Microservice Decompositon: A Case Study of a Large Industrial Software Migration in the Automotive Industry

Heimo Stranner, Stefan Strobl, Mario Bernhart and Thomas Grechenig
Research Group for Industrial Software, Vienna University of Technology, Vienna, Austria

Keywords: Microservices, Decomposition Approach, Maintainability, Scalability, Bounded Contexts, Facades.

Abstract: In a microservice architecture a set of relatively small services is deployed, who communicate with each other only over the network. Monoliths regularly suffer from poor scalability and maintainability. Several approaches for decomposing them into microservices have been proposed with the aim to improve these characteristics. However, precise descriptions of these approaches in combination with large scale industrial evaluations are still rare in academic literature. This case study focuses on a large ERP system in the automotive industry. We applied an approach based on the concept of bounded contexts for one such decomposition and documented necessary changes to the system, like the introduction of facades to facilitate incremental migration towards microservices in a non-distruptive manner. Further we conduct expert interviews to evaluate our findings. While the migration is still ongoing, we were able to achieve significant adoption rates of the new paradigm and a clear preference of architects and developers to use it. Development speed has also drastically improved.

1 INTRODUCTION

Large monolithic software systems commonly have a number of problems. Among the most prominent and pressing ones are increasing difficulty in maintaining and scaling them. Horizontal scaling of monoliths can only be done in discrete steps for the whole application and is not flexible. Relational databases shared among all instances may become bottlenecks (Pokorny, 2013).

Recently microservices have rapidly gained popularity. They can be seen as a variant of Service-oriented architecture (SOA) but others claim it is a new architecture. Technically it can be described as a more fine grained SOA (Zimmermann, 2017). In a microservice architecture there is a high number of small services that each solves only one task (Xiao et al., 2017). There is no universal consensus how big such a task and the corresponding microservice should be. Microservices communicate over lightweight protocols without complicated logic, commonly referred to as “dumb pipes” (Alpers et al., 2015).

A good microservice architecture improves maintainability because one can more easily reason about a single microservice in isolation (Dragoni et al., 2016). Every single service can be scaled independently of the other services. Highly scaleable data stores can be used instead of relational databases. Another benefit is reduced technology lock-in. Different microservices only have to be able to communicate with each other over the network, using mutually understood protocols. In a monolith all parts of the system are packaged in one executable and there is far less freedom in experimenting with different programming languages or technologies. Using this freedom is not without costs unfortunately. Employing a variety of different tools requires greater know how and there is a larger potential for running into problems with one of them. Maintenance also can become more costly as experts for all used technologies are required. Another disadvantage is that this architecture introduces a distributed system which incurs additional complexity compared to a monolith (Bakshi, 2017).

Creating a well suited software system for a given task is not easy. Especially if the system is a distributed system in general or based on a microservice architecture. The granularity of the services need to be determined and the borders need to be defined. It is sensible to draw borders where there are only few and well defined interactions between modules in a system. If there is no prior system or given company structure that can be mimicked, it is challenging to predict what borders will lead to the most maintain-

Stranner, H., Strobl, S., Bernhart, M. and Grechenig, T.
Microservice Decompositon: A Case Study of a Large Industrial Software Migration in the Automotive Industry.
DOI: 10.5220/0009564604980505
In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2020)*, pages 498-505
ISBN: 978-989-758-421-3

able, scalable or performant result (Hassan and Bahsoon, 2016).

Conventional criteria for software modularization proposed by academics like coupling and cohesion can theoretically be applied to such microservice decompositions. However they generally do not satisfy practitioners. In practice more diverse characteristics of the software are relevant and not just such one-dimensional considerations (Carvalho et al., 2019).

Many big corporations already have one or more monoliths in use which suffer from various problems such as poor maintainability and scalability (Hasselbring and Steinacker, 2017; Villamizar et al., 2015; Supulniece et al., 2015). It can be desirable to replace them with a microservice architecture but doing this in the most efficient way and achieving good results is far from trivial. Having access to a number of well applicable approaches, that have been empirically evaluated and help making the right decisions, are essential.

2 RELATED WORK

Academics as well as practitioners in the software industry proposed a number of different approaches to decompose monoliths (Balalaie et al., 2015; Newman, 2015). While there exist some academic evaluations of decomposition approaches, they cover academic or small industrial projects that are refactored into a microservice architecture (Šūpulniece et al., 2015). Decomposing large projects is complicated by having to understand a big system in order to determine where to draw borders between the newly split parts and the refactoring effort to do so. Most academic approaches which are focusing on a few formal criteria to decompose monoliths are rarely used in practice. The available tooling support is not considered helpful by practitioners (Carvalho et al., 2019).

Conway's law is cited repeatedly in the context of microservice decomposition. It states that organizations are bound to create system designs which resemble their communication structures (Alpers et al., 2015; Dragoni et al., 2016).

A migration can cut along existing boundaries between domain entities. This follows the Bounded Context (BC) pattern employed by Domain Driven Design (DDD) (Balalaie et al., 2016; Balalaie et al., 2015). The BCs should be incrementally decreased in size. At the beginning the monolith constitutes one BC and these are iteratively split up to match smaller domain concerns (Balalaie et al., 2015). A BC should have relatively few interdependencies with other BCs.

Microservice decomposition is a complex topic and there are no universal approaches that lead to good and efficient results for every project. Therefore a repository of patterns that can be used and that are proven to be useful under some circumstances is proposed (Balalaie et al., 2018).

3 METHODOLOGY

The case described below gives us the chance to evaluate the approach of utilizing the concept of bounded contexts to decompose a monolith into a series of microservices in the setting of a large industrial application. We observed how changes to the system are made and documented them. In order to evaluate the effects of the decomposition on the day to day development work we conducted expert interviews.

3.1 Research Objective

We describe the approach which is used for microservice decomposition in detail in section 4.

We shall focus on the quality aspects of maintainability and scalability because improving them is a major motivation to decompose large industrial systems and was crucial in this case.

- **RQ1:** How is a large industrial monolith affected by decomposing according to bounded contexts and introducing facades?
  - **RQ1a:** How is the maintainability affected?
  - **RQ1b:** How is the scalability affected?
- **RQ2:** Are there other decomposition approaches for large industrial monoliths that promise better decomposition results?

3.2 Case Study Subject

We apply the approach for decomposing a monolith into a microservice architecture to a large Enterprise Resource Planning (ERP) system in the automotive industry. The system is a Dealership management system (DMS) handling both the sales process of cars and the after sales and maintenance tasks commonly performed by car retailers. Interoperability with vast amounts of third party systems is provided both to comply with local regulations and to exchange information with other systems in the automotive ecosystem. In order to satisfy the needs of diverse customers from various countries, the system is very configurable including the ability to enable and disable various components and integrations. Said system is under active development and suffers from poor maintainability and scalability. Its monolithic architecture

has been identified as major problem and as a result the decision to decompose the system in smaller parts and gradually move towards a microservice architecture has been made.

There are two major functional parts of the software corresponding with the main process areas of car dealerships: the processes of selling a car referred to as *sales* and the processes of providing services to customers afterwards, called *after sales*. No formal requirements or specifications exists, but only informal descriptions in different formats spread over multiple systems including wikis and issue trackers.

The development of the new software began about ten years ago. Back then the project name was different and has since been changed once again. At the start Subversion (SVN) was used as version control system but in 2012 Git took over this role. Technology wise Java and the Spring Framework are used. As frontend framework Apache Tapestry is used. Several deployable web applications are built from the same code base through a complex graph of Maven dependencies and Spring configurations.

One major issue with the project is slow development time. This is to a large extent caused by the size of the biggest Git repository containing the code for several executables with lots of shared dependencies and lots of dependencies on other services. Historically there was one large SVN source code repository. Now there is a desire to reduce the size or use multiple repositories in order to improve the build granularity and development speed. Due to the monolithic nature of the application, this is not an easy task. Building the largest project on a developer workstation takes about ten minutes at the point of writing and starting one service takes about another five minutes. Continuous Integration (CI)-builds take between around 45 minutes to an hour, depending on server load. The high latency between making changes and being able to run them or having it tested in a reliable CI infrastructure makes developing much slower than desired.

Both management and the developers consider continuous delivery as desirable in the company. The progress towards fully employing continuous delivery has not progressed very far however.

3.3 Case Study Design

In order to answer the research questions about the current approach, a number of developers apply it to the system at hand and we observe changes to the system during a period of over two years. This is possible as the primary author is part of the group of developers tasked with implementing changes and new features in the system as well as to modularize its old monolithic architecture into a microservice architecture. This allows us to check if the approach is indeed well suited to the decomposition of large industrial monoliths following the case study methodology (Runeson et al., 2012). We also measure parts of the research questions, which can be directly measured, like the duration of a build on the CI server.

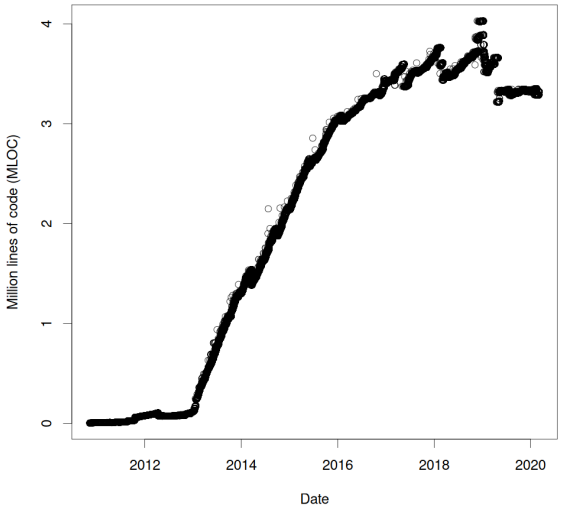

Figure 1: Loc history of the monolith.

In order to answer the research question about alternative approaches, these need to be found which is done by performing a literature review.

Like any system, the system at hand has some properties. Primarily there is no formal specification available, which is very common in the industry (Ozkaya, 2018).

We assess the approaches for decomposing monoliths into microservices found by searching for "microservice decomposition approach" on Google Scholar. Their applicability for decomposing the system at hand is determined considering the systems properties. To limit the search extend, only the 25 most relevant entries from the search query are considered.

4 USED DECOMPOSITION APPROACH

There are efforts to reduce the size of the current monolith and split out separate services belonging to business units which have their own bounded context. This is intended to increase the speed of development, the maintainability and scalability. Management expects teams to more independently own "their" part of the whole software system. These extraction efforts can be seen starting from 2017 as the number of

lines of code is abruptly reduced several times. Figure 1 shows that the number of lines of code has started to repeatedly swiftly decrease by significant amounts. This stems from the fact that some modules are already moved out to dedicated services.

4.1 Moving Existing Functionality out of the Monolith

In order to move a part of a monolith out into a microservice, architects identify a suitable part first. They take business considerations into account following the approach and identify a bounded context within the monolith that is suitable to be moved into a dedicated microservice or set of services. To make the migration easier, the bounded context should not be central in the monolith with huge amounts of chatty interactions but rather some bordering functionality. A bounded context with only incoming calls but with no outgoing calls into the rest of the system would be ideal. Technical architects make an intuitive decision supported by dependency statistics and integrated development environments (IDEs).

Once the decision is made which bounded context should become its own microservice, a process consisting of several steps begins.

Lets assume there are three services. *Service A* calls *Service B* which calls *Service C*. Only *Service B* is inside the bounded context which should be transformed to a new service.

4.1.1 Definition of Facades

Generally a facade hides the complex internal matters behind a clean interface. From the outside the system should only be accessed through this facade.

For the system at hand the desired communication pattern are rest calls. Therefore, the facades used have the form of rest interfaces.

Developers create new Maven modules containing the application programming interface (API) in the form of Representational state transfer (REST) interfaces and Data transfer objects (DTOs), which are transmitted.

Technical architects search for common usage patterns and try to define a small but usable REST API that can replace all prior usages.

4.1.2 Facade Implementation

As the second major step in the migration process the facades in the form of REST services have to be implemented.

When the same functionality already exists, this process is trivial. In most cases simple mappings and service calls are sufficient but in others more complex logic has to be implemented.

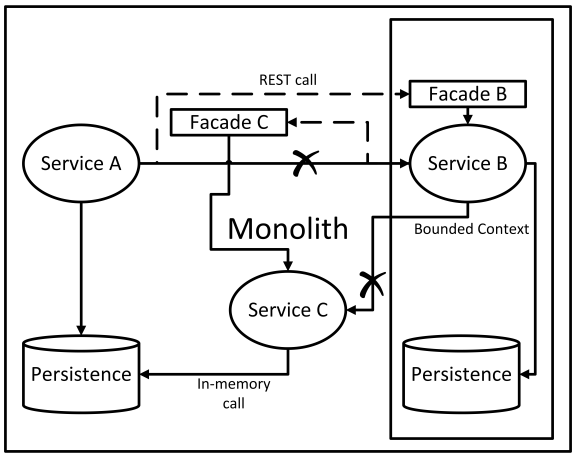

Figure 2: Facade calls are added.

4.1.3 Using the Facades

The third major step in splitting out a part of a monolith into a microservice is to actually use the facades in the form of REST endpoints and clients.

Developers search the rest of the monolith with IDEs for direct usages of the part which is about to be split out of it. The found usages circumvent the defined REST facade and have to be replaced.

Figure 2 shows the process of replacing in-memory calls across the new border with REST facade calls. Normal arrows represent in-memory calls and REST calls have arrows with dotted lines.

Many usages can very easily be replaced by REST calls and the corresponding DTOs. Transactions however become more complex. Distributed transactions are not desirable and also not supported in the system at hand. In many cases it is possible to move the whole transaction to one service and to provide aggregated data from the other service.

The visualized case also assumes that the database tables can be cleanly partitioned between the remaining monolith and the new microservice. Since bounded contexts are used, this is likely the case. In the remaining few cases the technical architects make intuitive decisions on how to proceed.

4.1.4 Moving out the Separated Service Part

The final part during the separation process finally results in a new microservice.

Developers create two separate Git repositories containing the API and the implementation of new service. Public and private code and data structures are cleanly separated that way.

The new application is also build on Jenkins, Sonar and the artifacts are deployed. Docker images are build and Helm Charts configure how the service should run in a cluster. The Uniform Resource Locators (URLs) for accessing the new microservices functionality is changed, as it is no longer a part of the monolith and therefore has a new base-URL.

4.2 Rewriting Part of the Monolith as a Microservice

The software quality is better in some areas and worse in others. Particularly problematic pieces with bad maintainability lend themselves well not to be moved out to their own microservice but to be replaced by a completely new microservice. In the short term it may be more effort to rewrite that functionality but the reward is substantial as the bad old code is completely replaced.

4.2.1 Facade Introduction and Implementation

In the beginning there are a set of services which should be removed. Lets assume this is just service *B1*. *Service B2* in the microservice provides similar functionalities and it is destined to replace *B1*. Figure 3 visualizes this scenario.

As a first step facades around the functionalities in the monolith which are about to be replaced are required. Again these facades also have to be implemented. The first implementation delegates to the existing implementation within the monolith.

The *Facade E* is added around the functionality of the services which are about to be replaced. For simplicity's sake the example this is just one service, but in most cases there is more than just one service. The REST endpoint facade (*Facade F*) is also added. This is the interface describing the corresponding REST endpoint of the new microservice. These facades have to be implemented.

4.2.2 REST Client Facade Implementation

Since a new microservice is not expected to immediately have all functionalities the old monolithic implementation has, the migration process takes some time. During this time one can use Spring profiles to switch between the old monolithic implementation and the new implementation which calls a microservice.

This is possible by providing a second implementation to *Facade E* which calls *Facade F* via REST.

In the end developers remove the old implementation (*Service B1*) and the implementation which forwards to the other service becomes the only implementation. Then the new implementation is always active, independent from Spring profiles.

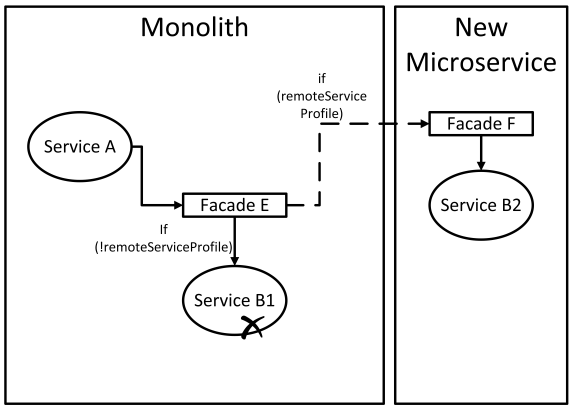

Figure 3: Profile controlling local or remote implementation.

4.3 Considering Alternative Approaches

The used approach is not necessarily optimal. Therefore, alternative approaches are also considered in the form of a literature review. The requirements for the program are rather loose and there is no specification available, even less so a formal one. It is not economically viable to create one for this project. It is also not common practice for projects of such a size (Carvalho et al., 2019). This rules out some more formal and automated approaches.

We find a lot of support for using bounded contexts. Several papers recommend using business capabilities and organizational boundaries as inspiration for where the monolith should be split into microservices, as already practiced. They describe boundaries between bounded contexts as excellent choices for microservice boundaries because the desired tight coupling inside a proper bounded context and thus a microservice is implied (Krylovskiy et al., 2015; Hasselbring and Steinacker, 2017; Shadija et al., 2017; Newman, 2015; Bakshi, 2017). One also recommends microservice boundaries between bounded contexts and additionally more fine grained boundaries between different storage and persistence requirements and maturities (Carneiro and Schmelmer, 2016). DDD is recommended in order to get a basic microservice architecture and to analyse usage patterns to further refine the architecture afterwards (Mustafa et al., 2018). While the focus of two papers is not about the decomposition, the papers shortly mention the same approach favourably (Alpers et al., 2015; Zimmermann, 2017).

Employing service facades are also recommended

as a first step to recover the architecture within a monolith. The decisions in which parts to cut follows DDD (Knoche and Hasselbring, 2018).

More formal approaches also exist and are often recommended. For large industrial systems without specification they are not feasible however (Carvalho et al., 2019).

5 RESULTS

In order to evaluate the answers given to the research questions, we interview experts on the system at hand and the modularization procedure. This procedure follows the well established methodology of an expert interview.

The experts answers confirm that the chosen approach is indeed well suitable to improve the situation at hand.

Nevertheless there are problems with the implementation of it. Non-technical reasons like a very limited budget and high pressure to release new features quickly cause most problems. Since only limited resources are used for the modularization effort, progress is slow.

The experts are content with focussing on business capabilities and bounded contexts as a basis for the modularization. The requirement for good Technical Architects (TAs) and Product Owners (POs) is stressed and that technical contexts are also important.

The experts consider more formal approaches infeasible for a project of the given size and budget. Creating formal models and specifications is estimated to be prohibitively expensive. While some help is considered helpful, it should not be too restrictive to the developers. Another two interviewed developers state that the overhead would be simply too high for a sizeable project. Since there is more complexity involved, the result is not expected to be any better by one interviewee.

They judge the usage of facades generally in a positive way, but mention that some issues were encountered when paired with insufficient domain knowledge and test automation.

The experts state that scaling the modularization effort up to multiple teams is easily possible as long as clear separations between different areas exist. During the modularization, a lot of files are changed and this leads to hard to solve merge conflicts if multiple persons are active on the same part of the source code.

Developers perceive working with the newer smaller services very positively. The main reasons are easier understandability, greatly improved development speed and faster feedback cycles. These qualities have a large positive impact on maintainability. One interviewee even calls the split inevitable but warns of overdoing it and creating too much tiny services. Those are less maintainable when any reasonable work spans multiple services and the API has to be changed all the time.

Working with a smaller service also allows to upgrade the code more easily as fewer dependencies have to be taken into consideration. This leads in the long term to a more modern application with which developers are more satisfied and prefer working with.

From a testers perspective the microservice approach is also preferable as a microservice can be more easily tested. Test automation is simpler as services can be tested in a more isolated way than the monolith.

5.1 Maintainability

RQ1a: asks how the maintainability is affected. There is a general consensus that the situation improves a lot, but there is still much to do.

For the new modules the feedback of the experts is very promising. Smaller modules are easier to understand, start and debug. Testing a smaller application also becomes easier and more efficient.

As testers primarily use quality assurance deployments and do not need to work with local code, deployment speed is an important part of the latency with which a tester can start to examine changes which also improves maintainability. The newer smaller modules can be deployed much faster than the remaining monolith.

A negative aspect mentioned is that some actions like version updates have to be repeated for many applications instead of only one monolith. This is not hard but cumbersome. Maintenance is greatly reduced per module but is required for more modules.

Developers understand smaller modules more easily and thus new developers get productive faster. This in turn improves the maintainability of these parts.

When the boundaries of a module are left, debugging becomes harder compared to a monolith where it is possible to debug across the whole application inside an IDE.

When developers change the API between modules, the effort is generally larger than in a monolith where such an API does not formally exist. As long as changes are internal to an application, the maintainability improves but as soon as multiple applications are affected the advantages diminish and new

problems arise. For backwards compatible changes a new API version has to be released in addition to the code changes. Other applications have no stress to upgrade. If a change is not backwards compatible however, all applications depending on the functionality have to upgrade at the same time or API versions have to be used. The effort is multiplied by the number of usages. For this reason it is crucial to keep the number of API changes minimal.

5.2 Scalability

RQ1b: is about how the scalability is affected. Scalability is also a widely given reason for a microservice architecture. In the project mixed results are reported. Theoretically once the modularization is finished it should allow to scale all applications very well according to interviewee D, but it is not experienced in the project yet as the system overall is perceived as rather slow regardless of the load and more replicas do not help directly with latency problems. If parts of the system are identified as under particular high stress, it is advantageous if this is a microservice, as administrators can independently scale it up.

While the scalability of a single service in isolation becomes easier, the whole orchestration becomes more complicated. Once this is properly taken care of however, the benefits of being able to scale every service according to its needs becomes prevalent. Container platforms and other operational details are therefore crucial for the success of the scalability improvements.

6 THREATS TO VALIDITY

For case studies it is always crucial to ensure the academic scalability. The results should not be specific to the system at hand but have to be generally applicable.

A literature review and the judgment of alternative approaches would come to the same result when conducted independently from the system at hand. Any approach that is not deemed applicable as potential alternative to the current approach, is also not applicable to most other large scale industrial monoliths as well.

Other approaches on more formal methods are deemed unsatisfactory both by the experts and in the academic papers (Ozkaya, 2018).

7 FUTURE WORK

Since the decomposition of the system at hand is not completed at the point of writing and it is expected that this will take a long time, another analysis once it is done could provide further insights. Long term effects of the modularization could be observed and the scalability improvements better evaluated when the system is deployed with a much larger user base and thus higher scalability demands.

Future work may include performing more case studies, further evaluating the chosen approach for other large scale industrial systems and combining the knowledge in larger meta analyses that generate results with statistical significance.

Since some alternative approaches found in the literature lack good tool support required for their widespread application to large systems, fully developing these tools, integrating them with practitioners tools like IDEs and performing case studies to evaluate their practical applicability could be further studies.

8 CONCLUSION

The evaluated large ERP system in the automotive industry was historically developed as monolith. It suffered from poor maintainability and scalability. In order to improve the situation, management made the decision to migrate to a microservice architecture.

The chosen approach is to use bounded business contexts in order to determine what should become a microservice. Developers then either rewrite relevant parts of the system as a microservice or extract them from the existing monolith. In order to minimize the disruption and allow gradual changes, they use facades to switch between different implementations of the same functionality as desired.

Developers started to move out pieces of the monolith into microservices while also rewriting some parts in the form of a completely new microservice. When a new microservice is created, which replaces old functionality, both implementations have to be maintained until the migration is complete. Therefore it is desirable that any such migration finishes soon.

Performing changes to a microservice is faster compared to a monolith. When working with the monolith, the IDE often lags, startup times are high and newer developers need to understand a lot about the architecture in order to be productive. The newer microservices in contrast are easier to understand, state of the art technologies are used and developers

get faster feedback both from local tools as well as the CI server.

To evaluate the results, we conducted expert interviews of stakeholders involved in the development of the system. They are familiar or directly involved with the modularization effort.

They agree that maintainability for the already modularized parts has greatly improved. Drastic changes in scalability are currently not visible but they agree that in general the smaller services are much more scalable.

The developers greatly prefer working on smaller services and are content with the applied approach. More formal approaches are largely disliked by the interviewed experts.

REFERENCES

Alpers, S., Becker, C., Oberweis, A., and Schuster, T. (2015). Microservice based tool support for business process modelling. *Proceedings of the 2015 IEEE 19th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations, EDOCW 2015*, (January 2017):71–78.

Bakshi, K. (2017). Microservices-based software architecture and approaches. *IEEE Aerospace Conference Proceedings*.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2015). Microservices Migration Patterns. (1):1–21.

Balalaie, A., Heydarnoori, A., and Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, 33(3):42–52.

Balalaie, A., Heydarnoori, A., Jamshidi, P., Tamburri, D. A., and Lynn, T. (2018). Microservices migration patterns. *Software - Practice and Experience*, 48(11):2019–2042.

Carneiro, C. and Schmelmer, T. (2016). *Microservices From Day One*.

Carvalho, L., Garcia, A., Assuncao, W. K., De Mello, R., and Julia De Lima, M. (2019). Analysis of the Criteria Adopted in Industry to Extract Microservices. *Proceedings - 2019 IEEE/ACM Joint 7th International Workshop on Conducting Empirical Studies in Industry and 6th International Workshop on Software Engineering Research and Industrial Practice, CESSER-IP 2019*, pages 22–29.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2016). Microservices: yesterday, today, and tomorrow. pages 1–17.

Hassan, S. and Bahsoon, R. (2016). Microservices and their design trade-offs: A self-adaptive roadmap. *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, pages 813–818.

Hasselbring, W. and Steinacker, G. (2017). Microservice architectures for scalability, agility and reliability in e-commerce. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, pages 243–246.

Knoche, H. and Hasselbring, W. (2018). Using Microservices for Legacy Software Modernization. *IEEE Software*, 35(3):44–49.

Krylovskiy, A., Jahn, M., and Patti, E. (2015). Designing a Smart City Internet of Things Platform with Microservice Architecture. *Proceedings - 2015 International Conference on Future Internet of Things and Cloud, FiCloud 2015 and 2015 International Conference on Open and Big Data, OBD 2015*, pages 25–30.

Mustafa, O., Marx Gómez, J., Hamed, M., and Pargmann, H. (2018). GranMicro: A Black-Box Based Approach for Optimizing Microservices Based Applications. In Otjacques, B., Hitzelberger, P., Naumann, S., and Wohlgemuth, V., editors, *From Science to Society*, pages 283–294, Cham. Springer International Publishing.

Newman, S. (2015). *Building Microservices*.

Ozkaya, M. (2018). Do the informal & formal software modeling notations satisfy practitioners for software architecture modeling? *Information and Software Technology*, 95(May 2017):15–33.

Pokorny, J. (2013). NoSQL databases: A step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82.

Runeson, P., Host, M., Rainer, A., and Regnell, B. (2012). *Case Study Research in Software Engineering*.

Shadija, D., Rezai, M., and Hill, R. (2017). Towards an understanding of microservices. In *2017 23rd International Conference on Automation and Computing (ICAC)*, pages 1–6.

Šūpulniece, I., Poļaka, I., Bērziņa, S., Ozoliņš, E., Palacis, E., Meiers, E., and Grabis, J. (2015). Source Code Driven Enterprise Application Decomposition: Preliminary Evaluation. *Procedia Computer Science*, 77:167–175.

Supulniece, I., Polaka, I., Berzisa, S., Meiers, E., Ozolins, E., Grabis, J., and Supulniece, I., Polaka, I., Berzisa, S., Meiers, E., Ozolins, E., & Grabis, J. (2015). Decomposition of Enterprise Application: A Systematic Literature Review and Research Outlook. *Information Technology and Management Science*, 18(1):30–36.

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., and Gil, S. (2015). Evaluating the Monolithic and the Microservice Architecture Pattern to Deploy Web Applications in the Cloud Evaluando el Patrón de Arquitectura Monolítica y de Micro Servicios Para Desplegar Aplicaciones en la Nube. *10th Computing Colombian Conference*, pages 583–590.

Xiao, Z., Wijegunaratne, I., and Qiang, X. (2017). Reflections on SOA and Microservices. *Proceedings - 4th International Conference on Enterprise Systems: Advances in Enterprise Systems, ES 2016*, pages 60–67.

Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. *Computer Science - Research and Development*, 32(3-4):301–310.