Bulletin of Electrical Engineering and Informatics
Vol. 14, No. 6, December 2025, pp. 4701~4712
ISSN: 2302-9285, DOI: 10.11591/eei.v14i6.10457

Evaluating maintainability metrics in microservices-based student registration systems

Gintoro[1], Eko Cahyo Nugroho[2]
[1]Department of Computer Science, School of Computer Science, Bina Nusantara University, Jakarta, Indonesia
[2]Department of Computer Science, BINUS Online Learning, Bina Nusantara University, Jakarta, Indonesia

Article Info

Article history:

Received Apr 11, 2025
Revised Sep 13, 2025
Accepted Sep 27, 2025

Keywords:

Change request
ISO/IEC 25010
Maintainability metrics
Microservices
Student registration system

ABSTRACT

As governments redefine educational policy and schools evolve their priorities, more schools must have software that recalibrates with minimal friction. To provide objective guidelines, this study rigorously measures maintainability attributes in a microservices-styled student registration platform, framing the assessment with the ISO/IEC 25010 maintainability specification. We steered each of the standard's maintainability sub-characteristics into defined quantitative constructs, executed in the context of a production microservices topology. Architectural and behavioural views were analysed using Structure101 in static tool runs, and unified modeling language (UML) model inspection anchored the derivation of key metrics, ensuring that stakeholder-defined structures and live microservices concurrency both shaped the evaluation. Results indicate moderate system modularity with average component dependency (ACD) of 2.14, propagation cost (PC) of 10.2%, and identification of one non-trivial cycle group involving three classes. Cohesion analysis revealed structural improvement opportunities in core classes such as admin and candidate lack of cohesion in methods 4 (LCOM4)≥2). The inheritance structure shows optimal characteristics with shallow depth (depth of inheritance tree (DIT)≤1), and controlled breadth (number of children (NOC)=2), supporting both analyzability and modifiability. These findings provide actionable insights for enhancing system maintainability in microservices architectures, particularly for educational domain applications requiring frequent policy adaptations.

Corresponding Author:

Gintoro
Department of Computer Science, School of Computer Science, Bina Nusantara University
KH Syahdan No. 9, Kemanggisan, Palmerah, West Jakarta, Jakarta, Indonesia
Email: gintoro@binus.ac.id

1. INTRODUCTION

Educational institutions worldwide face increasing pressure to digitize and optimize their student admission processes. The student registration process, known as *Penerimaan Peserta Didik Baru* (PPDB) in Indonesia, represents a critical operational component that must adapt to frequent policy changes, technological advances, and varying institutional requirements [1], [2]. Public schools encounter additional complexity due to government-mandated zoning policies and annual regulatory modifications, while private institutions must respond to evolving scholarship programs and fee structures [3]-[5].

Traditional monolithic systems struggle to accommodate the dynamic nature of educational policies and institutional requirements. The rigidity of conventional architectures leads to high maintenance costs, extended development cycles, and reduced system adaptability when implementing policy changes [6]. Educational software systems require architectural patterns that support rapid modification while maintaining

system reliability and performance [7]. Previous research has explored maintainability assessment in various software contexts, yet significant gaps remain for microservices-specific evaluation frameworks in educational domains, as illustrated in Table 1.

Table 1. Comparative analysis across research domains

Study	Domain focus	Architecture	Metrics applied	Key limitations
Dewi *et al.* [8]	University mobile apps	Monolithic	Basic ISO 25010	Reverse engineering emphasis, limited scope
Haoues *et al.* [9]	Health applications	Traditional	Scoring models	Subjective evaluation approach
Bogner *et al.* [7], [10]	Service-oriented systems	SOA/microservices	Custom frameworks	No educational domain validation
Hasan *et al.* [11]	General software	Migration studies	Architecture-specific	Transition focus, limited operational data
Current study	Educational PPDB	Microservices	ISO/IEC 25010+Empirical	Production-scale validation

Recent studies have examined microservices-specific challenges. Hasan *et al.* [11] investigated architecture maintainability transitions from monolithic to microservice systems, while Özdemir and Buzluca [12] proposed classification systems using code metrics and ISO/IEC 250xy standards. However, these studies lack empirical validation in real-world educational systems and comprehensive mapping of ISO/IEC 25010 sub-characteristics to actionable metrics.

The literature reveals three significant gaps: i) limited empirical studies applying ISO/IEC 25010 maintainability metrics specifically to microservices architectures in educational contexts, ii) absence of systematic mapping between ISO/IEC 25010 sub-characteristics and quantitative software metrics for student registration systems, and iii) lack of practical guidelines for interpreting maintainability measurements in the context of policy-driven software evolution.

This study addresses the identified gaps through the following novel contributions:

- Systematic framework development: we establish a comprehensive mapping between ISO/IEC 25010 maintainability sub-characteristics and quantitative software metrics specifically validated for microservices architectures in educational domains.
- Empirical validation: we provide the first large-scale empirical assessment of maintainability metrics in a production microservices-based student registration system serving 200+ educational institutions across Indonesia.
- Practical measurement guidelines: we deliver actionable interpretation criteria for maintainability metrics that enable software architects to make informed decisions about system evolution and refactoring priorities.
- Educational domain insights: we contribute domain-specific findings about maintainability challenges and opportunities in student registration systems, providing a foundation for future research in educational software architecture.

The remainder of this paper demonstrates these contributions through the following structure: section 2 presents our comprehensive method including metric selection, mapping procedures, and measurement techniques. Section 3 details the empirical results with quantitative analysis and interpretation of each maintainability metric. Section 4 provides critical discussion of findings, implications for practice, and comparison with industry benchmarks. Section 5 concludes with recommendations for future research and practical applications.

2. METHOD

This section outlines the systematic approach employed to evaluate maintainability metrics in the microservices-based student registration system, ensuring reproducibility and validity of results.

2.1. Research design

We adopted a quantitative empirical research approach to assess maintainability characteristics in a production microservices-based student registration system. The study follows a systematic measurement framework that maps theoretical maintainability concepts from ISO/IEC 25010 to observable software metrics, enabling objective evaluation of system quality attributes [13].

The research design incorporates three primary phases: i) systematic mapping of ISO/IEC 25010 maintainability sub-characteristics to quantitative metrics, ii) comprehensive data collection through static analysis and architectural examination, and iii) empirical evaluation with industry benchmark comparison.

2.2. System under study

The empirical evaluation focuses on Sokrates apps, a software-as-a-service (SaaS) platform serving 200+ educational institutions across Indonesia since 2013. Several factors influenced this selection:

- Production scale: real-world operational environment with substantial user base.
- Policy adaptation requirements: subject to frequent Indonesian educational policy changes.
- Mature architecture: established microservices implementation with documented evolution.
- Data accessibility: comprehensive architectural documentation and stakeholder cooperation.

While focusing on a single system implementation may limit immediate generalizability to other educational software architectures, the systematic method and comprehensive metric coverage provide a robust foundation for broader application. The system's substantial scale and operational complexity ensure that findings reflect real-world challenges rather than theoretical constructs. The PPDB (student registration) module represents the core functionality, encompassing seven primary functional areas with associated sub-functions as detailed in Table 2.

Table 2. Functional scope of PPDB module

No.	Metric	Description
1.	Batch management	Create, update, delete, and view operations
2.	Admission test management	Assessment creation, scoring, attendance tracking, and scheduling
3.	Candidate management	Personal data, parent information, and sibling records
4.	Registration processing	Application handling and file attachment management
5.	Scholarship administration	Merit-based evaluation and achievement tracking
6.	Payment processing	Fee calculation and transaction management
7.	Student onboarding	NIS generation and enrollment confirmation

2.3. Metric selection and mapping framework

Based on comprehensive literature analysis and ISO/IEC 25010 guidelines, we selected five categories of software metrics that directly correspond to maintainability sub-characteristics [14]. Table 3 presents the systematic mapping between ISO/IEC 25010 sub-characteristics and quantitative measurement instruments [15], [16].

Table 3. ISO/IEC 25010 maintainability sub-characteristics mapping

No.	Sub-characteristic	Metrics	Justification
1.	Modularity	Average component dependency (ACD), propagation cost (PC), and cycle group size	Measures component independence and coupling strength
2.	Reusability	Lack of cohesion in methods (LCOM), depth of inheritance tree (DIT), and number of children (NOC)	Evaluates cohesion and inheritance utilization
3.	Analysability	LCOM, ACD, PC, and DIT	NOC assesses code comprehension complexity
4.	Modifiability	ACD, PC, and cycle group size	Indicates change propagation potential
5.	Testability	Cycle group size, ACD, and PC	Measures test isolation feasibility

2.4. Data collection

For static analysis and data extraction, we employed Structure101 as the primary automated tool for extracting coupling and dependency metrics, supplemented by comprehensive manual analysis to ensure verification and enable detailed examination of architectural patterns [17].

The unified modeling language (UML) class diagram representing the system architecture underwent systematic analysis to identify class relationships, inheritance hierarchies, and dependency patterns. This dual approach combining automated tooling with human expertise ensured both efficiency and accuracy in architectural assessment [14].

Our measurement protocols encompassed four distinct analytical procedures to ensure comprehensive coverage of maintainability characteristics:

- Dependency analysis: complete enumeration of class-to-class dependencies using aggregation and association relationships.
- Cohesion evaluation: method-attribute interaction analysis for lack of cohesion in methods 4 (LCOM4) calculation using graph-based connectivity assessment.
- Inheritance examination: systematic traversal of inheritance hierarchies to determine DIT and NOC values.
- Cycle detection: manual inspection and tool-assisted identification of circular dependencies.

All measurements underwent rigorous quality assurance through independent verification using multiple analysis approaches. Automated tool results were systematically cross validated with manual calculations, and any inconsistencies were resolved through detailed architectural review and expert consultation.

2.5. Formal metric computation methods

According to von Zitzewitz, ACD is defined as the average number of components that any given component depends on, counting both direct and indirect dependencies (including itself) [14]. In thedependency graph, each component i has a "depends upon" value d, which represents the total number of nodes (components) that can be reached from i. The cumulative dependency for the system, called the cumulative component dependency (CCD), is the sum of all these values across n components. Therefore, the formula for ACD is given by (1):

$$\text{ACD} = \frac{\text{CCD}}{\text{n}} = \frac{\sum_{\text{i=1}}^{\text{n}} \text{d}_\text{i}}{\text{n}} \quad (1)$$

PC offers a normalized measure of the potential impact that changes in one component may have on an entire system [14]. The formula is (2):

$$\text{PC} = \frac{\text{CCD}}{n^2} \quad (2)$$

LCOM4 constructs undirected graphs representing method-attribute relationships within each class [14]. The computation involves:

- Create graph G with methods as vertices.
- Add edges between methods sharing attributes.
- Count connected components in G.
- LCOM4 value equals the number of connected components.

Relative cyclicity (R), R quantifies the extent of cyclic dependencies within a system [14]:

$$R = \sqrt{\frac{\sum_{i=1}^{m} k_i^2}{n}} \quad (3)$$

where k represents the number of cycle groups and g_i indicates the size of cycle group i.

3. RESULT AND DISCUSSION

This section presents comprehensive empirical results from the maintainability assessment, providing detailed analysis of each metric category and their implications for system evolution and maintenance.

3.1. System architecture overview

The Sokrates Apps PPDB module implements a microservices architecture with 21 distinct classes representing various functional domains. The architecture exhibits characteristics typical of modern microservices implementations, with clear separation of concerns across functional boundaries. Table 4 provides a comprehensive analysis of class dependencies identified through systematic architectural examination. Figure 1 (in Appendix) illustrates the simplified UML class diagram, demonstrating the relationships and dependencies between system components.

3.2. Comprehensive metric results

3.2.1. Average component dependency analysis

The ACD calculation yielded 2.14, derived from 45 component dependencies distributed across 21 classes within the system architecture. This result indicates that, on average, each component maintains dependencies with approximately 2.14 other components, representing reasonable modularity levels for a microservices architecture of comparable complexity [14]. Industry standards for microservices architectures typically recommend ACD values ranging between 1.5 and 3.0 for optimal maintainability characteristics. The observed value of 2.14 falls within acceptable boundaries, though it approaches the upper threshold, suggesting opportunities for further decoupling improvements through architectural refactoring [18].

Detailed analysis reveals the candidate class exhibits the highest dependency count (10), indicating a significant architectural concern that warrants immediate attention. This elevated coupling likely stems from the class serving as a central aggregate for student-related information, which while functionally logical within

domain modeling principles, introduces maintenance challenges during system evolution. The concentration of dependencies in a single class creates potential bottlenecks for independent service deployment and testing isolation [19].

Table 4. Class dependency analysis

No.	Class	Depends on (associated classes)	Number of dependencies
1.	User	–	0
2.	Admin	User and files	2
3.	Candidate	User, final payment, batch, schedule participant, candidate status, files, achievement, scholarship, parent, and sibling	10
4.	Batch	fee relation, batch step, candidate, and schedule	4
5.	Batch Step	Batch	1
6.	Schedule	Batch and schedule participant	2
7.	Schedule participant	Schedule, candidate, and result	3
8.	Schedule criteria	Result, assessment, and criteria	3
9.	Assessment	Schedule criteria, criteria, and score	3
10.	Result	Schedule participant, score, and schedule criteria	3
11.	Score	Assessment and result	2
12.	Criteria	Schedule criteria and assessment	2
13.	Payment	Candidate	1
14.	Final payment	Candidate	1
15.	Fee relation	Batch	1
16.	Candidate status	Candidate	1
17.	Files	Candidate and admin	2
18.	Achievement	Candidate	1
19.	Scholarship	Candidate	1
20.	Parent	Candidate	1
21.	Sibling	Candidate	1
		Total dependencies	45

Notes: a ***dependency*** means that a class uses, references, or aggregates another class.

The dependency distribution analysis shows that 85% of classes maintain dependency counts below 4, indicating generally healthy modular design. However, the presence of one class with exceptionally high coupling (candidate with 10 dependencies) significantly influences the overall ACD calculation, suggesting that targeted refactoring of this specific component could substantially improve system-wide modularity metrics.

3.2.2. Propagation cost assessment

The system architecture comprises 21 classes with a CCD of 45. Applying the standardized formula for PC calculation:

$$\text{PC} = \frac{45}{21^2} = \frac{45}{441} \approx 0.102 \quad (10.2\%) \tag{4}$$

The PC metric of 10.2% reveals that an arbitrary modification to a singular component is predicted to affect an estimated 10.2% of the entire architecture. Within the pertinent domain, this figure is a clear indicator of effective modularity, as the PCs remain below the 15% threshold widely regarded in the relevant literature as a benchmark for well-structured microservices systems [14]. The restrained propagation is consequential, particularly for platforms supporting Indonesian education, in which dynamic policy prescriptions necessitate surgical alterations rather than comprehensive redesigns.

Peer-reviewed investigations focused on microservices architectures consistently associate PCs below the 15% threshold with systems manifesting appropriate modularity attributes. By yielding a value of 10.2%, the architecture in question reaffirms the architectural intention to circumscribe unintended side-effects when policy-driven modifications are undertaken [20]. The observed PC therefore confirms prior theoretical expectations, further assuring that policy revisions, especially those enacted at peak admission intervals, can be operationalised with a minimized risk of disruption to unrelated functional areas.

Additionally, the quantified PC points toward the disciplined observance of bounded context patterns, which confer a clear locus of responsibility upon individual microservices. By truncating change scope to well-defined service perimeters, the architecture permits continuous, iterative compliance with evolving Indonesian educational policy, all of which can be accomplished without the necessity of comprehensive, global subsystems validation, thus preserving system availability in operational environments where peak usage coincides with the enactment of revised instructional regulations [17].

3.2.3. Cycle group size (relative cyclicity)

Systematic manual analysis of the UML class diagram architecture revealed the presence of a single cycle group involving three classes: admin, candidate, and files. This configuration results in one cycle group of size 3 within the overall system of 21 classes. The relative cyclicity calculation as (5) and (6):

$$\text{Cumulative cyclicity}=3^2=9 \tag{5}$$

$$\text{Relative cyclicity } R=\sqrt{\frac{9}{21}}\approx 0.655 \tag{6}$$

Although the design permits only a single cycle group, the detected circular dependency substantially exacerbates long-term maintenance, thus demanding specific architectural mitigation [14]. The interplay among admin, candidate, and files elements signals shared ownership and entwined lifecycles, obliging developers to make simultaneous, carefully synchronised modifications. Such coalescing introduces friction in evolution, since deploying or validating a single microservice outside the cycle becomes error-prone, undermining the affordance of segregated service development pipelines and automated integration tests.

The cycle truncates isolation, inverts dependencies within the same module, and forces stagnant or circular calls among units, therefore hindering granular unit coverage and tearing apart coherent data flow comprehension. Moreover, the most affected users are educational systems, whose ability to change policy on a tight schedule is outranked by the petty rigidity of the cycle. When a policy tightens for admin, the ripple requirement may silently touch candidate, already bound to the auxiliary, preserved-at-write files. The result is a torque propagation that necessitates a borderline inexpressive list of unit and downstream tests across the cycle, causing the policy deployment sink to attics of unfulfilled casual oversight across serialisation times and usage paths.

3.2.4. Actionable refactoring strategy for cycle resolution

To address the identified cycle, we propose implementing a dependency inversion pattern [20]:

- Extract file service interface: create abstraction for file operations.
- Implement repository pattern: separate data access from business logic.
- Apply event-driven communication: replace direct dependencies with messaging.

This refactoring approach would eliminate the cycle while preserving functional requirements.

3.2.5. Lack of cohesion of methods 4 cohesion analysis

The analysis of class cohesion patterns revealed several important findings across different system components [21], [22]:

- Admin class: multiple disconnected method groups operating on disjoint attribute sets, resulting in estimated LCOM4≥3. This indicates poor cohesion and violation of single responsibility principles.
- Candidate class: moderate cohesion with estimated LCOM4 of 2-3, suggesting opportunity for decomposition into focused aggregates.
- Supporting classes: exhibit good cohesion with LCOM4=1, indicating adherence to single responsibility principles.

3.2.6. Domain-driven decomposition strategy

For classes exhibiting poor cohesion, we recommend decomposition using domain-driven design principles. Candidate class decomposition:

- CandidateProfile (Core identification and academic records).
- FamilyContext (Parent and sibling relationships).
- FinancialRecord (Payment and scholarship information).

This decomposition would achieve LCOM4=1 for all resulting components while improving analyzability and testability.

3.2.7. Inheritance metrics (depth of inheritance tree and number of children)

Analysis of the inheritance structure shows a deliberately simple and controlled hierarchy designed to maximize maintainability. The DIT analysis reveals that both admin and candidate classes inherit directly from the User base class, giving the system a maximum DIT value of 1. This shallow inheritance depth minimizes cognitive overhead when tracing inherited behaviors and keeps overall complexity low, which improves system analyzability [14].

The NOC analysis indicates that the user class functions as the sole base class with exactly two direct subclasses: admin and candidate. This limited inheritance breadth (NOC=2) reflects balanced code reuse without creating excessive risks of modification propagation [23].

The inheritance framework exhibits deliberate architectural orchestration that equilibrates code reuse with asymmetric component autonomy,requirements intrinsic to successful microservices execution. Its flattened architecture parallels microservices doctrine, in which autonomy and explicit responsibility frontiers eclipse elongated inheritance levels [24].

3.3. Benchmark comparison and performance assessment

The system demonstrates strong performance in most maintainability dimensions, with particular strengths in inheritance design and PC control. Primary improvement opportunities exist in cohesion enhancement and cycle elimination, as shown in Table 5 [14], [25].

Table 5. Metric performance summary

Metric	Observed value	Educational benchmark*	General benchmark**	Assessment
ACD	2.14	1.8-2.5	1.5-3.0	Acceptable
PC	10.2%	<12%	<15%	Good
Relative cyclicity	0.655	<0.3	<0.5	Moderate concern
LCOM4 (core classes)	2-3	1-2	1 optimal	Needs improvement
DIT (maximum)	1	<2	<3 recommended	Excellent
NOC (maximum)	2	<4	<5 recommended	Excellent

*Educational software benchmarks derived from domain-specific literature [8], [9]
**General microservices benchmarks from industry studies [7], [10], [14]

3.4. Continuous integration and continuous deployment integration framework

We propose integrating maintainability metrics into development workflows through automated monitoring and quality gates that operate within continuous integration pipelines. The pipeline integration strategy incorporates a dedicated quality check stage that validates multiple maintainability thresholds including ACD limits of 2.5, PC maximums of 12%, LCOM4 thresholds of 2, and zero tolerance for cycle groups. When threshold violations occur, the system automatically generates refactoring recommendations based on identified architectural issues and updates the architecture dashboard with current maintainability status. This approach enables development teams to receive immediate feedback on architectural decisions while maintaining historical tracking of system evolution patterns, ensuring that maintainability concerns are addressed proactively rather than reactively during critical development phases [26].

4. DISCUSSION

4.1. Critical analysis of findings

The empirical assessment addresses significant gaps in existing microservices maintainability research within educational software domains. Previous studies focused on general-purpose systems without domain-specific validation. Our systematic approach provides the first comprehensive empirical assessment of ISO/IEC 25010 maintainability metrics in a production-scale educational microservices environment serving 200+ educational institutions.

Educational software systems present unique maintainability challenges stemming from their policy-driven nature, where regulatory changes require rapid system adaptations while maintaining operational stability during critical admission periods. These characteristics distinguish educational systems from general enterprise applications and necessitate specialized architectural considerations.

4.2. Interpretation of results

The measured ACD of 2.14, together with an ambient PC of 10.2%, testifies to a design where functional integration coexists with component autonomy in near-optimal proportions. Such equilibrium meets a pedagogic imperative: modern academic systems are predicated on intricate business processes that mandate seamless orchestration across variegated services, while simultaneous urgency for discrete component evolution de facto precludes monolithic coupling.

The low PC itself is evidence that effective coupling governance has been institutionalized, reinforcing a microservices paradigm that attenuates the reverberations of inadvertent changes. This property, in turn, confers the tactical option of effecting policy revisions that percolate only across discrete domains, eliminating the milder, often hazardous friction of system-wide refactoring. Furthermore, the measured inheritance architecture, characterized by a depth on inheritance tree of one and a breadth not exceeding two descendants, renders a macroscopic taxonomy that dilutes the well-established inheritance overhead in favour of manifest maintainability, itself the paramount operational autonomy in an academic-oriented architecture.

Cohesion deficiencies remain the paramount operational fragility. The class hierarchies of Admin and Candidate, both central to student administration, fall beneath the tight coupling acceptable limit, thereby

across upward of three metrics of cohesion clusters and of ten distinct feature dependencies, in the former and the latter case, respectively. Both central to those metrics, and both contriving to aggregate unrelated responsibilities into singular modules, the admin class not only suffers from unexplained visibility of the fifty other class members but effectively aggregates modules that, within newly gestated processes, emboss both friction and degradation stringent to the hollow services. Equally, candidate, by translating eleven distinct dependencies, retrofits property the intention of implementor grinding.

4.3. Comparison with existing research

Our findings provide empirical validation for theoretical frameworks established by Bogner *et al.* [7], [10] while revealing domain-specific characteristics not addressed in existing literature. The systematic mapping of ISO/IEC 25010 sub-characteristics to quantitative metrics addresses methodological gaps identified in recent literature [11], [12]. Previous studies either applied custom metrics without standard framework integration or utilized ISO standards without comprehensive quantitative validation.

Industry benchmark comparison reveals the evaluated system performs well relative to general microservices standards while presenting domain-specific characteristics. The observed ACD value falls within industry-recommended ranges but approaches the upper threshold, suggesting educational systems may inherently require higher coupling due to complex business process integration. The PC assessment shows superior performance with 10.2% significantly below the 15% threshold for well-designed systems.

4.4. Practical implications

The maintainability metrics framework demonstrates immediate practical value for educational software development teams through integration with modern development workflows. The proposed continuous integration and continuous deployment (CI/CD) integration enables continuous monitoring of architectural quality, allowing teams to identify potential maintainability issues before they become embedded in production systems.

Educational software systems require architectural patterns that support rapid modification without compromising system reliability. Our analysis reveals that traditional microservices principles require adaptation to address domain-specific requirements. The high coupling observed in the candidate class, while problematic from general microservices perspective, reflects educational business process realities where student information serves as central integration point for multiple functional areas.

4.5. Study limitations

The focus on single system implementation may limit immediate applicability to other educational software systems with different architectural patterns. However, the systematic methodology and comprehensive metric coverage establish robust foundations that other researchers can adapt and extend. The static analysis approach does not capture runtime behavioral characteristics that may influence maintainability in operational environments.

The temporal constraints of point-in-time analysis prevent assessment of maintainability evolution patterns over extended periods. Educational software systems undergo continuous evolution in response to policy changes, and longitudinal studies tracking maintainability metrics during this evolution would provide valuable insights into architectural degradation patterns.

4.6. Future research directions

Future research should incorporate dynamic analysis techniques complementing static architectural assessment. Runtime metrics including actual change frequency patterns, service interaction volumes, and deployment coordination complexities would provide comprehensive views of maintainability characteristics in operational environments.

Applying this systematic methodology to microservices architectures in other policy-driven domains such as healthcare, finance, and government services would validate findings generalizability while identifying domain-specific maintainability patterns. The systematic methodology provides foundations for developing integrated tools that continuously monitor maintainability metrics in microservices environments.

5. CONCLUSION

This study provides comprehensive empirical assessment of maintainability metrics in a microservices-based student registration system using ISO/IEC 25010 standards. Through systematic mapping of maintainability sub-characteristics to quantitative software metrics, we establish a practical framework for evaluating architectural quality in educational software systems.

This work articulates a structured method for subjecting microservices architectures to ISO/IEC 25010 maintainability evaluation, cataloguing precise recommendations for metric choice, measurement protocols, and evaluative thresholds. An exhaustive investigation of a live microservices-based system servicing over 200+ educational institutions reveals recurrent maintainability liabilities and trade-offs pertinent to such architectures. The examination recounts, firstly, reinforcing design attributes, deliberate coupling moderation, and restrained inheritance hierarchies, secondly, detectable liabilities of cohesion degeneration and circulatory dependencies, both of which furnish system-maintenance blueprints.

Empirical evidence substantiates that, despite meeting foundational microservices tenets, a salient dividend of modular isolation and coupling discipline, further architectural advancement is feasible. An architectural coupling density of 2.14, corroborated by a 10.2 percent pairing ratio, confirms coherent compartmental isolation, where specified core class cohesion deficits emerge as focused remodelling testimonies with expected backward compatibility. Demonstrated operational metrics yield graduated prioritisation for adaptive engineering interventions. The prescribed maintainability evaluation, therefore, possesses broad transportability; stakeholders may operationalize prescribed verification scripts across diverse microservices deployments, marrying quantitative introspection with continuous integration pipelines and, via such symbiosis, instituting forward-lean architectural stewardship. Lastly, the inquiry settles the thesis that rigorously grounded maintainability examination is no longer incidental, but rather foundational predictive calibration for microservices engagements, giving saliency to sectors that amass adaptation stockpiles determined by evolving statutory precept.

By integrating established theoretical quality frameworks and concrete measurement instruments, the research both deepens the empirical foundation of software architecture knowledge and offers direct, usable guidance for teams engaged in the development of educational software.

ACKNOWLEDGMENTS

The authors would like to express their gratitude to Bina Nusantara University for providing the support and resources necessary to carry out this study. We also thank the reviewers for their insightful comments and suggestions, which have greatly contributed to improving the quality of this paper. Additionally, we acknowledge previous research efforts in the domain of microservices design and maintainability, which have provided a valuable foundation for this study. Lastly, we extend our appreciation to colleagues and collaborators who offered constructive feedback and technical assistance throughout the research process.

FUNDING INFORMATION

The authors state that no funding was involved in the conduct of this research. This study was carried out independently without financial support from any funding agency, institution, or organization.

AUTHOR CONTRIBUTIONS STATEMENT

This journal uses the Contributor Roles Taxonomy (CRediT) to recognize individual author contributions, reduce authorship disputes, and facilitate collaboration.

Name of Author	C	M	So	Va	Fo	I	R	D	O	E	Vi	Su	P	Fu
Gintoro	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓			✓	
Eko Cahyo Nugroho		✓				✓		✓	✓	✓	✓	✓		

C : **C**onceptualization
M : **M**ethodology
So : **So**ftware
Va : **Va**lidation
Fo : **Fo**rmal analysis
I : **I**nvestigation
R : **R**esources
D : **D**ata Curation
O : Writing - **O**riginal Draft
E : Writing - Review & **E**diting
Vi : **Vi**sualization
Su : **Su**pervision
P : **P**roject administration
Fu : **Fu**nding acquisition

CONFLICT OF INTEREST STATEMENT

The authors state no conflict of interest. They declare that there are no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

DATA AVAILABILITY

The dataset used and analyzed in this study is publicly available on Zenodo. It can be accessed at the following link: https://doi.org/10.5281/zenodo.15167058. Researchers and practitioners can freely use the dataset under the terms of the applicable license.

REFERENCES

[1] V. M. Falolo, K. T. Capillas, N. A. Vergarra, and A. F. Cerbito, "Student Registration and Records Management Services Towards Digitization," *International Journal of Education Management and Development Studies*, vol. 3, no. 1, Mar. 2022, doi: 10.53378/352867.

[2] Sumarno, Lisyanto, and N. Basuki, "An Evaluation of the Implementation of the New SMA Student Admissions Zoning System In Medan City Using The Van Meter and Van Horn Policy Implementation Process Model," *IOSR Journal Of Humanities And Social Science (IOSR-JHSS)*, vol. 29, no. 8, pp. 52-60, Aug. 2024, doi: 10.9790/0837-2908065260.

[3] D. E. Lagman, L. H. Grefaldo, and J. R. Sarmiento, "Enhancing Student Enrollment Processes Through Online Systems," *Global Scientific Journal (GSJ)*, vol. 12, no. 5, pp. 961-971, 2024.

[4] C. K. Kusumah, "12-Years Compulsory Education Policy and Education Participation Completeness:Evidence from Indonesia: Evidence from Indonesia," *The Journal of Indonesia Sustainable Development Planning*, vol. 2, no. 2, pp. 187–201, Aug. 2021, doi: 10.46456/jisdep.v2i2.138.

[5] S. Romlah, A. Imron, Maisyaroh, A. Sunandar, and Z. A. Dami, "A free education policy in Indonesia for equitable access and improvement of the quality of learning," *Cogent Education*, vol. 10, no. 2, pp. 1-27, Dec. 2023, doi: 10.1080/2331186X.2023.2245734.

[6] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to Microservices: An assessment framework," *Information and Software Technology*, vol. 137, pp. 1-12, Sept. 2021, doi: 10.1016/j.infsof.2021.106600.

[7] J. Bogner, S. Wagner, and A. Zimmermann, "Towards a practical maintainability quality model for service-and microservice-based systems," in *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings*, Canterbury United Kingdom: ACM, Sept. 2017, pp. 195–198, doi: 10.1145/3129790.3129816.

[8] M. R. Dewi, N. Ngaliah, and S. Rochimah, "Maintainability Measurement and Evaluation of myITS Mobile Application Using ISO 25010 Quality Standard," in *2020 International Seminar on Application for Technology of Information and Communication (iSemantic)*, Semarang, Indonesia, Sept. 2020, pp. 530–536, doi: 10.1109/iSemantic50169.2020.9234283.

[9] M. Haoues, R. Mokni, and A. Sellami, "Machine learning for mHealth apps quality evaluation: An approach based on user feedback analysis," *Software Quality Journal* vol. 31, no. 4, pp. 1179–1209, May 2023, doi: 10.1007/s11219-023-09630-8.

[10] J. Bogner, S. Wagner, and A. Zimmermann, "Automatically measuring the maintainability of service- and microservice-based systems: a literature review," in *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, Gothenburg Sweden: ACM, Oct. 2017, pp. 107–115, doi: 10.1145/3143434.3143443.

[11] M. H. Hasan, M. H. Osman, N. I. Admodisastro, and M. S. Muhammad, "From Monolith to Microservice: Measuring Architecture Maintainability," *International Journal of Advanced Computer Science and Applications*, vol. 14, no. 5, 2023, doi: 10.14569/IJACSA.2023.0140591.

[12] O. Özdemir and F. Buzluca, "Evaluating Microservices Maintainability: A Classification System Using Code Metrics and ISO/IEC 250xy Standards," in *Proceedings of the 2024 13th International Conference on Software and Computer Applications*, Bali Island Indonesia: ACM, Feb. 2024, pp. 55–61, doi: 10.1145/3651781.3651790.

[13] ISO/IEC, "Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models, ISO/IEC 25010:2011(E)," *International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC)*, 2011.

[14] A. von Zitzewitz, "Using Software Metrics to Ensure Maintainability," in *Software Architecture Metrics*, O'Reilly Media, Inc., 2022, p. 9.

[15] J. Estdale and E. Georgiadou, "Applying the ISO/IEC 25010 Quality Models to Software Product," in *Systems, Software and Services Process Improvement, Eds., in Communications in Computer and Information Science*, vol. 896, pp. 492–503, 2018, doi: 10.1007/978-3-319-97925-0_42.

[16] F. H. Vera-Rivera, "Cognitive Complexity Points: A Metric to Evaluate the Design of Microservices-Based Applications," *Ingeniería y Competitividad*, vol. 26, no. 1, Mar. 2024, doi: 10.25100/iyc.v26i1.13145.

[17] F. M. Muthengi, D. M. Mugo, S. M. Mutua, and F. M. Musyoka, "A Simplified Approach to Establishing the Impact of Software Source Code Changes on Requirements Specifications," *Bulletin of Electrical Engineering and Informatics*, vol. 14, no. 1, pp. 543–550, Feb. 2025, doi: 10.11591/eei.v14i1.8736.

[18] A. M. Saleh and O. Enaizan, "Framework for Selecting the Best Software Quality Model for a Smart Health Application Based on Intelligent Approach," *Bulletin of Electrical Engineering and Informatics*, vol. 12, no. 3, pp. 1711–1727, Jun. 2023, doi: 10.11591/eei.v12i3.4945.

[19] C. Ciceri *et al.*, *Software Architecture Metrics*, 1st ed. O'Reilly Media, Inc., 2022.

[20] D. R. F. Apolinário and B. B. N. De França, "A Method for Monitoring the Coupling Evolution of Microservice-Based Architectures," *Journal of the Brazilian Computer Society*, vol. 27, no. 1, p. 17, Dec. 2021, doi: 10.1186/s13173-021-00120-y. d

[21] E. N. H. Kırğıl and T. E. Ayyıldız, "Predicting Software Cohesion Metrics with Machine Learning Techniques," *Applied Sciences*, vol. 13, no. 6, p. 3722, Mar. 2023, doi: 10.3390/app13063722.

[22] V. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," *IEEE Access*, vol. 11, pp. 88339–88358, 2023, doi: 10.1109/ACCESS.2023.3305687.

[23] R. Yılmaz and F. Buzluca, "A fuzzy logic-based quality model for identifying microservices with low maintainability," *Journal of Systems and Software*, vol. 216, p. 112143, Oct. 2024, doi: 10.1016/j.jss.2024.112143.

[24] P. Zaragoza, A.-D. Seriai, A. Seriai, H.-L. Bouziane, A. Shatnawi, and M. Derras, "Refactoring Monolithic Object-Oriented Source Code to Materialize Microservice-oriented Architecture:," in *Proceedings of the 16th International Conference on Software Technologies*, Online Streaming, --- Select a Country ---: SCITEPRESS - Science and Technology Publications, 2021, pp. 78–89, doi: 10.5220/0010557800780089.

[25] E. N. H. Kirgil and T. E. Ayyildiz, "Analysis of Lack of Cohesion in Methods (LCOM): A Case Study," in *2021 2nd International Informatics and Software Engineering Conference (IISEC)*, Ankara, Turkey, Dec. 2021, pp. 1–4, doi: 10.1109/IISEC54230.2021.9672419.
[26] Y. Jani, "Implementing Continuous Integration and Continuous Deployment (CI/CD) in Modern Software Development," *International Journal of Science and Research*, vol. 12, no. 6, pp. 2984–2987, Jun. 2023, doi: 10.21275/SR24716120535.

APPENDIX

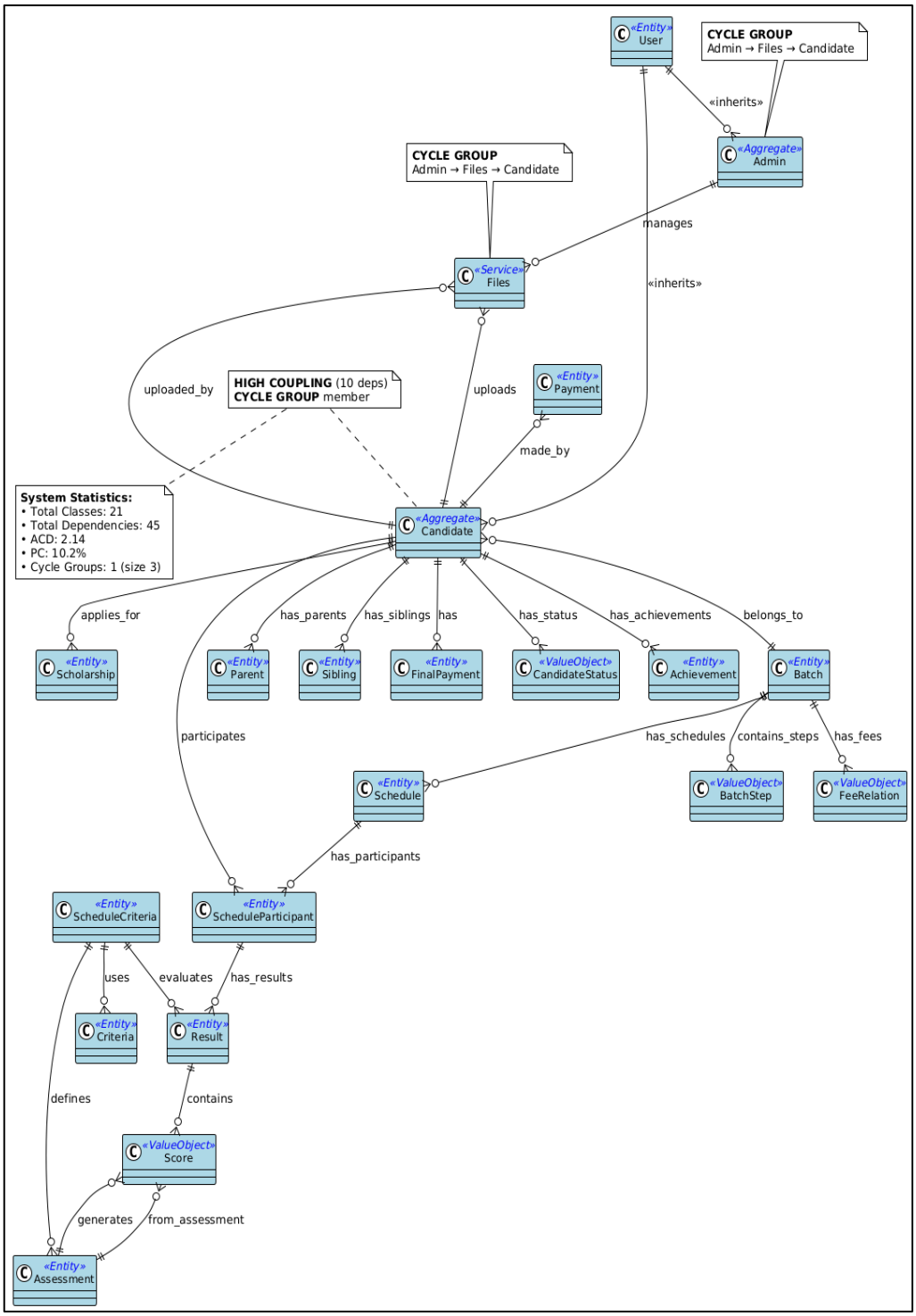

Figure 1. Simplified UML class diagram of PPDB microservices architecture (properties and methods omitted for clarity; focus on structural relationships for maintainability metric calculations, cycle group highlighted: admin→files→candidate)

BIOGRAPHIES OF AUTHORS

Gintoro is a lecturer from Binus University, Indonesia's School of Computer Science. He received his Computer Science degree from Bina Nusantara University in 1998. He also received a Master of Information System degree from Bina Nusantara University, Jakarta, in 2001. He currently serves as Educational Services Director at BINUS University, leading two sub-business units: Sokrates Empowering School and BINUS Center. His research interests include software engineering, software architecture, artificial intelligence, educational technology, and implementing technology in teaching and learning. He can be contacted at email: gintoro@binus.ac.id.

Eko Cahyo Nugroho is a Lecturer in the Department of Computer Science at Bina Nusantara University, Jakarta, Indonesia. With extensive expertise in server management, DevOps, and web and mobile programming using microservices architecture and AWS Cloud serverless solutions, his research focuses on developing scalable and cost-effective IT systems. He is dedicated to advancing software engineering practices and empowering future technology professionals through his innovative research and dynamic teaching approach. He can be contacted at email: eko.nugroho003@binus.ac.id.