Towards a Microservice Reference Architecture for Insurance Companies

Arne Koschel
Andreas Hausotter
Robin Buchta
Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: arne.koschel@hs-hannover.de

Alexander Grunewald
Moritz Lange
Pascal Niemann
Hochschule Hannover
University of Applied Sciences & Arts Hannover
Faculty IV, Department of Computer Science
Hannover, Germany
Email: andreas.hausotter@hs-hannover.de

***Abstract*—Microservices are meanwhile an established software engineering vehicle, which more and more companies are examining and adopting for their development work. Naturally, reference architectures based on microservices come into mind as a valuable thing to utilize. Initial results for such architectures are published in generic and in domain-specific form. Missing to the best of our knowledge however, is a domain-specific reference architecture based on microservices, which takes into account specifics of the insurance industry domain. Jointly with partners from the German insurance industry, we take initial steps to fill this gap in the present article. Thus, we aim towards a microservices-based reference software architecture for (at least German) insurance companies. As the main results of this article we thus provide an initial such reference architecture together with a deeper look into two important parts of it.**

Keywords–Microservices; Insurance Industry; Reference Architecture; SOA co-existence

I. Introduction

A current trend in software engineering is to divide software into lightweight, independently deployable components. Each component can be implemented using different technologies because they communicate over standardized interfaces. This approach to structure the system is known as the microservice architectural style [1]. A study from 2019 (see [2]) shows, the microservice architecture style is already established in many industries such as e-commerce. However, this is rarely the case for the insurance services industry.

Our current research is the most recent work of a long standing, ongoing applied research–industry cooperation on service-based systems. This includes cooperative work on traditional Service-Oriented Architecture (SOA), Business Rules and Business Process Management (BRM/BPM), SOA-Quality of Service (SOA-QoS), and microservices (cf, [3]–[6]), between the *Competence Center Information Technology and Management (CC_ITM)* from the University of Applied Sciences and Arts Hanover and two regional, middle-sized German insurance companies. The ultimate goal of our current research is to develop a 'Microservice Reference Architecture for Insurance Companies' jointly with our partner companies. This shall allow to build microservice conformant insurance application systems or at least such system parts.

When developing a reference architecture for our partner companies, several cornerstones and resulting challenges exist frequently in at least the German industry domain. Especially, insurance companies rarely start development 'in the green field', but must integrate and comply with existing application systems. For example, our partners both operate a SOA and additional 3rd party software, such as SAP systems, which both have significantly different characteristics, for example, for testing, release cycles, versioning, administration etc.

Nowadays, our partners would like to get the promised benefits of microservices, such as improved scalability, both technical and organizational through parallel execution and also parallel development, significantly faster release cycles (a few weeks or even days instead of quarters or several months) etc. However, a microservices-based approach to help them must still work well in 'cooperative existence' with their existing systems and SOA services. Thus, improvements or partial replacements of their existing software landscape for particular goals by means of microservices is fine, but a complete migration to the microservices architectural style is not a desired option.

On the one hand, there is a desire to raise the potential of the microservices approach and, on the other hand, to take into account requirements that result from the existing application landscape. This leads to several guidelines, respectively, questions to be answered by a reference architecture, such as, for example, 'Which information from business and technical services shall be provided for architects, developers, operators, etc.?', 'How to integrate with business processes – is service orchestration or choreography (or both) more suitable for microservices?', 'How to co-exist with the given SOA services and their Enterprise Service Bus (ESB)?', 'What about transactions and consistency?', 'Compliance aspects', and more. While initial research on reference architectures with microservices exists in general as well as in some domain-specific variations, we are not aware of such research for the insurance domain in particular.

In this article, we present our initial steps towards a microservices reference architecture for the insurance domain as mentioned above, that complies with the above-named cornerstones and guidelines. In particular, we present our initial logical reference architecture and more logical and technical details about two selected important components from it, namely logging and monitoring.

We organize the remainder of this article as follows: After discussing related work in Section II, we present our initial logical reference architecture in Section III. Afterwards, Section IV shows how details about the logging and monitoring parts of our reference architecture. Section V summarizes the results and draws a conclusion.

II. Related Work

The work related to our research falls into several categories. We will discuss these categories in sequence.

Publications of renowned authors in the area of microservices form the solid base of our research work. Worth mentioning are the basic works of Newman [7] and Fowler and Lewis [1]. The design of our reference architecture benefits from diverse microservices patterns as they are discussed by Krause [8] and Richardson [9].

The contribution of Angelov et al. [10] explains that a reference architecture is successful only if *context*, *objectives* and *design* can be brought into line. Our design refers to 'type 4', which amongst other things means that findings of the implementation of microservices-based application flow into the design of the target reference architecture. Here our previous work – a prototypic implementation of the *Partner Management System* – comes into play [11].

The closest relationship to our research has an article published by Yu et al. [12]. They present a microservices-based reference architecture in the context of *enterprise architecture*. However, this reference architecture aims to be applied to many organizations and is therefore rather generic, while our approach tries to meet the requirements of our industry partners from the insurance sector.

III. Reference Architecture for Microservices

In this section, we will present our logical reference architecture for microservices in the insurance industry (RaMicsV).

RaMicsV defines the setting for the architecture and the design of a microservices-based applications of our industry partners. The application's architecture itself is out of scope, as it heavily depends on the specific functional requirements.

When designing RaMicsV a wide range of restrictions and requirements given by the insurance company's IT management have to be taken in account. With respect to this contribution the most relevant are:

- Enterprise Service Bus (ESB): The ESB as part of the SOA must not be questioned.
- Coexistence: Legacy applications, SOA and microservices-based applications will be operated in parallel for a longer transition period. This means that RaMicsV has to provide approaches for the integration of applications from different architecture paradigms.
- Observability: To observe microservices-based as well as SOA and legacy applications in a comprehensive and consistent manner, a unified monitoring and logging approach has to be designed.

Figure 1 depicts the building blocks of RaMicsV which comprises layers, components, interfaces, and communication relationships. Components of the reference architecture are colored yellow, those that are out of scope are greyed out.

A component may be assigned to one of the following *responsibility areas*:

- **Presentation** includes components for connecting clients and external applications such as SOA services.
- **Business Logic & Data** contains the set of microservices to provide the desired application specific behavior.

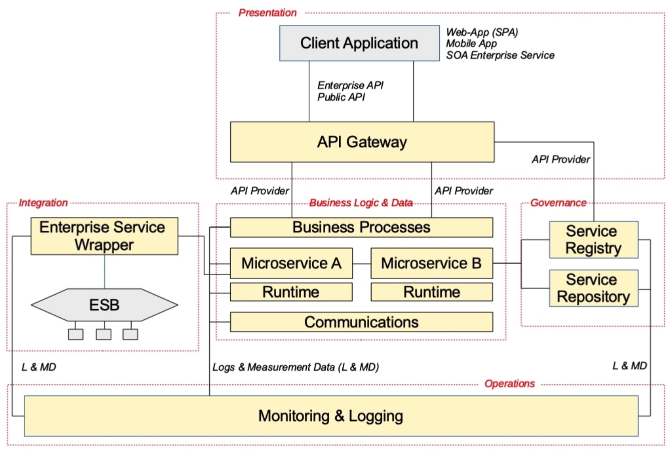

Figure 1. Building Blocks of the Logical Reference Architecture RaMicsV.

- **Governance** consists of components that contribute to meeting the IT governance requirements of our industrial partners.
- **Integration** contains system components to integrate microservices-based applications into the industrial partner's application landscape.
- **Operations** consist of system components to realize a unified monitoring and logging, which encloses all systems of the application landscape.

Components communicate either via HTTP—using a RESTful API, or message-based—using a Message-Oriented Middleware (MOM) or the ESB. The ESB is part of the *integration* responsibility area, which itself contains a message broker (see Figure 1).

In the next section, we will have a detailed look at the *operations* responsibility area in detail.

IV. Logging and Monitoring

In this section we privide details about the logging and monitoring parts of our reference architecture, starting with fundamental concepts, followed by a logical and technical reference architecture.

A. *Introduction to Logging and Monitoring*

In order to provide production-ready software, it is not enough to fulfill only the functional requirements. Figure 2 shows a typical process that is followed when creating production-ready microservices. Observability is assigned to the final quality attributes along with configurability and security. Only if these components have been considered, is it possible to speak of production-ready software [9]. In the following we would like to focus on the aforementioned requirements for the reference architecture, specifically on observability. We are concerned with the objective of how we can create a uniform, fully comprehensive, traceable environment for monitoring and logging.

In his book *Release IT*, Michael T. Nygard does not call this observability but transparency [13]. We do not distinguish between the different terms used for this purpose, but focus on the activities behind the terms, i.e., logging of data and subsequent monitoring. Logging includes the automatic

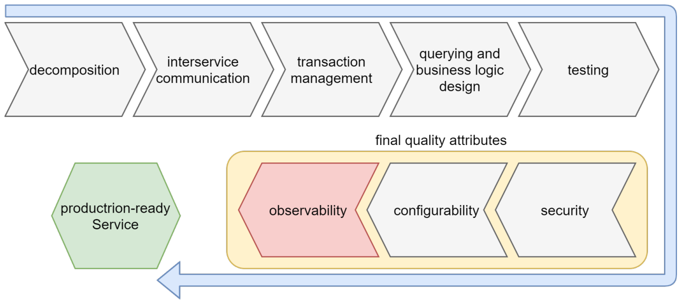

Figure 2. Typical building blocks for the development of a (micro)Service.

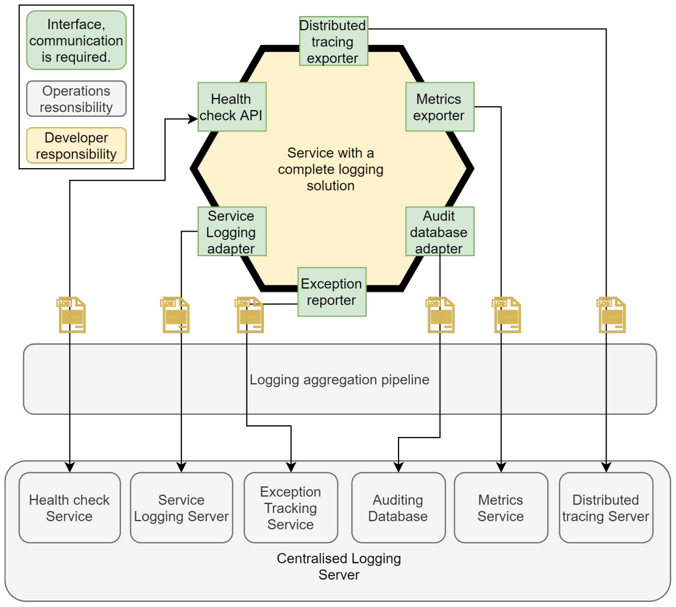

Figure 3. Exemplary implementation of all patterns in combination [9, adopted with modifications].

generation of messages. The generation is based on different triggers. The messages are sent to a location and collected there [14]. Monitoring includes the tools and processes that measure and manage the systems. Furthermore, it includes the process of extracting business value from the underlying data. This data is used to generate added value [15]. We will not go into the basics of monitoring and logging as this would exceed the scope. We will look at what is involved in a fully comprehensive logging of a service.

In the next subsection IV-B, patterns are presented that take account of the provision of data. The aim is to cover all areas of the service. It is important to note that we are focusing on the service and not on the environment in which it is located.

B. Patterns for Logging and Monitoring

In the following, the individual patterns that have been considered in Figure 3 are going to be explained. Figure 3 shows an example implementation of all patterns. In the Figure, the *log aggregation* is placed over all logs that are created so that when the pattern is implemented, all logs are considered.

We first consider the *health check API* pattern. The service receives an endpoint that provides information about its current health status. For example, Spring Boot Actuator can be used for this purpose, which automatically creates a health endpoint that can be adjusted if required [9]. This can be a simple ping for accessibility, but also a smoke test that ensures functionality. Figure 3 shows a bidirectional connection from the health check service to the corresponding API. This is because the endpoint must be queried, and the results obtained. The queries can take place periodically and/or before each invocation of the service. It is important to note that the health check service is a logical component and that requesting the endpoint and collecting the results may very well be two independent components [9].

Next, we are going to look at the *log aggregation* pattern. This is for aggregating all the logs of the multiple instances of a service, to be able to make themselves available together afterwards. This is important because you are interested in all the logs of the service and not just those of one instance. And if a particular instance is of importance, it should be found in the log entry [9].

In Figure 3, the log aggregation goes across all log entries, as the aggregation will refer to all logs regardless of the type. Here it becomes clear that the implementation of this pattern depends strongly on how the service is implemented. If there is only one instance, aggregation is not needed. The same applies to the implementation of the other endpoints. For example, if the audit logs are written directly to a database, no aggregation layer is needed. Again, this is only from a logical point of view.

The *distributed tracing* pattern is particularly important if control flows are of interest, and requests are being passed through multiple services. For this purpose, each request is given a unique ID and it is logged where and how long the request was in the individual services concerned.

The *application metrics* pattern is designed to collect metrics provided by the service. The developer is responsible for ensuring that valuable metrics can be collected, and the operator is responsible for managing them [9].

The *exception tracking* pattern considers the exceptions thrown by a service separately. Exceptions are also special to the service and require special attention. Here, the exceptions are duplicated and handled in detail if necessary. An alert function is optional. An attempt is made to prepare the information so that action can be taken as quickly as possible [9].

The last pattern we look at is the *audit logging* pattern. Here, all user actions are recorded. An audit log contains the identity of the user, the action taken, and the business objects involved [9].

Not all patterns can always be implemented in a meaningful way. In addition, the level of detail in which the individual patterns are implemented varies from application to application. Most of the time, a subset of the patterns presented is the right and sufficient choice to fully observe the service for the purpose it fulfills. If there are multiple instances of a service, log aggregation should be performed across all log types to evaluate the real behavior of the service. In the implementation of the patterns, there are sometimes clear responsibilities of the task areas, as can be seen in Figure 3. However, coordination at the interfaces is also unavoidable. The developers are responsible for creating decent log entries. The operators are responsible for what the users get to see. In the end, it can be said that monitoring and logging is essentially very similar to that of distributed systems.

C. Logical reference architecture

A big building block of the logical reference architecture for microservices (RA4MicsV) is monitoring and logging to add an observability layer to the architecture. One of our main goals was to identify the logical components to implement logging and monitoring in a microservice environment, while maintaining the requirement of coexistence mentioned in Section III. To accomplish this it was important that the logging and monitoring concept for the microservices could be integrated as well as possible into the existing environment, which consists of a combination of monolithic systems and SOA. Figure 4 shows the components of the system itself and the identified, logical components needed to implement an extensive logging and monitoring infrastructure. The key components for the logging and monitoring in this figure will be explained in detail down below.

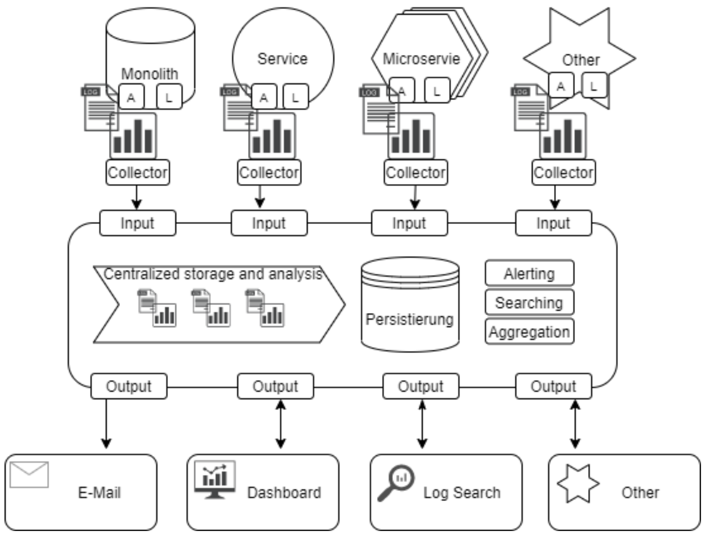

Figure 4. Logical reference architecture of the monitoring and logging environment

- **Agent (A):** Agents are some sort of external process or processes to instrument processes at runtime. There are two major methods agents use to instrument a service directly. The first is some external process or monitoring service that injects code into your service. The second method is through some sort of in-process agent that is imported to the runtime environment of a process and uses a system of user defined rules to trace specific actions [16].
- **Library (L):** Standardized libraries used in services that handle the key components for instrumentation and context propagation through a standardized API. Libraries can support a polyglot heterogeneous application by defining a relatively small API that supports the least-common set of features shared by all of the target languages [16].
- **Collector:** The functions of a collector varies from implementation to implementation, but in common cases the following functionalities are provided by an collector: Translate incoming data into another format for further processing, sampling and compute aggregate statistics about incoming data [16].
- **Centralized storage and analysis:** Responsible for gathering all of the telemetry data, storing it and analyzing it. As mentioned before, the functionality will vary widely based on the implementation [16].

Our model attempts a combination of white box and black box monitoring and logging, since the systems does not only consists of microservices. Apart from that, a whitebox model should generally be considered first when it comes to microservices [16].

D. Technical reference architecture

Since most monitoring and logging components vary in their functionality depending on the specific implementation chosen, this section deals with a sample implementation shown in Figure 5. The model is using a combination of the open source frameworks open telemetry for the instrumentation, elasticsearch as endpoint for the data and Kibana for visualization. The most important components will be explained in detail after a brief introduction of the technologies used.

1) Open Telemetry: Open Telemetry is a nascent project of the Cloud Native Computing Foundation (CNCF) and the result of a merger between the OpenTracing and OpenCensus projects. Its goal is to simplify the telemetry ecosystem by providing a unified set of instrumentation libraries and specifications for observability telemetry [16], [17].

2) Elasticsearch and Kibana: Elasticsearch is a distributed search and analytics engine, which provides near real-time search and analytics for all types of data. Kibana is the inhouse dashboard for visualizing and analyzing data as well as managing, monitoring and securing the elastic stack [18].

- **Open Telemetry Libraries:** In order to receive data, the targets need to be instrumentalized. Open Telemetry provides this mechanism throughout libraries, which support manual (code modified) instrumentation as well as automatic (byte-code) instrumentation [17].
- **Open Telemetry Collector:** The Collector is a vendor-agnostic implementation to receive, process, and export telemetry data. It is the default location instrumentation libraries export telemetry data and it can be deployed in two different ways. First is an agent running with the application or on the same host as the application and second is a gateway running as a standalone service typically per cluster, datacenter or region [17].

In addition, we added a Kafka-Queue and another collector as optional components to the architecture. The queue provides a kind of buffer for the data in case the endpoint is temporarily unable to ingest data or the endpoint is unreachable. The optional collector is deployed as a gateway to provide advanced capabilities such as tail-based sampling. In addition, the Gateway can limit the number of egress points required to send data as well as consolidate API token management [17].

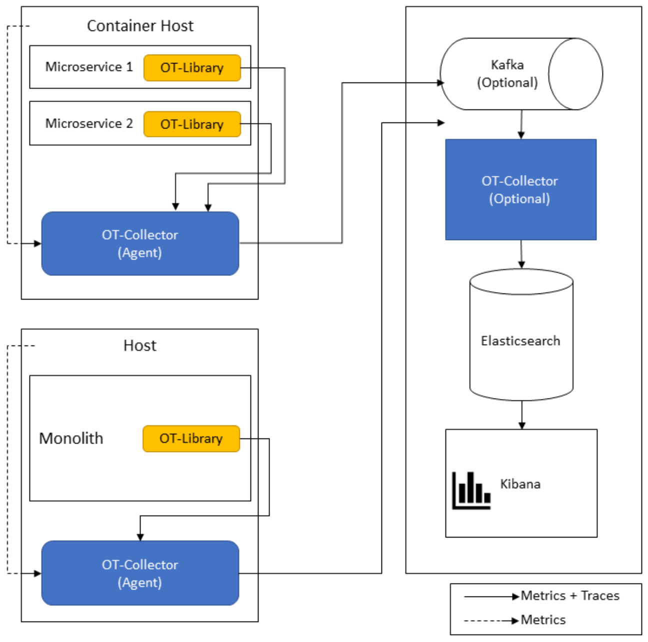

Figure 5. Technical reference architecture of the monitoring and logging environment using open telemetry

V. Conclusion and Future Work

In this article, we presented initial steps towards a reference architecture for microservices, which we are creating jointly with our partners from the insurance industry. The reference architecture aims to build compliant microservices-based applications that meet the specified guidelines and best practices.

We first give an overview of the architecture with its building blocks. We then focus on the operations responsibility area, by presenting conceptual and technical details on logging and monitoring.

The next steps in our research are the design of the business process component and the integration responsibility area. The latter is of particular interest as our partners operate a service-oriented landscape, so it's necessary to identify coexistence pattern to run a SOA and microservices-based applications concurrently.

References

[1] M. Fowler and J. Lewis, "Microservices a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, March 2014, [retrieved: 3, 2021].

[2] H. Knoche and W. Hasselbring, "Drivers and barriers for microservice adoption–a survey among professionals in germany," Enterprise Modelling and Information Systems Architectures (EMISAJ), vol. 14, 2019, p. 10.

[3] A. Hausotter, C. Kleiner, A. Koschel, D. Zhang, and H. Gehrken, "Always stay flexible! wfms-independent business process controlling in soa," in 2011 15th IEEE Intl. Enterprise Distributed Object Computing Conference Workshops. IEEE, 2011, pp. 184–193.

[4] A. Hausotter, A. Koschel, M. Zuch, J. Busch, and J. Seewald, "Components for a SOA with ESB, BPM, and BRM – Decision framework and architectural details," Intl. Journal On Advances in Intelligent Systems, vol. 9, no. 3,4, Dec. 2016, pp. 287–297, [Online]. Available: https://www.thinkmind.org/index.php?view=article&articleid=intsys_v9_n34_2016_6. [retrieved: 3, 2021].

[5] A. Hausotter, A. Koschel, J. Busch, and M. Zuch, "A Flexible QoS Measurement Platform for Service-based Systems," Intl. Journal On Advances in Systems and Measurements, vol. 11, no. 3,4, Dec. 2018, pp. 269–281, [Online]. Available: https://www.thinkmind.org/index.php?view=article\&articleid=sysmea_v11_n34_2018_4. [retrieved: 3, 2021].

[6] A. Koschel, A. Hausotter, M. Lange, and P. Howeihe, "Consistency for Microservices - A Legacy Insurance Core Application Migration Example," in SERVICE COMPUTATION 2019, The Eleventh International Conference on Advanced Service Computing, Venice, Italy, 2019, [Online]. Available: https://thinkmind.org/index.php?view=article&articleid=service_computation_2019_1_10_18001. [retrieved: 3, 2021].

[7] S. Newman, Building microservices: designing fine-grained systems. Sebastopol, California: O'Reilly Media, Inc., 2015.

[8] L. Krause, Microservices: Patterns and Applications: Designing fine-grained services by applying patterns. Lucas Krause, 2015.

[9] C. Richardson, Microservices Patterns: With examples in Java. Shelter Island, New York: Manning Publications, 2018.

[10] S. Angelov, P. Grefen, and D. Greefhorst, "A classification of software reference architectures: Analyzing their success and effectiveness," in 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, IEEE, Ed., 2009.

[11] A. Koschel, A. Hausotter, M. Lange, and S. Gottwald, "Keep it in Sync! Consistency Approaches for Microservices - An Insurance Case Study," in SERVICE COMPUTATION 2020, The Twelfth International Conference on Advanced Service Computing, Nice, France, 2020, [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=service_computation_2020_1_20_10016. [retrieved: 3, 2021].

[12] Y. Yu, H. Silveira, and M. Sundaram, "A microservice based reference architecture model in the context of enterprise architecture," in 2016 IEEE Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC). IEEE, 2016, pp. 1856–1860.

[13] M. Nygard, Release It! Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.

[14] A. Chuvakin, K. Schmidt, and C. Phillips, Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management. Waltham, Massachusetts: Syngress Publishing, 2012.

[15] J. Turnbull, The Art of Monitoring. Turnbull Press, 2014.

[16] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, Distributed Tracing in Practice - Instrumenting, Analyzing, and Debugging Microservices. Sebastopol, California: "O'Reilly Media, Inc.", 2020.

[17] The OpenTelemetry Authors, "Documentation | OpenTelemetry," https://opentelemetry.io/docs/ [retrieved: 3, 2021].

[18] Elastic, "What is Elasticsearch? | Elasticsearch Reference [7.11] | Elastic," https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html, [retrieved: 3, 2021].