Moving mountains – practical approaches for moving monolithic applications to Microservices

Nhiem Lu[1], Gert Glatz[2], and Dennis Peuser[2]
[1] University of Applied Science and Arts, Dortmund, Germany
[2] adesso AG, Dortmund, Germany
nhiem.lu@fh-dortmund.de, gert.glatz@adesso.de,
dennis.peuser@adesso.de

Abstract

The advent of Microservice Architecture led to a rethinking of legacy systems in many organizations. While Microservices provide numerous advantages, several challenges are faced when migrating a monolith application, in terms of technical, processual and human aspects. Using industry examples, best practice approaches will be identified for successfully moving from a monolith to Microservices.

1 Introduction – the need for speed

With the advances of the Digitization, the need to deliver fast digital solutions to the users rises, meaning many organizations must rethink how to proceed with their legacy systems. Often, these software behemoths were developed many decades ago and have grown organically over time. In the past, replacing a monolithic application was often not considered due to the effort involved and the technical capabilities at hand (Bucchiarone, 2018; Escobar, 2016). The introduction of Microservices changed this perspective, allowing the legacy application to be tackled in many little steps by splitting up all its functions and complexities into Microservices with fast results (Lewis, 2014; Dragoni, 2017; Thones, 2015).

Using Microservices for modern software architectures is gaining popularity and explains the need for organizations to think and act more agile and to adapt agile methodologies. In the past, a legacy system was classically developed by a large team using the waterfall model with standard project management methods. Various approaches in dealing with legacy applications exist, such as **rehosting**, **replatforming**, **repurchasing**, **refactoring/re-architecting**, **retiring** or **retaining** (Newman, 2015; Knoche H. a., 2018). These days, agile methods are required to deal with much smaller, nimble teams to implement Microservices on a smaller scope and in shorter time frame. Utilizing the advantage of being individually deployable, organizations can meet the requirements of speed and results in the Digital Transformation with Microservices easier (Bucchiarone, 2018; Pautasso, 2017).

However, many challenges are faced when attempting to move a legacy application to Microservices, including the decision to move to a cloud-native architecture (Balalaie A. a., 2016; Linthicum, 2016) or the restructuring of the codebase to clearly separate distinct business modules. In a monolithic application, the graphical user interface and the business logic are typically interdependent, something which must be considered while designing and implementing the Microservice Architecture. A common approach to deal with this is refactoring supported by tests (Bucchiarone, 2018; Taibi, 2017; Knoche H. a., 2018).

Based on industry examples we are highlighting how the move from a monolith to Microservices is conducted and what best practices can be identified to deal with the terms of technical, processual and human challenges. When done right, this approach can yield a resilient, scalable and maintainable distributed application while eliminating many of the disadvantages of legacy applications. Due to the polyglotism characteristic of Microservices, experienced and long-established software developers in an organization can still be integrated in the shift towards a modern architecture (Lewis, 2014; Balalaie A. a., 2018), without the need to learn new programming languages. Retaining the knowledge of the legacy system and eliminating the need to force out staff due to technical incompatibility, these aspects are especially important for governmental organizations.

2 Why Microservices?

In order to understand why Microservices are increasingly the means of choice for restructuring monolithic architectures, the challenges and how to overcome them can be considered from three distinctive perspectives: technical, business and human.

Applications with a monolithic structure need to be scaled horizontally and are limited to the implementation language used initially (Santis, 2016). Consequently, there are different disadvantages. On the one hand, resources are wasted unnecessarily due to the overall replication of the application. On the other hand, any complexity in the existing codebase is also replicated. Microservices can be individually scaled up and down within their architecture (Soldani, 2018). The difference in the data storage concept between monolithic and microservice-based architectures is the cornerstone of their independence and flexibility. While each Microservice stores its data on its own, monoliths rely on centralized databases. A further point that underlines the technical independence of Microservices is the fact that the programming language to be used can also be selected for each service in accordance to individual requirements (Balalaie A. a., 2016).

The various technical shortcomings reduce the scope of action from an economic point of view. The lack of agility of monolithic architectures can potentially slow down the entire further development of existing business models or the introduction of new ones. One cause is the ability to implement changed or new requirements, since these activities are increasingly complex to manage and development capacities are therefore minimized by the excessive time for implementation required (Knoche H. a., 2018). In Microservice-based applications, both the expandability and the further development through new functionality can be realized with significantly reduced effort due to clear responsibilities. In monolithic structures, the work of various teams on the same codebase results in large dependencies (Santis, 2016). Conversely, microservices are characterized by the possibility of clearly defining responsibilities with regard to the entire software development cycle, although Microservices may lead to more communication overhead if multiple services use a specific library or call a generic service.

At a time where users are more reliant on proper functioning and available digital applications, negative customer experiences such as repeated downtime can result in a loss of trust in any application. Due to the strong interdependencies, the failure of part of a monolithic application can lead to a cascade and thus to the total failure of a system. In Microservice Architectures, targeted safety mechanisms can be used to avoid such complete failures (Knoche, 2018). These lead to increased resilience. If a service fails, the others continue running and ensure the basic availability of an application. In line with the technical properties of monolithic and microservice-based architectures, the degree of documentation usually deteriorates as applications grow organically over the years and become more and more complex. Due to the smaller, individual size of each microservice-based architectures, documentation can be created more coherently and concisely and thus is easier to keep up-to-date (Santis, 2016). Another noticeable advantage is that a designated team of developers can focus on a service with a comparatively small codebase (Balalaie A. a., 2016). This supports the transfer of knowledge between different teams or the eventuality of a team being expanded or team members being exchanged. This is accompanied by the fact that monoliths do not rely on a contemporary technology stack. The retirement of long-time employees therefore causes the loss of often irretrievable knowledge that is crucial for maintaining the operation of running applications. On the other hand, it is difficult to attract young developers to advance and operate legacy systems, as technologies that are being considered as antiquated are seldom taught to students anymore and their use is seen as neither future-proof nor career-promoting (Knoche, 2018).

3 Practical Approaches – turning the mountain into a prophet

There are several reasons organisations choose to move away from their legacy system. Ranging from high maintenance cost and efforts, dwindling staff with skills in COBOL and other ancient programming languages, to high mainframe license costs. Although the intent of moving away from a legacy system is clear, it often fails due to the practical realization where organisations struggle to keep operations running while shifting to new system. Various approaches such as **new development, replatforming, refactoring** or **code transformation** each have their own strengths and weaknesses. There is no right or wrong, it depends on the expectations in terms of migration costs, project run time, risks management and how to deal with technical debt. A combination of different approaches is a suitable way to approach the complex matter of migrating a legacy system. Additionally, it must be decided if the migration will be conducted incrementally or in a big bang. While a big bang migration, replaces the legacy system in one swoop, an incremental migration does replace part of the systems step by step. The figure below illustrates potential solution patterns to combine multiple approaches in an incremental migration.

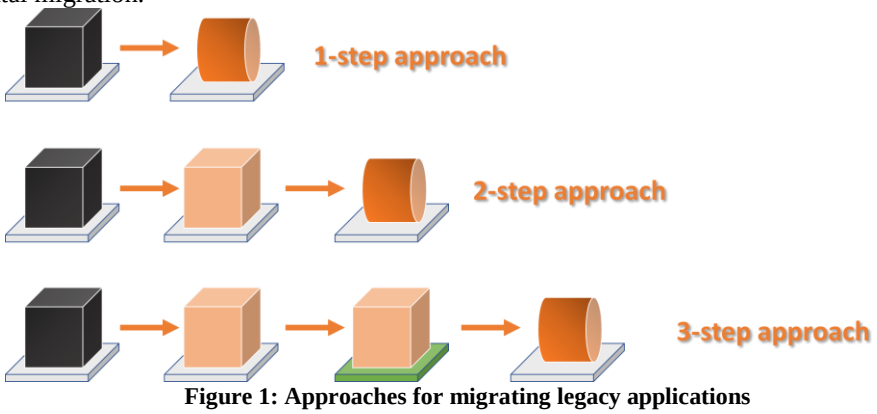

Figure 1: Approaches for migrating legacy applications

3.1 Transformers and Combiners – more than meets the eye

In 2015, a large government body in North Rhine-Westphalia, Germany, decided to retire its legacy system and mainframe environment by the end of 2018, to save in licensing and maintenance costs, as well as to prevent the total loss of the legacy system knowledge due to the aging workforce. Starting out with a feasibility study, the government body identified which applications can be retired, replaced or must be migrated. Part of the study was the analysis of the most suitable approaches for migrating the legacy applications. For example, an incremental transformation approach was chosen to replace the core invoicing application, with zero down time as a requirement. The new Microservice Architecture for this application is process and service oriented, utilises Web- and Java technologies and features a new User Interface for staff, eliminating the old-fashioned host panels for data entry. The transformation was carried out in six steps while establishing change management and knowledge transfer through the whole process.

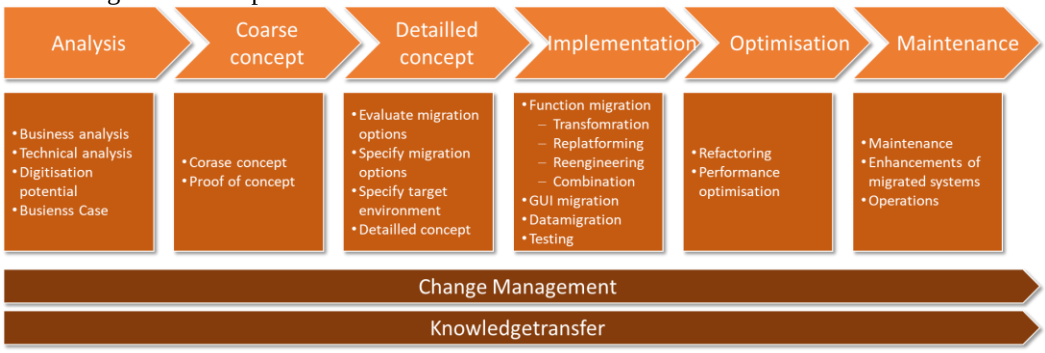

Figure 2: 6 step process for migration a legacy application

The actual migration was conducted in the following way, splitting up the legacy application into business function blocks and sorting them into a predetermined order when the blocks will be available in the new target environment. The optimal approach for each migrating each block was chosen, by either developing it new and realising them as Microservices or by transforming the existing code using a transformer solution such as adesso´s transformer. Utilising a transformer enables an automatic migration of legacy code such as Cobol or PL/I to Java on the target platform without impacting on its existing functionality, while performing integrated automated tests. By combining different approaches and using a Microservice Architecture (see figure below), the complex tasks of migrating a legacy system become more manageable due to the separation of individual functional blocks which could be migrated incrementally without any down time. The combined approach and the Microservice Architecture allowed existing staff to be involved in the process without the need to learn

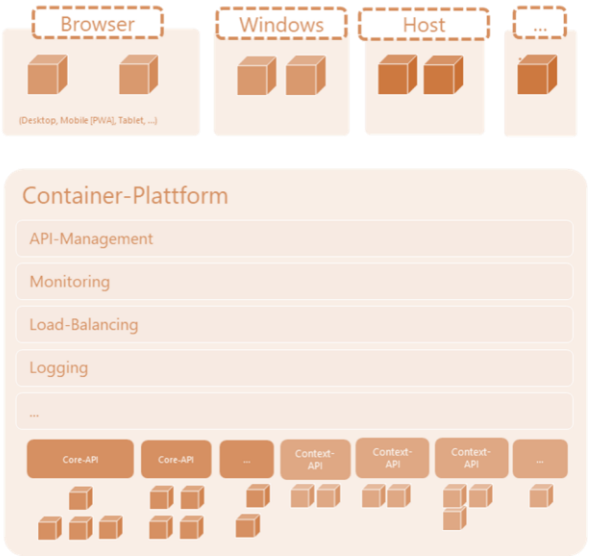

Figure 3: Microservice Architecture

a new programming language, focussing on passing on the system and business knowledge to other colleagues.

3.2 Refactoring – Trimming the fat

Many enterprise applications that were well designed have grown into monoliths over time which are hard to maintain, and even harder to adapt to changing requirements. One approach to modernize such a legacy application is to refactor it in accordance with business module boundaries. Regularly conducted acceptance tests are an essential tool to ascertain the application's functional requirements are met at all times. Through incremental steps, the monolithic application is transformed into a distributed application.

The starting point and most important part is identifying business module boundaries within the application (Evans, 2004; Wolff, 2015) to get a basis for refactoring. This is an interdisciplinary task requiring developers and business analysts to discuss the business domains of the enterprise application. Each business domain corresponds to a bounded context (Evans, 2004). The aforementioned stakeholders need to agree on a ubiquitous language (Evans, 2004), to facilitate writing acceptance tests for each business domain identified in the process. The acceptance tests provide the basis for incremental refactoring and extracting functionality corresponding to a bounded context, as a microservice. For this process to be most effective, it is important that developers are organized according to business domain, rather than the usual technical split according to application layers (user interface, business logic, database). A development team is first assigned to be responsible for a specific business domain, then the team refactors the codebase in order to modularize all the code that supports the bounded context of the domain. Afterwards, the resulting module can be turned into a microservice. Once this loose coupling has been established, modern programming language expertise present within the development team may be leveraged to convert the microservice into a more maintainable state. Additionally, architectural analysis tools like Structure101 (Structure 101, 2019) are a valuable asset in the ongoing refactoring process.

In general, several domain developer teams may simultaneously refactor the codebase, resolving code conflicts as they arise. Also, product managers need to be aware of the impact new requirements have on the whole restructuring process. This test driven and domain driven approach to transforming a monolithic enterprise application into microservices can yield a satisfactory result, provided developers and business analysts receive a sufficient amount of budget and time.

3.3 Low Code/Model-Driven Approach - Microservices in the middle

A different alternative approach for moving a monolithic system is to utilize a low code/model driven platforms (such a Appian, Vantiq, Mendix or K2) in a Microservice architecture. In recent years the popularity and use of such platforms has increased due to their advantages in improving agility, decreased costs, higher productivity, adaptability and faster transformation. In comparison to a classic transformer approach such as in 3.1, this approach does not transform existing code, it provides the means to generate software applications easily and without the effort of coding everything. Model driven software development (MDSD) platforms are a variant of such low code platforms focusing on generating software by the process models defined and configured in the platform. These platforms are the new middleware, utilizing Microservices to bridge the gap between applications on end user devices and the connectivity to ERP, legacy and cloud systems. The figure below illustrates the ORGENIC platform, an upcoming model driven software development platform. Existing functions of a legacy system as well as new functions can be modelled in the platform which transforms them automatically

into applications for the end user, while ensuring data transfer to the legacy system using Mircoservices. This model driven software development approach for migrating and enhancing functions of a legacy system is currently used in the Chamber of Commerce environment in Germany. It is useful, especially when delivering the required speed in Digital Transformation, giving the end user fresh applications with all required functionality without the need to change the legacy system. A potential migration approach is to model all required functions of the legacy application as well as new ones in the MDSD platform and store the data within or outside of the legacy environment. This ensures the legacy application can either be migrated incrementally or in a big bang fashion while keeping the efforts of actual coding low.

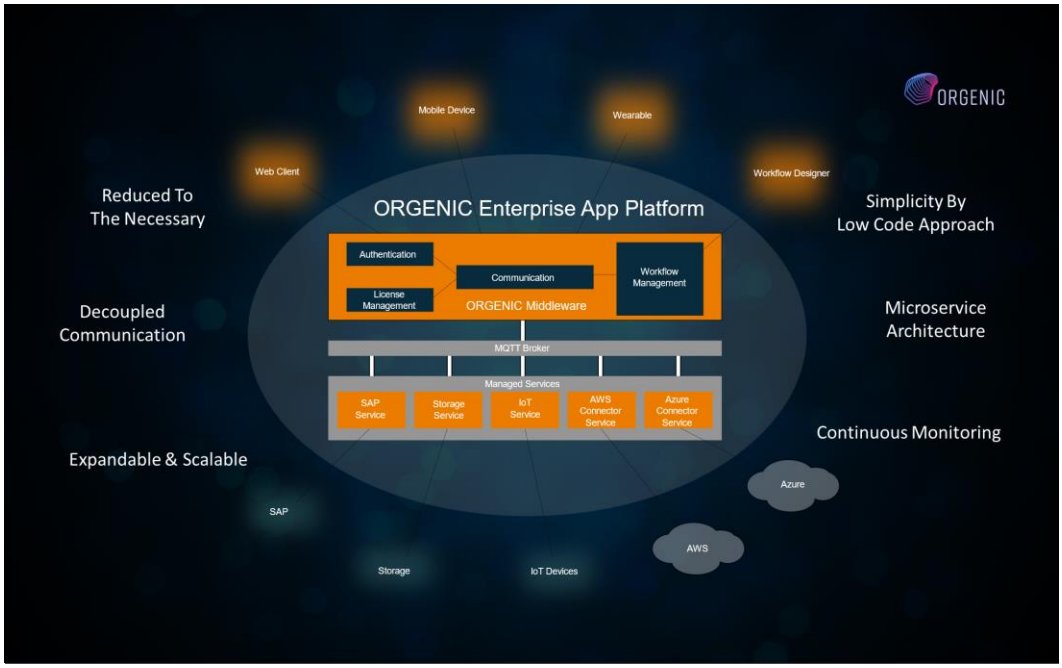

Figure 4: Model Driven Software Development - ORGENIC Platform

4 Conclusion

The once impossible task of migrating a legacy system has been made easier and foremost possible through the advances in software development. The principles of Microservice Architectures are a great fit for dealing with a monolithic system, splitting it up into tiny manageable blocks and utilising the advantages of Microservices. Although various approaches exist to migrate a legacy system, it always depends on the motivation of leaving the legacy environment and how the cost and time constraints impact the migration move. The advent of low code in combination with Microservices will lead to further increases in speed and functionalities for moving mountains.

References

Balalaie, A. a. (2016). Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software, 33 (3)*, pp. 42-52.

Balalaie, A. a. (2018). Microservices migration patterns. *Software: Practice and Experience, 48, (11)*, pp. 2019-2042.

Bucchiarone, A. a. (2018). From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software, 35, (3)*, pp. 50-55.

Dragoni, N. a. (2017). Microservices: yesterday, today, and tomorrow. In *Present and Ulterior Software Engineering* (pp. 195-216). Springer.

Escobar, D. a. (2016). Towards the understanding and evolution of monolithic applications as microservices. *Computing Conference (CLEI), 2016 XLII Latin American* (pp. 1-11). IEEE.

Evans, E. (2004). *Domain Driven Design.* Addison Wesley, p . 24-27, 335-337.

Knoche, H. a. (2018). Using Microservices for Legacy Software Modernization. *IEEE Software, 35 (3)*, pp. 44-49.

Lewis, J. a. (2014). Microservices: a definition of this new architectural term. *Mars*.

Linthicum, D. S. (2016). Practical use of microservices in moving workloads to the cloud. *IEEE Cloud Computing, 3, (5)*, pp. 6-9.

Newman, S. (2015). *Building microservices: designing fine-grained systems.* O'Reilly Media, Inc.

Pautasso, C. a. (2017). Microservices in Practice, Part 1: Reality Check and Service Design. *IEEE Software, 34, (1)*, pp. 91-98.

Santis, S. d. (2016). *Evolve the monolith to microservices with Java and Node, Poughkeepsie.* NY: IBM Corporation International Technical Support Organization.

Soldani, J. T.-J. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software,*, vol. 146, pp. 215–232.

Structure 101. (2019, 01 15). *Structure 101*. Retrieved from https://structure101.com

Taibi, D. a. (2017). Processes, motivations, and issues for migrating to microservices architectures: an empirical investigation. *IEEE Cloud Computing, (5)*, pp. 22-32.

Thones, J. (2015). Microservices. *IEEE software, 32, (1)*, p. 116.

Wolff, E. (2015). *Microservices.* dpunkt.verlag, p 44-46.