Presented to the Interdisciplinary Studies Program:

UNIVERSITY OF OREGON
APPLIED INFORMATION MANAGEMENT

Applied Information Management
and the Graduate School of the
University of Oregon
in partial fulfillment of the
requirement for the degree of
Master of Science

Evaluating the Operational, Performance, and Extensibility Characteristics of SOA, ESB, and Microservices Architectures

CAPSTONE REPORT

William D. Kurth

University of Oregon
Applied Information
Management
Program

Fall 2017

Academic Extension
1277 University of Oregon
Eugene, OR 97403-1277
(800) 824-2714

Approved by

_____

Dr. Kara McFall
Director, AIM Program

Evaluating the Operational, Performance, and Extensibility Characteristics of SOA, ESB, and Microservices Architectures

William D. Kurth

Abstract

SOA represents a newer software architecture that has spawned even newer technologies such as ESB and Microservices. This annotated bibliography examines the operational characteristics of each of these approaches regarding extensibility, performance, scalability, maintainability, and flexibility. It also considers the costs, both in runtime operations as well as for the development, maintainability, and reuse of each. Conclusions on the relative and absolute merits of each technology can be drawn from the selected literature.

Keywords: extensibility, performance, SOA, ESB, Microservices

Table of Contents

List of Tables and Figures

Introduction

Problem

As software systems become larger and more complex, interconnected, and mission critical, the ability to preserve and extend existing solutions becomes an important consideration to organizations (Sadi & Yu, 2014). There is an increasing need for software to allow for the seamless integration of, and interoperation with, other components (Strîmbei, Dospinescu, Strainu, & Nistor, 2015). Software that is *extensible* enables organizations to meet future growth requirements and make further additions and modifications necessary (Strîmbei et al., 2015). In this context**,** extensibility is "the ability to extend a software system with new features and components without loss of functionality or qualities specified as requirements" (Henttonen, Matinlassi, Niemelä, & Kanstrén, 2007, p. 3). Organizations need to extend their existing investments in software systems so that those systems can grow along with the organizations, without sacrificing performance characteristics (Johann, 2016). Flexible systems that can accommodate both unforeseeable growth in data as well as the output of both structured and unstructured data types are needed (Daki, Hannani, Aqqal, Haidine, & Dahbi, 2017).

The effects of poorly written software range from decreasing readability to causing an entire system to be rewritten. In any case, they cause additional costs and are often the source of runtime errors which cost even more (Sneed, 2014, p. 50).

Developing software using bad practices leads to a loss of extensibility (Tom, Aurum, & Vidgen, 2013). "It has been reported that software cost dedicated to maintenance and evolution activities is more than 80% of total software costs. In addition, it is shown that software maintainers spend around 60% of their time in understanding the code. This high cost could

potentially be greatly reduced by providing automatic or semi-automatic solutions to increase their understandability, adaptability and extensibility to avoid bad-practices" (Mansoor, Kessentini, Bechikh, and Deb, K., 2014).

Technical debt is one metaphor that refers to the consequences of poor software development (Tom et al., 2013). Ward Cunningham (1992) introduced the concept of technical debt in 1992, describing how "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" (Tom et al., 2013, p. 1499). Organizations typically go into technical debt to speed the development cycle, at the expense of extensibility. This can often be done to gain some short-term benefit, such as decreased development time or to meet budget constraints (Tom et al., 2013). Since Cunningham's introduction of the term, the suitability of technical debt as a way of explaining the various drivers of increasing costs throughout the life of a software system has been affirmed by the software development community (Tom et al., 2013).

Tom, Aurum, and Vidgen identified technical debt as a critical issue (2013). "The global technical debt bill is estimated by Gartner to be $US 500 billion in 2010 with the potential to double in five years' time" (Tom et al., 2013, p. 1498). Organizations with technical debt face the alternative of refactoring, or extending, existing systems, or the more expensive alternative of completely rewriting them (Johann, 2016). Johann notes that rewriting existing software systems is more costly, complicated, and less reliable than extending an existing system (2016). Maintenance and deployment costs are also higher for newly rewritten systems (Gouigoux & Tamzalit, 2017).

In an interview for *IEEE Software*, Dave Thomas, ACM Distinguished Member identified some of the inherent challenges with software rewrites. (Johann, 2016) points out "The

rewrite turns out to be a lot more complicated than expected. People typically don't really understand the system before they start changing it. In many systems, you don't have a specification, nor do you have tests. Unless you're prepared to develop a substantive body of tests and have the appropriate documentation, it's very difficult to accurately rewrite" (p. 107).

Many of the systems that are still in use were designed using monolithic architectures (Strîmbei, et al., 2015). Also, many enterprises have been struggling for a long time with the complexities and duplication inherent in countless redundant systems across multiple business units. Thus, a single instance of a service, e.g., centrally developed, maintained, tested, and could be shared across an unlimited number of business units is a highly desired target" (Kryvinska et al., 2013, p. 302). "In the last 10 years the market of the solutions dedicated to the interoperability has grown from $3.4 billion (2004) to $11 billion (2008) and over $20 billion in 2015" (Strîmbei et al., 2015, p. 13).

For organizations that choose to rewrite legacy systems or develop brand new systems, it is easier and more cost effective to develop and evolve networked applications by basing them on reusable distributed object computing middleware (Schmidt, 1999). Middleware is software that resides between applications and the underlying operating systems, network protocol stacks, and hardware (Schmidt, 1999). The integration of legacy systems is also typically achieved through the use of middleware (Strîmbei et al., 2015, p. 13).

SOA, ESB, and Microservices are three of the newer software development approaches that have attracted the interest of the computing industry and businesses that rely on having reliable, extensible, high-performance systems (Gouigoux & Tamzalit, 2017) (Salah, Zemerly, Yeun, Al-Qutayri, & Al-Hammadi, 2016). These technologies offer potential improvements in deployment, maintenance, and development costs (Gouigoux & Tamzalit, 2017). Among

software professionals, these technologies are acknowledged as the latest, most promising, and most prevalent technologies for creating extensible, scalable, maintainable system with high performance (Xiao, Wijegunaratne, & Qiang, 2016).

Service Oriented Architecture, or SOA, "evolved to be one of the most successful representations of the client-server architecture with an added business value that provides reusable and loosely coupled services" (Salah, Zemerly, Yeun, Al-Qutayri, & Al-Hammadi, 2016, p. 318). Systems that are designed using SOA, "an architectural style whose goal is to achieve loose coupling among interacting software agents" (Erickson, J., & Siau, K., 2008). Using SOA can help companies keep up with competitors who are often using the same architectural patterns that enable the reuse of software and services that lie at the core of SOA solutions (Kryvinska, Baroková, Auer, Ivanochko, & Strauss, 2013). With SOA, the automation of business processes can be aided by service composition (Psiuk, Bujok, & Zieliński, 2012). "The top drivers for SOA adoption include business flexibility, simplification and speed of integration" (Kryvinska et al., 2013, p. 301).

An ESB architecture is one that is modeled on the well-known and commonplace service buses utilized in computer hardware and network topologies (Exposito & Diop, 2014). It is a specific type of SOA that allows for synchronous and asynchronous interactions between consumers and providers, acting as a message mediator and router between the two (Exposito & Diop, 2014). ESBs promote easy application interoperability and can be configured to be fault tolerant (Exposito & Diop, 2014). Exposito and Diop (2014) identify increased availability, reliability, performance, and scalability as benefits of the use of ESB, as well as easier maintenance and evolution of the resulting software (Exposito & Diop, 2014).

Microservices are “a novel service-based architectural style with a strong focus on highly cohesive, loosely coupled services. A Microservice realizes a distinct architectural capability and exhibits a high degree of independence regarding development and operation” (Rademacher, Sachweh, & Zündorf, 2017, p. 38). “Micro-services” arose due to limitations and challenges with SOA (Strîmbei et al., 2015, p. 13). Microservices architectures aim to help business analysts and enterprise architects develop scalable applications that embody flexibility for new functionalities as businesses develop, such as scenarios in the Internet of Things (IoT) domain (Shadija, Rezai, & Hill, 2017).

The use of SOA, ESB, and Microservices offers promise for those who wish to create systems that are extensible rather than requiring expensive rewrites to accommodate growth (Gouigoux & Tamzalit, 2017). This study aims to provide a summation of the guidance that is available on different methods in designing, building, deploying, maintaining, scaling, and extending new or existing systems using SOA, ESB, and Microservices.

Purpose Statement

The purpose of this annotated bibliography is to present literature that provides guidance and best practices in identifying the newest architectural styles that are best suited for a particular environment and which will result in high-quality, maintainable, reusable, and extensible software systems. This information should empower IT professionals such as software architects, software engineers, systems architects, and data architects to make better and more informed choices about the architectural patterns they choose and how and why to apply these patterns. The study also serves the needs of the Chief Information Officer (CIO), who is responsible for an organization’s information systems and the support of these systems (Colisto, 2012).

Research Questions

Main question. What are the best practices for building software that is extensible and meets performance standards using a Service Oriented Architecture or the related architectural patterns of Enterprise Service Bus and Microservices?

Sub-Question. What are the relative advantages and disadvantages of Service Oriented Architecture, Enterprise Service Bus, and Microservices regarding extensibility and performance?

Audience

This study is for the elucidation and benefit of software architects, software engineers, systems architects and data architects who design new systems and need to extend or integrate older systems. This study will also appeal to the Chief Information Officer (CIO) who must approve plans for new or rewritten systems, as the effort and resources required to scrap legacy systems and replace them with new systems is substantial (Johann, 2016). In addition, the Chief Information Officer, as the senior manager in charge of the information system function in the organization, is uniquely positioned to drive innovation and business growth (Colisto, 2012).

The goal of the study is to produce a useful reference for mid-level to senior programmers and their managers to use when designing extensible systems. This study will inform these IT experts on the architectural patterns; functioning; and benefits in terms of the performance, ease of use, ease of extensibility, and sustainability of all three technologies.

Search Report

Search strategy. Initially, I started my search with the UO Library search and its Advanced search page. I was able to find relevant articles through the UO Library search but needed to find a method that would return results that were more specific and narrowed to my

topic. I started reviewing all the databases listed under the subject search of ‘Computer and Information Sciences’. I went through all seven of the databases listed and found additional, valuable results. One of the libraries listed was Google Scholar, which returned the most useful list of scholarly, peer-reviewed articles from academic or journal sources that were the closest fit to my topic. When reviewing the search results from Google Scholar I found that, although I was typically able to access the article’s information and abstract, I often needed a license to access the full text of the article. For those cases I could access the article through the UO Library to access the full text.

Key terms. Key terms used to return search results included the following:

- Extensibility.
- Performance.
- Comparison.
- Interoperability.
- Integration.
- Enterprise Service Bus.
- ESB.
- Microservices.
- SOA.
- Components.
- Software reuse.
- Integrability.
- Software Integrability.
- Software Integration.

Search engines and databases. The following search engines and databases provided literature pertaining to this topic:

- ACM Digital Library.
- IEEE Computer Science Digital Library.
- IEEE Xplore.
- IEEE Software.
- Computer Database.
- CiteSeer.
- Computer Source.
- Safari Tech Books online.
- Academic Search Premier
- ArXiv.org
- ScienceDirect
- Elsevier
- SpringerLink
- Google Scholar

Journals. The following Journals and publications provided literature pertaining to this subject.

- Journal of Computational Science
- Information Sciences
- Journal of Systems and Software

Documentation Method

References are documented and tracked using three methods. The first method uses Zotero to save articles into a specified folder. Zotero provides the ability to search and sort the saved articles by title, creator, and date. Zotero also provides the ability to create references in an APA format. However, Zotero did not always accurately populate the references in APA format. In some cases, I had to manually format the references Zotero provided.

The second method involved using Microsoft Word and Excel to record the URIs and abstracts of articles. Often this step was necessary when Zotero was unable to return a reference in APA format, or the source material could not immediately be accessed or downloaded.

The third method was to simply download articles of interest. Documents were downloaded in PDF, Microsoft Word, or HTML format. I stored documents in subdirectories to classify and organize the material.

Reference Evaluation Criteria

Each reference was initially evaluated based on how well the title, abstract, and full text match the search criteria used to highlight the problem under study or some aspect of that problem. The Center for Public Issues Education (n.d.) document entitled Evaluating Information Sources provides an outline of the characteristics to consider when evaluating research sources. Key characteristics used in evaluating literature are its authority, timeliness, quality, relevancy, and lack of bias (Center for Public Issues Education, n.d.).

Authority. I established authority by only using sources that ar**e** published in (a) peer-reviewed journals, (b) current books, (c) recognized conference proceedings, and (d) white papers from reputable professional organizations; these publications generally go through strict

editing processes (Mills, n.d.). A reference was determined as authoritative if it is peer-reviewed and if the author has professional credentials.

Lack of bias. I determined lack of bias by the absence of any sales pitch or product bias in the source itself. I rejected literature if the author or article is in the position of selling any products or services. I considered conflicts of interests in the selection of sources, and rejected those sources where the author or publisher had a recognizable conflict of interest.

Quality. The quality of the writing style was an important factor in the evaluation of potential sources. The author had to demonstrate a firm grasp of the subject matter; present the topic in a clear and intelligible manner; use proper punctuation, spelling, and grammar; and cite sources before drawing conclusions. I favored articles that included multiple perspectives over those that put forth a single perspective.

Timeliness. The technologies of study are fairly new within the area of software and systems development, which is a rapidly changing and evolving field. I therefore generally preferred more recent sources over older sources, and selected sources that were published in 2014 or later. One category of exception to this date range is sources that describe foundational techniques, upon which the newer techniques of SOA, ESB, and Microservices are built. Foundational texts that were published as early as 1992 were selected to provide background on the topic.

Relevancy. I initially established relevancy using keywords and various combinations of keywords. My next step was to establish relevancy by reading the source's abstract or introduction to determine how well the paper addresses the problem of study. Finally, I read the entire text to determine how well the paper addresses the subject and problem. I also took the

number of citations of a work, if available, into consideration; the more times a work had been cited, the more relevant and authoritative the source is.

Annotated Bibliography

The following annotated bibliography presents 18 references on three architectural styles for software development: SOA, ESB, and Microservices. These references provide qualitative and quantitative examinations of the relationships between the three styles. The sources provide information on the performance characteristics of each of these architectural styles and their potential extensibility. Historical and foundational materials are provided for context.

Each annotation consists of three elements: (a) the full bibliographic citation, (b) an abstract, and (c) a summary. The summaries present relevant conclusions and inferences to be drawn from the source. They are the observations, measurements, and opinions of the authors of the individual references.

Background and history of SOA, ESB, and Microservices

Erickson, J., & Siau, K. (2008). Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of Database Management*, *19*(3), 42+. Retrieved from http://go.galegroup.com/ps/i.do?p=AONE&sw=w&u=s8492775&v=2.1&it=r&id=GALE%7CA191393402&asid=9009dc7d68d5504adc8d898c367b2d0c

Abstract: Service-oriented architecture (SOA), Web services, and service-oriented computing (SOC) have become the buzz words of the day for many in the business world. It seems that virtually every company has implemented, is in the midst of implementing, or is seriously considering SOA projects, Web services projects, or service-oriented computing. A problem many organizations face when entering the SOA world is that there are nearly as many definitions of SOA as there are organizations adopting it. Further complicating the issue is an

unclear picture of the value added from adopting the SOA or Web services paradigm. This article attempts to shed some light on the definition of SOA and the difficulties of assessing the value of SOA or Web services via return on investment (ROI) or nontraditional approaches, examines the scant body of evidence empirical that exists on the topic of SOA, and highlights potential research directions in the area.

Summary. This article is important to this study because it raises several fundamental questions about the definition of SOA. The authors note the disparity of definitions and underlying technologies that have been used to describe and define a Service Oriented Architecture. The origins of SOA are traced from the advent of web services and earlier; previous SOA technologies include CORBA (Common Object Request Broker Architecture) (Kumar et al., 2015), XML (eXtensible Markup Language) (Curbera et al., 2002), SOAP (Simple Object Access Protocol) (Curbera et al., 2002), and UDDI (Universal Description, Discovery and Integration) (Kumar et al., 2015), amongst others. The authors identify loose coupling between different components in the system as one of the signatures of any SOA definition. Another key aspect of any SOA solution the authors identify is that SOA is often comprised of distributed systems. Operational and developmental cost reductions are also motivating factors driving SOA architectures. An interesting conclusion the authors reach is that an ESB is not necessarily a form of SOA but any SOA will almost always incorporate some form of an ESB.

The authors note that a lot of ESB solutions have become commercialized, and that there is a difference between the general concept of an ESB and a number of vendor-specific ESB solutions. The authors define SOA as an architectural style for separating and modularizing different components that may be integrated to achieve some business goal, as opposed to a specific product or technology. The authors call for a common definition of SOA, and also note

the need for more academic research on the topic of SOA. Specific areas where the authors recommend research into SOA include commercial solutions, the connection with web services, and overall metrics and measurements. The authors point out that, at the time of publication, there was very little academic research done on SOA, its performance and operational characteristics. Only 25 of over 800 articles written on the subject were identified as coming from academic sources. The authors conclude that more research must be done on the relative costs, benefits and drawbacks of SOA and web services as opposed to other architectural patterns.

Henttonen, K., Matinlassi, M., Niemelä, E., & Kanstrén, T. (2007). Integrability and extensibility evaluation from software architectural models – A case study. *The Open Software Engineering Journal*, *1*(1). https://doi.org/DOI:10.2174/1874107X00701010001

Abstract. Software systems are composed of components acquired from different sources, e.g. subcontractors, component providers, and open source software providers. Therefore, integrability is one of the most important qualities in software development. Extensibility is especially important in open source software systems because they evolve according to the needs of the user community and often into a direction not originally foreseen. Integrability evaluation refers to testing if separately developed components work correctly together. Extensibility evaluation focuses on how new features, originated from customers' demands or new emerging technologies, could easily be developed and exploited in systems without losing existing capabilities. The impact of changes to the system also has to be estimated. This can be done by a method called IEE, which enables extensibility and integrability evaluation from software architectural models. The contribution of this paper is to introduce the IEE method and illustrate how it is to be used with a real-world case study. In the case study, we

applied the IEE in evaluating the architecture of an existing open source tool. Evaluation revealed a need to introduce two new extension points to the architecture and also that an integration framework is needed to integrate the tool under evaluation with other supporting tools.

Summary. The authors focus on the importance of extensibility and integrability when designing software architectures. The authors give a comprehensive overview of a software architecture and describe the concept of integrability, and extensibility, as it applies to open source software. Integrability and extensibility are especially important in open source software systems because they, by definition, evolve and change according to the needs of the user and developer communities, which is often in a direction that could not possibly be foreseen as newer versions are developed.

The authors also took part in a long-term research project on the QADA® (Quality Driven Architecture Design and Analysis) methodology; part of their findings identify integrability and extensibility as Quality Attributes.

The paper is useful because it provides one method to measure software extensibility and integrability as an IEE (Extensibility and Integrability Evaluation). The authors describe different ways to achieve integrability using plug-in architectures. The authors provide detailed analyses of the advantages and disadvantages of the strategies, along with quality evaluations of the techniques. Two strategies the authors identify as useful are a plug-in architecture with specific support for new types of SQL clients and a plug-in for support of administrator functions.

The authors also find that support for parallel development of plug-ins is important to achieve integrability. Key findings include the value of creating extensible and integrable software, especially as software development becomes more of a collaborative effort where

several technical and business partners are involved in the process. This value is reflected in the ability to reuse software and thus save on software development costs.

This article is one that is frequently cited in articles on SOA, ESBs, and Microservices and describes a potentially useful tool to measure software complexity.

Hérault, C., Thomas, G., & Fourier, U. J. (2005). Mediation and Enterprise Service Bus: A position paper. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE*. Retrieved November 13, 2017 from: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=B4F99BD6D6CF065700F531E5E1EC1C8E?doi=10.1.1.142.7416&rep=rep1&type=pdf

Abstract. Enterprise Service Buses (ESB) are becoming standard to allow communication between Web Services. Different techniques and tools have been proposed to implement and to deploy mediators within ESBs. It turns out however that current solutions are very technology oriented and beyond the scope of most programmers. In this position paper, we present an approach that clearly separates the specification of the mediation operations and their execution on an ESB. This work is made within the European-funded S4ALL project (Services For All).

Summary. The authors provide a definition of the Mediator pattern, which is a layer of intelligent middleware services, and describe how ESB is a realization of this pattern that provides integration between different software services, clients, and data. They propose that Mediators can be implemented as either a piece of code that intercepts requests, or as a web service. The authors decompose the mediation layer into several subcomponents such as examiners, transformation, and routers. The authors identify mediation as a key strategy for decoupling services across an ESB and as a way to decouple the service implementation from its

bindings. The authors note that there is not a single definition of an ESB architecture, reflecting that these architectures are more closely defined by their commercial product packaging than by a formal definition. They note that ESBs do not provide sufficient software engineering abstractions to give a high-level comprehension of the architecture. The authors provide an example that illustrates the components and functioning of an ESB-based architecture.

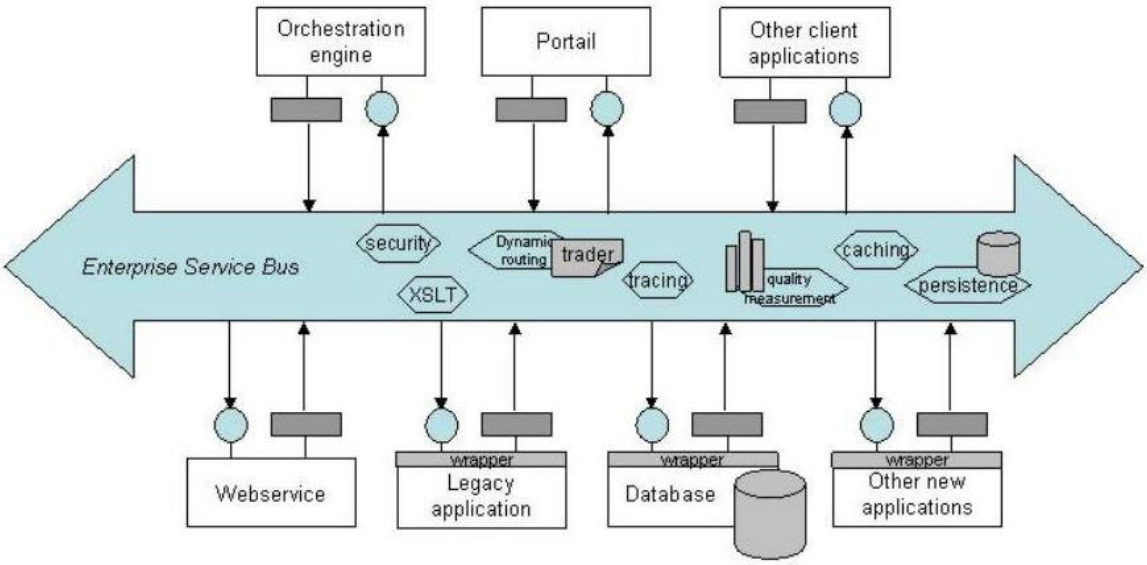

Figure 1. Example Enterprise Service Bus Architecture (Hérault, Thomas, & Fourier, 2005).

In the second half of the paper the authors discuss a European Union funded research project into software services and architectures for services, including J2EE and ESB, named S4ALL project (Services for All). The objectives of this study were to a) study service creation processes, b) specify and implement the best service infrastructure based on the type of services, languages, and containers being used, and c) demonstrate selected application in the telco and industrial fields. One of the findings of this study was that decoupling is an essential part of an ESB reuse and integration solution. The authors identify mediation as one possible path to the desirable behavior of loose coupling. Based upon this research, the authors conclude that a

higher-level abstraction is needed for ESB and when integrating web services; specifically, a higher level of mediation is needed. Finally, the authors identify a need for a platform-independent model and meta language to describe this layer. This paper is important because it provides a good foundation for understanding the inner workings of ESBs.

Johann, S. (2016). Dave Thomas on innovating legacy systems. *IEEE Software*, *33*(2), 105–108. https://doi.org/10.1109/MS.2016.38

Abstract. Host Sven Johann speaks with Dave Thomas, ACM Distinguished Member, entrepreneur, and researcher, about the tradeoffs and constraints facing developers as they work with legacy systems.

Summary. This paper presents a conversation between Sven Johann, a senior consultant at innoQ, and Dave Thomas, ACM Distinguished Member, entrepreneur, and researcher. Thomas discusses issues, surrounding legacy systems, at great length, including the problems with refactoring or rewriting these older systems, which businesses often depend on. He notes that systems can be difficult to change, even those that are well-written. He also points out that rewrites of existing software are very time consuming and require extensive testing tools to ensure the accuracy of the rewrite. Thomas points out that rewriting an existing system in a new language just to have it rewritten in a newer language, or even just to reduce technical debt, is not a good reason for a rewrite. Thomas presents a very balanced viewpoint on some of the problems and most often used solutions associated with legacy systems and refactoring or rewriting them to enable better performance and add capabilities.

Thomas concludes that there is a major tradeoff between writing software well and writing software quickly and notes that often the quicker solutions are less scalable and maintainable. Many of his recommendations to improve quality are for small additions or

incremental re-writes or refactoring, rather than major overhauls of parts or all of a legacy system. Thomas asserts that understanding the pressing business issues and addressing those in a prioritized and measured way is more important than these major overhauls. Thomas favors leaving legacy code as it is and writing new code that adds real value, either through cost reductions or by bringing in new revenue. This paper is important because it provides some foundational considerations in software architecture relating to reuse, rewriting, and scalability.

Lewis, J., & Fowler, M. (2014). Microservices. [Web log]. Retrieved November 17, 2017, from https://martinfowler.com/articles/microservices.html

[Abstract] The term "Microservice Architecture" has sprung up over the last few years to describe a particular way of designing software applications as suites of independently deployable services. While there is no precise definition of this architectural style, there are certain common characteristics around organization around business capability, automated deployment, intelligence in the endpoints, and decentralized control of languages and data.

Summary. This is one of the primary foundational articles about Microservices. It provides some fundamental definitions of microservices; developing a single application as a group of small services, each running in its own process and communicating with lightweight mechanisms such as HTTP. These services are built around business capabilities and are also independently deployed by automated deployment mechanisms. The main body of this work lists what the authors consider to be the main characteristics of Microservices. Those being:

1) Using services as components (rather than libraries) because services are independently deployable. This also has the effect of creating more explicit component interfaces.

2) Writing applications based on business processes rather than on technology layers such as UI, database, or server-side logic.
3) Development teams focused on the product, for the life of the product, as opposed to a more traditional project-based approach where, once the project is finished it is handed off to a maintenance team.
4) Leaving the discovery intelligence in the service itself, as opposed to an ESB architecture, where routing logic is centralized.
5) Governance is decentralized, as the Microservices themselves tend to be.
6) Data management is also decentralized, leaving individual services, that may 'own' some part of the data, assume the responsibility to manage that data.
7) Infrastructure automation through continuous integration and continuous deployment or delivery. This also includes continuous testing as part of the integration and deployment processes.
8) Designing applications so that they can tolerate service failures. This in turn highlights the need for monitoring and logging of services, to know when services fail.
9) Evolutionary design that allows for decomposition as a further tool to enable application developers to control changes in their application without slowing down change.

The authors finish with a balanced view of how important Microservices may be in the future, noting several companies currently employing this technology, including Amazon and Netflix. They also note that they cannot be certain that Microservices is going to be the major software development direction of the future. Instead, they point out several other factors that

may influence how well Microservices are received. Some of these factors include how well existing code might be ‘componentized’, the ensuing interface changes in moving to components, as well as the skill level of any teams embarking on a Microservice project. The authors do end by saying that they do think that Microservices are a worthwhile technology to pursue; being cautiously optimistic about its future. This is one of the most cited, and earliest, articles on Microservices. Their web site also touches on other areas of enterprise software and Microservices.

Shadija, D., Rezai, M., & Hill, R. (2017). Towards an understanding of Microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)* (pp. 1–6). https://doi.org/10.23919/IConAC.2017.8082018

Abstract. Microservices architectures are a departure from traditional Service Oriented Architecture (SOA). Influenced by Domain Driven Design (DDD), microservices architectures aim to help business analysts and enterprise architects develop scalable applications that embody flexibility for new functionalities as businesses develop, such as scenarios in the Internet of Things (IoT) domain. This article compares microservices architecture with SOA and identifies key characteristics that will assist application designers to select the most appropriate approach.

Summary. The authors provide a short history and the benefits of various software trends, starting with structured programming and continuing through Object-Oriented and Component-Oriented programming and finally SOA, ESB, and Microservices. The authors draw a straight line from earlier software engineering efforts, to increase reusability and extensibility, to ESBs and Microservices, the latest methods developed to reach those goals. The authors present several definitions of Microservices and describe how they can be implemented. Most of

the definitions are centered around distributed computing and loose coupling; these approaches are highly cohesive and can be independently deployed.

The authors propose that Microservices are a realization of SOA and may or may not necessarily be small or micro. The authors describe the difference between Microservices, XML web services, and ESBs. They view ESBs as a more heavyweight solution than Microservices or XML web services, and note that ESBs tend to be large and because of their very definition, inflexible. The authors propose that the robust nature of ESBs also makes them inflexible.

The authors identify a key difference between an architecture based on XML Web Services and a Microservice architecture as the decreased reliance on heavyweight middleware. Citing Lewis and Fowler (2014), the authors list several characteristics of Microservices including:

1) Systems are modular and decomposed into loosely coupled software components.
2) Each Microservice is organized around a business capability.
3) Products not projects – the design focus is shifted to business capabilities as opposed to system functional areas, such as user interface (UI) or database.
4) Smart endpoints and dumb pipes – A Microservice endpoint encapsulates all the functionality it needs to operate, as opposed to needing a messaging middleware layer, such as an ESB being needed.
5) Microservices have decentralized data management and governance and are distributed, as opposed to more monolithic architectural styles (Shadija et al., 2017).

The authors note some constraints of Microservices, including that the enterprise/software architect must be able to specify the appropriate bounded contexts for a service, such as defining the domains in which they may run and providing for adequate

scalability. The authors conclude that Microservices are a coarse-grained realization of SOA at the business level, whereas SOAP (Simple Object Access Protocol) services are at an application level. Finally, they state that the two can coexist, noting that legacy applications may use Microservices to facilitate their growth in scope and capabilities. They conclude by noting that Microservices and other coarser grained services, such as those based on SOAP, are both subsets of SOA.

This article is important because it lists several definitions for Microservices and contrasts the strengths and weaknesses of ESB and SOA architectures to those of Microservices. The authors note the complementary natures of these technologies. They also provide a list of Microservices characteristics.

Operational Characteristics of SOA, ESB, and Microservices

Bhadoria, R. S., Chaudhari, N. S., & Tomar, G. S. (2017). The performance metric for Enterprise Service Bus (ESB) in SOA system: Theoretical underpinnings and empirical illustrations for information processing. *Information Systems, 65*(Supplement C), 158–171. https://doi.org/10.1016/j.is.2016.12.005

Abstract. Now days, the businesses are going online and e-Commerce industry is on its boom. In this changing era of development, services are to be Robust, Agile, Accessible and Available to its clients. For secured and guaranteed delivery of services, every big organization is shifting their service delivery model to Enterprise Service Bus (ESB). It promises to set up a strong guideline to build System Oriented Architecture (SOA) system, which leverages multiple services from different application domains. This paper presents an analytical survey of ESB on different parameters influencing the performance of SOA in the present changing scenario and service patterns.

Summary. The authors describe the various services provided by an ESB, including service repositories, policy-based secure messaging, a communication protocol, service discovery, mediation flows, messaging in service bus, message routing, message transformation, message heterogeneity, monitoring, and logging. The authors state that there is not a standard definition of an ESB but note that several authorities have defined it as middleware that also supports multiple integrations and communications with other services. The authors identify high performance, software quality, scalability, and agility as essential, and challenging, qualities of an ESB.

The authors note that there are two broad categories for message routing in literature: web-based service reliability and static routing patterns. Additionally, they describe the four parts of an ESB architecture that comprise the generic functionality of an ESB: a) a mechanism to handle messages, b) a message transformation mechanism, c) a routing mechanism, and d) a message container. The authors present the features and problems of several commercial and open source ESB products, including Mule ESB, Talend ESB, Tibco's Active Matrix ESB, IBM Websphere, Oracle ESB, and Window Azure bus. The authors advise the consumer to be cognizant of the limitations and capabilities of commercial or proprietary ESB solutions.

The authors define ESB as a type of middleware that can deliver services fast and easily, but note that agility and scalability are major challenges in setting up the enterprise software and that better solutions than are currently available from most ESB implementations are needed. The authors also state that integration of new services can be problematic with ESBs.

The authors recommend open source solutions as often being less complex and more flexible than commercial or proprietary solutions. They identify the main advantage of proprietary ESBs as performance in their specific operating environments. The authors conclude

that ESBs are not an entirely new idea, but instead are simply another solution for connecting services in a given enterprise.

The authors assert that routing still remains an issue in ESB architectures and propose a solution that includes pattern based routing and dynamic routing. They describe integration of services as an essential part of any ESB implementation that includes commercial ESB products. The authors identify areas of concern when implementing ESBs as manageability, cost, ongoing developer support and guidance, usability, adaptability, and flexibility. They also note that since most ESB solutions are now either commercial or open source, it is important to understand the level of support provided by any service level agreements (SLAs), the availability of ongoing developer and technical support, and any licensing issues. The authors report that despite the challenges, organizations such as IBM, Oracle, and Microsoft are focusing on ESB technologies. The authors do state that ESB technologies are still the best in-class for developing, deploying, managing, and integrating multiple services on a common platform.

The authors include a diagram of an example ESB architecture.

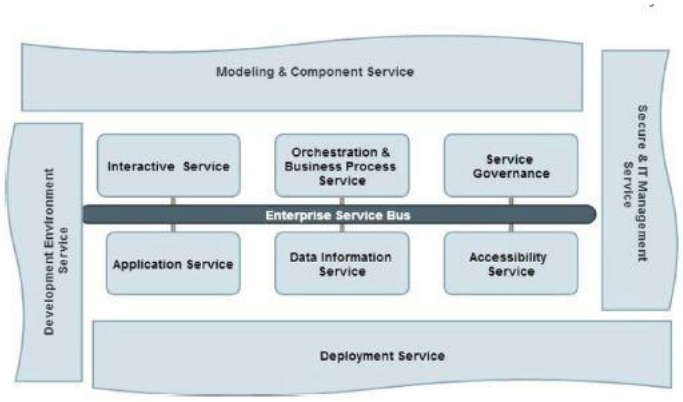

Figure 2. Example ESB Architecture (Bhadoria, Chaudhari, & Tomar, 2017).

This article is important because it gives a detailed view into the structures, costs, and problems commonly associated with ESBs.

Rais, A. A. (2016). Interface-based software integration. *Journal of Systems Integration*, *7*(3), 79–88. https://doi.org/10.20470/jsi.v7i3.261

Abstract. Enterprise architecture frameworks define the goals of enterprise architecture in order to make business processes and IT operations more effective, and to reduce the risk of future investments. These enterprise architecture frameworks offer different architecture development methods that help in building enterprise architecture. In practice, the larger organizations become, the larger their enterprise architecture and IT become. This leads to an increasingly complex system of enterprise architecture development and maintenance. Application software architecture is one type of architecture that, along with business architecture, data architecture and technology architecture, composes enterprise architecture. From the perspective of integration, enterprise architecture can be considered a system of interaction between multiple examples of application software. Therefore, effective software integration is a very important basis for the future success of the enterprise architecture in question. This article will provide interface-based integration practice in order to help simplify the process of building such a software integration system. The main goal of interface-based software integration is to solve problems that may arise with software integration requirements and developing software integration architecture.

Summary. The author examines various ways that software integration can occur. He also lists Standish's "10 laws of CHAOS" (2009) or failure factors in software projects. The most significant lessons from this list are that swift decisions are typically better than long, drawn out analyses; users are both your best friend and worst enemy because a skilled user who can communicate their requirements well, can contribute to a project's success, otherwise they can also contribute to its failure. Iterative development and delivery is preferable over delivery of

a very complex project all at one time. The only way to eat an elephant is one bite at a time – complexity causes confusion and cost.

The author proposes that one way a SOA can be achieved is by the use of an integration mediator such as an Enterprise Service Bus (ESB). He provides an example that uses the façade design pattern from Gamma, et al. (1995), along with service layers, interface layers, and integration layers as a description of interface- or mediator-based software architectures. He ultimately proposes that well-defined software requirements along with a good software architecture can be the solution to many software quality issues. The author also notes that better requirements, and architectural structures, to enforce quality, are an important part of creating software that is more extensible and integrable. The authors found that the foundational deciding factor of whether a project ultimately succeeds or fails is the requirements.

The authors list poor technical quality, poor development processes, and human factors as impediments to software quality and propose proper architecture design as a potential solution. They assert that software integrations can be more successful by using interfaces to define software and clarify business rules. The authors propose that interfaces can help in understanding the requirements, business needs, and architecture of a software solution and can also aid in defining the software's integration patterns and points.

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 318–325). https://doi.org/10.1109/ICITST.2016.7856721

Abstract. Applications developed to fulfil distributed systems needs have been growing rapidly. Major evolutions have happened beginning with basic architecture relying on initiated

request by a client to a processing side referred to as the server. Such architectures were not enough to cope up with the fast ever-increasing number of requests and need to utilize network bandwidth. Mobile agents attempted to overcome such drawbacks but did cope up for so long with the growing technology platforms. Service Oriented Architecture (SOA) then evolved to be one of the most successful representations of the client-server architecture with an added business value that provides reusable and loosely coupled services. SOA did not meet customers and business expectations as it was still relying on monolithic systems. Resilience, scalability, fast software delivery and the use of fewer resources are highly desirable features. Microservices architecture came to fulfil those expectations of system development, yet it comes with many challenges. This paper illustrates how distributed systems evolved from the traditional client-server model to the recently proposed microservices architecture. All architectures are reviewed containing brief definitions, some related work and reasoning of why they had to evolve. A feature comparison of all architectures is also provided.

Summary. Early implementations of distributed systems contributed to the development of client-server and SOA architectures. The authors note that SOA was never agile enough to keep up with changing business and customer requirements due to its monolithic structure. These factors lead to the evolution of software, eventually leading to Microservices. The authors note that Microservices are often compared to SOA, even though, by definition, Microservices are more granular, agile, and reusable then their monolithic SOA counterparts. The authors nevertheless assert that Microservices are a realization of SOA without the ESB. They point out how Microservices and Microservice development are especially well suited for the age of virtual machines, the cloud, and deployment tools such as Docker.

The authors mention REST messaging APIs as the best way for Microservices to be called and to call each other because SOAP and other XML based messaging protocols have too much overhead to make efficient use of the Microservice pattern. The authors list several potential challenges in implementing a Microservice solution such as team coordination; tightly coupling services; increased network traffic; the need for increased security; and determining which containers to use, and how best to use them, for any specific environment. They also list the benefits of Microservices, which include their scalability, reusability, and quick deployment time.

The authors conclude that scalability and loose coupling are best realized using Microservices. The authors suggest that Microservices can provide most of the features that the other architectures can, although they caution that each architecture, including Microservices, has its own set of unique challenges. The authors finish with the observation that each type of architecture has its own appropriate place and that even the application of Microservices should include a healthy consideration for the overhead, security, and performance characteristics in the environment the Microservices are being hosted within.

This article is important because it lists and traces the origins of Microservices, ESBs, and traditional SOA architectures, as well as contrasting their relative strengths and weaknesses.

Strîmbei, C., Dospinescu, O., Strainu, R.-M., & Nistor, A. (2015). Software architectures – Present and visions. *Informatica Economica*, *19*(4), 13–27. https://doi.org/10.12948/issn14531305/19.4.2015.02

Abstract. Nowadays, architectural software systems are increasingly important because they can determine the success of the entire system. In this article, we intend to rigorously analyze the most common types of systems architectures and present a personal opinion about

the specifics of the university architecture. After analyzing monolithic architectures, SOA architecture and those of the micro-based services, we present specific issues and specific criteria for the university software systems. Each type of architecture is rundown and analyzed according to specific academic challenges. During the analysis, we took into account the factors that determine the success of each architecture and also the common causes of failure. At the end of the article, we objectively decide which architecture is best suited to be implemented in the university area.

Summary. The authors trace the evolution of the major software architectural designs, from monolithic architectures, to SOA, to Microservices, lightly touching on ESBs in a SOA context. They define the characteristics of a SOA approach as including a message-based approach for service communication, a Web Services Description Language (WSDL) for describing service interfaces, and a registry of available services. The advantages of a SOA approach include reduced development time and costs, lower maintenance costs, high quality services, lower integration costs, and reduced risk. For Microservices they use Fowler's (2014) description of software services "to describe a particular way of designing software applications as suites of independently deployable services" (para. 2). The authors add that Microservices also encompass Domain Driven Designs, continuous delivery, and on-demand virtualization and follow the single responsibility principle (Haoyu & Haili, 2012).

The authors also cite Lewis and Fowler (2014) in identifying key characteristics of Microservices:

1) Componentization via services.
2) Organized around business capabilities.

3) Products not projects: the development team owns the product during its full life-time.
4) Smart endpoints and dumb pipes: “microservices aim to be as decoupled and as cohesive as possible”; cohesive meaning that they encapsulate their own (complete) business logic, decoupled meaning to communicate through simple messaging or lightweight messaging bus.
5) Decentralized governance: avoid standardization and overhead, use patterns like tolerant reader and consumer driven contracts (service evolution pattern).
6) Decentralized data management.
7) Infrastructure automation covering continuous delivery, continuous integration, automated deployments, automated tests and service versioning management (DevOps).
8) Design for failure: tolerate the failure of services; manage failures: detect and restore faulty services.
9) Evolutionary design: service decomposition (from SOA design principles) (Strîmbei et al., 2015).

The authors include a diagram (Figure 3) that illustrates the general structure of a monolithic versus Microservices architecture.

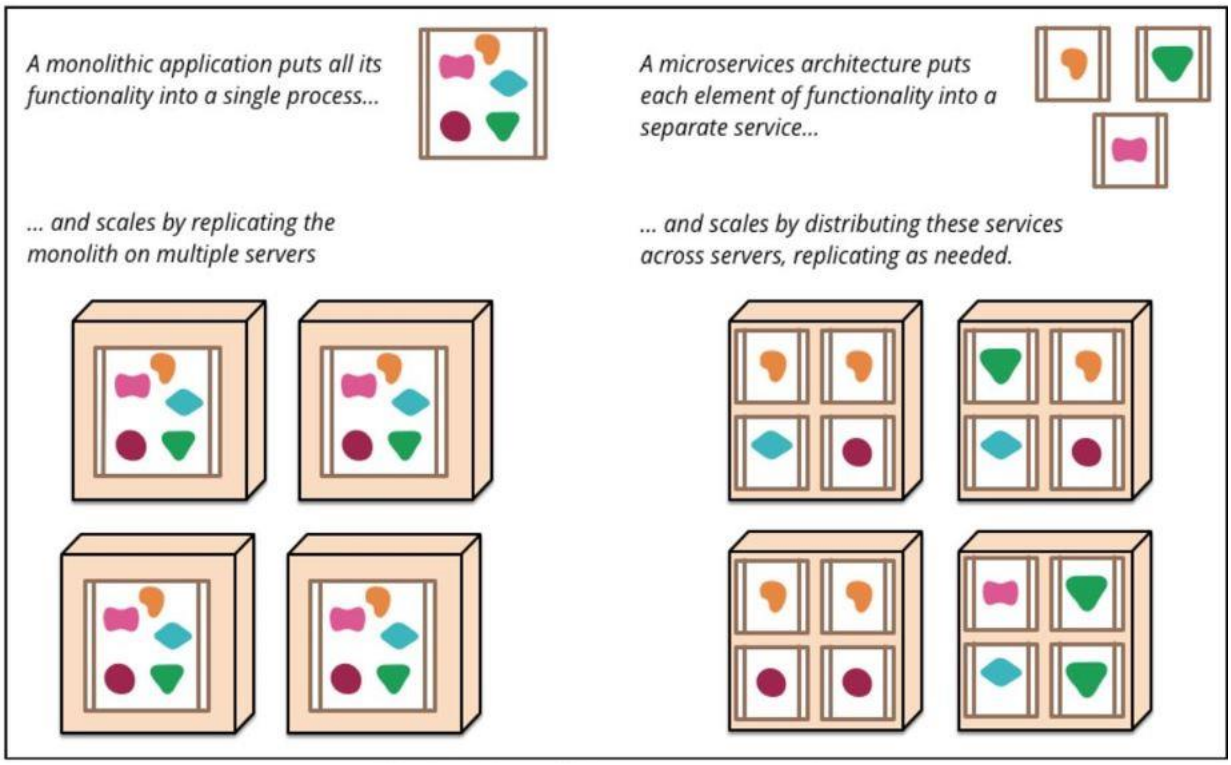

Figure 3. Software architectures, present and visions. (Lewis & Fowler, 2014, as cited by Strîmbei, Dospinescu, Strainu, & Nistor, 2015).

The authors note that some of the characteristics of Microservices that differentiate them from monolithic SOA are that Microservices tend to be asynchronous, relying on a publish and subscribe model, where SOA is more synchronous. The authors also describe Microservices as having faster messaging, whereas SOA and ESBs are smarter but have more dependencies.

One conclusion was to categorize Microservices as being more lightweight and therefore more agile than their heavyweight ESB counterparts. The authors define another key feature of Microservices architectures as the movement of a lot of the complexity from the monolith into the network layer. The authors assert that Microservices are found to be highly autonomous and exhibit extreme flexibility in terms of their functionality and replaceability.

This article is useful for this study because it lists some of the main design requirements for a good Microservice architecture. It also compares features and costs of Microservice architectures to those of ESBs and monolithic SOA architectures.

Thönes, J. (2015). Microservices. *IEEE Software, 32*(1), 116–116.

https://doi.org/10.1109/MS.2015.11

Abstract. In this excerpt from Software Engineering Radio, Johannes Thönes talks with James Lewis, principal consultant at ThoughtWorks, about microservices. They discuss microservices' recent popularity, architectural styles, deployment, size, technical decisions, and consumer-driven contracts. They also compare microservices to service-oriented architecture and wrap up the episode by talking about key figures in the microservice community and standing on the shoulders of giants. The Web extra at http://www.se-radio.net/2014/10/episode-213-james-lewis-on-microservices is an audio recording of Tobias Kaatz speaking with James Lewis, principal consultant at ThoughtWorks, about microservices. They discuss microservices' recent popularity, architectural styles, deployment, size, technical decisions, and consumer-driven contracts. They also compare microservices to service-oriented architecture and wrap up the episode by talking about key figures in the microservice community and standing on the shoulders of giants.

Summary. This article is a transcript of part of a Radio Podcast with James Lewis. He defines Microservices as a small application or service that can be independently deployed, scaled, and tested and that has a single responsibility. Lewis notes that one of the reasons that Microservices have become so prevalent over the past few years is because of all the technical debt that organizations have built up, along with the need to scale and be more efficient in delivering new functionalities. Microservices address these needs. Lewis adds that one of the important considerations is to keep the actual service stack lightweight.

One of the major contrasts Lewis draws between Microservices and ESBs is that while there have been a lot of promises made by ESBs vendors, he has never seen an ESB actually live up to those promises or even succeed. Lewis identifies one of the vulnerabilities of ESBs as their requirement that all the logic, routing, and data transformation are kept in one place. He describes ESB architectures as looking nice on paper, with straight lines drawn to one central locus of control, but that in reality, ESB architectures are still incoherent and spaghetti like. He also advises that adding more services in an ESB architecture results in more integration issues.

Lewis notes that Microservices move a lot of the logic into the network layer. Microservices use practices from the Domain Driven Design community, and also offer better operational automation of deployment and integration (DevOps). Lewis sees Microservices as being driven by better practices than those of monolithic SOA or ESB architectures, and therefore as being preferable. Lewis concludes that Microservices are the future of cloud- and service-based computing architectures. The value in this article is the frank discussion, from an experiential point of view, of the real-world viability of ESB, SOA, and Microservices architectures.

Ueda, T., Nakaike, T., & Ohara, M. (2016). Workload characterization for Microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 1–10). https://doi.org/10.1109/IISWC.2016.7581269

Abstract. The microservice architecture is a new framework to construct a Web service as a collection of small services that communicate with each other. It is becoming increasingly popular because it can accelerate agile software development, deployment, and operation practices. As a result, cloud service providers are expected to host an increasing number of microservices that can generate significant resource pressure on the cloud infrastructure. We want to understand the characteristics of microservice workloads to design an infrastructure optimized for microservices. In this paper, we used Acme Air, an open-source benchmark for

Web services, and analyzed the behavior of two versions of the benchmark, microservice and monolithic, for two widely used language runtimes, Node.js and Java. We observed a significant overhead due to the microservice architecture; the performance of the microservice version can be 79.2% lower than the monolithic version on the same hardware configuration. On Node.js, the microservice version consumed 4.22 times more time in the libraries of Node.js than the monolithic version to process one user request. On Java, the microservice version also consumed more time in the application server than the monolithic version. We explain these performance differences from both hardware and software perspectives. We discuss the network virtualization in Docker, an infrastructure for microservices that has nonnegligible impact on performance. These findings give clues to develop optimization techniques in a language runtime and hardware for microservice workloads.

Summary. The authors observe that the popularity of Microservices is largely because of the agile methodological roots and the coincidental rise of DevOps. Drawing off Lewis and Fowler (2014), who initially proposed the Microservice architectures, the authors bring to light several of the architectural constraints of Microservices. One of these constraints is that Microservices running in containers can be expected to generate more pressure on computer systems than traditional monolithic services running as native processes do. The authors examine the performance characteristics of Microservices, particularly in a cloud environment, using the Acme Aire open source benchmark for web services across several statistically differentiated treatments of Microservices performance. They also use Docker as the container for these Microservices in these experiments and Docker bridge for virtualized networks. The authors measured performance across the following treatments:

a) Throughput measurements and ratios for monolithic versus Microservices architectures.
b) Scalability comparisons between monolithic and Microservices.
c) Path length, or number of nodes travelled, to reach a specific service for each architecture.
d) CPU cycles executed for each service executed.

The authors found that Microservices: (a) had about a third of the throughput of traditional monolithic services, (b) were about even to monolithic services in terms of scalability, (c) had path lengths per call that were about three times that of traditional monolithic architectures, and d) only slightly higher CPU Cycles than traditional monolithic architectures. They found that the performance of Microservices can be 79.2% less than that of a monolithic architecture on the same hardware. For the Node.JS implementation, 4.22 times the resources were consumed as were consumed for the monolithic service. They also found that bridge overhead, using Docker, causes a 33.8% degradation of throughput. Node.JS Microservices, using Docker and its associated bridge, used 30.8% more cycles per instruction than the same Node.JS application written as a monolithic service. Tests were conducted on 1, 4, and 16 core servers; in most cases, the ratio of monolithic response, gateway, cycles per instruction, and response time remained the same.

Although the authors do acknowledge that the agile and DevOps methodologies associated with Microservices can accelerate development, they caution developers to be aware of possible performance considerations caused by the underlying architecture. As the number of Microservices deployed increases on a given platform, they expect these performance issues will become more prevalent.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (pp. 583–590). https://doi.org/10.1109/ColumbianCC.2015.7333476

Abstract. Cloud computing provides new opportunities to deploy scalable application in an efficient way, allowing enterprise applications to dynamically adjust their computing resources on demand. In this paper, we analyze and test the microservice architecture pattern, used during the last years by large Internet companies like Amazon, Netflix and LinkedIn to deploy large applications in the cloud as a set of small services that can be developed, tested, deployed, scaled, operated and upgraded independently, allowing these companies to gain agility, reduce complexity and scale their applications in the cloud in a more efficient way. We present a case study where an enterprise application was developed and deployed in the cloud using a monolithic approach and a microservice architecture using the Play web framework. We show the results of performance tests executed on both applications, and we describe the benefits and challenges that existing enterprises can get and face when they implement microservices in their applications.

Summary. The authors approach monolithic SOA, ESBs, and Microservices from a cloud computing perspective. For the purposes of this study they worked with an unnamed commercial SaaS company and observed and assisted in the development of their service-based application. Their platform was AWS EC2. They used a pair of programs sending messages to each one of a set of monolithic- and Microservices-based endpoints to simulate workloads. JMeter was used to execute, regulate, and measure the response time for each of these services.

Services were developed using Java, Scala, the Play2 web framework, and Postgresql. Two services were developed: one that ran on an EC2 instance type c4.xlarge and was more heavyweight, and the other that ran on an EC2 instance type m3.medium and was more lightweight. The results of their measurements show that for the larger platform configuration, Microservices have a slightly higher response time than their monolithic counterparts. Average response times for the larger services were measured at 3229 milliseconds for the Microservice implementation as opposed to 2837 milliseconds for the monolithic architecture. However, for the medium-sized configuration, Microservices outperform the monolithic architecture. In this case the Microservice implementation had a response time of 210 milliseconds as opposed to 280 milliseconds for the monolithic solution; these results were for average response time.
The authors ran other tests to get the average response time and the 90% line response time (the value below which 90% of the requests fall), with similar results. The authors also note that since Microservices are more granular, and therefore take up fewer resources, they therefore cost less operationally than monolithic architectures. In fact, the authors found Microservices to cost 17% less in infrastructure costs.

The authors assert that SOA implementations, in general, can be expensive, complex and time consuming. The authors also note that even though ESB products are generally designed to handle hundreds or even thousands of users, they do not scale well for Internet applications and often become constraining factors in terms of performance, with high levels of latency. They also note that ESBs are complex and time-consuming to configure, especially in a cloud or other large-scale environment, largely due to problems with scalability, and that they represent a centralized control point for messaging, routing, and configuration. The authors identify ESBs as a single point of failure. They assert that Microservices are a way to bypass a lot of these issues,

as Microservices are more agile; easier to replace, update, and manage than ESBs; tightly associated with business processes; and easier to scale and deploy.

Ultimately, the authors concede that a monolithic architecture may be more appropriate for small applications with a limited number of users but note that even these applications may eventually migrate to becoming a Microservice architecture. Finally, the authors conclude that further investigation in terms of costs; scaling; extensibility; fault tolerances; and the use of various containers such as Docker, Amazon EC2 Container Service, and AWS Lambda is needed. The authors also point to further investigation into how existing SOA and ESB solutions can be migrated to more Microservices-based architectures and environments. This paper is useful because it contrasts all three architectures: monolithic SOA, ESB, and Microservices. It reports the results of some specific runtime comparisons and considers the relative agility, extensibility, scalability, and manageability of these architectural patterns.

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., … Lang, M. (2016). Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (pp. 179–182). https://doi.org/10.1109/CCGrid.2016.37

Abstract. Large Internet companies like Amazon, Netflix, and LinkedIn are using the microservice architecture pattern to deploy large applications in the cloud as a set of small services that can be developed, tested, deployed, scaled, operated and upgraded independently. However, aside from gaining agility, independent development, and scalability, infrastructure costs are a major concern for companies adopting this pattern. This paper presents a cost comparison of a web application developed and deployed using the same scalable scenarios with

three different approaches: 1) a monolithic architecture, 2) a microservice architecture operated by the cloud customer, and 3) a microservice architecture operated by the cloud provider. Test results show that microservices can help reduce infrastructure costs in comparison to standard monolithic architectures. Moreover, the use of services specifically designed to deploy and scale microservices reduces infrastructure costs by 70% or more. Lastly, we also describe the challenges we faced while implementing and deploying microservice applications.

Summary. The authors conduct a follow-on study to an earlier study that examined cost and performance comparisons of web applications implemented using a Monolithic SOA approach, a Microservice approach operated by the cloud customer, and a Microservice approach hosted by a cloud provider. This study is concentrated on contrasting monolithic and Microservices architectures. The authors also shift from deployment on a SaaS platform to IaaS and PaaS platforms. The authors note that companies such as Netflix, Amazon, and LinkedIn are moving to Microservices architectures, and one of the first issues they address is that of deployment tools and DevOps automation tools such as Docker, Chef, and Puppet. One of these DevOps tools, AWS's lambda, becomes one of the control variables for measuring and comparing performance between different configurations of monolithic and Microservices using either no deployment, containerization tools, or AWS lambda; AWS is unique because it offers a per-request cost structure.

The study used two different services, each having distinctly different response times and CPU load structures. One was fairly heavyweight in terms of its CPU usage and response time while the other was lightweight with a faster response time. In this environment, a gateway service was also necessary to use AWS lambda. JMeter was used to run and measure the simulated calls to each service in each environment. The results showed that under normal load

conditions, the Microservices/AWS lambda architecture could handle more requests per minute than the monolithic architecture; 450 requests per minute were handled by the Microservice architected service as opposed to 420 and 350 requests per minute for JAX/RS and Play/Java written services, respectively. The measured costs to run the Microservices were 9.5% (JAX/RS) to 13.42% (Play/Java) less than that of a monolithic structure. During peak loads the response time of Microservices without any containerization jumped to over 3 times that of monolithic services.

The authors found that costs for Microservices were slightly less than the costs of monolithic architectures. The monthly cost of the Microservice architecture was determined to be $390.96 USD in contrast to $403.20 USD for the monolithic architecture. Furthermore, the Microservice architecture supported more requests per minute in the three defined scenarios.

Average and especially peak response times for Microservices were found to be higher though. This result changed when measuring response times of Microservices used in conjunction with AWS lambda. For Microservices with AWS lambda the performance times were better than under any of the other three scenarios. The authors note that costs are lower using Microservices in conjunction with AWS lambda. They also find that a Microservice implemented with Play can reduce costs by between 9.5% and 13.42%. Using AWS lambda can help reduce costs by between 50.43% and 57.01% for different test scenarios used, in contrast to a simple Microservice architecture. Cost reductions can also be between 55.14% and 62.78% of the costs per scenario as compared to a monolithic architecture implemented in Jax-RS, and between 70.23% and 77.08% for the Play implementation.

The authors' final conclusions are that the increased agility, slightly better performance, lower costs, and finer granularization of services should be balanced with the cost required for

companies to adopt new practices, processes, technical challenges, and methodologies in developing Microservices. Other issues that should be addressed are concerns about the number of Microservices actually needed, fault tolerance, and which deployment tools to use. This study is especially important in its presentation of actual runtime performance and cost comparisons between a monolithic SOA architecture and a Microservices architecture.

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development*, *32*. https://doi.org/10.1007/s00450-016-0337-0

Abstract. Some microservices proponents claim that microservices form a new architectural style; in contrast, advocates of service-oriented architecture (SOA) argue that microservices merely are an implementation approach to SOA. This overview and vision paper first reviews popular introductions to microservices to identify microservices tenets. It then compares two microservices definitions and contrasts them with SOA principles and patterns. This analysis confirms that microservices indeed can be seen as a development and deployment-level variant of SOA; such microservices implementations have the potential to overcome the deficiencies of earlier approaches to SOA realizations by employing modern software engineering paradigms and Web technologies such as domain-driven design, RESTful HTTP, IDEAL cloud application architectures, polyglot persistence, lightweight containers, a continuous DevOps approach to service delivery, and comprehensive but lean fault management. However, these paradigms and technologies also cause a number of additional design choices to be made and create new options for many “distribution classics” type of architectural decisions. As a result, the cognitive load for (micro-)services architects increases, as well as the design, testing and maintenance efforts that are required to benefit from an adoption of microservices. To initiate and frame the buildup of architectural knowledge supporting microservices projects,

this paper compiles related practitioner questions; it also derives research topics from these questions. The paper concludes with a summarizing position statement: microservices constitute one particular implementation approach to SOA (service development and deployment).

Summary. The author's main tenet is that Microservices are just a specialized form of a SOA architecture. The author asserts that nothing in a Microservice architecture violates SOA, or does not apply some principle from SOA. The author notes that case studies from other literature reviews support the idea that Microservices draw from some of the newest and best practices in software design. These best practices include a) domain-driven design and test-driven development methodologies, b) continuous deployment into lightweight containers, and c) lean approaches to software and systems management. He notes that decentralization and failure isolation are also found in Microservice architectures.

The author traces the origins of Microservices from the Agile development methodology community and also cites Lewis and Fowler (2014) in generating a list of seven key tenets of Microservices:

a) Fine-grained interfaces to single-responsibility units that encapsulate data and processing logic, typically exposed via RESTful HTTP resources or asynchronous message queues.
b) Business-driven development practices and pattern languages such as domain-driven design.
c) Cloud-centric application development practices.
d) Multiple computing paradigms, such as functional and imperative, and storage paradigms, such as relational databases and several types of NoSQL stores.
e) Lightweight containers, such as Docker, are used to deploy services.

f) Decentralized and continuous delivery.

g) DevOps, Lean, but holistic and largely automated approaches to configuration, performance, and fault management.

Zimmerman notes that the enterprise service bus and its commercial incarnations and products have been criticized by members of the Microservices community as overly heavyweight, inflexible and unmanageable. He does acknowledge the integration between ESB and Microservices capabilities as necessary, arguing that the message routing and transformation patterns from the world of ESB needs to be supported and somehow adapted to fit into the Microservices world.

The author concludes that Microservices are one particular implementation, deployment, and development methodology of a SOA; one that is complementary to traditional SOA. The author finds that SOA is here to stay and that Microservices realizations need to combine SOA principles and patterns with modern software engineering practices to be successful.

This work is valuable in that it provides a list of seven tenets of Microservices and compares Microservices to ESB and SOA architectures, noting similarities and differences.

Best practices in deploying SOA, ESB, and Microservices

Baude, F., Filali, I., Huet, F., Legrand, V., Mathias, E., Merle, P., … Lorre, J.-P. (2010). ESB federation for large-scale SOA. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (pp. 2459–2466). New York, NY: ACM. https://doi.org/10.1145/1774088.1774597

Abstract. Embracing service-oriented architectures in the context of large systems, such as the Web, rises a set of new and challenging issues: increased size and load in terms of users and services, distribution, and dynamicity. A top-down federation of service infrastructure

support that we name “service cloud” and that is capable of growing to the scale of the Internet, is seen as a promising response to such new challenges. In this paper, we define the service cloud concept, its promises and the requirements in terms of architecture and the corresponding middleware. We present some preliminary proofs of concept through the integration of a JBI-compliant enterprise service bus, extended to our needs, and a scalable semantic space infrastructure, both relying on an established grid middleware environment. The new approach offers service consumers and providers a fully transparent, distributed and federated means to access, compose and deploy services on the Internet. Technically, our contribution advances core service bus technology towards the service cloud by scaling the registries and message routers to the level of federations via a hierarchical approach, and by incorporating the communication and coordination facilities offered by a global semantic space.

Summary. The authors recognize that there is an eminent need for complementary communication and coordination means to enable traditional ESB technologies to scale up to internet-scale systems and applications. The authors propose, discuss, and report on the results from a study that involved building a prototype of a federation of distributed service busses (ESBs) for internet-sized and SaaS applications. The main goal of the study was to add more scalability to an ESB solution in those specific application environments, particularly where the solution is likely to grow over time. Key to their technologies were those of a technical registry, leading to a distributed registry architecture, and the capability to route messages within a federation. Federations enable the cooperation between different ESBs or distributed ESBs (DSBs).

The authors also make use of what they term to be semantic spaces, which they define as a fusion of tuple space computing, borrowed from parallel processing; blackboard-style problem

solving, borrowed from artificial intelligence; and semantic technologies into a distributed (semantic) data management platform. They base their prototypes on the open-source, ObjectWeb2 PEtALS ESB, which already supports distributed ESBs. They use this in combination with open source OW2 ProActive Grid technology, which enables the authors to create the portability, distributability, and scalability of the federation of the PEtALS they use.

Traditional ESB architectures rely on older client-server based communication methods. What the authors accomplished in their study is to create more powerful communication patterns (e.g., publish-subscribe, event-driven, and semantic-oriented) that allow further decoupling of the communicating entities in terms of time, processing flow, and data schema. In doing so they go beyond traditional ESB architectures and their limitations, creating more efficiency, transparency, flexibility, and scalability.

The authors conclude that building a system that includes integrated support for semantics and event-based communication mechanisms serves as a basis for shared data management and collaborative activities. The authors have extended the traditional ESB to be part of a federation that can scale to web-sized publishing and reading. The integrated support for semantics allows for direct links to reasoning or mediation techniques.

This paper is important to this study because it outlines a method to overcome some of the traditional shortcomings of ESBs as their application domains grow larger and scalability and performance become more problematic as a result.

Gouigoux, J. P., & Tamzalit, D. (2017). From monolith to Microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 62–65). https://doi.org/10.1109/ICSAW.2017.35

Abstract. MGDIS SA is a software editing company that underwent a major strategic and technical change during the past three years, investing 17 300 man-days [sic] rewriting its core business software from monolithic architecture to a Web Oriented Architecture using microservices. The paper presents technical lessons learned during and from this migration by addressing three crucial questions for a successful context-adapted migration towards a Web Oriented Architecture: how to determine (i) the most suitable granularity of micro-services, (ii) the most appropriate deployment and (iii) the most efficient orchestration?

Summary. The authors present lessons learned from an implementation of Microservices at MGDIS SA, a French software vendor of applications that target public collectivities. The authors assert that most SOA implementations have not lived up to their hype, mainly because services are not reused often enough to make it worth the effort to develop them separately. The authors elected not to implement an ESB solution as that would require too large of an implementation, integration, and deployment footprint. ESBs are also considered to be an older technology and the authors were looking for something more agile in development, maintenance, reuse, and rewritability.

The authors used Amazon lambda service along with Docker and Docker web hooks. Reuse and easy replacement were the main benefits the authors observed in this architecture. Performance, in terms of response time of services, was also another benefit.

The authors show that with the right container and deployment models and tools, Microservices can be a viable, high-performance solution. In 99% of the experimental results a Microservice service took between 70 and 300 milliseconds to respond. The same service implemented in a traditional SOA approach took close 3000 milliseconds, a 95% increase. The authors list four main advantages of Microservices:

a) More reuse of developed micro services. Since these Microservices are developed around specific business needs and capabilities they tend to be reused more often, saving development time for new Microservices.
b) The replacement of existing Microservices is easier and faster than more monolithic services, largely since developers have an easier time developing test cases for more narrowly defined Microservices and also simply due to their smaller code size.
c) With the correct underlying platform architecture and containers, Microservices can exhibit better performance than traditional monolithic service calls, such as calls to a SOAP service.
d) The use of Microservices causes a decrease in support time and resources needed; the authors acknowledge that this may be problematic because of older, legacy code mixed in with newer Microservices.

The authors note that although the final, long term (5-10 years) benefits of using Microservices have yet to be measured, they are already seeing concrete business benefits in their use. Ultimately, they acknowledge that many of the issues they face call for a combination of technical, methodological, and management approaches and solutions.

This article presents results that are useful for this study from experiments run on Microservices versus a traditional SOA architecture. It also compares the two approaches and lists the advantages of Microservices.

Xiao, Z., Wijegunaratne, I., & Qiang, X. (2016). Reflections on SOA and Microservices. In *2016 4th International Conference on Enterprise Systems (ES)* (pp. 60–67). https://doi.org/10.1109/ES.2016.14

Abstract: Today's Enterprises are facing many challenges in the service oriented, customer experience centric and customer demand driven global environment where ICT is becoming the leading enabler and partner of the modern enterprise. In the last decade, many enterprises have invested heavily in SOA-aligned IT transformations, but not harvested what SOA promised to provide. Now the API and Microservice paradigm has emerged as the "next big thing" for delivering IT outcomes to support the modern enterprise, with many technology vendors and service jumping on the bandwagon. This paper undertakes a critical investigation of the key concepts around SOA, API and Microservices, identifying similarities and differences between them and dispelling the confusion and hype around them. Based on our discussion and analysis, this paper presents a set of recommendations and best practices on the effective use and management of enterprise software components, drawing upon the best of SOA, API and Microservice concepts and practice.

Summary. The authors identify the introduction of SOA as occurring in the early 2000s and discuss various definitions that exist for SOA, recognizing loose coupling as one of its key concepts. Other definitions include decomposing everyday business applications into individual business functions and processes, and an architectural pattern in software design in which application components provide services to other components via a communications protocol. Other essential features associated with SOA include componentization into services and high performance and reuse. The authors note that Microservices are more business centric, less monolithic, and more independent than traditional SOA architectures. The authors also note that Microservice APIs are built on web standards such as HTTP and REST.

The authors introduce supporting layers in Microservices such as an Enterprise Registry, API management, proxies, domains, and containers. They present a side-by-side comparison of

traditional SOA to Microservices and note their similarities and differences, both from a consumer point of view and an internal, structure consideration. Some of these differences include the use of a service registry and service bus for traditional SOA, compared to an API registry and API management layer for Microservices. The authors identify the major difference between the two as the level of autonomy between individual services, describing Microservices as being not just loosely coupled but entirely uncoupled and running in their own container. The authors also note that Microservices have a decentralized governance model, as opposed to traditional SOA.

The authors assert that moving large legacy systems and applications from a monolithic SOA architecture to a Microservices-based architecture may not really be feasible because the legacy applications usually contain interdependent services that cannot be easily separated from each other. The authors assert that splitting these interdependent services would be very expensive, time consuming, difficult, and ultimately infeasible. They also note that many Microservices tend to be very fine-grained, reflecting the CRUD (Create, Read, Update, Delete) model of software and database development introduced by Tupper (2011), and that the business logic needs to be written into these services at the appropriate layer. Some of these larger, coarser business logic services may be traditional SOA applications that call finer-grained Microservices to accomplish the logic.

The authors conclude that Microservices and SOA are really complementary technologies that inform each other. The authors offer a bi-modal approach to using Microservices in conjunction with more traditional, legacy, monolithic SOA applications. Mode 1 resembles a legacy system and provides its scalability, efficiency, and safety. Mode 2, a more Microservices-based approach, provides more agility and speed. A service model for the second approach is

presented, which the authors call a two-speed approach to systems design. The authors assert that this model completes a complementary view of traditional and Microservices architectural approaches. The authors note that the two worlds of Microservices and traditional SOA are loosely coupled to each other, but not entirely separable. They conclude by calling for seamless integrations between the Microservice API approach to written services and traditional SOAs.

This article is useful for this study because it provides several definitions and viewpoints of SOA architectures and Microservice architectures. The authors note the complementary traits and recommend that the two architectures be operated together.

Conclusion

The authors in the majority of the resources in this annotated bibliography could not give a single definition of Service Oriented Architecture, Enterprise Service Bus, or Microservices. The various authors and sources offered several definitions of these technologies from academia and industry, with overlap among the properties associated with each architectural style. One point of similarity among these technologies and most of the definitions is the recognition that all are attempting to achieve a loosely coupled, efficient, and cost-efficient method to expose services amongst themselves and for outside consumers (Bhadoria et al., 2017; Erickson & Siau, 2008; Strîmbei et al., 2015; Xiao et al., 2016).

This annotated bibliography gathers scholarly sources to present background information on the rise of SOA, ESB, and Microservices architectures; their operational characteristics; and best practices in deploying these technologies. The sources note both distinctions between the three architectures as well as areas of commonality. They also highlight the relative strengths and weaknesses of each approach.

Background and history of SOA, ESB, and Microservices

Academic sources on each of the architectural sources varied, with the most sources identified for SOA, the oldest technology, and the newest and fewest number of sources identified for Microservices, the newest technology. A review of the literature reveals that the SOA, ESB, and Microservices architectural patterns have their roots in the need for interoperability among software systems, using such mechanisms as RPCs, CORBA, SOAP, and other client-server technologies (Erickson & Siau, 2008). SOA, ESB, and Microservices represent some of the newest technologies and methodologies for software architecture (Lewis & Fowler, 2014). Those who have created these architectures have also been motivated by the need

to lessen the real cost, monetarily and performance-wise, of technical debt (Johann, 2016). Ward Cunningham (1992) introduced the concept of technical debt in 1992, describing how "shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" (Tom et al., 2013, p. 1499).

SOA has been around the longest of the three architectures of study and may be considered the mother of ESB and Microservices (Salah et al., 2016). SOA represents an evolution from client-server to a service-oriented architecture, with the added benefits of reusability and loosely coupled services (Salah et al., 2016). ESB and Microservices are both part of the overall SOA family (Salah et al., 2016), and multiple authors note that there is nothing that forces ESBs or Microservices to operate mutually independent of each other (Xiao et al., 2016; Zimmermann, 2017). In fact, these authors assert that by using ESBs and Microservices together, the architectures complement each other in terms of their capabilities and performance characteristics (Xiao et al., 2016; Zimmermann, 2017).

Enterprise service buses have become more defined by their proprietary commercial and open source implementations, with no single accepted definition (Hérault et al., 2005). There are both open-source and proprietary ESB vendors, each offering varying levels of support and Service Level Agreements (Bhadoria et al., 2017).

Microservices are the newest of the technologies discussed (Lewis & Fowler, 2014). Microservices are grounded in the latest technological innovations and trends, including agile methodologies and DevOps operations (Ueda et al., 2016). Lewis & Fowler (2014) note that it is yet to be seen if Microservices become a dominant and lasting architectural pattern. Microservices can be thought of as breaking down Monolithic SOA-based applications into smaller, more manageable and reusable *microservices* that are easier to update and change

(Lewis & Fowler, 2014). As the newest technology, Microservices offer much promise but, along with SOA and ESB, also face potential issues (Salah et al., 2016).

Operational characteristics of SOA, ESB, and Microservices

Some of the key attributes of SOA, ESB, and Microservices include modularizing components, distributing components, and enabling reuse and extensibility (Gouigoux & Tamzalit, 2017). Scalability, ease of maintenance, and the ability to add monitoring and governance are additional benefits of these architectures (Lewis & Fowler, 2014). In addition, Lewis and Fowler (2014) note the ease of configuration in setting up routings, mappings, and service discovery to accommodate different businesses or network topology concerns as additional benefits of the three architectures. Extensibility and the capability to integrate new software modules are also important aspects when measuring software reusability (Henttonen et al., 2007).

Monolithic in nature, SOA is often harder to change, maintain, and extend than more agile approaches (Johann, 2016). SOA does gain advantages in its prevalence, the higher number of developers familiar with the technology, and often its performance and horizontal scalability (Salah et al., 2016). SOAs can however be costly and may be difficult to scale (Xiao et al., 2016). They are hard to extend and by their very nature take more effort and time to modify and extend (Fowler, 2015). Adding new services in a SOA architecture is often difficult because the whole application must be updated (Thönes, 2015).

Some of the most notable advantages observed for ESBs include their speed, fault tolerance, and easy governance and maintenance. (Exposito & Diop, 2014). In addition, the commercialization of ESBs has resulted in the availability of support and service level agreements that are vendor and platform specific (Exposito & Diop, 2014). ESBs require

configuration but also provide mediation and monitoring (Bhadoria et al., 2017; Hérault et al., 2005).

ESB architectures may make adding new services more problematic due to the proprietary natures of many ESB solutions (Bhadoria et al., 2017). ESBs tend to be less flexible in terms of their scalability, configuration, and extensibility than the other architectures (Zimmermann, 2017). Routing still remains an issue in ESB architectures, as traditional rule-based approaches do not provide mechanisms for dynamic reconfigurable routing (Bhadoria et al, 2017). Having a dynamic, reconfigurable routing mechanism would overcome a major limitation of ESBs (Bhadoria et al., 2017). Other areas of concern include security weaknesses and the limited availability of ongoing support and guidance supplied by any specific ESB vendor (Bhadoria et al., 2017).

ESBs by their nature use messaging, transformation, and routing mechanisms (Xiao et al, 22017). These elements can require more configuration and governance and may also contribute to ESBs being harder to extend (Xiao et al., 2016). Villamizar et al. (2015) also note that ESBs may be too monolithic and may not scale well for web-sized applications. Baude et al. (2010) found that by creating federations of multiple ESBs, working together, issues with scaling can be overcome.

ESBs also consume more resources than Microservices (Zimmerman, 2017). ESBs operate as a central locus of control, which, depending on the operational and business environment, may be considered a positive or negative attribute, with the positive view identifying the advantages of a single point of control while the negative view highlights the single point of failure (Thönes, 2015). ESBs suffer from limitations in performance capabilities when scaled to a globally internet-scaled ecosystem (Baude et al., 2010).

Microservices have roots steeped in agile methodologies and culture and show the promise of several advantages that SOA and ESB do not, including a reliance on business processes as opposed to system functional areas such as the user interface (UI) or database (Shadija et al., 2017); easier scalability; easier reusability; and lower operational, infrastructure, and development costs (Gouigoux & Tamzalit, 2017; (Villamizar et al., 2016). Reuse and extensibility of Microservices are easy, largely because they are less monolithic than SOA architectures and easier to update, reuse, or rewrite than components used in an ESB architecture (Johann, 2016). Microservices tend to be very fine-grained, reflecting the CRUD (Create, Read, Update, Delete) model of software and database development introduced by Tupper (2011). Microservices may not necessarily be micro in size but are considered more lightweight than ESBs (Shadija et al., 2017).

The research on the performance benefits of Microservices has been inconclusive. Different authors reported different performance results for Microservices, largely depending on which containerization tool was used and how it was configured (Ueda et al., 2016; Villamizar et al., 2015; Villamizar et al., 2016). Costs for Microservices were noted, in some studies, as being lower than those of monolithic architectures (Ueda, 2016; Villamizar et al., 2015; Villamizar et al., 2016). Performance and infrastructure and monthly operational costs can vary greatly depending on how Microservices are implemented (Ueda, 2016; Villamizar et al., 2015; Villamizar et al., 2016).

Microservices and the current trend of DevOps are well suited for each other (Lewis & Fowler, 2014). Zimmermann (2017) uses one of Lewis and Fowler's (2014) characteristics of Microservices to define DevOps: "Lean, but holistic and largely automated approaches to configuration, performance and fault management are employed, which extend agile practices

and include service monitoring" (p. 303). Continuous deployment and integration are key tenets of DevOps and vital for the success of Microservices (Strîmbei, 2015).

Best practices in deploying SOA, ESB, and Microservices

Well-defined requirements, an understanding of business needs, and a viable architecture in which integration points and patterns are well-defined are necessary for a successful software implementation (Rais, 2016). Some technologies, such as Microservices, lend themselves to continuous integration and deployments, whereas traditional SOAs and ESBs do to a lesser extent (Strîmbei et al., 2015). Carefully defining the interface boundaries between services is recommended as a best practice when designing systems using all three architectures (Lewis & Fowler, 2014; Rais, 2016).

Issues that arise when deploying SOA, ESBs, and Microservices include incompatibility with the software and networking environment; the level of average and peak usage expected, which can affect response times as well as operational costs; and the unforeseeable requirements of future deployments, upgrades, and extensions (Baude et al., 2010; Gouigoux & Tamzalit, 2017; Xiao et al., 2016).

A best practice for SOA implementations is to ensure they are comprised of reusable and loosely coupled services (Salah et al., 2016). Employing agile development methodologies is another recommendation when developing monolithic SOA applications (Salah et al., 2016). Xiao et al. (2016) recommend that SOA architectures encompass some coherent functionality that is important to the business itself. Strîmbei et al. (2015) also note that SOAs should be comprised of composable, or integrable components, with reusability being a primary criterion for defining a service.

Considerations when choosing a particular ESB product or implementation include usability in terms of the environments and platforms that are supported; adaptability in terms of the availability of adapters to other technologies or vendor-specific platforms; and flexibility in terms of the ability to customize a service or product (Bhadoria et al., 2017). Integration is a required part of any commercial ESB, but it has also been recommended to use open source solutions, as those are often less complex than commercial products (Bhadoria et al., 2017).

ESB products have been designed to support the workloads of enterprise applications with hundreds or thousands of users, but when ESBs are used with Internet-scale applications that have hundreds of thousands or even millions of users, they become bottlenecks, generating high latencies and providing a single point of failure (Villamizar et al., 2015). ESBs were not designed with the cloud in mind, so it is difficult to add or remove servers to support them on demand (Villamizar et al., 2015). Creating federations of ESBs along with some monitoring tools may be useful in identifying and surmounting these limitations (Baude et al., 2010).

Agile development is one of the most prevalent schools of thought for software and systems development (Gupta & Gouttam, 2017), and Microservices have roots firmly within the agile movement (Ueda et al., 2016). Lean or agile approaches to software and systems management also work well with Microservices (Zimmermann, 2017). Due to the nature of agile methodologies and practices from the Domain Driven Design Community (Thönes, 2015; Ueda et al., 2016), Microservices have strong ties to DevOps and the continuous integration and continuous deployment methodologies that DevOps embraces; a best practice is to make use of these methodologies and techniques when embarking on a Microservices project (Strîmbei et al., 2015). Another best practice when developing Microservices is to employ test-driven development methodologies (Zimmermann, 2017).

One characteristic of Microservices is that they are meant to be developed in an environment that is product-based or focused on the business processes, and subsequently owned by the same team for the lifetime of the service or application (Lewis & Fowler, 2014). Lewis and Fowler (2014) assert that this approach is more effective for Microservices development and support than traditional project-based approaches where teams are assembled for development, testing, and deployment of an application that is then handed off to a maintenance group.

The environment in which Microservices operate impacts the success or failure of this architecture, including considerations of cloud deployments (Ueda et al., 2016; Villamizar et al., 2015; Villamizar et. al., 2016). Options for organizations are to deploy Microservices in a Software as a Service (SaaS), Infrastructure as a Service (IaaS), or Platform as a Service (PaaS) model (Villamizar et al., 2015). Typically, companies use IaaS and PaaS to gain efficiency, particularly during peak usage times (Villamizar et al., 2015). High availability and continuous deployment are also reasons companies move applications to a cloud platform (Villamizar et al., 2015). The deployment and container technologies are important, with Docker and AWS Lambda showing great promise in terms of cost and performance for future work using Microservices (Villamizar et al., 2015); Villamizar et al., 2016).

Employing a set of monitoring and logging services is also a best practice for Microservices due to their decentralized nature and to ensure that applications based on them are fault, or service failure, tolerant (Lewis & Fowler, 2014). “Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both architectural elements (how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received)” (Lewis & Fowler, 2014). Operators can then be quickly alerted when a service fails, or needs to be refreshed (Lewis & Fowler, 2014).

Another consideration when employing these architectures, particularly for Microservices, is what, if any, containerization is used (Gouigoux & Tamzalit, 2017). Containers are lightweight, virtual machines built to run specific applications; they contain only those services necessary for the application to run, and are sealed from each other (Gouigoux & Tamzalit, 2017). The ability to reuse components, easy replacement of components or services, configurable runtime environments, strong performance, and ease of deployment are some of the reasons to use containerization (Gouigoux & Tamzalit, 2017). Careful selection, configuration, and tuning of containers is required to keep costs down and performance up (Villamizar et al., 2016). Xiao et. al (2016) note that Microservices benefit from supporting layers, such as enterprise registries and repositories, and API management to aid in service discovery. They also mention the addition of proxy APIs to segregate and expose APIs to only certain groups of consumers, both internal and external. Some of the coarser business logic services may be best handled by traditional SOA applications that call finer-grained Microservices to accomplish their logic (Xiao et. al, 2016).

Monolithic SOAs have the advantage of being the oldest and best understood of the three choices presented in this study (Shadija et al., 2017). There is also a huge code base of systems and applications, including web-based applications, that was developed using a monolithic SOA architecture that is available for reuse (Fowler, 2015). A reasonable, iterative, and extensible response to the need to leverage this large code base is the idea of using Microservices to extend existing monolithic SOA software (Xiao et al., 2016). In fact, moving from a monolithic to a Microservice architecture in carefully planned incremental steps can also preserve runtime performance (Lewis & Fowler, 2014). Lewis and Fowler (2014) note that the business logic

needs to be written into these services, which may be traditional SOA applications that call finer-grained Microservices, at the appropriate layer.

Xiao et al. (2015) note that dividing a monolith into smaller Microservices is often more successful than starting from scratch. Other experts assert that it may not be feasible to split a legacy, monolithic SOA-architected system into a Microservice-based system, indicating there are conflicting viewpoints on the possibility of success when attempting to divide SOA systems into Microservices (Salah et al., 2016; Xiao et al., 2015).

The decision of whether to use SOA, ESBs, or Microservices depends largely on the environment in which the system will be run, the application types and sizes it will be expected to support, and the skills and interests of the development staff and the business (Baude et al., 2010; Gouigoux & Tamzalit, 2017; Xiao et al., 2016). The most important consideration for those selecting among the architectures is the ability to expand and extend these systems as businesses change or grow in new directions (Baude et al., 2010; Salah, 2016). Reliability, performance, and responsiveness also deserve consideration when considering these technologies (Rais, 2016). When deploying and employing these architectures, many of the issues associated with these technologies need to be met with effective methodological and management approaches (Gouigoux & Tamzalit, 2017). Gouigoux and Tamzalit (2017) stress the need to choose the right team for the right service implementation.

Summary

Most of the references in this annotated bibliography note that ESBs and Microservices are specialized cases or implementations of Service Oriented Architectures (Bhadoria et al., 2017; Gouigoux & Tamzalit, 2017; Lewis & Fowler, 2014; Salah et al., 2016; Shadija et al., 2017; Strîmbei et al., 2015; Thönes, 2015; Villamizar et al., 2015; Xiao et al., 2016;

Zimmermann, 2017). Several studies also contrasted traditional monolithic SOA architectures to ESBs and Microservices (Salah et al, 2016; Villamizar et al., 2015; Zimmermann, 2017), with key findings that include the observation that ESBs tend to be a more heavyweight, inflexible, and unmanageable solution, where Microservices are more lightweight and fine-grained (Zimmerman, 2017). Microservices are the newest of the three technologies of study, and are mentioned in the literature as a convenient way to extend existing monolithic, legacy SOA applications (Fowler, 2015; Knoche, 2016). ESB applications offer a more centralized, but harder to extend, solution than Microservices or monolithic SOAs (Xiao et al., 2016).

All of the architectural design patterns of study have their places in modern systems designs (Zimmermann, 2017). SOA provides an overriding architecture and is the place where many legacy applications live (Zimmermann, 2017). Enterprise Service Buses provide additional routing, governance, and monitoring facilities (Bhadoria et al., 2017). Microservices can be used to easily extend existing applications, and with careful design can in some cases be used to create new ones (Xiao et al., 2016). The performance of each technology largely depends on the mechanisms, tools, and methodologies used for the deployment and the run-time environment of each type of technology (Ueda et al., 2016; Villamizar et al., 2015; Villamizar et al., 2016).

References

Baude, F., Filali, I., Huet, F., Legrand, V., Mathias, E., Merle, P., … Lorre, J.-P. (2010). ESB federation for large-scale SOA. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (pp. 2459–2466). New York, NY: ACM. https://doi.org/10.1145/1774088.1774597

Berners-Lee, T., Hendler, J., & Lassila, O. (2001). The Semantic Web. *Scientific American, 284*(5), 34-43. Retrieved from http://www.jstor.org/stable/26059207

Bhadoria, R. S., Chaudhari, N. S., & Tomar, G. S. (2017). The performance metric for Enterprise Service Bus (ESB) in SOA system: Theoretical underpinnings and empirical illustrations for information processing. *Information Systems, 65*(Supplement C), 158–171. https://doi.org/10.1016/j.is.2016.12.005

Center for Public Issues Education (n.d.). Evaluating Information Sources. University of Florida. Retrieved from http://ce.uoregon.edu/aim/Capstone1Perm/evaluateinfo.pdf

Chamberlain, R., Schommer, J. (2014). Using Docker to support reusable research. Tech. Rep., Technical report. Retrieved November 28, 2017 from: doi:10.6084/m9.figshare.1101910

Colisto, N. R. (2012). The CIO playbook: Strategies and best practices for IT leaders to deliver value. Wiley Online Library. Retrieved from https://ebookcentral-proquest-com.libproxy.uoregon.edu

Cunningham, W. (1992) The WyCash portfolio management system. *Procedures of ACM Object-Oriented Programming Systems, Languages and Applications*. New Orleans, LA: OOPSLA.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N., & Weerawarana, S. (2002). Unraveling the web services web: An introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, *6*(2), 86–93. https://doi.org/10.1109/4236.991449

Daki, H., Hannani, A. E., Aqqal, A., Haidine, A., & Dahbi, A. (2017). Big data management in smart grid: Concepts, requirements and implementation. *Journal of Big Data*, *4*(1), 13. https://doi.org/10.1186/s40537-017-0070-y

Diepenbrock, A., Rademacher, F., & Sachweh, S. (2017). *An ontology-based approach for domain-driven design of Microservice architectures*. Gesellschaft für Informatik, Bonn, 1777-1791. https://doi.org/10.18420/in2017_177

Erickson, J., & Siau, K. (2008). Web services, service-oriented computing, and service-oriented architecture: Separating hype from reality. *Journal of Database Management*, *19*(3), 42+. Retrieved from http://go.galegroup.com/ps/i.do?p=AONE&sw=w&u=s8492775&v=2.1&it=r&id=GALE%7CA191393402&asid=9009dc7d68d5504adc8d898c367b2d0c

Exposito, E. and Diop, C. (2014) *Smart SOA platforms in cloud computing architectures*. Hoboken, NJ: John Wiley & Sons, Inc. doi: 10.1002/9781118761489 Retrieved November 3, 2017 from: https://ebookcentral-proquest-com.libproxy.uoregon.edu/lib/uoregon/detail.action?docID=1734298

Fink, J., Librarian, D. S., & University, M. (2014). Docker: A Software as a Service, operating system-level virtualization framework. *The Code4Lib Journal*, (25). Retrieved from http://journal.code4lib.org/articles/9669?utm_source=feedburner&utm_medium=feed&utm_campaign=Feed%3A+c4lj+

Fowler, M. (2015, June 3). MonolithFirst. [Web log]. Retrieved November 27, 2017, from https://martinfowler.com/bliki/MonolithFirst.html

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software.* Reading, MA: Addison-Wesley.

Gode, D. K., Barua, A., & Mukhopadhyay, T, (1990). *On the economics of the software replacement problem.* ICIS 1990 Proceedings. Retrieved on November 24, 2017 from: http://aisel.aisnet.org/icis1990/26/

Gonzalez, D. (2016) *Developing Microservices with Node.JS*. Birmingham, AL: Pakt Publishing.

Gouigoux, J. P., & Tamzalit, D. (2017). From monolith to Microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 62–65). https://doi.org/10.1109/ICSAW.2017.35

Gupta, S., & Gouttam, D. (2017). Towards changing the paradigm of software development in software industries: An emergence of agile software development. In *2017 IEEE International Conference on Smart Technologies and Management for Computing, Communication, Controls, Energy and Materials (ICSTM)* (pp. 18–21). https://doi.org/10.1109/ICSTM.2017.8089120

Haoyu, W. & Haili, Z. (2012). Basic design principles in software engineering. *2012 Fourth International Conference on Computational and Information Sciences* (pp. 1251–1254). https://doi.org/10.1109/ICCIS.2012.91

Henttonen, K., Matinlassi, M., Niemelä, E., & Kanstrén, T. (2007). Integrability and extensibility evaluation from software architectural models – A case study. *The Open Software Engineering Journal*, *1*(1). https://doi.org/DOI:10.2174/1874107X00701010001

Hérault, C., Thomas, G., & Fourier, U. J. (2005). Mediation and Enterprise Service Bus: A position paper. In *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE)*. Retrieved November 13, 2017 from: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=B4F99BD6D6CF065700F531E5E1EC1C8E?doi=10.1.1.142.7416&rep=rep1&type=pdf

Johann, S. (2016). Dave Thomas on innovating legacy systems. *IEEE Software*, *33*(2), 105–108. https://doi.org/10.1109/MS.2016.38

Knoche, H. (2016). Sustaining runtime performance while incrementally modernizing transactional monolithic software towards Microservices. *ICPE '16 Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, 121-124. https://doi.org/10.1145/2851553.2892039

Krause, L. (2016). Microservices: Theory and application. In *Applicative 2016*. New York, NY: ACM. https://doi.org/10.1145/2959689.2960082

Kryvinska, N., Baroková, A., Auer, L., Ivanochko, I., & Strauss, C. (2013). Business value assessment of services re-use on SOA using appropriate methodologies, metrics and models. *International Journal of Services, Economics and Management*, *5*(4), 301–327. https://doi.org/10.1504/IJSEM.2013.059578

Kumar, S., Srivastava, S., & Singh, A. (2015). Web services: Past, present, and future. *International Journal of Computer Applications, 118*(19), 9-13. Retrieved November 20, 2017 from: https://s3.amazonaws.com/academia.edu.documents/42583874/pxc3903254.pdf?AWSAccessKeyId=AKIAIWOWYYGZ2Y53UL3A&Expires=1511243601&Signature=af4Feuo

x0AdxziBPIduGMfJkOjk%3D&response-content-disposition=inline%3B%20filename%3DWeb_Services_Past_Present_and_Future.pdf

Leppänen, M., Mäkinen, S., Lahtinen, S., Sievi-Korte, O., Tuovinen, A. P., & Männistö, T. (2015). Refactoring-A shot in the dark? *IEEE Software*, *32*(6), 62–70. https://doi.org/10.1109/MS.2015.132

Lewis, J., & Fowler, M. (2014). Microservices. [Web log]. Retrieved November 17, 2017, from https://martinfowler.com/articles/microservices.html

Mansoor, U., Kessentini, M., Bechikh, S., and Deb, K**.** (2014). Code-smells detection using good and bad software design examples. *COIN Report No. 2014009,* Retrieved November 13, 2017 from http://ai2-s2-pdfs.s3.amazonaws.com/4d45/1fe9370e86c193428365bb966089fb2ef84e.pdf

Mills, A. (n.d.) Evaluating the credibility of your sources (n.d.). Retrieved November 11, 2017 from http://www.college.columbia.edu/academics/academicintegrity

Montesi, F., & Weber, J. (2016). Circuit breakers, discovery, and API gateways in Microservices. Retrieved from http://search.arxiv.org:8081/paper.jsp?r=1609.05830&qid=1508628438514ler_nCnN_-141752822&qs=ESB+and+microservices

Nord, R. L., Ozkaya, I., Kruchten, P., & Gonzalez-Rojas, M. (2012). In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture* (pp. 91–100). https://doi.org/10.1109/WICSA-ECSA.212.17

Ouertani, S. (2015) From Microservices to SOA. *Service Technology Magazine, XCI,* 4-10. Retrieved November 1, 2017 from

http://servicetechmag.com/system/application/views/I91/ServiceTechMag.com_Issue91_online.pdf

Psiuk, M., Bujok, T., & Zieliński, K. (2012). Enterprise Service Bus monitoring framework for SOA systems. *IEEE Transactions on Services Computing*, *5*(3), 450–466. https://doi.org/10.1109/TSC.2011.32

Rademacher, F., Sachweh, S., & Zündorf, A. (2017). Differences between model-driven development of service-oriented and microservice architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)* (pp. 38–45). Retrieved from: https://doi.org/10.1109/ICSAW.2017.32

Rais, A. A. (2016). Interface-based software integration. *Journal of Systems Integration*, *7*(3), 79–88. https://doi.org/10.20470/jsi.v7i3.261

Sadi, M. H., & Yu, E. (2014). Analyzing the evolution of software development: From creative chaos to software ecosystems. In *2014 IEEE Eighth International Conference on Research Challenges in Information Science (RCIS)* (pp. 1–11). https://doi.org/10.1109/RCIS.2014.6861055

Salah, T., Zemerly, M. J., Yeun, C. Y., Al-Qutayri, M., & Al-Hammadi, Y. (2016). The evolution of distributed systems towards microservices architecture. In *2016 11th International Conference for Internet Technology and Secured Transactions (ICITST)* (pp. 318–325). https://doi.org/10.1109/ICITST.2016.7856721

Salvadori, I., Huf, A., Mello, R. dos S., & Siqueira, F. (2016). Publishing linked data through semantic microservices composition (pp. 443–452). In *Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services*, ACM. https://doi.org/10.1145/3011141.3011155

Schmidt, D. (1999). Why software reuse has failed and how to make it work for you. Retrieved October 25, 2017, from https://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html

Shadija, D., Rezai, M., & Hill, R. (2017). Towards an understanding of Microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)* (pp. 1–6). https://doi.org/10.23919/IConAC.2017.8082018

Sneed, H. M. (2014). Dealing with technical debt in agile development projects. In *Proceedings of the 6th International Conference on Software Quality (SWQD)* (pp. 48–62). Retrieved November 3, 2017 from https://doi.org/10.1007/978-3-319-03602-1_4

The Standish Group International (2009). CHAOS Summary 2009: The10 Laws of CHAOS. Retrieved from: https://www.classes.cs.uchicago.edu/archive/2014/fall/51210-1/required.reading/Standish.Group.Chaos.2009.pdf

Strîmbei, C., Dospinescu, O., Strainu, R.-M., & Nistor, A. (2015). Software architectures – Present and visions. *Informatica Economica*, *19*(4), 13–27. https://doi.org/10.12948/issn14531305/19.4.2015.02

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1), 116–116. https://doi.org/10.1109/MS.2015.11

Tom, E., Aurum, A., & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, *86*(6), 1498–1516. https://doi.org/10.1016/j.jss.2012.12.052

Tupper, C. (2011). *Data architecture: From zen to reality*. San Francisco, CA: Elsevier Science. Retrieved from http://ebookcentral.proquest.com/lib/uoregon/detail.action?docID=675402

Ueda, T., Nakaike, T., & Ohara, M. (2016). Workload characterization for Microservices. In *2016 IEEE International Symposium on Workload Characterization (IISWC)* (pp. 1–10). https://doi.org/10.1109/IISWC.2016.7581269

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (pp. 583–590). https://doi.org/10.1109/ColumbianCC.2015.7333476

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., … Lang, M. (2016). Infrastructure cost comparison of running web applications in the cloud using AWS Lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)* (pp. 179–182). https://doi.org/10.1109/CCGrid.2016.37

Xiao, Z., Wijegunaratne, I., & Qiang, X. (2016). Reflections on SOA and Microservices. In *2016 4th International Conference on Enterprise Systems (ES)* (pp. 60–67). https://doi.org/10.1109/ES.2016.14

Zimmermann, O. (2015) Do Microservices pass the same old architecture test? Or: SOA is not dead–Long live (Micro-) services. *Position Paper for SEI SATURN 2015 Workshop.* Retrieved November 1, 2017 from https://www.researchgate.net/profile/Olaf_Zimmermann2/publication/282323615_Do_Microservices_Pass_the_Same_Old_Architecture_Test_or_SOA_is_Not_Dead_-_Long_Live_Micro-Services_Position_Paper_for_SEI_SATURN_2015_Workshop/links/560bbc2d08ae7fa7b886f3a9.pdf

Zimmermann, O. (2017). Microservices tenets. *Computer Science - Research and Development, 32*. https://doi.org/10.1007/s00450-016-0337-0

Zimmermann, O., Pautasso, C., Hohpe, G., & Woolf, B. (2016). A decade of enterprise integration patterns: A conversation with the authors. *IEEE Software*, *33*(1), 13–19. https://doi.org/10.1109/MS.2016.11