Software and Cloud Technology: Transforming a Monolithic Application into Microservices

Viktor Meyer - viktorme@kth.se | David Ljunggren - dljungg@kth.se
KTH Royal Institute of Technology, Stockholm, Sweden

Abstract
This essay reviews challenges that organizations might face when migrating from a monolithic architecture to microservices. It provides a detailed background of software architectural patterns and the modern cloud environment as a force multiplier. More importantly, it highlights microservices with its benefits of scalability and loose coupling, at the same time as it illustrates various drawbacks such as initial costs and complexity. Finally, the essay reviews strategies for addressing migration challenges.

** We certify that generative AI, incl. ChatGPT, has not been used to write this essay. Using generative AI without permission is considered academic misconduct.*

CCS Concepts
• Software and its engineering $\rightarrow$ ***Software design tradeoffs; • Networks*** $\rightarrow$ *Cloud computing;*

1. Introduction

Software architecture is a field which has garnered an increasing amount of attention in recent years. A key concept within software architecture is that of architectural patterns. *Architectural patterns* provide proven solutions to architectural issues and aid in the documentation of architectural design choices [AZ05]. They also ease communication with key stakeholders through common vocabulary, and help characterize the qualities of a system [Kas+18]. Choosing the appropriate architecture for the problem at hand is thus an important factor. However, it is not a simple task. Correctly predicting the scale, complexity, and intricacies of software during the early stages of development is non-trivial.

With the advent of cloud computing, the maintenance, deployment and usage of software has seen new possibilities. Distributed architectural patterns such as the microservice architecture has gained a sharp increase in popularity [Guo+16]. Despite some clear advantages of fine-grained distributed architectures, this short essay illustrates that there is not a one universal pattern that is without its drawbacks. Instead, the choice is largely nuanced.

The problem of architectural pattern choice is multi-faceted with almost every architecture having its own bespoke advantages and disadvantages. However, this is only part of the issues that engineering organizations face in practice. Indeed, it is common for organizations to start their projects using one architectural pattern; only to later find out that the pattern is bad fit [TLP18]. To some extent, this is unavoidable, whether something works in theory can be very different from practice. This essay aims to illustrate challenges that arise when an organization has a need to transform from one architectural pattern to another. Specifically, we intend to highlight considerations and best practices when migrating from a monolithic into a microservice architecture.

2. Problem

It is a nuanced and complex process to migrate from an already adopted architectural pattern into a new one. There is extensive material on the advantages and challenges of many popular architectural patterns. However, the considerations around specific migration *processes* from $Pattern_A$ to $Pattern_B$ is hard to find.

3. Background

The background introduces three common architectural patterns: *Monolith Architecture*, *Distributed Monolith Architecture*, and *Microservice Architecture*. In addition, a brief overview follows for the modern cloud environment and the capabilities it unlocks.

3.1. Architectural patterns

Architectural patterns serve to characterize a set of high-level design decisions that embody a software system [KEM12]. Software architects value the following factors when choosing an architectural pattern: Usability, Testability, Security, Performance, Modifiability, Interopability, and Availability [Kas+18]. *Coupling* is a central measure to these factors, and approximates the extent to which a piece of software module depends on another [RPL08].

For example, correctly implemented *loosely coupled* architectures tend to be easier to modify since its modules can be worked on independently [Mah07]. Additionally, they may provide superior scalability with system parts being independently scalable. By the same token, it is fairly straightforward to see that a *closely coupled* architecture will elicit barriers to scalability where the only choice is to scale the whole application at the instance level [PMA19].

3.1.1. Monolith Architecture

A *monolith* can be characterized as a software application whose modules can not be logically executed independently [Dra+17].

The monolithic architectural pattern prevents independent module execution by encapsulating all features into a single application. This kind of design is closely coupled, and each piece of logic that handles a request is handled by the same process [PMA19]. Monolithic applications make up for the majority of software, and can be seen as the traditional way of modelling software systems.

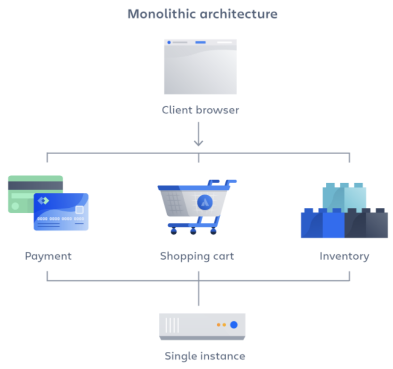

Figure 1: *All logical functionality is served and contained by a single instance definition in a monolithic architecture [Cha23].*

The most striking advantage of the monolithic pattern is the initial simplicity with regard to development [New19]. There is less need for strategic planning and execution with the monolithic approach as all functionality that the program offers is within the same codebase. Furthermore, since the entire application is served through a single process, there is no need for different modules of the software program to exchange data through APIs. I.e., network communication, latency, and security considerations are less of a concern when compared to other more distributed pattern choices.

In recent years, the disadvantages of the monolithic architectural pattern have become more evident [New19]. Specifically, when a monolithic system grows, its complexity increases significantly. This can make larger projects exceedingly difficult to manage and develop efficiently. This may be attributed to the patterns innate close coupling, whenever a small change is done to a monolithic codebase, it has the potential to alter the entire software program. Additionally, the coupling characteristics makes it hard to effectively scale performance as it must often be done vertically [†].

3.1.2. Microservice Architecture

A *microservice architecture* can be defined as a distributed system where all its modules can execute independently of each-other [Dra+17]. In other terms, the architecture aims to separate functions and processes that can run independently of each other, and gather those that do not into its own deployable module.

[†] Vertical scaling entails adding additional compute resources to a single node, as opposed to adding more cluster nodes with equal compute [ME20].

A practical example of the microservice pattern could be an E-commerce website that separates its search, shopping, and payment systems from each other [Cha23]. Such a design does not require that the separated systems have zero interaction, but they should be largely self-contained and independently deployable.

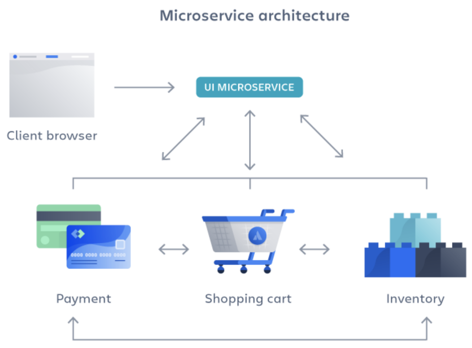

Figure 2: *A microservice architecture separates functionality across multiple independently deployed service instances [Cha23].*

There are various advantages to the microservices architectural pattern. One of the most significant advantages is developer scalability [Tai+17]. Operations such as adding, removing or updating a microservice can largely be carried out by developers without interfering or interrupting any other services. Since the microservices are isolated, they can also be scaled for performance by business needs as necessary. Another advantage of the microservice architectural pattern is its enhanced fault isolation. If one part of the system fails, it is not as likely that the entire application is affected as compared to monolithic applications.

Disadvantages include deployment complexity and costly initial development which typically requires senior developers with vast domain experience [Tai+17]. Furthermore, while microservice architectures tend to be less complex than their monolith counterparts; they can still grow complex as system size increases.

3.1.3. Distributed Monolith Architecture

A *distributed monolith* architecture is one that consists of multiple services that for some reason must be deployed together [New19].

The distributed monolith shares a lot of characteristics with the microservice pattern. The major difference is that for a distributed monolith, the services are not independently deployable [New19]. This in itself is a form of coupling which means that the distributed monolith pattern elicits a higher degree of coupling.

This higher degree of coupling may seem innocent at first but the reality is that it negates some of the core selling points otherwise found in microservices [New19]. For example, since there are logical dependencies between services, updating a microservice can no longer be carried out by developers without interrupting its dependent services. In a way, the distributed monolith can be described as an unwanted side effect of poorly implemented microservices rather than a planned architecture. It often bears the disadvantages of both microservices and the monolith, all while failing to deliver on the significant upsides of either pattern.

3.2. Cloud Computing

Cloud computing is a term used to describe both a platform and a type of application [Bos+07]. A cloud computing platform provisions, configures, reconfigures and deprovisions servers as required, and a cloud server can be either a physical or a virtual device. The term commonly refers to the on-demand availability and configurability of data storage, networking, and compute.

3.2.1. Widely Accessible

Modern cloud computing platforms provide state-of-the-art infrastructure and services at the push of a button [SR19]. Public cloud computing offerings enable large organizations as well as single individuals to benefit from highly capable infrastructure. In many instances, this makes the traditionally high costs of private self-hosted data centers a thing of the past for businesses.

In a sense, modern cloud platforms commoditise processing power into a resource that can be easily traded for currency, e.g., United States Dollar. Indeed, this is very attractive since it allows users to self-service and pay only for the cloud services as they are used.

4. Key Considerations for Architecture Migration

The microservice architectural pattern often offers more attractive characteristics compared to the monolith architectural pattern (see Section 3.1). Even if this is true in general, it is not uncommon for the architectural pattern preference and fit to change over time as an engineering organization matures [Kas+18]. E.g., software projects may choose the monolith architecture to quickly get up and running, and later find that they need to switch to a microservice approach. The question then becomes: what considerations need to be accounted for when considering a migration from monolith to microservices? What processes goes into a practical migration?

Sam Newman's seminal literature *"Monolith to Microservices"* outlines specific considerations for migrating from monolith to microservices [New19]. It argues that if there are other less extreme options that does not entail migrating to a new architectural pattern, those are worth considering first. There can be a very real cost of change to making larger architectural changes. It is common for organizations to consider microservices with the intent of: Improving Team Autonomy, Reducing Time to Market, Scaling Cost-Effectively for Load, Improving Robustness, Scaling the Number of Developers, and Embracing New Patterns. Newman emphasizes that the microservice architecture itself is never the goal, instead, organizations should value the associated outcomes. With such a realization in mind, that outcomes are what is valuable, *it may be the case that many of the outcomes can be attained through other means that a full architecture migration*.

As for the migration process itself, Newman suggests that the architecture migration process must not be underestimated in complexity [New19]. The recommended approach is to within an organization: Establish a Sense of Urgency, Create a Guiding Coalition, Communicate the Change Vision, and Empower Employees for Broad-Based Action. Further emphasis is put on adopting a incremental adoption strategy where it is not a binary switch with a single deployment that changes from monolith to microservices. Instead, *a monolith should be piece-wise split into smaller services by domain that can co-exist with the monolith.*

The concept of separating functionality by domain is known as Domain Driven Design (DDD) and lends itself especially well to microservices [EE04]. It decouples functionality into its own isolated units, which has the advantage of allowing developers to look at smaller pieces of software instead of getting lost in what may otherwise be a very large codebase. *If domain boundaries are correctly identified, it means that the services themselves are loosely coupled and can be operated on independently*. Isolating domain boundaries is out of this essay's scope, we refer to [EE04].

In the paper *"Processes, Motivations, and Issues for Migrating to Microservices Architectures: An Empirical Investigation"* by Davide Taibi, Valentina Lunarduzzi and Claus Pahl, the authors conclude that issues related to migration can be divided into three primary categories [TLP17]. Those three categories are *technical issues*, *economical issues*, and finally *psychological issues*. In the technical category, database migration along with data splitting was deemed to be a consideration. The paper recommends splitting existing databases such that each microservice accesses its own private database for loose coupling. Another technical consideration is the increased complexity that comes with setting up the orchestration layer for microservice operation. Economical issues in the paper are mainly seen as a result of technical issues. Psychological considerations include conservatism of more senior software developers when it comes to adopting new technology.

5. Conclusion

This essay has illustrated that the considerations that go into the migration process from monolith to microservices is complex. In particular, it has identified some of the key considerations when performing an architecture migration. The complexity that comes with migration should not be underestimated, and migrating should be avoided if there are simpler options at hand that yield similar outcomes. If the only choice is to perform a migration, the migration should be split into incremental steps and be conducted by domain rather than be implemented as single binary step.

References

[EE04] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[AZ05] Paris Avgeriou and Uwe Zdun. "Architectural patterns revisited-a pattern language". In: (2005).

[Bos+07] Greg Boss et al. "Cloud computing". In: *IBM white paper* 321 (2007), pp. 224–231.

[Mah07] Zaigham Mahmood. "Service oriented architecture: potential benefits and challenges". In: *Proceedings of the 11th WSEAS International Conference on COMPUTERS*. Citeseer. 2007, pp. 497–501.

[RPL08] Romain Robbes, Damien Pollet, and Michele Lanza. "Logical coupling based on fine-grained change information". In: *2008 15th Working Conference on Reverse Engineering*. IEEE. 2008, pp. 42–46.

[KEM12] Mohamad Kassab, Ghizlane El-Boussaidi, and Hafedh Mili. “A quantitative evaluation of the impact of architectural patterns on quality requirements”. In: *Software Engineering Research, Management and Applications 2011* (2012), pp. 173–184.

[Guo+16] Dong Guo et al. “Microservices architecture based cloudware deployment platform for service computing”. In: *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. IEEE. 2016, pp. 358–363.

[Dra+17] Nicola Dragoni et al. “Microservices: yesterday, today, and tomorrow”. In: *Present and ulterior software engineering* (2017), pp. 195–216.

[TLP17] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. “Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 22–32.

[Tai+17] Davide Taibi et al. “Microservices in Agile Software Development: A Workshop-Based Study into Issues, Advantages, and Disadvantages”. In: *Proceedings of the XP2017 Scientific Workshops*. XP ’17. Cologne, Germany: Association for Computing Machinery, 2017. ISBN: 9781450352642.

[Kas+18] Mohamad Kassab et al. “Software architectural patterns in practice: an empirical study”. In: *Innovations in Systems and Software Engineering* 14 (2018), pp. 263–271.

[TLP18] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. “Architectural patterns for microservices: a systematic mapping study”. In: *CLOSER 2018: Proceedings of the 8th International Conference on Cloud Computing and Services Science; Funchal, Madeira, Portugal, 19-21 March 2018*. SciTePress. 2018.

[New19] Sam Newman. *Monolith to microservices: evolutionary patterns to transform your monolith*. O’Reilly Media, 2019.

[PMA19] Francisco Ponce, Gastón Márquez, and Hernán Astudillo. “Migrating from monolithic architecture to microservices: A Rapid Review”. In: *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. 2019, pp. 1–7.

[SR19] Jayachander Surbiryala and Chunming Rong. “Cloud computing: History and overview”. In: *2019 IEEE Cloud Summit*. IEEE. 2019, pp. 1–7.

[ME20] Victor Millnert and Johan Eker. “HoloScale: horizontal and vertical scaling of cloud resources”. In: *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 2020.

[Cha23] Chandler Harris. *Microservices vs. Monolithic Architecture*. https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. Online; accessed 29 March 2023. 2023.