Measuring Modularity in JavaScript-Based Microservices from Software Architectural Perspectives

Claudia Cahya Primadani[1,*], Seonah Lee[1,2,*]

[1]Department of AI Convergence Engineering, Gyeongsang National University, 501 Jinjudaero, Jinju-si, Gyeongsangnam-do, CO 52828 Republic of Korea

[2]Department of Aerospace and Software Engineering and Department of AI Convergence Engineering, Gyeongsang National University, 501 Jinjudaero, Jinju-si, Gyeongsangnam-do, CO 52828 Republic of Korea

Abstract

As businesses embrace concepts for rapid, scalable, and maintainable software systems, microservices adoption is gaining traction. Meanwhile, the current work on software architecture metrics focuses mainly on measuring modularity metrics in object-oriented programming, particularly in the Java environment. To address this gap, we propose a method for measuring fundamental quality attributes coupling and cohesion to evaluate architectural modularity specifically in the context of JavaScript-based microservices. In addition, we conduct a case study with coupling and cohesion measurement methods for evaluating architectural modularity in JavaScript-based microservices. We finally discuss future directions of refining these metrics in the dynamic context of microservices design.

Keywords

software architecture, microservices architecture, software architecture metric

1. Introduction

The introduction of microservices architecture into the industrial technological landscape of software development has altered the way applications are developed and deployed. Microservices provide outstanding flexibility and scalability, allowing businesses to construct sophisticated systems built of loosely connected, independently deployable services [1]. Within this paradigm, JavaScript has become a common language for both frontend web development and backend development, and its role has expanded to include microservices implementation. As the implementation of microservices in JavaScript environments grows, it becomes increasingly important to evaluate the architectural quality of these systems.

However, previous research on software architecture metrics primarily focused on object-oriented programming (OOP) languages such as Java. For example, Panichella et al. (2021) investigated structural coupling in 17 Java-based open-source projects that used docker in their implementation [2]. Milić et al. (2022) assessed the quality of software architecture in Java-based Jakarta EE monolithic and microservice software architectures for architecture optimization

ISE 2023: 2nd International Workshop on Intelligent Software Engineering, December 4, 2023, Seoul

*Corresponding author.

cc.primadani@gmail.com (C. C. Primadani); saleese@gnu.ac.kr (S. Lee)

[3]. Zhong et al. (2023) measured coupling in 15 Java-based open-source projects with a little business function that used Spring Cloud and used the most recent version of each project [4]. Abadeh et al. (2023) introduced modularity vulnerability to assess the impact of module failure on the modularity factor on 8 C# project designed as an object-oriented programming language [5].

To narrow down the gap between the practices (e.g., implementation by Javascript) and the metrics (e.g., metrics for Java), we propose a method for measuring coupling and cohesion to evaluate architectural modularity, specifically in the context of JavaScript-based microservices. Modularity is regarded as a property that influences software robustness [5]. Coupling and cohesion indicate the modularity or independence of services or modules, as well as the maintenance effort [2, 4, 6]. Measuring those factors will assist developers in determining whether changes in specific modules will have an impact on other modules [1]. For cohesion, we adopt the Chidamber and Kemerer Lack of Cohesion in Methods (LCOM). For coupling, we measure abstractness and instability values. We then visualize the balance between those values to understand whether the system is well-designed. We apply our metrics to an exampled micro-service system, a so-called ticketing system, as a case study.

The rest of this paper is organized as follows: Section 2 introduces the background of architectural styles and technology used. Section 3 describes the implementation of architecture and the metric used to evaluate it. Section 4 reports the results. Finally, Section 5 concludes the findings and discusses future work.

2. Background

In this section, we brief the architectural styles and technologies employed in this research.

2.1. Microservices

Microservices are an architectural concept that divides complicated software systems into smaller, self-contained parts that can be deployed independently without impacting the rest of the system, allowing for more flexibility and agility. Furthermore, because of the small size of the units and the independence of the services, they can be deployed frequently and quickly to gather input and make adjustments to create a better system [7, 8, 2]

The main characteristics of the Microservices Architecture (MSA) are:

1. **Decomposition:** A complicated system can be divided into more manageable components, which facilitates development, maintenance, and scalability build based on business capabilities [7, 8, 9].
2. **Independence:** Each microservice is self-contained, having its own code, data storage, and, in certain cases, a database. The services may be replaced, upgraded, and scaled without interfering with other services. This autonomy enables independent development, deployment, and scalability [7, 8, 9].

These characteristics make microservices architecture appropriate for software systems with complexity, dynamic requirements, and rapid scalability.

Microservices provide advantages but often create obstacles. As the system expands, it becomes increasingly important to strike a balance between coupling and cohesiveness. Because microservices are supposed to be loosely coupled, the complexities of maintaining low coupling while retaining strong cohesion become increasingly apparent [10].

Microservices architecture also poses a cost management problem owing to higher infrastructure, operational, and development expenses. It is difficult to allocate expenses to services and teams. Limiting expenditures while preserving system performance is a delicate balance [7, 10].

2.1.1. JavaScript

Because of its adaptability and widespread acceptance, JavaScript has become a popular choice for deploying microservices in modern software development. JavaScript is an essential programming language for modern software applications due to its flexibility in implementing object-oriented or functional programming paradigms, a rich ecosystem of libraries, frameworks, and tools, and widespread use in software development, including microservices and serverless applications.

Object-Oriented Programming (OOP) and Functional Programming (FP) are two separate paradigms in JavaScript. OOP is all about constructing objects that contain data and functionality, and it frequently involves inheritance via prototypes. This method provides structure but can result in complicated inheritance hierarchies. In contrast, functional programming stresses pure functions and immutable data, increasing predictability and modularity. Because of the adaptability of JavaScript, developers may combine various paradigms, selecting the optimal technique for their project's goals and coding tastes.

Versatility of JavaScript as an object-oriented or functional programming language makes it perfect for implementing microservices. Developers can tailor their approach to meet the demands of their services, ranging from standard object-oriented structures to data-driven paradigms. This flexibility, however, can lead to compatibility concerns and complicated communication across services, demanding well-defined interfaces and communication protocols in order to preserve architecture integrity and cohesiveness. Therefore, measuring the coupling and cohesion is really important to asses the adaptation of microservices and get insight about maintenance effort, and warn the developer whether changes in service will harm the other services. Furthermore, JavaScript processes incoming requests asynchronously, with a single-thread processing model. For parallelized performance, many processes are required. Microservices frequently employ many Docker containers for scaling up or scaling out, putting additional strain on computers [11, 8]. Furthermore, the developer often orchestrates the containers using kubernetes which enables dynamic load balancing, self-healing, efficient resource allocation, and resilience, facilitates application scalability, and lets developers work with diverse programming languages and technology stacks [12].

3. Experimental Setup

3.1. Microservices Implementation

We employed a microservices architecture in our trial setup, with JavaScript as the major programming language. Docker containers were used to encapsulate each microservice, providing consistent and isolated settings. Kubernetes, a container orchestration technology, played a critical role in managing, growing, and orchestrating these microservices.

3.1.1. Ticketing System

In our studies, we used a ticketing system[1] as a case study to demonstrate the fundamentals of microservices architecture. Figure 1 illustrates the architectural design, illustrating a clear and comprehensive picture of the structure and component interactions of the ticketing system. The system has five distinct services, and each service has its own set of functionality, as shown in Table 1

JavaScript was used to develop the systems. We utilize express.js as the backend framework. The system is built as a Docker container and is orchestrated using Kubernetes.

In this study, we manually analyze the code for each function or class to assess the additional library use as well as the inter- and intra-service dependencies that will be counted as efferent coupling. The sample of intra-service and inter-service dependencies from the ticketing system is shown in Figure 2. As shown in Figure 2, the listener class or function from the services is dependent on the publisher class or function from other services considering inter-service dependency. The listener will be triggered to carry out their function if the publisher has an execution function. The listener class will be impacted by modifications to the publisher's code or design. This coupling should be noticed by the developer. Consequently, even though inter-services are not directly imported from the same class or function, they are nevertheless computed as efferent couplings.

[1]https://github.com/primakashi/ticketing

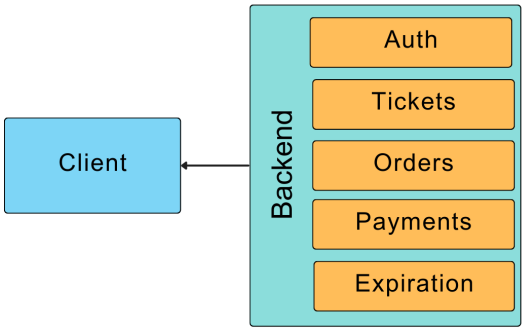

Figure 1: Design Architecture Microservices: Ticketing System

Table 1
List of Services in Ticketing System

No.	Name of Services	Function / Class	Number of Function / Class
1	Auth	Signup, signin, signout, currentUser, password	5
2	Tickets	indexTicket, createTicket, showTicket, updateTicket, OrderCancelledListener, OrderCreatedListener, TicketCreatedPublisher, TicketUpdatedPublisher	8
3	Payments	createCharge, OrderCancelledListener, OrderCreatedListener, PaymentCreatedPublisher	4
4	Orders	indexOrder, newOrder, showOrder, deleteOrder, ExpirationCompleteListener, PaymentCreatedListener, TicketCreatedListener, TicketUpdatedListener, OrderCancelledPublisher, OrderCreatedPublisher	10
5	Expiration	expirationQueue, OrderCreatedListener, ExpirationCompletePublisher	3

3.2. Modularity Metric

We evaluated the microservices architecture using the modularity measure of the software architecture. This metric was a critical indicator of the system design. By evaluating the modularity of our microservices, we received significant insights into the architecture capability to improve maintainability, reusability, and scalability. This metric also ensured that the system

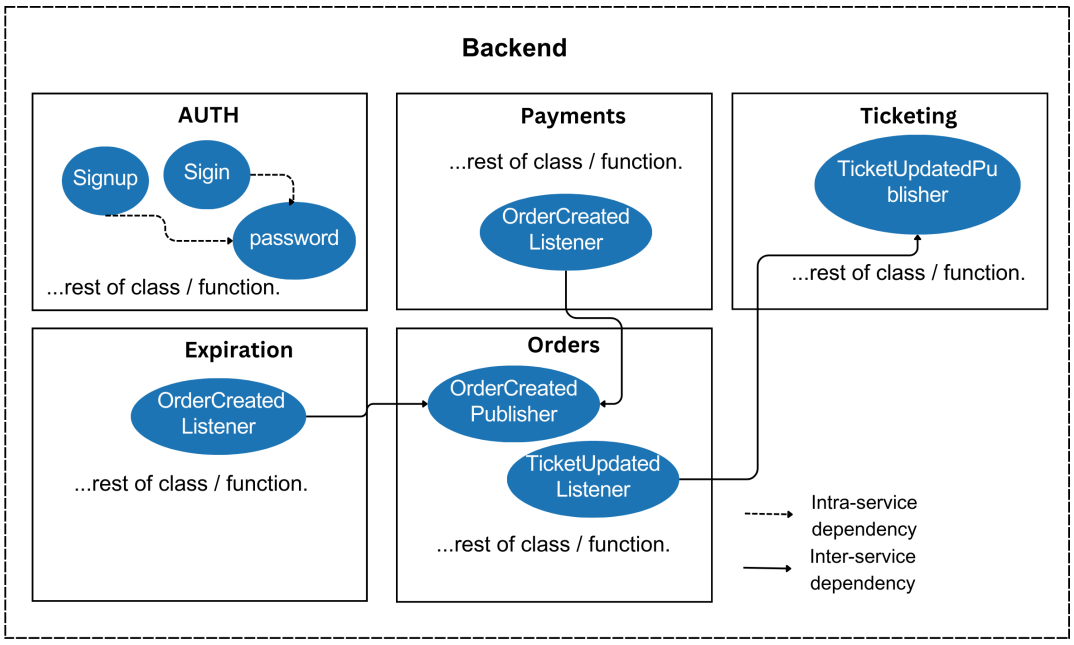

Figure 2: Illustration inter- and intra-service dependencies

components were properly separate and were capable of operating independently [12].

3.2.1. Software Architecture Metrics

Software architecture metrics are critical elements of assessing and measuring the design quality and performance of a software system. These metrics give objective assessments of architectural characteristics like modularity, complexity, and connection. Developers and architects may make knowledgeable judgments, uncover possible difficulties, and optimize the design to fulfill particular goals and needs by examining these data. In essence, software architectural metrics are critical for guaranteeing software system dependability, maintainability, and scalability. Our primary focus in this study is on the modularity measure in the context of software architecture. We carefully studied how effectively the system's components could be divided into self-contained modules and evaluated the degree of independence these modules demonstrated. Modularity describes a logical grouping of similar code, such as classes in an object-oriented language or functions in a structured or functional language. Most programming languages have techniques for modularity [13, 14].

This study will specifically assess modularity by assessing the amount of coupling and cohesion within the software architecture. Measuring coupling allows us to examine the interdependence of distinct modules, whereas measuring cohesion determines the strength of logical linkages inside specific modules.

3.2.2. Cohesion

Cohesion defines the degree to which a module's elements are contained inside it, demonstrating their relative connection. A coherent module should be packed together since splitting it up would necessitate coupling it through calls across modules [14].

We assess cohesion using the LCOM metric. The Chidamber and Kemerer Lack of Cohesion in Methods (LCOM) metric assesses the structural cohesiveness of a module, which is typically a component [14]. The metric determines if a class breaches the single-responsibility principle by constructing a dependency network between all methods and fields. It counts the number of components that are not linked. In an ideal environment, this value would be 1, but if it is greater, the class can be divided into smaller ones [15]. The LCOM is calculated using an equation that is described in equation 1

$$LCOM = \begin{cases} |P| - |Q|, & if |P| > |Q| \\ 0 \text{ otherwise} \end{cases} \tag{1}$$

P is the number of methods that do not access shared variables while Q is the number of shared variables.

3.2.3. Coupling

Coupling in software design refers to the degree of interdependence or connection between distinct components or modules inside a system. It measures how tightly these components rely on one another. Strong coupling indicates considerable dependency, while weak coupling

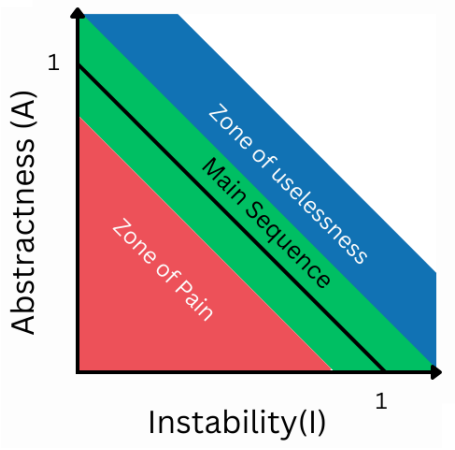

Figure 3: Optimal balance of distance metric

shows greater independence. Coupling can be measured by calculating abstractness, instability, and distance from the main sequence.

Abstractness is defined as the ratio of abstract artifacts (classes, interfaces) to concrete objects (implementation). It compares abstraction to implementation. No abstractions resulting in a single function, while too many abstractions make it difficult for developers to comprehend how things work together [14]. Abstractness is calculated by equation 2. In equation 2, m^a is the number of abstract element while m^c is the number of concrete objects.

$$Abstractness = \frac{\sum m^a}{\sum m^c} \tag{2}$$

The instability metric assesses the volatility of a code base. High instability suggests a higher likelihood of breaking pieces of code as a result of intense coupling, such as when a functionality is distributed to numerous classes [8]. Instability measured by afferent coupling and efferent coupling. Efferent coupling is the number of outside modules used by the current module, or it is simply the number of outside modules or libraries imported by the current module. Meanwhile, afferent coupling is the number of outside modules that use current module [10, 14]. Instability scores are measured using equation 3.

$$Instability = \frac{EfferentCoupling}{EfferentCoupling + AfferentCoupling} \tag{3}$$

The distance metric expects an ideal balance of abstractness and instability, allowing developers to measure the distance from the main sequence metric for classes that are close to this idealized line illustrated in Figure 3

In an ideal scenario, a well-structured module or service should indeed exist in the ”green area” , which represents a state of optimal balance of abstractness and instability [14]. This zone signifies that the module maintains just the right level of interconnection with other components, neither overly entangled nor excessively isolated.

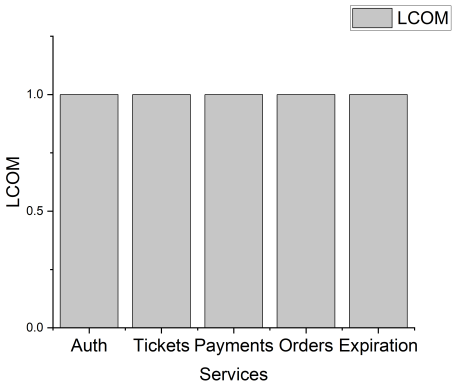

Figure 4: Result of LCOM evaluation

4. Experiment Result

4.1. Cohesion

As for cohesion metric, Figure 4 illustrates the result of cohesion evaluation by LCOM. Figure 4 shows a remarkable pattern in which all services earn an LCOM score of 1. This assessment demonstrates that the services portrayed have a consistent and unified design. Services with an LCOM score of 1 are frequently easier to understand, manage, and alter since they have a concentrated and self-contained structure.

The framework utilized in this ticketing system emphasizes the Single Responsibility Principle (SRP) while creating routing APIs to guarantee that each route handler function inside the system is focused on a single, well-defined responsibility. This not only improves cohesiveness but also simplifies maintenance and encourages reusability.

4.2. Coupling

We discovered that the efferent coupling exceeded the afferent coupling in our coupling measure evaluation illustrated in Figure 5 (a), suggesting that there were more dependencies outside the system components than within. The integration of external libraries into the codebase, which naturally produced stronger efferent coupling by generating dependency on other resources, was the fundamental cause of this difference. Despite the external dependencies, the overall system instability remained substantial, emphasizing the importance of careful control of these external dependencies.

However, in our coupling metric assessment, we received a 0 score in abstractness, according to Express.js default design philosophy of simplifying web routing. This score indicates that Express.js reduces the need for complicated abstraction layers, as its fundamental design handles HTTP routing complexity efficiently. Rather than adding levels of abstraction, Express.js encourages developers to create routes and associated handlers directly.

The overall performance of the ticketing system in coupling is remarkable, with a balanced ratio between abstractness and instability shown in Figure 5 (a). Among all services, Orders

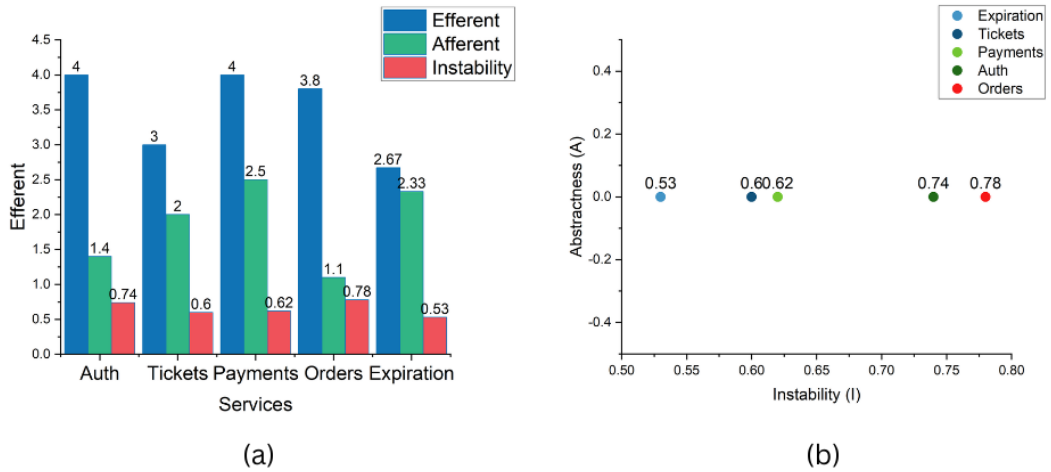

Figure 5: Result of coupling evaluation. (a) result of instability defines by efferent and afferent coupling, (b) result of distance metric

and Auth services are visually closest to the green area.

5. Conclusion

Our studies employed the ticketing system as a real-world case study to evaluate the modularity metric inside a JavaScript-based microservices architecture. We examined the subtle balance of coupling and cohesion, evaluating the interdependencies among system components while ensuring that each module remained self-contained and focused on specific tasks. This study reinforces the relevance of modularity and coherent design in microservices, demonstrating their potential to improve maintainability and flexibility in software systems.

In future studies, we plan to deepen our understanding of software architecture metrics and investigate other assessment methodologies to improve decision-making abilities for maintaining software architecture. We want to broaden our knowledge base by looking at other metrics that might give more information about software architecture. Furthermore, we intend to conduct research that will lead to the creation of practical and user-friendly tools for assessing these metrics.

References

[1] D. R. Apolinário, B. B. de França, A method for monitoring the coupling evolution of microservice-based architectures, Journal of the Brazilian Computer Society 27 (2021) 17.

[2] S. Panichella, M. I. Rahman, D. Taibi, Structural coupling for microservices, arXiv preprint arXiv:2103.04674 (2021).

[3] M. Milić, D. Makajić-Nikolić, Development of a quality-based model for software architecture optimization: A case study of monolith and microservice architectures, Symmetry 14 (2022). URL: https://www.mdpi.com/2073-8994/14/9/1824. doi:10.3390/sym14091824.

[4] C. Zhong, H. Zhang, C. Li, H. Huang, D. Feitosa, On measuring coupling between microservices, Journal of Systems and Software 200 (2023) 111670.
[5] M. N. Abadeh, M. Mirzaie, An empirical analysis for software robustness vulnerability in terms of modularity quality, Systems Engineering (2023).
[6] S. Silva, A. Tuyishime, T. Santilli, P. Pelliccione, L. Iovino, Quality metrics in software architecture, in: 2023 IEEE 20th International Conference on Software Architecture (ICSA), IEEE, 2023, pp. 58–69.
[7] D. Shadija, M. Rezai, R. Hill, Towards an understanding of microservices, in: 2017 23rd International Conference on Automation and Computing (ICAC), IEEE, 2017, pp. 1–6.
[8] K. Bakshi, Microservices-based software architecture and approaches, in: 2017 IEEE aerospace conference, IEEE, 2017, pp. 1–8.
[9] M. Waseem, P. Liang, M. Shahin, A. Di Salle, G. Márquez, Design, monitoring, and testing of microservices systems: The practitioners' perspective, Journal of Systems and Software 182 (2021) 111061.
[10] H. Vural, M. Koyuncu, Does domain-driven design lead to finding the optimal modularity of a microservice?, IEEE Access 9 (2021) 32721–32733.
[11] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: 2016 IEEE international symposium on workload characterization (IISWC), IEEE, 2016, pp. 1–10.
[12] K. Sethi, Y. Cai, S. Wong, A. Garcia, C. Sant'Anna, From retrospect to prospect: Assessing modularity and stability from software architecture, in: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, IEEE, 2009, pp. 269–272.
[13] N. Ford, R. Parsons, P. Kua, P. Sadalage, Building evolutionary architectures, " O'Reilly Media, Inc.", 2022.
[14] M. Richards, N. Ford, Fundamentals of software architecture: an engineering approach, O'Reilly Media, 2020.
[15] C. Ciceri, D. Farley, N. Ford, A. Harmel-Law, M. Keeling, C. Lilienthal, J. Rosa, A. von Zitsewitz, R. Weiss, E. Woods, Software Architecture Metrics: Case Studies to Improve the Quality of Your Architecture, O'Reilly, 2022.