with the service, thus triggering the development team to follow up.

In a microservices architecture, you can dynamically scale services with heavy loads; this makes resource use effective. Micro-services that are small and autonomous are easier to deploy and have little potential to cause system failure when something goes wrong [18]. By leveraging Docker containers, instant services can be implemented with lower overhead than through operating-system virtualization[19]. These containers run on a cluster-management infrastructure such as Apache Mesos to manage load balancing between containers in the cluster [20].

B. Asynchronous Communication Pattern

A microservices-based application is a distributed system running on multiple processes or services. Services must interact using inter-process communication such as HTTP, AMQP, or RPC calls. It is very important to consider the choice of inter-service communication patterns and execution flow in the microservice architecture. This communication can occur synchronously or asynchronously [21].

Synchronous communication is a communication style in which the caller waits until an answer is available. This communication style is widely used because it is simple and easy to implement. Although synchronous calls are simpler to understand, debug, and implement, a few trad-offs are considered. Synchronous communication makes services vulnerable to cascading failures. If downstream services fail or take too long to respond, resources can run out quickly. This can cause a domino effect on the system. Synchronous integration is not recommended for inter-service communication. They do not allow microservices to become autonomous, and also, in one service failure, the overall performance was affected. As synchronous dependence between microservices increases, the overall response time for clients becomes worse.

In the asynchronous type of communication, the caller does not need to wait for a response from another service, so dependence between services can be avoided. In addition, asynchronous communication allows several services to be called in parallel. The application of asynchronous communication is possible with several variations. At least three common techniques are typically used in inter-service communication in microservices architectures [22].

1) HTTP-based Communication: The service called the service destination directly using the HTTP protocol in this inter-service communication. Usually, HTTP-based communication is synchronous communication where the service caller takes the next step until the service call is complete. Apart from synchronous inter-service communication, an HTTP-based communication, we can also make service calls in asynchronous HTTP-based communication. Asynchronous HTTP-based communication is carried out with HTTP polling, where the service makes requests to other services and then check separately to find out the status of the request. With this approach, services remain isolated from each other, and the coupling is loose. The downside is that it creates additional HTTP requests on the second service. This also causes complexity to the client as it now has to check the progress of the request.

2) Message-based Communication: Another communication pattern that we can use in a microservice architecture is message-based communication. Unlike HTTP-based communication, the services involved do not communicate directly. Instead, a service pushes messages to a message broker; then, other services can choose to subscribe to messages at a broker they care about. This eliminates a lot of the complexity associated with HTTP communication. In this type of communication, checking the request's progress can be done using the Message-Id obtained from the message broker. To communicate properly, each service must make a contract regarding the structure of the message and its contents; this shows that there is still a coupling between services.

3) Event-driven Communication: Another communication pattern is event-driven communication. Unlike messaging patterns where the service must know the message's structure and content, this approach does not require it. Communication between services takes place via events that individual services produce[23]. Message brokers are still needed here as the service can write their events to them. However, unlike the messaging approach, the consuming service does not need to know the event's details; they react to events. Services can listen to events they care about, and they know what logic to execute in response to them. This pattern makes services loosely coupled as no payload is included in the event [24].

C. Choreography-based

Microservice architecture is a collection of small services, with each service having a specific function. This service module cannot perform well in isolation and requires some type of media to interact and share data. There are two ways to unify these service modules: microservice orchestration and microservices choreography [25].

The orchestrator (central controller) handles all microservices interactions in microservices orchestration. It transmits events and responds to them. Microservice orchestrations are more like centralized services. It calls one service and waits for a response before calling the next service. It follows the request-response type paradigm[26].

In microservice choreography, each microservice performs its activities independently, and it does not require any instructions. This is like a decentralized way of broadcasting data known as events. Services that are interested in the event use them and take action. This is also known as reactive architecture[27]. The service knows what to react to and how-to, which is more like an asynchronous approach. So, this approach can be used to solve the inter-service interdependence problem that exists in the orchestration approach [28].

D. Smart Village Application

The smart village application is a village-based online marketplace that facilitates various business actors' buying and selling processes in a village. Business actors can upload the products they offer. There are 5 types of products managed in this application: lodging reservations, tourist attraction tickets, culinary purchases, purchasing knick-knacks, and purchasing show tickets. Products uploaded can be ordered by visitors. After making a payment, the system sends a voucher via email. The voucher can be redeemed according to the product ordered. Payment processing can be made online via credit card, ATM, mobile or internet banking, as well as digital money.

Apart from being a medium for buying and selling online, this smart village application is also used as a medium to introduce each village's potential. This application is targeted to facilitate all village-based business units in Bali in promoting and selling their products globally. With high potential users in smart village applications, the system architecture must be scalable, fast response, and fault tolerance. In addition, the potential for the system to be developed in the future is very high, such as adding rating features, sales reports, mapping village potential, integrating delivery services, and a recommendation system that makes it easier for visitors to plan tourist visits. This requires that the application be developed by applying a design that is easy to develop.

E. Web Service Implementation Methodology

The Web Service Implementation Methodology defines a systematic approach to Web Service development by leveraging agile software development methodologies and extending that methodology by defining Web Service-specific activities. This methodology defines a set of general practices that create a method-independent framework, which most software teams can apply to developing Web Service applications. The Web Service Implementation Lifecycle refers to developing a Web Service from the requirement to deployment. The Web Service implementation lifecycle typically includes the following stages: Requirements Phase; Analysis Phase; Design Phase; Coding Phase; Test Phase; Deployment Phase. These phases may overlap with each other during the implementation process [29].

1) Requirement phase: This requirement phase aims to understand business requirements and translate them into microservices requirements in terms of features and functional and non-functional requirements. The requirement analysis process must involve the project stakeholders to obtain a suitable requirement. After this, the requirements of the analysis results are communicated to the development team.

2) Analysis Phase: In the analysis phase, the micro-service requirements are further refined into a conceptual model that the technical development team can understand. In this phase, architectural analysis is also carried out to define high-level structures and identify micro-service interface contracts.

3) Design Phase: The detailed microservice design is carried out in this phase. In this phase, it is necessary to define the micro-service interface contracts identified in the analysis phase. The defined interface contract must identify the appropriate data element and type and the mode of interaction between services.

4) Coding Phase: The coding and debugging phases for microservices implementation are basically very similar to the coding and debugging phases based on other software components. The main difference lies in the creation of an additional microservice interface wrapper. Additional microservices must be deployed to the Web Server / Application Server before test clients can use them.

5) Test Phase: For testing microservices, testers must also perform interoperability testing between different platforms and client programs apart from testing for correctness and completeness of functions. In addition, performance testing should be carried out to ensure that the microservices able to withstand the maximum loads and stresses specified in the non-functional requirements specification [30].

6) Deployment Phase: The deployment phase aims to ensure microservices are properly deployed. This phase run after the microservices are tested. The deployer's main task is to ensure that the microservices are properly configured and managed as well as to run post-deployment tests to ensure that microservices are ready to use.

III. RESULT AND DISCUSSION

A. Requirement Result

The requirements analysis process is carried out by analyzing the smart village application's functional and non-functional requirements. The non-functional analysis is carried out by analyzing technology architecture by identifying the technologies needed to develop smart village applications. Technology analysis is carried out on the hardware and software. The results of the non-functional analysis can be seen in Table 1.

TABLE I
NON-FUNCTIONAL REQUIREMENT

Technology	Description
Vue.js	Front-end framework that used to create client applications
SLIM Framework	PHP micro-framework that used to create microservices
MySQL	As a database in each microservice
Swagger	As an interface and documentation for each microservice
Docker	As a container or container for each microservice
Kong	An open-source that used to create an API gateway
RabbitMQ	An open sources software that is used as a message broker

System functional analysis is carried out by identifying each business process contained in the smart village application. Functional systems are then grouped based on similar functionalities. Table 2 provides functional grouping and mapping information in the smart village application. Each microservice can represent one or more functional groups. Similar functional groups can be combined into the same microservice.

TABLE II
FUNCTIONAL GROUP

Group	Functional
Product	- Manage product data
Email	- Send notification email
Customer	- Manage customer data
Owner	- Manage data owner
Order	- Order products - Manage order data
Payment	- Make transactions to a payment gateway - Receive transaction status from the payment gateway

Authentication	- Login - Forgot the password - Change the password

From the analysis of functional group relationships, 7 microservices were produced, which will be developed in the smart village application. Table 3 is information about microservices that will be developed in the smart village application.

TABLE III
MICROSERVICES IN SMART VILLAGE APP

Functional Group	**Microservice**
Product	Product microservice
Email	Email microservice
Customer	Customer microservice
Owner	Owner microservice
Order	Order microservice
Payment	Payment microservice
Authentication	Auth microservice

B. Smart Village Microservices Architecture

The Smart Village Microservices Architecture is designed based on the results of the non-functional analysis. The Smart Village Microservices Architecture can be seen in Fig 1. This architecture consists of 4 main components, i.e., Client Application, API Gateway, Microservices, and Event Bus.

1) Client Application: The Client Application is a website-based application that the user can access directly. This application is an interface between users and the smart village application. The client application is built with a modern front-end framework, namely Vue.js, and then hosted on a web server.

2) API Gateway: API Gateway is a service-based application that is used as an intermediary so that client applications can interact with several microservices. This API Gateway serves as an access gateway from outside (internet network) to inside (internal microservice network). The application client cannot directly access the microservices; the request must go through the Gateway API then be forwarded to the microservices. This aims to increase the security of microservices. In this architecture, the API Gateway was built using Kong (written in Lua).

3) Microservices: This microservice component consists of a collection of microservices developed for the smart village application. Each microservice represents a business process in the smart village application. There are 7 microservices, i.e., Product microservice, Mail microservice, Owner microservice, Customer microservice, Order microservice, Auth microservice, and Payment microservice. Every microservice on this architecture was built using the SLIM framework (written in PHP).

4) Event Bus: The Event Bus is a component that regulates communication between microservices. In this architecture, RabbitMQ is used to perform this task. RabbitMQ is one of the most widely used open-source message brokers. RabbitMQ service bus acts as a link between several microservices where microservices can publish messages under the different number of queues available in the RabbitMQ service bus. Other microservices could subscribe to these messages available in the RabbitMQ service bus queue. The microservices performed their logical functions after receiving the event.

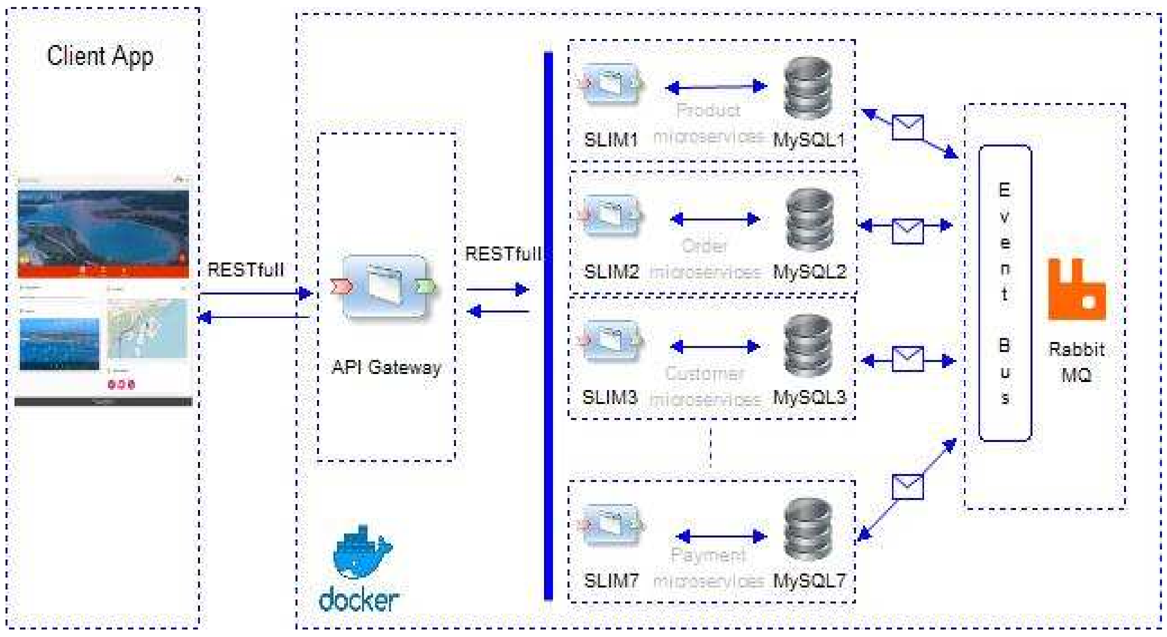


Fig. 1 Smart Village Microservices Architecture

C. Microservice Result

Each microservice is developed independently and runs on a different node. In the smart village application, each microservice is developed using SLIM. SLIM is a PHP micro-framework explicitly developed for creating web services. While the database used is MySQL.

1) Product microservice: The functions that exist on this microservice, i.e., listing product, getting the product by product id, creating a new product, updating product, delete the product. Design API for product microservice can be seen in Table 4. Design is made according to the REST perspective.

TABLE IV
PRODUCT MICROSERVICES API

Method	URI	Use Case
GET	api/v1/product	Listing product
GET	api/v1/product/{id}	Get product by id
POST	api/v1/product	Create new product
PUT	api/v1/product	Update product
DELETE	api/v1/product/{id}	Delete product

2) *Customer microservice:* The functions in this microservice include listing customers, getting customer by id, creating new customer, updating customer, delete the customer. The design API for customer microservices can be seen in Table 5.

TABLE V
CUSTOMER MICROSERVICE API

Method	URI	Use Case
GET	api/v1/customer	Listing customer
GET	api/v1/customer/{id}	Get customer by email
POST	api/v1/customer	Create new customer
PUT	api/v1/customer	Update customer
DELETE	api/v1/customer/{id}	Delete customer

3) *Owner microservice:* The functions that exist in this microservice include listing owner, get owner by id, create new owner, update owner, delete owner. Design API for the microservice owner can be seen in Table 6.

TABLE VI
OWNER MICROSERVICE API

Method	URI	Use Case
GET	api/v1/owner	Listing owner
GET	api/v1/owner/{id}	Get owner by email
POST	api/v1/owner	Create new owner
PUT	api/v1/owner	Update status order
DELETE	api/v1/owner/{id}	Delete owner

4) *Order microservice:* The functions that exist in this microservice create new orders, get the order-by-order id, list orders, update order status. The design API for microservice orders can be seen in Table 7.

TABLE VII
ORDER MICROSERVICE API

Method	URI	Use Case
GET	api/v1/order	Listing order
GET	api/v1/order/{id}	Get order by id
POST	api/v1/order	Create new order
PUT	api/v1/order	Update owner

5) *Email microservice:* The functions in this microservice include sending emails according to templates such as registers, forget passwords, invoices, payment statuses, and product vouchers. The process sends an email using the Mailgun service. The design API for microservice orders can be seen in Table 8.

TABLE VIII
EMAIL MICROSERVICE API

Method	URI	Use Case
POST	api/v1/mail/register	Sending email register
POST	api/v1/mail/forgetpass	Sending email forget
POST	api/v1/mail/invoice	Sending email invoice
POST	api/v1/mail/payment	Sending email payment
POST	api/v1/mail/voucher	Sending email voucher

6) *Payment microservice:* The functions that exist in this microservice include making transactions to payment gateways and receiving transaction status from payment gateways. The design API for payment microservice can be seen in Table 9.

TABLE IX
PAYMENT MICROSERVICE API

Method	URI	Use Case
POST	api/v1/pay	Create payment
POST	api/v1/pay/notification	Recive payment response

7) *Auth microservice:* The functions that exist in this microservice include user authentication, changing passwords, and resetting passwords. The design API for auth microservice can be seen in Table 10.

TABLE X
AUTH MICROSERVICE API

Method	URI	Use Case
POST	api/v1/auth	Login
PUT	api/v1/auth	Change password
POST	api/v1/auth/reset	Reset password

D. API Gateway Result

API Gateway is used as an intermediary to interact with microservices so that client applications can interact. The client application cannot directly access the microservices, and the request must go through the API Gateway, then be forwarded to the microservices. API Gateway contains a mapping between the API route on the API Gateway and the API route on the microservices. Detailed route mapping can be seen in Table 11.

TABLE XI
API GATEWAY MAPPING ROUTE

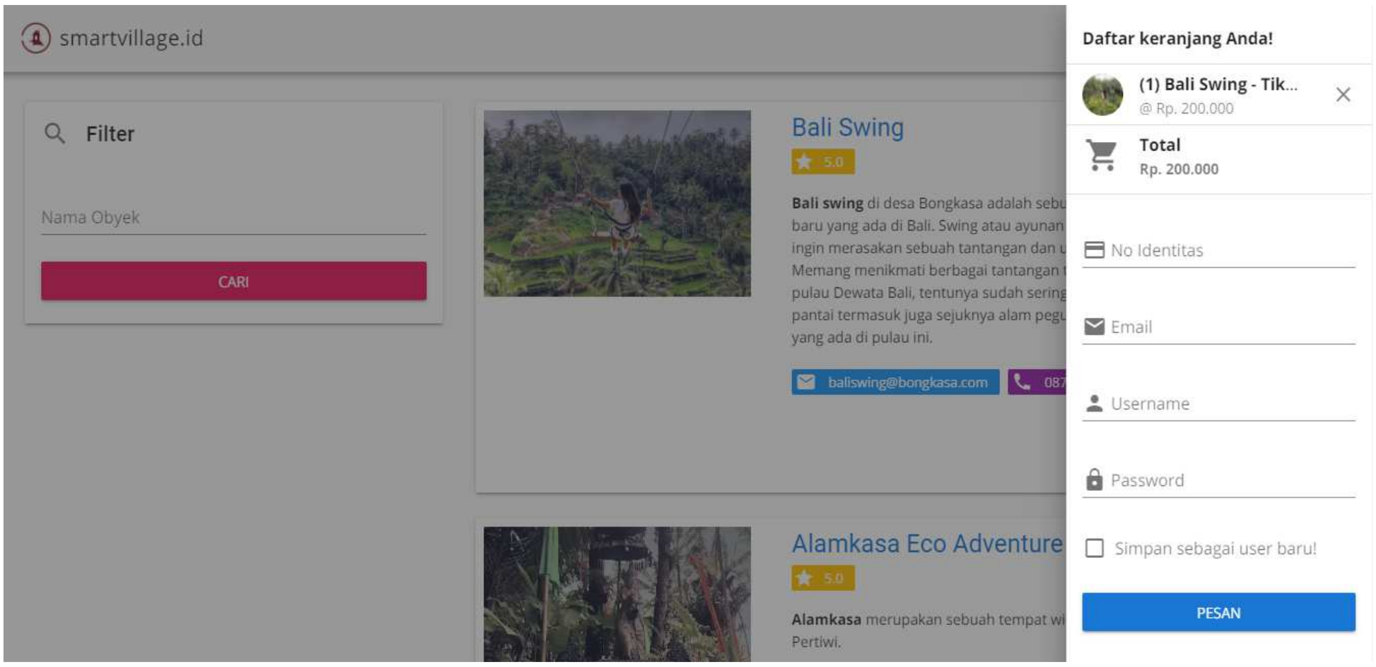
Method	URI	Use Case
GET, POST, PUT	/product	Product /api/v1/product
GET, DELETE	/product/{id}	Product /api/v1/product/{id}
GET, POST, PUT	/customer	Customer /api/v1/customer
GET, DELETE	/customer/{id}	Customer /api/v1/customer/{id}
GET, POST, PUT	/owner	Owner /api/v1/owner
GET, DELETE	/owner/{id}	Owner /api/v1/owner/{id}
GET, POST, PUT	/order	Order /api/v1/order
GET	/order/{id}	Order /api/vi/order/{id}
POST	/mail/register	Email /api/v1/mail/register
POST	/mail/forgetpass	Email /api/v1/mail/forgetpass
POST	/mail/invoice	Email /api/v1/mail/invoice
POST	/mail/payment	Email /api/v1/mail/payment
POST	/mail/voucher	Email /api/v1/mail/voucher
POST	/pay	Payment /api/v1/pay
POST	/pay/notification	Payment /api/v1/pay/notification
POST, PUT	/auth	Auth /api/vi/auth
POST	/auth/reset	Auth /api/auth/reset

E. Event Bus Result

RabbitMQ, as an event bus is used as a communication regulator between services. Microservices can publish messages; then, other services can subscribe to the messages. In RabbitMQ we need to create several events and their producers and consumers.

F. Smart Village Client Result

The smart village client application is built using the Vue.js framework. This application's main functions include displaying products according to search parameters, buying products, registering, logging in, viewing transaction history, managing products, and managing transaction history. The results of the Smart Village Client can be seen in Fig 2.



Bali swing di desa Bongkasa adalah sebu
baru yang ada di Bali. Swing atau ayunan
ingin merasakan sebuah tantangan dan u
Memang menikmati berbagai tantangan
pulau Dewata Bali, tentunya sudah sering
pantai termasuk juga sejuknya alam pegu
yang ada di pulau ini.

Alamkasa merupakan sebuah tempat wi
Pertiwi.

Fig. 2 Smart Village Client Result

IV. Conclusion

This paper describes how to implement a microservices architecture in a smart village application. The implementation process starts with the functional and non-functional requirements analysis phase, microservice analysis and design, coding, testing, and deployment. The resulting architecture consists of four main components: the Client application, API Gateway, Microservices, and the Event Bus. The client application is the user's interface to interact with the smart village application. API Gateway is used to keep microservices from being directly consumed by the public, making the architecture more secure. The microservices section consists of 7 independent microservices to be easier to scale and develop. The event bus or message broker is needed so that communication between services can run asynchronously; this is very effective at increasing the speed of the response to the client because it does not wait for a response from other related services. In addition, the process or transaction continued to run with a message broker even though there is a problematic service. With the scheme in the message broker, the message can be sent until consumer service is available.

Acknowledgment

This research was supported by the Directorate of Research and Community Service, Director General of Development and Research Enhancement, Ministry of Research, Technology, and Higher Education. We thank everyone who contributed to completing this paper in one way or another. Hopefully, this research can be useful.

References

[1] L. De Lauretis, "From monolithic architecture to microservices architecture," *Proc. - 2019 IEEE 30th Int. Symp. Softw. Reliab. Eng. Work. ISSREW 2019*, pp. 93–96, 2019, doi: 10.1109/ISSREW.2019.00050.

[2] C. Richardson, "Pattern: Microservice Architecture," 2018. .

[3] K. Gos and W. Zabierowski, "The Comparison of Microservice and Monolithic Architecture," 2020, doi: 10.1109/MEMSTECH49584.2020.9109514.

[4] F. F. Scattone and K. R. Braghetto, "A microservices architecture for distributed Complex Event Processing in smart cities," *Proc. - 2018 IEEE 37th Int. Symp. Reliab. Distrib. Syst. Work. SRDSW 2018*, pp. 6–9, 2019, doi: 10.1109/SRDSW.2018.00012.

[5] K. Malyuga, O. Perl, A. Slapoguzov, and I. Perl, "Fault Tolerant Central Saga Orchestrator in RESTful Architecture," *Conf. Open Innov. Assoc. Fruct*, vol. 2020-April, pp. 278–283, 2020, doi: 10.23919/FRUCT48808.2020.9087389.

[6] R. Mufrizal and D. Indarti, "Refactoring Arsitektur Microservice Pada Aplikasi Absensi PT. Graha Usaha Teknik," *J. Nas. Teknol. dan Sist. Inf.*, vol. 5, no. 1, pp. 57–68, 2019, doi: 10.25077/teknosi.v5i1.2019.57-68.

[7] I. F. Rozi, R. Ariyanto, A. N. Pramudita, D. R. Yunianto, and I. F. Putra, "Implementation of microservices architecture on certification information system (case study: LSP P1 State Polytechnic of Malang)," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 732, no. 1, pp. 0–6, 2020, doi: 10.1088/1757-899X/732/1/012085.

[8] M. Mena, A. Corral, L. Iribarne, and J. Criado, "A Progressive Web Application Based on Microservices Combining Geospatial Data and the Internet of Things," *IEEE Access*, vol. 7, pp. 104577–104590, 2019, doi: 10.1109/ACCESS.2019.2932196.

[9] Z. Lyu, H. Wei, X. Bai, and C. Lian, "Microservice-Based Architecture for an Energy Management System," *IEEE Syst. J.*, vol. 14, no. 4, pp. 5061–5072, 2020, doi: 10.1109/JSYST.2020.2981095.

[10] Q. Zhou, K. Zheng, L. Hou, J. Xing, and R. Xu, "Design and implementation of open LORa for IoT," *IEEE Access*, vol. 7, pp. 100649–100657, 2019, doi: 10.1109/ACCESS.2019.2930243.

[11] V. Lenarduzzi, F. Lomio, N. Saarimäki, and D. Taibi, "Does migrating a monolithic system to microservices decrease the technical debt?," *Journal of Systems and Software*, vol. 169. 2020, doi: 10.1016/j.jss.2020.110710.

[12] T. Cerny *et al.*, "On Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices," *IEEE Access*, vol. 8, pp. 159449–159470, 2020, doi: 10.1109/ACCESS.2020.3019985.

[13] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *arXiv*, vol. 17, no. 2, pp. 155–158, 2018.

[14] A. Vivas and J. Sanabria, "A Microservice Approach for a Cellular Automata Parallel Programming Environment," *Electron. Notes Theor. Comput. Sci.*, vol. 349, 2020, doi: 10.1016/j.entcs.2020.02.016.

[15] J. Herrera and G. Molto, "Toward Bio-Inspired Auto-Scaling Algorithms: An Elasticity Approach for Container Orchestration Platforms," *IEEE Access*, vol. 8, pp. 52139–52150, 2020, doi: 10.1109/ACCESS.2020.2980852.

[16] A. Smid, R. Wang, and T. Cerny, "Case Study on data communication in microservice architecture," *Proc. 2019 Res. Adapt. Converg. Syst. RACS 2019*, no. June 2020, pp. 261–267, 2019, doi: 10.1145/3338840.3355659.

[17] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation," *IEEE Access*, vol. 10, 2022, doi: 10.1109/ACCESS.2022.3152803.

[18] N. Nikolakis *et al.*, "A microservice architecture for predictive analytics in manufacturing," in *Procedia Manufacturing*, 2020, vol. 51, doi: 10.1016/j.promfg.2020.10.153.

[19] P. Sha, S. Chen, L. Zheng, X. Liu, H. Tang, and Y. Li, "Design and Implement of Microservice System for Edge Computing," in *IFAC-PapersOnLine*, 2020, vol. 53, no. 5, doi: 10.1016/j.ifacol.2021.04.137.

[20] N. C. Coulson, S. Sotiriadis, and N. Bessis, "Adaptive Microservice Scaling for Elastic Applications," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4195–4202, 2020, doi: 10.1109/JIOT.2020.2964405.

[21] "Communication in a microservice architecture," *Microsoft Documentation Website*, 2020.

[22] K. Galbraith, "3 methods for microservice communication," *Logrocket Website*, 2019.

[23] G. Ortiz, J. A. Caravaca, A. Garcia-De-Prado, F. Chavez De La O, and J. Boubeta-Puig, "Real-time context-aware microservice architecture for predictive analytics and smart decision-making," *IEEE Access*, vol. 7, 2019, doi: 10.1109/ACCESS.2019.2960516.

[24] E. Djogic, S. Ribic, and D. Donko, "Monolithic to microservices redesign of event driven integration platform," *2018 41st Int. Conv. Inf. Commun. Technol. Electron. Microelectron. MIPRO 2018 - Proc.*, pp. 1411–1414, 2018, doi: 10.23919/MIPRO.2018.8400254.

[25] Choreography pattern - Azure Architecture Center", *Docs.microsoft.com*, 2020. [Online]. Available: https://docs.microsoft.com/en-us/azure/architecture/patterns/ choreography. [Accessed: 06- Dec- 2020]

[26] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in Microservice Architecture," *Int. J. Adv. Comput. Sci. Appl.*, vol. 9, no. 8, 2018, doi: 10.14569/ijacsa.2018.090804.

[27] P. Valderas, V. Torres, and V. Pelechano, "A microservice composition approach based on the choreography of BPMN fragments," *Inf. Softw. Technol.*, vol. 127, 2020, doi: 10.1016/j.infsof.2020.106370.

[28] F. Dai, Q. Mo, Z. Qiang, B. Huang, W. Kou, and H. Yang, "A Choreography Analysis Approach for Microservice Composition in Cyber-Physical-Social Systems," *IEEE Access*, vol. 8, pp. 53215–53222, 2020, doi: 10.1109/ACCESS.2020.2980891.

[29] E. Lee, P. Tan, Y. Cheng, and X. XU, "Web Service Implementation Methodology," *Organ. …*, no. September, pp. 1–35, 2005.

[30] A. Avritzer *et al.*, "Scalability Assessment of Microservice Architecture Deployment Configurations: A Domain-based Approach Leveraging Operational Profiles and Load Tests," *J. Syst. Softw.*, vol. 165, 2020, doi: 10.1016/j.jss.2020.110564.