Kaarel Saal

APPLYING MICROSERVICE ARCHITECTURE TO DEVELOPING EVOLVABLE SYSTEMS IN SMALL TO MEDIUM SIZED SOFTWARE PROJECTS

MBA Thesis

Supervisor: Inna Švartsman, MSc

Tallinn 2017
Microservices have emerged as a software architectural style comprised of multiple fine-grained self-contained components. Several industry case studies and research papers have focussed on the implementation of microservice architecture in the context of large organizations and large software projects. Thus when microservices are discussed it is often cautioned that the benefits come with considerable downsides, leaving the overall impression that only the very large and resourceful organizations are able to benefit from implementing this architectural style. However, there seems to be a gap in the literature investigating the microservice architectures in the context of small to medium software projects. Our work aims to fill that gap and explore the applicability of microservices for smaller software projects. The findings will be of direct interest to technology start-ups and teams within a larger organization.

We will be using a case study of a real-life project and focus on how applying microservice architecture to an existing monolithic software affects the evolvability characteristic of the system. To quantify evolvability as the system’s ability to adapt to change we look at changes in two key areas – complexity and the rate of change of a software system. When measuring complexity we’ll be using the metrics Cyclomatic Complexity and Cognitive Complexity. To quantify the rate of change we’ll be using a combination of two metrics – the number of product deployments and the number of source code commits.

We observe that the complexity of the system on average becomes more manageable as we move from a single large monolithic code base towards smaller individually maintainable services. We also observe the increase in the rate of change of the overall system. Both of these observations contribute towards improving the evolvability of a system. By applying educated compromises microservice architecture can be beneficial for small software projects.
RESÜMEE


Töö tulemusena täheidame, et ühe monoliitse rakenduse ümber disainimine mitmest väiksemast rakendusest koosnevaks süsteemiks muudab tarkvara süsteemi kui terviku keerukus hallatavamaks. Samuti suurenes muudatuste rakendamise sagedus. Mõlema tähendamise põhjal on võimalik kasu saada ka väiksematel tarkvara projektidel.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>2</td>
</tr>
<tr>
<td>RESUMEE</td>
<td>3</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>4</td>
</tr>
<tr>
<td>LIST OF FIGURES AND TABLES</td>
<td>6</td>
</tr>
<tr>
<td>GLOSSARY AND TERMS</td>
<td>7</td>
</tr>
<tr>
<td>1. INTRODUCTION</td>
<td>9</td>
</tr>
<tr>
<td>1.1. Research Problem</td>
<td>9</td>
</tr>
<tr>
<td>1.2. Related Work</td>
<td>11</td>
</tr>
<tr>
<td>1.3. Hypothesis</td>
<td>12</td>
</tr>
<tr>
<td>1.4. Purpose</td>
<td>12</td>
</tr>
<tr>
<td>1.5. Research Approach</td>
<td>13</td>
</tr>
<tr>
<td>2. BACKGROUND</td>
<td>14</td>
</tr>
<tr>
<td>2.1. Monolith Architecture</td>
<td>14</td>
</tr>
<tr>
<td>2.2. Microservice Architecture</td>
<td>15</td>
</tr>
<tr>
<td>2.2.1. Characteristics of a Microservice</td>
<td>17</td>
</tr>
<tr>
<td>2.2.2. Technical Benefits</td>
<td>18</td>
</tr>
<tr>
<td>2.2.3. Business Value</td>
<td>20</td>
</tr>
<tr>
<td>2.2.4. Challenges</td>
<td>22</td>
</tr>
<tr>
<td>2.3. Cloud Computing</td>
<td>23</td>
</tr>
<tr>
<td>2.4. Monolith First or Microservices First</td>
<td>24</td>
</tr>
<tr>
<td>2.5. Refactoring Monolith to Microservices</td>
<td>26</td>
</tr>
<tr>
<td>2.6. Alternatives to Microservice Architecture</td>
<td>27</td>
</tr>
<tr>
<td>2.7. Existing Case Studies</td>
<td>28</td>
</tr>
<tr>
<td>3. CASE STUDY</td>
<td>30</td>
</tr>
<tr>
<td>3.1. Motivation</td>
<td>31</td>
</tr>
<tr>
<td>3.2. Analysis of the Existing System</td>
<td>34</td>
</tr>
<tr>
<td>3.3. Deployment Pipeline</td>
<td>36</td>
</tr>
<tr>
<td>3.4. Development Workflow</td>
<td>36</td>
</tr>
<tr>
<td>3.5. Proposed Solution</td>
<td>37</td>
</tr>
<tr>
<td>3.6. Practical Implementation</td>
<td>39</td>
</tr>
<tr>
<td>4. ANALYSIS</td>
<td>42</td>
</tr>
<tr>
<td>4.1. Key Performance Indicators</td>
<td>42</td>
</tr>
<tr>
<td>4.2. The Good</td>
<td>45</td>
</tr>
<tr>
<td>4.3. The Bad</td>
<td>46</td>
</tr>
</tbody>
</table>
4.4. The Future ........................................................................................................................................... 47

5. CONCLUSION AND RECOMMENDATIONS ..................................................................................... 50
  5.1. Conclusions ...................................................................................................................................... 50
  5.2. Recommendations ............................................................................................................................ 51
  5.3. Future Work ...................................................................................................................................... 52

REFERENCES ........................................................................................................................................... 53

APPENDIX 1. PRODUCT DASHBOARD EXAMPLES ............................................................................. 57
APPENDIX 2. DATA COLLECTION: PRODUCT DEPLOYMENTS ......................................................... 58
APPENDIX 3. DATA COLLECTION: PRODUCT CHANGES ..................................................................... 59
APPENDIX 4. DATA COLLECTION: COMPLEXITY ................................................................................ 60
APPENDIX 5. GUESTJOY EXPERIENCE REPORT .................................................................................. 61
LIST OF FIGURES AND TABLES

Figure 1. Microservice premium ................................................................. 10
Figure 2. The isolated ideal vs tightly coupled reality of monolith architecture ........ 25
Figure 3. Keeping database schemas separate for different modules ...................... 27
Figure 4. Guestjoy journey ....................................................................... 30
Figure 5. Monolith architecture .................................................................. 31
Figure 6. Hybrid system: an intermediate state between monolith and microservice architecture .............................................................................. 38
Figure 7. Hybrid system: an actual intermediate state after parts of the monolith had been broken up ........................................................................ 40
Figure 8. AWS Load Balancer configuration ...................................................... 41
Figure 9. Product deployments before and after applying microservice architecture .... 43
Figure 10. Product changes (code commits) before and after applying microservice architecture .............................................................................. 44
Figure 11. A possible microservice architecture of a future Guestjoy system ........... 48
Table 1. Product complexity before and after applying microservice architecture ..... 44
GLOSSARY AND TERMS

AWS – Amazon Web Services, a provider of computing infrastructure in the cloud, data storage services, web application deployment location and other supporting cloud services.

Big ball of mud – an expression used to describe a software system that lacks a perceivable architecture. Often a result of time pressure. A type of software anti-pattern that hinders system’s evolvability by increasing complexity and slowing down development.

CloudWatch – an AWS service specifically designed to store, search and visualise events occurring in a software system. Useful for monitoring and troubleshooting a system.

Complexity – characterises the difficulty of understanding and changing a system due to the number of interaction points within the system.

Cyclomatic Complexity – a software metric used to measure the complexity of a software system using a mathematical model.

Cognitive Complexity - a software metric used to measure the relative difficulty of cognitively understanding software systems.

CSV – Comma Separated Values, a common data exchange format.

Docker – one of the tools commonly used to help isolate services in a microservice architecture. A specific implementation of a software container.

Evolvability – the ability of a system to adapt to change.

Git – a tool to track changes made in software.

HTTP – Hypertext Transfer Protocol, a set of rules that regulate how computers communicate with each other. Commonly used in microservice architecture for communication between services.

JavaScript – a formal language used to build software systems.

JSON – JavaScript Object Notation, a common data exchange format.

Load balancer – a tool to distribute work among workers, commonly used to distribute messages to services in a microservice architecture.
**Microservice** – a software application providing a single business functionality or capability.

**Microservice Architecture** – an architectural style providing guidance on how to design individual microservices that work together as a single software system, with the emphasis on the scope of an individual service.

**Monolith** – an architectural style where all various, even unrelated, business capabilities are provided by a single monolithic service.

**SOA** – Service-Oriented Architecture, an architectural style that more broadly describes a system comprised of services.

**Software container** – an abstraction of computer resources, narrower and more lightweight compared to virtual machines. Often used to host individual microservices in a microservice architecture.

**UI** – User Interface, (often visual) means of interacting with a computer system or device.

**Virtual machine** – an abstraction of computer resources. Often to host systems designed with microservice architecture.

**WS-*** – used to refer to a collection of web service standards often associated with implementations of SOA.

**XML** – Extensible Markup Language, a common data exchange format.
1. INTRODUCTION

“Every budget is an IT budget. Every company is an IT company. Every business leader is becoming a digital leader.”

– Stephen Prentice, Vice President of Gartner (Life in Estonia, 2014)

A similar idea is echoed by Marek Kiisa who has described his women’s lingerie manufacturing company as an “IT company that additionally happens to be great at sewing” (Gabral, 2017). Numerous sources have documented how intimately interwoven technology and business are at every level of an organization. Information technology can affect companies by having a flattening effect on the structure (organisation chart) of an organisation (Laudon & Laudon, 2014, p. 120). At the same time, the architecture of an organisation affects the architecture of the product they are building as observed by Conway’s Law: “organizations which design systems (in the broad sense used here) are constrained to produce designs which are copies of the communication structures of these organizations” (Conway, 1968). Technology matters. Technology choices matter.

Microservice architecture is a recently emerging style of designing and evolving software systems. It is usually discussed in the context of very large systems tackling the problem of massive user base and scale (Richardson, Who is using microservices?, n.d.). Thus a large body of literature and industry conferences concentrate on the problems of and benefits for very large organizations leaving some authors to suggest “that most small companies wouldn't necessarily benefit from adopting microservice architecture” (Fowler S., 2017).

1.1. Research Problem

Even though industry experts agree that using a microservice architecture adds complexity to a software solution, the general consensus seems to be that eventually at some point in a software project’s lifetime, once it becomes big and complex enough, microservice architecture is a more forward-looking approach than a single monolith application as summarised by one of the industry thought leaders (Fowler M., Microservice Premium,
2015) “As size and other complexity boosters kick into a project I've seen many teams find that microservices are a better place to be”.

A definitive consensus does not yet exist whether a software project should start with a monolith architecture or with microservice architecture. If microservice architecture is the preferred end goal for a successful software project then why not do it right in the first place. This work will give an overview of the existing discussions in the literature on the topic but will not attempt to solve this problem. It has been observed that at the beginning of a software project development teams demonstrate greater productivity when using a monolith approach compared to a microservice architecture. The observation how microservice architecture increases complexity at the beginning of a project in comparison to monolith architecture is illustrated in Figure 1 below. As the project evolves, however, this difference in productivity and complexity between a monolith and microservices approach is expected to flip.

Figure 1. Microservice premium
Source: (Fowler M., Microservice Premium, 2015)
An area we are interested in with our work is the applicability of microservice architecture to small software projects. One of the most pressing challenges in software development is the ability to adequately respond to change. In this paper, we call this the **evolvability** characteristic of a software system. In the words of Dr. John Christian evolvability is “the capacity of a system to successfully adapt to changing requirements throughout its lifecycle without compromising architectural integrity” (A Quantitative Approach to Assessing System Evolvability, 2004). Microservices have many benefits that improve software system’s ability to adapt to changes. Change in this context can be new user requirements or a new business direction. Clearly the faster a business can adapt to market changes the greater its competitive advantage. Closely and interrelated to evolvability is the **complexity** of a software system. “Managing complexity is the most important technical topic in software development. In my view, it’s so important that Software’s Primary Technical Imperative has to be managing complexity.” (McConnell, 2004, p. 78). The more complex a system is the more difficult it is to understand and reason about and therefore make safe changes to it. Complexity, as defined in the vocabulary of international standard for systems and software engineering, is “the degree to which a system's design or code is difficult to understand because of numerous components or relationships among components” (IEEE, 2010). This applies well to software projects and describes almost perfectly the problem with monolith systems. As Christian continues “an evolvable system must meet the new needs of the customer in a more cost effective manner than developing a new system”. This is one of the reasons legacy software often gets completely rewritten at a significant cost. Because it would cost even more to change the existing software.

### 1.2. Related Work

The term “microservice” was first used in 2011 (Fowler & Lewis, Microservices, 2014). According to (Pahl & Jamshidi, 2016) the term has been consistently used from 2014. Their 2015 meta-analysis of previously published studies identified just 21 relevant publications of which two were journal publications, three were an academic thesis and majority conferences and workshops. The analysis concluded that research on microservices is still in a formative stage, suggesting that “more experimental and empirical evaluation of the
benefits is needed”. A similar view has been expressed by industry thought leaders Martin Fowler and James Lewis:

“When our experiences so far are positive compared to monolithic applications, we're conscious of the fact that not enough time has passed for us to make a full judgement.” (Fowler & Lewis, Microservices, 2014).

1.3. Hypothesis

In this paper, we put forward a hypothesis that using modern tools and concepts the challenges introduced by microservice architecture are reduced to the point where it is practical to apply microservice architecture to small software projects.

By modern tools and concepts, we predominantly mean cloud infrastructure, virtualization, automation etc.

By practical we mean that the cost of changing an existing system is less than construction a new system as defined in (Christian, 2004).

By a small software project we mean ~5 distinct, individually developed and deployed services working together as a system.

1.4. Purpose

Successful case studies of applying microservice architecture exist but they tend to focus on large-scale systems. Our work will explore the applicability of the microservices approach to software development in small software projects. For our current work, we do not need to define a hard line between a small and a large software project. We are also not interested in the size as in lines of code because one can have a service filling a single business function using just a few hundred lines or several thousand. We are interested in the size of a software project in terms of the number of distinct functional capabilities it has, each of which can be mapped to a microservice. Case studies showing the usefulness of microservice architecture at 100 plus distinct services exist. Instead of concerning ourselves at which number of services small system becomes large, we will simply demonstrate the practicality of this
approach with a project consisting of five services. By service, we mean microservice, the definition of which we will be discussing at a greater length later in this paper.

Following the conclusions by (Pahl & Jamshidi, 2016) in their research that more experimental evaluation of microservices is needed the work presented in this paper explores the applicability of microservice architecture for small to medium sized software projects. Specifically the benefits and drawbacks of implementing such solutions at smaller scale. This should be of heightened interest to small software teams and modern technology startups.

1.5. Research Approach

We use a case study based research methodology. The case is a hotel communication management platform developed by Guestjoy. We will first give an overview of the existing solution, the relevant history and reasons for choices made. We will then lay out the desired outcome after microservice architecture is applied and the stepwise path from a monolith system to a system comprised of microservices. Through this practical case study, we will observe the impact of challenges introduced by microservice architecture as well as improvements it brings to the project and to software development in general.
2. BACKGROUND

2.1. Monolith Architecture

“The term monolith has been in use by the Unix community for some time. It appears in The Art of Unix Programming to describe systems that get too big.” (Fowler & Lewis, Microservices, 2014). A monolith application is developed as one single deployable unit. It commonly uses a Three-Layered architecture comprising of the presentation layer, business layer and data layer (Microsoft, 2008). This is a typical style software projects traditionally begin their life as a proof of concept or a minimum viable product. The reason for that is small monolithic applications are quick to work with and kick off a new project. Some of the benefits of monolith applications are (Richardson, Microservice Patterns MEAP, 2017):

- Simple to develop,
- Simple to test,
- Simple to deploy,
- Simple to scale.

These benefits result from the fact that monolith applications are usually a single deployable unit. IDE-s and traditional developer tools are designed for single code bases. A single application is easier to reason about and test compared to multiple interrelated applications. They are also simple to scale – just add more instances behind a load balancer. However, the monolith architecture is in a way the victim of its own success. As the popularity of the application grows, the number of users grows, the number of features grows, the code base grows, the number of developers grows, the management overhead grows. The benefits of a monolith architecture mostly apply to software at the beginning of its life. To contrast the initial benefits there are also drawbacks to a large monolith application:

- Rising complexity of the single codebase,
- Increasingly more difficult to fix problems,
- Development of new features is slowing down,
- Build times are getting longer,
- Startup times are getting longer,
- Deployment takes incrementally more time and is often driven by a heavy process,
- Scaling is becoming more challenging,
- Reliability is suffering,
- Long term technology lock-in.
As the code base of an application grows, it gets more difficult to understand, it becomes harder to fit all the interdependencies into developer’s head at once. Each new change makes the situation incrementally worse. IDE’s become slower when working with large codebases, the edit-build-run-test cycle increases, reducing the overall developer productivity. When initially the deployment of a monolith application was simple and quick, as the software grows this too gets more challenging. This is because different parts of the solution can have conflicting resource requirements. Some parts may need to keep large quantities of data in memory, requiring hardware with large RAM, other parts may need to do CPU intensive processing, requiring hardware with a lot of CPU. As the application works a single monolithic unit, the single hardware unit the software runs on has to satisfy all these different requirements at the same time. With a monolithic architecture, it is also common that problems say a memory leak in one noncritical module crashes the entire instance possibly losing any unsaved work and leaving the system in an inconsistent state. As time goes by technologies change and improve. With a monolith application built as a single unit, it is nearly impossible to start using new alternative technologies side by side with the existing stack. Sometimes unmaintained third party dependencies may make it impossible to upgrade even the one chosen technology framework.

2.2. Microservice Architecture

In principle microservice architecture is utilizing a well-known approach to solving hard problems and managing complexity by “dividing a system into subsystems at the architecture level so that your brain can focus on a smaller amount of the system at one time” (McConnell, 2004, p. 841).

The field of microservices is in its early evolutionary stages (Fowler & Lewis, Microservices, 2014). As a result, the exact definition of microservice architecture is still open to debate. Almost every industry expert, practitioner and author seems to have their own understanding. For example, a Gartner presentation suggests microservices have “explicit dependencies on other microservices” (Thomas, 2014) while in contrast, other authors propose a true microservice should be completely autonomous i.e. have no dependencies. All the while mentions in the academic literature are scarce to the point that
a meta-analysis of previous research concluded that the microservice architecture is an emerging but still an under-researched domain (Pahl & Jamshidi, 2016). In addition to ambiguity around the definition of microservices, the technical implementation details and best practices are still evolving. For example, whether it is a good idea to explicitly version microservice endpoints (e.g. HTTP request URL) (Newman, Building Microservices, 2015, pp. 65-66) or if it is rather an antipattern (Fowler S. J., 2016, p. 10).

This chapter aims to consolidate some common concepts upon which to stand the current work. While Fowler & Lewis are cautious of providing a clear definition for microservice architecture, merely explaining that “/…/ microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms /…/” (Microservices, 2014).

There is no consensus today how big, or small for that matter, a microservice should be. Some definitions take the name literally claiming one service should be ca 100 lines of code (Richardson, Microservice Patterns MEAP, 2017). While others call this very same definition a nanoservice antipattern (Rotem-Gal-Oz, 2014). In general though despite the name it seems size alone is not the best way to define a microservice as there are systems that “vary from a team of 60 with 20 services to a team of 4 with 200 services” (Fowler M., Microservice Premium, 2015). Richardson goes on to offer his own definition that crucially does not use the size of the service at all but rather focuses on a cohesive set of responsibilities:

“An architectural style that functionally decomposes an application into a set of services”

Another definition centred on responsibility by (Horsdal, 2017):

“Microservices as an architectural style is a lightweight form of Service Oriented Architecture where the services are tightly focused on doing one thing each and doing it well”

accompanied with a definition of a microservice that emphasises narrowly scoped capabilities:

“A microservice is a service with one, and only one, very narrowly focused capability that a remote API exposes to the rest of the system”

or
“[Microservice is] a small, replaceable, modular, independently developed and independently deployed software application that is responsible for performing one function within a larger system” (Fowler S. J., 2016).

Others (Nadareishvili, Mitra, McLarty, & Amundsen, 2016) focus on deployability aspects of services:

“A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication”

and introduce the concepts of evolvability into the architecture:

“Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices”.

From these definitions, we can summarise that the prefix “micro” refers to the size of functionality or the number of capabilities one microservice provides rather than physical size either as a memory footprint or lines of code. Whereas microservice architecture is a way of designing software systems composed of microservices.

2.2.1. Characteristics of a Microservice

In addition to concrete single sentence definitions, microservices are often described as a set of characteristics. Below is a collection of microservice characteristics:

- Responsible for one single capability/functionality (Horsdal, 2017)
  - Small enough to fit in your head (Lewis, 2013)
  - Small enough that you can throw them away (Lewis, 2013)
- Bounded by business context (Nadareishvili, et al., 2016)
- Independently / individually deployable (Nadareishvili, et al., 2016) (Horsdal, 2017)
- Owns its own data store (Horsdal, 2017) (Wolff, 2016) (Richardson, 2017)
- A small team can maintain a handful of Microservices (Wolff, 2016)
  - Located in different source code repositories (Lewis, 2013)
- Evolutionary design (Fowler M., Microservices Guide, n.d.)
- Replaceable (Horsdal, 2017) (Wolff, 2016)
- Automated infrastructure (Nadareishvili, et al., 2016)
- Automated build and deployment (Fowler M., Microservices Guide, n.d.)
2.2.2. Technical Benefits

“At the software-architecture level, the complexity of a problem is reduced by dividing the system into subsystems. Humans have an easier time comprehending several simple pieces of information than one complicated piece. The goal of all software-design techniques is to break a complicated problem into pieces. The more independent the subsystems are, the more you make it safe to focus on one bit of complexity at a time. Carefully defined objects separate concerns so that you can focus on one thing at a time. Packages provide the same benefit at a higher level of aggregation.” (McConnell, 2004, p. 79).

Here McConnell is talking about managing the complexity of objects, classes and packages. In 2004 when the second edition was released cloud, as we understand today, did not exist and it was still 7 years to the first mention of microservices in 2011. But at the software architecture level, the same principles hold as we increase the abstraction from packages to libraries to services. Essentially microservice architecture is about dividing a system into subsystems. The benefits of microservice architecture are to contrast the problems with the monolith architecture. The characteristics of microservices described earlier bring along some interesting technical advantages to a software project. Microservices allow for fast, independent delivery of individual parts within a larger system. They support parallel development by establishing a hard-to-cross boundary between different parts of your system (Tilkov, 2015). Different parts of a system change at a different pace. Microservices are bound by business context. In fact, if two services need to frequently communicate with each other or developers of the two services feel a strong urge to access the same data store, it is a sign that these services might not be properly aligned to their bounded context, a concept introduced in Domain-Driven Design (Evans, 2003). Even though Evans’ book precedes the movement of microservices it is recognised these two concepts are complementary. Logical boundaries within a monolith system do not provide hard boundaries, they are not easy to see – they don’t work well in practice. With microservices, they are much easier to see. So when a system is split into microservices strong boundaries are implicit enforced (Evans, 2016). Each business context tends to have different technical requirements and a different rate of change in requirements. This allows a system built with microservices to change at a different pace. If a single service is so small it fits into a developer’s head in its entirety, it helps to think more clearly and work faster, in other words,
to increase programmer productivity. This is colourfully illustrated by the engineering team at Karma (Karma, 2016):

“The biggest boost from microservices is programmer productivity: we don’t have to keep the whole thing in our heads! It’s all about getting rid of distractions and focusing on what is happening in front of us now instead of worrying about breaking stuff somewhere else.”

The distributed nature forces the designers of microservices systems to assume that everything can fail at any time. This will naturally force a structure that localises problems. When a failure occurs in one microservice it can be restarted or replaced and the work will be picked up from where it left off. Updates to microservices can be tested independently as each service is decoupled from others. This makes the test surface much smaller and faster. When automated tests run slowly, developers are less likely to execute them or pay attention to them. This increases the risk of introducing bugs which are much more difficult to fix later in the development cycle than at the point of introduction. Faster test cycles mean developers are more likely to run automated tests continuously and implement more test covering more use cases, therefore reducing the risk of introducing new bugs with new changes.

Because each microservice is decoupled from other they can be scaled up and down individually to meet system load. Different services not only have different hardware requirements but also different idle and peak load characteristics. By running each microservice in a separate process or virtual container allows more dynamic and smarter utilization of infrastructure resources. By deploying multiple instances of the same service reduces the risk of that capability being completely lost by a failure in one instance. By deploying the same instance to multiple physical regions service availability is improved even further.
Below is a summary of technical advantages microservices have over monoliths:

- Programmer productivity,
- Faster development,
- Faster testing,
- Faster deployment,
- Easier to identify and track down bugs,
- Embrace failure (robustness),
- Smart dynamic scaling,
- High availability and robustness.

2.2.3. Business Value

The discussed technical advantages directly translate into various business value propositions. A lot of the problems and costs in software development are related to the ability of the software system to change – in other words evolve. Change is inevitable, we know that requirements will change, customer needs will change, technologies will change; better and cheaper technologies will become available and provide a competitive advantage to those businesses who are able to adopt them most effectively.

“Microservices, when done well, are malleable, scalable, and resilient and they allow a short lead time from the start of implementation to deployment in production.” (Horsdal, 2017).

For a system to be able to keep evolving it has to embrace change in its fundamental design. Over the course of time businesses change their business models and direction, sometimes within a very short time span in order to respond to market changes and customer need. A technical solution that does not interfere with business goals and can quickly adapt to changing environment will help maintain the company a competitive advantage over its competitors.

According to a meta-study of seven sources the median cost of fixing software errors in the later phases of the software lifecycle than in the earlier phases is greater by a factor of 50 (Stecklein, et al., 2004) and by a factor of 5 in noncritical software systems (Boehm & Basili,
Applying Microservice Architecture To Developing Evolvable Systems
In Small To Medium Sized Software Projects

2001). Faster development, testing, deployment and bug fixing cycles can therefore significantly reduce lifetime costs of software projects. Further cost optimizations come from the smarter utilisation of infrastructure as individual parts of the entire solution can be scaled up and down independently of other parts of the systems as opposed to a monolith application where everything has to be scaled together resulting in inefficient resource usage. Provisioning software containers within existing infrastructure is faster and more cost effective than provisioning virtual machines. In larger corporations scaling up the number of virtual machines often involves time-consuming and costly provisioning processes. Which is partly the reason scaling back down is often not done. Because once the development team has acquired a resource (e.g. access to a virtual machine or database) they don’t want to give it up even if they don’t use it for time being. The maintenance of infrastructure may be outsourced and billed by the number of infrastructure units even if that resource is not being used by the owning company.

Thanks to each microservice being individually managed, teams can choose the best technical tools for the job. But on top that businesses can choose the best people for each individual capability by now being able to hire people with that exact skillset without constraints around nonrelated technologies (Sahasrabudhe, n.d.).

“Every company in an IT company” (Life in Estonia, 2014). Making the right technology choices matters. Choosing technical solutions that keep up with changes and evolve with the business can decide the success or failure of a company. Below is a summary of business value propositions and competitive advantages microservice architecture can bring to the table over monolith architecture:

- Evolvability (keeping up with market changes):
  - Rate of change,
  - Speed of change,
- Time to market,
- Brand reputation,
- Reduced cost:
  - Development,
  - Infrastructure,
- Better hiring.
2.2.4. Challenges

There are no silver bullets and neither are microservices. It should be obvious that just by writing systems with microservices is not magically going to make non-productive software developers more productive. Some of the advantages of this architectural style, unfortunately, come with their own points of concern that should have their due course of attention when choosing a suitable architecture for a software system. By running each service individually and forcing them to communicate across process, often across machine boundaries over the network, brings along well-known issues with distributed systems, sometimes also referred to as fallacies of distributed computing (Gosling, n.d.).

“When you use microservices you have to work on automated deployment, monitoring, dealing with failure, eventual consistency, and other factors that a distributed system introduces” (Fowler M., Microservice Premium, 2015).

There is overhead associated with managing large numbers of services compared to managing just a handful. Deployments distributed components need to be automated and well-orchestrated, monitoring infrastructure needs to support tracking communication flow across multiple services, operations teams need to manage a larger number of infrastructure units, which may require support by operations tooling. End to end testing of an entire system gets more complicated. In order to effectively set context boundaries and split a system into multiple microservices a good knowledge of the business domain is essential. If errors are made during establishing service boundaries, these may be difficult to retrofit later due to established service interfaces and message contracts that other parts of the system may already depend on. Below is a summary of challenges associated with the microservice architecture:

- Operational complexity:
  - Deployment pipeline of distributed components,
  - Monitoring distributed services,
- Testing distributed services,
- Communication between distributed services,
- Too many services / incorrect boundaries,
- Transferring functionality between microservices.
2.3. Cloud Computing

When just considering the technical benefits and business value then microservices seem like a great choice. However, the challenges are also quite real. These are mostly a combination of problems described in the fallacies of distributed computing as well as more complicated operational requirements due to the distributed nature of microservices. This is where cloud computing comes into the picture. When maintaining an in-house infrastructure the team or business typically has to keep up to date with operating system patches, manage hardware failures, manage hardware supply chains etc. A lot of the administration overhead that would be required to operate a large scale system composed of microservices can be “outsourced” to cloud providers. With Infrastructure as a Service (SaaS) cloud offerings a business can completely remove any operational overhead related to hardware procurement, and a lot of the hardware setup and configuration. With Platform as a Service (PaaS) cloud offerings a business can additionally remove operational overhead related to operating system (OS) procurement, configuration, security patching, OS upgrades etc. By going completely over to serverless architecture (ThoughtWorks, n.d.) (Roberts, 2016) one can reduce the operational overhead even further by removing the need to manage virtual machines, software containers etc.

It is common for modern software projects to embrace the cloud (e.g. Amazon Web Services, Microsoft Azure). Either by running organization’s entire operations on the cloud, or part of it on the cloud or on a private cloud, implementing some of the cloud techniques internally on organization’s private infrastructure. Or a combination of the described options. The advances in Software as a Service and Platform as a Service offerings by different cloud providers has reached a point where it is trivially easy to deploy and operate software solutions on cloud infrastructure from the simplest proof of concept projects to fully production ready monolith and service oriented architectures. The commoditization of infrastructure by cloud providers in parallel with improvements to virtualization technologies is an enabler to the adoption of microservice architecture. It greatly reduces the operational complexity and cost associated with such distributed style bringing the entire concept economically closer to the budget and capabilities small organizations and small software projects. Thanks to cloud computing it is not necessary to be on the scale of Netflix or Uber to benefit from microservice architecture. The feasibility of microservices is additionally supported by advances in virtualization (software containers) and maturing
technologies that provide solutions for microservices like distributed deployment, dynamic scaling, monitoring of distributed systems etc.

2.4. Monolith First or Microservices First

When a new technology becomes available it can sometimes generate a great deal of excitement among early adopters, potentially using the new approach in situations where it doesn’t quite fit or creates more problems than solves. It takes time and trial and error to find the right balance between different approaches. Often new technologies do not completely replace the old ones but rather increase the heterogeneity of the landscape of choice. It remains up to the decision maker to choose the right solution for a specific circumstance. The same applies today for microservices. It is new and intriguing and everyone wants to be involved. However, the implied complexity has led many to recommend not to use a microservice architecture unless a project has evolved big enough to warrant the costs associated (Fowler M., Monolith First, 2015).

“/…/ don't even consider microservices unless you have a system that's too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application.” (Fowler M., Microservice Premium, 2015)

There is a debate among industry practitioners and thought leaders whether new projects should always start with a more traditional monolith architecture or should greenfield projects begin with a microservice architecture from day one. The advocates of the monolith first architecture make the following points:

- Microservice architecture introduces its own complexity adding to the cost and risk of the overall software project (Fowler M., Microservice Premium, 2015).
- If you are still figuring out what you are doing and are not building a SAAS type service you should start with a monolith (Cockcroft, 2014).
- It is easier to split up something you already have and understand the inner workings of. So consider starting monolith first and break things up when they are stable (Newman, Building Microservices, 2015).
- If the problems that created a monolith big ball of mud are not solved then moving to a microservice architecture will simply end up in a distributed big ball of mud (Brown, 2014).
Yet there are others who argue the opposite, namely, that one should start developing their software solution with microservice architecture. Some aspects supporting microservices first approach:

- It is costly to refactor a tangled monolith into discreet autonomous services.
- Microservices enforce a hard boundary between services which logical boundaries in monolithic services do not provide (Evans, DDD and Microservices: At Last, Some Boundaries!, 2016).
- Automation (including build, testing, deployment etc.) which is a vital aspect of microservice architecture from day one has become a recommended practice and a time saver even in the case of small software projects and small teams.

The problem with the monolith first approach is that in theory it should be developed in a nicely modular manner so that later on each module can be separated as a microservice as the need arises. In practice, however, it is very hard to avoid coupling between parts of the system that shouldn’t really be tightly connected. The discrepancy of an ideal software project and an actual one is illustrated in Figure 2.

But even the microservices first advocates caution that one should only entertain the idea of developing a system using microservice architecture if one “believes their system is big enough to warrant this” (Tilkov, 2015). There are several case studies detailing the transformation to a microservice architecture, but all of them point out to a lower or higher degree that one, and often the main reasons to use this approach are the emerging scalability and performance problems when serving millions of users, like the report by Balalaie, Heydarnoori & Jamshidi (Migrating to Cloud-Native Architectures Using Microservices: An Experience Report, 2015).
2.5. Refactoring Monolith to Microservices

A monolith first approach, that assumes a later transformation into microservice architecture, has to follow strict boundaries in order to avoid long and costly code refactoring. Detailed methods of how exactly to perform this transition is outside the scope of the current work. Books and academic papers that go in depth on this topic exist. For example “Towards a Technique for Extracting Microservices from Monolithic Enterprise Systems” (Levcovitz, Terra, & Valente, 2015) and “Migrating to Cloud-Native Architectures Using Microservices: An Experience Report” (Balalaie, Heydarnoori, & Jamshidi, 2015). A sometimes overlooked key concern, when designing an in-process module that would later be turned into a standalone microservice, is data storage. Even if a team succeeds in maintaining context boundaries within their software, the same is not always the case in the database, which is often the biggest stumbling block when moving from monolith to microservices (Newman, Microservices For Greenfield?, 2015). With the microservice architecture, it is recommended to keep each service’s data private and only accessible to that service alone. There are several patterns to achieve this (Richardson, Pattern: Database per service, n.d.):

- **Private-tables-per-service**: each service owns a set of tables that must only be accessed by that service;
- **Schema-per-service**: each service has a database schema that is private to that service;
- **Database-server-per-service**: each service has its own database server.

Thus when designing a monolith for the future refactoring to microservices in mind these patterns can already be applied. For example, each module in a monolith might use its own database schema while observing that there be no database level referential integrity between modules as illustrated in Figure 3 below.
Failing to enforce strict boundaries between schemas of different modules can result in a situation where a future refactoring into microservices becomes impossible or a very difficult undertaking (Levcovitz, Terra, & Valente, 2015).

2.6. Alternatives to Microservice Architecture

Some authors liken microservice architecture to service-oriented architecture (SOA) questioning the need for a new term pointing out that microservices promote the same principles as SOA before vendors contaminated these concepts with WS-* and their enterprise service bus offerings (Rotem-Gal-Oz, 2014). From the definitions of each architectural style, it is clear that the come from the same noble place. It has been suggested we should “think of microservices as a specific approach for SOA” (Newman, Building Microservices, 2015, p. 9). It does seem that SOA, wrongly or not, is often associated with expensive vendor solutions, long implementation times and slow development cycles. A microservice architecture movement as we witness seems to be an attempt to reboot and throw away the heavyweight “misconceptions” that have been bolted onto basic SOA principles and specifically emphasises composing systems of independent and lightweight services.
Another alternative approach is Self-Contained Systems (SCS) as “an architecture that focuses on a separation of the functionality into many independent systems, making the complete logical system a collaboration of many smaller software systems” (SCS: Self-Contained System, n.d.). SCS positions itself architecturally at a slightly different conceptual angle where it “can be combined with more fine-grained microservices”. Main areas where the two approaches differ is SCS-s ideally do not communicate with each other and they typically include a user interface. In that sense, SCS can be viewed as a more coarse-grained implementation of the microservice architecture.

Ultimately we believe the alternative approaches mentioned in this section and microservice architecture all have similar good intentions at heart and promote the similar ideas at a slightly different angle or with a historical baggage. The term microservice, however, seems to have resonated with industry practitioners and stands as a label that marks the current forefront of software architecture advancement on the evolutionary path towards more robust ways of building software.

2.7. Existing Case Studies

Below is a very limited overview of microservice architecture case studies by select brands that are likely to be familiar to wider audiences. What stands out among all of those is they are large software project either measured by the amount of human or financial resources or the number of technical services.

**Netflix**

According to their former Cloud Architect, Netflix transitioned from a monolith application to microservices around 2010 (Cockcroft, 2014). Today Netflix has a market capitalisation of over 67 billion USD on Nasdaq and is the go-to reference when the term microservices comes up in discussions. They have ~30 engineering teams, over 500 microservices, ~2 billion edge API requests per day according to their Manager of Cloud Platform Engineering group (Tonse, 2014).
Spotify

A global music streaming service which according to Crunchbase has received over 1.5 billion USD in investments has a microservice architecture supporting their one product of 810 active services maintained by over 600 developers in over 90 teams (Goldsmith, 2015).

Uber

A global transportation platform provider developing a mobile app which according to Crunchbase has received over 8.8 billion USD in investments. They have 1700 engineers managing ~700 services according to Uber’s Chief Architect (Ranney, 2016).

Lyft

An Uber competitor with over 2.6 billion USD in investments according to Crunchbase. As reported by Lyft software engineer they have a microservice architecture with over 100 services, over 10 000 hosts and over 2 million requests per second (Klein, 2017).

Other

Apart from blue chip brand names Martin Fowler reports in his experience microservice systems typically “vary from a team of 60 with 20 services to a team of 4 with 200 services” (Fowler M., Microservice Premium, 2015).
3. CASE STUDY

This work presents a real-life implementation of a microservice architecture in Guestjoy, a startup developing software as a service product that helps hotels around the globe increase their upsell and cross-sell revenues by respectfully managing their guest communication. This software integrates with hotel property management systems (PMS) to retrieve guest stay information and helps hotels increase their revenue per customer via a comprehensive upsell process. This is achieved by handling on behalf of the hotel pre-stay communication, post-stay communication, guest feedback, satisfaction surveys, mediating TripAdvisor (https://www.tripadvisor.com/) ratings and reviews, helping the hotel promote useful offers like room upgrades if available, restaurant vouchers, SPA discounts etc. To visualise the core business landscape the software supports, a typical hotel guest journey with relevant communication interaction points is illustrated in Figure 4.

![Guestjoy Journey Diagram](image)

**Figure 4.** Guestjoy journey

The entire system is comprised of several distinct business capabilities, which is reflected in the application source code as different functional modules as well as different dashboards on the user interface (UI). For example, pre-stay, post-stay and confirmation emails can be managed and designed under the “Email templates” section in the UI (Appendix 1). Orders and other stats on additionally generated sales are available under the menu item “Upsell”,

```
visitor feedback, survey results, TripAdvisor ranking etc. are available under “Feedback”.

The UI sections correspond largely but not exactly to capability groups in the source code. The reason why UI sections do not map directly to specific individual modules in the source code is that often UI pages are aggregate views of information provided by multiple modules. This is a common microservice architecture pattern serving as an integration point between microservices (Wolff, 2016, p. 130).

### 3.1. Motivation

Before our case study commenced Guestjoy core application was designed using a monolith architecture as illustrated in Figure 5. On this diagram, the hexagon represents a single deployable unit, which in this case is the entire software solution. Within circles are represented distinct functional business capabilities.

![Figure 5. Monolith architecture](image)

Similarly to the story of many monoliths the beginnings of the software, and in fact the company Guestjoy, were much more compact. As the business evolved so did the application. Under high time pressure and intense resource constraints, it always seemed quicker to add a “small” piece of new functionality somewhere within the existing monolith framework reusing as much existing code as possible. This does indeed make a fine strategy for proof of concepts, quick experimental hacks to explore new possibilities and software at the beginning of their lifecycle when the future of the system is all but certain. After about
a year in the making, growing user base and expanding the team, however, questions around
the long-term sustainability of this approach started to come up more and more frequently.

To exemplify a typical business scenario that is impacted by technology choices we will
look at a real life situation:

A guest makes a new reservation with a hotel. Guestjoy receives their booking
details from hotel’s PMS system. At an appropriate time (typically a
preconfigured number of days before the guest is due to arrive) Guestjoy sends
a welcome email on behalf of the hotel to the guest. Because the guest was merely
on a waitlist (the booking was not yet confirmed) the guest becomes upset and
demands from the hotel an explanation, compensations and a discount on his
upcoming visit.

The reason for the erroneously sent email was the incorrect handling of reservation status in
the booking data file. The problem was easy enough to identify and fix. However, the harm
had already been done. There is a cost associated with hotel personnel working with the
upset customer, there are monetary losses due to compensation, there is brand damage to the
hotel in the eyes of the visitor and there is brand damage to Guestjoy in the eyes of the hotel.
Now it is impossible to avoid any problems from ever occurring in software. What can make
a difference is how the problem gets dealt with. The how entails both communications with
the offended customer as well as the speed at which the issue is fixed and deployed. Fixing
the root problem, in this case, was the easy part, it was updating the live system where things
got trickier. Due to the monolithic architecture in order to update the module responsible for
importing guest reservation data a new version of the entire system had to be pushed out.
That means a change to the way one single field in the guest import file is handled required
an update to a completely unrelated offers page on the website. This forces the engineering
team to consider much more carefully how and when the update can be rolled out in order
to minimise any potential risks to other parts of the monolithic system. The functionality of
importing and parsing guest reservation data is a completely separate business capability and
doesn’t interact with the rest of the system in any way at all. It, therefore, makes a perfect
candidate for an independently deployable microservice. If this change in the architecture
was realised the engineering team could have fixed the problem and updated the
microservice responsible for importing reservations in production probably in less than an
hour, given appropriate priority. What would the impact to brand reputation be if Guestjoy
could in less than an hour after becoming aware of the problem contact the customer and say the problem has been fixed and will never occur again? Meeting and exceeding customer expectations can make or break a young company just getting their business running. Problems will arise, failures will happen, good technology choices will support managing critical business situations not become an impediment of their own. The managers and the engineering team over at Guestjoy identified a number of key scenarios, discussed below, where the current monolithic architecture was slowly becoming a bottleneck in meeting business and technology goals.

Firstly the development to production cycle time was increasing as the “physical” size of the application grew. For example, even the smallest change or bug fix in the guest import part of the application required repackaging/compilation, uploading, unpacking, installing, verifying of the entire system. Despite the fact that logically there is practically zero common ground between the source code providing guest import functionality and source code responsible for rendering the offers page. We can similarly reason about the other functional modules, represented as circles in Figure 5 above, with the exception of cross-cutting concerns like logging.

The second concern was multiple developers working on the code base at the same time. Even if they work on completely unrelated modules avoiding file merge conflicts they need to deploy the entire system as a single unit. Agile practice promotes small but frequent releases. With a monolith approach, this can become a point of contention when multiple people work on the application at the same time.

The third concern was increasing client-side load times. Due to the nature of the chosen technology stack and its implementation specifics, the web browser in the client role was downloading unrelated parts of the application noticeably slowing down client load times and thereby negatively impacting end user experience. There are a number of solutions to the specific problem of client load times. However in combination with other mentioned concerns and the desire to potentially trial alternative technologies in the future was pushing the thinking in the engineering team towards splitting up the monolith.

Fourth concern was downtime of the entire system whenever the smallest update was deployed. For example, in order to deploy a fix to the XML parsing code within guest import module, the entire application suffered a downtime. As the application was growing so was the time of service not being available to Guestjoy clients and also their visitors. For
example, if a hotel guest interested in booking a service from the hotel happened to do this during the update caused by guest import module that guest would experience an unresponsive web page potentially giving up the purchase. This scenario translates to not just lost revenue but also brand damage to the hotel and to Guestjoy.

To summarise existing concerns with the monolith architecture:

- Increasing “physical footprint” of the software,
- Increasing contention during parallel development work,
- Increasing deployment time,
- Increasing application load time.

The above technical concerns translate to the following business issues:

- Decreasing time to market,
- Customers having negative experience with the software,
- Customers losing revenue,
- Brand damage.

The analysis described in the previous paragraph drove the team and the company towards a decision to explore solutions before the problems deepen and accumulate.

### 3.2. Analysis of the Existing System

In order to establish suitable context boundaries required for splitting a monolith application into microservices, we need to analyse the responsibilities and behaviour or the current system. For this purpose, we have identified distinctive functional capabilities as illustrated in Figure 5 above. The role of each within the greater system will be discussed in the following paragraphs. The term module in these discussions is a logical module. They are an integral part of the monolith system as various (JavaScript) code files, not always clearly identified by software packages, logical packages, namespaces or classes.

**Guest Import** – Integrates with hotels’ PMS-s to import guest reservation information. The core information contains guest’s name, email, the start of stay, end of the stay, reservation status (booking, cancellation). The property management software space is very fragmented with no standardised data formats in use. Not only does every system implement their own data schema, they also have varying levels and quality of integration methods. Some have
an API we can access, some are able to generate a reservation report either in XML or CSV and email that file to Guestjoy, some send the reservation data inline as the message body in the email, some clients are only able to mediate booking request emails from online travel agents like Booking.com. Guest Import module is able to acquire this data from all the various sources and parse the contents to a uniform data model. It then stores this sanitised data in Guestjoy database, ensuring data privacy requirements are met. This module does not have a UI. It has no other interactions with the rest of the system.

**Transactional Email** – Sends pre-stay, during-stay, post-stay and far-stay emails at the right time. The time to send the emails is configured per email type per hotel within the admin web interface. That is also where the contents of emails are managed. Transactional Email module simply assumes this information exists and sends out emails at the correct time. A planned near-term new feature for this module is to handle bounces and complaints. For example to automatically understand a bounced email response, examine the reason for the failed delivery and if the reason is, for example, non-existent email address then mark that email invalid so the system knows not to send any further correspondence to that address. This module does not have a UI. It is used by the admin service to get email previews and to send email configuration data.

**Job Scheduler** – Runs various recurring tasks at preconfigured time. These are currently mostly various data cleanup jobs, TripAdvisor statistics per hotel, survey statistics, offer statistics, activities in the surrounding areas etc. This data is stored in a database and used by various other parts of the system. For example to suggest activities in a pre-stay email or to display various dashboards to hotel managers helping them track their position in TripAdvisor rankings. This module does not have a UI. It has no other interactions with the rest of the system.

**Concierge** – Set of web pages displaying useful information about the hotel like the various services, amenities, contact details, interactive map, special offers etc. One of the pages under Concierge is a landing page where guests are automatically directed to when they click on offers in the welcome email. On that landing page, they can place an order for that item (e.g. airport transfer) and browse other offers. This module has a web UI and interacts with backend data sources that contain hotel details, special offers and some guest related information.
Applying Microservice Architecture To Developing Evolvable Systems
In Small To Medium Sized Software Projects

Admin – An umbrella module exposing most of the settings and configuration pages an hotelier will use to set up their hotel. This is also where a hotel manager has a visual access to email templates, rating statistics, various analytics etc. This module has a web UI and interacts directly with many of the other modules/services.

Analytics – Module responsible for collecting, organising and generating various analytics like the number of orders made via Guestjoy upsell channels. This module has a web UI.

Survey / Feedback – Set of functionality related to managing guest feedback and post-stay surveys. Hotels can optionally enable a feature whereby guests will receive a warm goodbye email after their stay has ended. A hotel can optionally configure a custom set of questions they would like their guests to answer in a survey. This correspondence can also be configured to automatically send guest ratings to TripAdvisor. This module has a web UI.

3.3. Deployment Pipeline

The system is built on top of Meteor web framework (https://www.meteor.com/). It is a JavaScript based development kit building on top of Node.js (https://nodejs.org) enabling rapid prototyping through the entire application stack (database, middleware, frontend) as well as mobile development. The ease of use and speed of Meteor development has been paramount to the success of Guestjoy. The tooling around Meteor supports easily automated deployment to Amazon cloud (https://aws.amazon.com/) in a Docker (https://www.docker.com/) container. Virtual machine provisioning, networking, load balancing etc. is handled by AWS.

3.4. Development Workflow

Developers write application code with their laptops and upload changed source code files to a staging server on AWS. They then log into the staging server and run a Meteor packaging tool Meteor Up (https://www.npmjs.com/package/mup). This tool automatically builds and packages the application, configures a Docker container, uploads the packaged image to preconfigured machine on Amazon and runs basic post-deployment health checks.
3.5. Proposed Solution

As this case is not a greenfield development so the problems and risks discussed in “2.4 Monolith First or Microservices First” do not apply. The existing boundaries are fairly stable and the engineering team has accumulated sufficient amount of domain knowledge. The infrastructure resources are outsourced to Amazon and largely automated. The deployment pipeline is simple and fully automated. The system does not yet exhibit “web scale” throughput issues and is, therefore, an ideal case to study the applicability of microservice architecture to small software projects. Based on literature analysis and industry reports microservices should address the problems the project is experiencing with the current monolithic architecture.

To alleviate the problems with the monolith application identified at the beginning of this chapter we proposed a gradual refactoring to a microservice architecture. We would start by choosing one module from existing the existing set within the monolith app (as identified earlier) and extract the functionality into a standalone service. After the first module has been transformed into a microservice we would pick the next module, turn it to a standalone microservice and repeat the process until all of the modules currently internal to the monolith are running as microservices. That means there is a period between the monolith architecture and microservice architecture where we have a hybrid system comprising of a monolith application with gradually decreasing functionality and an increasing number of independent microservices. This intermediate state is illustrated in Figure 6 below.
Taking a gradual approach and making each individual change at every step as small as possible is a practical guidance described in the literature (Newman, Building Microservices, 2015, p. 79). The same source also discusses the importance of keeping API-s between microservices as technology agnostic as possible (p. 39). The monolith is built using a JavaScript based framework so it has a great out of the box support for HTTP. Serving the World Wide Web HTTP has proven over time to be a good technology agnostic communication solution. Literally, any popular programming library has support for HTTP. Its interoperability does have a cost in the relative size of HTTP messages and overhead introduced by the protocol (IETF HTTP Working Group, n.d.). In certain high throughput scenarios, this might be a concern and should be evaluated against other options. However, performance is not one of the driving motivations for this project. Considering the above we decided on HTTP as the communication mechanism between different microservices. Keeping in mind that as and if needs arise we have not closed the door for other technologies in this regard. For example, if at some point a service is developed that requires communication with very different characteristics we would still be able to implement a suitable new protocol for that specific purpose. We propose JSON for the data transfer protocol for similar reasons: support in the technology stack currently in use, wide support in most popular frameworks and languages, performance requirements that satisfy our project’s needs.
3.6. Practical Implementation

The actual implementation process started with a number of discussions and design meetings with the engineering team. Due to the nature of the project and the company in a fast-changing and competitive market, the project lacked formal documentation and architectural designs. The technical system had grown organically with the business. During whiteboard sessions, the current state of the monolith architecture was visualised and formalised. This gave a good basis to analyse the current state of affairs and propose an action plan to take forward, as discussed in previous chapters.

The case began with first moving the existing code base to a Git source repository to improve collaboration among the growing engineering team. We then extracted the Guest Import module into an independent service. The decision to move Guest Import capability first was based on the circumstance that it was fairly well isolated within the monolith application, its responsibilities and bounded context was clear, it had frequent updates that weren’t relevant to other parts of the system and it was clear it had a number of new features (integrations with new PMS-s) to be implemented in the near future. This made it an ideal example to encapsulate as a microservice. Initially, we moved all the code related to importing guests into a new Git repository as verbatim as possible in order to minimize the risk of introducing new bugs. The system now consisted of two services – app (the old monolith application) and guest-import. Next, we introduced coarse-grained acceptance tests to each import format. The old system had only been manually tested. After we had automated tests running and passing we felt confident amending the code to remove some of the dependencies that were a remnant of the full stack framework of the app not required in a backend only lightweight service. At first, we kept the old code in place within the app and ran the two services in parallel to compare and ensure their behaviour is identical. After a staging period when we felt confident that the new service behaves like the old monolith we removed all code from the app related to guest import functionality. This completed the process of extracting the first service and breaking in new development and deployment procedures.

In parallel to the work of extracting the Guest Import service other members of the engineering team worked on implementing a Facebook chat bot service – a completely new feature to help guests communicate with the hotel they are staying in and ask for real time
Applying Microservice Architecture To Developing Evolvable Systems
In Small To Medium Sized Software Projects

information. This effort followed the new microservice architecture developing the chatbot as a standalone service completely independent of other parts of the system.

Next up was Job Scheduler which encapsulates recurring tasks and smaller batch jobs. It was decided to put all the tasks of the various nature as discussed previously into a single service because with a small engineering team the overhead of maintaining each task in its own microservice does not make up for the advantages of separating the logic as perhaps a purist approach might mandate.

As a result of this undertaking, the intermediate architectural picture as depicted in Figure 7 closely resembles the proposed hybrid architecture in Figure 6 with the addition of chatbot microservice.

Figure 7. Hybrid system: an actual intermediate state after parts of the monolith had been broken up

In addition to extracting modules into services at the source code level, an AWS load balancer was configured to map external routes to relevant microservices. For example, the Guest Import microservice runs on port 1001 but this functionality was previously available externally as part of the monolith app on port 80. How the load balancer ties together different services is visualised in Figure 8.
At this stage of the project, the function of the load balancer is routing external HTTP URLs to corresponding subsystems rather than distributing load because pragmatically this is not yet required. However, this new configuration serves as a good basis for future when the need to distribute usage load might arise.

Previously the single monolith application logged system diagnostics messages to the console and certain key events to a database. With a distributed microservice architecture a more thorough monitoring approach is needed. As the first step in this path, we have considerably increased the amount of information the software is logging. We still store certain key business events into a database but the various services now send all the additional diagnostics and metrics information to Amazon CloudWatch, a place to collect and monitor log files, track metrics and set up alerts based on various thresholds.
4. ANALYSIS

4.1. Key Performance Indicators

We take a snapshot of the current state of the case study three months after the initiative to break up the monolith architecture to microservices was launched and analyse the results, where they met the expectations and where they diverged. We will be concentrating on two main areas of interest that contribute to the evolvability of a system: the rate of change and complexity. As stated previously we believe evolvability is one of the key properties of a software system. Similar ideas have been expressed elsewhere in the literature including (McConnell, 2004, pp. 464-465, 841) listing characteristics of software quality to be maintainability, readability, testability, understandability among others.

To quantify the rate of change we use the following KPI-s: Product Deployments and Product Changes. The first represents the number of times the product has been installed, technically corresponding to the number of deployments of each service. The second represents how many times a product has been changed. Technically we use the number of program source code commits to proxy this business property.

To quantify the changes in system complexity, which in turn contributes to evolvability, we use the following KPI-s: Cyclomatic Complexity and Cognitive Complexity. The first is a classic metric to measure “testability and maintainability” (McCabe, 1976), a mathematical model which quantifies the number of linearly independent paths through program’s source code. McCabe defines cyclomatic complexity with a formula $V(G) = e - n + p$, where $V(G)$ is the cyclomatic number of a graph $G$ or complexity of a program, $e$ is the number of edges in the graph, $n$ is the number of node sin the graph and $p$ is the number of connected programs. A simplified explanation is to just count the number of decision points in a program (McConnell, 2004, p. 458). In practice, cyclomatic complexity has proven great for indicating the number of tests required to cover a program’s execution paths but has a number of shortcomings when it comes to maintainability and therefore evolvability as detailed in a white paper that introduces a cognitive complexity metric (Campbell, 2017). Cognitive Complexity is a new software metric intended to more accurately measure human’s ability to read and understand source code that defines a software program. Technically we will be measuring the cyclomatic complexity of each service, average
cognitive complexity of source files in each service and average cognitive complexity of functions in each service.

The raw data as well as the technical commands to capture the data are listed in Appendix 2 and Appendix 3. Complexity metrics are collected using software quality inspection tool SonarQube (Appendix 4).

Figure 9 below shows how the number of deployments to production before and after we started our work splitting the system into multiple services. Before we just had a single monolith application. We collected the number of deployments per week over the course of four weeks. After we applied microservice architecture we continued to collect data for the number of deployments of the monolith service (now with reduced functionality that is being filled by extracted microservices) as well as the average number of deployments over all the services in the new system design.

![Figure 9. Product deployments before and after applying microservice architecture](image)

While Figure 10 illustrates how the architectural change has impacted the number of source code commits or product changes.
Figure 10. Product changes (code commits) before and after applying microservice architecture

Table 1 shows how the split into microservices has changed the complexity characteristics of the product as a whole. Complexity is one of the prime factors affecting the evolvability of a system. The more complicated a system is the more difficult it is to apply changes to it without compromising its integrity or introducing new problems.

Table 1. Product complexity before and after applying microservice architecture

<table>
<thead>
<tr>
<th>Product Complexity</th>
<th>Cyclomatic Complexity</th>
<th>File Cognitive Complexity</th>
<th>Function Cognitive Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before – Monolith App</td>
<td>3579</td>
<td>24.5</td>
<td>2.7</td>
</tr>
<tr>
<td>After – App Service</td>
<td>2804</td>
<td>18.8</td>
<td>2.3</td>
</tr>
<tr>
<td>After – Guest Import Service</td>
<td>470</td>
<td>31.3</td>
<td>4.6</td>
</tr>
<tr>
<td>After – Email Service</td>
<td>591</td>
<td>28.1</td>
<td>7.5</td>
</tr>
<tr>
<td>After – Job Scheduler Service</td>
<td>331</td>
<td>19.5</td>
<td>2.7</td>
</tr>
</tbody>
</table>

As we partition a large monolith into smaller isolated services we expectedly reduce cyclomatic complexity per service. Notably, however, the sum of the measure of all services comprising the new system (4196) is much greater than the measure of the old Monolith App (3579). This is due to two major factors. Firstly the new microservices have had new functionality added to them. But also, there is some functionality duplication in the form of certain cross-cutting concerns like logging and utility functions. We can see a very interesting change in cognitive complexity which much more accurately measures
Applying Microservice Architecture To Developing Evolvable Systems
In Small To Medium Sized Software Projects

Maintainability of a software system. As we take away functionality from the Monolith App reducing it to a smaller App Service we observe a reduction in the average cognitive complexity measure indicating the increased maintainability of that part of the system. At the same time, there is a sharp spike in the average cognitive complexity of two services that have been isolated from the monolith app – Email Service and Guest Import Service. This indicates that a lot of the complexity related to readability and maintainability was encapsulated in these parts of the old Monolith App. Even though the extracted Guest Import Service has cognitive complexity almost two times that of the new App Service the service itself is now much smaller as indicated by cyclomatic complexity. This suggests that overall the now isolated Guest Import Service is easier to reason about. This is to correlate the idea that in order to manage complexity one should aim to divide “a system into subsystems at the architecture level so that your brain can focus on a smaller amount of the system at one time” (McConnell, 2004, p. 841). Furthermore, as the Guest Import Service is independent of other parts of the system we can evolve it independently too – adding new features and being more aggressive when working on reducing the cognitive complexity in a more targeted manner.

4.2. The Good

Increasingly one of the main points of concern for the company was the ability to add new features and fix problems in their software. In other words, the increasing difficulties in changing the core product were becoming an impediment to progressing the business. By extracting the most volatile parts (e.g. Guest Import) of the system into isolated individually deployable services we have observed an increase in the rate as well as the speed at which these parts of the system are being changed.

Another area of contention for the company was the decreasing ability to do parallel work on the product. This problem is especially obvious when more than one person is working in the same area in the product. But quite often even when the work of engineers affected different business areas within the software they would touch a common configuration or storage or another indirect common dependency thereby causing a conflict when merging the work of different developers into a single deployment. This sometimes resulted in one engineer halting his work to wait for a colleague to finish his work and save it to the central
repository. As reported by the engineering team these types of delays are much less common now. The team also reports there is less fear in making changes to the product and in releasing new versions of the product. This is due to each change and each deployment having a much-reduced potential impacting on the overall system, therefore, localising and minimising the risk of major system-wide errors. For example, if a developer introduces a change to the Guest Import service (as illustrated in Figure 7) that for whatever reason contains an error so bad the entire service stops working, this will now have no effect on the rest of the system. This is because Guest Import provides a completely distinct business capability and is deployed as an independent service. Due to the nature of the Guest Import functionality even if the entire service is not operational for a day the impact on the business is barely noticeable. This would not have been the case before with a monolith application if an error in the guest import functionality rendered the process running the whole system non-operational. We have observed that with the system split into multiple isolated services the business is now able to work on multiple changes in parallel in a safer manner minimising the risk of failures to the system as a whole as well as downtime of websites visible to hoteliers and hotel guest.

We have observed situations where from the moment a problem is reported with one of the services it is fixed and deployed to production within minutes. This is a compound result of the “physical size” of each individual service now being a lot smaller and reduced risk of negatively affecting the entire system.

4.3. The Bad

When a single system is split into multiple smaller systems making sense of each individual part becomes easier at the expense of reasoning about the systems as a whole becoming harder. As noted in the literature more effort needs to be put into monitoring (Wolff, 2016, p. 77) solutions. We found this concern to hold true with our case study as summarised by the company involved in their final experience report (Appendix 5). Extra effort was put in to introduce a common logging framework to be used in every microservice. This then created a need to collect and analyse this information in a single place. It would become overwhelming if interpreting logs of every service would require contacting each of these services individually in isolation. This new complexity, however, was quite conveniently
handled by Amazons CloudWatch, a ready to use monitoring service. Each microservice sends diagnostics and other logging data to a central storage where CloudWatch provides easy search and monitoring functionality. Amazons free monitoring and log aggregation solution is not the most advanced on the market but does adequately fill the initial needs of the project of the size we are interested in this case study. In fact, the current level of monitoring and logging is more thorough and saves more time compared to the situation before the case study launch. Although this is not an inherent property of microservice architecture it serves as an example that depending on the state of the project some additional work caused by moving to the new design could be partially offset by the fact that some of it would have had to be done anyway even when staying with the monolith architecture.

Another additional overhead we have observed is managing common functionality and certain cross-cutting concerns. As every microservice is developed in separate source control repositories, every piece of common code will require duplication. To come back to the logging example when changes are made to the structure or configuration of logging these will currently have to be duplicated in every service. With a handful of services like in our case study, this is manageable on an individual service level. However, as the number of services and therefore instances of code duplication increase another approach like central log configuration should be introduced. We feel the size of our example project does not yet warrant this extra effort and complexity.

Last but not least as a relic of the old monolith application the system still has database level integration between microservices. Database integration in microservice architecture is considered an anti-pattern that should be avoided at all costs (Newman, Building Microservices, 2015, p. 78). The decisions around data integration between microservices have been discussed with the engineering team and the current conclusion is to be mindful of the potential issues and come back to this matter case by case as developer resource priorities relax.

4.4. The Future

At the current point in time, the project has five services, each contained in their own source code repository and deployed individually and independently of each other. At this size of the system, the current deployment mechanism is satisfactory. We do see that when the
number of services to be deployed increases this is an area that will likely need extra
development effort to streamline and combat the increasing complexity of managing the
deployment pipeline. Also monitoring the communication flow between multiple services
will likely require more attention in the near future. This is where something like a
correlation id (Newman, Building Microservices, 2015, p. 162) becomes useful.

Based on the current analysis of the system, following the design in Figure 6, we envision a
potential architecture where the remaining modules have been split into microservices in
Figure 11.

![Figure 11. A possible microservice architecture of a future Guestjoy system](image)

If the business expects rapid growth of their client base then other aspects the team might
want to pay attention is load and capacity planning and high availability. At that point,
deployment will probably have to take place into multiple servers, perhaps into multiple
geographical areas. Also, multiple instances of the same service are likely to be deployed to
the same area. This further increases the complexity of the deployment pipeline and system
monitoring. So additional costs at that point will have to be considered by the business.

However when the company grows their engineering team then it will be easier to onboard
new developers as they can be assigned to one microservice at a time. Starting work on an
isolated service providing only a single business capability takes a lot less effort from a new
team member not yet familiar with the system. As the business evolves and grows this will help the company maintain speed and agility. We have already seen the first signs of this effect in the development of a new service based on Facebook Chatbot which was not part of the original monolith application but came later as a new value adding and differentiating product feature.

We want to note potential future benefits related to data privacy. The project currently handles data like names and email addresses that falls under the personal information category. The legislation around data privacy is likely to get stricter and broader in the future. Also, the company may wish to enter new business opportunities that require more personal information, one example being credit card payments. If data sensitive business capabilities are isolated into distinct microservices this will make it easier to ensure their compliance to various legal requirements also reducing the surface area of any security and privacy audits.
5. CONCLUSION AND RECOMMENDATIONS

5.1. Conclusions

Among IT practitioners there is a common belief that microservice architecture is suitable for only big and very complex software systems. At the same time, sometimes the same authors who warn about the complexities of this approach (“/…/ don't even consider microservices unless you have a system that's too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application.”) will also point out that eventually as a software system grows “microservices are a better place to be” (Fowler M., Microservice Premium, 2015). There is a debate in the industry over the benefits and challenges of microservice architecture which often concludes microservices are most suitable for very large systems comprising of hundreds of services (e.g. Netflix, Amazon). In this paper, we proposed that it is practical for small software projects to apply microservice architecture and benefit from this style of software design. We have observed several trends in the way related technology has matured in the recent years and in combination with the correct compromises our case study concludes that microservices can be a viable architecture even for software projects already as small as five distinct services. This is partly because the challenges that come with microservice architecture have a much less negative impact on the overall complexity in small systems compared to large systems. For example to monitor and troubleshoot a system comprised of five services in a single datacentre one doesn’t need all the extra tooling and infrastructure that a system consisting of hundreds of services in multiple geographic regions running in parallel would need. Yet at the same time, this separation into isolated services allows us to change the system at a faster rate allowing the business to bring new features and products to market quicker. The ability to change a software product more frequently is additionally supported by making the overall complexity of each individual service easier to manage. Considering that managing complexity is one of “the most important technical topics in software development” (McConnell, 2004), these observations together support our thesis on applying microservice architecture to developing evolvable software systems.
5.2. Recommendations

Based on this case study we feel confident recommending microservice architecture to small and medium sized software projects. We do caution though, that care needs to be taken when considering which auxiliary concepts supporting distributed service based architecture are truly required. These requirements will likely be specific to each individual project. Our case study has successfully applied deliberate educated compromises in the following three areas of microservice architecture:

1) maintaining integration at the database level,
2) avoiding comprehensive system-wide monitoring too early in the project lifecycle,
3) using simple single datacentre automate deployment pipeline.

For example, if a monolithic system is being decomposed into microservices, initially the new distributed system might choose to keep using the same database. This can reduce the deployment and maintenance related overhead as well as reduce design complexity. At least for the time being because one of the problems with a monolith design is that over time as the number of interdependencies grows, the very same previously simple design starts to become complex design. This does mean that otherwise functionally independent microservices are still integrated at the database level. But as a deliberate educated compromise, it will allow the project to start benefiting from other aspects of the architecture. If a system is not comprised of hundreds of services a comprehensive monitoring solution might not be necessary. It takes considerable resource to develop and maintain this. If a project does not require the system to run in multiple geographic regions in parallel the deployment pipeline can be much simplified. All the while choosing to get the benefits of microservice architecture that are most relevant to a particular project, be it faster time to market or resilience to failures or something else. As a software system grows and the need arises, a business can gradually apply additional economically viable properties (e.g. multi-region deployment and monitoring) that even better facilitate the microservice architecture.
5.3. Future Work

According to a research mapping study conducted in 2015 “microservices research is still in a formative stage. More experimental and empirical evaluation of the benefits is needed.” (Pahl & Jamshidi, 2016). In addition to practical recommendations to other small to medium sized software projects, our study aims to make an empirical contribution to a pool of works that new future academic studies and meta-analysis can take advantage of.

Also, depending on how the project used in our case study evolves over time, it could make a suitable candidate for a longitudinal study on the benefits and challenges that a microservice architecture brings about. The metrics used in the current study are suitable to be either continuously or retrospectively used to assess and compare the state of the project after the new architecture has been in use for a longer period, perhaps over several years.
REFERENCES


Applying Microservice Architecture To Developing Evolvable Systems
In Small To Medium Sized Software Projects


Applying Microservice Architecture To Developing Evolvable Systems In Small To Medium Sized Software Projects


Appendix 1. Product Dashboard Examples
Appendix 2. Data Collection: Product Deployments

Product deployments before and after applying microservice architecture:

<table>
<thead>
<tr>
<th>Product Deployments</th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>W4</th>
<th>W5</th>
<th>W6</th>
<th>W7</th>
<th>W8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before – Monolith App</td>
<td>6</td>
<td>3</td>
<td>9</td>
<td>15</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>After – App Service</td>
<td>6</td>
<td>11</td>
<td>2</td>
<td>14</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>After – System Total</td>
<td>22</td>
<td>51</td>
<td>13</td>
<td>30</td>
<td>25</td>
<td>29</td>
<td>22</td>
<td>23</td>
</tr>
<tr>
<td>After – Average per Service</td>
<td>7</td>
<td>13</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Notes: W1…W8 - Week 1 … Week 8

Every time a service is deployed a record is automatically added to system log database with a unique identifier and a timestamp. A database query was then used to retrieve the number of deployments of each service for a period of time. For example, a query to fetch all deployments of App Service might look like:

```javascript
db.getCollection("systemLog").find({
    source: "startup",
    service: "app",
    createdAt: {
        $gte: new ISODate("2017-03-27"),
        $lt: new ISODate("2017-04-03")
    }
});
```
Appendix 3. Data Collection: Product Changes

Product changes (code commits) before and after applying microservice architecture:

<table>
<thead>
<tr>
<th>Product Changes (Code Commits)</th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>W4</th>
<th>W5</th>
<th>W6</th>
<th>W7</th>
<th>W8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before – Monolith App</td>
<td>29</td>
<td>16</td>
<td>8</td>
<td>11</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>After – App Service</td>
<td>29</td>
<td>31</td>
<td>15</td>
<td>24</td>
<td>35</td>
<td>49</td>
<td>8</td>
<td>42</td>
</tr>
<tr>
<td>After – System Total</td>
<td>53</td>
<td>114</td>
<td>49</td>
<td>57</td>
<td>72</td>
<td>87</td>
<td>31</td>
<td>78</td>
</tr>
<tr>
<td>After – Average per Service</td>
<td>13</td>
<td>29</td>
<td>12</td>
<td>14</td>
<td>18</td>
<td>22</td>
<td>8</td>
<td>20</td>
</tr>
</tbody>
</table>

Notes: W1…W8 - Week 1 … Week 8

The number of code commits is taken from respective Git repositories using the following command (where app/ is the path to Git repository of the App Service):

```
git -C app/ log --no-merges --since=2017-03-01 --until="today" --pretty="%aE %aI %s" -reverse
```
The configuration file `sonar-project.properties` that controls source code scanning:

```
sonar.projectKey=guest-import
sonar.projectName=guest-import
sonar.projectVersion=1.0
sonar.sources=.
sonar.sourceEncoding=UTF-8
sonar.exclusions=**/test/**,.meteor/**,.deploy/**
```

Screenshot illustrating SonarQube’s complexity metrics dashboard after the SonarQube Scanner had completed metrics collection on the Guest Import microservice:

Version of SonarQube used: 6.3.1

Version of SonarQube Scanner: 3.0.1.733
Microservice Architecture Experience Report

Alar Ülem, CEO
May 5th 2017

As an IT company, our daily flow revolves around product development. If development quality or speed goes down, it impacts our business significantly. With startups, its move fast or die.

Our core product used to be one single codebase. This meant if we wanted to add a feature or fix an issue, we had to take down the entire system temporarily. In many cases updates did not go as planned. This caused the downtime to be longer than expected. When we had small number of users, these gaps in the service were not noticed. But as we grew, it was harder and harder to make changes. Sometimes even the smallest restart would cause someone to report an issue. This in term caused unnecessary work for the support and damaged our reputation. We don't want that our product will be associated with something unreliable. But other than service continuity, we ran into a second problem when we expanded our development team - understanding the product. Big monolithic codebases are harder to understand for newcomers. This makes it very difficult to get going early on.

To tackle both these issues we decided to pursue microservice architecture route. On paper the solution seemed perfect. Split the app into more manageable chunks that communicate with each other over the API. Initially the project seemed a simple thing to do, but it took us longer as some larger functions needed to be split. In many cases it was hard to decide how the API will be designed. But now that we have switched over to new architecture, I can see the benefits. Now we can do code pushes almost whenever we want. This means we move faster. If we encounter any issues with deployments, their impact will be just one service. In most cases this will not be even noticed by our customers. For longer term we are thinking about switching to different programming frameworks or even languages. With microservices we can do this gradually. There is no more technology lock-in. In the past we occasionally had performance issues where the entire product became slow and unresponsive. Now we can finally understand, what part of the product causes it. In the future when our development team grows, it will be much easier to assign new people managing only parts of the product. As a CEO, I don't have to worry about mistakes, which can take down the entire system. Changing development team members is now much easier because of smaller scope.

On the negative side of adopting microservice architecture, is that overall speed of fixing issues or adding features where multiple services are affected has slowed down. I'm worried that in the future we will have situations where all involved services work fine, but for the user there is an error and it is hard to figure out, where the problem lies. Impact of systems management so far is low, but some admin overhead has increased. Currently this is not relevant.

Overall the switch to microservice architecture is initially costly but I feel, it will pay off very fast. Benefits substantially outweigh the cons.