JURNAL EMACS
[Engineering, MAthematics and Computer Science] Vol.7 No.2 May 2025: 167-173
e-ISSN: 2686-2573
DOI: 10.21512/emacsjournal.v7i2.13003

‘DREAMS D’: New Matrix Evaluation for Software Architecture

Zulfany Erlisa Rasjid[1*], Ivana Yoshe Aldora[2], Welly Piyono[3], Risma Yulistiani[4], Hady Pranoto[5]

[1-5] Computer Science Program, Computer Science Department, School of Computer Science,
Bina Nusantara University,
Jakarta, Indonesia 11480
zulfany@binus.ac.id, ivana.aldora@binus.ac.id, welly.piyono@binus.ac.id,
risma.yulistiani@binus.ac.id, hadypranoto@binus.ac.id

*Correspondence: zulfany@binus.ac.id

Abstract – *The microservices software architecture is highly popular and commonly used in developing large-scale systems. Does this mean that microservices are superior, or could older architectures like monolithic be more adaptable to modern developments? The selection of software architecture is crucial to support overall system performance, quality, and user experience. Effective evaluation also plays a significant role in assessing system performance. In this paper, an evaluation matrix model is proposed, called 'DREAMS D,' comprising of seven vital components to test the quality of systems built using specific architectures. The focus is on microservices and monolithic architecture as our sample Software Architectures. The evaluation is conducted through a systematic review, and each architecture is scored based on factors such as Development, Response time, Error handling, Availability, Maintenance, Scalability, and Deployment. The result shows that microservices architecture scores higher in most evaluation criteria, suggesting better suitability for complex and adaptive systems. However, monolithic architecture may still be appropriate for simpler systems due to its lower cost and straightforward integration. This study provides a structured and measurable framework for assisting developers and organizations in making strategic decisions when choosing or transitioning between software architectures. The DREAMS D matrix can be used as a reference model for future evaluations or as a foundation for extending the framework to other architectural paradigms such as serverless or event-driven systems.*

Keywords: *Microservices; Monolithic; Software Architecture; Deployment; Evaluation*

I. INTRODUCTION

Software Architecture (SA) plays a fundamental role in the development of systems (Lim et al., 2021). SA can be defined as the relationship among components, functionalities, and design principles within a software system (Sahlabadi et al., 2022), (Yang et al., 2021), (Venters et al., 2018), (Hasselbring, 2018). The appropriate selection of software architecture can enhance system credibility (Yang et al., 2002), thereby influencing user experience (Bao et al., 2011) and creating software that is high-quality, robust, and adaptable. One method to assess software quality is through evaluation (Yan et al., 2020). Due to that reasons, an evaluation matrix is proposed to measure the quality of software architecture using several key factors: development cost, development effort, response time, error fault, availability, maintenance, scalability, and deployment, in short the ‘DREAMS D’ matrix.

The methodology employed is a systematic review, presented as a comparison table of evaluation factors between monolithic and microservice architectures, both of which are prevalent in software development across various industry scales.

The objective of this paper is to assist developers in selecting an appropriate SA during the software design process before entering the deployment stage or determine the importance of switching to a different system architecture (SA) in the development of an existing application.

1.1 Literature Review

Evaluation in the context of software involves systematic assessment of the quality, performance, reliability, and suitability of software according to predetermined requirements and objectives (Sommerville & Sawyer, 2014). Evaluation is crucial to ensure that the developed software meets minimum expected standards such as performance, reliability, security, and functionality. Additionally, evaluation is valuable for optimizing software performance by identifying weaknesses that need improvement, thereby making issue identification more efficient before user deployment (Pfleeger & Atlee, 2015). The evaluation matrix we propose includes several factors:

- Development is an effort to improve, enhance, and adapt the product to follow current trends, preferences, and social conditions (Zhang et al., 2021).
- Response time is a critical component in system performance evaluation as it relates to how quickly the system can respond to user actions to produce appropriate outputs (Amurrio et al., 2020).
- The concept of software fault proneness is unclear and can be evaluated through multiple methods. Errors can arise at any phase of the SDLC, and some may escape detection during testing, only to become apparent during actual use in the field (Phung et al., 2023).
- Availability refers to the system's ability to sustain operation or accessibility despite component failures or cyber-attacks (Tlili & Chelbi, 2022).
- Maintenance can be defined as the system's ability to be modified, upgraded, and repaired, or its adaptability (Zhou et al., 2020).
- Scalability is the system's ability to handle increased workloads without compromising overall system performance (Chechina et al., 2017).
- Deployment is a series of procedures to activate all software services so they can be accessed by users (Aksakalli et al., 2021).

These seven factors are considered sufficient to support the development of robust, efficient software systems that can adapt to future needs. In these case the evaluation conducted using two types of SA: microservices and monolithic architectures.

1.2 Microservices and Monolithic Architecture

Microservices are a software development model that breaks down each function/feature into smaller, simpler components, making deployment easier due to their independent nature (Lewis & Fowler, 2014), (Posta, 2016), (Rajesh, 2016). This architecture was first pioneered by Netflix in 2011 and gained popularity in subsequent years, being adopted by companies such as Amazon, eBay, Zalando, Spotify, Uber, Airbnb, LinkedIn, Twitter, Groupon, and Coca-Cola. The architecture of Microservice can be seen in figure 1.

The microservice architecture consists of two services: 'city service' and 'route service', each with separate databases and web API routes. When a user accesses one or both services, the user request is forwarded separately through the API gateway to the appropriate service. Once the request has been processed, the web API of each service sends the response back to the API gateway, which then forwards it to the user as output.

Monolithic architecture combines all modules, features, functions, databases, and servers into a single application unit (Dragoni et al., 2017). This architecture is still widely used today because of its centralized control over interconnected components.The architecture of Monolithic can be see in figure 2.

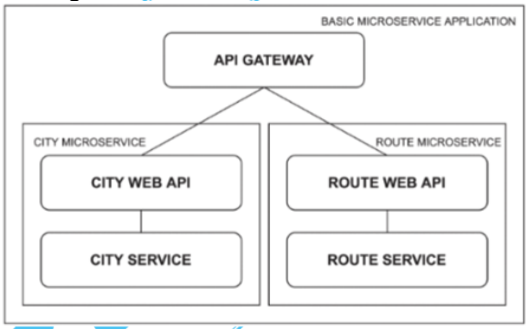

Figure 1. Microservices Logical Architecture

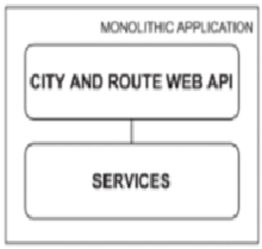

Figure 2. Monolithic Logical Architecture

The monolithic architecture consists of two services, 'city service' and 'route service', combined into a single component with a shared database and a single web API. When a user makes a request, it is forwarded through the city and route web API and directly sent to the service for processing. Once the request is processed, the response is sent back to the user through the city and route web API as output.

In summary, evaluating SA with DREAMS D matrix evaluation is crucial for developing software systems that meet high standards of performance, reliability, and adaptability.

II. METHODS

The method used in these paper is a systematic review through collecting data from several journals related to the evaluation of monolithic and microservice architecture software, comparing factors such as ease of maintenance, availability, response time, development, deployment, error/fault, and scalability.

In the maintenance section, our focus is on evaluating the ease with which developers can modify the software post-release. For availability section, this paper assess the likelihood of the system, whether built with microservice or monolithic architecture, experiencing server downtime, errors, and other failures. The response time section examines the system's speed in responding to user actions when accessing software features. In the scalability section, this paper wants to evaluate the effort required by developers to enhance or add new features in the future.

In the deployment section, the paper aims to evaluate the ease of the deployment process for both architectures and its impact on the overall system. The development section assesses the effort required by developers during system development, including the software testing process. Finally, the error/fault section examines the overall impact of errors on the system and the effectiveness of the system's recovery process.

Next, in the process of collecting journals, authors separated each journal by keywords following the pattern "factor" + "software architecture," for example, "maintenance in microservice." If no journals were found using these keywords, authors modified the keywords to "factor" + "analysis" or "factor" + "in software," such as "availability analysis." Another approach that this paper took was to gather several systematic review journals related to microservice and monolithic architectures and then search for additional factors not covered in these journals by consulting other general journals.

III. RESULTS AND DISCUSSION

DREAMS D MATRIX selects several critical components to test and analyze the quality of software developed with specific architectures. Shown in Table 1.

Table 1. Components to test and analyze

Scaling Factors	Monolithic	Microservice
Development	Monolithic architectures are typically developed as a unified whole simultaneously, so each module is integrated into one with complexity ranging from low to high. Therefore, in development, there are specific requirements such as compatible operating systems, versions, and others, making it less flexible compared to microservices. However, in terms of cost, monolithic architectures are cheaper because the development process is conducted only once on a large scale for the entire system (Mendonça et al., 2021), (Bajaj et al., 2021).	From the paper, it can be concluded that microservices have an advantage in development when dealing with higher complexity and more components because each module in the system is independent and can adapt to containerization, thereby facilitating deployment across different operating systems. On the other hand, this increases the development costs for each separate component/module of the system (Malhotra et al., 2024).
Response Time	From the experimental results, it can be concluded that when the number of virtual machines (VMs) used is still 1, the performance of monolithic architecture is better than that of microservices. This is evidenced by the throughput (handling request) reaching 24% with Java, while microservices with a single VM (MSx1) only reach 9%. The monolithic	From the experimental results, it can be concluded that as the number of virtual machines (VMs) increases, the performance of microservices is better than that of monolithic architectures. This is evidenced by the vertical scaling efficiency of microservices reaching 200%, compared to only 50% for monolithic architectures. Furthermore, in terms of distributed computing based

	architecture is capable of handling 2 times and 1.37 times more requests in .NET and Java, respectively, compared to microservices. The CPU usage and Java/.NET configuration do not significantly affect throughput, which is around 3.5% (Blinowski et al., 2022).	on throughput, microservices are more dominant compared to monolithic architectures, even though both are already Pareto efficient (Blinowski et al., 2022).
Error/Fault	Monolithic architectures have more complex error handling involving testing and integration of the entire system when there are changes to the code or system development because all modules are interconnected as a single unit. This requires coordination with the entire development team (Mendonça et al., 2021), (Bajaj et al., 2021), (Cerny et al., 2020).	Microservices have better error handling compared to monolithic architectures because of their independent nature. This allows fixes and updates to be applied separately to specific modules without affecting unrelated components. Similarly, re-testing of new code can be done independently from unrelated parts of the system (Mendonça et al., 2021), (Bajaj et al., 2021), (Cerny et al., 2020).
Availability	Monolithic architectures have lower availability compared to microservices because the components in monolithic architectures are integrated into a single unit. Therefore, when an error occurs, it affects all related modules/compone nts.	From the paper, it can be concluded that the availability of microservices is higher compared to monolithic architectures because each component is separate. Therefore, when an error occurs in one part of the system, the overall system can continue to operate without disruption (Auer et al., 2021).
Main-tenance	According to Auer, F et al. (Auer et al., 2021) maintenance in monolithic architectures is more complex compared to microservices because the development team needs to consider the overall system architecture and the interaction between components/modu les. Therefore, when developing and modifying the system, testing needs to be conducted comprehensively across all related components.	Microservices have easier maintenance compared to monolithic architectures because each component is separate. Thus, when a bug or code error occurs, it does not affect other components, and the re-testing process for updates or code fixes is only conducted on the relevant system components due to their loose coupling nature (Auer et al., 2021).
Scalability	From the experimental results, it can be concluded that scaling up a monolithic architecture is better than microservices in a single VM condition with low complexity and a smaller number of users because its distributed computing is lower than that of microservices. This means that when user requests are too many, the performance and efficiency of the monolithic	From the experimental results, it can be concluded that scaling up microservices is superior with vertical and horizontal scaling achieving a total increase of 30% compared to monolithic architecture, with Pareto efficiency higher than monolithic in cost-route service to check load distribution. This demonstrates good distributed computing from the microservices

	architecture will decrease (Blinowski et al., 2022).	architecture when handling large numbers of requests. In this case, the testing was conducted using Java and C#.NET with the help of the Azure Cloud platform (Blinowski et al., 2022).
Deployment	The paper concludes that monolithic architecture utilizes the concept of simultaneous deployment. Each component in the monolith is tested first before deployment, so if the system encounters issues or changes in the code, the entire system undergoes retesting, and the latest fixes are queued for deployment. This process heavily depends on team coordination within the system because it is vulnerable to failures in CI/CD during redeployment (Malhotra et al., 2024).	The paper concludes that independent deployment can enhance the resource efficiency of microservices by implementing the principles of continuous integration/continuous development (CI/CD). This is because each component is deployed separately and can be fixed at any time. However, in some conditions, it can be problematic because independent deployment takes more time and occurs gradually, making documentation more difficult (Aksakalli et al., 2021).

Development effort is tested to gauge the resources required in the overall system development process, including total costs incurred by the development team, system compatibility levels, and versioning. For example, monolithic architectures are developed as a unified whole, resulting in lower costs compared to microservices. However, in terms of compatibility, microservices excel due to their containerization capabilities and flexibility.

Response time is tested to measure the system's resilience and speed in handling user requests, typically through throughput indicators. For example, in a single VM scenario, monolithic architectures excel in handling user requests initially. However, as system complexity increases, microservices, with their distributed computing capability through load balancing, become more effective in handling user requests.

Error handling is tested to assess how systems developed with specific architectural models manage errors and faults. For example, microservices demonstrate superior error handling in bug contexts because each module within its components is separate, allowing independent fixes and re-testing of code.

Availability is tested to measure the total operational time of the system and the impact of failures on the overall system. For example, in microservices, if a failure occurs, the entire system remains unaffected because each component is separate. In contrast, in monolithic architectures, a failure in one component affects the entire system due to their interconnected nature.

Maintenance is tested to assess the system's capability to evolve through modifications and fixes. For example, microservices architecture excels in large-scale or complex systems and allows independent system development compared to monolithic architectures, which are integrated into a single system.

Scalability is tested to assess the system's ability to scale horizontally and vertically. For example, vertical scaling efficiency in microservices can reach 200%, whereas monolithic architectures typically achieve only 50% efficiency when the number of VMs increases or system complexity rises. Horizontal scaling involves adding server instances to manage user load, while vertical scaling entails upgrading components such as CPU, memory, and RAM within a single server.

Deployment is tested to understand how the system is deployed, including its components and their integration. For instance, microservices exhibit independent deployment of components and modules.

Based on the seven points, the author proposes the following scores for the overall evaluation components:

Table 2. Evaluation

Architecture	D	R	E	A	M	S	D	**Total Score**
Monolithic	**2**	**2**	**1**	**1**	**2**	**2**	**2**	**12**
Microservice	**2**	**3**	**3**	**3**	**3**	**3**	**2**	**19**

Scoring Criteria:

1. The scale used for evaluation ranges from 1-3, where:
   - 1 means poor,
   - 2 means fair,
   - 3 means good.
2. Sum the total score from all factors. Based on the final score:
   - A score of 1-7 indicates that the SA is not suitable for system development.
   - A score of 8-15 indicates that the SA is fairly stable for system development.
   - A score of >15 indicates that the SA is suitable for system development.

As a note, the author's evaluation is based on a scenario of a complex and highly adaptive system, with a developer team having diverse programming language backgrounds.

IV. CONCLUSION

The evaluation matrix model proposed in this paper aims to simplify the process for development teams in choosing a suitable software architecture (SA) for system development. It serves as a benchmark to determine whether an ongoing or completed system project should be transitioned to a different software architecture, considering seven primary factors that define software quality. However, further research is crucial to evaluate the effectiveness of this matrix model with alternative architectural models. This is particularly important as scoring assessments in architectures with uncertain conditions must align closely with desired software requirements. Moreover, the study's focus on monolithic and microservices architectures underscores the need for broader investigation into other software architecture.

REFERENCES

Aksakalli, I. K., Celik, T., Can, A. B., & Tekinerdogan, B. (2021). Systematic approach for generation of feasible deployment alternatives for microservices. IEEE Access, 9, 29505–29529. https://doi.org/10.1109/ACCESS.2021.3057582

Amurrio, A., Azketa, E., Gutierrez, J. J., Aldea, M., & Harbour, M. G. (2020). Response-time analysis of multipath flows in hierarchically-scheduled time-partitioned distributed real-time systems. IEEE Access, 8, 196700–196711. https://doi.org/10.1109/ACCESS.2020.3033461

Auer, F., Lenarduzzi, V., Felderer, M., & Taibi, D. (2021). From monolithic systems to microservices: An assessment framework. Information and Software Technology, 137, 106600. https://doi.org/10.1016/j.infsof.2021.106600

Bajaj, D., Bharti, U., Goel, A., & Gupta, S. C. (2021). A prescriptive model for migration to microservices based on SDLC artifacts. Journal of Web Engineering, 20(3), 817–852. https://doi.org/10.13052/jwe1540-9589.20312

Bao, T., Liu, S., & Wang, X. (2011). Research on trustworthiness evaluation method for domain software based on actual evidence. Chinese Journal of Electronics, 20(2), 195–199.

Blinowski, G., Ojdowska, A., & Przybyłek, A. (2022). Monolithic vs. microservice architecture: A performance and scalability evaluation. IEEE Access, 10, 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803

Cerny, T., et al. (2020). On code analysis opportunities and challenges for enterprise systems and microservices. IEEE Access, 8, 159449–159470. https://doi.org/10.1109/ACCESS.2020.3019985

Chechina, N., et al. (2017). Evaluating scalable distributed Erlang for scalability and reliability. IEEE Transactions on Parallel and Distributed Systems, 28(8), 2244–2257. https://doi.org/10.1109/TPDS.2017.2654246

Dragoni, N., et al. (2017). Microservices: Yesterday, today and tomorrow. In Present and Ulterior Software Engineering (pp. 195–216). Springer. https://doi.org/10.1007/978-3-319-67425-4_12

Hasselbring, W. (2018). Software architecture: Past, present, future. In The Essence of Software Engineering (pp. 169–184). Cham, Switzerland: Springer.

Lewis, J., & Fowler, M. (2014, March). Microservices: A definition of this new architectural term. https://www.martinfowler.com/articles/microservices.html

Lim, S., Henriksson, A., & Zdravkovic, J. (2021). Data-driven requirements elicitation: A systematic literature review. Social Networking and Computational Science, 2(1), 1–35.

Malhotra, A., Elsayed, A., Torres, R., & Venkatraman, S. (2024). Evaluate canary deployment techniques using Kubernetes, Istio, and Liquibase for cloud native enterprise applications to achieve zero downtime for continuous deployments. IEEE Access, 12, 87883–87899.

https://doi.org/10.1109/ACCESS.2024.3416087

Mendonça, N. C., Box, C., Manolache, C., & Ryan, L. (2021). The monolith strikes back: Why Istio migrated from microservices to a monolithic architecture. IEEE Software, 38(5), 17–22. https://doi.org/10.1109/MS.2021.3080335

Pfleeger, S. L., & Atlee, J. M. (2015). Software engineering: Theory and practice. Pearson

Phung, K., Ogunshile, E., & Aydin, M. (2023). Error-Type—A novel set of software metrics for software fault prediction. IEEE Access, 11, 30562–30574. https://doi.org/10.1109/ACCESS.2023.3262411

Posta, C. (2016). Microservices for Java developers: A hands-on introduction to frameworks containers. O'Reilly Media.

Rajesh, R. (2016). Spring microservices. Packt Publishing.

Sahlabadi, M., Muniyandi, R. C., Shukur, Z., & Qamar, F. (2022). Lightweight software architecture evaluation for industry: A comprehensive review. Sensors, 22(3), 1252.

Sommerville, I., & Sawyer, P. (2014). Requirements engineering: A good practice guide. John Wiley & Sons.

Tlili, L., & Chelbi, A. (2022). Availibility modeling for dependent competing failure process of deteriorating systems. In 2022 IEEE Information Technologies & Smart Industrial Systems (ITSIS) (pp. 1–6). IEEE. https://doi.org/10.1109/ITSIS56166.2022.10118425

Venters, C. C., Capilla, R., Betz, S., Penzenstadler, B., Crick, T., Crouch, S., et al. (2018). Software sustainability: Research and practice from a software architecture viewpoint. Journal of Systems and Software, 138, 174–188.

Yan, B., Yao, H.-P., Nakamura, M., Li, Z.-F., & Wang, D. (2020). A case study for software quality evaluation based on SCT model with BP neural network. IEEE Access, 8, 56403–56414. https://doi.org/10.1109/ACCESS.2020.2981872

Yang, F., Mei, H., Lu, J., & Jin, Z. (2002). Some discussion on the development of software technology. Acta Electronica Sinica, 30(12A), 1901–1906.

Yang, T., Jiang, Z., Shang, Y., & Norouzi, M. (2021). Systematic review on next-generation web-based software architecture clustering models. Computer Communications, 167, 63–74.

Zhang, X., Tan, Y., & Yang, Z. (2021). Analysis of impact of requirement change on product development progress based on system dynamics. IEEE Access, 9, 445–457. https://doi.org/10.1109/ACCESS.2020.3046753

Zhou, H., Gao, S., Qi, F., Luo, X., & Qian, Q. (2020). Selective maintenance policy for a series-parallel system considering maintenance priority of components. IEEE Access, 8, 23221–23231. https://doi.org/10.1109/ACCESS.2020.2969279