Stepwise Migration of a Monolith to a Microservices Architecture: Performance and Migration Effort Evaluation

Diogo Faustino*
*Supervised by António Rito Silva

***Abstract*—The agility inherent to today's business promotes the definition of software architectures where the business entities are decoupled into modules or services. However, there are advantages to having a rich domain model, where domain entities are tightly connected because it fosters reuse. On the other hand, the split of the business logic into modules or services, encapsulated through well-defined interfaces, and the introduction of inter-service communication foster an agile development but introduce a cost in terms of performance.**

In this paper, we analyze the impact of migrating a rich domain monolith into a modular architecture and sequentially into microservice architecture, both in terms of the development cost associated with the refactoring, and the performance cost associated with the execution. The current state of the art analyses the migration of monolith systems into a microservices architecture, but we observed that migration effort and performance issues are already relevant in the migration to a modular monolith and concluded the impact of establishing a microservice architecture with a rich domain model and inter-service communication on the performance. Additionally, we also addressed challenges exclusive to the microservice architecture such as eventual consistency of the databases and the deployment of the services.

***Index Terms*—Domain-driven design, Modular architecture, Refactoring Cost, Performance evaluation, Microservices, Eventual consistency.**

I. Introduction

Domain-driven design [1] advocates the division of a large domain model into several independent bounded contexts to split a large development team into several small, and business focused, teams to foster an agile software development. Additionally, these bounded contexts can be implemented as modules, each one of them with a well-defined interface, to further decouple the teams by reducing the number of required interactions between them. This modularization is also suggested as an intermediate step of the migration of a monolith to a microservices architecture [2].

The correct identification of what should be the responsibilities associated with each module is not trivial, and has to be done through several refactoring steps [1], [3], [4]. These refactoring operations can be more easily performed in a strongly connected domain model where the business logic is scattered among the domain entities, a rich domain model, and in the absence of interfaces between the domain entities [5]. So, it is common that projects start with a single domain model, because in the first development phases the domain model is not completely understood. Premature modularization adds a cost to development, because of the need to refactor modules' interfaces, since it initial design does not capture the correct abstractions.

The use of interfaces between modules requires the transformation of data between them to encapsulate their domain models, anticorruption layers [1], which implies significant changes do the domain model, and have impact on the performance of the system. Therefore, the process of modularizing a monolith system, or further migrating it to a microservices architecture [6], has to address these problems.

The migration into a microservice architecture further encapsulates the bounded contexts into independent processes that become cohesive services, capable of being individually deployed and isolated through an API that provides the anti-corruption layers. Therefore, the modules can provide the groundwork for achieving the services without costly ramifications to implement them and allow to focus the refactoring efforts in more intricate aspects introduced with the architecture, such as inter-process communication and decentralized data management.

Research has been done on the comparison of the performance quality between a monolith system and its correspondent implementation using a microservices architecture, but these results are sometimes contradictory, e.g. [7], [8], addresses different characteristics of a microservices system, e.g. [9], [10], or are evaluated using simple systems, e.g. [11], where the monolith functionalities do not need to be redesigned in order to be migrated to the microservices architecture.

In this thesis, we describe the refactoring process of a large monolith system into a microservice architecture, removing circular dependencies between modules, and adapting each of these modules into independent services with well defined REST APIs, focusing on the refactoring effort and the performance. Concerning the refactoring, we describe the types of refactoring that were applied and measure their impact on the migration effort. Regarding performance, we describe the architectural tactics that were applied to improve the performance of the microservice architecture, and compare it with the modular monolith performance.

From the point of view of the microservice architecture, we observed the benefits of establishing the modular architecture in terms of refactoring necessary in the migration process and present the refactoring made to the interfaces and modules to obtain the services. Similar to the modular monolith, the results presented a significant impact on the latency due to the remote invocations and network overheads. Additionally, it could also be observed how the data consistency was affected by a decentralized approach.

In the next section we describe the architectural elements of a microservice architecture from a modular architecture. Then, in section III, we present the refactoring activities that adapt the modules into services, the refactoring necessary to the interfaces for the inter-service communication, the behavioral changes for a decentralized data approach and discuss some performance tactics that should be considered during the process. A large monolith system is introduced in section IV, in which we describe how it migrated into a microservice by applying the migration refactorings. In section V the migration process is evaluated in terms of refactoring cost, performance, and data consistency and the results are discussed in section VI, in which they are generalized to applications that have a similar architecture and how these results apply to the migration into a microservice architecture. Section VII presents related work and section VIII the conclusions.

II. Microservice Decomposition

The decomposition of a rich and complex domain into loosely decoupled subdomains is a fundamental procedure into obtaining, respectively, the modules and services for the modular monolith and microservice architecture. Therefore, by implementing a modular monolith, the decomposition of the domain is achieved through the modules which then can be converted into services with little effort. Additionally, the two types of interactions, uses and notification, applied in the modular monolith, are an initial fit to the introduction of inter-service communication.

The microservice architecture defines two types of inter-service communication, synchronous communication, where a service requests the information through an API and awaits its response, and asynchronous communication, which can undergo between multiple variants depending on the requirements of the architecture, but mainly allows for services to share information without any dependencies. The synchronous request/response communication presents a similar behaviour to the uses interaction and allows to remotely invoke an API using protocols such as HTTP, to retrieve information. Therefore, the uses interaction can be replaced by this type of communication without redesigning the features of the service. On the other hand, the asynchronous communication allows to adapt the notification interaction by maintaining the communication between the services without creating circular dependencies. This is achieved through an event-driven asynchronous communication, in which a publisher service sends events to a message broker and then, the message broker publishes the event to subscribed services. In addition, the event-driven communication plays an important role on the consistency of the information, establishing eventual consistency between the databases of the services.

At the end of the migration, the microservice architecture is composed by front-end services that contain the user interface and act as a gateway to the back end services, and back-end services responsible for the features. Each service is implemented through a well defined API compatible with the necessary inter-service communication, persists the information in separate databases and are independently deployable.

III. Refactoring

Most of the refactoring effort to achieve a microservice architecture from the modular monolith is focused on the interfaces of the services, since both the uses and notification interaction have to be adapted into the appropriate type of communication. This requires a redesign of the provides/requires interfaces to implement synchronous communication, and the publisher/subscriber interfaces to introduce the message broker and implement the asynchronous communication.

To implement the synchronous communication, the provides interface of the services were implemented as a REST API, that offers public endpoints for fetching the information of the services, which then can be accessed by the requires interface through HTTP requests. The refactoring process consisted of mapping the publicly available methods from the interface to an unique URI and to an appropriate request type, GET, POST, PUT, and DELETE, according to the actions of the method on the information. On the other hand, the requires interface needs to match the changes of the provides interface and implement a remote access through the defined URL of the REST method with the appropriate mapping in terms of parameters and type of request. Therefore, the cost associated to the refactoring of the interfaces depends on the number of methods and quality of the interface, but in general these mappings are simple to implement.

An additional requirement related to the provides interface introduced with the microservice architecture is the compatibility of the *dtos* to the serialization required in a remote invocation. Generally, this is a trivial process since the *dtos* have simple structure and mostly carry domain information, however when carrying non-serializable information, this can introduce additional refactoring effort to adapt without affecting the behaviour of the application.

Focusing on the notification interaction, the refactoring effort focuses on the publisher and subscriber interfaces to introduce the message broker. This is due to how the notification interaction provides the groundwork into achieving the event-driven communication by implementing the events and removing the dependencies between the modules. Overall, the introduction of a message broker is not a complex task and mostly requires a publish and listen method for respectively the publisher and subscriber interfaces to communicate with the broker. An important refactoring is the introduction of eventual consistency through this communication, where a publisher service notifies subscribed services of relevant changes to maintain consistency between them. This can have different impacts on the consistency and refactoring cost depending on the changes to the information and the indirect associations between services.

Similarly to what occurred in the modular monolith, the definition of independent services and the introduction of synchronous communication raised several performance problems associated with the remote invocations, fine granularity of the functionalities, and the large amount of information transferred in *dtos*. The following optimizations were applied:

- Implementation of additional caches in the services for faster information retrieval speed and to reduce the number of remote invocation for faster performance. These caches are created upon service creation and the data is preserved until the services stop their execution, this is very effective since chosen data remains immutable throughout the execution.
- After evaluating the performance, a similar optimization from the modular monolith was applied to the *dtos*, where a few were adapted into containing an extensive amount information associated with several interrelated domain entities, in order to decrease the number of remote invocations thus increasing the performance of the application.

IV. Case Study

The LdoD monolith[1] is a digital archive for digital humanities that offers features like searching, browsing and viewing the original text fragments, different variations of the text fragments, as well as different editions of a book. It also provides a way for users to create their own (virtual) editions of the book, in which they can add, remove and order the fragments as they wish. Other features include a recommendation feature, that defines a proximity measure between different fragments according to a set of criteria, a game feature, which implements a serious game where users tag text fragments, a reading feature, which suggests reading sequences of the text fragments, and a visual feature that provides graphical visualizations of the relations between text fragments.

The monolithic application is implemented in Java using the Spring-Boot framework[2] and an Object-Relational Mapper (ORM) to manage the domain model's persistence. The domain model has 71 domain entities and 81 bidirectional relations between domain entities, which resulted in a strongly coupled rich domain model. This solution provides high reusability, because the business logic is split among the domain entities and can easily be reused, and also due to the bidirectional relations that facilitate the navigation in the domain model. In total, the monolith has 25.862 lines of Java code, 20.039 lines of JSP code and 7.721 lines of JavaScript code.

The modular monolith implements the same features as the monolith. These features are implemented as a set of modules that apply the uses and notification interactions to preserve the dependencies between features. The LdoD microservice architecture migrated the modules into seven different types of back-end services with well defined REST API, capable of being independently deployed and providing the same functionalities from the respective module, two client-side front-end services and a server-side front-end service responsible for the user interface, processing and routing the requests for the respective back-end services. These services apply the described synchronous HTTP request/response type of communication between the services to exchange information and maintain the application behavior from the uses interaction. Additionally, the services *Text*, *User*, *Game*, *Virtual*, *Recommendation* and *Front-End* implement a database per service pattern accordingly to the subdomain.

The notification interaction is implemented as an event-driven asynchronous communication using an ActiveMQ message broker responsible for managing the events between the services. Using the `Fragment` removal example, when a text fragment is deleted by a instance of *Text*, an event is published into the message broker, which then will be responsible for publishing this event to all subscribed services and remove the fragment from the virtual editions that refer to it. Due to the decentralized data management, eventual consistency is also described in the removal example, meaning that inconsistent information is a possibility and its impact needs to be evaluated.

Another important aspect of the microservice architecture is the deployment of the services for an efficient scalability of the resources into features that are regularly accessed from several services. The LdoD microservice architecture is capable of managing and deploying several instances of the services through the Kubernetes technology. Note that, the Kubernetes is responsible for the dynamic aspects from the services, like the service discovery, networking, resources management and load balancing.

V. Evaluation

The evaluation presents the impact of the migration from the modular monolith to the microservice architecture from the perspectives of refactoring and performance.

A. Refactoring to Microservices

From the perspective of refactoring into a microservice architecture, the changes were focused on two different major aspects of the modular architecture: the provides and requires interface and the corresponding *dtos* from each service. Table I presents the impact of refactoring the uses interaction into synchronous communication in terms of mapped methods in the interfaces of each service to incorporate a REST API and remote invocations. It can be observed significantly high percentages of modified methods where at least 83% of the methods had to be mapped. This may look like a significant migration effort, but these changes were simple changes to implement, consisting of repetitive mapping procedures.

[1]https://ldod.uc.pt/
[2]https://spring.io/

	Text		User		Virtual		Recomm	
	P	R	P	R	P	R	P	R
Modified/Total Methods	91/109 (83%)	-	43/47 (91%)	-	152/155 (98%)	6/6 (100%)	6/6 (100%)	7/7 (100%)
New Methods	16	-	2	-	2	18	1	0
Modified/Total Dtos	6/15 (40%)		2/3 (67%)		9/14 (64%)		3/11 (27%)	

	Game		Search		Visual		Front-End	
	P	R	P	R	P	R	P	R
Modified/Total Methods	14/14 (100%)	9/9 (100%)	2/2 (100%)	14/14 (100%)	-	9/9 (100%)	-	145/153 (95%)
New Methods	0	2	0	0	-	1	-	42
Modified/Total Dtos	2/6 (33%)		5/17 (29%)		1/4 (25%)		64/74 (86%)	

TABLE I: Impact of the refactoring in the provides interfaces (P) and requires interface (R) and dtos from each of the services in terms of refactored methods and dtos

However, a few incompatibilities were detected that directly affected the refactoring cost which should be taken into consideration. An incompatibility was the transference of non-serializable information between the modules, which could not occur in remote invocations. This behaviour had a more serious consequence, since it could affect areas outside the scope of the API. A way to address this situation is by implementing a custom serialization for that type of information, which would allow to write the object into a JSON format that could be sent between the services and be applicable to every usage throughout the interfaces. This also allows the reuse of the information in the *dtos* and maintains the refactoring in the scope of the API. On the other hand, if the serialization cannot be achieved, it is necessary to individually address their usages and adapt how the information is transferred, while maintaining the behaviour of the functionality. Therefore, note that the quality of the interfaces in the modular monolith has an effect on the refactoring effort and it is important to keep the information transferred between the modules simple for an efficient migration.

Additionally, it could also be observed in Table I a surprisingly high number of modified *dtos* from the back-end and front-end services to, respectively, serializable and deserializable formats, despite their simple structure. This was due to an optimization tactic applied to the back-end *dtos* in the modular monolith that implemented them as an entry point into obtaining additional information related to a domain entity, for an easier access. However, this proved to be unsuited for remote communication due to JSON serialization requirements not being met. By default, the JSON serialization requires the invocation of every *getter* field publicly available in order to be serialized, but this resulted in the creation of larger *dtos* that contained all the information they referred to, without it being required. Therefore, this was an inefficient behaviour for the inter-service communication, since it increased the network overheads and had to be refactored to avoid performance degradation in the functionalities.

Note that, this problem was more significant in the *Front-End* service, because it is responsible for most of the deserialization process of the *dtos* and had to match the refactoring applied to them in the serialization changes. In addition, the deserializable *dtos* remained responsible for providing the entry point to fetch the additional information and were refactored similarly to a requires interface to implement remote communication. Overall, despite the high percentages, the refactoring was simple to implement.

B. Performance

To evaluate the impact that the new architecture has on the performance of the system, we performed performance tests on the previously described features of the application, using the JMeter[3]. The testing was done with the application running inside Docker containers on a dedicated machine with an Intel I7 6 cores, 16GB of RAM and a 1TB of SSD.

Four functionalities were selected for the analysis. They were chosen due to their impact in terms of: the number of the domain entities they interact with, the number of modules/services accessed by the functionality, and the amount of processing required by the functionality.

The *Source Listing* functionality presents the listing of the archive sources, where a source corresponds to a physical source document. When using this functionality, the end user obtains all the information about each one of the 754 sources, such as date, dimensions, and type of ink used in the document. The implementation of this functionality is done through an interaction between the *Front-End* and *Text* modules/services.

The *Fragment Listing* functionality is implemented in the modular monolith and microservice as an interaction between the *Front-End* and *Text* modules and services respectively to present the list of all text fragments. There are 720 text fragments in the archive. For each fragment it is presented the information of its several interpretations, where each fragment can have 2 to 7 interpretations.

The *Interpretation View* functionality presents an interpretation of a text fragment. Its implementation in the modular monolith and microservice requires several interactions between the *Front-End*, *Text*, *User* and *Virtual* modules/services. To do the performance test it was chosen an interpretation of a virtual edition, to have a test that includes more different domain entity types.

The *Assisted Ordering* functionality orders the fragments according to a set criteria, such as date and text similarity (tf-idf). This ordering requires more than 250 000 fragment comparisons (considering for instance a virtual edition with

[3]https://jmeter.apache.org/

720 fragments, we get $720 * 719/2 = 258\ 840$ comparisons between fragments), and each comparison requires information about the fragment for up to 4 criteria. For its implementation the front-end accesses 4 back-end modules/services, *Text*, *User*, *Virtual* and *Recommendation*. Because this functionality repeatedly interacts through the same set of data, the information about the fragments is cached in the monolith implementation, to improve performance. The modular monolith implementation also uses this cache and the microservice architecture implemented two additional caches due to avoid performance deterioration caused by remote invocations of the same set of data.

From the performance results between the architectures, it could be observed that the microservice architecture had a severe negative impact on the performance, both in terms of latency and throughput, independently of the functionality and information in the database. The *Fragment Listing* and *Source Listing* functionalities experienced significant latency values for 720 fragments and huge performance differences between the architectures for 100 fragments. This is due to the number of remote invocations, which accumulate latency due to network overheads and affect the performance as the number of invocations and information increases. To supplement these results, the number of remote invocations between the services for each functionality was measured and it could be observed 4283/28540 and 854/5966 remote invocations for 100 and 720 fragments in the database, on the *Fragment Listing* and *Source Listing* functionalities. Therefore, this fine-grained behaviour combined with large amounts of information correlates to the performance degradation of the functionalities in the microservice architecture

The *Interpretation View* also underwent through similar degradation of the performance, however the variation was significantly lower compared to the previous functionalities. This is due to not only the difference in the amount of information, but also the required number of remote invocations being relatively low. The combination of both these factors helped in reducing the overhead times. On the other hand, the difference in performance between the architectures is still considerably high, meaning that even with reduced information and invocations, the performance of the microservice architecture is worse that the monolith. In a positive note, the microservice architecture did not impact the performance of the *Assisted Ordering* functionality. This is mainly due to two reasons. First, the use of additional caches in the microservice architecture that stored information of the domain, proved to be effective in optimizing the performance of the functionality. Because the functionality repeatedly interacts through the same set of data, these caches reduced the number of remote invocations necessary to perform the functionality. Second, the functionality is computationally demanding, which reduces the impact of the remaining distributed communication on the overall performance.

However, it is also important to note that, before the optimization, the *Assisted Ordering* functionality presented an unusable experience due to millions of invocations performed. This demonstrates a serious impact of migrating a computationally demanding functionality with fine-grained interactions to a microservice architecture. Through the use of caches, we were able to match the performance of the modular monolith and reduce the number of remote invocations to 524/4388 for respectively 100 and 720 fragments in the database.

C. Microservice Optimization

In the previous section, it was observed a correlation between the performance degradation and the number of remote invocations performed in all four functionalities. Therefore, in order to improve the performance, an optimization was introduced into the microservice application that effectively reduced the number of remote invocations on all four functionalities to the minimum to improve the performance.

1) Refactoring: From the previous analysis, we concluded that most of the performance degradation came from the high number of fine-grained invocations between the services. Therefore, to establish a more coarse-grained behaviour in the functionalities, an optimization was implemented in the microservice architecture that consisted of increasing the amount of domain information preemptively sent in certain *dtos*, which were frequently utilized together, to replace the remote invocations for local invocations. This would allow reducing the network overheads and improve the performance. By adding additional methods to the *dtos*, we cached the information and supported the composition of several *dtos* in a single object. With this simple change, the information is now accessible through a single invocation, where before would require multiple inter-service invocations. However, it is necessary to consider some trade-offs when choosing the information to send, to avoid sending information that is not necessary which will introduce additional overheads. Therefore, it is necessary to analyse whether the data is frequently used together by the functionalities.

2) Performance: In terms of network usage, the optimization was successful in reducing the remote invocations, while simultaneously improving the performance of the functionality. It could be observed a drastic decrease in the number of remote invocations from the *Source Listing* and *Fragment Listing* functionalities into a minimum value of 1 and 5 remote invocations respectively, independently of the information in the database. In terms of performance, the results confirmed that a major factor of the previous performance degradation came from their fine-grained behaviour. The *Source Listing* functionality significantly improved the latency and throughput with a single coarse-grained invocation and, a similar situation occurred in the *Fragment Listing*. Therefore, it shows the benefits of coarse-grained granularity in the functionalities.

Despite the fine-grained behaviour, the performance of *Interpretation View* also benefited from the reduction of the number of invocations to 48. On the other hand, the *Assisted Ordering* remained with similar performance despite the reduction of the remote invocations, but this is due to the low effects of the remote invocation on the latency of a computational expensive functionality.

However, despite the performance improvements, the final results were still considerably worse than the monolith architectures. This is still the consequence of the network overheads from the remote invocations, even in a coarse-grained context. Note that, a major performance bottleneck came from the serialization/deserialization process that becomes more noticeable as the number of invocations and information increase. An example of this bottleneck occurs in the *Source Listing* functionality where this process corresponds to 82% of the latency of the functionality after the optimization. This is a significant impact that by itself is far superior than the latency of the functionality in the modular monolith.

D. Performance - Deployment

An important aspect of the microservice architecture is the independent deployment of the services and the scalability benefits it offer. The LdoD microservice is composed by independent services that allow to scale the instances and the resources of the services for demanding functionalities. Therefore, it is important to evaluate the benefits of this deployment aspect on the application.

In this section, the performance of different features of the microservice architecture are evaluated in terms of latency and throughput, by running different run-time deployments in a cloud environment under separate network workloads. The main goal is to compare the performance between run-time deployments with different number of instances deployed, to understand the resource usage and scaling benefits of the run-time deployments. Three functionalities of the previous performance test cases were chosen for this scenario: *Source Listing*, *Fragment Listing* and *Interpretation View*.

Two run-time architectures were deployed for this testing. First, a single instance deployment composed by a single instance of each service, in order to provide the performance values for a basic deployment that will be used as the reference base values of the application in a cloud environment. Second, a multi instance deployment composed by five instances of each of the following services: *Text*, *Virtual* and *Front-End*, and a single instance of the remaining services. This allows to evaluate how increasing the resources of specific services affected the performance.

For this performance evaluation scenario, two workload scenarios were designed to measure the performance of the application: a sequential workload and a concurrent workload. The sequential workload simulates a normal usage, implemented with the same load test settings described in the local environment. On the other hand, the concurrent workload is responsible for a heavier usage of the application, which simulates 50 different users concurrently invoking the functionalities and evaluates how both run-time deployments performed under this scenario. This testing was done with the services being deployed into a Google Kubernetes Engine cluster with 8 nodes, 16 vCPU and 32 GB of memory.

From the results of the test cases, it could be observed some benefits and drawbacks of the different run-time versions of the architecture under different usages of the application. In what concerns the concurrent workload, there is a significant throughput increase of running multiple instances of specific services for all three functionalities. The *Source Listing* and *Fragment Listing* functionality, which under normal usage already has significantly high latency values, especially with 720 text fragments in the database, had a throughput increase between 150% to 200%, which is a significant improvement of the scalability. This is due to how deploying more instances increases the use of resources and supports parallel processing of the requests.

Similar performance benefits could also be observed in the *Interpretation View* functionality from the parallel processing. But, note that this functionality has significantly less information and latency which allowed both versions to provide a reasonable end-user experience for such a heavy workload. However, we could also observe how poorly the single instance version of the architecture performed under a concurrent workload for functionalities with large amounts of information like the *Source Listing* and *Fragment Listing*, and with fine-grained invocations like the *Interpretation View*.

Focusing on the sequential workload, the measured results were very similar to the ones obtained in the previous performance evaluation section but with much more latency due to its deployment into the remote cluster and the cluster server location. Note that, there is a slight latency increase in all three functionalities under the multi-instance version when compared to the single version. This can be explained by the necessary internal load balancing that occurs between the services in the multi-instance version.

Overall, we could observe a scaling benefit of a microservice architecture, however, there was a significant performance degradation of running the microservice application in a cloud environment compared to our local deployment. Despite the throughput increase of the multi-instance version, the latency values were significantly high especially for functionalities with large amounts of information like the Fragment and *Source Listing*, resulting in a general bad user experience. This is due to the additional network overheads that are introduced with remote invocation through a real network, which is not fit for large payloads of information or fine-grained invocations.

In the LdoD microservice architecture, two improvements can still be implemented that will benefit the performance in a cloud deployment. First, a redesign of functionalities like the Fragment and *Source Listing* functionalities to implement a pagination pattern, which reduces the information to manageable data sets while maintaining a coarse-grained behaviour. This allows to reduce the network overheads and improve the performance independently of the information in the database. Second, the introduction of additional caches to reduce the number of remote invocations and improve the overall performance of the functionalities.

E. Data consistency

The decentralization of the application data introduces challenges to the architecture concerning the data consistency. This is due to the transactional behaviour between the different databases, where, in some cases, transactions that span across multiple services cannot implement ACID properties between them. In a distributed context, there are four different types of possible transactions between the services: (1) multiple read transactions; (2) single write transaction that ends the sequence of transactions; (3) multiple write transactions that span multiple services or a single write transaction in a service but not the last in the sequence; (4) write transactions that require notifications of the changes to other services.

The LdoD microservice application implemented a simple transactional behaviour between the services, that does not require the implementation of a distributed transaction to preserve the ACID properties on most functionalities. This is due to how the sequence of transactions between the services is mostly composed of read transactions and, in some cases, a single write transaction to end the sequence, which causes no harm to the consistency of the information. Note that, there were a few exceptional cases of multiple write transactions between the services, but by structuring the write transactions, the consistency of the information is respected even in case of failures.

In general, read transactions do not have an effect on the consistency since most of the information of a service does not depend on other services and the local transactions are able to provide the ACID properties in the database. However, when reading information of a domain entity related to events, these transactions are affected by eventual inconsistencies. As previously stated, the LdoD microservice architecture implements an event-driven asynchronous communication between the services that, due to the decentralized data management, introduces eventual consistency to the architecture. This communication addresses the write transactions between the services that require notification to be kept consistent, but at the same time it does not offer the usual ACID properties to the databases. Therefore, the eventual consistency is an important aspect that was introduced into the LdoD Archive that needs to be evaluated in terms of the effects on the information and its impact on the functionalities of the application.

With the introduction of the decentralized data approach and the asynchronous event-based communication, the application depends on the use of the different types of events to achieve data consistency between the different services. Most of the events are related with the remove of a domain entity. For instance, upon the removal of a `Fragment` from the *Text* service, the *Virtual* service needs to receive the appropriate event to delete any reference to this text fragment from the database. This means that it is important to understand the different type of events sent between the services, how they affect the data consistency of the different services and how eventual states affect the different functionalities.

In the microservice architecture, it could be observed nine different types of events that affect different services and domain entities with different degrees of frequency and impact on the information. Most events are published upon the removal of domain entities that are referenced from services that use the modules where the events occur, which affects their consistency. Additionally, the events can trigger a chain of events which further increases the inconsistency of the application.

Concerning the impact of the events, the domain entities `Fragment` and `ScholarInter` from the *Text* service, and `User` from the *User* service have the highest impact on the consistency from the information of the application, affecting three different databases and multiple domain entities. In addition, they are also responsible for a chain of events in the *Virtual* service by triggering three different types of events, `VirtualEditionInter-Remove`, `VirtualEdition-Remove`, and `Tag-Remove`, that further increase the inconsistency in the services. Note that, the impact of establishing multiple databases in this case study correlated to the indirect associations of the domain entities.

The events have a higher impact if they trigger a large number of changes, which fortunately, are the ones that have a lower frequency. On one hand, the archive has a predefined set of fragments that do not change, which means that they are almost like immutable entities. On the other hand, it is very uncommon to delete the archive users. Therefore, the *Text* and *User* service information remains static throughout the execution of the application under a normal context and are only removed under exceptional contexts with administrator privileges, which reduces the complexity of managing the consistency.

Therefore, under a normal context, most of the inconsistencies result from the *Virtual* service subdomain and their associations to the `ClassficationGame`. The *Virtual* service has a significant number of events that are frequently published because a end-user can interact with its virtual edition, removing some of their entities. On a positive note, the impact of the inconsistent information is mainly focused on the Game service and has a low effect on the overall information of the application.

In another perspective, it is important to analyse how eventual consistency affects the application from the perspective of an end user, by evaluating the behaviour of the different functionalities under inconsistent states. The data inconsistencies may have different types of impact, depending on the specific functionality and the type of events. In the *Interpretation View*, the functionality performs as expected despite any inconsistent information between the *User* and *Virtual* service displaying the presence of tags and categories of the removed user until the databases are eventually consistent. This behaviour is the result of no direct communication between the *Virtual* and *User* services, since the basic information of the user is duplicated in the *Virtual* database. Therefore, having a low impact on the behaviour of the functionality.

On the other hand, the *Virtual Edition Listing* and *Game Listing* functionalities present a more serious consequence of the inconsistent information in the architecture, where the functionalities cannot perform under inconsistent states. This occurs because when the functionalities try to obtain the data from the services, it is not present, resulting in a failed request.

In general, the decentralized data approach and eventual consistency achieved through the event-driven asynchronous communication proved to be an efficient alternative to a distributed transactional behaviour between the services in the LdoD microservice architecture, where most of the functionalities presented the same behaviour and were not affected by inconsistencies between the databases. This is mainly due to the simple transactional behaviour of the functionalities that do not span across different services, which allows for in most situations to apply the ACID local transactions and maintain the information consistent, thus having an overall low impact.

To improve the LdoD Archive performance, different caches were implemented in the modular and microservice architecture. However, throughout their use in the application, the caches might become inconsistent as the information changes and affect the consistency of the application. In this case study, the overall impact from inconsistencies in the caches was low due to the immutability of the information, which minimized the drawbacks of caching the information. However, it could also be stated that as the mutability of the information increases, the worse the effects on the consistency of the application. Therefore, it is important to consider which information can be cached and how the inconsistent information can affect the application.

VI. Discussion

The process of modularizing a monolith and migrating it to a microservice architecture requires extensive refactoring of the application and has a significant impact on the performance. The modular monolith and microservice architecture share the modularization requirements to address the decomposition of the domain into the modules/services, while also addressing the granularity of the interaction between the domain entities for performance reasons. Therefore, the modular monolith offers a beneficial groundwork for achieving a microservice architecture that massively reduces the refactoring effort by accomplishing the modularization process through well-encapsulated modules that serve as the foundation of services and reduce the development effort.

Through the evaluation, we addressed some concerns that are often neglected in the literature that focus more on technical aspects like communication technology, running environments, and performance benchmarks. In what concerns the refactoring effort, the inter-service communication requires changes in the interfaces of each service and the cost is directly related to the size and quality of the API. Even though the refactoring is composed of small changes, a poor quality interface can increase the refactoring cost and propagate the changes to the service features. Therefore, if these constraints are considered when designing the modular interfaces, it will help in the introduction of remote invocations.

On the other hand, we could also observe a consequence of migrating a modular monolith with a significant number of uses relationships like the LdoD Archive in terms of coupling between the services. As previously stated, the behaviour of the uses interaction corresponds to a synchronous request/response style of communication between the services in order to maintain the dependencies. However, the synchronous communication results in the coupling of the services being too tight due to the fine-grained behaviour, which becomes problematic due to the weight of remote invocation and results in serious performance degradation.

Therefore, in the context of a stepwise migration of a monolith into a microservice architecture, the intermediate step of a modular monolith is advantageous, because it highlights complexities that might have to be addressed before implementing a microservice architecture and helps on the decision on how to migrate. Note that with the modular monolith, the developers can predict the coupling of the services, the expected performance degradation of the communication and decide the type of communication between the services to detect functionalities that can be affected by the lack of ACID transactional behaviour.

In what concerns the performance, the impact on the performance was a major factor from the migration into a modular and microservice architecture that was thoroughly evaluated. Focusing on the modular monolith, the impact on the performance was related to the amount of information sent between the modules through the *dtos*, while the number of inter-module invocations did not have a relevant effect. Note that, the modular monolith, when the amount of information transferred between modules is low, can match and even obtain better performance in some cases due to faster accesses to the database through the unique identifiers that were introduced with the modularization. However, contrary to the modular monolith, the performance of the microservice architecture was also affected by the number of inter-service invocations. These two factors combined should be avoided when designing the microservice architecture, where there is a performance degradation as the amount of transferred information and the number of invocations increases.

In the context of the migration process, this study confirmed the complexity of synchronous communication in a microservice architecture in both a local and cloud scenario and addressed some possible performance optimizations, however the performance degradation proved to be far too severe compared to the scalability benefits. A positive side of the LdoD microservice architecture was the evaluation of the data consistency, since it showed an example of a simple case of eventual consistency which did not require complex solutions like SAGAs or two-phase commit like protocols to maintain the information consistent. However, this solution does not address cases with multiple write transactions that require some sort of compensation in case of failures, which would significantly increase the migration effort and introduce new challenges.

The following threats to the validity of this study were identified: (1) it is a single example of a migration; (2) it depends on the technology and programming techniques used in the monolith.

Despite being a single case study, it has some level of complexity and the literature lacks descriptions of the problems and solutions associated with the migration from monolith to microservices architecture. In this study, it was addressed both the migration into a modular and microservice and provides feedback of challenges faced in the migration that benefits the overall process.

The technology and programming techniques used in the implementation of the monolith follow an object-oriented approach, where the behavior is implemented through fine-grained interactions between objects. A more transaction script based architecture may result in different types of problems. The conclusions of this study apply when the monolith is developed using a rich object-oriented domain model. On the other hand, the monolith is implemented using Spring-Boot technology which follows the standards of web application design.

Additionally, the case study presented specific types of functionalities that either requested significant amounts of information or was implemented through fine-grained invocations with a strong synchronous behaviour. Different type of functionalities can present other performance results, but in general most functionalities in the LdoD Archive were similar to the Interpretation View. Therefore, we provided a good coverage of the functionalities.

VII. Related Work

There are several challenges when migrating from monolith systems to a microservices architecture [12], [13], such as the effort to redesign the monolith and the performance impacts, and we can find in the literature the description of the migration of some large monoliths [14]–[16].

There are several reports on the migration of large monoliths to a microservices architecture. The migration described by Gouigoux and Tamzalit [14] discusses the aspects of service migration, deployability and orchestration. It reports an improvement in the performance, but does not describe details of the refactorings and optimizations done. Bucchiarone et al [15] describe the lessons learned from their migration of a monolith in the banking domain, focusing on the benefits and challenges of the new system, but doe not discuss the migration effort and the impact on performance. Barbosa and Maia [16] discuss the migration of a large monolith to a microservices architecture, where the monolith business logic is implemented using stored procedures and they focus on the process of identifying microservices. Megargel et al [17] provided a a practice-based view and a methodology to transition from a monolith application into a cloud-based microservice architecture. However, the migration effort was not addressed but they did observe benefits from the migration including on the performance. On the other hand, an opposite point of view was presented by Mendonça [18] that discussed the decision to revert the microservice architecture back into a monolith, due to the burdens of a microservice architecture, but even in this case, refactoring and performance were not addressed.

Therefore, it is necessary to have more case studies that describe the migration efforts and the architectural trade-offs, besides the advantages, and drawbacks, of the final product. In this paper we contribute by describing the refactoring, and related effort, associated with the migration of a large monolith to a modular monolith, which can be used as an intermediate step of the migration.

On the other hand, there is work on impact that the migration of a monolith to a microservices architecture has on performance, although it follows different perspectives.

Ueda et al [8] compare the performance of microservices with monolith architecture to conclude that the performance gap increases with the granularity of the microservices, where the monolith performs better. Villamizar et all [7] show different results, concluding that in some situations the performance is better in the microservices context and that it reduces the infrastructure costs, but request time increases in microservices due to the gateway overhead. Al-Debagy and Martinek [10] conclude that they have similar performance values for average load, and the monoliths performs better for a small load. In a second scenario the monolith has better throughput, but similar latency, when the system was stressed in terms of simultaneous requests. Bjørndal et al [19] benchmark a library system, that has 4 use cases and considers synchronous and asynchronous relations between microservices. They observe that monolith performs better except for scalability. Therefore, they identify the need to carefully design the microservices, in order to reduce the communication between them to a minimum, and conclude that it would be interesting to apply these measures in systems that are closer to the kind of systems used by companies. Guamán et al [20] designed a multi-stage architectural migration, compared the performance of the monolith stages to the microservice architecture and observed a worse latency from the microservice architecture. Flygare et al [21] found that in their case study the monolith performed better in terms of latency and throughput while consuming less resources than the microservice architecture.

Some other perspectives compare the performance of monolith and microservices systems in terms of the distributed architecture of the solution, such as master-slave [22], the characteristics of the running environment, whether it uses containers or virtual machines [9], the particular technology used, such as different service discovery technologies [10], or other microservices deployment aspects [11]. A major aspect of the performance is the type of inter-service communication and the technology used. Hong et al [23] compare the performance of synchronous and asynchronous communication and concluded that the asynchronous approach offered a more stable performance overall but with a lower response request performance. Fernandes et al [24] presented similar results with the asynchronous communication outperforming the REST communication in performance and data loss prevention on a large data context. Shafabakhsh et al [25] built upon Fernandes et al [24] research and conclude a benefit of synchronous communication under small loads.

These results show that an asynchronous communication tends to be more suitable to a microservice architecture, however a behaviour suited synchronous communication like the REST API is more accessible to the migration from a modular monolith. Our approach allows to separate concerns when measuring the performance impact of the migration, and led us to conclude that it is already visible when migrating to a modular monolith, in the absence of distributed communication and microservices implementation technologies. And contrarily to some of the related work, in which there is no redesign of the functionalities of the monolith, we highlight that such redesign is required and has impact on the performance.

VIII. Conclusion

With this work, we described the migration from a large object-oriented monolith into a microservice architecture using a modular monolith as a middle stage, while analysing the migration effort to achieve both, the modular and microservice architectures and the overall impact on the performance.

The results showed that the modular monolith can be used as an intermediate artifact that facilitates the migration into the microservice architecture, while providing a more agile software development environment before tackling the challenges of a microservice architecture. In addition, the modular monolith also helped in addressing concerns regarding the inter-service communication, eventual consistency and performance optimizations before the migration.

The migration into a microservice architecture revealed a significant impact on the application for both refactoring and performance. Most of the migration cost was connected to the modules and the interfaces to implement services with the desired inter-service communication, in which the quality of the modules and interfaces has a significant impact on the cost. On the other hand, a serious consequence of the migration was the large impact on performance associated with the inter-service communication, which required functionalities to be redesigned into having coarse-grained interactions.

Overall, the migration to a microservices architecture presented several challenges with different levels of impact on the refactoring effort, performance, and data consistency, which highly depended on the application structure and semantics. In the LdoD Archive, the migration presented a serious impact on the performance due to the network overheads that proved to be far too high compared to the previous architectures, but on the other hand, it offered a more scalable and manageable architecture.

References

[1] E. Evans, *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] C. Richardson, *Microservices Patterns*. Manning, 2019.

[3] W. F. Opdyke and R. E. Johnson, "Creating abstract superclasses by refactoring," in *Proceedings of the 1993 ACM Conference on Computer Science*, ser. CSC '93. New York, NY, USA: Association for Computing Machinery, 1993, p. 66–73. [Online]. Available: https://doi.org/10.1145/170791.170804

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., 2018.

[5] D. Haywood, "In defense of the monolith," in *Microservices vs. Monoliths - The Reality Beyond the Hype*. InfoQ, 2017, vol. 52, pp. 18–37. [Online]. Available: https://www.infoQ.com/minibooks/emag-microservices-monoliths

[6] M. Fowler and J. Lewis, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[7] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *2015 10th Computing Colombian Conference (10CCC)*, 2015, pp. 583–590.

[8] T. Ueda, T. Nakaike, and M. Ohara, "Workload characterization for microservices," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 2016, pp. 1–10.

[9] A. M. Joy, "Performance comparison between linux containers and virtual machines," in *2015 International Conference on Advances in Computer Engineering and Applications*, 2015, pp. 342–346.

[10] O. Al-Debagy and P. Martinek, "A comparative review of microservices and monolithic architectures," in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, 2018, pp. 149–154.

[11] F. Tapia, M. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied Sciences*, vol. 10, no. 17, 2020.

[12] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: An industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29–2909.

[13] M. Kalske, N. Mäkitalo, and T. Mikkonen, "Challenges when moving from monolith to microservice architecture," in *Current Trends in Web Engineering*, I. Garrigós and M. Wimmer, Eds. Cham: Springer International Publishing, 2018, pp. 32–47.

[14] J. Gouigoux and D. Tamzalit, "From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017, pp. 62–65.

[15] A. Bucchiarone, N. Dragoni, S. Dustdar, S. T. Larsen, and M. Mazzara, "From monolithic to microservices: An experience report from the banking domain," *IEEE Software*, vol. 35, no. 3, pp. 50–55, 2018.

[16] M. H. Gomes Barbosa and P. H. M. Maia, "Towards identifying microservice candidates from business rules implemented in stored procedures," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 41–48.

[17] A. Megargel, V. Shankararaman, and D. K. Walker, "Migrating from monoliths to cloud-based microservices: A banking industry example," in *Software Engineering in the Era of Cloud Computing*. Springer, 2020, pp. 85–108.

[18] N. C. Mendonca, C. Box, C. Manolache, and L. Ryan, "The monolith strikes back: Why istio migrated from microservices to a monolithic architecture," *IEEE Software*, vol. 38, no. 05, pp. 17–22, sep 2021.

[19] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F. B. Kessler, and T. Wien, "Migration from monolith to microservices: Benchmarking a case study," 2020, unpublished. [Online]. Available: http://10.13140/RG.2.2.27715.14883

[20] D. Guaman, L. Yaguachi, C. C. Samanta, J. H. Danilo, and F. Soto, "Performance evaluation in the migration process from a monolithic application to microservices," in *2018 13th Iberian Conference on Information Systems and Technologies (CISTI)*. IEEE, 2018, pp. 1–8.

[21] R. Flygare and A. Holmqvist, "Performance characteristics between monolithic and microservice-based systems," Bachelor's Thesis, Faculty of Computing at Blekinge Institute of Technology, 2017.

[22] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34.

[23] X. J. Hong, H. S. Yang, and Y. H. Kim, "Performance analysis of restful api and rabbitmq for microservice web application," in *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 2018, pp. 257–259.

[24] J. L. Fernandes, I. C. Lopes, J. J. P. C. Rodrigues, and S. Ullah, "Performance evaluation of restful web services and amqp protocol," in *2013 Fifth International Conference on Ubiquitous and Future Networks (ICUFN)*, 2013, pp. 810–815.

[25] B. Shafabakhsh, R. Lagerström, and S. Hacks, "Evaluating the impact of inter process communication in microservice architectures." in *QuA-SoQ@ APSEC*, 2020, pp. 55–63.