Adopting Microservices
Migrating a HR tool from a monolithic architecture

Tiago Santos
tiago.costa.santos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

November, 2018

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;

1. Introduction

Microservices are a recent architectural style to design systems or applications that have roots from the classical service-oriented architecture. At its core it aims to develop an application as a suit of independent modular units and that communicate via its exposed interfaces. The main difference from SOA are the communication mechanisms. When using microservices, the standard is to communicate via HTTP, each service exposes one or more REST API's that can be pooled by any other service.

There are a lot of concepts highly related with the microservices. Concepts that, in order to take full advantage of the style should be applied in every implementation. Some of the most important ones come from the Domain-Driven Design[1] study area. With this architectural style, each service only operates within a bounded context, implementing all the required logic to handle a limited set of requirements that regard the same business function. As a consequence, managing each service, its data and all its logic becomes significantly easier when compared with managing a piece of software developed using traditional approaches, a monolith.

One of the great advantages of Microservices is the high level of independence of each service. Each microservice should be completely independent from others, having no explicit dependencies while also able to govern its context. This will ultimately increase the flexibility and scalability of any application since each service can grow as required. Furthermore, it is also works as a great enabler for agile methodologies. Each microservice can be developed and supported by an independent team that can make the decisions that best fit their goals so they can focus on delivering the best software at an higher speed than before. On the other hand, it is also safe to say that microservices are not only a software architectural style but also deeply related with the processes and structure of an organization.

To fully understand the Microservices style it is also required to comprehend its drawbacks and the complexity that can be introduced by adopting them. One cannot adopt microservices simply hoping it will solve scalability problems or make
processes more agile. The adoption/migration process is something that needs to be planned in detail. The level of distribution brings complexity to data consistency management, performance, fault tolerance and resilience. Furthermore, a microservices ecosystem becomes highly dynamic introducing challenges like service discovery, load balancing and routing[2].

Ultimately, not all systems or applications are suitable for the adoption of microservices. In some cases, the extra effort required to implement microservices might not be worth when weighted with the possible gains. This will only result in decreased productivity[3].

With this work we intend to explore this style and its challenges. By performing a migration process ourselves we intend to expose problems that should be common to every migration process and argue about its solutions. At the same time we also want to evaluate the resulting product in order to understand the impact of the migration on the targeted product.

2. Background

Microservices are a product of the evolution of different aspects. Not only advancements that regard technologies but also on how people perceive and think about software. There are 5 aspects that we identify as great enablers for the surge of the microservices architectural style.

First, the organic evolution of architectures. Building software as a set of modules as been a long desire of any architect. Besides being easier to reason about the system itself, it also makes the development, support and managing processes easier to everyone involved. This is first observed in Component-Based Software[4], the 'modules' here where components that would encapsulate a set of related functions or data. It is in Service-Oriented Computing that we first see the notion of services. This was the style that set the root values for the well-known Service Oriented Architecture (SOA). With Services it was possible to achieve a 'separation of concerns', as each service would encapsulate certain operations while at the same time it was also possible to offer complex functionality by composing different services. Services would also be easier to scale by replication and easier to integrate using standard communication protocols[2]. As mentioned microservices derive from SOA, more focused on modularization and exposing functionality using lightweight protocols than on integration and composition.

The second important aspect is the evolution of the web. This new web is more focused on interactions between users and services and also much more dynamic[5]. This was enabled by new and more flexible protocols like HTTP/1.1, and REST, new frameworks that allowed implementation of dynamic pages using the MVC pattern like Angular and React and lightweight data formats like JSON. These new technologies are the root components of any Microservices implementation.

Advancements in Virtualization also played a very important role in the path to microservices. New virtualization techniques and technologies were more efficient and easier to use. With operating-system-level virtualization, the technique used in container technologies[6], deploying a small unit of software to a virtual machine was something to consider. Something that was not possible before due to all the overheads and complexity of traditional tools.

These improvements regarding Virtualization, also enabled the evolution of Cloud Computing. With this recent evolution, services Cloud-Service Providers like AWS, Azure and Google Cloud were able to offer an 'infinite' amount of resources to users for a very cheap price. It was possible to give a container image of our unit of software to a cloud provider and he would run it for a small percentage of the price it would take to run on our own physical resources. Ultimately, it allowed software architects to think about smarter ways to deploy, scale and manage their applications - Microservices.

Finally, the last aspect to which we attribute responsibility for the surge of microservices is the appearance of new Agile methodologies, in particular, DevOps. As companies become more flexible and agile it is required to improve the communication structure. Furthermore, responsibilities and ownership should be clear for every piece of software that is produced. DevOps aims to increase collaboration by looking at the development and operation responsibilities as one, "you build it, you run it"[7]. This way, one team composed by different roles can fully develop and support a piece of software independently, improving the efficiency of all the processes involved. As mentioned before, Microservices is also an organizational change. This organization structure is a perfect fit for Microservices, since each service is independent and implements a vertical layer of the business requirements, teams should be structured in a way that enables that.

All factors considered, with microservices, we should have a small multi-functional team supporting a set of services[8]. The level of independence of services allows for the development process to also be independent. There are testing, deployment or integration concerns that result in constraints. Ultimately, it results in higher speed of delivery.

The main advantages of the style are the de-
centralization of the governance and data management processes due to the independence level. It also allows the automation of processes like testing and deploying using pipelines that enable fast feedback about the state of the service. Since it is not possible to control the environment where the service runs, it should be well tested and able to handle different types of failures gracefully becoming more resilient than average. Furthermore, each service can be upgraded and changed freely, decreasing cost and resistance of change. As an independent component, each service's code can be stored in its own repository what makes the whole code base of the application easier to manage.

On the other hand, as mentioned before, microservices also have drawbacks. Among them, data consistency management is due to the level of distribution. Since a service needs to be more resilient, testing also becomes more complex. Typically, to fulfill one request, a microservice needs to perform calls to other services as well, this can be a problem to availability. If a service is not resilient, it might fail more often resulting in bad user experience.

3. Related Work
We broke down the problem of adopting microservices in two different challenges and started our research from there. Since we are starting from the point where we have one application that we want to migrate for the microservices style, first, we need to identify the microservices candidates in the application. Only then we can actually migrate it to the new architectural style.

3.1. Identifying Microservices
As mentioned, a microservice operated within a bounded context, a concept from Domain-Driven Design[1]. In order to create new services, we need to identify the seams in the overall context of the application. These seams are the boundaries that separate each bounded context, and guide us on defining the new services.

The first option to tackle this problem described in Sam Newman’s book[9]. This is the more organic approach, it is based on exploration of the system with focus on the data layer. Looking for data dependencies that can be broken and shared data that be served differently. With these database refactoring operations we have a starting point for refactoring the other layers, grouping business logic and the user interface’s elements that are related with the data of each module. Note that with this, we are understanding how data will be distributed and we can also start working on the necessary mechanisms to assure consistency.

There are also more medically approaches, like the one presented in the work of Levcovitz et al.[10]. The authors defined a process to identify microservices based on the decomposition of a system. Defining subsystems and mapping the dependency relationships of every layer of the application into a graph. Microservices would then be identified by splitting the graph.

Finally, we analyzed a work that defined a tool to automate this process[11]. This tool would provide suggestions for microservices by analyzing the source code of a project. With the information generated a graph would be generated, that would finally be processed by a clustering algorithm. The graph could be generated using different policies relating the relationships between code pieces and what they relate to, but it was also possible to build it using information about the authors of changes.

While not mentioning it, the last two approaches are also defined over the core concepts of Domain-Driven Design and while they seam like a good options, they have different types of limitations. Thus, we concluded that, to our work, it would be best if we applied these core concepts ourselves by exploring the system.

3.2. Experience Reports on Migration
While exploring the problem of migrating to microservices, we found multiple experience reports that documented the migration process. However, this is a problem hugely attached with context, there is no ‘one solution to all’. As such, while we were able to gather some useful insight, most of these reports did not dive deep into the decision process of every problem faced like we were expecting.

In the Gouigoux et al.[12], the authors are more focused on the results of the migration, documenting improvements on performance of the new system as well as reduction of costs regarding development.

In different a context, the works of Dragoni et al.[13] and Balalaie et al.[14], while registering the gains of the migration also provide the some more insight on the problems of the migration and their possible solutions. Most specifically, problems like service discovery, load balancing and orchestration and tools that can help like Docker Swarm or Kubernetes and process automation using pipelines built with GoCD or Jenkins.

3.3. Migration Patterns
As a result of our research we have also found some works that defined patterns to be applied during a migration process that were also valuable.

The work of Balalaie et al.[15], defines different patterns that can be applied in different situations during a migration. Exposing the problem, the common solution and most of the times, the
tool that can help us implementing, providing a sort of guide to developers.

Finally, arguing that a migration should not be a fast change but instead done at a moderated pace the strangler application provides the method to do so[16]. This provides a framework to make a service-by-service migration, removing functionality from the monolith and porting it to new services gradually, allowing fast feedback, and adaptability on the go.

With this research we were able to identify some key technologies and concepts to help us with our migration. With this work we provide an updated view on this subject, while going into more detail on the problems faced during the migration, something that most reports did not do.

4. Migrating to Microservices
With the insight from the related work we can now start working on our migration. This is a complex process and our focus will be more on the problems faced during its course that on the actual process of development.

4.1. Migration Fit
So that any application or system be considered as a candidate for a migration to microservices it should present a certain set of requirements. These requirements can vary given the context, but they should be relatable with the growing size of an application, user-base or load that it should support.

In our case, we found a proprietary application that while small in size at the moment it presented requirements that were expected to increase the usage load. More specifically, this was an internal HR tool that was in the process of productization. This would mean more users, more features and different usage patterns. Factors that in our opinion would make it a good fit for migration.

4.2. Splitting the Monolith
This first problem was to define the new microservices from the monolith code base. At the moment, there were already in place efforts to achieve a good modularization regarding the application, something that made our work easier. The databases were already separated into different schemas, respecting different contexts, so we were only required to define the boundaries in the upper layers. After this work, we were able to define four different microservices that handled their own bounded context’s data and operations.

Furthermore, after reaching the UI layer of the application we were faced with a particular implementation. This application had its UI defined as a separated project, an Angular application that was serve to the user’s browser. Given that it did not make sense to work on a new UI, we reused 100% of this code and defined a new microservice to serve these resources. While the designation of service is not very appropriated, this microservice had particular and interesting aspects. The fact was that it reassembled an API gateway, in the end it aggregated all service’s operations and served it to the users alongside the UI.

In the end of this process, we defined five microservices:
- **Core** - Handling Security and Collaborators
- **Workgroups** - Handling Teams Data/Operations
- **Calendar** - Handling Vacation Data/Operations
- **Notificator** - Supporting service to send out notifications about events in the system.
- **UI** - Provides UI assets and fetches data from all other services

Finally, in order to make the source code easier to manage we decided to move the code of each service to a new repository. This is the most common applied pattern on this matter, besides becoming easier to manage, it also becomes easier to navigate, explore and faster to compile. Additionally, from the refactoring operations we were able to define a common library that implemented useful modules to be used by every service. This was a simple way to keep every service always updated with respect to the common library. In the end, instead of on big and confusing repository we now had 6 smaller repositories.

4.3. Serialization
In order to make our data available to other services, we needed to provide it in a smart and efficient way. There are two protocols that have the spotlight at the moment, JavaScript Object Notation (JSON) and Protocol Buffers (ProtoBufs).

We decided to choose JSON in our application for the following reasons. While ProtoBufs are discussed as more efficient, our application is not very data intensive, thus the gains would be lesser. Additionally, implementing ProtoBufs takes a significant amount of effort. Furthermore, JSON provides a better flexibility, which for use is a great advantage. We can update our data structures without being worried with contract breaking and serialization errors. With JSON, we can serve all the data that have and allow the consumer to consume whatever it needs.

4.4. Communication
Due to the distributed nature of the new microservices architecture the system becomes more verbose. The new forms of communication between services needs to be defined.

The first use case for the new flow of communication is when there are synchronous data re-
quests. These are requests that one service performs to another service to be able fulfill a given request. These requests are performed to a service's interface via HTTP and should be handled carefully. This kind of requests are what brings dependencies to our system, possibly putting our service's availability in risk. On the other hand, since we have developed all the services in interaction, we have more control over the environment which means that the availability problem is less worrisome. A possible solution for situations like these is to merge the interacting services, something that, for us, did not make sense.

The second communication use case is when there is the need for asynchronous event propagation. For example, if an entity gets deleted in one of the services, it is required to delete all this entity's data in all other services. We solve this problem by applying the event sourcing pattern[17]. This allows us to propagate updates by having the service where the change happens emitting events with the data about the state change. This way a consuming service will analyze the data on the event and decide if there is any update needed on his data.

We implement this behaviour using the Advanced Message Queuing Protocol (AMQP), more specifically, an open-source implementation RabbitMQ. RabbitMQ allows us to send message to a broker that will forward them to the right recipients. We have used an RabbitMQ's topic deliver pattern, an implementation of the classical publish-subscribe pattern. It enables 1-to-many message delivery based on routing keys. In RabbitMQ, recipients subscribe to topics. These topics are dot-separated strings which are flexible supporting the use of wildcards. But when a message is delivered to a broker it has a routing key, a concrete topic. The broker will then check if any subscriber has subscribed to a topic that matches the routing key of the delivered message and forward it in thoses cases.

4.5. Security
One important factor in any system is security. In our microservices system, there is one service that is responsible for all the data and operations regarding the security requirements, the Core service. How could we secure the other services?

One option was to route every request through the Core service and after checking authentication, forward it to the right recipient service. One one hand, we are protecting other services from unwanted load, limiting its exposure. However, we are also creating a dangerous bottleneck in our system. In the case of a service failure, the whole system would be impossible to use and that is not smart.

The solution to this problem was simplified since this system was already used as the main authentication tool for all the organization's tools. There was in place an implementation of the OAuth2 protocol, which is one of the main mechanisms for authentication and authorization in distributed systems. This protocol is adaptable to a lot of different scenarios it worked perfectly in this by implementing the Implicit Grant Flow. This flow assures that any user that is authenticated in the system is attributed a unique code, a Bearer Code that proves its identity. This code is then passed when requests are performed to any service and each service only needs to validate this token to prove the users identity. While this still makes almost every request to go to the Core service, this way of doing it, by pooling is more efficient. This is a case of HTTP synchronous requests that we have discussed previously.

4.6. Service Orchestration
Service Orchestration is one concern known from the early stages of this work but from the research done we were able to find the right tooling for this. We knew, from the start the Docker was the virtualization tool to work with, and we also found Kubernetes a very powerful tool on managing Docker containers, so we had our answer.

There are a lot of concepts in Kubernetes that allow us to abstract from the all physical elements of the infrastructure. It allows us to define a service with fixed endpoints while the containers keep being created and destroyed as necessary. Furthermore, it is possible to associate a Volume to the containers behind a service allowing the persistence of data even when those keep being changed. The last useful feature to mention, is configuration management. Kubernetes also offers mechanisms to feed containers with configurations that support changes and updates to be pushed to the associated containers.

Our deployment style consists is based on Docker and Kubernetes. For each service there is a Docker container that hosts the service itself and a second container that hosts the database service. Additionally, there is a container for the RabbitMQ instance and another one for the UI server. Containers can link to each other by a given name that is resolved by the Kubernetes DNS module and expose one or more ports to the outside of the cluster so they can be accessed from the exterior.
case of need we could port our solution to another provider.

Initially, we were thinking about AWS as our cloud provider, however, after some research we decided to go with Google’s Cloud Platform. The change in our option was mainly due to the use of Kubernetes. Google is involved in the development of Kubernetes from the early stages, and they offer the Google Kubernetes Engine (GKE) that simply put, is a Kubernetes instance on Google’s resources. Thus, we could simply apply our deployment directly in a GKE instance. Furthermore, their pricing model for running Kubernetes in GKE is different that the one offered by the other providers. With GKE we only pay for the nodes that run our services making all Kubernetes’ services like cluster management and DNS run for free.

For the actual deployment we created a 2-node cluster that added up to 4vCPUs and 15 GB of RAM where we created all our services and started all our containers. Note that this cluster is scalable and can be adjusted as needed. For example in the case we need more instances of a certain container. This can even be done automatically using other Google Cloud’s features.

4.8. Continuous Delivery and Integration
Now that we have everything defined, we can automate all the tasks that need to be executed from the moment we apply changes to the code base to the moment that those changes are visible to the user.

We do this by using the continuous deliver and continuous integration pipeline. This pipeline is built using Jenkins, that, from research, we found to be the right solution for this problem. We have created a new VM instance in Google Cloud to serve as our Jenkins server and linked it to our repositories in Git. With this, everytime that a push is done to code in the repository, Jenkins starts its work. First, it fetches the needed code base, the service and the common library. Then compiles and builds the service executable. It then produces the Docker image and pushes it to the proper repository. Finally it updates the Kubernetes deployment in GKE by updating the container image and any configuration if necessary. This process that could take several hours because it needed human interaction, is now automated and can be executed in 5 minutes. Furthermore, it is possible to extend the pipeline as needed for example for quality assurance purposes.

5. Evaluation
To evaluate our work we have defined some key factors that should be measured to understand the impact of the migration. The first, are concerning the quality of software, we want evaluate Size, Complexity, Coupling and Cohesion. These factors are important because when we think about microservices and consider a migration it is because we are in need for improvements in these areas. Microservices are characterized by being highly decoupled and having high cohesion. Furthermore, we see microservices as more manageable code bases and that has a lot to do with the size and complexity of each service. In addiction, there are two other factor that we find interesting to document, infrastructure costs and speed of delivery. From research we found that costs seam to be a result of the migration, so we think that this is interesting to document. And the speed of delivering features to the user is a key to see how adaptable is the system.

5.1. Size
Size has direct impact on maintainability, the bigger the service the harder to maintain over time.

The first metric used to this factor is Lines of Code (LOC), which is usually used to describe any system. With the refactoring operations, we were able to build a common library that contained about 12% of the code of the original monolith, which means that we a achieved some code reusability.

\[
\text{LOC(Monolith)} = 48875 \\
\text{LOC(Microservices)} = 47966 \\
\text{LOC(Microservices - avg)} = 2754
\]

LOC(Monolith) = 48875
LOC(Microservices) = 47966
LOC(Microservices - avg) = 2754

However, when we compare the number of lines of code in both architectures, we do not see a significant decrease. This is because the monolith had already a good modularity level, and had been developed according multiple good practices. Thus the level of code was already high.

The second metric used to measure size is Weighted Service Interface Count (WSIC). WSIC is the number of exposed interfaces summed with the total number of operations exposed. The logic behind it is that the more operations and interfaces a service exposes the bigger it is.

<table>
<thead>
<tr>
<th>Core</th>
<th>Calendar</th>
<th>Workgroups</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

This metric should be used as way to compare services. By calculating the average of the values (12) and identify outliers. There is a clear outlier in our system, the Core service. This is because this service serves more functionality and is responsible for more data. This might also be a sign that this service might be in need of more refactoring operations, maybe even a splitting operation.

The size factor does not provide a lot of insight in our case since most all of the code was reused...
and all the previous operations and interfaces were kept only with little additions. Nevertheless, these metrics are important and as a system grows they provide a way to compare services.

5.2. Complexity
Complexity also has a hard impact on maintainability, a complex system is always harder to manage and maintain. To evaluate complexity, we have used the Total Response for Service (TRS) metric. It counts the number of operations executed to fulfill one operation. More specifically, the number of method calls that one method performs to return its value. An higher value means that operations need to call a lot of methods being then more complex.

| Table 2: Total Response for Service per Service |
|-----------------|--------|--------|--------|
| Core            | 117    | 37     | 39     |

There is a clear disparity the measurements shown in Table 2. The Core service has a significantly higher value that the other services. Like we have discussed, this service exposes more operations so it is expected that the TRS value is higher. Furthermore, some of these operations implement security concerns that tend to be more complex needing more auxiliary methods, a factor that also has impact on this value.

5.3. Coupling
Coupling measures the level of interdependencies and interconnections between services. In a microservices-based system, services should be loosely coupled.

The used first metric is Services Interdependence in the System. This metric looks at the system and identifies the dependencies that are in it. In our resulting system, we have some clear dependencies. First, since the UI service is just a consumer, it depends on all the other services. Second there is also a dependence between the Core service and the Workgroups service, since that for some operations, the first needs data from the second. Finally, because of the fact that the Core service implements the security concerns every service depends on it. With this, we can identify one circular dependency. The Core service depends on the Workgroups service, which again depends on the Core service. This is a dangerous cycle that might put the availability of the whole system in a fragile situation.

The next set of metrics used for Coupling are related. The Absolute Importance of the Service (AIS), expresses the number of clients that invoke at least one operation of a service. Whereas, the Absolute Dependence of the Service metric (ADS), looks at dependencies from the opposite direction, it describes the number of services that a certain service depends on. Finally, Absolute Criticality of the Service (ACS) can provide us an overview on the coupling of the system using the other two metrics.

<table>
<thead>
<tr>
<th>Table 3: Coupling Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
</tr>
<tr>
<td>AIS</td>
</tr>
<tr>
<td>ADS</td>
</tr>
<tr>
<td>ACS</td>
</tr>
</tbody>
</table>

From the values in Table 3 we can conclude 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 the UI service is not important. We have discussed that the UI service is not a typical service, so we should not be too strict when looking at these metrics.

With microservices we are aiming for low coupling values, however we should not be strict when looking at the values. To a system with more microservices, we recommend to use average values to compare the developed services. Again, identifying outliers is the key to maintain the system.

5.4. Cohesion
Cohesion measures the to which level a service operates only within its bounded context. For measuring cohesion we have used three relatable metrics. Service Interface Data Cohesion (SIDC) expresses cohesion by analyzing the data parameters of the exposed operations. This is a metric more appropriated to when contract protocols like SOAP are used. In our case we use HTTP interfaces that consume JSON, so in a way we achieve the highest value possible. Service Interface Usage Cohesion (SIUC) 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. This is measured by a ratio between the number of operations used by all clients and the number of exposed operations. Finally, the Total Service Interface Cohesion (TSIC) provices and overall look over cohesion, using a normalized sum of the two previous metrics.

<table>
<thead>
<tr>
<th>Table 4: Coupling Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIDC</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>SIUC</td>
</tr>
<tr>
<td>TSIC</td>
</tr>
</tbody>
</table>

From the values in Table 4, we can see that our
services have values close to 1 which means that they achieve a good level of cohesion. A note to the Core service that presents a slightly lower value that the other services, which means that its cohesion might possible be improved in the future by applying some refactoring operations.

5.5. Performance
Performance is important in any system. With the migration, we needed to assure that the application could at least provide the same levels of performance than before.

For this, we have defined a set of workloads to be executed by both architectures. These workloads consisted in a significant amount of requests to one of the most expensive write operations in the system. To execute this test we have used the previously described deployment for our microservices architecture, and we have also created a VM instance on Google Cloud's with the same specifications, same RAM and CPUs where we deployed the monolith.

![Figure 1: Workload Execution Chart](image)

Seen in Figure 1, the graphic showing the execution time for each of the workloads, with increasing number of requests, 1000, 2500 and 5000. We can see that as the number of requests grows, the microservices architecture outperforms the monolith.

Not only we were able to provide the same level of performance, we have also improved it. Furthermore, with this architecture, by scaling there is always possible to provide more resources achieving even better performance when if requested.

5.6. Infrastructure Cost
For reasons mentioned before infrastructure costs are important to document. In a microservices ecosystem, having services being deployed in a cloud provider could have a big impact on the overall cost of an application. Even though resources are now a cheap commodity, costs can add up to an expensive bill if not carefully managed.

Our initial application was deployed as a single instance in the organization's own resources for internal use only so there were virtually no costs. With our new deployment style, we would be looking at a monthly base cost of 140 EUR for the compute resources. Additionally, there would then be costs for disk storage (0.04 Eur/Gb) and network traffic received by the Load Balancers (2 Eur/Gb) but it would be much less significant.

5.7. Speed of Delivery
Finally, the last factor to be measured is speed of delivery. With microservices, the ability to changes should be improved. Something that is known is that the time between the moment when changes are made to the code and when they are deployment to production is time when improvements made are not yet available to the user. This might have impact, for example, in profits, a bug that was making users stop using the product could be fixed faster, a feature to capture new users could be deployed faster.

On this aspect we were able to achieve great improvements. Before, this process could take hours due to the need for human interaction. Now, with our CI/CD pipeline, we could see it in production, available to the user, in a short period of 5 minutes.

6. Conclusions
With this work we were able to confirm that the migration process is complex. Adopting microservices should not be something to taken lightly, it should be planed for, and all the advantages and drawbacks need to be weighed to assure that the migration makes sense.

The migration should always be motivated by requirements that are predicted to change the scale of the system. At the same time, the process should only happen when a system reaches a mature state so that it can happen in a controlled way. Furthermore, defining good practices to be respected during the process and having a good good definition of the boundaries of the contexts will simplify the process of migration.

We have also confirmed that the Strangler Application is a very useful patters for the adoption of microservices, it allows the continuous development on new functionality while supporting the extracting the features of the monolith.

We have addressed a number of interesting problems that we faced during our migration that we believe are interesting to document. There are of course other problems that are more related with our context that would not add much value to external entities. We hope that this will be useful for other microservices adopters.

Overall, we have a very positive experience with microservices, we were able to deliver a product that feature wise, is the same but that has more capabilities relating scalability, performance and maintainability. Furthermore, from an architectural point of view, we were able to simplify the sys-
tem while achieving good quality regarding coupling and cohesion.

There is a case to be made about the scale of our system, and its relation with the experience of migrating to microservices. We have seen examples of organizations working with thousands of microservices which adds more complexity and proposes even more challenges. On the other hand, we believe that we have worked on the base problems of the migration, and that from there the problems faced are just a composition of some of the ones discussed here.

7. Future Work
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.

References


