment process, the research opens new avenues for measuring and enhancing the maintainability of microservice architectures, marking a significant advance in the field and providing fresh insights and methodologies for the evaluation of microservice maintainability [18].

3 PROPOSED APPROACH

3.1 General Structure of the Model

We propose a hierarchical quality model for microservice architecture design, based on the ISO/IEC 250xy standards. The model decomposes high-level quality characteristics into measurable metrics, providing a comprehensive view of microservice quality. To assess the quality of microservice software, we use clustering algorithms to group microservices into different categories based on their quality characteristics.

In this study, we constructed our model based on the Quality Measurement Reference Model (QM-RM) introduced in ISO/IEC 25020. The QM-RM delineates the interplay between a quality model and the creation of Quality Measures (QMs) from Quality Measure Elements (QMEs). By leveraging the QM-RM as our base, we established a methodical and exhaustive approach for evaluating software quality, enabling a nuanced and holistic assessment of a software product's overall quality. In our approach, the code metrics we employed serve as the Quality Measure Elements (QMEs), while our focus on Testability and Modifiability are reflected as the Quality Measures (QMs). This integration ensures that our evaluation metrics are deeply rooted in the principles set forth by the QM-RM framework.

Building on this foundation, we employ clustering algorithms to categorize microservices based on their distinct quality characteristics. This methodology not only aids in the systematic assessment of microservices but also highlights those requiring additional focus or enhancement. By identifying and classifying microservices in this manner, our model facilitates a more targeted approach to improving and maintaining the quality of microservice architectures. This use ensures a dynamic and adaptive model, capable of addressing the evolving needs and complexities inherent in microservice-based systems.

3.2 Selecting Quality Characteristics

Maintainability, as characterized by ISO/IEC 25010, is the measure of how effectively and efficiently a product or system can be modified for improvement, correction, or adaptation to environmental changes and shifting requirements [1]. According to ISO/IEC 25010, maintainability encompasses five sub-characteristics: modularity, reusability, analyzability, modifiability, and testability [1]. This study focuses on evaluating the maintainability of microservices, particularly emphasizing two critical sub-characteristics: testability and modifiability. By concentrating on these aspects, we aim to provide a more detailed understanding of how microservices can be effectively maintained and adapted in response to evolving requirements and technical challenges.

3.2.1 Modifiability. Modifiability is a key aspect in software systems, especially in microservices architectures. In this context, modifiability refers to the ability to make changes to the system without causing disruptions or introducing new errors [7]. This is incredibly important given the dynamic nature of microservices, where ongoing evolution and adjustment to changing requirements are common [16]. Microservices, being small, independent units that communicate through well-defined APIs, require the capability to be modified without impacting the overall system [7].

The importance of modifiability in microservices architecture is highlighted by the necessity for seamless updates and changes to individual services without triggering cascading failures or disruptions in other services [7]. The independence in development and deployment of microservices means that altering one service should not affect others, which is vital for maintaining each service's autonomy [16]. Additionally, in distributed environments like cloud computing, where microservices are often employed, coordinating changes across multiple services poses challenges. Modifiability is crucial in easing this process and minimizing error risks [7].

Literature also underscores the significance of finding a balance between decentralization and standardization in microservices architectures for ensuring evolvability [2]. This balance is essential for modifiability, as it permits localized changes in individual services while upholding a degree of standardization for overall system coherence [2]. Furthermore, automating tasks related to modifiability, such as measuring maintainability and extracting microservices from monolithic applications, is emphasized as key due to the complexity and costs associated with these processes [15] [5].

In conclusion, modifiability is a critical component of microservices architecture, facilitating adaptability and maintenance in these dynamic, distributed systems. It ensures that changes to individual services can be implemented without disrupting the entire system, thereby supporting the agility and autonomy of microservices.

3.2.2 Testability. Testability in microservice architecture refers to the ease with which developers can test individual services and the system as a whole. It is crucial for ensuring the quality and reliability of the overall system. Microservices, being small and autonomous, can be challenging to test in isolation. However, strategies such as designing services with well-defined interfaces, using dependency injection, and writing unit tests can enhance the testability of microservices [10]. Improved testability allows for quick bug identification and resolution, ultimately leading to a more reliable system [11].

Furthermore, testability plays a vital role in reducing the cost of maintenance and support for microservices. Well-tested microservices make it easier for developers to identify and rectify issues, thereby reducing the time and resources required to maintain the system [11]. By focusing on improving testability, developers can build more reliable and maintainable microservice systems, thus enhancing their overall effectiveness and efficiency.

In conclusion, testability is an essential aspect of microservices architecture. By implementing measures to enhance testability, developers can ensure the reliability and maintainability of their microservice systems, ultimately leading to improved performance and reduced maintenance costs.

3.3 Metrics to Measure Microservice Maintainability

Code metrics are measurements used to assess the quality of code, such as the number of lines of code (LOC) and cyclomatic complexity (CC) [12]. These metrics can be collected using static code analysis tools like Sonargraph or can be calculated manually. Source code metrics play a crucial role in evaluating the quality of a codebase [14]. By integrating these metrics in a contextual and systematic manner, developers can perform analyses and make inferences about the code [13].

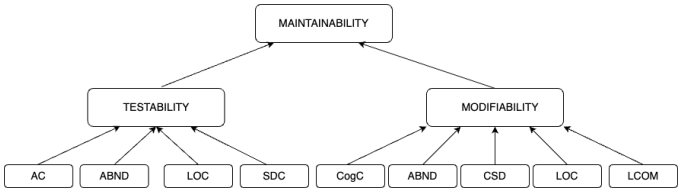

Figure 1: Proposed Hierarchical Model for Measuring Microservice Maintability

In our study, we analyzed several code metrics to accurately measure testability and modifiability. We utilized a total of six different code metrics for this purpose. Detailed descriptions of the metrics we used are provided in Table 1.

3.4 Proposed Hierarchical Model

In our research, we utilized a tailored set of metrics for the calculation of testability and modifiability values of microservices. For testability, the metrics involved were Lines of Code (LOC), Average Complexity (AC), Average Block Nesting Depth (ABND), and Service Dependency Count (SDC). For determining modifiability, we employed Lines of Code (LOC), Cognitive Complexity (CogC), Average Block Nesting Depth (ABND), Code Smell Density (CSD), and Lack of Cohesion in Methods (LCOM). Based on these calculated values, we further determined the maintainability value of each microservice, adhering to the ISO/IEC 25020 standards. Our methodology incorporates a hierarchical model, which is depicted in Figure 1.

Building on this foundation, we initially employed the k-means clustering algorithm to each microservice software's testability and modifiability values. This algorithm enabled the segmentation of these values into three distinct groups: Low, Medium, and High. This stratification was essential for a detailed understanding of each microservice's performance in terms of testability and modifiability.

To further enhance our hierarchical structure, we utilized these computed testability and modifiability values. We reapplied the k-means clustering algorithm, leveraging it to calculate the maintainability values for each service. This application of the k-means algorithm in our hierarchical model allowed us to accurately evaluate and categorize the maintainability of microservices, providing an in-depth analysis that is both comprehensive and aligned with industry standards.

3.5 Measurement Of The Maintainability Value of Microservices

In this study, we present a novel methodology specifically designed for evaluating the maintainability of microservices in modern software systems. This approach is structured to directly address the unique challenges presented by microservices architecture, ensuring a comprehensive assessment.

Table 1: Metrics and Definitions

Metric Name	Metric Definition	Purpose of Usage
Lines Of Code (LOC)	Lines Of Code (LOC) is a software metric used to measure the size of a software program by counting the number of lines in its source code.	A higher LOC generally indicates a larger, more complex codebase, which can lead to challenges in maintenance. Larger codebases are often more difficult to understand, more prone to bugs, and can be cumbersome to refactor, impacting their maintainability.
Average Block Nesting Depth (ABND)	Average Block Nesting Depth (ABND) is a metric that measures the average depth of nested blocks of code (like loops or conditionals) within a software program. It calculates how deeply blocks of code are nested within each other on average across the entire codebase.	A higher ABND indicates deeper nesting of code blocks, which can lead to increased complexity and difficulty in understanding and modifying the code. This complexity can negatively affect the maintainability of the software, as deeply nested code is often harder to read, debug, and test.
Service Dependency Count (SDC)	Service Dependency Count (SDC) is a metric that measures the number of other microservices that a microservice depends on. It is calculated by counting the number of unique microservices that are called or referenced by a microservice.	SDC is an important metric for evaluating the quality and maintainability of a microservices architecture. A high SDC indicates that a microservice is tightly coupled to other microservices. This can make the microservice more difficult to understand, maintain, and evolve.
Average Complexity (AC)	Average Complexity (AC) measures the mean complexity of software elements, usually calculated by averaging the complexities of individual methods or functions within a codebase.	High AC often indicates that the codebase contains numerous complex elements, which can complicate maintenance tasks like debugging, testing, and implementing new features.
Code Smell Density (CSD)	Code Smell Density (CSD) is a metric used to assess the frequency of code smells within a software codebase. Code smells are patterns in the code that, while not technically incorrect, suggest design problems.	A higher CSD indicates a larger presence of potential design and quality issues within the code, which can lead to problems with maintainability. Code smells can make the code more difficult to understand, extend, or modify, and they often increase the risk of bugs or errors during future development.
Cognitive Complexity (CogC)	Cognitive Complexity (CogC) is a metric that measures the difficulty of understanding a unit of code. It is calculated by counting the number of breaks in the linear flow of the code.	CogC is an important metric for evaluating the quality and maintainability of a software system. A high CogC indicates that the code is difficult to understand, which can lead to errors, bugs, and maintenance problems.
Lack of Cohesion in Methods (LCOM)	Lack of Cohesion in Methods (LCOM) is a software metric measuring the cohesion degree in a class by analyzing the relationships between its methods and fields. Higher LCOM scores indicate less cohesion.	In the context of microservices, LCOM helps in evaluating the modifiability aspect of maintainability. A high LCOM in a microservice class suggests potential improvement areas for better modifiability, leading to enhanced overall maintainability.

The process begins with the application of the k-means clustering algorithm to the segregation of code metric values. This algorithm, widely acclaimed for its effectiveness in data segmentation, operates by partitioning the dataset into distinct clusters. In our case, these are three clusters representing low, medium, and high metric values. The k-means algorithm functions by initially assigning random centroids and then iteratively adjusting these centroids to minimize the variance within each cluster, thereby categorizing each code metric into a specific group. This technique offers a nuanced understanding, as demonstrated in the categorized groups of each code metric for the project tested, outlined in Table 4.

Further, in assessing the testability of microservices, the same k-means algorithm is employed to analyze key metrics such as Lines of Code (LOC), Average Complexity (AC), Average Block Nesting Depth (ABND), and Service Dependency Count (SDC). The algorithm segregates these metrics into three levels of testability - low, medium, and high - by creating clusters based on the similarities in their values. This clustering is critical for an accurate evaluation

of the ease of testing each microservice, effectively reflecting their inherent complexity and interdependencies.

The methodology then shifts its focus to the modifiability of the microservices. Here, different metrics, including LOC, Cognitive Complexity (CogC), ABND, Code Smell Density (CSD) and Lack of Cohesion in Methods(LCOM) are subjected to the k-means clustering algorithm. This algorithm's application facilitates the categorization of these metrics into three distinct levels of modifiability. The process of clustering, which involves the iterative adjustment of centroids to form well-defined groups, is instrumental in determining the adaptability of each microservice to changes and modifications.

In the final stage, the methodology integrates the results from the testability and modifiability assessments. This integration is achieved by reapplying the k-means clustering algorithm, combining the scores from the previous phases into a unified analysis. This results in a comprehensive categorization of microservices into various levels of maintainability. The use of the k-means algorithm in this final phase follows the same principle of minimizing within-cluster variance, thereby offering a holistic view of the microservices' long-term sustainability and efficiency by considering both their testability and modifiability.

Overall, this methodology provides a detailed and nuanced evaluation of microservices, encompassing crucial aspects such as testability, modifiability, and maintainability. By employing the k-means clustering algorithm consistently across different phases, the methodology presents a systematic and quantitative approach to address the complexities of microservices architecture, thus enhancing the long-term functionality and success of software systems.

4 EXPERIMENTS AND RESULTS

In our study, we collected and classified code metrics from three different open-source microservice projects. We tested our model on the TrainTicket project. The TrainTicket project, developed on a microservice architecture foundation, has been chosen as the application area for our model. In this open-source system, we conducted a detailed examination of each of the 34 microservices. We have computed the Modifiability, Testability, and Maintainability values for each service using our model, these figures and the code metrics of the examined services are presented in table 4. Testability, modifability and maintability values are categorized as 'Low' (L), 'Medium' (M), and 'High' (H) for ease of interpretation.

To test the accuracy of our model, we had the 34 microservices in the TrainTicket project evaluated by three different experts. The maintainability assessments provided by experts for each service are displayed in Table 5, within columns E1, E2, and E3. The consensus decisions of the experts are listed in the 'Decision' column, while the results calculated by our model are displayed in the 'Maintainability by Model' column.

The primary objective of our project is the early detection of microservices with low maintainability values, enabling us to take preventive measures. Out of the 34 microservices we examined, experts identified six as having low maintainability. In our analysis using our model, we accurately determined the maintainability status of five out of these six services labeled as low. Furthermore, we correctly identified 26 out of the 29 services not marked as having low maintainability, achieving a high level of accuracy. The performance measurements of our model are comprehensively detailed in Table 2 and Table 3.

Table 2: Confusion Matrix

		Evaluators	
		LOW	Not LOW
Predicted (Model)	LOW	TP = 5	FP = 2
	Not LOW	FN = 1	TN = 26
Total		6	28

To calculate the success of our model, we used the following metrics:

- **True Positive (TP)**: The count of microservices correctly predicted as low maintainability by our model that were also labeled as LOW by the evaluators.
- **False Positive (FP)**: The number of microservices not labeled as LOW by the evaluators but incorrectly predicted as low by our model.
- **False Negative (FN)**: The count of microservices labeled as LOW by the evaluators but incorrectly predicted by our model.
- **True Negative (TN)**: The number of microservices not labeled as LOW by the evaluators and correctly predicted by our model.

The methodologies implemented for performance calculation encompass the following key indicators:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{F-Measure} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

These metrics provide a comprehensive evaluation of our model's effectiveness in accurately identifying microservices with low maintainability.Additionally, our model has been successfully validated in identifying microservices categorized as Medium and High maintainability. This demonstrates the robustness of our approach in not only pinpointing services with potential maintainability issues but also in affirming the adequacy of those with moderate to high levels of maintainability, underlining the model's comprehensive applicability across a range of maintainability scenarios.

Table 3: Performance Of Model

Recall	**Precision**	**F-Measure**	**Accuracy**
83.33%	71.43%	76.92%	91.18%

Table 4: Statistical Values For The Metrics

ServiceID	AC	CogC	SDC	LOC	CSD	ABND	LCOM	TESTABILITY	MODIFIABILITY	MAINTAINABILITY
M1	1	0	5	850	1.659	0.00	2.80	H	H	H
M2	2.64	13	2	400	1.400	0.82	2.00	H	H	H
M3	3.49	11	2	329	912	1.08	2.00	H	H	H
M4	4.05	31	5	507	1.183	0.98	2.00	H	M	M
M5	2.52	8	1	374	909	0.76	3.40	H	H	H
M6	2.36	27	0	771	376	1.03	3.00	H	H	H
M7	2.84	15	1	886	214	0.79	2.41	H	H	H
M8	15.96	71	4	692	1.214	2.62	2.00	L	L	L
M9	6.5	67	5	594	808	2.62	2.29	L	L	L
M10	1.74	11	0	451	288	0.63	2.17	H	H	H
M11	1.63	7	0	406	296	0.63	2.67	H	H	H
M12	1.94	11	1	623	385	0.51	4.33	H	H	H
M13	1.98	21	0	614	407	0.66	2.43	H	H	H
M14	5.67	36	2	382	942	2.00	2.00	H	M	M
M15	7.32	62	3	701	514	2.15	3.00	L	L	L
M16	4.03	63	3	1204	449	1.45	2.69	L	L	L
M17	1.2	6	0	727	220	0.67	3.33	H	H	H
M18	4.64	99	1	1147	549	1.35	4.25	M	L	M
M19	4.35	100	1	1175	553	1.23	4.25	M	L	M
M20	13.1	37	11	647	1.236	2.05	1.67	L	L	L
M21	13.1	37	11	641	1264	2.05	1.67	L	L	L
M22	2.25	22	0	546	311	0.80	2.50	H	H	H
M23	7.82	76	8	648	1142	1.76	3.00	L	L	L
M24	6.95	37	3	559	662	2.43	2.00	M	M	M
M25	2.54	24	0	632	269	0.72	3.00	H	M	M
M26	6.18	26	3	413	1017	2.46	2.00	M	M	M
M27	1.8	11	2	462	498	0.56	2.33	H	H	H
M28	1.98	22	0	548	365	0.60	2.17	H	H	H
M29	1.84	29	0	500	520	0.66	2.50	H	H	H
M30	2.5	59	4	1065	770	1.07	3.67	M	L	M
M31	2.07	12	5	630	746	1.02	4.14	H	M	M
M32	2.75	74	4	1190	689	0.91	3.20	M	L	M
M33	1.96	17	1	620	597	0.77	3.78	H	H	H
M34	4.36	20	0	376	186	1.37	1.80	H	H	H

5 CONCLUSION

This study introduces a distinct quality assessment framework designed to evaluate the maintainability of microservices. Central to this framework is a hierarchical structure, where maintainability is determined by two essential factors: testability and modifiability. The proposed approach incorporates advanced clustering algorithms to meticulously analyze code metrics and features, leading to an effective method for calculating a microservice's maintainability score based on these factors.

For the verification of the proposed model, it was implemented in a renowned open-source project within the Microservice Architecture (MSA) realm. This involved a detailed analysis of the microservices' source code and a comparative evaluation of the model's predictions against empirical data. The results demonstrated the model's effectiveness in providing developers with practical insights into the maintainability attributes of microservices, highlighting its real-world applicability.

Future research endeavors are directed towards enhancing the model's robustness by integrating real-time performance data with the existing code-based metrics. This comprehensive validation will be carried out in a broader and more complex MSA ecosystem, encompassing an extensive array of microservices. This strategic approach is expected to refine the model's accuracy and utility in maintenance predictability, significantly enriching the field of microservice maintenance evaluation techniques.

REFERENCES

[1] 2019. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Quality measurement framework. ISO/IEC 25020:2019.

[2] J. Bogner, J. Fritzsch, S. Wagner, and A. Zimmermann. 2021. Industry practices and challenges for the evolvability assurance of microservices. *Empirical Software Engineering* 26, 5 (2021). https://doi.org/10.1007/s10664-021-09999-9

[3] J. Bogner, S. Schlinger, S. Wagner, and A. Zimmermann. 2019. A Modular Approach to Calculate Service-Based Maintainability Metrics from Runtime Data of Microservices. In *Product-Focused Software Process Improvement*. 489–496. https://doi.org/10.1007/978-3-030-35333-9_34

Table 5: Comparative Evaluation of Human Expertise and Proposed Model Outcomes

MS. Num.	E1	E2	E3	Decision	Maintainability by Model	MS. Num.	E1	E2	E3	Decision	Maintainability by Model
M1	M	H	M	M	H	M18	M	M	M	M	M
M2	H	H	H	H	H	M19	M	L	M	M	M
M3	M	H	H	H	H	M20	L	L	M	L	L
M4	H	H	H	H	M	M21	L	L	M	L	L
M5	H	H	H	H	H	M22	M	H	H	H	H
M6	M	M	M	M	H	M23	L	L	L	L	L
M7	M	H	M	M	H	M24	H	M	H	H	M
M8	M	L	M	M	L	M25	H	H	M	H	H
M9	M	L	L	L	L	M26	M	L	L	L	M
M10	H	H	H	H	H	M27	H	H	H	H	H
M11	M	H	H	H	H	M28	H	H	H	H	H
M12	M	H	H	H	H	M29	H	H	M	H	H
M13	H	H	M	H	H	M30	M	M	M	M	M
M14	M	M	L	M	M	M31	H	M	H	H	H
M15	L	L	M	L	L	M32	M	L	M	M	M
M16	M	M	M	M	L	M33	M	H	M	M	H
M17	H	H	M	H	H	M34	L	M	M	M	H

[4] J. Bogner, S. Wagner, and A. Zimmermann. 2017. Automatically Measuring the Maintainability of Service- and Microservice-Based Systems. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process An.* https://doi.org/10.1145/3143434.3143443

[5] J. Bogner, S. Wagner, and A. Zimmermann. 2017. Automatically measuring the maintainability of service- and microservice-based systems. https://doi.org/10.1145/3143434.3143443

[6] J. Bogner, S. Wagner, and A. Zimmermann. 2017. Towards a Practical Maintainability Quality Model for Service- and Microservice-Based Systems. https://doi.org/10.1145/3129790.3129816

[7] J. Bogner, S. Wagner, and A. Zimmermann. 2019. Using architectural modifiability tactics to examine evolution qualities of service- and microservice-based systems. *Sics Software-Intensive Cyber-Physical Systems* 34, 2-3 (2019), 141–149. https://doi.org/10.1007/s00450-019-00402-z

[8] N. Dragoni et al. 2017. *Microservices: Yesterday, Today, and Tomorrow.* Springer, Cham. https://doi.org/10.1007/978-3-319-67425-4-12

[9] J. Fritzsch, J. Bogner, S. Wagner, and A. Zimmermann. 2019. Microservices migration in industry: intentions, strategies, and challenges. (2019). https://doi.org/icsme.2019.00081

[10] N. Goel and M. Gupta. 2012. Testability estimation of framework based applications. *Journal of Software Engineering and Applications* 05, 11 (2012), 841–849. https://doi.org/10.4236/jsea.2012.511097

[11] S. Hassan, R. Bahsoon, and R. Kazman. 2020. Microservice transition and its granularity problem: a systematic mapping study. *Software Prac. Experience* 50, 9 (2020), 1651–1681. https://doi.org/10.1002/spe.2869

[12] R. Kasahara, K. Sakamoto, H. Washizaki, and Y. Fukazawa. 2019. Applying gamification to motivate students to write high-quality code in programming assignments. (2019). https://doi.org/10.1145/3304221.3319792

[13] G. Lacerda, F. Petrillo, and M. Pimenta. 2020. Dr-tools: a suite of lightweight open-source tools to measure and visualize java source code. (2020). https://doi.org/10.48550/arxiv.2008.03547

[14] F. Rabbi, S. Hossain, and M. Arefin. 2022. Scma: a lightweight tool to analyze swift projects. (2022). https://doi.org/10.18293/seke2022-006

[15] K. Sellami, M. Saied, and A. Ouni. 2022. A hierarchical dbscan method for extracting microservices from monolithic applications. https://doi.org/10.1145/3530019.3530040

[16] D. Taibi, V. Lenarduzzi, and C. Pahl. 2019. Microservices anti-patterns: a taxonomy. https://doi.org/10.1007/978-3-030-31646-4-5

[17] Wan Yan and Fu Shuai. 2022. Application of microservice architecture in commodity erp financial system. *International Journal of Computer Theory and Engineering* 14, 4 (2022).

[18] R. Yilmaz and F. Buzluca. 2021. A Fuzzy Quality Model to Measure the Maintainability of Microservice Architectures. (2021). https://doi.org/10.1109/IISEC54230.2021.9672417