Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems

Conference Paper · November 2022

DOI: 10.1109/ICADEIS56544.2022.10037390

CITATIONS
16

READS
517

4 authors:

Alam Rahmatulloh
Siliwangi University
134 PUBLICATIONS **1,062** CITATIONS
SEE PROFILE

Fuji Nugraha
Siliwangi University
7 PUBLICATIONS **46** CITATIONS
SEE PROFILE

Rohmat Gunawan
Siliwangi University
51 PUBLICATIONS **340** CITATIONS
SEE PROFILE

Irfan Darmawan
Telkom University
79 PUBLICATIONS **456** CITATIONS
SEE PROFILE

Event-Driven Architecture to Improve Performance and Scalability in Microservices-Based Systems

1st Alam Rahmatulloh
Department of Informatics
Siliwangi University
Tasikmalaya, Indonesia
alam@unsil.ac.id

2nd Fuji Nugraha
Department of Informatics
Siliwangi University
Tasikmalaya, Indonesia
fujinugraha16@gmail.com

3rd Rohmat Gunawan
Department of Informatics
Siliwangi University
Tasikmalaya, Indonesia
rohmatgunawan@unsil.ac.id

4th Irfan Darmawan
Department of Information System
Telkom University
Bandung, Indonesia
irfandarmawan@telkomuniversity.ac.id

2022 International Conference Advancement in Data Science, E-learning and Information Systems (ICADEIS) | 978-1-6654-6387-4/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICADEIS56544.2022.10037390

Abstract—**The microservice architectural style can replace the monolithic architecture because of the flexibility to adapt to changing technologies and helps to better organize the development team. However, in its implementation there are still problems when communication between services in microservices uses HTTP synchronous or based on API-Driven. In addition, scalability and performance need to be considered in a microservice architecture. The solution offered to these problems is to apply container technology which is integrated with Event-Driven Architecture (EDA) (asynchronous) to handle internal communication between microservices. So that the results of this study can overcome the problems of scalability and performance in microservices. EDA response time is faster with a percentage increase of 19.18%, as well as a lower EDA error rate of 34.40%, although EDA CPU usage is higher with a percentage decrease of 8.52% compared to API-Driven Architecture. EDA uses more CPU resources.**

Keywords—container technology, docker, event-driven architecture, kubernetes, microservices

I. Introduction

Experts state that currently the world has entered the Industrial 4.0 era. which has the nature of volatility, uncertainty, complexity, ambiguity (VUCA), namely the state of the world with the nature of rapid change, lack of predictability, the absence of a causal chain, and the blurring of reality [1]–[3]. This has an impact on the realm of technology, namely how to create a system architecture that has good capabilities and high scalability.

The microservice architecture is gaining attention from the industry because of the capabilities it offers in optimizing system architecture. This is supported by the fact that according to the International Data Corporation (IDC) by the end of 2021, 80% of cloud-based applications will be developed using a microservice architecture [4]. The microservice architectural style can replace the monolithic architecture because of its flexibility to adapt to technological changes and help better organize the development team [5]. In a microservice architecture, services can stand independently, so the development carried out by a team on one service will not affect other services. Many large companies have developed their applications towards microservice architectures such as Netflix, Spotify, Amazon, LinkedIn, SoundCloud and other companies [6].

The microservice architecture with the various advantages it offers, there are still some problems in its implementation including: frequent communication between services in microservices via synchronous HTTP can reduce system performance [4]. Synchronous HTTP communication is a communication that is usually done by microservices with an API-driven architecture (API-Driven Architecture) [7]. In addition, scalability and performance need to be considered in the microservice architecture [8].

Several solutions to deal with scalability and performance issues in microservice architectures have been tried in previous research including: the use of containers which provide an easy way to scale operations by creating more copies of the service, can help deal with scalability issues [9], and elasticity [6]. Docker is a representative technology that applies containerization techniques, has lightweight characteristics, can help and run many microservices which contribute to higher resource utilization, so as to improve microservice performance [10]. Apart from that, the experimental comparison of RESTful API and RabbitMQ performance analysis on microservice web applications shows that, when a large number of users send requests to the web application at the same time RabbitMQ is more stable than the REST API communication method [11]. Event-Driven can be applied as a way of communicating between microservices, when there is a large volume of data that needs to be processed and when no response is expected [12].

In this study, container technology will be tried to be applied to a microservice architecture that is integrated with an Event-Driven Architecture (event-based architecture) to handle internal communication, so that it can overcome scalability and performance problems in microservices.

II. The Material and Method

A. Microservices

Microservices or microservices can be interpreted as a collection of small independent services or processes that usually communicate to form complex applications [13]. The development carried out by the development team on a microservice will not affect other services because of its independent nature. In contrast to monolithic architectures, microservices encourage independent deployment and can be developed using different technology stacks [14]. Each microservice is built around a business capability, runs in its own process, and communicates with other microservices in the application through lightweight mechanisms. The microservice architectural style can be seen as a natural extension of service oriented architecture (SOA), which emphasizes self-management or self-service, and is lightweight [15].

B. Steganography

Containers can be defined as lightweight operating systems that can work directly inside the host operating system [16]. Containerization wraps the application code along with the associated configuration files such as libraries and all dependencies needed to run the application. Containers are abstracted from the host operating system, thus, being portable and self-contained that can run on multiple platform [17]. Many service providers have adopted it for a number of reasons including: (1) to reduce complexity when using microservices; (2) to easily scale, remove, and deploy parts of a system or application; (5) to increase flexibility by using different frameworks and tools; (4) to improve the overall scalability; and (5) increase system resilience [18]. One of the popular application-oriented container approaches to containers is Docker. Docker relies on Linux kernel features, such as namespaces and control groups.

The Docker container encapsulates the application and its software dependencies, and the encapsulated application can run on a different Linux machine than the Docker machine [19]. In making the container, a Docker Image is needed which includes library data, commands, and other application needs. Docker images can be created through a configuration file called a Dockerfile.

Another approach can be done effectively using Kubernetes which is an open-source platform for managing containerized applications including managing workloads and services. Kubernetes is designed to automate deployment, scaling, and operation of containerized applications [17] and has capabilities for portability and extensibility [20]. In its implementation Kubernetes can be assisted by Skaffold which is a command-line tool for handling workflows, building, pushing, and deploying applications or services. Scaffold can be applied to local or remote Kubernetes clusters during application development [21].

Containers can be defined as lightweight operating systems that can work directly inside the host operating system [16]. Containerization wraps the application code along with the associated configuration files such as libraries and all dependencies needed to run the application. Containers are abstracted from the host operating system, thus, being portable and self-contained that can run on multiple platform [17]. Many service providers have adopted it for a number of reasons including: (1) to reduce complexity when using microservices; (2) to easily scale, remove, and deploy parts of a system or application; (5) to increase flexibility by using different frameworks and tools; (4) to improve the overall scalability; and (5) increase system resilience [18]. One of the popular application-oriented container approaches to containers is Docker. Docker relies on Linux kernel features, such as namespaces and control groups.

The Docker container encapsulates the application and its software dependencies, and the encapsulated application can run on a different Linux machine than the Docker machine [19]. In making the container, a Docker Image is needed which includes library data, commands, and other application needs. Docker images can be created through a configuration file called a Dockerfile.

Another approach can be done effectively using Kubernetes which is an open-source platform for managing containerized applications including managing workloads and services. Kubernetes is designed to automate deployment, scaling, and operation of containerized applications [17] and has capabilities for portability and extensibility [20]. In its implementation Kubernetes can be assisted by Skaffold which is a command-line tool for handling workflows, building, pushing, and deploying applications or services. Scaffold can be applied to local or remote Kubernetes clusters during application development [21].

C. Event-Driven Architecture (EDA)

Event-Driven Architecture (EDA) refers to microservice systems that are loosely coupled and exchange information or data with each other through publish and listen events. EDA allows information to be absorbed into an event-driven ecosystem and then broadcast to the listening service or the service that will receive the event [22]. EDA communicates using message events and works asynchronously, in contrast to the API-Driven Architecture which communicates using API calls and works synchronously [7]. In recent years, EDA has been widely used in several domains such as network instruction detection, sensor networks, stock market, fast trading, realtime system control, healthcare monitoring, mobile and wearable computing. The main reason is that EDA provides solutions for developing distributed systems that facilitate high flexibility and concurrency [23].

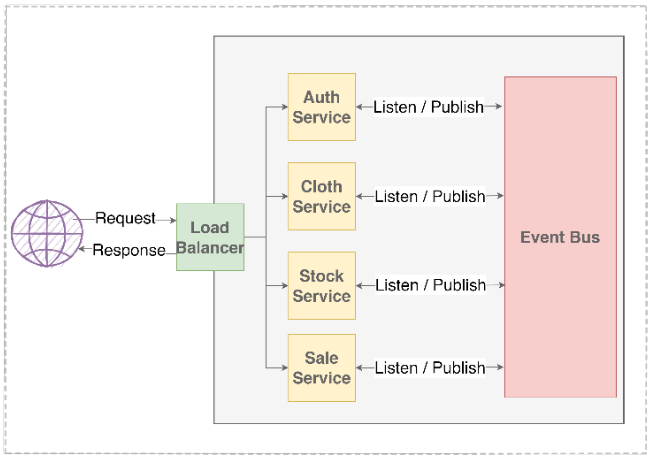

Fig. 1. Event-Driven Architecture Communication Process in Microservice

Fig. 1. describes the event-flow process, that when a client from outside the microservice ecosystem makes a request, it will be forwarded by the Load Balancer and directed to the destination service, where the related service will process the request and publish the event to the event-bus where other services do it. The listen event will receive event notifications to participate in processing requests or synchronizing data. After the request has been processed, it will return a response to the client. Event-bus creation can be assisted using Apache Kafka [22], RabbitMQ [11], or NATS Streaming.

D. Methodology

Fig. 2. is a research stage starting from the literature study process, the application stage (RAD Methodology), the measurement stage, and drawing conclusions.

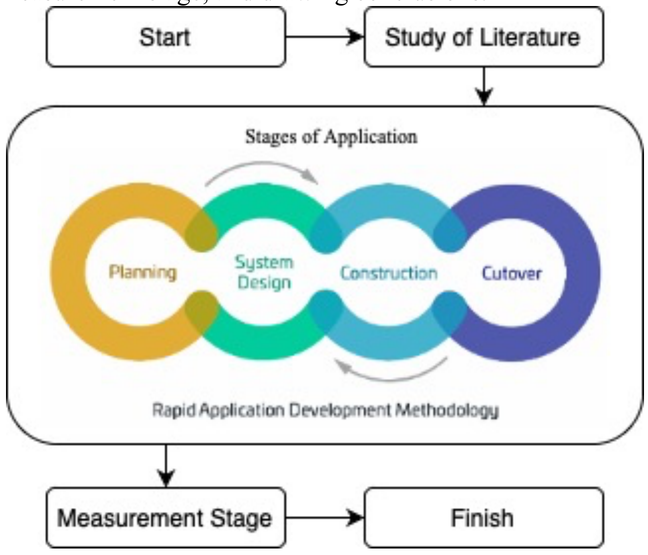

Fig. 2. Research Methodology

1.1. Study of literature

At this stage, learn all things related to Microservice, Containerization, and Event-Driven Architecture from various library sources in the form of books, journals, research reports, theses, and theses that have been done as well as the results of library searches on the internet.

1.2. Stages of Application (RAD Methodology)

The implementation stages in this research use the Rapid Application Development (RAD) Methodology which has four stages, namely planning, system design, construction, and cutover.

1) Planning

At this stage, data flow planning is carried out, development and application of architecture to applications, including planning for software development requirements such as programming languages, tools, libraries, frameworks, and other resources.

TABLE I. SOFTWARE ARCHITECTURE AND SOFTWARE NEEDS

Name	**Version**	**Description**
Docker	Latest	Application oriented container
Kubernetes	1.22.2	Container orchestrator
Skaffold	v2beta20	Kubernetes support
NATS Streaming	0.22	Event bus
Typescript	4.4.4	Programming languange
Node.js	17.1.0	Runtime environment
Express.js	4.17.1	App framework
MongoDB	5.0	Database
Jest	27.1.0	Testing framework
Stan.js	0.3.2	Client communication tool

2) System Design

System Design is the stage of designing the system architecture in accordance with the planning stages that have been carried out previously. Architectural design will be carried out using diagrams.net which is a tool for creating data flows, wireframes, UML and so on.

3) Construction

Construction is the construction stage in accordance with the system design stage. In addition, at this stage testing is also carried out on the application as well as communication between services, if there are still discrepancies, you can return to the system design stage and then return to the construction stage. This is done until it is in accordance with the previous planning stage.

4) Cutover

This stage is the final stage where the system architecture has been running well and in accordance with the initial planning. In this study, if the system architecture has been implemented, measurements will be made for research needs.

1.3. Measurement Stage

At this stage, measurements will be made by evaluating the performance of the implemented architecture. Measurements taken include CPU usage [17] in milliseconds when processing requests from clients, response time in milliseconds when handling a number of requests from clients, and error rate in percentage [11] when handling a number of requests beyond the system's capabilities. The measurement stage is assisted by using JMeter to determine the results of the response time and error rate, and assisted by the process.cpuUsage() API from Node.js to determine the results of CPU usage.

III. RESULT AND DISCUSSION

A. System Design

This research adopts a simple business process from a fabric store information system in the warehousing section until the goods are sold. Fig. 4. describes a system design that defines 4 main services Auth, Cloth, Stock, and Sale, as well as 1 additional service NATS Streaming (event-bus). Each main service is equipped with a configuration to connect to the database which in this case uses MongoDB. Each service has tasks including: Auth to handle all user authentication activities and management of users who can access the system, Cloth to handle the entire process of making fabric metadata, grouping fabrics, and pricing each fabric, Stock managing handle the entire fabric calculation process, both in the stock-in process (stock in) and during the stock-out process (out of stock/sold fabrics), and Sale handles activities when the fabric is sold or the fabric is issued. In addition, NATS Streamer acts as the main bridge in the communication process between the 4 main services, both the publishing process (sending data) and the listening process (receiving data).

B. Measurement Mechanism

Measurements will be made on applications that have been built and the related communication architecture has been applied. The measurement process itself will compare the

Event-Driven Architecture with the API-Driven Architecture to see the amount of performance improvement. The application of the API-Driven Architecture to the applications that were built was not documented in the research, considering that the focus of the research conducted was on the Event-Driven Architecture. Fig. 3. describe the API-Driven Architecture created.

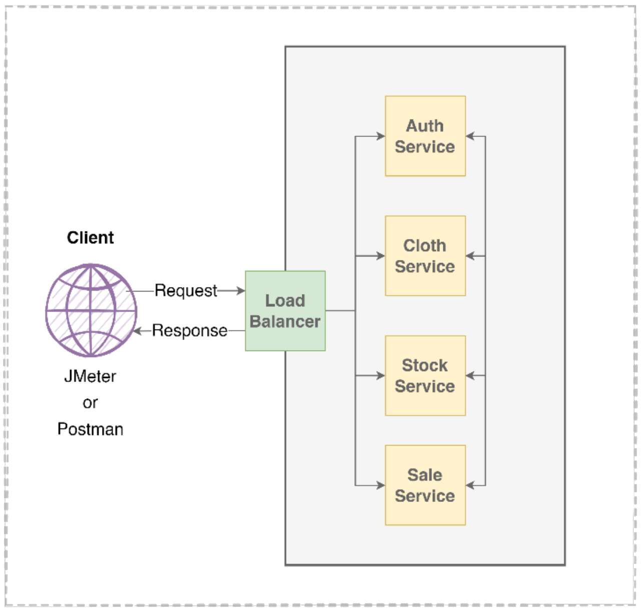

Fig. 3. API-Driven Architecture Communication Process in Microservice

The measurement mechanism includes the preparation of performance measurements for each measurement parameter, namely response time, error rate, and CPU usage. For each parameter, the measurement is carried out with the same number of requests, namely 100, 200, 300, 400, and 500). Measurements are carried out on one of the application endpoints that have been built with the most complex criteria when storing and processing data. The measurement mechanism that will be applied applies to both Event-Driven Architecture and API-Driven Architecture, so that both architectures receive the same treatment when taking measurements.

TABLE II. RAMP-UP PERIOD

Total Request	**Ramp-up Period (second)**		
	Response Time	**Error Rate**	**Cpu Usage**
100	200 s (2 s/request)	100 s (1 s/request)	-
200	800 s (4 s/request)	600 s (3 s/request)	-
300	2400 s (8 s/request)	1500 s (5 s/request)	-
400	6400 s (16 s/request)	2800 s (7 s/request)	-
500	16000 s (32 s/request)	4500 s (9 s/request)	-

The ramp-up period shows how long it takes to execute a specified number of requests. For example, 100 requests must be completed within 200 seconds, in the sense that 1 request only has a maximum of 2 seconds to complete the process. If it exceeds this time, the request process will be delayed and will be completed after other requests. However, if there is no time left for other requests, the request process is terminated and will experience an error because the system is unable to complete the request according to the specified ramp-up period. The ramp-up period is only used when measuring average response time and error rate. When measuring CPU usage, you don't need a ramp-up period because it will run indefinitely, in order to find out how much CPU usage is when processing requests from users/clients.

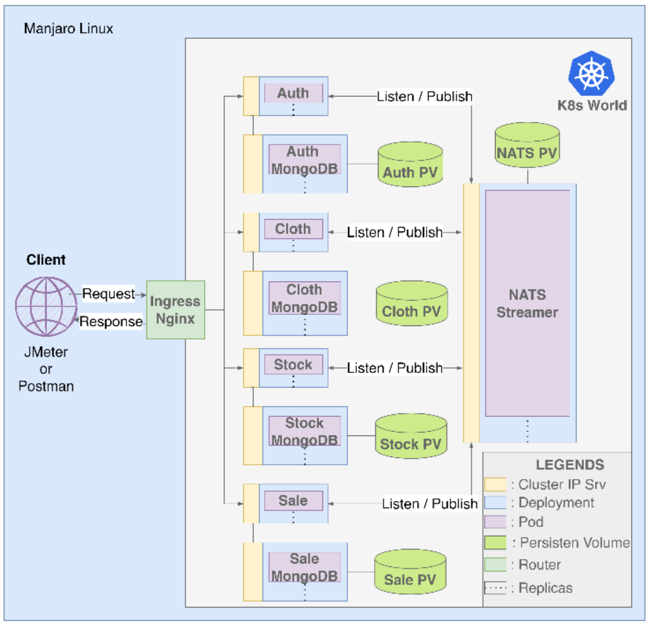

Fig. 4. System Design

The measurement stage uses JMeter to measure the response time and error rate. In addition, setting up CPU usage measurements programmatically uses the help of the Node.js process.cpuUsage() API.

C. Benchmark Result

The measurement environment is exactly the same as the application development environment and the application of the architecture to the applications that have been built. Measurements will be carried out on the Linux Manjaro 21.1.6 Pahvo operating system.

TABLE III. ENVIRONTMENT

Detail	**Description**
Operating System	Manjaro 21.1.6 Pahvo (Linux)
Kernel	x86_64 Linux 5.10.70-1-MANJARO
Disk	110GB
CPU	AMD A4-9120 RADEON R3, 4 COMPUTE CORES 2C + 2G @ 2x 2.25GHz
GPU	AMD STONEY
RAM	8GB

The measurement results from the two architectures are then presented in the form of tables and graphs, to see the differences between the two. After that, the results presented by the two architectures are calculated using the percentage increase or decrease formula.

$$percentage = \frac{final\ value - initial\ value}{initial\ value} \times 100\%$$

TABLE IV. RESPONSE TIME RESULT

Total Request	**Event Driven**	**API Driven**	**Difference**
100	2096 ms	2247 ms	151 ms
200	2905 ms	3852 ms	947 ms
300	3277 ms	3895 ms	618 ms
400	4433 ms	5215 ms	782 ms
500	5805 ms	6859 ms	1054 ms
Average	3703.2 ms	4413.6 ms	710.4 ms

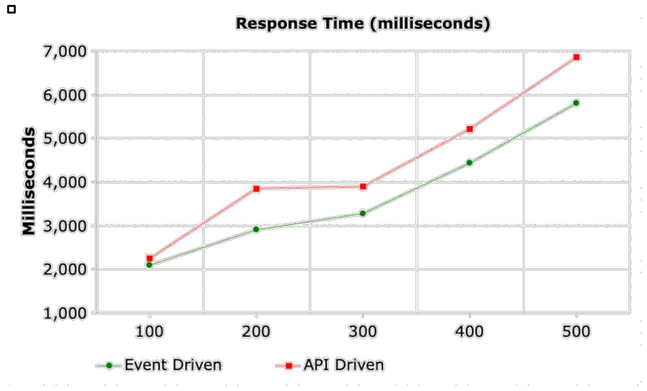

Fig. 5. Response Time Result Line Chart

As shown in Table III and Fig. 5. In terms of response time speed, Event-Driven Architecture outperformed API-Driven Architecture by 19.18%. In measuring the error rate shown in Table IV and Fig. 6. Event-Driven outperforms because it has a lower error rate than API-Driven Architecture with an increase of 34.40%. Although the CPU Usage measurement as shown in Table V and Fig. 7. Event-Driven uses more CPU resources than API-Driven Architecture with a decrease of 8.52%.

TABLE V. ERROR RATE RESULT

Total Request	Event Driven	API Driven	Difference
100	0 %	0 %	0.00%
200	7 %	11 %	4.00%
300	10,33 %	11,33 %	1.00%
400	10,50 %	9,25 %	1.25%
500	4 %	11,20 %	7.20%
Average	6.37%	8.56%	2.19%

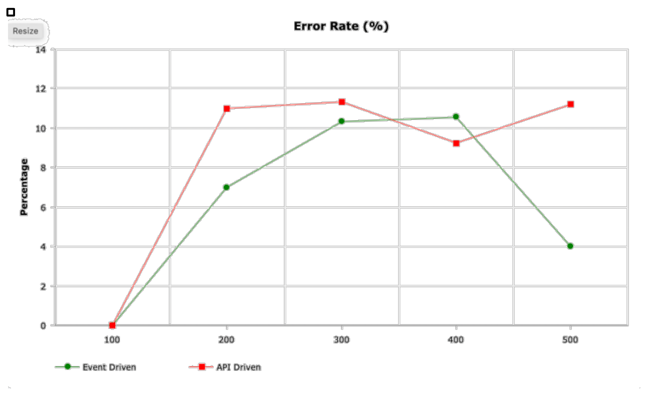

Fig. 6. Error Rate Result Line Chart

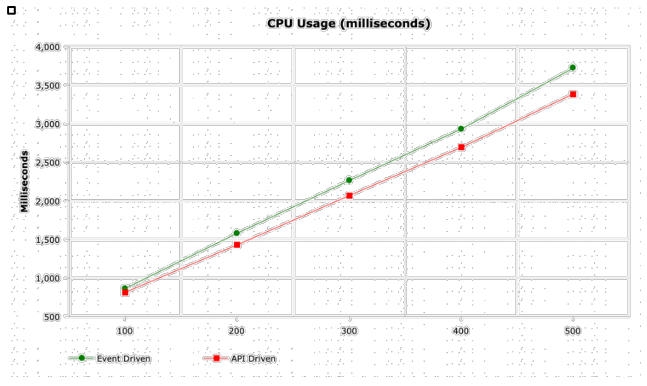

Fig. 7. CPU Usage Result Line Chart

TABLE VI. CPU USAGE RESULT

Total Request	Event Driven	API Driven	Difference
100	865.32 ms	814.76 ms	50.56 ms
200	1521.37 ms	1561.92 ms	150.51 ms
300	2265.83 ms	2071.35 ms	194.49 ms
400	2932.64 ms	2694.58 ms	238.05 ms
500	3721.51 ms	3386.90 ms	334.60 ms
Average	2272.97 ms	2079.32 ms	193.64 ms

IV. CONCLUSIONS

Based on the results of the research that has been done, it can be concluded that the system was successfully built using container technology which is integrated with the Event-Driven Architecture on the Microservice architecture. In addition, the application of container technology (Kubernetes) can make it easier to scale services on a Microservice architecture, both when scale-in and scale-out.

The amount of performance improvement when implementing container technology that is integrated with the Event-Driven Architecture is known through the measurement process of the system that has been built. Measurement is done by comparing the Event-Driven Architecture and API-Driven Architecture. From the measurement results, it is known that the performance of the Event-Driven Architecture is better than the API-Driven Architecture with a number of requests (100, 200, 300, 400, and 500) being carried out on the system simultaneously. From the results of the request, it can be seen that the Event-Driven Architecture response time is faster with a percentage increase of 19.18%, the error rate of Event-Driven Architecture is lower with a percentage increase of 34.40%, although CPU usage Event-Driven Architecture is higher with a percentage decrease of 8.52%. compared to API-Driven Architecture considering that Event-Driven Architecture uses more CPU resources.

REFERENCES

[1] A. Nowacka and M. Rzemieniak, “The Impact of the VUCA Environment on the Digital Competences of Managers in the Power Industry,” *Energies (Basel)*, vol. 15, no. 1, p. 185, Dec. 2021, doi: 10.3390/en15010185.

[2] E. Šimková and M. Hoffmannová, “Impact of VUCA Environment in Practice of Rural Tourism,” Mar. 2021, pp. 746–757. doi: 10.36689/uhk/hed/2021-01-074.

[3] N. J. Pearse, “Change Management in a VUCA World,” in *Visionary Leadership in a Turbulent World*, Emerald Publishing Limited, 2017, pp. 81–105. doi: 10.1108/978-1-78714-242-820171005.

[4] M. Waseem, P. Liang, and M. Shahin, “A Systematic Mapping Study on Microservices Architecture in DevOps,” *Journal of Systems and Software*, vol. 170, p. 110798, 2020, doi: 10.1016/j.jss.2020.110798.

[5] A. Balalaie, A. Heydarnoori, and P. Jamshidi Dermani, “Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture,” *52*, pp. 1–13, 2016.

[6] P. Di Francesco, P. Lago, and I. Malavolta, “Architecting with microservices: A systematic mapping study,” *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019, doi: 10.1016/j.jss.2019.01.001.

[7] E. K. Kannedy, “Event-Driven Architecture,” *medium.com*, 2018. https://medium.com/bliblidotcom-techblog/event-driven-architecture-ef3a312180ee (accessed Mar. 10, 2021).

[8] S. Li *et al.*, “Understanding and addressing quality attributes of microservices architecture: A Systematic literature review,” *Information and Software Technology*, vol. 131, p. 106449, 2021, doi: 10.1016/j.infsof.2020.106449.

[9] D. Trihinas and G. Pallis, “DevOps as a Service : Pushing the Boundaries of Microservice Adoption Taking the Pulse of DevOps in the Cloud,” *IEEE Computer Society*, no. June, pp. 65–71, 2018.

[10] H. Kang, M. Le, and S. Tao, “Container and microservice driven design for cloud infrastructure DevOps,” *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, pp. 202–211, 2016, doi: 10.1109/IC2E.2016.26.

[11] X. J. Hong, H. Sik Yang, and Y. H. Kim, “Performance Analysis of RESTful API and RabbitMQ for Microservice Web Application,” *9th International Conference on Information and Communication Technology Convergence: ICT Convergence Powered by Smart Intelligence, ICTC 2018*, pp. 257–259, 2018, doi: 10.1109/ICTC.2018.8539409.

[12] A. Akbulut and H. G. Perros, “Performance Analysis of Microservice Design Patterns,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019, doi: 10.1109/MIC.2019.2951094.

[13] R. A. Putra, “Analisa Implementasi Arsitektur Microservoces Berbasis Kontainer Pada Komunitas Pengembang Perangkat Lunak Sumber Terbuka (OpenDayLight DevOps Community),” *Jurnal Sistem Infomasi Teknologi Informasi dan Komputer (Just It) Universitas Bina Nusantara Magister Manajemen Sistem Informasi Jakarta*, pp. 150–162, 2018.

[14] L. Baresi, M. Garriga, and A. De Renzis, “Microservices identification through interface analysis,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10465 LNCS, pp. 19–33, 2017, doi: 10.1007/978-3-319-67262-5_2.

[15] J. Soldani, D. A. Tamburri, and W. J. Van Den Heuvel, “The pains and gains of microservices: A Systematic grey literature review,” *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018, doi: 10.1016/j.jss.2018.09.082.

[16] M. Fihri, R. M. Negara, and D. D. Sanjoyo, “Implementasi & Analisis Performansi Layanan Web Pada Platform Berbasis Docker Implementation & Analysis of Web Service Performance Based on Docker Platform,” vol. 6, no. 2, pp. 3996–4001, 2019.

[17] L. P. Dewi, A. Noertjahyana, H. N. Palit, and K. Yedutun, “Server Scalability Using Kubernetes,” *TIMES-iCON 2019 - 2019 4th Technology Innovation Management and Engineering Science International Conference*, pp. 1–4, 2019, doi: 10.1109/TIMES-iCON47539.2019.9024501.

[18] H. Khazaei, C. Barna, N. Beigi-Mohammadi, and M. Litoiu, “Efficiency analysis of provisioning microservices,” *Proceedings of the International Conference on Cloud Computing Technology and Science, CloudCom*, vol. 0, pp. 261–268, 2016, doi: 10.1109/CloudCom.2016.0051.

[19] M. Plauth, L. Feinbube, and A. Polze, “A performance survey of lightweight virtualization techniques,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10465 LNCS, pp. 34–48, 2017, doi: 10.1007/978-3-319-67262-5_3.

[20] T. Menouer, “KCSS: Kubernetes container scheduling strategy,” *Journal of Supercomputing*, vol. 77, no. 5, pp. 4267–4293, 2021, doi: 10.1007/s11227-020-03427-3.

[21] K. P. Singh, “Easy Kubernetes development with Skaffold,” *dev.to*, 2020. https://dev.to/karanpratapsingh/easy-kubernetes-development-with-skaffold-2ic8 (accessed Nov. 16, 2021).

[22] G. Jansen and J. Saladas, “Advantages of event-driven architecture,” *Developer IBM*, 2020. https://developer.ibm.com/technologies/messaging/articles/advantages-of-an-event-driven-architecture/ (accessed Mar. 13, 2020).

[23] S. Tragatschnig, S. Stevanetic, and U. Zdun, “Supporting the evolution of event-driven service-oriented architectures using change patterns,” *Information and Software Technology*, vol. 100, no. March, pp. 133–146, 2018, doi: 10.1016/j.infsof.2018.04.005.