Developing Microservice-Based Applications

Subjects: Computer Science, Software Engineering

Contributor: Alen Suljkanović , Branko Milosavljević , Vladimir Inđić , Igor Dejanović

Microservice Architecture (MSA) is a rising trend in software architecture design. Applications based on MSA are distributed applications whose components are microservices. MSA has already been adopted with great success by numerous companies, and a significant number of published papers discuss its advantages. However, there are several important challenges in the adoption of microservices such as finding the right decomposition approach, heterogeneous technology stacks, lack of relevant skills, out-of-date documentation, etc.

microservice architecture   model-driven engineering   software architecture

1. Introduction

Microservice Architecture (MSA) is a rising trend emerging from the enterprise world. This architectural style is defined by Lewis and Fowler [1] as “an approach to developing a single application as a suite of small services, each running in its own process and communicating using lightweight mechanisms, often an HTTP resource API”.

MSAs are particularly suitable for cloud infrastructures as they greatly benefit from the elasticity and rapid provisioning of resources [2], but also for the modernization of legacy systems [3].

According to the survey published by the Eclipse Foundation, in 2018 (2018 Jakarta EE Developer Survey Report—https://jakarta.ee/documents/insights/2018-jakarta-ee-developer-survey.pdf (accessed on 13 October 2021)), about 46% of organizations developed their applications in the form of microservices. A survey performed in 2019 (2019 Jakarta EE Developer Survey Report—https://jakarta.ee/documents/insights/2019-jakarta-ee-developer-survey.pdf (accessed on 13 October 2021)), again by the Eclipse Foundation, shows that MSA is the leading architecture for implementing Java in the cloud.

Although evidence shows an increase of distributed systems developed in the form of MSA, studies presented in [3][4][5] show that it is still problematic to find skilled developers familiar with microservices. Microservice developers need to familiarize themselves with the variety of technologies and tools in a timely manner. Furthermore, supporting polyglot programming may require additional tooling, processes, and knowledge [6]. Being polyglot is an important characteristic of MSA as it gives developers the flexibility to choose the technology stack and languages that work best for their needs. However, introducing many languages and frameworks may actually decrease the overall understandability and maintainability of the system. As shown by Wang et al. [7], some organizations already regulate language diversity by restricting the number of programming languages used in a microservice-based system to a few core languages. While this decreases the problems of understandability and maintainability, a study performed by Baškarada et al. [4] shows that the opportunity to use heterogeneous technology stacks was singled out by most interviewees as one of the most significant drivers for MSA adoption.

Due to the higher complexity of MSA, migration of the monolithic legacy system to MSA can cause high development costs. The empirical study performed by Lenarduzzi et al. [8] shows that the technical debt of microservices grows 90% slower than in the corresponding monolithic legacy system. Lenarduzzi et al. [8] also recommend companies define a set of service templates as a way to ease the development of new microservices, but also not to delay important architectural decision making, as this will cause more effort in the future. Having service templates helps, but developers are still required to manually introduce the necessary changes to create a new microservice or to update the existing microservices to comply with architectural changes.

Documenting MSA properly can also be a challenge. Frequent changes in the architecture often lead to wrong and out-of-date architecture models [9]. In addition, communication relationships between microservices and the specific APIs through which they communicate are often missing from such models [9]. With the highly decentralized development and design of microservices, it becomes challenging to maintain a centralized reference of the architectural design [10][11]. Due to this, the system's architecture deviates from the original design over time.

2. Comparison of MSA with Other Architectural Styles

The terms *architecture* and *architecture description* are introduced by ISO/IEC/IEEE 42010:2011 standard [12]. An *architecture description* expresses an architecture of a system-of-interest [12]. *Architecture* encompasses fundamental concepts or properties of a system in its environment embodied in its elements and relationships and in the principles of its design and evolution [12]. Architecture descriptions are used by software teams to improve communication and cooperation among stakeholders, enabling them to work in a comprehended and coherent manner. An architecture description comprises architecture views and models. Gorski [13] presents the software architecture model 1+5 that encompasses various architectural views for modeling business processes, describing use cases and their realizations, interaction and contract agreements between services, and deployment.

MSA is a *service-based architecture*, just like Service-Oriented Architecture (SOA). Even though MSA and SOA represent very different architectural styles, they share many characteristics. Services are a primary architecture component used to implement and perform business and nonbusiness functionality in both MSA and SOA [14]. Both MSA and SOA are generally distributed architectures and also lend themselves to more loosely coupled and modular applications [14]. Furthermore, the implementation of a service is hidden behind its publicly available API. Due to this, the implementation of a service can be entirely changed without affecting the rest of the system as long as the API changes are backward compatible [14].

Although MSA and SOA both rely on services as the main architecture component, they vary greatly in terms of service characteristics [14]. Differences are shown in service taxonomy (i.e., how services are classified within an architecture), service ownership, and service granularity.

As shown in **Table 1**, microservices have limited service taxonomy. There are only two service types: *functional services* and *infrastructure services*. Functional services implement specific business operations or functions, whereas infrastructure services implement nonfunctional tasks such as authentication, authorization, auditing, logging, and monitoring. In MSA, infrastructure services are not exposed to the outside world and are only available internally to other services [14]. On the other hand, in SOA, there are four types of services. *Business services* are abstract, coarse-grained services that define core business operations. These services are devoid of implementation details and usually only contain information about a service name and expected inputs and outputs. *Enterprise services* are concrete, coarse-grained services that implement the functionality defined by the business services. These services are generalized and shared across the organization [14]. *Enterprise services* can contain business functionality, but usually, they rely on application and infrastructure services. *Application services* are fine-grained services that are bound to a specific application context. They provide functionality not found in the enterprise services. *Infrastructure* services implement the same tasks as in MSA. In addition, there is a significant difference in service ownership as development teams are responsible for full support and development of a service throughout its life cycle (also known as the “you build, you run it” principle) [15]. Microservices are small, fine-grained services (hence “micro”), whereas services in SOA range in size from very small to large enterprise services.

Table 1. Comparison of service characteristics in MSA and SOA.

Architectural Style	Service Taxonomy	Service Ownership
MSA	Functional services	Application development teams
	Infrastructure services	Application development teams
SOA	Business services	Business users.
	Enterprise services	Shared services team or architects.
	Application services	Application development teams.
	Infrastructure services	Application development teams or infrastructure services teams.

MSA and SOA also differ with regard to data sharing and service coordination. In the case of data sharing, MSA promotes a style where microservices share as little data as possible, whereas SOA promotes the diametrically opposed concept of sharing as much data as possible [14]. For this reason, a microservice and its associated data represent a single unit with minimal dependencies, which facilitates maintenance and deployment of the microservice (i.e., microservice can be changed and redeployed without affecting the rest of the system). Multiple services can be composed together as a new service. This process is known as *service composition*. Service composition implies the existence of *service coordination*. For service coordination, MSA focuses on service choreography, whereas SOA relies on both service choreography and service orchestration [14]. The term *service* *orchestration* refers to the coordination of multiple services through a centralized mediator, whereas *service choreography* lacks the mediator [14]. Implementation of service coordination is a non-trivial and error-prone task. However, there are several tools that allow developers to automate this process [16][17]. As a result of this approach, microservices are responsible for interaction with others [18]. Of course, one can still utilize orchestration; however, this is not a typical approach [18]. Due to the mentioned differences, systems built on SOA tend to be slower than microservices and require more time and effort to develop, test, deploy, and maintain [14]. In addition, a service composition should be efficient with minimal execution time and energy consumption. An approach for execution time and energy efficient service composition is presented by Li et al. in [19]. The order of service tasks execution is determined by a *service scheduling* [20]. Resource allocation is another important process that affects the performance. The goal of resource allocation is to select resources for component instances and determine the number of component instances needed to meet performance and reliability requirements [20]. Usually, microservices have more components with less functionality that require fewer resources.

MSA also differs from Monolithic Architecture (MA). In MA, components rely on the sharing of resources of the same machine (memory, databases, or files) and are therefore not independently executable [21]. Monolithic applications are usually internally split into multiple services and/or components, but they are all deployed as a single solution. Unless the application is becoming too big, monolithic applications are easier to develop [22]. However, large monolithic applications suffer from the following problems [15]: they are difficult to maintain and evolve, it is hard to add or update libraries due to *dependency hell*, change in only one module requires rebooting the whole application, etc. Microservices succeed in mitigating these problems and are gaining in popularity due to the following characteristics:

- *Size*: Because microservices implement a limited amount of functionalities, their code bases are small which limits the scope of a bug [15]. The small size also provides benefits in terms of service maintainability and extendability. A small service can be easily modified or rebuilt from scratch with limited resources and in a limited time [23].
- *Independence*: Each microservice in MSA is operationally independent of others and communicates with other microservices through their published interfaces [23]. This has several benefits: (i) microservices are independently deployed, (ii) the *new* version of microservice can co-exist with the *old* version, and the microservices that use the *old* microservice can be gradually modified to use the *new* microservice [15], (iii) changes in one microservice do not require the reboot of the whole system, and (iv) scaling MSA implies deploying or disposing of instances of microservices with respect to their load [24].

3. MSA Design Patterns

A design pattern is a reusable solution for a problem that occurs in a specific context. Richardson [25] proposes a taxonomy and describes design patterns used in MSA. In the rest of the paper, design patterns from the following categories will be considered: *Deployment* patterns, *Communication* patterns, *External API* patterns, *Reliability* patterns, and *Service Discovery* patterns. Each of the previously mentioned categories can contain multiple different design patterns, as shown in **Table 2**.

Table 2. Microservice design patterns. Descriptions of all presented designed patterns are given by Richardson [25].

Purpose	Name	Description
Deployment	Single instance per host	Each service instance will be deployed to a separate host.
	Multiple instances per host	Multiple service instances deployed to a single host.
	Single instance per container	Each service instance will be deployed to a separate container.
	Serverless	Service instances are run by the deployment infrastructure that hides any concept of servers.
Communication	Remote Procedure call	Services use RPC-based protocol for communication.
	Messaging	Services communicate in asynchronous manner, via passing messages.
External API	API gateway	Provides a single entry point for all external clients.
	Back-end for front-end	Provides a single entry point for each client separately.
Reliability	Circuit breaker	Prevents a network or service failure from cascading to other services.
Service discovery	Client-side discovery	Client service obtains the location of the server service.
	Server-side discovery	Client services' send requests to the server service via the router.
	Service registry	Contains service instances and their locations.

Deployment patterns provide a solution to the problem of packaging and deploying microservices. Patterns such as *Single Instance Per Host*, *Multiple Instances Per Host*, *Single Instance Per Container*, and *Serverless* are all deployment patterns. In the case of the *Single Instance Per Host* and the *Multiple Instances Per Host*, a *Host* can be either a physical or a virtual machine (VM). The most popular deployment pattern, both in industry and academia, is the *Single Instance Per Container*, followed by the *Single Instance Per Host*, where a host is a VM [26]. There are several reasons why containers are preferred over VMs [26]: (i) creating and launching container images are often very fast, (ii) the same physical server can hold more containers than VMs due to their size, and (iii) more than one container can use a single operating system, which in turn reduces the overhead of licensing costs compared to the VM.

Microservices must handle requests from external clients, but often they must collaborate to perform a certain task. The role of *Communication* patterns is to provide a solution for a problem of communication between the parts of the system. The most prevalent communication patterns in microservice-based systems are *Remote Procedure*

Call (RPC) and *Messaging*. With RPC, microservices can communicate either in a synchronous or asynchronous manner, whereas with *Messaging* the communication is always asynchronous. According to Aksakalli et al. [26], it is not possible to determine the most popular communication pattern, since the selected pattern can change over time as the system evolves.

How microservices will establish a connection is further described by *Service Discovery* patterns. The *Service Registry* pattern ensures the existence of a service registry, which represents a database that contains the data model of available microservices deployed in the platform [27]. A service registry is a critical part of the system and must be highly available. The responsibility for registering or unregistering a microservice within a service registry can be put either on the microservice itself (*Self-Registration*) pattern or the third party (*Third-Party*) pattern. Similarly, the responsibility for discovering a microservice can be either on the client side (*Client-Side Discover*) pattern or on the server side (*Server-Side Discovery*) pattern.

External API patterns describe exactly how external clients will communicate with microservices. An *API Gateway* provides a single entry point for all external clients. To avoid a single point of failure, multiple instances of an API gateway are usually deployed [26]. Additionally, a separate API gateway can be provided for each kind of client (web app, mobile, etc.). This variation of an *API Gateway* pattern is a *Back-End for Front-End* pattern.

Microservices are *built to fail* [28]. The solution for the problem of preventing the microservice failure to cascade to other microservices is provided by a *Reliability* pattern named *Circuit Breaker*.

4. Existing MSA Frameworks

This section will briefly describe current state-of-the-art tools for describing the architecture of distributed systems based on microservices.

MAGMA (*Maven Archetype for Generating Microservice Architectures*) is a tool that is based on the Maven build management system. It aims at accelerating the development of microservices architectures (MSA) by generating infrastructure code that is: (i) specifically configured for the target application domain; (ii) directly runnable; (iii) extensible with user-defined templates [29].

AjiL is a tool for creating and describing MSAs based on the Eclipse Modeling Framework (EMF) (Eclipse Modeling Framework—https://www.eclipse.org/modeling/emf/ (accessed on 10 January 2021)). AjiL comprises a graphical editor and a template-based code generator. The generator converts existing AjiL diagrams into runnable MSA, based upon Java and the Spring framework [30].

TheArchitect is a rule-based system used for generating serverless microservices [31]. MSAs in TheArchitect are created by providing system requirements into TheArchitect's input wizard. The obtained information is processed by the predefined set of models afterward. These models are subsequently analyzed by the code generator, which produces the application code.

MicroBuilder is a tool for generating REST-based (REST–Representational State Transfer) microservices [32]. It consists of two modules: MicroDSL and MicroGenerator. MicroDSL is a domain-specific language (DSL) used to describe the architecture of the microservices. MicroGenerator uses the MicroDSL specifications to generate Java applications based on the Spring framework.

JHipster (JHipster — https://www.jhipster.tech/ (accessed on 13 July 2022))is a tool for generating, developing, and deploying web applications and MSAs. MSAs in JHipster are modeled by using the JHipster domain language (JDL). Modeling of microservices in JDL can be performed quickly due to its user-friendly syntax. JDL specifications are transformed automatically into runnable Java applications by the JHipster code generator.

LEMMA (Language Ecosystem for Modeling Microservice Architecture) [33] is a set of Eclipse-based modeling languages and model transformations for developing MSA. These languages provide different modeling viewpoints for different roles in a microservice development team. By introducing explicit modeling viewpoints, LEMMA decomposes the system into smaller, more specialized parts. Because of this, each role is presented only with the information relevant for that role. Just like AjiL, LEMMA is also based on the Eclipse ecosystem.

Jolie is a service-oriented programming language [34]. A service, in Jolie, is composed of two parts: behavior and deployment. A behavior part defines the implementation of the service's functionalities, whereas the deployment part defines the necessary information for establishing communication links between services. The Jolie interpreter is implemented in Java, and it comes with a Java API to interact with it.

Silvera is a declarative language developed for the domain of microservice software architecture development [35]. The language is designed in a way that directly implements design patterns related to the domain of MSA. Silvera is a lightweight language. Because of this, it can be used in any text editor. The simple textual notation also enables easier collaboration through version control systems. Silvera supports an arbitrary number of programming languages and their versions. Various implementations of microservices, API gateways, service registries, and message brokers can also be supported easily. This allows developers to use the best tool for the job and also supports experimentation. Silvera uses microservice-tailored metrics to evaluate the architecture of the designed system. Silvera comes with a small number of dependencies and is available at the *Python Package Index* (Python Package Index—https://pypi.org/ (accessed on 18 January 2022)). Results of the Silvera evaluation study show that participants, on average, implemented the task ∼124% faster when using Silvera than when implementing the same task manually. In addition, 50% of the participants had a task completion of 100% when using Silvera, as opposed to 33.33% with the manual approach.

References

1. Sorgalla, J. Ajil: A graphical modeling language for the development of microservice architectures. In Proceedings of the Microservices 2017 Conference, Extended Abstracts, Odense, Denmark, 25–26 October 2017.

2. Perera, K.; Perera, I. A Rule-based System for Automated Generation of Serverless-Microservices Architecture. In Proceedings of the 2018 IEEE International Systems Engineering Symposium (ISSE), Rome, Italy, 1–3 October 2018; IEEE: Piscataway, NJ, USA, 2018; pp. 1–8.
3. Terzić, B.; Dimitrieski, V.; Kordić, S.; Milosavljević, G.; Luković, I. Development and evaluation of MicroBuilder: A Model-Driven tool for the specification of REST Microservice Software Architectures. Enterp. Inf. Syst. 2018, 1–24.
4. Sorgalla, J.; Wizenty, P.; Rademacher, F.; Sachweh, S.; Zündorf, A. Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. SN Comput. Sci. 2021, 2, 1–25.
5. Rademacher, F.; Sachweh, S.; Zündorf, A. Aspect-oriented modeling of technology heterogeneity in microservice architecture. In Proceedings of the 2019 IEEE International Conference on Software Architecture (ICSA), Hamburg, Germany, 25–29 March 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 21–30.
6. Montesi, F.; Guidi, C.; Zavattaro, G. Service-Oriented Programming with Jolie. In Web Services Foundations; Springer: Berlin/Heidelberg, Germany, 2014; pp. 81–107.
7. Wang, Y.; Kadiyala, H.; Rubin, J. Promises and challenges of microservices: An exploratory study. Empir. Softw. Eng. 2021, 26, 1–44.
8. Lenarduzzi, V.; Lomio, F.; Saarimäki, N.; Taibi, D. Does migrating a monolithic system to microservices decrease the technical debt? J. Syst. Softw. 2020, 169, 110710.
9. Kleehaus, M.; Matthes, F. Challenges in Documenting Microservice-Based IT Landscape: A Survey from an Enterprise Architecture Management Perspective. In Proceedings of the 2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC), Paris, France, 28–31 October 2019; IEEE: Piscataway, NJ, USA, 2019; pp. 11–20.
10. Bushong, V.; Abdelfattah, A.S.; Maruf, A.A.; Das, D.; Lehman, A.; Jaroszewski, E.; Coffey, M.; Cerny, T.; Frajtak, K.; Tisnovsky, P.; et al. On Microservice Analysis and Architecture Evolution: A Systematic Mapping Study. Appl. Sci. 2021, 11, 7856.
11. Waseem, M.; Liang, P.; Shahin, M. A systematic mapping study on microservices architecture in devops. J. Syst. Softw. 2020, 170, 110798.
12. ISO/IEC/IEEE. ISO/IEC/IEEE Systems and Software Engineering—Architecture Description; ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000); IEEE: Piscataway, NJ, USA, 2011; pp. 1–46.
13. Górski, T. The 1+5 Architectural Views Model in Designing Blockchain and IT System Integration Solutions. Symmetry 2021, 13, 2000.

14. Richards, M. Microservices vs. Service-Oriented Architecture; O'Reilly Media: Newton, MA, USA, 2015.
15. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, today, and tomorrow. arXiv 2016, arXiv:1606.04036.
16. Autili, M.; Di Salle, A.; Gallo, F.; Pompilio, C.; Tivoli, M. CHOReVOLUTION: Service choreography in practice. Sci. Comput. Program. 2020, 197, 102498.
17. Serhani, M.A.; El-Kassabi, H.T.; Shuaib, K.; Navaz, A.N.; Benatallah, B.; Beheshti, A. Self-adapting cloud services orchestration for fulfilling intensive sensory data-driven IoT workflows. Future Gener. Comput. Syst. 2020, 108, 583–597.
18. Cerny, T.; Donahoo, M.J.; Trnka, M. Contextual understanding of microservice architecture: Current and future directions. ACM SIGAPP Appl. Comput. Rev. 2018, 17, 29–45.
19. Li, J.; Zhong, Y.; Zhu, S.; Hao, Y. Energy-aware service composition in multi-Cloud. J. King Saud-Univ.-Comput. Inf. Sci. 2022, in press.
20. Gorski, T.; Woźniak, A.P. Optimization of business process execution in services architecture: A systematic literature review. IEEE Access 2021, 9, 111833–111852.
21. Bucchiarone, A.; Dragoni, N.; Dustdar, S.; Larsen, S.T.; Mazzara, M. From monolithic to microservices: An experience report from the banking domain. IEEE Softw. 2018, 35, 50–55.
22. Namiot, D.; Sneps-Sneppe, M. On micro-services architecture. Int. J. Open Inf. Technol. 2014, 2, 24–27.
23. Dragoni, N.; Lanese, I.; Larsen, S.T.; Mazzara, M.; Mustafin, R.; Safina, L. Microservices: How to make your application scale. arXiv 2017, arXiv:1702.07149.
24. Gabbrielli, M.; Giallorenzo, S.; Guidi, C.; Mauro, J.; Montesi, F. Self-reconfiguring microservices. In Theory and Practice of Formal Methods; Springer: Berlin/Heidelberg, Germany, 2016; pp. 194–210.
25. Richardson, C. Microservice Patterns; Manning Publications: Shelter Island, NY, USA, 2017.
26. Karabey Aksakalli, I.; Çelik, T.; Can, A.; Tekinerdogan, B. Deployment and communication patterns in microservice architectures: A systematic literature review. J. Syst. Softw. 2021, 180, 111014.
27. Houmani, Z.; Balouek-Thomert, D.; Caron, E.; Parashar, M. Enhancing microservices architectures using data-driven service discovery and QoS guarantees. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, VIC, Australia, 1–14 May 2020; IEEE: Piscataway, NJ, USA, 2020; pp. 290–299.

28. Newman, S. Building Microservices; O'Reilly Media: Newton, MA, USA, 2015.
29. Alen Suljkanović; Branko Milosavljević; Vladimir Inđić; Igor Dejanović; Developing Microservice-Based Applications Using the Silvera Domain-Specific Language. *Applied Sciences* **2022**, *12*, 6679, 10.3390/app12136679.
30. Philip Wizenty; Jonas Sorgalla; Florian Rademacher; Sabine Sachweh; MAGMA. *Proceedings of the 11th European Conference on Software Architecture: Companion Proceedings* **2017**, -, 61-65, 10.1145/3129790.3129821.
31. Jonas Sorgalla; Philip Wizenty; Florian Rademacher; Sabine Sachweh; Albert Zündorf; AjiL. *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings* **2018**, *1*, 61–65, 10.1145/3241403.3241406.
32. Branko Terzić; Vladimir Dimitrieski; Slavica Kordić; Gordana Milosavljević; Ivan Luković; Development and evaluation of MicroBuilder: a Model-Driven tool for the specification of REST Microservice Software Architectures. *Enterprise Information Systems* **2018**, *12*, 1034-1057, 10.1080/17517575.2018.1460766.
33. Jonas Sorgalla; Philip Wizenty; Florian Rademacher; Sabine Sachweh; Albert Zündorf; Applying Model-Driven Engineering to Stimulate the Adoption of DevOps Processes in Small and Medium-Sized Development Organizations. *null* **2021**, -, -.
34. Fabrizio Montesi; Claudio Guidi; Gianluigi Zavattaro; Service-Oriented Programming with Jolie. *null* **2013**, -, 81-107, 10.1007/978-1-4614-7518-7_4.
35. Alen Suljkanović; Branko Milosavljević; Vladimir Inđić; Igor Dejanović; Developing Microservice-Based Applications Using the Silvera Domain-Specific Language. *Applied Sciences* **2022**, *12*, 6679, 10.3390/app12136679.