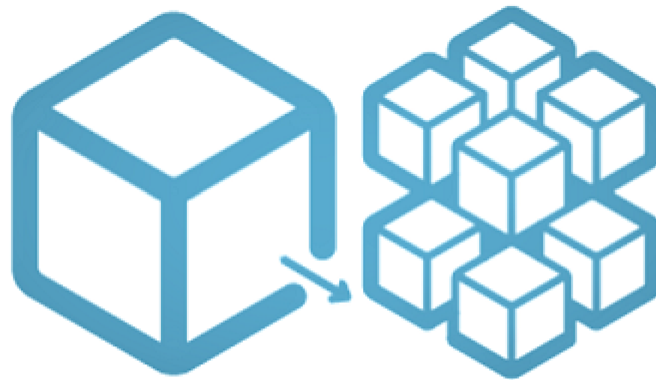


Adopting Microservices

Migrating a HR tool from a monolithic architecture

Tiago Costa Santos

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha

Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. José Alberto Rodrigues Pereira Sardinha
Member of the Committee: Prof. António Manuel Ferreira Rito da Silva

November, 2018

Acknowledgments

First of all, there are some acknowledgements that need to be made to some particular people that in various ways have contributed to this achievement.

A big thanks to Premium Minds and in particular to André Camilo for his guidance. Not only him, but everyone in Premium Minds had influence on this work trusting me with their product and being available to help when needed.

To Prof. Alberto Sardinha, for all his guidance, support and the reviews from the early stages of development of this work.

To my friends at RRR that supported me not only during this last year developing this work, but also during all my academic life. Supporting me through the bad times and sharing the good ones.

And last, but not least, to my parents and all my family. Without their support, confidence and core values this would never be possible.

Abstract

Microservices are fairly a recent trend, a new style to design web applications or systems. Building modular software composed by a set of highly decoupled services, contrasting with the traditional ways that produce one unique monolith. An increasing number of companies have been adopting microservices so they can respond to new requirements or simply to make the process of development easier and smoother for everyone involved. However, microservices are a double-edged sword, while they can solve some problems they also introduce complexity in the context. The migration process is highly attached with the context of the monolith, its purpose, development process and requirements. Thus, this process is almost unique for each company, as they should take the decisions that best fit their resources and goals. On the other hand, there are problems that are common to almost every migration process despite the context. This second type of problems are the ones we aimed to address in this work. We do this by performing a migration ourselves on an application that presents the requirements for such change. Adopting microservices is about managing trade-offs while assuring that we meet our requirements. During the process, we document problems encountered while arguing about options and our solution. Furthermore, we evaluate resulting product using well defined metrics on software quality attributes so that we can understand the impact of the migration on the product.

Keywords

Microservices; Migration; Monolith; Service Oriented Architecture; Cloud Computing;

Contents

1 Introduction **1**

1.1 Goals and Contributions 5

1.2 Organization of the Document 5

2 Background and Related Work **7**

2.1 Background 9

2.1.1 The evolution of Architectures 9

2.1.2 The evolution of the Web 10

2.1.3 Cloud Computing 11

2.1.4 Virtualization 12

2.1.5 DevOps 12

2.1.6 Microservices 13

2.2 Related Work 17

2.2.1 Identifying Microservices 17

2.2.2 Experience reports regarding the adoption of Microservices 19

2.2.3 Patterns definition for the Adoption of Microservices 21

3 Migrating to Microservices **25**

3.1 Migration Fit 27

3.1.1 Requirements 27

3.1.2 Technology 28

3.2 Splitting the Monolith 28

3.2.1 User Interface Implementation 29

3.2.2 Source Code Management 30

3.3 Serialization 31

3.3.1 ProtoBuffs 31

3.3.2 JSON 32

3.4 Communication 33

3.4.1 Synchronous Data Requests 33

3.4.2 Asynchronous Event Propagation . . . 34
3.4.2.A Event Sourcing . . . 34
3.4.2.B AMQP . . . 34
3.5 Security . . . 35
3.6 Service Orchestration . . . 37
3.6.1 Kubernetes . . . 37
3.6.2 Deploying our Services . . . 37
3.7 Moving to the Cloud . . . 40
3.8 Continuous Delivery and Continuous Integration . . . 41

4 Evaluation **43**
4.1 Size . . . 45
4.2 Complexity . . . 47
4.3 Coupling . . . 48
4.4 Cohesion . . . 51
4.5 Performance . . . 52
4.6 Infrastructure Cost . . . 53
4.7 Speed . . . 54

5 Conclusion **55**
5.1 Limitations . . . 58
5.2 Future Work . . . 58

List of Figures

2.1 Architectural Structure of traditional SOA and Microservices 10
2.2 Organization Silos 13
2.3 Deployment Process Comparison 14
2.4 Conway's Law in practice 15
2.5 Process of applying the strangler application pattern 22

3.1 HTTP Request Chain 33
3.2 RabbitMQ Topic Based Delivery 35
3.3 OAuth2 Implicit Grant Flow 36
3.4 Successfully built Jenkins Pipeline 42

4.1 Workload Execution Chart 53

List of Tables

4.1 Lines of Code per Service 46

4.2 Weighted Service Interface Count per Service 46

4.3 Total Response for Service per Service 48

4.4 Service Dependencies in the System 49

4.5 AIS per Service 49

4.6 ADS per Service 50

4.7 ACS per Service 50

4.8 SIUC per Service 52

4.9 TSIC per Service 52

Listings

3.1 Protobuf Definition for Simplified Employee Model 31
3.2 JSON representation for Simplified Employee Model 32
3.3 Events Implementation 34
3.4 Generic Database Service Volume Claim 38
3.5 Calendar Service Definition 38
3.6 Calendar Deployment Definition 39

1

Introduction

Contents

1.1 Goals and Contributions 5

1.2 Organization of the Document 5

The software engineering world is a fast-moving one, software architects are continuously working to find the best way to build systems or applications so they can keep delivering high quality software to the client while managing their teams efforts and making sure that there is a general understanding of what needs to be done and how. With that in mind, making the process of building software more simple and straightforward has always been a goal for software developers. Consequently, new architectures and development processes keep being improved or created. Microservices is the most recent trend on the software architecture community and what happens with every trend is that everyone tries to adopt it even when they do not fully understand it. In the Microservices community this is refereed as the microservice envy [1].

At its core, microservices style aims to develop an application as a suit of small services, each one running a separate process. Those services are built around business capabilities and independently deployed with low centralized management [2]. By adopting a microservices architecture, developers will end up with a set of individual highly decoupled services that can be independently developed, tested and deployed by an unique teams. This contrasts with the what has been the trend until now, to develop a self-contained application that is independent from other applications, refereed as a monolith. While an individual monolithic application has higher functionality than a microservice since it is able to perform any task required to the business context, the same can only be achieved by composing or combining microservices. A monolith not only handles all the required business logic but also all the data management operations and user interfaces definition and often times everything ends up intertwined making maintenance more complex and not favoring the continuous development. With the microservices style, each service only operates within a bounded context, these are defined around business functions using domain-driven design concepts [3]. As a consequence, data managing processes become easier as each service should have its own database and stores only the data that is useful for its purpose. What makes the style so special and a trend with such traction is that since they are independent, each microservice, can be developed with the technological stack that better fits its purpose but also, be deployed in response to demand. Surpassing the flexibility and scalability capabilities of the traditional monolith and allowing the focus to be on speed of delivering instead of maintenance processes. In the end of the day developers will have a less complex code base to manage, simple exposed interfaces to maintain and less complex pieces of software to support. Microservices are also highly related with agile software developing methodologies so that the focus is on the speed of delivering value which is also a factor for it to be so disruptive. The DevOps methodology that makes use of practices like Continuous Delivery, Continuous Integration and Automated Building are some of the concepts that are highly related with the style. This means that microservices are not only an architectural style, but also highly related with the structure and processes of an organization.

However, understanding Microservices is also understanding its drawbacks and the fact that it isn't

a silver bullet that will simply fix an application scalability or code base management problems. Having one application divided into distributed pieces that communicate over the web introduces complexity that might hurt performance. Extra latency and longer response times on requests are the most common problems, in some cases it might be something that the user can perceive which ultimately damages the user experience. This means that microservices is something that needs to be properly planed and it will require extra effort from developers in order to implement it. On the other hand, for some applications that extra effort can end up not being worth since the complexity introduced by microservices will only result in decreased productivity. This is refereed by Martin Fowler as the Microservices Premium [4]. Research has validated this complexity and in the work of Nicola Dragoni et al. [5] some other points were made. While one would assume that small-size components lead to lower fault density, it has been found that they often have a very high fault density [6]. With the high interdependency level that microservices have it ultimately leads to decreased availability. Furthermore, as stated before, the authors state that since communication is done over the network it induces latency that negatively impacts performance. While some of the problems like service discovering, load balancing and routing, for example, are already known to the distributed systems community there is extra complexity when assuring these mechanisms in a very dynamic environment such as a microservices ecosystem. This dynamism results from the fact that scaling is done on demand which causes that the number of instances of a given services isn't known upfront. Thus, all the refereed mechanisms have to be able to handle runtime operations and being constantly adjusting to the changes in the system. From an operations point of view, monitoring an ecosystem of microservices is also a complex process. Deciding what metrics are essential and setting up alerting systems are central issues so that actions can be taken as soon as possible. Reporting is often automated but diagnosing the issues that cause irregularities requires knowledge and the traceability of actions that happen in the system. Most early adopters of the style have been very open, documenting their experience on the adoption and documenting their processes of change while contributing with solutions to some of the problems found. Among them there are companies like Amazon [7] [8], Netflix [9, 10], Uber [11, 12] and others [13], some of which were doing Microservices long before the term came to be. To be sure that the system will perform as expected it has to be intensely tested and this is a problem by itself that requires higher base coverage and often code posterior code inspection to assure quality. But assuming that a system is designed for failure, it still needs to uphold that statement. Making sure that it survives under heavy load during peak time but also that when there are faults these are contained and don't end up propagated to other dependent services.

All the problems mentioned above are part of what makes the microservices style such a complex approach to system design. They require decisions that need to be made in an knowledgeable way so that the resulting system is what was initially proposed, otherwise other problems will later surface in a phase where decisions that were already taken will not be trivially reversed. But before this problems

become real concerns to architects and developers, it is first needed to break the monolith into microservices. This is the first obstacle that needs to be surpassed in the process of migration and it can only be done looking deeply into the system's architecture, from the data storage to user interaction layer, and executing it step-by-step, or better, service-by-service.

1.1 Goals and Contributions

The adoption of the microservices style has been, since the beginning, a very empirical process, with teams discovering what was required next at the moment, always blindly searching for the best solution for the problems that keep surging. With this work, we have executed the process of migration ourselves on an application that had the right requirements. We were able to understand the main difficulties and challenges that are proposed during the process of migration while documenting our experience working on the solutions for such problems. This is our take on trying to mitigate the complexity of adopting microservices and help others to to successfully deliver a microservices based product while answering to the imposed concerns like the ones expressed before. Furthermore, we provide an evaluation of resulting product using well defined metrics so it is possible to compare the initial and final state of the application and draw our conclusions about the migration itself.

1.2 Organization of the Document

This thesis is is organized as follows. In Chapter 1 we have provided a introduction to our work and our contributions. In chapter 2 we present what microservices are and where they came from and look into other work which is related to what we presented. Next, in chapter 4 we first talk about the application chosen to be migrated, following with a documentation on the migration process and its challenges. After the documentation work, in chapter 4 we expose the measurements made to the system and finally in chapter 5 we draw our conclusions about our work and expose the work set for the future.

2

Background and Related Work

Contents

2.1 Background 9
2.2 Related Work 17

2.1 Background

In this chapter we will discuss on the factors that lead to the microservices style. How it was always the desired structure of a system that was made possible by the evolution of different aspects and technologies.

2.1.1 The evolution of Architectures

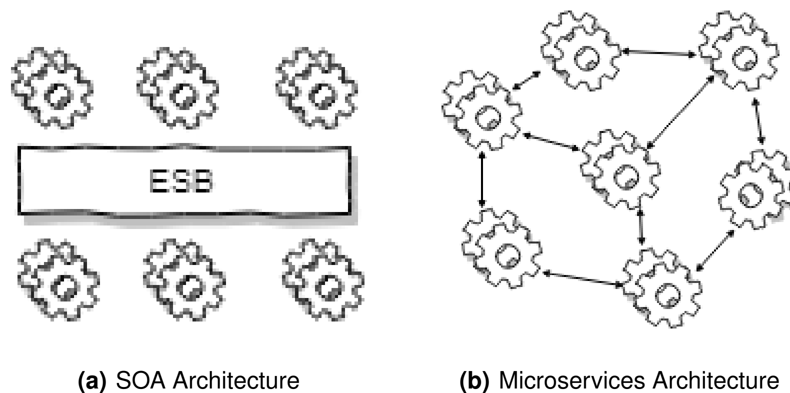
It has been a long desire to develop software as simple pieces so that each piece would only be concerned with what is included in a given scope. This way a system could be built by combining this reusable pieces that communicate through their published interfaces [14]. This concern was first addressed in the late 90s in what was called component-based software [15]. A component ”is a software package, a web service, a web resource, or a module that encapsulates a set of related functions (or data)” [16]. A modular, cohesive and reusable unit serving as an abstraction for a sub-domain of the system.

From Component-Based the software architecture paradigm kept on evolving, in the next iteration Service-Oriented Computing (SOC) was the new ideology that could assure the desired ”separation of concerns”. With the shift to the web, software became more oriented to it and the concept of ”services” started to gain relevance: While the web was initially idealized for people to use, at the time it was agreed be experts that it would need to evolve in the near future so that it could better support automated use. [17]. This automated use would be required by machines that would need to interact with each other to serve a purpose and this would be the ideology behind ’Services’. Services were going to be software components oriented to the web. The services ideology had its advantages. First, it was easier to scale by replication (and load balancing); Secondly, services were modular and could be reused, thus complex services could be built by combining other services; Third, the development process was easier since that after defining a service’s interface external entities could use it as is. The way a service worked was no longer the concern but only its inputs and outputs; Finally, it was easier to integrate services by using standard communication protocols. [5]

As businesses became more software and web oriented it was clear that the way that software was designed become more attached to the business topology. Organization’s processes, infrastructure and information flows would be, in a way, represented in the system’s architecture. Business capability became a focus when designing software and the Service Oriented Architecture (SOA) [18] came to be the style able to answer to these new requirements, becoming a de facto strategy to build software.

While there is some debate on if Microservices aren’t just SOA, it’s safe to say that they are related as in Microservices are a way to do SOA but that doesn’t mean that they are the same [19].

In Figure 2.1 it is possible to see that at its core these two approaches have different concerns.



(a) SOA Architecture **(b)** Microservices Architecture

Figure 2.1: Architectural Structure of traditional SOA and Microservices

SOA is more about how to compose an application to offer more complex functionality by integration of different services that often use a strict communication protocol via an Enterprise Service Bus. In the end these services can serve multiple business processes. While Microservices focus on the modularization of an application offering functionality by cooperation using non-strict and lightweight communication protocols HTTP and REST [20].

2.1.2 The evolution of the Web

The Web itself has shifted towards what is called Web 2.0 [21], a web built with communities in mind, more focused on the interactions between users and services and offering more dynamic functionalities [22]. The components that compose the web became more complex. Initially there were web servers using CGI to serve HTTP requests for static data, now new components developed using new frameworks are built to serve dynamic content. This dynamism resulted from the capability to embed executable code alongside the HTML interface code and it allowed the delivery of personalized content for users using data collected by other services. Furthermore these frameworks also offered a much cleaner separation between view logic and application logic to developers. In the Java world, the Model2 pattern allowed developers to encapsulate their applications into different "layers", putting application code into Java Servlets, data classes into Java beans and view logic into Java server pages. This would later lead to the origin of the MVC pattern [23] that is now widely popular with frameworks like Rails, AngularJS or React. IPC technologies were also adjusting to what was to come. The then standard SOAP allowed IPC over HTTP but feeding browsers with content wasn't the only goal of this new paradigm and text based serialization formats like XML and JSON become widely used for services that feed data to or performed actions over other services. The SOAP protocol and WS-* standards became increasingly complex and heavily dependent on specific implementations in application servers provoking developers into migrating to lighter and more flexible protocols like REST [24], which are the backbone concepts of

the Microservices Style.

2.1.3 Cloud Computing

Evolution also happened in the way that resources were made available to developers. Initially with Grid Computing [25] and more recently with new Cloud Computing models, resources started to be provided as general utilities. Being leased and released by users through the Internet according to demand. This resulted from the success of the web [26] but also enabled its further evolution.

The fact that one could have access to an "infinite" pool of resources with no up-front investment, low operation costs and on a on-demand provisioning (making it highly scalable) cleared the mind of companies of concerns about their physical resources. While monitoring is still required, planing and estimating the usage of physical resources in order to allocate them accordingly was no longer required as it is the provider's responsibility to have resources available. This allowed organizations to be more focused on developing better services and offering better products to users.

The widespread availability of services like the Amazon's AWS, Microsoft's Azure, Google's Cloud Platform among others allowed a new line of thought relating applications, the economical one - if the resources were available on demand why was the whole application being scaled increasing the total costs instead of just scaling the parts that actually need to handle more load? Provoking architects into thinking about reducing the infrastructure costs by scaling in a smarter way, breaking software into pieces that could be individually deployed - Microservices.

With the most recent advances in cloud computing there are capabilities that work in favor of microservices based applications. For example, it is possible to restrict access to certain resources. If there is a service that should only serve certain services it is possible to hide it from any other external entities providing restricted access using network isolation instead of authentication. It is also possible to increase or decrease the number of running instances during peak load times using simple mechanisms. Furthermore, it is possible to automatically maintain updated replicas of data and define tasks that run only at a defined time or given event without consuming extra resources all the time. On the other hand, one common concern with the widespread use of these technologies is what is called vendor lock-in [27]. Since there are no standards between all the cloud resources providers, each one is developing their own technologies, meaning that after committing to a given vendor and develop on top of their stack, revert that decision requires the replacement of technologies, ultimately making vendor portability a delicate and difficult concern. One example of that is the Amazon Machine Image (AMI). Before Docker became what it is today, Amazon developed their own virtualization technique, AMI, it is conceptually similar to Docker images, but it would only work in AWS. If at any time an organization wanted to migrate a another vendor's platform they would need to use a different virtualization technology.

2.1.4 Virtualization

The models of Cloud Computing mentioned above are only possible due to virtualization technologies and techniques, leveraging of those concepts to achieve the goal of providing computing resources as a utility [28]. However traditional virtualization tools had some overheads related with usage complexity and resources efficiency. That being said, handling those overheads to host a small component like a microservice didn't really seamed like an option, the whole process didn't introduced great advantages.

With the development of the operating-system-level virtualization technologies, referred as containers [29], those concerns were no longer an issue. Containers are lighter and more flexible than the traditional virtualization technologies, while requiring less resources and providing excellent levels of performance. Docker has now become an industry standard and with it is possible to have a simple application running in a container, alongside with other applications in the same host and in a secure way. This was one factor that also made developers rethink the way they were building software making the difference between the deployment as a Monolith and a Microservices based application.

2.1.5 DevOps

Agile methodologies remote back to late 90's while the term "Agile" was only coined in 2001 [30]. These methodologies were developed in order to respond to the uncertainty of the software development environment, being able to respond to rapid changes recognizing that planing has its limits. However, being agile is more than that. Agility is also empowering teams trusting in their processes and that they are competent and able to organize themselves. Working to efficiently communicate with stakeholders, involving the customer in the development process in order to deliver working software faster. Building products incrementally, adding value to the final product with periodic releases each one planned individually and at proper time [31].

While organizations become more flexible, these different stages become silos as depicted in Figure 2.2, communicating only trough artifacts. This might result in disinterest over other team's responsibilities. At its core, DevOps aims to increase collaboration by looking at software development (dev) and software operations (ops) as shared responsibilities. Involving different roles like developers, system administrators, quality assurance personnel (and others) in the same team [32]. Motivating the joint reasoning about the product and knowledge sharing from different areas where communication didn't used to be so effective because of the siloed structure. For instance, if the development team is confronted with the problems of supporting a system during its lifecycle it will be more motivated into reducing those problems. Furthermore, DevOps intends to decrease the time between the moment when a change is committed and the moment when the changed product is deployed [33], making use of various tools to automate or aid aspects during the development and delivery processes.

With the adoption of the DevOps mentality, organizations started forming cross-functional teams that could fully develop and maintain a piece of software. The sense of ownership was better defined as responsibilities were no longer spread, this can be seen early in Amazon's ideology: "you build it, you run it" [8]). Furthermore, as speed to market was a concern of the methodology, tooling became an important factor that started to be introduced. The introduced tools would automate management tasks like testing, building and deploying, greatly reducing overheads in the whole development process. This also made the development of smaller software units possible since the size of the developed software was no longer something that would dictate how many work-hours it would require to do those tasks, it was all done automatically.

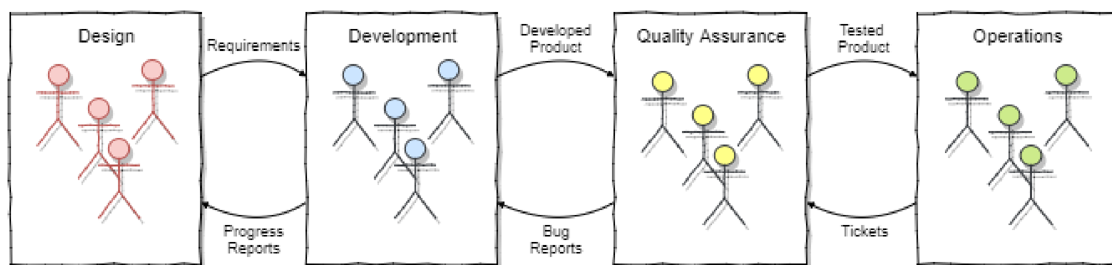


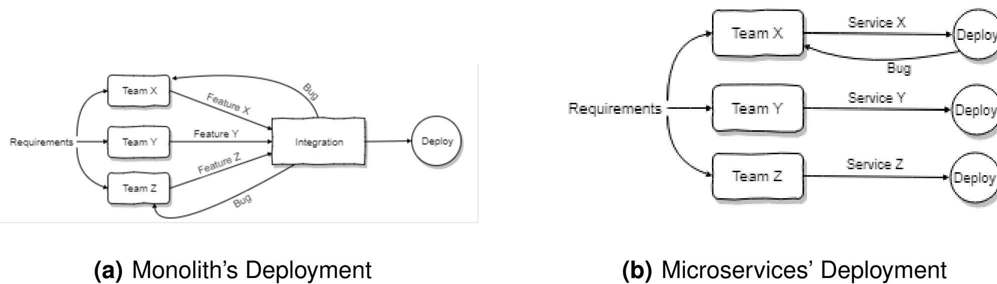
Figure 2.2: Organization Silos

2.1.6 Microservices

Now that is clear where do microservices come from and what driven their development we can expose their advantages and disadvantages more clearly and talk about their philosophies.

Microservices are still services as defined before. What characterizes them is not exactly their size, but the fact that they are simple, loosely coupled and individually deployable components that when combined serve the functionality of a complex system. In Figure 2.3 it is possible to see that the fact that the work from different teams needs to be integrated might cause delays in the release of new or reworked features in case of bugs [34]. With microservices that situation is avoided as each service's release is is only limited to their own bugs. On the other hand, microservices have to be integrated, something that might open the possibility to extra errors, but this is only caused if previously defined interfaces are changed, which is the equivalent to contract breaking in the classical protocols. This situations, however, are diminished as it is a team's responsibility to maintain a service assuring that current and future releases still comply with the documentation provided. In any case, when in situations where greater changes are needed mechanisms like versioning and route redirection can be used.

The name, microservices, might induce some error, since the use of "micro" seams to imply that a



(a) Monolith's Deployment **(b)** Microservices' Deployment

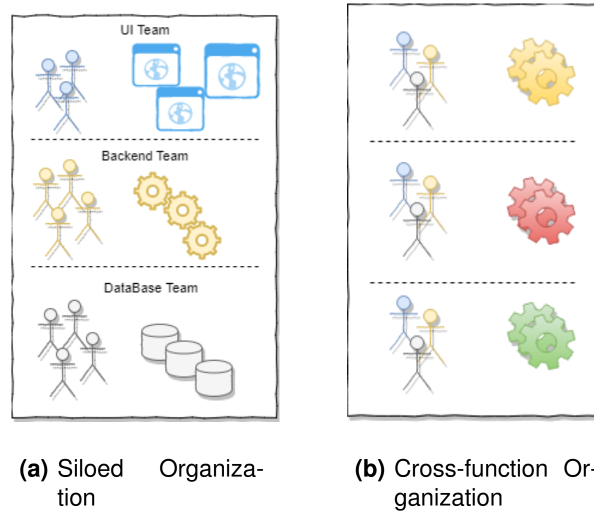
Figure 2.3: Deployment Process Comparison

service is a very small piece of software however that's not always the case, microservices can vary in size. This is something that can also be observed by comparing the size of teams that are using the style. From small setups where each person is responsible for one service to Amazon's notion of the Two Pizza Team [35] where a team which is responsible for a service should be small enough to be fed with two pizzas. This enables a whole new dynamic inside the organization. Monoliths are developed by different teams that are divided by competences, backend, middleware and UI specialists resulting in siloed software. Using microservices implies having smaller teams that are responsible for a few services. This results in the formation of cross-functional teams that will develop a service top to bottom organized around business capabilities. This was firstly observed by Melvin Conway and since then is refereed as the Conway's Law [36]. Depicted in Figure 2.4, this principle states that a system's structure will always end up looking like the communication structure of the organization that built it. A clear example of this is the popular 3 layer architecture, depicted in Figure 2.4(a), the clear separation between the layers, database, business logic and interfaces, represents how classical software development companies are structured, having specialists teams for each of those areas.

Advantages

The style has some features that work in favor of developers and organizations that enable a more efficient way to deliver software and reduce inter-team dependency. These are the aspects that move organizations and developers into introducing it in their systems [19].

- `Decentralized Governance` Also mentioned above, the fact that it requires a small and independent team to develop a service reduces the time spent coordinating and organizing different teams in order to deliver a product. Furthermore, teams can develop each service using the technologies that best fit its purpose, from programming language to database technology or UI/UX framework (when it is the case) as long as there is an explicit published interface that is maintained.



(a) Siloed Organization **(b)** Cross-function Organization

Figure 2.4: Conway's Law in practice

- `Decentralized Data Management` Since a microservice is always bounded to a limited context it will only perform the tasks necessary to that given context, only managing the data that is required for those tasks, thus it is possible that different services use related data but the way that it is stored and/or identified in each one isn't related and not directly translated.

- `Infrastructure Automation` Given that a microservice is "small" piece of software it is easier to test and deploy than a monolith meaning that these tasks are now easier to automate. The basic ideologies of Continuous Delivery [37] are that you get rapid feedback about the state of your new build and that you can perform push-button deployments of your software. For this to happen the life-cycle of the software must be automated using Deployment Pipeline that includes all the testing stages and configuration fetching for a given environment producing a deployment-ready image of your component.

- `Design for failure` As consequence of using services and seeing it as components is that since one cannot control the whole environment where the application is running as happens in monoliths each one must be able to handle failures of other services gracefully, this requires intensive testing and good monitoring resulting in service resilience.

- `Evolutionary Design` Another consequence of componentization is that it allows developers to change and upgrade and even replacing services freely to ensure functionality and performance meaning that the cost and resistance to change is decreased.

- `Code-Base Management` Each microservice is an independent component and as such there is no need to keep the code from a whole application together allowing an easier storage in separate

repositories. This results in multiple small repositories contrasting with the way it was usually done with monoliths. Additionally, it reduces the complexity of code inspection, traceability and even the process of introducing the system to a new developer as he can easily go through a service without much effort.

Drawbacks

The level of independence and componentization of the Microservices style has some drawbacks since an application results of orchestration of multiple services that are not local but are distributed across the web. Hence, there are some new concerns that need to be taken into consideration [19].

– `Data Management` While it was mentioned as an advantage before the fact is that is a common practice to have each service with its own storage but this means that assuring consistency is no longer trivial. Either consistency isn't a problem to an application or a way to manage distributed transactions will be required what can induce complexity and performance issues. This is one of the most complex problems to solve with microservices and it has been a very discussed topic [38, 39].

– `Testing` In a microservices ecosystem there are multiple interactions where every service makes requests to other services through open protocols. While each individual service should be well tested, often times it is hard to predict the content of incoming requests since they can come from different entities and serve different purposes. Furthermore, there is the addition of the asynchronicity aspect and the fact that a microservices ecosystem is very dynamic resulting in complications when it comes to find the source of a problem.

– `Availability Management` Usually, to fulfill a request a microservice performs requests to other services and this can continue down a request chain. Even though developers aim to high levels of availability (3 nines availability and more) for each service that is added to that call chain, a service has its availability decreased. Additionally, while one could assume that small-size components lead to a lower fault density, it has been found that that small-size software components often have a very high fault density [5].

– `Trust` A fairly new concern to the style is trust management, issues resulting in conversations about standardization [40]. While there should be trust among teams inside the same organization since they work for the same goals, that is not the real problem here. The fact that different organizations need to have their services communicating in order to serve their goals means that trust in external organizations is needed. But if each team exposes functionality developed as they best see fit how can an external entity be sure that it is reliable and built using best practices and

that it wont hurt the availability or performance of their service? Standardization would come to answer this concerns but it would also mean the need to define a stack that is acceptable to a wide range of people or organizations. This is something that would ultimately damage the freedom associated with the style.

2.2 Related Work

While there is a large body of research work on microservices, for this work we focused on the issues that are central to the intended goal. First, how to define boundaries in a monolith so that we can identify the most appropriate microservices to be developed. Secondly, how the process of migration was executed by other, what were their problems and how they have tackled them.

2.2.1 Identifying Microservices

Our path towards a microservices architecture starts by taking a look into the monolith we have at hands, inspecting dependencies in order to identify candidates for microservices. Decomposing a system into modules is not a recent concern, however the difference with microservices is that they need to be decomposed in a way that it allows for services to be independent reducing the possible dependencies.

This step on the process of migration is complex and requires knowledge and exploration of the system's architecture deserving a full chapter in Sam Newman's book [41]. In it the author uses concepts of Domain Driven Design to reason about the system's architecture and guide on where to define boundaries, or seams, that will help us define the bounded contexts of our application. Focusing on the Database layer and analyzing the application' schemas in order identify microservices by removing dependencies. Dependencies are removed by breaking foreign keys and serving static data in a different way. Foreign key dependencies can be resolved by moving data that needs to be shared between contexts into the responsible services so that an interface can be exposed. This will allow the data to be provided directly to third parties or will enable the development of a higher level service that can aggregate that data and serve it in a more consistent and completed way. While static data that needs to be propagated to different services needs to be moved into configuration code or into a new service to be provided consistently. These database refactoring operations introduce a new problem into the system related with transactions, and the author provides some guidelines to solve them. Due to the new distributed state of the data, transactions that used to happen in the system might require crossing the newly established boundaries, thus they can no longer happen in the same way. Assuring that either all or none of the planned actions are executed requires other mechanisms since they fall under the responsibility of different services. The solution to this problem will depend on the type of consistency that

the application requires. In the cases where there are more relaxed requirements, it is possible to aim for eventual consistency. Retrying the execution of operations at a later point in time which only requires a queue system or a log file where actions can be registered. However, in more exigent environments compensating transactions that undo certain operations or backend processes might be needed to ensure the consistency level required. On the other hand the distributed transactions mechanisms like the two-phase commit or Paxos also solve this issues but overheads will be introduced that need to be reasoned about.

In Levcovitz et al. [42] a system is described as a three layer application having user interfaces, the server side application and a database. The authors propose a five step methodology to identify microservices in a system adding an final sixth extra step to deploy them. Starting by decomposing the schema by mapping each table into subsystems where each subsystem serves a business area of the organization. A dependency graph is then created where 3 types of dependencies are represented. It is possible to observe the businesses functions that serve each facade, business functions that depend on others and lastly which tables have data that serve each business function. The identified facades and tables that serve them are grouped into pairs. As a result each subsystem composed by a set of facade-table pairs that are possible microservices candidates. For each pair the facade in the pair and dependent business functions business need to be inspected at a code level in order to understand if it can result in an actual microservice. A microservice is then defined by the purpose it serves, inputs received, outputs produced, features (business functions) it implements and databases it relies on. While it doesn't mention Domain Driven Design concepts we can see that it also departs from the same schema analysis as Sam Newman in his book and it also results in the definition of microservices using boundaries and contexts around business functions.

While the graph decomposition might be a good way to have an overview to the system helping on defining the seams, the authors propose to decompose a system into subsystems under the assumption that each subsystem has its own data store. However this separation isn't always clear in an application. Most of the times, monolithic applications share the same data store between different business capabilities, ultimately having tables that serve multiple of those functionalities. Particular situations like the possibility of shared table between subsystems or when different resulting subsystems need to cooperate in a given operations, are something that proposed method does not solve, requiring an extra effort when it faced with them.

In Mazlami et al. [43] a different approach works towards the automation of this process, a tool was developed to provide algorithm based recommendation for microservices candidates. To achieve these recommendations the tool starts by analyzing the code from a version control system. In the first phase of the process a monolith is characterized by a set of class files, a sequence of change events and a set of developers. This data is then used to build a undirected edge-weighted graph

representation of the monolith were classes and the coupling between are represented. This graph is built using one of the coupling strategies that are easily related with above-mentioned concepts. Logical coupling groups classes that are changed together while Semantic coupling groups classes that hold code relating the same ”things”. Both these first strategies work towards defining bounded contexts and defining components with a single responsibility. On the other hand contributor coupling clusters classes by incorporating team-based factors in order to reduce communication with external teams and optimizing internal communication, an application of the Conway's Law and better defining ownership. Finally, the graph is then processed by a clustering algorithm that will cut it into different pieces each one representing a microservice candidate.

The fact that this process is using class files as the atomic unit of computation to build the initial graph is described as a limiting factor. It is pointed that using functions instead could improve the granularity and precision of the code reorganization, thus providing better recommendations. Nevertheless, this is an interesting approach since it doesn't required the deep analysis of the schemas as the methodologies mentioned before and it can be executed fast, in order of minutes for logical and contributor coupling and hours in some cases using semantic coupling. Also it can use organizational factors which is important since restructuring teams will be later required.

In any work of relating the topic of splitting a system we will always be directed to Domain Driven Design concepts as described in Eric Evan's work [3]. Although these are not recent topics, the support for automation of these tasks is not yet at a mature level, in any case the representation of the system using graphs and performing a cluster analysis in order to identify microservices seams to be a good approach.

2.2.2 Experience reports regarding the adoption of Microservices

In order to understand how to adopt the microservices and design our approach we need to look into past works. There are some reports where authors relate their experience with applications in different business contexts but also the improvements they were able to achieve after the completion of the migration.

In Gouigoux et al. [44] the core business software of a company was rewritten from monolithic architecture to a web oriented architecture using microservices. Starting by reasoning about the ideal granularity of a microservice, unavoidably going through the Domain Driven Design Concepts as we did in the section before. As for their new architecture, it is pointed that the container technologies (Docker) were a great enabler making deployment easier and that orchestration was done by an ESB to bigger customers and HTTP based communication (using webhooks) for smaller ones. While there is no precise information on the actual migration, the report is more focused on the comparison of the initial and final state of the system, describing it as success that resulted in an overall improvement on the

application. The increase of reuse lead to less time of developing in cases of similar applications which ultimately results in lower financial costs. Replacement was also facilitated due to the new level of componentization what ultimately also results in a reduction of costs. Lastly, there were great improvements in the field of performance with response times reduced to times between 70 and 300 milliseconds when some of them used to take 1000 ms.

In a more critical context, the work of Dragoni et al. [45] aimed to reconstruct a critical system belonging to one of the largest banks in Europe. In it, the authors describe the need for scalability as the main reason for the migration and they present what they define as the requirements for achieving the desired level of scalability. Automation of management activities like testing, deploying, configuration, host provisioning and relocation of services. Orchestration in order to manage the infrastructure and services containers. Service discovery providing DNS lookups but also health-checking mechanisms and achieving geographic scalability by using location in the translation process. Load balancing to assure all replicas are used and that the load is distributed among them. Lastly, clustering to optimize the usage of resources and enable the elasticity of services to adjust to the load. Choosing the Docker as their containerization technology the authors covered a mentioned concerns using different technologies. Infrastructure automation was achieved by implementing a continuous delivery and continuous deployment pipeline using a GoCD server. GoCD is integrated with Docker Swarm which manages containers, this being able to provide orchestration, clustering and service discovery. Communication between services is handled by RabbitMQ which implements load balancing by distributing messages among each queue' subscribers. There was the need of developing extra services that implemented supportive functions: service logging, monitoring and configuration, but also data synchronization, failover handling and tracing. Comparing the initial and final architecture, with the new implementation the goal of increasing scalability was achieved while also resulting in a less complex system (to manage, deploy and administrate) with improved availability and reduced costs.

In Balalaie et al. [46] a Software as a Service (SaaS) application was redesigned due to new requirements. The initial architecture went through multiple refactorings until one that met all requirements was achieved. Initially pointing the components required to make the most of the microservices style we can find common points with other works, a configuration server, a service discovery, a load balancer, circuit breaker for fault tolerance and an edge server serving as an API Gateway into the system. The process of migration is then described in some steps, some of them related with the reshaping of the application's components into services but also some related with general microservices requirements. Among these last we have the introduction of a continuous integration pipeline using Jenkins as CI server, Gitlab for version control and Artifactory for storing artifacts. All services were developed using Spring Boot which offers integration with Docker. Continuous Delivery is implemented using Spring Boot capabilities, Spring Boot Configuration Server to keep source code and configuration separated and Spring Boot

Context to feed configuration values to builds. Clusterization was delegated to CoreOS, a linux distribution developed with container clusterization in mind and Kubernetes, a tool that takes in docker images and a policy file and makes sure those policies are fulfilled. Closing the work, the authors point the extra complexity introduced by the distributed stated of the system but also make some other repairs. It is stated that the technology freedom that the microservices enable should be handled carefully in order to avoid chaos, for that they recommend the usage of service templates for each technology stack on the system so that development starts from a common trusted point.

The mentioned experience reports do not actually dive in depth on the process of migration as one could expected. This can partially be because the process it self is very ad-hoc and related with the context, thus reporting it might not be of so much interest for the authors since they are focused on the improvements achieved. Nevertheless, it is possible to identify some requirements that need to be fulfilled for a successful microservices architecture as well as some technologies that answer to those concerns.

2.2.3 Patterns definition for the Adoption of Microservices

There have also been works that try to formulate useful patterns for the adoption of microservices, not only for migration of applications but also applicable to early development phases.

In the work of Balalaie et al. [47] the goal was to help developers and organizations planing the migration of their systems. In it, the authors document a series of situational patterns that are applied at different stages of migration depending on the context and problem targeted at a given time. Among these patterns we can identify some steps taken in works mentioned before. Enabling continuous integration, developing an edge server or introducing components for service discovery, load balancing and circuit breaking are all different migration patterns. But also the introduction of containerization, clustering, orchestration monitoring are present in the list. Some of the patterns are not described in depth but the technologies that can aid in each of them are mentioned, pointing developers the direction to go at each step of the process.

The Strangler Application method was coined by Martin Fowler [48] and the idea is to incrementally move functionality out of the legacy system while maintaining it. One key factor on the migration process is when and how retire the old system and replace it with microservices. Such crucial change should not be performed in a big-bang style but rather more slowly. There are multiple factors that can go wrong if one tries to completely replace the currently working system with the newly developed set of services. Often times an organization cannot handle such a big and sudden change smoothly and lack of experience with the style can lead to questionable decision making. By applying this pattern, features that were moved out and newly developed features can be introduced into the application but offered by a separate service, while the legacy system can continue to be maintained. An overview of the process

can be seen in Figure 2.5. This process needs to be transparent to the user and for that the first thing to do is to introduce a broker/proxy into the system. Since each functionality is offered by the in-retiring application or by an newly developed service, all requests need to pass through the broker in order to be forwarded to the right service.

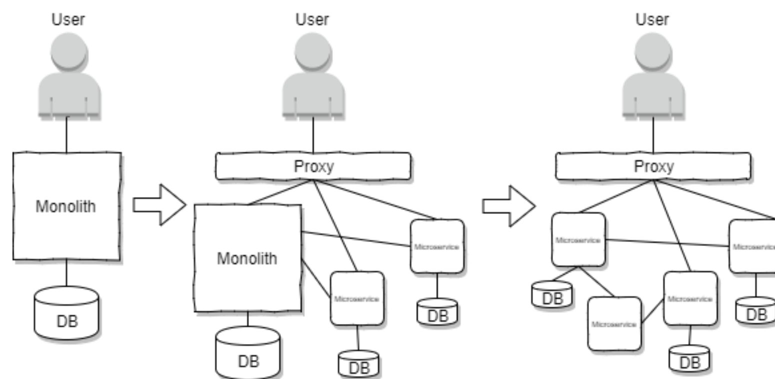


Figure 2.5: Process of applying the strangler application pattern

Initially the new services and the legacy system will need to communicate in order to obtain data to fulfill requests but the goal is to achieve a state where there is no need for that to happen and the legacy system can finally be retired [49]. Using this pattern organizations can, while developing, have faster feedback on their methods and decisions by evaluating the performance on a newly deployed service. This will result in higher quality software and better user's experience since there is the possibility of making adjustments early on the process instead of only realizing a bad decision when a large part of the system is on production.

With this previous research we were able to gain some useful insight before we actually perform the migration.

From the discussion about how to break the monolith we could understand that, at this moment, the only viable option to define mircroservices is by exploration work according the Sam Newman's [41] guidelines, thus applying Domain Driven Design concepts.

From the experience reports we gathered while, in most cases, we could not fully grasp the faced problems and found solutions we were able to gather some interesting mentions on useful technologies. Docker is definitely the top choice for virtualization while Jenkins can solve our problems regarding continuous deployment and integration. Kubernetes seems a very interesting tool that has been growing fast and is able to provide key features that can solve a lot of the problems regarding the management of the microservices environment.

Also as a result of this research, we believe that the Strangler Pattern can be the key to a successfully migration. The ability to receive fast feedback and do the migration gradually are definitely concerns that should be taken into account and the pattern provides the way to do it.

Finally, we believe that, with this work we will be able to provide an updated view on what a migration to microservices looks like. During these last years due to the microservices hype a lot of new tools have been developed, in a few months any tool can become obsolete. Now that this hype is stabilizing we can see the tools and approaches that have surfaced and are actually being widely used and accepted. Furthermore, we intend to explore and document the challenges faced during the migration and their solutions, something that, in most reports did not seam to be the focus.

3

Migrating to Microservices

Contents

3.1 Migration Fit . . . 27
3.2 Splitting the Monolith . . . 28
3.3 Serialization . . . 31
3.4 Communication . . . 33
3.5 Security . . . 35
3.6 Service Orchestration . . . 37
3.7 Moving to the Cloud . . . 40
3.8 Continuous Delivery and Continuous Integration . . . 41

As described before, there are a lot of concerns that should be considered during the process of migration. In this chapter we first discuss the requirements that make an application fit for the migration to microservices. Then, we document the main problems that we have faced during the migration and explain the decisions taken that, ultimately, led to the final microservices architecture. This work is more focused on the problems that need to be solved than the actual development process.

3.1 Migration Fit

In order to explore the problem of adopting microservices we first needed an application to actually migrate. The application we are using in this project for documenting the migration process is proprietary, belonging to Premium Minds[1]. Not all applications are suitable for a migration to microservices, they should present some certain requirements. In this section, in order to provide a perspective on the app and why it was chosen for the migration a brief description and insight will be provided.

3.1.1 Requirements

The app, named Rolodex, was started as a side project in the organization and since then it has been growing at a moderated pace with development and support being provided by different people on the organization. Recently, it grew becoming one of the main HR tools inside the organization offering functionalities to support personal data and vacation time management for all collaborators, as well as teams management in a centralized way. At the time there were plans for extensibility in order to offer new functionalities and due to the internal success of the application there were also plans for its productization. This meant that an application that was built for the scale of this particular organization would have to be able to offer more functionality to organizations with, possibly, more collaborators in the same way as before. Having already a moderate size, this recent requirements would result in a even larger code base, thus more complex to manage. With the newly planned functionalities the application was going to have multiple features to be developed while also maintaining the old ones and, as explained before, this would create dependencies in the deployment process resulting in possible delays to new releases. Finally, in order to offer this application to new clients there was the need to port the application to the cloud. So far it has been running as a single instance on premises since it is able to respond to the demand of the organization this way. However, a new client might have a superior demand, thus it is required to offer the ability to scale in order to respond to that.

In summary, these were the requirements that, in our opinion, made this application a good candidate for the migration from the traditional monolith that it was to a microservices-based architecture, the growing number of features, the move to the cloud and the ability to answer to a dynamic demand.

[1] www.premium-minds.com

3.1.2 Technology

With respect to technologies and frameworks, the application was built using The Scala Programming Language[2] making use of the Akka[3] framework, with a backend database in PostgreSQL and user interface built in Angular[4], the Javascript framework. Finally, the application was deployed using Docker containers, one for the web application and a second for the database. This was done in the organization's own resources and using Docker Swarm with Docker Compose configuration files.

3.2 Splitting the Monolith

The first step of the migration process is, as shown in the previous section, to define the new boundaries of the system by identifying microservices candidates so that we can extract them and build the new services. This is, for sure, the most important step of the migration since these boundaries will define how the final system will work and interact. If we find ourselves defining these seams wrongly it will affect the performance of the system and the complexity of the migration as well.

From the previous research we were able to identify some tools that would help automating this process, however we also concluded that those automating tools do not yet have the maturity enough to be used in production projects. In any case, any tool or approach identified in the research would always take us back to the core values of what it is to define seams in a system, and those were well expressed in the works of Sam Newman [41] and Eric Evans [3] in terms of Domain Driven Design concepts with database and source code analysis tasks. And that is what we decided to do.

In terms of database structure, efforts towards modularization were already applied in such way that all data was stored in different schemas with respect to its context. From a fast and careless analysis one could assume that from each schema a different microservices could be created, however that was not the case. Moving up in the application layers we ended up finding modules that would need to get from other contexts, mix that data and expose it that way. One particular case of this, is the data from the Teams and Collaborators contexts that in some cases would be mixed before exposed. Another slightly different case of dependency between data in the business layer has to do with the Security and Collaborators contexts, since each entity in the security context was directly linked with an entity in the collaborators context. While we could separate the first two contexts, due to the importance of security, this dependency could not be relaxed.

With this analysis work we concluded that we could define four different microservices, each one taking care of its own Bounded Context.

- **Core** - Handling Security and Collaborators Data/Operations

[2] www.scala-lang.org/
[3] www.akka.io/
[4] www.angular.io

- **Workgroups** - Handling Teams Data/Operations
- **Calendar** - Handling Vacation Data/Operations
- **Notificator** - Supporting service to send out notifications about events in the system.

These services are our small units of code, they answer to specific business needs and they can be used by any other tools to retrieve or update information. One example of this, is described in the following section.

3.2.1 User Interface Implementation

The ability that users have to adapt to new interfaces is a known limitation for every application designer, as such one of the requirements of this migration was that the application's UI should remain unchanged. After being recently redesigned it did not make sense for the organization to work on a new implementation simply because of this process of migration. This would also enable us to focus completely on finding the best structure for the back-end, which was already discussed in the previous section.

As mentioned before, this particular user interface was implemented using Angular allied with the capabilities of Scala.Js a scala library for front-end development. Using these frameworks and technologies allowed this user interface to be implemented as an external project. It is exposed through a server that simply provides the assets to the user's browser that ultimately uses them to generate the UI. The UI then runs on the user's local machine and fetches data and other assets by sending HTTP requests to the application's back-end that are exposed via REST interfaces. This is extremely important, since that, due to this implementation we were able to reuse 100% of the implemented interface needing only to update the request endpoints to each of the services address. Due to this reasons we were able to expose the UI in the same way as we exposed a service and a fifth microservice was defined:

- **UI** - Provides UI assets and fetches data from all other services

The designation of service is not very appropriated in this case since this piece of software does not implement any business function. However there is the need of this microservice to serve as an UI aggregator.

We started to look at this new 'service' and realized the resemblances that it has with a typical API gateway [50]. Besides providing the UI assets, by being able the serve information from all the services to the user by pooling each service individually,this new service, also hides the level of granularity forced by the microservices from the user, making most of the new distributed architecture invisible to the user. Those are, without question, the main goals of an API Gateway. Thus, due to this UI implementation it was also possible to solve a problem that would eventually surface.

While we see this case an advantage, some might see it as not so correct way to implement a microservices architecture since it makes the UI implementation totally independent from each service implementation. During the preliminary research, one thing that was mentioned as a point to achieve a good microservices architecture was that each microservice should be a vertical slice of the system, in such way that it would implement its own data access, business logic and user interfaces layers. This implementation does not exactly respects that, since there is an unique implementation to the whole system.

In any case, the adoption of microservices is all about trade-offs, balancing complexity with performance and effort, and this case we believe that it was the correct decision to make. The complexity that would be introduced in the process due to the modularization and separation of the UI elements would be far superior that any possible achievable gains. Another point for vertical microservices implementation was that it made possible the definition of cross-functional teams to manage a set of microservices, however, given the scale of this project a small team is easily able to manage and support the whole system.

3.2.2 Source Code Management

Still the context of splitting our monolith, we needed to redefine how the project's code structure was managed. In its initial state the application had all its code base in a singular git repository divided in several sub-directories and modules, hard to explore, navigate and keep track of dependencies. Furthermore, it would take around 4 minutes for the system to perform a clean compilation of the project. While it might seam not so critical, during development, the time spent compiling adds up to a large sum and by splitting the source code we were also expecting some improvements on this aspect. From the literature, we found that the most common approach to make the source code easier to manage was to create a new repository for each microservice. It makes perfect sense, since each service is now developed at its own pace and the idea is to decouple it from any other services, thus keeping code separated.

From the initial refactoring operations performed on the project we were also able to define a library with common and utility source code that every service could depend on to implement its own functionality. In the end, we created a new Git repository for each service' source code but also a new repository for this common library so that updates to it could be propagated to every service, forcing development process always depend on the most recent version. The way we chose to do this is by cloning this Common Library repository into the Working Directory of each service, but also making each service's repository ignore this particular sub-directory so that this library does not get pushed to the services' repository.

In conclusion, from a single monolithic repository, six new smaller repositories were created to com-

ply with the newly distributed source code. This would also enable us to define the Continuous Integration/Continuous Deployment mechanisms which are discussed later in this work.

3.3 Serialization

Long has passed the time when XML was the standard format to serialize the data to offered by services. There are now more efficient and readable formats that are, at the same time, more flexible. The problem at this point was to define how the data being offered by our services was going to be made available. At this point we were looking at two different options, JSON and Protobufs. There is an ongoing debate on which one is the best serializing format to current applications that we will briefly expose ahead but it is fair to say that each one has its advantages.

3.3.1 ProtoBuffs

Protobufs, short for Protocol Buffers, were internally developed by Google and later open-sourced. They are described as a language-neutral, platform-neutral extensible mechanism for serializing structured data [51], something similar to what XML offered but with much better performance. This is achieved having every message format defined using a schema, as shown in Listing 3.1 the schema definition for the class Employee allowing optional and mandatory fields and type specification. This schema model will be used for generating the necessary code for the usage of protobufs. This code is responsible for generating the binary data that actually gets transferred to other services, message format validation, type validation and also backwards compatibility. Any defined schema must be distributed to any service that pretends to consume the data available via protobufs, otherwise it is simply not possible making integration with other services a little more complex to manage.

In short, the performance offered by protobufs come at the cost of flexibility, since data has to respect the defined schema, and readability since the binary format that is used *on wire* is not readable for developers or any other entity except for the protobufs compiler.

Listing 3.1: Protobuf Definition for Simplified Employee Model

```
1 message Employee {
2     required string uid = 1 [(scalapb.field).type = "java.util.UUID"];
3     required string firstName = 2;
4     required string surname = 3;
5     required string email = 4;
6     optional string nickname = 5;
7     optional string birthDate = 6 [(scalapb.field).type = "java.time.Instant"];
```

```
8     optional string entryDate = 7 [(scalapb.field).type = "java.time.Instant"];
9     optional string companyPhoneNumber = 8;
10    optional string personalPhoneNumber = 9;
11    required string creationTimestamp = 15 [(scalapb.field).type = "java.time.Instant"];
12    required string updateTimestamp = 16 [(scalapb.field).type = "java.time.Instant"];
13 }
```

3.3.2 JSON

JSON, or Java-Script Object Notation, has been the most popular choice for a while up until the surge of ProtoBufs. Being outperformed by Protobufs, it tends to be chosen over it for its focus on flexibility and readability. Seen in Listing 3.2, a JSON representation for the same model as the protobuf implementation in Listing 3.1, that is the actual content of the message that gets sent. It is possible to see that some optional fields are not present, and all data is encoded as strings with no particular order. It is also possible to send more fields than expected since there is no defined structure. On the other hand, there is also no assurance that the fields that one service might need will be present and that might result in deserialization errors. The great bonus from flexibility comes from the policy that is normally used in JSON-based services, *'expose everything, consume what you need'*. In which every service makes all its data available and well documented for external sources but any service that will consume that data will only consume the fields that it requires. This is only possible because due to JSON serialization/deserialization processes. While it does not have impact on the owner of the data, it makes processes of consumer services more simple.

In conclusion, JSON's flexibility is the big key point, making integration with other services external or internal simple and without creating any other dependencies.

Listing 3.2: JSON representation for Simplified Employee Model

```
1 {
2     "email": "tiago.santos@org.com",
3     "creationTimestamp": "2018-05-11T14:09:43.818Z",
4     "personalPhoneNumber": "",
5     "updateTimestamp": "2018-05-11T14:09:43.818Z",
6     "surname": "Santos",
7     "firstName": "Tiago",
8     "uid": "91bb39d8-852e-4a5e-a99c-9fd09593352a",
9 }
```

Even though some implementations for protobufs models were already in place, our final choice was to use JSON. Most of the data offered by the services in this project is to be consumed by the User Interface which is a Javascript-based project and this is the environment where the performance discrepancy between JSON and Protobufs performance is less noticeable [52]. One other advantage is the library support for JSON. Most languages already have different implementations for JSON's processes, for the library in use, Akka, JSON is already the default serialization format, while implement protobuf serialization would require more efforts.

3.4 Communication

Due to the new distributed architecture of the system it has become more verbose. Requests that did not happen before are now essential to the proper functioning of the application. In a microservices ecosystem, as in any distributed system, there is the need to distinguish the communication types, this could be due to its purpose or simply due to time constraints.

3.4.1 Synchronous Data Requests

On one hand we have requests that are generated in order to fetch data from other services so it is possible to produce a response to a given request as exemplified in Figure 3.1. These are synchronous requests that happen via HTTP and are something that should be carefully handled, since its from here that we start to define services dependencies and putting our services availability in a sensitive position. In our case, we have developed every service that we are handling with, meaning that the availability of our services is only dependent on our other services. Thus, we have more control over our environment which makes the availability problem is less aggravated. In our case this type of requests only happen where we initially identified data dependencies (Section 3.2). That is for security purposes (authentication across services) which we will address later and due to the Collaborators/Teams dependency. One possible solution to this would be to replicate data across services but it did not made sense since it would practically mean to re-merge the services.

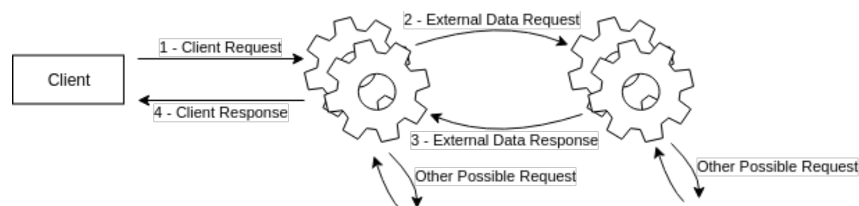


Figure 3.1: HTTP Request Chain

3.4.2 Asynchronous Event Propagation

On the opposite side of the spectrum we have a new kind of introduced communication that has the goal of propagating data updates in order to assure data consistency across all services. The best way to make this asynchronous communications happen is by using an event-driven pattern and message queues. The need comes from the requirement to keep all the data up to date, all the time. For example, in our case, if an employee is removed from the system, all his vacations in the Calendar service should be deleted and he should also be removed from all his teams on Workgroups.

3.4.2.A Event Sourcing

Event Sourcing [53] is a particular pattern on the event-driven playbook. At its core, it relies on state propagation to assure consistency. The idea is that on a service's action an event is generated with the information about that state change and its gets sent to other services and stored in an event store, or journal, allowing event queries, event replays and service state reconstruction.

In our implementation, each service generates an event for every Create, Update and Delete action over its entities. As seen in Listing 3.3, the class Event has two attributes, correlationID, which can be generated on the event creation or reused from other event in order to represent causality between events and a name, which represents the action that triggered the event. All other event classes extend this class adding the specific data about the state change, this way any service that receives the event has all the necessary data to act accordingly, creating, updating or deleting its own data with respect to the changed entity.

Listing 3.3: Events Implementation

```
1  trait Event(correlationID : UUID, name : String)
2
3  case class WorkgroupCreateEvent(uidWorkgroup : UUID,
4                                  employees : Seq[UUID],
5                                  isManager : Boolean,
6                                  correlationID : UUID,
7                                  name : String = "WorkgroupCreated")  extends Event
```

3.4.2.B AMQP

The Advanced Message Queuing Protocol[5] is an open standard for passing business messages between applications, it defines the implementation of the middleware necessary for message-oriented

[5] www.amqp.org

communication. The most widely used open-source implementation of this protocol is RabbitMQ[6], which provides a lightweight and easy to deploy message broker for any purpose. This is the technology we choose to use to assure the delivery of our event-based messages to our services.

RabbitMQ defines the concepts of routing keys, topics and bindings that allow us to use the system to delivery a set of messages to the recipients that desire them, in a Publish-Subscribe style. In RabbitMQ this is called Topic based delivery. Represented in Image 3.2, the RabbitMQ process for this topic based delivery. Any client that wishes to receive messages, services S1 and S2, creates a queue, qS1 and qS2 respectively, and then contact the broker system in order to create a binding, that is, associate a topic to that queue. Service S1 wishes to receive any message with topic started in calendar and Service S2 only wishes to receive events with the topic workgroup.delete. When any message is delivered to the broker it has a routing key, the broker will then deliver that message to any queue binded to the topic that matches the key. In this case, when service Sp generates a message and delivers it to the broker with routing key calendar.event.create, it will only be delivered to service S1, the only one who needs to act over this state change.

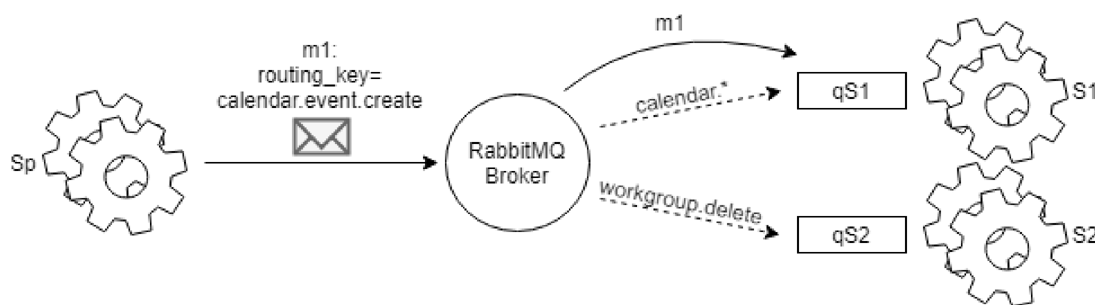


Figure 3.2: RabbitMQ Topic Based Delivery

3.5 Security

One of the key concerns during the migration was how to secure this new ecosystem composed by multiple services, and where all the security concerns are only answered by one of them.

One simple, but also naive, way to do this was to make every request go through the Core service, since that is the service with the implementation for security directives, serving as a proxy for any of the other services. After verifying identity it would forward the request for the proper service. This would also allow the limited exposure to external traffic, having one single service exposed to the outside network while the others would work in a private network. However, this would bring extra load to the Core service which would affect its performance, besides, every service needs be able to identify the user making changes to data so it can be registered. Ultimately, this would not work for us.

[6] www.rabbitmq.com

This process was simplified since this application was already the main tool for authenticating and authorizing users and applications to access resources across the organization. As such, there were already mechanisms in place to assure this properties. Authentication/Authorization across the organization's tools was implemented using the OAuth2 protocol[7], which is one of the main choices for implementing security mechanisms in distributed environments.

A description of the authentication process is shown in Figure 3.3. The Client application, will authenticate an user (resource owner) via the browser (user-agent) and after the authentication process, steps 1 to 6, will receive an access token which it will be able to use as an identifier for future requests proving the user's identity. In our system, our authorization server is the Core service. It also acts as a resource server for the Collaborators data, but authentication in that scope is not a problem since it happens locally. However, that is not possible in the new services, since they cannot verify identities. Thus, they have to request to the Core service to verify the tokens that are sent with the requests that they receive. For this, we had to extend the Core service interface defining a directive to verify if a given token had rights to perform certain actions. Finally, on every request, every service will poll the Core service for validation of tokens, this is a case of the aforementioned Http Synchronous Requests (Section 3.4.1).

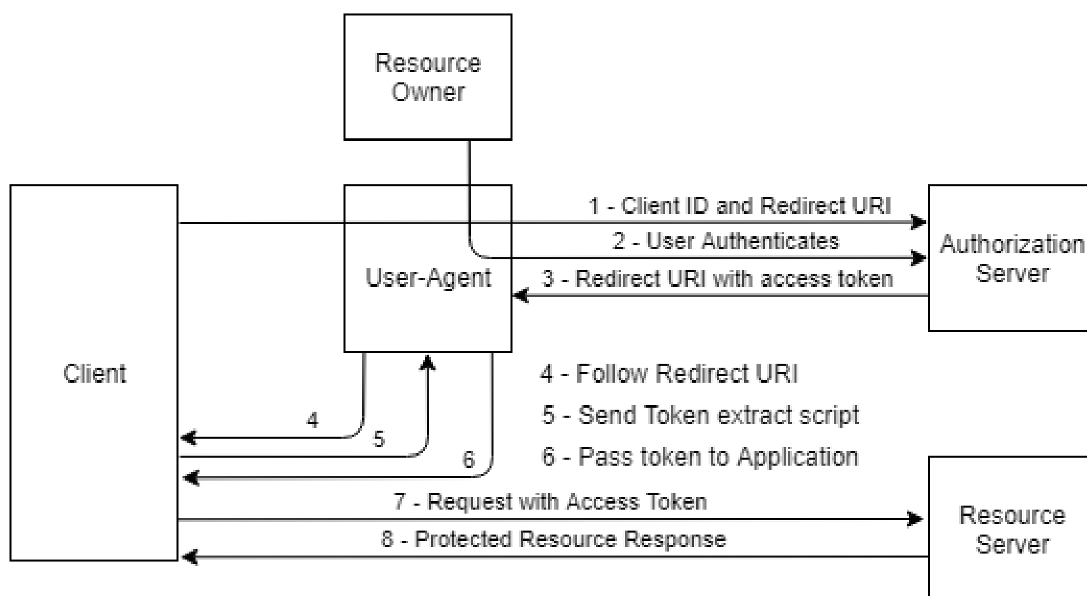


Figure 3.3: OAuth2 Implicit Grant Flow

[7] www.oauth.net/2/

3.6 Service Orchestration

Having our initial application deployed as Docker containers confirmed that that was the way to go. Due to increase of popularity of Docker, the Docker ecosystem has also been growing supported by a large open-source community. One of the most recent Docker power tools is Kubernetes[8]. It allows us to ease the deployment process while offering capabilities of scaling and management like logging and monitoring, but also service discovering and networking.

3.6.1 Kubernetes

To fully understand Kubernetes and how to deploy an application using it there are some concepts that we first we need to explain.

- **Pod** - Is the smallest unit of deployment in Kubernetes, it represents a running task in the cluster. Typically, a running container. Pods are mortal getting deleted after being stopped and are impossible to resurrect.
- **Deployment** - Provides declarative updates for Pods. We can define a desired state and the Deployment Manager will enforce that state over the managed Pods. Increasing or decreasing the number of replicas of a pod for example.
- **Service** - Is a level of abstraction behind Pods. Defining a set of pods and a policy to access them. While Pods' might be built and destroyed they cannot be relied on. A service is 'eternal' and always provides an entrypoint to any of the pods it is responsible for.
- **Volume** - Is an abstraction for a file system. When a Pod/Container is destroyed all files within are also destroyed. However one might require persistence, volumes solve that problem.
- **ConfigMap** - Is an object that stores configurations to later feed them to containers, possibly via volume mounts.

All thess Kubernetes' objects pods run on or are stored in a cluster, which can be managed via the Kubernetes API. Feeding the API yaml configuration files so it can create and deploy the desired objects.

3.6.2 Deploying our Services

As typical in a microservice deployment, each service should have its own database. However this database should be persistent, a service might need to be stopped and re-deployed and data should not be deleted on the process. As such, just like every service, every service's database is deployed in

[8] www.kubernetes.io

a separated container. In order to assure the persistence of all the data in the database a (persistent) Volume is created so that Kubernetes API can reserve the required space in the cluster for the data. Besides that, a Service and a Deployment is created for the purpose of deploying a database service, this type of services are internal to the cluster, being impossible to connect to them from the outside. The application service is then linked to the database service via a configuration by providing the service's internal name that Kubernetes is able to resolve using its own DNS. In Listing 3.4 we can see a sample configuration for the persistence storage of a service. It requests 1Gb of space from the cluster and the 'ReadWriteOne' access policy defines that only one pod can read or write at the same time in the volume.

Listing 3.4: Generic Database Service Volume Claim

```
1  apiVersion: v1
2  kind: PersistentVolumeClaim
3  metadata:
4    name: service-pv-claim
5  spec:
6    accessModes:
7      - ReadWriteOnce
8    resources:
9      requests:
10       storage: 1Gi
```

Every other service is exposed using a Service of type LoadBalancer which allows the exposition of the service to outside of the cluster and enable some load balancing capabilities across pods belonging to that service. Seen in Listing 3.5 the configuration for the Calendar service. It will take all pods with name 'calendar-service' under its management, create the DNS entry to link the name 'calendar-service' with its IP and finally, expose the port 9884 externally for TCP traffic and forwarding it it to the port with same number (targetPort) of the created pods.

Listing 3.5: Calendar Service Definition

```
1  apiVersion: v1
2  kind: Service
3  metadata:
4    name: calendar-service
5  spec:
6    selector:
```

```
7       app: calendar-service
8       tier: backend
9     ports:
10      - protocol: TCP
11        port: 9884
12        targetPort: 9884
13    type: LoadBalancer
```

A deployment is then required to define the pods what will actually run the service containers. Following the example line, in Listing 3.6 the Calendar Service Deployment. Declaring the name and type for service, and setting the number of desired replicas witch the Deployment Manager will enforce, in this case, only 1. Then much like a typical Docker set-up we set up the container deployment. Defining the container image and how often it should be pushed from the repository, always meaning that everytime a pod is created the image should be pushed and also its exposed ports, 9884 in this case. Finally have a ConfigMap object called 'calendar-conf' that includes all configuration files for the Calendar service application and that is mounted under the '/data' directory, where the application is expecting those files.

Listing 3.6: Calendar Deployment Definition

```
1  apiVersion: extensions/v1beta1
2  kind: Deployment
3  metadata:
4    name: calendar-service
5  spec:
6    replicas: 1
7    template:
8      metadata:
9        labels:
10         app: calendar-service
11         tier: backend
12     spec:
13       containers:
14         - name: rolodex-calendar
15           image: repo/some:image
16           imagePullPolicy: Always
17           volumeMounts:
18             - name: calendar-conf-volume
19               mountPath: /data
```

```
20            ports:
21              - name: http
22                containerPort: 9884
23        volumes:
24            - name: calendar-conf-volume
25              configMap:
26                name: calendar-conf
```

Finally, an additionally service and deployment were required so we could create our RabbitMQ deployment, exposed as the same way as the database services, only accessible from within the Kubernetes cluster.

3.7 Moving to the Cloud

Having our deployment style defined, we could now think about how we could apply it in a cloud provider so we could free our own resources and prepare the product for other companies. One of our main concerns was to avoid vendor lock-in, and of course we would also require a provider where we could deploy our services in the way we previously defined, using Kubernetes.

We initially thought about Amazon was the provider that best fitted our interests. Mainly due to all the embedded features and flexibility, but also because of its large popularity. AWS has a market share on the cloud provider scene significantly larger than any other provider. Although, after defining our deployment style with Kubernetes, we found out that to deploy it in AWS it would require more configuration to set up the necessary infrastructure for Kubernetes to run autonomously.

Ultimately, we ended up choosing Google Cloud as our cloud provider. Google has been involved in the development of Kubernetes since the early stages and they have built the Google Kubernetes Engine which is basically a Kubernetes deployment in Google's cloud infrastructure. This means that they should be more capable to integrate their services with the ones provided by Kubernetes, while also being the first ones to adopt new features and versions of Kubernetes as they are released. Furthermore, from running Kubernetes locally, we would already have all the necessary resources to manage GKE, and it was possible to port our deployments to GKE with no changes being required. Finally, Google Cloud Platform turns out to be the provider with cheaper prices for a Kubernetes instance, charging only for the resources visible to the user, the nodes that run our services, while cluster management and other services are free of cost.

So we could be able to run our microservices in GKE we created a 2-node cluster composed by 'n1-standard-2' machines, making up a total of 4 vCPUs and 15 Gg of memory, which was more than enough to run all our services (on a single instance policy). Note that this cluster was created with no

special concerns for the optimization of resource usage or minimizing costs as we were making use of the free trial that Google offers to experiment their services. Kubernetes is then responsible to allocate resources in an efficient way as long as there are resources to do so. Even when a cluster is scaled up or down, Kubernetes will adjusts the running pods to make better use of the resources available, or delete them in the case where resources are not sufficient to support the infrastructure.

Finally, to build our services in our GKE instance we simply needed to make a local Kubernetes engine to connect to the created cluster, this was done using Google Cloud' SDK. Then we could simply use Kubernetes commands as if we were working in our local machine and access the exposed services from anywhere via the IPs that were created by the engine.

3.8 Continuous Delivery and Continuous Integration

Now that we have our services defined, source code correctly separated into different repositories and our deployment figured out we were able to automate the management processes. The goal was to reduce human involvement in the processes, making them run faster and smoothly while assuring its correctness and making sure that everything that ends up at user's reach is validated and works as expected. From early in the process we knew that Jenkins[9] would be our automation system for these processes, it was already used on other projects, is open-source and is configurable/customizable.

To deploy our Jenkins server we, again, used Google Cloud's resources. We created a VM instance where we installed and configured Jenkins and all the necessary tools to perform the deployment actions actions like Git, Sbt, Docker and Kubernetes.

What we wanted to do with Jenkins was to push new features or fixes immediately to the production environment. That is, on source code changes (assuming that they are correct and respect company's standards) it would ultimately force the update of the running containers in Kubernetes' services. For this, we created a pipeline that would be triggered by a commit action on Github's repository (via web hooks) and that would follow the steps that we would manually need to perform. As seen in Figure 3.4 it starts by fetching the common library from the code repository so it can compile the project. Then it builds and pushes the new Docker container image to the Docker Image Repository. Finally, it updates the configurations configurations and update the deployment in the Kubernetes cluster.

One great advantage of Jenkins is that its pipelines can be configured through special configuration files, Jenkinsfiles. We can code our pipeline in one of those files and add it to our source code repository. Pipelines still need to be created from within the Jenkins tool, but we can point the Jenkinsfile in the code repository so Jenkins can configure the pipeline before running it. This allows us to add or remove steps to the pipeline by editing the file, being able to manage the build process as code.

[9] www.jenkins.io/

Figure 3.4: Successfully built Jenkins Pipeline

4

Evaluation

Contents

4.1 Size 45
4.2 Complexity 47
4.3 Coupling 48
4.4 Cohesion 51
4.5 Performance 52
4.6 Infrastructure Cost 53
4.7 Speed 54

In this chapter we will take a look at our system, how it evolved and how it compares to the state it initially was. Using well defined metrics to classify both states of the system and objectively compare them so we can clearly expose the effect of the migration on this product. From further research we were able to gather different metrics used to evaluate service based applications. These measurements are important because they provide us more information about the impact of the migration, and on the overall quality of the resulting system.

We defined five key attributes that in our opinion are relevant to understand the quality of the software. When adopting microservices, we are aiming to achieve a more manageable application, that has a lot to do with Size and Complexity. Microservices are also characterized by being highly decoupled and having high cohesion so these are also factors to measure. The last metric for quality is Performance. With the migration it is important to be able to, at least, be able to achieve similar levels of performance.

Additionally, we decided to measure two other factors that are more organizational oriented that we think that are also interesting to understand the impact of the migration. From research we found that costs seem to be a result of the migration, so we think that this is interesting to document. One other interesting factor to document is speed of delivery, with microservices we expect processes to be faster, and the speed of delivering features to the user is a key to see how adaptable is the system.

On the next sections we describe the quality in evaluation, why it is important and then expose the used metrics and the measurements performed.

4.1 Size

The size of a service has direct impact on maintainability in such way that the bigger the services, the harder they are to maintain. In a microservices based system the size of each service tends to be smaller that general, however it is still important to evaluate this metric in terms of average and deviation. Having a service that has a larger size that the most hints that it might be a service in need of refactoring operations, possibly a process of separation.

Lines of Code

The Lines of Code (LOC) is, usually, the metric to use when we want to describe the size of a system, or application in general. It can provide us with a general overview on effort required for change as in, a system with more Lines of Code, may require more effort to extend or fix. In order to measure the lines of code of each service and the common library we used an open source tool, CLOC[1] on a freshly cloned repository so that code from the build engine and dependencies would not get considered.

[1] www.github.com/AlDanial/cloc

Table 4.1: Lines of Code per Service

Core	Calendar	Workgroups	Common	UI	Total
7293	1294	1029	1400	36680	47696

In Table 4.1 we can see the Lines of Code per service and Library. Comparing the backend services, we can see that the **Core** service is the one with more lines of core. The reason for this is that this service implements more functionalities that the other backend services. Furthermore, security concerns and its implementations tend to be more delicate and complex, thus more verbose. The **UI** service is then largely denser than any other service, but this is due to the early decision to not make changes in it which also means that no refactoring operations were performed. Finally, we have our **Common** library, it is only used by the backend services which means that we were able to refactor the backend code and make about 12% of it reusable in every service.

Considering that the original application had a code base with 48875 LOC, we noticed a decrease in the code base of 1179 Lines of Code. We attribute this disparity to the lack of implementation of the other backend service identified, the **Notificator** service, which would have a similar size to the other backend services.

This reduction on Lines of Code is not very significant, but we attribute this to the fact that the monolith was already developed following good practices, thus being able to achieve a good level of modularity.

Weighted Service Interface Count

Weighted Service Interface Count (WSIC) is defined as the weighted number of exposed interfaces or operations per service as defined in the WSDL documents [54]. While we do not have WSDL documents to establish the interface contracts in our services, we do expose our operations in a maintained way and support every operation and its documentation to assure usability. Furthermore, HTTP methods as used in our interfaces are a basic method of communication on the web, thus we are not attributing weights to our operations, always using the default weight of 1. This metric much more service oriented than Lines of Code since it enables us to infer the size of a service through the interfaces or operations that it exposes. Something not possible by looking at the number of lines of code. Note that with this metric we will only be analyzing the backend services since the **UI** service is only a consumer of the other services operations.

Table 4.2: Weighted Service Interface Count per Service

	Core	Calendar	Workgroups
Interfaces	3	1	1
Operations	16	7	7
Weighted Service Interface Count	19	8	8

WSIC(Avg) = 12

In Table 4.2 we can see the interfaces and operations per service. Since the **Core** service has a wider operational context that any other service it also exposes more operations via more interfaces that is why the WSIC metric is so high for this service, this might be a hint that this service might need more refactoring operations. The values for the remaining services do not express a particular meaning as they are closer to the average. In a small context like this the WSIC metric might not provide a lot of information, but it becomes more interesting as the system size grows since it should be used to compare a set services.

Evaluating size is important in a microservices environment, while microservices should not have a fixed standard size, its still important to identify outliers and understand why they are in such condition. Using both, Lines of Code and Weighted Service Interface Count per Service, we can get a more accurate and insightful information. Comparing values with the average ones and calculating the ration between these two metrics enables us to identify services that might be over-growing making them possible candidates for splitting.

In this aspect the differences from the initial to the final state are not so significant, a lot of the code was reused and the operations that each service initially had, were obviously kept. No significant changes on implementations were made to make us comment on the evolution of the size of the system.

4.2 Complexity

Complexity is another important aspect of the system, also relatable with size. It also has impact on maintainability, growing complexity makes a system harder to manage. We have already discussed the complexity introduced in the system with the adoption of microservices. The following metrics try to express the complexity of each service which will consequently make up the complexity of the whole ecosystem.

Total Response for Service

The Total Response for Service (TRS) is defined in terms of Response for Operations (RFO) that represents the number of operations that need to be executed to respond to a particular request. The TRS is the sum of RFO for all the exposed operations of a service. Lets assume we have method A, that to produce its return value it performs calls to method B and method C. In this case, A has a TRS of 2 because it performs 2 calls to other methods. If either the B or C method performed calls to other methods the value would be higher.

We can express complexity with this metric by understanding the size of call chains in our application. Higher values mean longer call chains, which ultimately hint at more complex operations and services.

Table 4.3: Total Response for Service per Service

	Core	**Calendar**	**Workgroups**
Average Response for Operation	8.3	5.2	5.6
Total Response for Service	117	37	39

TRS(avg) = 64.3

In Table 4.3 we can see the calculated values for TRS while also the average RFO for each service. Here we can see that while the average response for operation for the services is somewhat linear, however the disparity in the value for TRS is more significant. This is because the **Core** service implements more operations that the other services. In the previous section we have concluded that the **Core** service was bigger in size and argue that it was due to the complexity of implementation for the security concerns. By calculating this metric we could confirm that argument.

Evaluating the value TRS by itself might feel somewhat abstract, but just like size metrics mentioned above, this value is to be used in service comparisons, calculating the average and identify outliers.

Other Metrics

There are also two other metrics that while interesting, are not actually applicable in our context. The first, **Number of Versions per Service** is defined as the number of versions that are used for a service The more versions each service in the system has, the more complex it is to manage. In our context, we only have one version for each of our services, but as the system grows and evolves backwards compatibility might become a concern, and this is where versions come into play.

The second metric is **Service Support for Transactions**. More specifically, the amount of service that support or are aware of distributed transactions. There are no distributed transactions in our system since we chose to use asynchronous events and provide eventual consistency.

We can measure both these metrics but they do not have an expressive value in our context. The application in question has not reached the point where it needs to incorporate these mechanisms to respond to requirements.

4.3 Coupling

Coupling measures the level of interdependencies and interconnections between services. In a microservices-based system, services should be loosely coupled. It is also a factor that impacts maintainability, the

more dependent services are, the harder they are to manage and maintain. The following set of metrics express those properties allowing us to identify bottlenecks and failure points in our system.

Table 4.4: Service Dependencies in the System

	Core	**Calendar**	**Workgroups**	**UI**
Core			x	
Calendar	x			
Workgroups	x			
UI	x	x	x	

Services Interdependence in the System

Services interdependence in the system, SIS, is defined as the number of services that depend on each other. Introducing dependencies is a delicate topic in systems in general, on the other hand in a microservices environment a solo service probably cannot offer the some function without having to fetch something from other services. In any case, introducing circular dependencies, depending on the use case, we might be entering a very dangerous field since the lack of availability of one service becomes even more noticeable.

As seen in Table 4.4, in our system, there is **one** circular dependency. It is observed between the **Core** and **Workgroups** services. Since it is the **Core** service that implements the security concerns, all other services use its exposed operations. The circular dependency happens because some of the operations that handle collaborators data in the **Core** service, need data from the **Workgroups** service which already depends on the **Core** as mentioned above. Ideally, there should be zero circular dependencies in a system, which hits us that we should look at these two services again and try to understand if it makes sense and what can we do to break this dangerous dependency.

Absolute Importance of the Service

The Absolute Importance of the Service (AIS), expresses the number of clients that invoke at least one operation of a service. This will allow us to understand which services are more essential to our system and how they can better be optimized to minimize any negative impact on the system.

Table 4.5: AIS per Service

	Core	**Calendar**	**Workgroups**	**UI**
Absolute Importance of the Service	3	1	2	0

AIS(avg) = 1.25

Inferred from the values on Table 4.4, the values in Table 4.5 confirm that our **Core** service is the most important of the system, due to the reasons stated before. Like other metrics above, this becomes more interesting in a larger system, where by calculating the average of this metric we can identify outliers.

Absolute Dependence of the Service

In a similar scope to AIS, the Absolute Dependence of the Service metric, looks at dependencies from the opposite direction, it describes the number of services that a certain service depends on. That is, the number of services that a service S invokes operations from.

Table 4.6: ADS per Service

	Core	**Calendar**	**Workgroups**	**UI**
Absolute Dependence of the Service	1	1	1	3

ADS(avg) = 1.5

Once again, in Table 4.6, we can see confirm what has been described before, while this metric does not add much after analyzing the AIS values, it will contribute to the definition of the next metric.

Absolute Criticality of the Service

The Absolute Criticality of the Service (ACS) can provide us an overview on the coupling of the system using the other metrics mention in the section. Calculated from the multiplication of both AIS and ADS values, it will make explicit how critical each service is. This helps defining how much effort should be invested in each service, to clearly identify bottlenecks, and help us tune the system to offer better response. Since that services that are invoked from other services, but make invocations to other services tend to be the most potentially problematic parts of a system.

Table 4.7: ACS per Service

	Core	**Calendar**	**Workgroups**	**UI**
Absolute Criticality of the Service	3	1	2	0

ACS(avg) = 1.5

We can state that the **Core** service is the most important part of our system and at the same time its bottleneck. However, there is not much we can do to about this aspect. The security concerns implemented by this service are core to the functionality of this system and that cannot be avoided. On the other hand, we cannot say that **UI** service is not important or critical, since it is the only way that

users can use the application. We have discussed that this is not a typical service, as such we should be too strict when looking at this metric.

These metrics and its values need to be seen through the right perspective so they can provide useful insight. The best way to do that has been mentioned and it is by analyzing the average values for each metric and finding outliers. Those should be the candidates needing improvements and better tuning. The goal is that the coupling metrics have low values, however such thing might not be possible and trying to set low standards will do more harm than good to the system, making developers finding ways to circle this problem.

4.4 Cohesion

A microservice is (or should be) a service that acts only within a bounded context. That is what cohesion tries to evaluate, to which extent the operations of a service contribute for one only functionality. A highly cohesive service is good goal in system design and it positively contributes to the system maintainability.

Service Interface Data Cohesion

This metric expresses cohesion of a service by analyzing the parameters data types for every operation exposed by a service interfaces. Service Interface Data Cohesion (SIDC) is a ratio between the number of operations with common data types and the total number of discrete data types that the service's interfaces knows. Having values between 0 and 1 where values close to one represent a higher cohesion.

This metric was much more significant in the SOAP era since operations were defined over the web by explicitly defining its parameters. In the REST paradigm this metric is also related with the serialization mechanisms and, besides being hard to measure it its value is arguable. For this reasons we have have decided to relax this metric.

In a microservices-based system the common practice is to expose HTTP interfaces and use a lightweight format to communicate, in our case it was JSON. All operations from all our services can only ingest JSON, which means that we are able to achieve a value of **1 to all our services**.

Service Interface Usage Cohesion

This cohesion metric, evaluates invocation behaviour of clients that use a service. A service is considered more cohesive if all its operations are used by every consumer service. It is defined as a ratio between the total number of operations used by every consumer and the number of its defined operations multiplied by its consumers. Just like SIDC, it has values between 0 and 1 and values close to 1

indicate more cohesion.

Table 4.8: SIUC per Service

	Core	Calendar	Workgroups
Service Interface Usage Cohesion	0.375	1	0.43

SIUC(avg) = 0.60

In Table 4.8 we can see the values calculated for this metric to our services. Knowing that the **UI** service uses all operations defined in all services, the low value for the **Core** service is due to the security aspects since every other service only uses a maximum of two operations from this service. On the other hand, the **Calendar** service has the highest service possible since it is only used by the **UI** service, while the **Workgroups** service has a still lower than average value but there is no space for optimization since the **Core** service only needs to invoke one operation from it.

Total Service Interface Cohesion

Finally, in the TSIC metric tries to provide an overall look over cohesion. It is a normalized sum of the two previous metrics allowing us to evaluate a bigger picture for each service instead of focusing on just one aspect.

Table 4.9: TSIC per Service

	Core	Calendar	Workgroups
Total Service Interface Cohesion	0.69	1	0.72

TSIC(avg) = 0.80

From the values observed in Table 4.9 we can conclude that our services' cohesion is acceptable throughout our services. These metrics should be evaluated using averages and we believe that the average value is very acceptable. Now, there is a case to be made about the **Core** service, its cohesion can probably be improved through refactoring or even splitting operations.

4.5 Performance

From a user's point of view using the newly architected application or the old one feels just the same, a non-informed user would not be able to notice the difference. This is mostly due to the user interface which is the same. However, performance is what influences the user's perception of the application, so it is important to make sure that, with the migration, there were no negative effects.

In order do evaluate performance, we have defined a set of workloads to be executed by a remote client in both applications and compare its response time. Note that both architectures were deployed under the same conditions, using Google Cloud's resources. For the microservices architecture, we used the deployment described in Section 3.7, while for the monolith we have created a VM instance with the same RAM and CPUs.

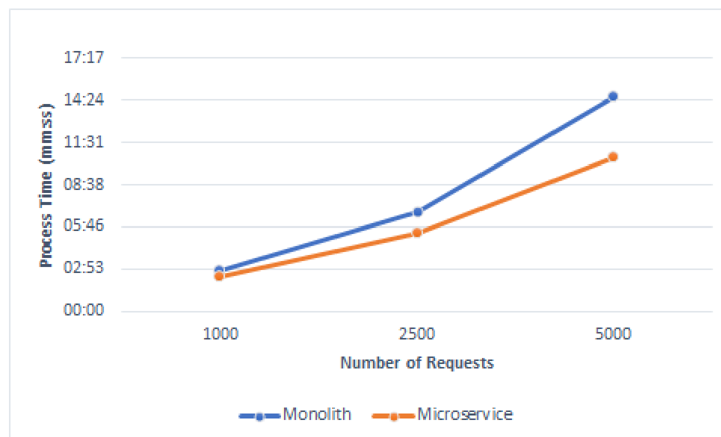


Figure 4.1: Workload Execution Chart

Seen in Figure 4.1, the graphic showing the execution time for each of the workloads, with increasing number of requests, 1000, 2500 and 5000. The requests used are all for write operations on the most expensive operation, creation of new employees. We can see that the **Core** service outperforms the monolith. The difference grows as the number of requests increases.

With microservices we are able to achieve good performance levels while at the same time being able to adapt and scale each service when needed. We can then assure that the performance will always be acceptable and that it will not suffer from execution in a different context is important to assure that it can scale to accept other kinds of loads. This is enabled by both, Kubernetes and Google Cloud tools. Kubernetes is able to scale service deployments by looking at the CPU usage of the pods executing it, while Google Cloud's Cluster Autoscaler can scale the resources cluster when Kubernetes finds itself unable to provision new pods. Ultimately, it will always be possible to require more resources dynamically to respond to any kind of usage load, not without adding up extra cost, of course.

4.6 Infrastructure Cost

Cost is also a factor to be considered. In a microservices ecosystem, having services being deployed in a cloud provider carelessly might have a big impact on the overall cost of an application. As mentioned before, compute resources and now a cheap commodity, however its easy to add up an expensive bill by with a growing system.

Our initial application was deployed as a single instance on the organization physical resources, so there was virtually no costs associated. However, with the productization the best choice is to make the application available using a provider's resources.

With our current deployment (which could be optimized) we would be looking at a monthly cost of **140 EUR** for the computing instances. This would be the base cost, there would then be extra cost for persistence (~0.04 Eur/Gb) and network traffic received by the Load Balancers (~2 Eur/Gb) but it would be much less significant.

4.7 Speed

Microservices are also about speed. More specifically, speed of delivery which can be defined as a combination of speed of developing and speed of integration/deployment.

One of the big advantages of the introduction of microservices is that the application is now a set of small(er) services. In small contexts it is easy to conclude that development happens faster, since it should be required less effort to implement new features. While we were not able to measure this aspect of the migration, we can argue that, a developer that was familiar with the application should able to deliver new funtionalities at a good pace, and that the migration will probably not have much of an impact on his performance. However, introducing a new developer to the product becomes easier, and thus he will be able to bring value faster.

On the second aspect, the speed of integration, we have clear improvements. With our CI/CD pipeline implemented using Jenkins we were able to fully automate the process. After a feature was completed and a new release approved, an artifact was generated to be deployed. However until it would finally be deployed, it would require communication with people from the infrastructure team so they could fit the deployment of the app in their work schedule. Ultimately, it could take days until the new version is finally deployed. With our pipeline we can have new features available to the user in **3 to 5 minutes** (depending on the service). This is a great improvement. It allows us to release more often and more flexible with the product, being able to deploy fixes faster. Furthermore, with Jenkins we can also introduce the automation of some Quality Assurance processes.

5

Conclusion

Contents

5.1 Limitations 58
5.2 Future Work 58

We have said it multiple times, but the fact is, and this worked proved it to us, that microservices are indeed a complex approach to system design. It brings advantages and disadvantages, that we have already discussed on preliminary research, and for us the improvements overcame the downsides of it. However that's not always the case, and what worked for us, might not be acceptable for others.

Something not trivial is understanding when microservices should be adopted. While starting by building an application as microservices from the beginning might look like a good idea, there is an high chance that it will crumble over its premises. This could be because there is no clear awareness about the scale of the system, or what kind of requirements will surge. We will end up introducing unnecessary complexity in the system, drastically decreasing adaptability/flexibility from the beginning. That being said, our opinion is that the adoption of microservices should happen at a stage when the system has an high maturity level where requirements are stable or can be properly planed for.

The adoption of microservices should be motivated by clear requirements. In our case it was the productization and continuous growth of the application that set the conditions for us to think about it. While this is very situational and context related, requirements that can be predicted to change the scale or reach of the system, increasing the user base and handled load or extending the system context should be seen as viable arguments for the adoption of microservices.

Since it should happen when a system is already mature, it means that the adoption of microservices will take the form of a migration. There will already exist a 'monolithic' application that needs to be migrated to a microservices-based architecture. This migration step should not be fast or simple and in some cases, organizations take years to complete it. It is, however, possible to make a future adoption of microservices less complex. Defining good practices, and respect them during development and designing modular software are two key points that will contribute to the simplification of the migration process. In our case, this was very important, the fact that the application at hand was implemented according the organization's good practices and implemented following a good modular scheme made it easier to migrate.

To assure that the migration occurs smoothly it should be possible to have feedback about the system changes so that it is possible to make adjustments and improve the migration process on-the-go. For this, we believe that the Strangler Application pattern helped us, not only to get acquainted with the processes involved but also to understand how to tune the system and each microservice, allowing us to comprehend the impacts of such changes on the system.

The migration to a microservices-based architecture is a challenging theme, not only on the system design context but also in implementation concerns. There are different problems that one has to face during the process. In this work we have described the ones that we have faced during our work with focus on the implementation phase. There are more problems that we did not address deeply, like the definition of microservices, because they are very situational. On the other hand there are also problems

that we did not have to face during our process that others might have experienced. This is the reason why works on microservices tend to lose some of its value to readers, because they are very attached to the private context where it was applied. In any case we believe that the problems that we researched and addressed in our work are valuable to a lot of microservices adopters.

Overall, we have had a positive experience with the adoption of microservices. We were able to deliver a product that looks like the one that was in place but that has a lot more capabilities. In terms of scaling we are now able to adjust its performance to the requirements more properly. We have increased the speed to which development happens and features are made available to the user. All this while delivering software with good quality that was simplified and achieving a good architectural level in what respects coupling and cohesion. Improving processes related to extensibility, maintainability and interoperability, properties hugely related with microservices.

5.1 Limitations

There is one aspect of our work that might be seen as a limitation on our perspective of microservices, that is, the scale. We have talked about organizations with ecosystems composed by hundreds or even thousands of microservices and in this work we were handling 5 services. Having a higher number of agents talking to each other does increase complexity, thus proposing even more challenges. In any case we believe that we were able to set a good foundation on microservices, and some of its problems. And even if it was possible to grow our microservices ecosystem in our context it would not add much more than what we discussed here since the process repeats itself for every new service.

5.2 Future Work

In what respects the implementation in this work, through our evaluation we have identified that the core service as a possible candidate for further refactor or splitting operations. This service should be re-evaluated in order to understand exactly if there is any further change that can be made.

Regarding the adoption of microservices, there are steps of the migration process that can be further explored in future work. One of the main problems is how to define microservices within the monolith, understand how to better define seams. A second research topic is on the organizational side of the migration. To explore how teams and people of an organization interact before and after the adoption of microservices, because that is something that should also change during the process.

Bibliography

[1] “Microservices envy,” *Tecnhology Radar — Thoughtworks*, [Online; accessed on November 16, 2017].

[2] M. Fowler, “Microservices resource guide,” [Online; accessed on November 6, 2017]. [Online]. Available: https://martinfowler.com/microservices/#what

[3] E. Evans, *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[4] M. Fowler, “Microservice premium,” *martinfowler.com — ThoughtWorks*, 2015, [Online; accessed on November 16, 2017]. [Online]. Available: https://martinfowler.com/bliki/MicroservicePremium.html

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: yesterday, today, and tomorrow,” pp. 195–216, 2017.

[6] L. Hatton, “Reexamining the fault density component size connection,” *IEEE software*, vol. 14, no. 2, pp. 89–97, 1997.

[7] “Episode 40: Interview werner vogels,” *Se-radio.net*, 2006. [Online]. Available: http://www.se-radio.net/2006/12/episode-40-interview-werner-vogels/

[8] “A conversation with werner vogels,” *ACM Queue*, vol. 4, 2006. [Online]. Available: http://queue.acm.org/detail.cfm?id=1142065

[9] R. Meshenberg, “Microservices at netflix scale: Principles, tradeoffs & lessons learned,” *GOTO 2016 Amsterdam*, 2016. [Online]. Available: https://www.youtube.com/watch?v=57UK46qfBLY

[10] “Netflix techblog,” *Medium*. [Online]. Available: https://medium.com/netflix-techblog

[11] E. Haddad, “Service-oriented architecture: Scaling the uber engineering codebase as we grow,” *Uber Engineering Blog*, 2015, [Online; accessed on December 12, 2017]. [Online]. Available: https://eng.uber.com/soa/

[12] E. Reinhold, "The opportunities microservices provide at uber engineering," *Uber Engineering Blog*, 2015, [Online, last accessed, December 12, 2017]. [Online]. Available: https://eng.uber.com/building-tincup/

[13] "Who is using microservices," *Microservices.io*. [Online]. Available: http://microservices.io/articles/whoisusingmicroservices.html

[14] M. Fowler, "Published interfaces," *martinfowler.com — ThoughtWorks.*, 2012, [Online; accessed on November 12, 2017]. [Online]. Available: https://martinfowler.com/bliki/PublishedInterface.html

[15] S. Clemens, G. Dominik, and M. Stephan, "Component software: Beyond object-oriented programming," *Addison-Wesley*, 1998.

[16] Wikipedia, "Component-based software engineering — wikipedia, the free encyclopedia," 2017, [Online; accessed October 10, 2017]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Component-based_software_engineering&oldid=801426450

[17] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet computing*, vol. 9, no. 1, pp. 75–81, 2005.

[18] "Soa manifesto," *http://www.soa-manifesto.org/*, [Online; accessed October 15, 2017].

[19] M. Fowler and J. Lewis, "Microservices," *ThoughtWorks*, 2014, [Online; accessed on November 12, 2017]. [Online]. Available: http://martinfowler.com/articles/microservices

[20] R. Despodovski, "Microservices vs. soa – is there any difference at all? - dzone integration," *dzone.com*, 2017, [Online; accessed November 20, 2017]. [Online]. Available: https://dzone.com/articles/microservices-vs-soa-is-there-any-difference-at-al

[21] T. O'reilly, "What is web 2.0," 2005.

[22] G. Viswanathan, P. D. Mathur, and P. Yammiyavar, "From web 1.0 to web 2.0 and beyond: Reviewing usability heuristic criteria taking music sites as case studies," *Hämtad februari*, vol. 27, p. 2015, 2009.

[23] G. E. Krasner, S. T. Pope *et al.*, "A description of the model-view-controller user interface paradigm in the smalltalk-80 system," *Journal of object oriented programming*, vol. 1, no. 3, pp. 26–49, 1988.

[24] N. Slater, "Using containers to build a microservices architecture," *Medium — AWS Startup Collection*, 2015, [Online; accessed on November 11, 2017]. [Online]. Available: https://medium.com/aws-activate-startup-blog/using-containers-to-build-a-microservices-architecture-6e1b8bacb7d1

[25] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," pp. 1–10, 2008.

[26] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, 2010.

[27] T. Dillon, C. Wu, and E. Chang, "Cloud computing: issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*. Ieee, 2010, pp. 27–33.

[28] A. Shawish and M. Salama, "Cloud computing: paradigms and technologies," pp. 39–67, 2014.

[29] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[30] "What is agile software development?" *Agile Alliance*, [Online; accessed on December 16, 2017]. [Online]. Available: https://www.agilealliance.org/agile101/

[31] "Manifesto for agile software development," *Agile Manifesto*, [Online; accessed on December 16, 2017]. [Online]. Available: http://agilemanifesto.org/

[32] J. Humble and J. Molesky, "Why enterprises must adopt devops to enable continuous delivery," *Cutter IT Journal*, vol. 24, no. 8, p. 6, 2011.

[33] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[34] A. Cockcroft, "Migrating to cloud native with microservices," *GOTO 2014 Berlin*, 2014, [Online; accessed on December 20, 2017]. [Online]. Available: https://youtu.be/DvLvHnHNT2w

[35] J. Choi, "The science behind why jeff bezos's two-pizza team rule works," 2014.

[36] M. E. Conway, "How do committees invent," *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[37] M. Fowler and J. Humble, "Continuous delivery," *martinfowler.com — ThoughtWorks*, 2014, [Online; accessed on November 17, 2017]. [Online]. Available: https://martinfowler.com/bliki/ContinuousDelivery.html

[38] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, "Migrating enterprise legacy source code to microservices: On multi-tenancy, statefulness and data consistency," *IEEE Software*, 2017.

[39] C. Posta, "The hardest part about microservices: Your data," 2016. [Online]. Available: http://blog.christianposta.com/microservices/the-hardest-part-about-microservices-data/

[40] S. Fowler, *Production-Ready Microservices. Building Standardized Systems Across an Engineering Organization*. O'Reilly Media, 2016.

[41] S. Newman, *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.

[42] A. Levcovitz, R. Terra, and M. T. Valente, "Towards a technique for extracting microservices from monolithic enterprise systems," 2016. [Online]. Available: http://arxiv.org/abs/1605.03175

[43] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," pp. 524–531, 2017.

[44] J.-P. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," 2017.

[45] N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "Microservices: Migration of a mission critical system," *arXiv preprint arXiv:1704.04173*, 2017.

[46] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Migrating to cloud-native architectures using microservices: an experience report," 2015.

[47] A. Balalaie, P. Jamshidi, and A. Heydarnoori, "Microservices migration patterns," 2015.

[48] M. Fowler, "Stranglerapplication," *martinfowler.com — Thoughtworks*, 2014, [Online; accessed on December 5, 2017]. [Online]. Available: https://www.martinfowler.com/bliki/StranglerApplication.html

[49] M. Rook, "The strangler pattern in practice - michiel rook's blog," 2016. [Online]. Available: https://www.michielrook.nl/2016/11/strangler-pattern-practice/

[50] C. Richardson, "Microservices: Decomposing applications for deployability and scalability," 2017. [Online]. Available: https://www.infoq.com/articles/microservices-intro

[51] "Protocol buffers — google developers," 2018, [Online; accessed on June 15 2018]. [Online]. Available: https://developers.google.com/protocol-buffers/

[52] S. Peyrott, "Beating json performance with protobuf," 2017, [Online]; accessed on June 20 2018. [Online]. Available: https://auth0.com/blog/beating-json-performance-with-protobuf/

[53] M. Fowler, "Event sourcing," 2005, [Online]; accessed on July 12 2018. [Online]. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[54] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *International Conference on Service-Oriented Computing*. Springer, 2008, pp. 41–52.