TÉCNICO LISBOA

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

Diogo Alexandre dos Reis Faustino

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisor: Prof. António Manuel Ferreira Rito da Silva

Examination Committee

Chairperson: Prof. Pedro Miguel dos Santos Alves Madeira Adão
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Filipe João Boavida Mendonça Machado Araújo

October 2021

Acknowledgments

I would like to thank my parents for their friendship, encouragement and caring over all these years, for always being there for me through thick and thin and without whom this project would not be possible. I would also like to thank my grandparents, aunts, uncles and cousins for their understanding and support throughout all these years.

I would also like to acknowledge my dissertation supervisors Prof. António Rito Silva for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times in my life. Thank you.

To each and every one of you – Thank you.

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 thesis, 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 microservice 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.

Keywords

Domain-driven design, Modular architecture, Refactoring Cost, Performance evaluation, Microservices, Eventual consistency.

Resumo

A agilidade inerente dos negócios de hoje promove a definição de arquiteturas de software onde as entidades de negócio são dissociadas em módulos ou serviços. No entanto, existem vantagens em ter um modelo de domínio rico, onde as entidades de domínio estão fortemente conectadas, porque promove a reutilização. Por outro lado, a divisão da lógica de negócios em módulos ou serviços, encapsulados por meio de interfaces bem definidas e a introdução de comunicação entre-serviços promove um desenvolvimento ágil mas apresenta um custo em termos de desempenho.

Nesta tese, analisamos o impacto de migrar um monolítico de domínio rico para uma arquitetura modular e sequencialmente para uma arquitetura de microserviços, tanto em termos do custo de desenvolvimento associado à refatoração, como ao custo do desempenho associado à execução. O estado atual da arte analisa a migração de sistemas monolíticos para uma arquitetura de microserviços, mas observámos que o esforço de migração e os problemas de desempenho já são relevantes na migração para um monólito modular e concluiu-se o impacto de estabelecer uma arquitetura de microserviços com um modelo de domínio rico e uma comunicação entre-serviços no desempenho. Além disso, também abordamos desafios exclusivos da arquitetura de microserviços, como a consistência eventual das bases de dados e a instalação dos serviços.

Palavras Chave

Design orientado a Domínio, Architetura Modular, Custo de refactorização, Avaliação de desempenho, Microserviços, Consistência Eventual

Contents

List of Figures

List of Tables

Listings

1

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 development cost, because of the need to refactor modules' interfaces, since its initial design does not capture the correct abstractions. Also, 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 an 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 are isolated through an API that encapsulates its implementation and provides the anticorruption 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 of the architecture such as inter-process communication and decentralized data management.

A key aspect when migrating into a microservice architecture from a modular context is the communication mechanism between the services that has to replace the inter-module invocations, which implies significant changes to the interfaces and impact on the performance to adopt the appropriate communication. 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 to be migrated into the microservices architecture.

One of the main goals of this thesis is to describe the refactoring process of a step-wise migration of a monolith with a large domain into a microservice architecture using the modular monolith as an intermediary step. Through the modular monolith, most of the refactoring focuses on decomposing the

modules into independent services with well-defined API and provides insight on the type of refactoring to achieve several aspects of the microservice architecture, such as inter-service communication. This thesis also measured the impact of the refactoring process on the cost to implement the microservice architecture by evaluating the refactoring effort and the advantages of a step-wise migration.

In this thesis, we evaluated several aspects of the microservice architecture to understand the impact of the migration on performance and data consistency. We also described optimization tactics to improve these architectural aspects and compared them to the performance and optimizations applied to the modular monolith. The results showed a negative impact on the performance in each step of the migration. In the modular monolith, data transformation of the anti-corruption layers affected the performance, even in the absence of remote invocations. On the other hand, remote communication also had a severe impact on the performance of the microservice architecture due to network overheads and fine-grained interactions between the services.

This thesis is organized as follows: chapter 1 contextualizes the modularization procedure and the migration into a microservice architecture with the goals of this thesis, chapter 2 provides a literature review of related works of the migration from a monolith into a microservice architecture and the different aspects of the architecture, chapter 3 provides the background on the different architectures of the LdoD Archive, monolith and modular monolith, with an evaluation of the refactoring and performance costs, chapter 4 provides the overview of the migration of a modular monolith into a microservice architecture, the overview of the LdoD microservice architecture and the evaluation of the refactoring costs, chapter 5 provides the performance evaluation of the microservice architecture, the evaluation of the data consistency and the discussion of the observed migration and, finally, in chapter 6 we draw the conclusions of the migration into a microservice architecture and the future considerations.

2

Related Work

Contents

2.1 Monolith to Microservice

In order to migrate from a monolith system into a microservice architecture, there are several challenges [12–14], that the developers need to face, such as the effort to redesign the monolith and the performance impacts. There are in the literature the description of the migration of some large monoliths [15–17] which provide reports of the migration process into a microservice architecture.

The migration described by Gouigoux and Tamzalit [15] discusses the aspects of service migration, deployability and orchestration, however it does not discuss how to achieve the desired service granularity in detail. Instead, the migration focuses on the strong link between the service and the container technology used for the running environment and the architectural choices for a deployment that fitted the application under study. A surprising improvement of the performance in terms of latency was observed as the main benefit from the migration, but it was not discussed the refactoring and optimizations applied to the architecture in order to achieve it.

Bucchiarone et al [16] discusses the experience of the migration from a real world monolith scenario of a banking domain into a microservice architecture, focusing on the lessons learned, benefits and faced challenges. It provides an overview from the migration into a microservice architecture but it does not address the refactoring effort required or the impact of the performance, focusing on aspects like the scalability, automation, monitoring and other aspects related to the deployment. Barbosa and Maia [17] discuss the migration of a large monolith to a microservice architecture, where the monolith business logic is implemented through stored procedures and they focus on the identification of the services.

Megargel et al [18] provided a practice-based view and a methodology to transition from a monolith application into a cloud-based microservice architecture. However, despite addressing the different technology aspects and describing the migration process, the refactoring 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 [19] that discussed the decision to revert the microservice architecture back into a monolith, focusing on the burdens from the microservice architecture and the lessons learned. The main burden of the microservice architecture came from the inability to independently manage and deploy the services, which consequently resulted in the developers facing a more complex architecture to develop without any benefits. Additionally, the researchers also addressed the cost of different aspects from the microservice architecture such as the scalability, security isolation and deployment, but even in this case, refactoring effort and performance were not addressed.

Therefore, it is necessary to have more research to describe the migration efforts and performance impact of the migration of a monolith to a microservice architecture.

2.2 Microservice Performance Evaluation

In what concerns the performance of a microservice architecture, there are several research studies on this topic that cover different subjects relevant to this study, such as the impact from migrating a monolith into a microservice on the performance, and the impact of the chosen technologies, like the running environment and inter-service communication protocols.

2.2.1 Monolith and Microservice Comparison

Ueda et al [8] compare the performance of microservices with the monolith architecture to conclude that the throughput gap increases with the granularity of the microservices, in which the monolith performed significantly better and suggests that improving the communication between the services is a key factor to improve the performance under a microservice model. Villamizar et al [7] shows different results, concluding that in some situations the latency of the microservice architecture is better and reduces the infrastructures costs, despite the increase of hosts including a gateway application.

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. Guamán et al [20] designed and implemented a step-wise architectural migration into a microservice architecture by identifying and decomposing the functionalities, defining REST API for the functionalities in the monolith and converting them into independent services. Despite not addressing the migration effort, the researchers compared the performance of the monolith and the intermediary stages to the microservice architecture and concluded a worse performance in terms of latency from the microservice architecture compared to the monolith stages.

Bjørndal et al [21] 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. Flygare et al [22] found that in their case study the monolith performed better in terms of latency and throughput while consuming less resources than the microservice architecture. In addition, it also observed some throughput benefits in running the microservice in a cluster instead of running in a single computer.

Some other perspectives compare the performance of monolith and microservices systems in terms of the distributed architecture of the solution, such as master-slave [23], 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, 24].

2.2.2 Inter-service communication

A major aspect from the microservice architecture is how the services exchange information between them, how it affects the performance and does the technology affect the results. Hong et al [25] evaluated the performance from different communication methods, REST API and RabbitMQ, under different circumstances and concluded that RabbitMQ offered a more stable performance overall but with a lower response request performance. Fernandes et al [26] also compared the REST API performance to RabbitMQ as the Advanced Message Queuing Protocol asynchronous communication and obtained similar conclusions where the RabbitMQ far outperformed the REST API in performance and data loss prevention for intensive amounts of information. Shafabakhsh et al [27] performed an experimental research to compare the popular forms of IPC between the services to build upon Fernandes et al [26] research and concluded that REST API performs well under low loads while asynchronous approaches scale better under higher 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 and reduces the complexity of redesigning the services into an asynchronous behaviour. On the other hand, there are different technologies that have an impact on the performance of the application. Johansson [28] compared the REST API and GraphQL API and also showed that the serialization technology can have an impact on the performance of the communication. In this study, the REST API with JSON performed consistently better than the addressed alternatives but it was also mentioned that the performance from REST API would improve if a binary serialization protocol were to be utilized instead of the JSON structure.

Jayasinghe et al [29] analysed the impact of decomposing the services in terms of performance and observed that the services performance correlates to different factors, such as service demand, concurrency, decomposition strategy and the number of remote calls. Therefore, we can observe a relation to the performance of the inter-service communication and the technology like REST that will be further discussed in the thesis.

2.3 Data consistency

An additional concept introduced with the migration from a monolith into a microservice architecture is the decentralization from the data management since each service can be responsible for managing a dedicated database and maintaining the data consistent between them. However, there are several challenges in order to address the data management and consistency of the information in the microservice architecture. Furda et al [30] provided a description of the data consistency challenge and how the eventual consistency pattern can be applied, while Laigner et al [31] determines the short comings from

state of the art data management systems and evaluates the challenges associated with methodologies such as the eventual consistency.

In what concerns the eventual consistency, Rudrabhatla [32] evaluated different aspects of eventual consistency techniques, such as performance and complexity, in the context of a microservice with NO-SQL databases, but did not discussed the refactoring and complexities addressed to implement these techniques. In another research, Laigner et al [33] design the migration from a big data system monolith into an event-driven microservice architecture that utilize events to communicate between the services but did not address the eventual data consistency techniques directly and mentioned that complex data flow proved to be difficult to troubleshoot.

Kookarinrat and Temtanapat [34] described a very in depth decentralized microservice approach where it described types of communication similar to the implemented notification interaction in the LdoD Archive, however it did not address the consistency of information between the services in the architecture. Lesniak [35] provided a few real examples from a microservice architecture where eventual states proved to have a significant impact on the integrity of the information and how it affected the functionalities. However, despite addressing these challenges, the researches did not mention the effort to resolve them.

Therefore, through the work applied in this thesis, we intend to offer an additional case study from a mostly synchronous architecture that utilizes an event driven approach to achieve eventual consistency and measure the impact of eventual states in the application.

3

LdoD Background

Contents

The LdoD Archive web application is a digital archive for digital humanities focused on ”The Book of Disquiet” by Fernando Pessoa that offers different features, like searching, browsing, and viewing the original text fragments, different variations of the text fragments, as well as different editions of a book. In addition, the users can create and manage their own virtual editions and taxonomies of the book, capable of adding or removing fragments as they wish, and perform different features that measure the proximity distance from each fragment according to a set of available criteria like tf-idf. Other relevant features can also be observed such as a game feature, which implements a serious game, a reading feature, which implements reading sequences from text fragments of the book, and a visual feature, which provides a graphical visualization between different text fragments.

The initial development of the LdoD application resulted in a monolith architecture with a strongly coupled rich domain model A.1 composed by 71 domain entities with 81 bidirectional associations between them. The monolith architecture promoted high reusability due to how the business logic was appropriately divided into domain entities and how efficient it was to navigate in the domain model through the bidirectional relations. In more practical terms, the LdoD Archive was implemented using a Domain-driven design in the Java language with the Spring-Boot framework, responsible for creating a web-based application. In addition, the Fenix-Framework project was utilized as the Object-Relational Mapper that manages the domain model by handling all the transactional behaviour, database management, and domain specifications. This specification is implemented through a custom Domain Modelling Language file (DML) that specifies all the domain entities with the respective associations between them.

However, the monolith, when faced with new requirements to support digital archives from additional books, it was necessary to separate certain features that were common between the different digital archives, like the management of texts, from LdoD specific features that could not be applied to other digital archives. This motivated the decision to decompose the LdoD Archive into a set of modules responsible for the different features of the application.

3.1 Modular Decomposition

Through the work of a previous thesis [36], the monolith domain model was decomposed into several non-interrelated subdomains that resulted into a set of modules, which are encapsulated through interfaces. The modular monolith defines two different types of interactions between the modules: the uses interaction that represent the dependencies from a module subdomain, where a module uses another to obtain the necessary information exclusively through their interfaces, and a notification interaction, where the module notifies any subscribed modules from any relevant changes. The correct application of these two types of inter-module interactions allows to define a modular architecture without any circular dependencies between the modules. The uses interaction defines a dependence in which the

module that uses requires the used module to be present. However, the notification interaction does not define any dependencies, since the behaviour of the module that notifies does not depend on the correct implementation of the module that subscribes to the events.

The decomposition process distinguishes two different types of modules: back-end modules, with well-established interfaces that contain a part of the domain model and are responsible for providing the application features, and front-end modules, responsible for providing the end-user interface while interacting with the back-end modules. In addition, the interfaces are defined accordingly to the uses interaction where a module that is used implements a *provides interface* that matches the *requires interface* from a dependent module. The interface is responsible to transfer information from the subdomain entities in a shared format through data transfer objects (*dtos*), to enforce a good encapsulation of the modules with an anti-corruption layer, which allows for a better development process with smaller teams independently working on each of the modules.

Focusing on the notification interaction, the modules can notify subscribed modules of relevant changes through an event-based communication that shares events with minimal but relevant information to inform the necessary modules without creating a dependency between them. This is also achieved through interfaces, more specifically a *publish interface*, for the module that notifies, and a *subscribe interface*, for the modules subscribed to notifications, that establish an inverse flow of control with no dependencies, where they only need to agree on the event structure.

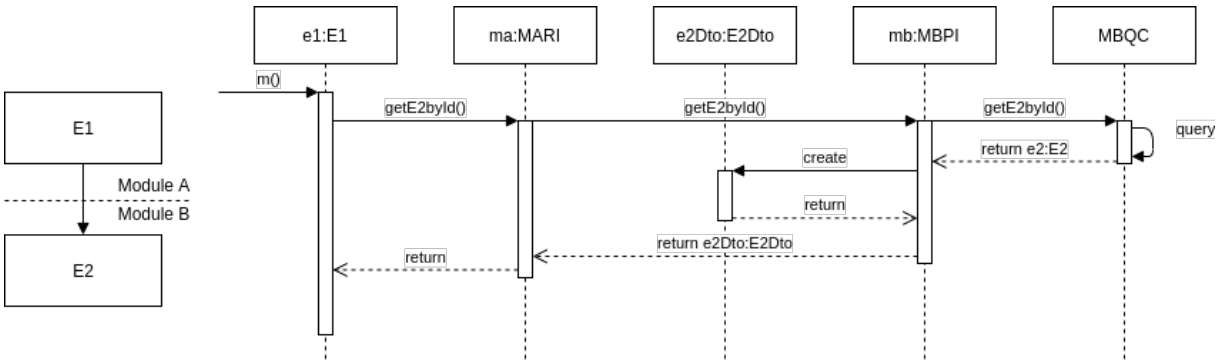

Figure 3.1: Example of an entity obtaining a *dto* of a domain entity in another module

Figure 3.1 illustrates an example from a uses relationship between two separate modules, `A` and `B`, and two domain entities, `E1` and `E2`, that used to have a unidirectional association before the decomposition. The left side from the Figure 3.1 shows the unidirectional association that needs to be removed and refactored into a dependency from module `A` to module `B`, so that the former is aware of entities of the latter, while the latter is independent and can notify the module `A` from any relevant changes. Note that, while module `A` can be dependent on the domain information of `E2`, the `E2` domain entity is never transferred between the modules, and instead a `E2Dto` data transfer object is defined and transferred.

In addition, the right side of the figure defines the sequence of actions from the uses interaction

between module A and module B, where module A obtains an instance of e2Dto:E2Dto. This interaction initiates from the execution of the method m that interacts with the domain entity e1:E1 and requires information from the domain entity e2:E2. To obtain the e2Dto:E2Dto, module A requires interface (MARI) requests the information through the function getE2byId(), that matches a function provided by the module B provides interface (MBPI), and a unique identifier for the entity e2:E2 obtained from the e1:E1 that represents an indirect association to it. Upon receiving the request, module B provides interface (MBPI) queries module B singleton query class (MBQC), to obtain the E2 instance through the unique identifier sent by module A, creates a e2Dto:E2Dto from the queried e2:E2 domain entity and returns it to MARI.

3.2 LdoD Modular Architecture

In what concerns the LdoD application, the domain model was decomposed into several different modules, each associated to the main features of the archive, that established uses and notification interaction between them to maintain the dependencies from these features:

- Text: represents the base functionality of this application and focuses on the information of the text fragments, their interpretations and the expert editions;
- User: represents the base user functionalities of the archive, such as registration, authentication and access authorization;
- Virtual: represents the features related to the virtual editions, such as their creation, tagging, categorization and managing. To maintain the dependencies between the virtual editions, it uses the *Text* and *User* module to access the text fragments and users information associated with the respective editions;
- Recommendation: contains the functionalities that calculate the similarity distances, given a set of criteria like the tf-idf (Term Frequency Inverse Document Frequency), between the fragments and uses the *Virtual* and *Text* module information;
- Game: contains some features that assist a serious tagging game offered by the archive that classifies the fragments from a virtual edition and uses the *Virtual* module;
- Search: represents the search feature that allows to search the archive for all types of interpretations and editions in the *Text* and *Virtual* modules;
- Reading: uses *Text* and *Recommendation* modules to provide the reading recommendation functionalities;

- Visual: offers a graphical interaction with the archive, using the *Recommendation*, *Virtual* and *Text* modules;

In addition, the modular monolith has a server-side front-end module that provides the user interface and is implemented using a server-side technology, Java-Server Pages (JSP). Additionally, there are two client-side front-end modules responsible for providing the *Visual* and *Game* modules user interface. Besides the described uses interactions, the modules also interact through notifications to guarantee that the state is kept in a consistent state. An example from this interaction occurs in the *Text* module upon the removal of a text fragment, where this module publishes an event that is subscribed by the *Virtual* module that removes the fragment from all the virtual editions that refer to it.

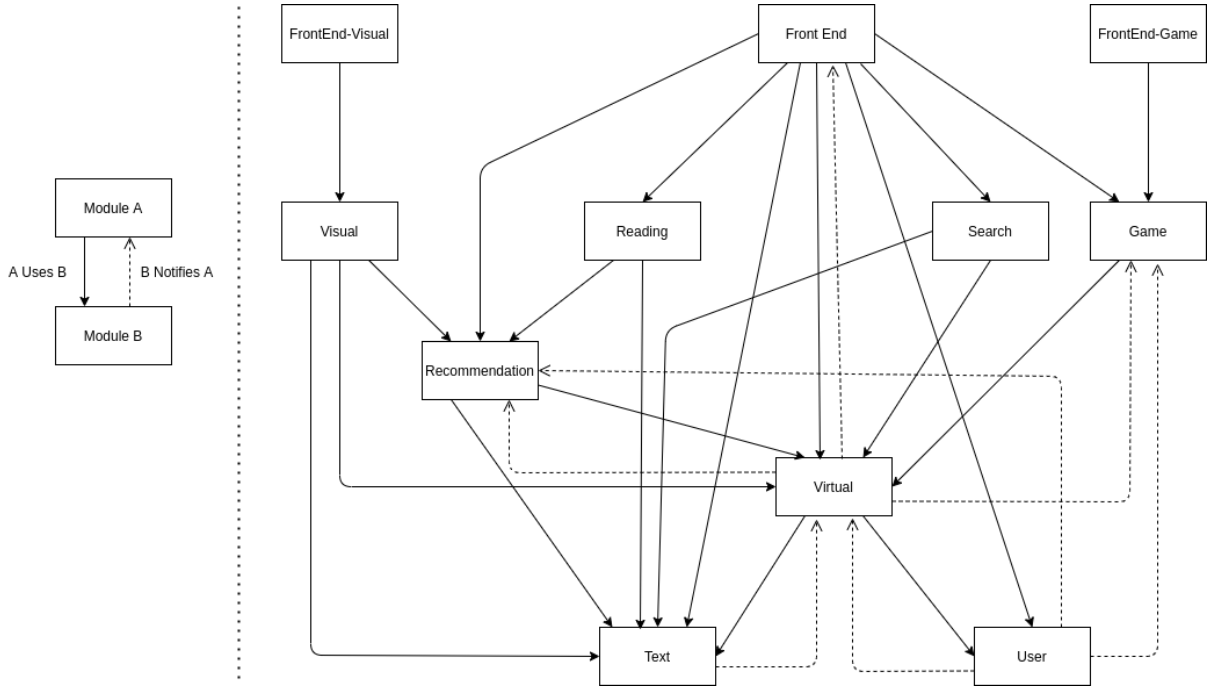

Figure 3.2: Modular architecture overview with the uses and notification interactions

Figure 3.2 presents an overview of the LdoD modular monolith architecture with the different types of modules and their respective uses and notification interactions. Note that, the core modules *Text* and *User* only implement a *provides interface* since they do not rely on any module, while the remaining back-end modules implement a *provides* and *requires interface* to respectively provide their functionalities and request services from other modules. The *Front-End* module, on the other hand, provides the entry-way for the user requests through the web controllers and implements a *requires interface* that connects to the different back-end modules to provide the functionalities. An aspect of this architecture is how decomposing a rich domain into several different modules can result in a larger number of interactions between the modules that, consequently, affects the performance as the application grows larger and

new modules are introduced. On the other hand, achieving this architecture benefits the development process by allowing smaller teams to work on different modules independently and offers the groundwork to further migrate the architecture into a microservices architecture.

3.3 Refactoring to Modular Monolith

To decompose the LdoD domain model, the refactoring effort focused on the associations between domain entities that belonged to different subdomains, in a way that these relations are replaced by either an uses or notification interaction and avoid any circular dependencies between the modules. This associations also require the implementation of *dtos* and events to implement the interactions in the modular context. Therefore, the cost from this refactoring is associated with the number of relationships between the domain entities of different modules. Note that, indirect relations are also an issue that needs to be dealt with since, despite not existing relation between two entities belonging to different modules. It may be possible that information of the entities to be exchanged between the modules as a parameter of the invocation. This is a dependence problem addressed by the Demeter Law [37].

Another important aspect that needed to be addressed was the presence of god classes that centralized the intelligence of the system. In the LdoD domain model, this behaviour was found in the form of a superclass named `LdoD`, implemented as a singleton that connected to all the domain entities from different subdomains and needed to be separated into different modules. This refactoring was not complex and involved the identification of which methods belong to a subdomain and moved them into a new singleton inside a module. The only exception is when the `LdoD` class has methods with multiple inter-modules accesses. A similar situation occurred in the presence of inheritance associations between modules, where a parent class had subclasses present in different modules. The solution, in this case, was to duplicate the code in the superclass in each one of the subclasses.

For the identified aspects, the general refactoring solution was removing the associations between the modules by keeping a unique identifier of the domain entity used in the different module that uses it. This unique identifier is used to obtain the *dto* with the information which was, previously, accessed through the removed association. It is also used in the events, to inform about the occurrences in the domain entities.

	Text	User	Virtual	Recommendation	Reading	Game	Visual
Modified Entities/Total Entities	23/42 (55%)	2/5 (40%)	14/17 (82%)	1/2 (50%)	0	3/5 (60%)	0
Defined Dtos/Total Entities	10/42 (24%)	1/5 (20%)	7/17 (41%)	0/2 (0%)	0	0/5 (0%)	0
Modified Relations/Total Relations	6/38 (15%)	4/7 (57%)	7/26 (27%)	0/2 (0%)	0	0/6 (0%)	0
Removed Relations/Total Relations	2/38 (5%)	0/7 (0%)	3/26 (12%)	2/2 (100%)	0	5/6 (83%)	0

Table 3.1: Impact of the refactoring in the domain model

In what concerns the effort to migrate from the monolith into the modular monolith, a considerable

refactoring effort occurred when decomposing the LdoD monolith rich domain model into the different modules. Table 3.1 presents the impact on the domain model of the LdoD Archive, in terms of modified entities and their relations in the different modules. It can be observed a significant amount of modified entities and defined *dtos* from the different modules, where more than 50% of the domain entities had to be modified, even reaching 82% in the *Virtual* module. These two aspects reflect a considerable refactoring effort applied in the domain to separate it into subdomains and adapt the behaviour of the application to access the information through the interfaces and the defined *dtos*. However, while the refactoring values in the *Text* module might seem elevated for a core module with no dependencies, these changes were fairly simple to implement and were mainly focused on the removal of circular dependencies through the use of events.

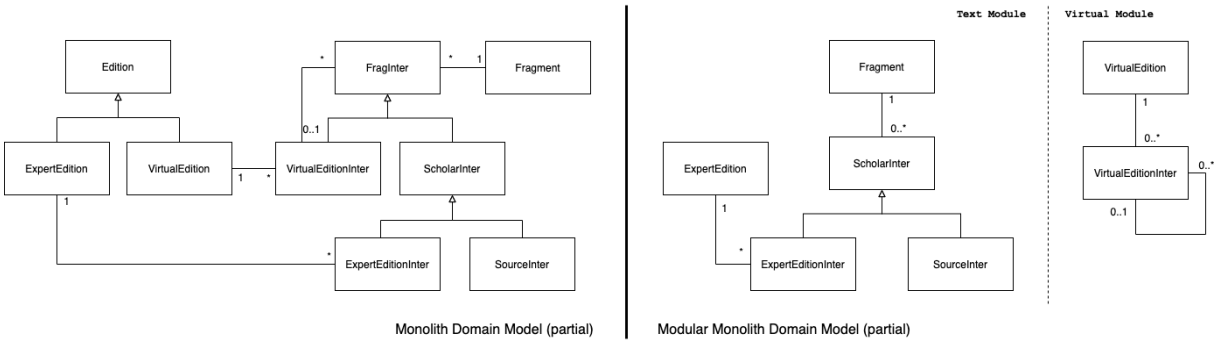

Figure 3.3: Domain model refactoring (partial)

In what concerns the different associations between the domain entities, it is important to address two different refactoring situations that were observed: associations between domain entities that belong to different modules, that had to be modified into invocations through the respective module *requires interface*, and the appropriate *dtos* while maintaining the purpose and behaviour from the association, and removed associations, which correspond to associations that became represented by unique identifiers in the domain entities and the notification interaction between the modules.

A main example of these two refactorings is presented in how the fragment interpretations were modified into separate modules, since originally an abstract class `FragInter` represented all the different types of interpretations, `SourceInter`, `ExpertEditionInter`, and `VirtualEditionInter`, from the application that had to be refactored in order to be separated into the *Text* and *Virtual* module. Figure 3.3 illustrates the refactoring applied, where the `FragInter` entity is removed, its association to the `Fragment` is modified into an association from the `ScholarInter` and another association from the `Fragment` to the `VirtualEditionInter` is removed. Since in this latter association, the `VirtualEditionInter` still depends on the `Fragment`, it will use the interface and an appropriate unique identifier to access the information through inter-module invocations.

3.4 Performance Evaluation

To assess if the migration into the modular architecture was a viable alternative for the LdoD application in terms of performance, a load testing scenario was designed that evaluated the behaviour and performance of the application under a significant usage and compared it to a similar scenario in the monolith architecture. Therefore, allowing to compare the impact on the functionalities from the perspective of an end user.

3.4.1 Specifications

The load testing analysis focused on different functionalities that were chosen accordingly to the number of domain entities they interact with, the number of modules that are interacted with and the amount of processing required. The functionalities chosen were the following:

- Source Listing: Presents the listing of the archive sources, where a source corresponds to a physical source document, from the text fragments. 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. This functionality is done through interactions between the *Front-End* and *Text* modules.
- Fragment Listing: Presents the listing of all text fragments present in archive with additional information of its several interpretations. This functionality is implemented through multiple interactions between the *Front-End* and *Text* modules.
- Interpretation View: Presents an interpretation of a chosen text fragment, displaying any existing tags, annotations and categories and any related editions. This functionality interacts with the *Front-End*, *Virtual*, *User* and *Text* module in order to obtain information of the different domain entities related to this interpretation. It is relevant to chose an interpretation that includes a significant number of domain entities to have multiple inter-module invocations.
- Assisted Ordering: Orders the fragments accordingly to a set of criteria, such as date and text similarity (tf-idf) and was due to the number of interacted modules, *Front-End*, *Text*, *Virtual*, *Recommendation* and *User*, and the amount of processing required. This functionality requires more than 250 000 fragment comparisons, and each comparison requires a significant amount fragment information.

Therefore, for each functionality a load testing test case was designed to simulate 50 sequential user requests to each of the functionalities under two different loads of information in the database, for 100 and 720 fragments respectively. This testing allows to evaluate the latency and throughput of

the different functionalities of the architecture and was done with the application running inside Docker containers on a dedicated machine with an Intel I7 6 cores, 16 GB of RAM and a 1TB of SSD and using JMeter as the load testing tool responsible for this performance scenario.

3.4.2 Optimizations

Before discussing the performance results, it is important to address different optimization strategies used in the different functionalities to improve the performance of the modular application. Throughout the work of the thesis [36], it could be observed some performance deterioration was caused by a significant number of interface calls being used to obtain the information from the evaluated functionalities. The performance could be optimized by reducing the interface usage and increasing the granularity of the functionalities.

To decrease the number of invocations from the interfaces, an optimization was implemented that increased the amount of information stored in the *dto*. This was due to how frequent information from the used domain entities was not being included in their respective *dtos* and so, required another access through the interfaces, which resulted in additional queries to the database for the same domain entities. This initial implementation of the *dtos* resulted in a high penalty on the performance of the *Fragment Listing* and *Source Listing* functionalities.

An additional optimization was also applied to specific functionalities, *Fragment Listing* and *Source Listing*, that further increased the granularity from the functionalities. The optimization consisted of implementing a composed *dto* object to be utilized by these functionalities containing a list of all *FragmentDtos*, where each *FragmentDto* contains all the information required by the functionalities. This effectively allows to reduce the inter-module invocations and database queries and, therefore increasing the performance of these functionalities.

Concerning the *Assisted Ordering*, this functionality performed considerably worse under the modular architecture, providing an unusable experience to the end-users. This was due to a consequence from refactoring the previously described 3.3 associations regarding the `VirtualEditionInter` and `ScholarInter` into inter-module invocations, which proved to be extremely problematic to the performance of this functionality due to a significant number of inter-module invocation. As previously stated, the *Assisted Ordering* requires a significant number of comparisons in their behaviour, where each of the interpretations needs to access this information from the *Text* module multiple times in each comparison. This resulted in over 370 million inter-module invocations for the `ScholarInter` information, which shows a severe consequence of the refactoring from an association. To address this bottleneck, an optimization was made to the `VirtualEditionInterDto` that consisted of caching the information from the associated `ScholarInter` entities, which effectively reduced the number of inter-module invocations and increased the performance of the functionality to acceptable values.

3.4.3 Performance Results

Functionality	Source Listing			Fragment Listing		
Samples	1x50			1x50		
	Monolith	**Modular**	**Variation**	**Monolith**	**Modular**	**Variation**
Avg Time (ms)	17/61	22/629	29.4%/931%	103/350	61/1095	-41%/213%
Min Time (ms)	15/57	20/586	33.3%/928%	84/335	57/1026	-32%/206%
Max Time (ms)	27/74	33/680	22.2%/819%	143/405	94/1172	-34%/190%
Std. Dev.	3.22/2.88	2.90/23.32	-	12.87/13.88	6.08/36.93	-
Throughput (/sec)	55.1/16.2	43.7/1.6	-20.7%/-90.1%	9.6/2.9	16.1/0.91	68%/-69%
Functionality	**Interpretation View**			**Assisted Ordering**		
Samples	1x50			1x50		
	Monolith	**Modular**	**Variation**	**Monolith**	**Modular**	**Variation**
Avg Time (ms)	29/25	24/37	-17%/48%	137/3801	165/12295	20%/224%
Min Time (ms)	26/21	21/34	-19%/62%	124/3738	149/11326	20%/203%
Max Time (ms)	41/39	34/52	-17%/33%	164/3880	195/13355	19%/244%
Std. Dev.	3.14/2.77	2.57/3.91	-	9.47/40.67	10.2/575.84	-
Throughput (/sec)	33.3/38.3	40.9/26.4	23%/-31%	2.3/0.24	2.2/0.08	-4%/-67%

Table 3.2: Performance results for sequentially executing 50 times each functionality for 100 and 720 fragments in the database while running inside Docker containers before optimizing the microservice architecture. Results are separated by / in each cell, for instance, by sequentially executing 50 times the Source Listing functionality in the monolith we observed an average latency of respectively 25, and 151, milliseconds, where there are respectively 100, and 720, fragments in the database (25/151).

Table 3.2 presents a side-by-side comparison from the performance results of the monolith architecture and the optimized modular architecture for respectively 100 and 720 fragments in the database. It can be observed that the performance of the modular architecture remained similar to the monolith under 100 fragments in the database, with a slight performance decrease in the *Source Listing* and *Assisted Ordering* functionalities in terms of latency and throughput. But, on the other hand, the modular monolith was able to obtain slightly outperform the monolith in the *Fragment Listing* and *Interpretation View* under low information. This is due to a faster retrieval of the domain entities using their unique identifiers received from the *dtos* because of fewer round trips to the database.

However, a significant performance degradation occurs in three different functionalities, *Fragment Listing*, *Source Listing*, and *Assisted Ordering*, under 720 fragments in the database, increasing the latency by at least 200% and reaching in the case of the *Source Listing* over 900%. This is due to how increasing the information from the functionalities, increased the number of generated *dtos* for 720 fragments. Therefore, it shows a significant impact of the modularization aspect under huge amounts of information. Note that, the *Interpretation View* functionality barely was affected by increasing the information in the database, since it does not depend on the number of fragments.

Regarding the *Assisted Ordering*, the described optimization proved to be effective since the functionality had a similar performance variation compared to the listing functionalities, despite being the most computationally demanding functionality and having the worst performance impact before the optimization. Additionally, the monolith architecture implemented different caches that stored the vectors

used for the computation to improve the performance, which also proved to be beneficial in the modular context to reduce the inter-module invocations. However, even after the optimization, the drawbacks of inter-module invocations were still noticeable in the performance of the functionality.

In general, the modular monolith architecture provides a beneficial development environment for smaller and business-focused teams that fosters an agile development approach. However, the run-time application of a modular monolith executes inside a single process that is unable to independently scale each of the modules and does not provide failure isolation between them. On the other hand, the microservice architecture addresses both concerns and offers more flexibility to the development environment. In the context of a step-wise migration, the modular monolith presents an impact on the performance and migration cost of the application even before migrating to the microservice. But, at the same time, it provides the groundwork for the microservices, which should be beneficial to the migration. Therefore, evaluating the migration into the microservice architecture is fundamental in understanding the benefits and consequences of a step-wise migration.

4

LdoD Microservice Architecture & Implementation

Contents

One of the core objectives of this dissertation is to implement the LdoD Archive as a microservice application from the described modular architecture and evaluate the migration process from a modular monolith as an intermediary stage. To accomplish this goal, the modular monolith architecture had to be further decomposed and refactored to migrate the modules into services and the modules' interactions into remote invocations. Additionally, it was also necessary to address new challenges introduced with the microservice architecture, like eventual consistency and deployment of the services.

4.1 Microservice Architecture

The microservice architecture shares similar fundamental requirements for modularization as the modular monolith. In the microservice architecture, the services are independent processes with well-established boundaries that focus on specific subdomains and functionalities of the application, meaning that the modules provide the foundation to implement the services. Therefore, by developing a modular monolith, the developers only have to focus on further decomposing the modules into services and on adapting the defined module interactions into appropriate inter-service communication.

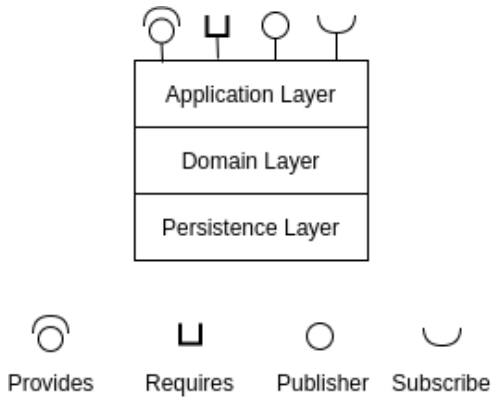

Figure 4.1: Architectural structure of a service

Figure 4.1 presents the architectural structure of the services decomposed from the defined modules. This services are implemented by applying an layer architecture style that is composed of three different layers:

- Application Layer: responsible for the service API, which in this case corresponds to the implementation of the *provides*, *requires*, *publish* and *subscribe* interface using a distributed communication technology.

- Domain Layer: responsible for the business logic from that subdomain that uses the persistence layer to access the database information.

- Persistence Layer: responsible for the database access, in case of persistent information being required for the service.

Overall, implementing modules as services is a straightforward process since the modules represent the bounded context required by microservices and provide the layered structure without rewriting any code. A decomposition also occurs in the interfaces to replace the module interactions with inter-service communication to maintain the behavior of the features in a distributed context.

4.1.1 Inter-service communication

Inter-service communication is a fundamental feature of the microservice architecture that allows the services to exchange information and provide functionalities. The microservice architecture defines two types of inter-process communication: synchronous communication, where the service requests the information and awaits its response, and asynchronous communication, which can undergo different variants depending on the requirements of the architecture but, in general, allows the service to share information without any dependencies. Therefore, a beneficial aspect from the modular monolith is how the uses and notification interaction are an initial fit to, respectively, introduce synchronous and asynchronous communication.

To replace the uses interaction, the proposed solution defined synchronous request/response communication style between the services, since it provides a similar behaviour through the network without requiring a redesign from the behaviour of the application. This communication performs a typical request/response interaction using a technology built around the HTTP protocol, like REST, where a service sends an HTTP request to an API of another service. However, this type of communication is expected to affect performance due to network overheads and the blocking nature of the communication.

On the other hand, asynchronous communication allows adapting the notification interaction by maintaining the communication between the services without creating circular dependencies. This communication implements an event-driven asynchronous communication with a publish-subscribe messaging pattern where the publisher service sends events to a message broker and then the message broker publishes the event to subscribed services. The message broker plays an important role in the architecture by allowing the services to communicate without requiring the components to explicitly be aware of each other while also managing the events and the services.

4.2 LdoD Microservice Refactoring

4.2.1 Additional Modular Refactoring

In the previous chapter, we described the decomposition and refactoring process in detail to develop the LdoD modular architecture and evaluated the impact on the refactoring cost and performance. This allowed us to provide a background on the modular architecture and connect it to the microservice architecture. However, some additional refactoring was required due to non-refactored associations between different subdomain entities and a lack of encapsulation from the modules.

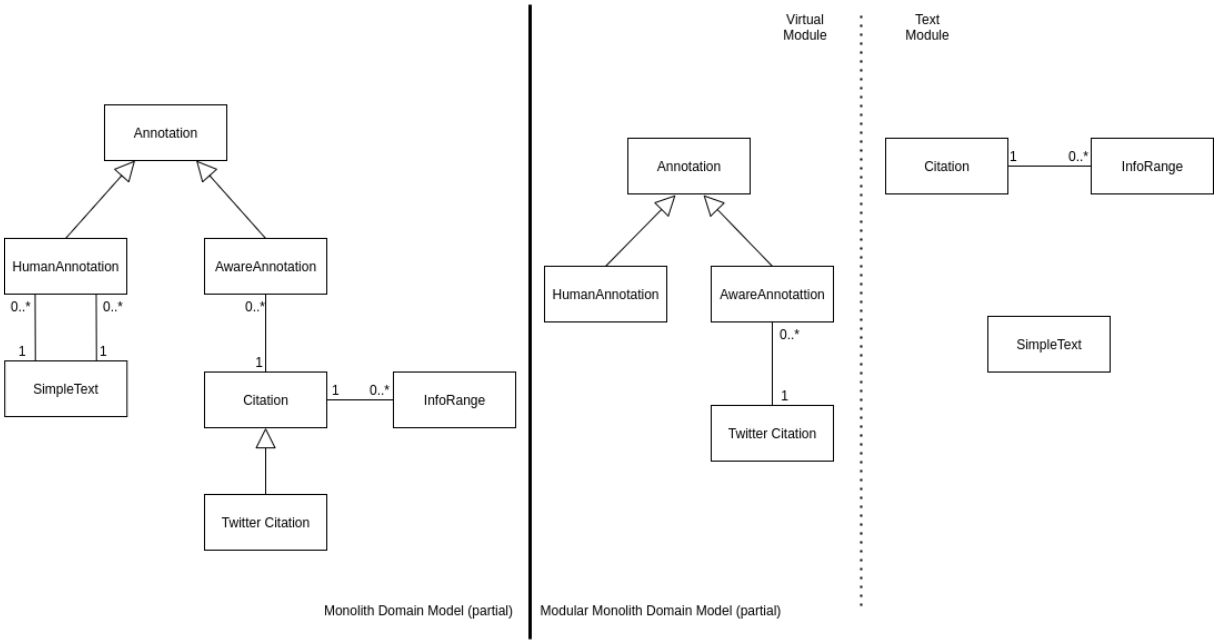

Figure 4.2: Refactoring from the Citation domain model

In the first scenario, it was found associations between the *Text* and *Virtual* subdomains entities that went undetected during the refactoring procedure which needed to be appropriately addressed. Figure 4.2 illustrates the refactoring applied to the associations between the domain entities: `AwareAnnotation`, `TwitterCitation`, and `HumanAnnotation`, that belong to the *Virtual* subdomain and the `Citation` and `SimpleText` domain entities that belong to the *Text* subdomain. These inter-domain associations corresponded to an inheritance and three one-to-many associations that needed to be effectively removed and replaced by indirect associations.

Before removing the inheritance association, we could observe that the `AwareAnnotation` was only associated with the subclass `TwitterCitation`, which allowed to remove the association to the `Citation` domain entity without creating an indirect association. On the other hand, the removal of the inheritance association proved to be slightly more difficult due to an association inherited by the `TwitterCitation`

to the `InfoRange`, which also had to be removed. In this case, the refactoring maintained the association of the `InfoRange` to the `Citation` domain entity and established an indirect association from the `TwitterCitation` to the `Citation` domain entity. This allows `TwitterCitation` to access the info ranges through unique identifiers with inter-module invocations.

The remaining non-refactored associations were found between the `Annotation` and `SimpleText` domain entities, where the subclass `HumanAnnotation` not only used the information from the latter but was also directly accessed from the *Text* module in order to be removed, creating a circular dependency. This refactoring was also achieved through the removal of the association and the implementation of unique identifiers to replace it with a uses interaction. But, in addition, the removal method from the `SimpleText` was also adapted into a notification interaction with a newly defined event that effectively removed the circular dependency.

The second problem was more frequently detected but had a much lesser refactoring effort to correct. It consisted of multiple situations where modules would bypass the interfaces and directly access the information of the service, which compromised the encapsulation of the modules. The lack of encapsulation can be problematic in the development of the modular architecture and on the migration into microservice architecture, since the parallel development by focused teams is affected and introduces additional refactoring effort to the migration. The refactoring solution consisted of simple changes to the module's interfaces by creating new methods in interfaces that accessed the required information and enforcing the uses interaction between the modules.

4.2.2 Inter-service communication

Most of the refactoring effort to achieve a microservice architecture from the modular monolith focuses on the interfaces since the uses and notification interaction were implemented through interfaces and have to be adapted into implementing the appropriate type of communication. Therefore, it requires a redesign of the *provides* and *requires interface* to implement synchronous communication and the *publisher* and *subscriber interface* to introduce the message broker and implement an event-driven communication.

4.2.2.A Synchronous Communication

To achieve synchronous communication, the *provides interface* of each service was implemented as a REST API, that offers public endpoints for fetching the information of the service, which then can be accessed by the *requires interface* through HTTP requests. In more practical terms, the necessary changes made to a *provides interface* involved the mapping of every publicly available method of the interface to an unique URI, capable of being remotely accessed, and providing a serializable response.

In general, this is a simple procedure to implement but, there are factors that can affect the implementation like the method parameters, the response object, and the type of effect on the data from the service. These factors are important to consider when choosing the type of mapping to apply since the different types of requests provided by REST, `GET`, `POST`, `PUT` and `DELETE`, have specific actions (CREATE, READ, UPDATE, DELETE) which have to be compatible with the method. For instance, we can observe in 4.1 a real example of the mapping applied to two different methods of the *Text* service *provides interface* which resulted in two separate types of mapping, `GET` and `POST` respectively. This is due to how each of the methods affects the information in the database, where the `getFragmentByXmlId`() performs a read transaction that retrieves the information from a text fragment given its unique identifier, which corresponds to a READ action provided by the `GET` request. On the other hand, the `createFragment`() method performs a write transaction that creates a new domain instance, corresponding to a CREATE action that is best suited for a `POST` request.

An additional aspect related to the *provides interface* is the compatibility of the *dtos* with the serialization requirement. In this type of communication, the serialization procedure is very important to exchange the information between the services and requires serializable response objects to successfully exchange information throughout the network. Therefore, when migrating into a microservice architecture from a modular monolith, a requirement for serializable *dtos* is introduced into the architecture. Generally, this is a trivial process since the *dtos* mostly carry essential information about a domain entity, however, some situations may require more refactoring effort due to non-serializable information being referred in the *dto* that needs to be refactored. An example of this refactoring could be found in the *dtos* that utilized map data structures using other *dtos* as mapping keys which by default could not be serialized. This introduced a slight refactoring effort to custom serialize the data structure in a way that the behaviour of the functionalities was respected.

Listing 4.1: Example GET and POST methods of the provides interface from Text service

```
1   @GetMapping("/fragment/xmlId/{xmlId}")
2   @Atomic(mode = Atomic.TxMode.READ)
3   public FragmentDto getFragmentByXmlId(@PathVariable(name = "xmlId")
4       String xmlId) {
5           logger.debug("getFragmentByXmlId: " + xmlId);
6           return getFragmentByFragmentXmlId(xmlId).map(FragmentDto::new)
7                   .orElse(null);
8   }
9
10  @PostMapping("/createFragment")
11  @Atomic(mode = Atomic.TxMode.WRITE)
```

```
12      public FragmentDto createFragment(@RequestParam(name = "title") String title,
13          @RequestParam(name = "xmlId") String xmlId) {
14              return new FragmentDto(new Fragment(TextModule.getInstance(),
15                  title, xmlId));
16      }
```

Listing 4.2: Example GET and POST methods of the requires interface from Front-End service

```
17      public FragmentDto getFragmentByXmlId(String xmlId) {
18          return   webClient.build()
19                  .get()
20                  .uri("/fragment/xmlId/" + xmlId)
21                  .retrieve()
22                  .bodyToMono(FragmentDto.class)
23                  .blockOptional().orElse(null);
24      }
25
26      public FragmentDto createFragment(String title, String xmlId) {
27          return webClient.build()
28                  .post()
29                  .uri( uriBuilder -> uriBuilder
30                          .path("/createFragment")
31                          .queryParam("title", title)
32                          .queryParam("xmlId", xmlId)
33                          .build())
34                  .retrieve()
35                  .bodyToMono(FragmentDto.class)
36                  .block();
37      }
```

From the perspective of the service, the *requires interfaces* need to remotely access the URL of a method from a service *provides interface* to request the information. This was achieved through a framework responsible for executing HTTP calls to match URLs of the *provides interface* method that used to be locally called. In this case study, the services utilize the Spring WebClient framework to provide synchronous blocking HTTP communication between the services. Listing 4.2 presents two methods from the *Front-End* service requires interface that matches the REST *provides interface* example methods of the *Text* service. From these methods, we can observe an example on how the *Front-End* service

requests the information or creates a new text fragment through an HTTP request with the appropriate type, URI and parameters that matches the provided methods in the *Text* service. Note that, the behaviour of the uses interaction is respected through a blocking call that guarantees that the information is available upon the request, before continuing the execution.

4.2.2.B Event driven Asynchronous Communication

The notification interaction is implemented as an event-driven asynchronous communication in the LdoD microservice architecture, focusing the refactoring effort on introducing a message broker. The message broker is responsible for managing the events and establishing a publish-subscribe type of communication between the services seamlessly, without creating circular dependencies. To introduce a message broker, the *publish* and *subscribe interfaces* of the services need to implement a `publish` and `listener` method that respectively publishes and listens to events from the message broker.

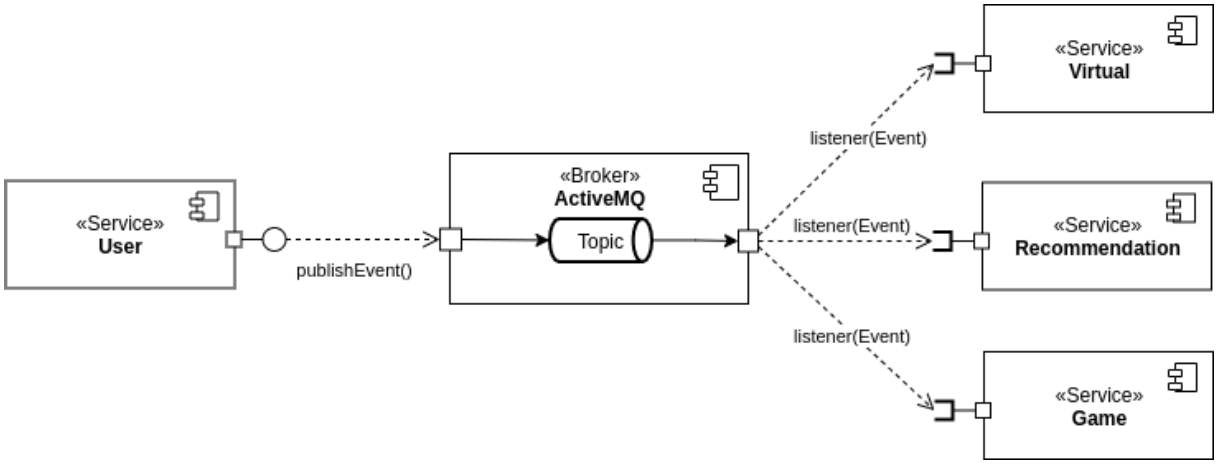

Figure 4.3: Event-driven communication example with an ActiveMQ message broker

Figure 4.3 illustrates how the event-driven asynchronous communication operates between the different services and the message broker in the case of an `User_Remove` event. Upon the removal of a user, the *publish interface* of the *User* service is responsible for sending the appropriate event through the network into the message broker using the `publish` method, where the event is stored in a topic to be eventually published to all subscribed services. The message broker is then responsible for publishing the event to every subscribed service, which has an appropriate *subscriber interface* that listens to the events and processes them accordingly.

Therefore, in the context of the microservice architecture, the event-driven asynchronous communication becomes responsible for maintaining the data consistency between the databases, since each subdomain has an independent database that needs to be kept consistent in write transactions that span through multiple services. The removal of an user is an example of eventual consistency between the

User and *Virtual* services that was introduced with the migration.

Note that, the notification interaction of the modular architecture allows to detect some occurrences that affect the consistency of the information before decentralizing the data into multiple databases. This can be done by analysing the impact on data consistency, once in a distributed context, of the events that are sent in the modular notification interfaces.

4.2.3 Microservice Deployment

The microservice architecture introduces several challenges when addressing the deployment of the services to take full advantage of its decentralized architectural structure, such as service discovery, networking, and, service and resource management that offer a more scalable architecture. In this case study, the Kubernetes technology implements the deployment configuration of the LdoD microservice architecture, allowing to automate the deployment, management, and scaling of the services. In addition, the microservices execute inside Docker containers to facilitate their deployment and improve resource usage.

The deployment of the architecture addresses two different aspects: containerization of the services and the deployment configuration. In what concerns the containerization, this process consists of building a Docker image containing the service's executable JAR and correct version of the JDK through a Dockerfile, capable of running the application inside the container. Then it allows for a fast build and startup of the services, allowing the Kubernetes to easily deploy and shut down the services for efficient management.

On the other hand, the deployment configuration depends heavily on the architecture to deploy, but, in general, it has to configure the deployment of each component, such as the services, databases, and message broker. Kubernetes implements the deployment through configuration files that describe the run-time properties of the components in terms of active replicas, volumes, and network configurations. Listing 4.3 shows an example of a configuration file to deploy the Text service, where we can observe how to set the number of replicas, network ports, and volumes to persist information.

Listing 4.3: Text Deployment example file

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: docker-text
5 spec:
6   selector:
7     matchLabels:
```

```
8         app: docker-text
9     replicas: 1
10    template:
11      metadata:
12        labels:
13          app: docker-text
14      spec:
15        containers:
16          - name: docker-text
17            image: docker-text:v1
18            imagePullPolicy: Never
19            ports:
20              - containerPort: 8081
21            .
22            .
23            .
24            volumeMounts:
25              - name: corpus
26                mountPath: "/opt/ldod/corpus/"
```

Through the previous deployment configuration file, the different services are now capable of being deployed. However, it is still necessary to configure the internal networking of the cluster to allow inter-service communication. This is achieved through a service configuration file for each component that configures the settings required from the Kubernetes cluster to provide the service discovery and route the requests to the respective services using a service local DNS name. Listing 4.4 presents an example service deployment file of the Text service, which configures the internal ports for the node and the internal DNS name to use for the routing of requests.

Listing 4.4: Text Service example file

```
1  kind: Service
2  apiVersion: v1
3  metadata:
4    name: docker-text
5    labels:
6      name: docker-text
7  spec:
8    ports:
```

```
9       - nodePort: 30163
10        port: 8081
11        targetPort: 8081
12        protocol: TCP
13    selector:
14      app: docker-text
15    type: NodePort
```

In terms of effort, the deployment of the LdoD microservice architecture required the configuration of service and deployment configuration files for each of the deployable components, which in this case study corresponds to the services, message broker, and databases. Note that Kubernetes manages the dynamic aspects of a run-time microservice architecture in an autonomous way and reduces the implementation cost to individually address those concerns, thus proving beneficial to the architecture.

4.2.4 Refactoring Cost

	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 4.1: 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

From the perspective of the refactoring effort, most changes were focused on two different aspects of the modular architecture: the *provides* and *requires interface* and the *dtos* of each service. Table 4.1 presents the impact of refactoring the uses interaction into synchronous communication in terms of modified methods in the interfaces of each service. 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 refactoring effort, but most of these changes were small changes, consisting of repetitive mapping procedures as the ones described in 4.2 and 4.1. Thus, making it a simple process to implement.

However, a few incompatibilities were detected that directly affected the refactoring cost, which should be taken into consideration when implementing the interfaces in the modular monolith. An incompatibility was found in methods of the *provides interface* that utilized multiple complex objects as different parameters, which was not supported by the HTTP protocol. This is due to the body of a HTTP request only being parsed once. To address the incompatibility, the solution defined additional serializable *dtos*,

which function as a wrapper for the parameters and allow the parsing of all the parameters at once. In terms of effort, this is a simple solution to implement with low refactoring effort.

A second incompatibility was the transference of non-serializable information between inter-module invocations, which cannot be sent in remote communication. This behavior had a more serious consequence since it could affect areas outside of the scope of the API and increase the refactoring effort. For instance, the Recommendation service utilized non-serializable in the response objects of the methods and introduced additional refactoring in the Front-End service to process the changes. A way to address this problem is by implementing a custom serialization for that type of information that allows writing the object into a JSON format. The proposed solution allows the serialization of that type of information in remote communications and maintains the refactoring in the scope of the API. However, the refactoring effort highly depends on the object and its capability to be serializable.

On the other hand, if serialization cannot be achieved, it is necessary to individually address their usages and adapt how the information is transferred while maintaining the behavior of the functionality. Therefore, note that the quality of the interfaces in the modular monolith affects the refactoring effort and, in addition, it is important to keep the information transferred between the modules serializable to reduce the migration cost.

Additionally, it could also be observed in Table 4.1, the number of modified *dtos* from the back-end and front-end services to, respectively, serializable and deserializable formats. A surprisingly high number of *dtos* from the back-end services were modified into serializable formats, despite their simple structure. This was due to an optimization tactic applied in the modular monolith that implemented the *dtos* as an entry point into obtaining additional domain information related to it, for easier access. However, this is not suited for remote communication due to the JSON serialization requirements not being met. By default, the JSON serialization requires the invocation of every *getter* method 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 requested. This introduces redundant information to the *dtos* and increases the network overheads of the remote invocations. Therefore, to avoid performance degradation, a refactoring was applied that removed this information.

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 the *dtos* from the serialization changes. In addition, the deserializable *dtos* remained responsible for providing the entry point to fetch the additional information and had to be refactored into remote invocations similarly to a *requires interface*. Therefore, despite the high percentages, the changes were simple to implement.

4.3 LdoD Microservice Architecture

In this case study, the LdoD Archive was implemented as a microservice architecture composed by seven different types of back-end services: *Text*, *User*, *Virtual*, *Recommendation*, *Game*, *Search* and *Visual*, that provide the same features as the corresponding modules with well defined REST API endpoints. In addition, the LdoD microservice architecture also presents two client-side front-end services responsible for the visual and game interpretations and a server-side front-end service responsible for the user interface. Note that, the *Front-End* service implements a similar behaviour to an API gateway that processes the user requests and routes them into the back-end services.

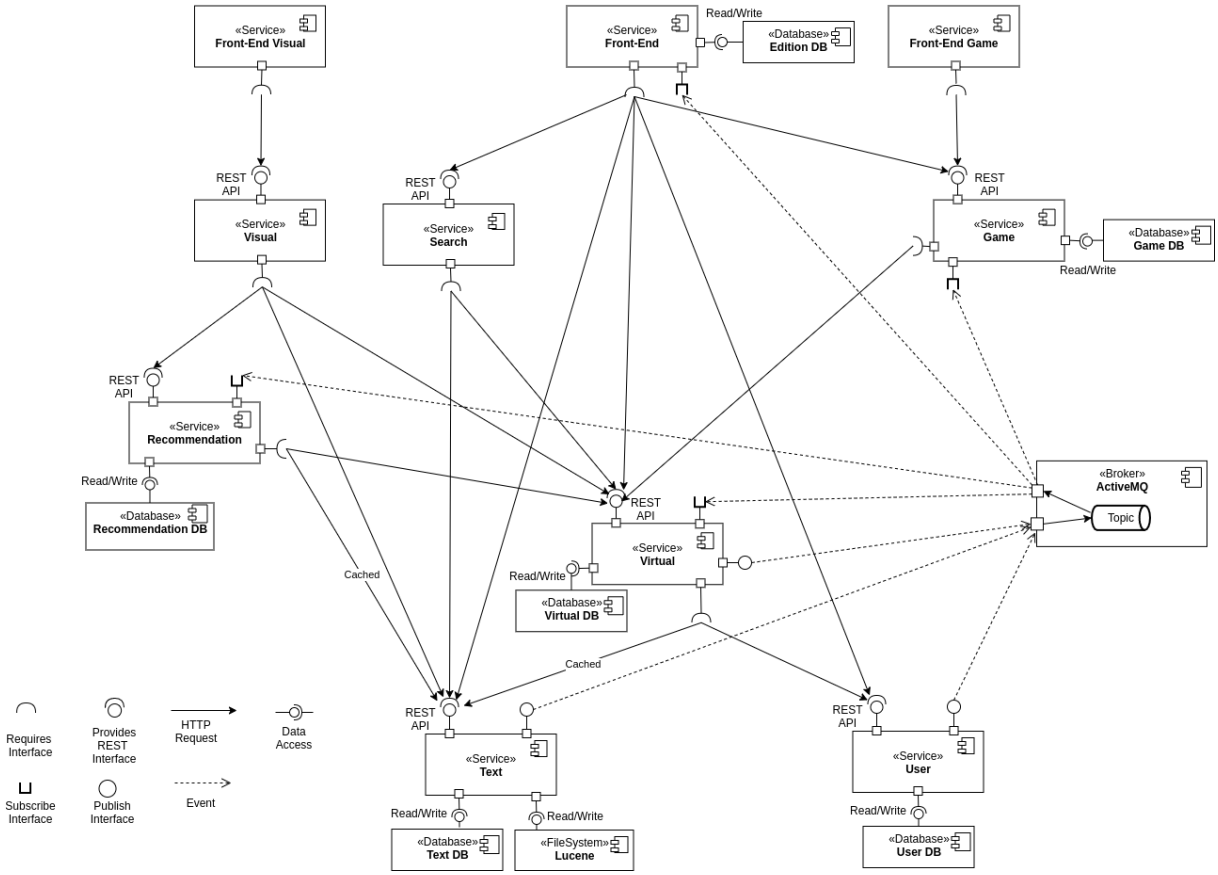

Figure 4.4: LdoD Microservice Architecture Component and Connector View

Figure 4.4 presents a component and connector overview of the LdoD microservice architecture with the respective services, interfaces and inter-service communication. These services apply the synchronous request/responses communication through the REST API and HTTP protocol to exchange the information and maintain the dependencies between the services. An 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, like the core services *Text* and *User*, for

better performance. To achieve scalability, the LdoD microservice architecture is capable of managing and deploying several instances of the services through the Kubernetes deployment technology. Docker containers containerize the services and allow them to be independently deployed. Note that, the Kubernetes technology is beneficial to this architecture since it manages the dynamic aspects of the services like service discovery, networking, resources management and load balancing in a transparent manner.

Focusing on the asynchronous communication, the event-driven asynchronous communication is implemented using an ActiveMQ message topic that manages the publisher services, *Text*, *User* and *Virtual*, the subscriber services, *Virtual*, *Recommendation*, *Game* and *Front-End*, and the respective events in order to maintain the data consistent between them. As we can observe in 4.4, the LdoD microservice architecture implements a decentralized data approach where the information from the subdomains of the *Text*, *User*, *Game*, *Virtual*, *Recommendation* and *Front-End* are stored in separate databases. Additionally, events are used to maintain the consistency between the decoupled databases.

5

Evaluation

Contents

The results presented in this section evaluate different aspects and consequences of migrating the LdoD modular monolith into microservice architecture, as described in section 4. In this context, it is important to analyse the performance of the application since introducing inter-process communication is expected to have a cost on the performance of the application, measure how increasing the service's resources affects the performance and evaluate the impact of eventual consistency on the application.

5.1 Performance Evaluation - Local

In terms of performance, it is important to evaluate the performance of the microservice architecture and compare it to the modular and monolith architecture values discussed in section 3.4 to understand the benefits and drawbacks from the migration. The performance of the monolith and modular architecture was measured through load testings that evaluate the user experience and its performance in terms of latency and throughput. Therefore, a similar evaluation was done in the microservice architecture.

	Text	**User**	**Virtual**	**Recommendation**	**Reading**	**Game**	**Visual**
Source Listing	8/42 (19%)	0/5 (0%)	0/17 (0%)	0/2 (0%)	0	0/5 (0%)	0
Fragment Listing	9/42 (21%)	0/5 (0%)	0/17 (0%)	0/2 (0%)	0	0/5 (0%)	0
Interpretation View	22/42 (52%)	1/5 (20%)	6/17 (35%)	0/2 (0%)	0	0/5 (0%)	0
Assisted Ordering	21/42 (50%)	1/5 (20%)	3/17 (18%)	1/2 (50%)	0	0/5 (0%)	0

Table 5.1: Coverage of the domain entities by each of the functionalities

A load testing was designed for the microservice architecture that evaluated the previously described functionalities: *Source Listing*, *Fragment Listing*, *Interpretation View* and *Assisted Ordering*. These functionalities were initially selected due to how they interacted with a significant number of domain entities and services, as presented in Table 5.1, and the amount of processing required. For each functionality, a load test case was implemented that simulated 50 sequential requests to the functionality under two different loads of information in the database, 100 fragments and 720 fragments respectively. The testing was done using JMeter as the load testing tool with a run-time architecture composed of a single service instance per microservice running inside a Docker container on a dedicated machine with an Intel I7 6 cores, 16 GB of RAM and 1TB SSD.

Table 5.2 presents the performance results of the microservice architecture compared with the modular architecture results. As can be observed in Table 5.2, the microservice architecture had a severe negative impact on the performance, both, in terms of latency and throughput, independently of the functionality and amount of information in the database. The *Fragment Listing* functionality presented an extremely negative experience to the user reaching an average of 32767 ms latency for 720 fragments in the database with a variation of 2893%. In addition, it also experienced latency variation values of 7843% for 100 fragments in the database. Similar differences in performance were observed on the *Source Listing* functionality. These performance values are a consequence of the number of remote

Functionality	Source Listing			Fragment Listing		
Samples	1x50			1x50		
	Modular	Microservice	Variation	Modular	Microservice	Variation
Avg Time (ms)	22/629	982/8384	4364%/1233%	61/1095	4845/32767	7843%/2893%
Min Time (ms)	20/586	896/7876	4380%/1244%	57/1026	4258/30331	7370%/2856%
Max Time (ms)	33/680	1147/9487	3376%/1295%	94/1172	6131/35414	6422%/2922%
Std. Dev.	2.90/23.32	71.17/345.25	-	6.08/36.93	440.74/1455.47	-
Throughput (/sec)	43.7/1.6	1.0/0.12	-98%/-92.5%	16.1/0.91	0.20/0.03	-99%/97%
Functionality	Interpretation View			Assisted Ordering		
Samples	1x50			1x50		
	Modular	Microservice	Variation	Modular	Microservice	Variation
Avg Time (ms)	24/37	200/202	733%/446%	165/12295	859/11013	421%/-10%
Min Time (ms)	21/34	177/188	743%/453%	149/11326	799/10748	436%/-5%
Max Time (ms)	34/52	246/273	624%/425%	195/13355	994/11457	410%/-14%
Std. Dev.	2.57/3.91	13.85/14.71	-	10.2/575.84	44.93/118.44	-
Throughput (/sec)	40.9/26.4	5.0/4.9	-88%/81%	2.2/0.08	0.87/0.09	-61%/12.5%

Table 5.2: Performance results for sequentially executing 50 times each functionality for 100 and 720 fragments in the database while running inside Docker containers. Results are separated by / in each cell, for instance, by sequentially executing 50 times the Source Listing functionality in the modular we observed an average latency of respectively 22, and 629, milliseconds, where there are respectively 100, and 720, fragments in the database (22/629).

invocations necessary to implement the functionality. In each request from a service, the latency value increases due to the network overheads involved in remote communication that affect the performance as the information and number of invocation increases. Therefore, the results show a severe drawback of remote invocations on the performance.

To supplement the performance results, we executed a network analysis that traced an execution from each functionality to measure the number of remote invocations between the services. From these results, we could observe that the *Fragment Listing* and *Source Listing* functionalities executed 4283/28540 and 854/5966 remote invocations, respectively, for 100 and 720 fragments in the database. These results represent a very high network usage for a single use of the functionalities that correlates to the performance degradation experienced in the microservice application. Since, in each invocation, the network overhead time accumulates additional latency that the previous architectures do not experience.

The *Interpretation View* functionality presented a similar negative impact of the inter-service communication on the performance as the previously described functionalities. On the other hand, the variation values were significantly lower and, the functionality was still capable of providing a reasonable experience to the end-user. This is due to, not only a significant difference in the number of remote invocations, but also the amount of information sent in each invocation that reduced the network overhead times. Through the network analysis, we were able to observe that the execution of the functionality only requires 137 invocations between the services. However, the variation values were still considerably high when compared to the previous architectures, meaning that even with a small number of invocations, and transferred information, the functionalities in the microservice architecture perform severely worse.

Surprisingly, the microservice architecture did not impact the performance of the *Assisted Ordering* functionality. This is mainly due to two reasons. First, the use of caches proved to be effective in optimizing the performance of the functionality. In the modular monolith, a cache was already implemented that stored the computational results used in each request, but additionally, two more caches were implemented during the development of the *Recommendation* service that stores domain information related to the fragments. 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 non-cached distributed communication on the overall performance.

Note that, before the optimization, the *Assisted Ordering* functionality presented an unusable experience to the user with latency values per request over an hour and millions of inter-service invocations for 720 fragments in the database. This demonstrates the serious impact of migrating a computationally demanding functionality with fine-grained interactions to a microservice architecture. With 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, consisting of lightweight information. Thus, proving to be an effective optimization tactic in the microservice architecture.

5.2 Microservice Optimization

5.2.1 Refactoring

In the previous analysis, we observed a significant impact from the migration on the performance, where the functionalities provide a bad end-user experience. In addition, we observed a correlation between the performance degradation and the number of remote communication from the functionalities. Therefore, an optimization of the architecture was required to improve the performance and end-user experience. The main goal of the optimization is to reduce the inter-service communication without any major redesign of the functionalities.

From the analysis of the microservice implementation, we could observe that most of the performance penalties were due to a high number of remote fine-grained invocations where the services would request additional domain information related to a *dto*. This implementation resulted in individual requests to obtain the information, increasing the inter-service communication. Therefore, to establish a coarse-grained invocation, the optimization strategy increased the amount of domain information preemptively sent in certain *dtos* that are frequently used together, which allows replacing the remote invocations for local invocations. This allows to reduce the network overheads and improve the performance. The code in A.1 presents an example of the optimization applied to the `ScholarInterDto` where

it can be observed the different methods introduced. By adding these methods, we cached the information and supported the composition of several *dtos* in a single response object. Note that, with this simple change to the *dtos*, the information is now accessible through a single remote invocation where in the previous implementation would require several different requests.

In terms of cost, the optimization required a small refactoring effort in six different *dtos*, `FragmentDto`, `SourceDto`, `ScholarInterDto`, `VirtualEditionDto`, `VirtualEditionInterDto` and `UserDto`. It consisted on the introduction of additional *getter* fields that access the information through the interfaces. However, it is necessary to consider some trade-offs to avoid sending unsolicited information. Therefore, it is crucial to analyse whether the data is frequently used together by the functionalities.

5.2.2 Optimization Results

Functionality	**Source Listing**			**Fragment Listing**		
Optimization	**Before**	**After**	**Variation**	**Before**	**After**	**Variation**
Avg Time (ms)	982/8384	438/4317	-55%/-49%	4845/32767	725/5715	-85%/-83%
Min Time (ms)	896/7876	420/4161	-53%/-47%	4258/30331	701/5621	-84%/-81%
Max Time (ms)	1147/9487	486/4431	-58%/-53%	6131/35414	781/6074	-87%/-83%
Std. Dev.	71.17/345.25	14.16/67.7	-	440.74/1455.47	16.4/66.8	-
Throughput (/sec)	1.0/0.12	2.3/0.23	130%/92%	0.20/0.03	1.4/0.18	600%/500%
Invocations	854/5966	1/1	-99.9%/-99.98%	4283/28540	5/5	-99.9%/-99.98%
Functionality	**Interpretation View**			**Assisted Ordering**		
Optimization	**Before**	**After**	**Variation**	**Before**	**After**	**Variation**
Avg Time (ms)	200/202	155/140	-23%/-31%	859/11013	886/11444	3%/4%
Min Time (ms)	177/188	138/129	-22%/-31%	799/10748	766/10780	-4%/0.3%
Max Time (ms)	246/273	184/176	-25%/-36%	994/11457	1434/13049	44%/14%
Std. Dev.	13.85/14.71	10.2/8.7	-	44.93/118.44	114.39/576.47	-
Throughput (/sec)	5/4.9	6.4/7.1	28%/45%	0.87/0.09	0.85/0.085	-2%/-6%
Invocations	137/137	48/48	-65%/-65%	524/4388	509/3622	-3%/-18%

Table 5.3: Performance results and number of remote invocations for sequentially executing 50 times each functionality for 100 and 720 fragments in the database while running inside Docker containers before and after optimizing the microservice architecture. Results are separated by / in each cell, for instance, by sequentially executing 50 times the Source Listing functionality in the microservice before the optimization, we observed an average latency of respectively 982, and 8384, milliseconds, where there are respectively 100, and 720, fragments in the database (982/8384).

Table 5.3 presents the results of the optimization of the microservice architecture in terms of both performance and network usage. In terms of network usage, the optimization was successful in reducing the remote invocation required for each functionality, while simultaneously improving the performance of the functionality. It can 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 invocation respectively, independently of the number of fragments in the database. The performance results of both of these functionalities confirmed that a major factor of the performance degradation observed in this migration resulted from the high number of inter-service communication. The *Source Listing* functionality, through a single remote invocation, obtained a performance improvement of 55%/49% for latency

and consequently a 130%/92% throughput increase, while the *Fragment Listing* performed significantly better with a latency increase of 85%/83% and a throughput increase of 600%/500%. Therefore, these results show that despite the increased amount of information sent in each of these requests, the performance of a functionality benefits from coarse-grained interactions between the services.

In the *Interpretation View*, it could also be observed a slight improvement between 22% to 28% of the performance with a slight reduction from the network usage from 137 to 48 remote invocations. Before the optimization, this functionality already had a low latency and network usage compared to the other functionalities, however the optimization still proved to be beneficial to reduce the overheads in a functionality with low information. On the other hand, the *Assisted Ordering* functionality remained with similar performance values despite the 18% reduction of the remote invocations number, but this is easily explained due to the low impact of the remote invocations on the extensive computational requirements of the functionality.

However, while the different optimizations proved beneficial to the performance, the final results are still considerably worse than in the previous architectures, independently of the amount of information and number of remote invocations. This is the consequence of additional network overheads that are introduced with remote invocations. Note that, a major performance bottleneck of the remote invocations came from the need to serialize and deserialize the *dtos* on each invocation, due to how it introduces additional latency that becomes more noticeable as the information increases. For instance, we measured and observed that even in a coarse-grained communication, the serialization and deserialization time of the *Source Listing* functionality corresponded to 82% of the average latency of the functionality, reaching a serialization time over 3000 ms and a deserialization time over 500 ms. This is a considerable impact on the performance of the functionality, which by itself is already superior to the latency of the functionality in the modular monolith.

5.3 Performance Evaluation - Cloud

A key aspect of the microservice architecture that motivates the migration is the scaling benefits of the application, which can be hard for the monolith application to provide as the user base and complexity of the application grows. As previously stated, the LdoD microservice application is composed by independent services responsible for specific features, that allows to achieve a more scalable approach of the application. This is due to how the services can be horizontally scaled by focusing the resources and increasing the number of replicas of the services responsible for the demanding functionalities for better performance. Therefore, it is important to evaluate how beneficial it is to scale the different services in terms of performance.

In this section, the performance of different features of the LdoD microservice architecture are eval-

uated in terms of latency and throughput, by running different run-time deployments in a cloud environment under different network workloads.

5.3.1 Specifications

The main goal is to compare the performance between two different run-time deployments with a different number of instances deployed, to understand how services use resources and how they perform under a heavy load. In addition, the results also provide an insight on the scalabity of the architecture.

For the evaluation, it is necessary to choose which functionalities are going to be tested to decide which services to replicate. In this scenario, three functionalities from the previous performance testing were chosen to be evaluated: *Source Listing*, *Fragment Listing* and *Interpretation View*, due to their domain coverage and suitability for a heavy load scenario. Note that the Assisted Ordering is a computationally demanding functionality that requires a high amount of available resources to be scalable and would not be adequate for this performance scenario due to a lack of available resources.

The two run-time versions of the architecture are:

- Single-Instance: The single instance deployment version of the architecture is composed of a single instance from each type of service in order to provide the performance values for a basic deployment that will be used as the reference base values of the application running in a cloud environment.
- Multi-Instance: The multi-instance deployment version of the architecture is composed of 5 instances from each of the following services, *Text*, *Virtual* and *Front-End* services, and a single instance of the remaining services. This allows us to evaluate how increasing the resources of specific services affected the performance of the application.

Therefore, for this performance evaluation scenario, two load testing scenarios were designed to measure the performance under different network workloads: a sequential workload and a concurrent workload. The sequential workload simulates a normal usage by a user, implemented with the same load test settings already 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 simultaneously invoking the functionalities and allows to evaluate how the different run-time versions perform 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.

5.3.2 Performance Results

Table 5.4 presents the results of the test cases. It can be observed some benefits and drawbacks of the different run-time versions of the architecture under different usages of the application. In the concurrent

Functionality	Source Listing					
Samples	1x50			50x1		
	Single	Multi	Variation	Single	Multi	Variation
Avg Time (ms)	1768/21806	2075/20329	17.4%/-7%	71791/1021199	24232/297820	-66.2%/-70.8%
Min Time (ms)	1531/18004	1944/17839	27%/-1%	67074/1007652	17745/219864	-73.5%/-78.2%
Max Time (ms)	2452/28400	3256/33605	32.8%/18%	73926/1028616	27893/400253	-62.3%/-61.1%
Std. Dev.	124.0/3061.41	179/2946.77	-	1909.2/6687	2975/64587	-
Throughput (/sec)	0.6/0.046	0.48/0.05	-20%/9%	0.68/0.05	1.8/0.125	164.71%/150%
Functionality	Fragment Listing					
Samples	1x50			50x1		
	Single	Multi	Variation	Single	Multi	Variation
Avg Time (ms)	2848/24830	3573/26210	26%/6%	122648/1654885	40912/441837	-66.6%/-73.3%
Min Time (ms)	2660/22586	3202/23616	20%/5%	99085/1627447	33263/346403	-66.4%/-78.7%
Max Time (ms)	3182/26388	3893/40782	22%/55%	127078/1661867	46205/574864	-63.6%/-65.4%
Std. Dev.	97.18/843.5	175.6/2390.2	-	5044/9183	3815/83013	-
Throughput (/sec)	0.35/0.04	0.28/0.038	-20%/-5%	0.4/0.03	1.1/0.09	175%/200%
Functionality	Interpretation View					
Samples	1x50			50x1		
	Single	Multi	Variation	Single	Multi	Variation
Avg Time (ms)	313/313	372/399	18.9%/27.5%	5425/6829	2673/3693	-50.7%/-45.9%
Min Time (ms)	273/272	321/325	17.6%/19.5%	2501/3723	1048/2597	-58.1%/-30.2%
Max Time (ms)	397/701	491/682	23.7%/-2.7%	7291/8116	3852/4343	-47.2%/-46.5%
Std. Dev.	29.18/62	38.08/51.5	-	1607/1340	846.5/453	-
Throughput (/sec)	3.2/3.2	2.7/2.5	-15.6%/21.9%	6.8/6.2	12.9/11.5	89.7%/85.5%

Table 5.4: Performance results for sequentially executing 50 times each functionality and for 50 users concurrently executing each functionality for 100 and 720 fragments in the database while deployed in Google Kubernetes Engine cluster. Results are separated by / in each cell, for instance, by sequentially executing 50 times the Source Listing functionality we observed an average latency of respectively 1768, and 26259, milliseconds, where there are respectively 100, and 720, fragments in the database (1768/26259).

workload, there was 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, with an 89% throughput increase. 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 can 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* 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, which introduces a slight latency that becomes more noticeable as the latency of a normal request decreases.

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 should benefit the performance, specially in a cloud deployment. First, a redesign of functionalities that require large amounts of information 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.

5.4 Data consistency

The decentralization of the application data introduces challenges to the architecture concerning its data consistency, due to the transactional behaviour between the different databases, since 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 it is not the last in the sequence; (4) write transactions that require notification to other services of the changes.

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 most of the transactional behaviour between the services is 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.

An example of this behaviour occurs when removing the `ClassificationGame` in the *Game* service, which requires the removal of related tags in the *Virtual* service. In case of failures, this could result

in inconsistent information between the services since a tag would be removed in *Virtual*, while the game would persist due to a later failure in the transaction. By removing the tags at the end, the write transaction in the *Virtual* service must be successful for the transaction in Game to finish, thus preserving the consistency even in case of failures.

In general, read transactions do not affect the consistency since most of the information in a service does not depend on other services and, the local transactions can provide the ACID properties to the database. However, when reading the 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 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, eventual consistency is an important aspect introduced into the LdoD Archive that needs to be evaluated in terms of the effects on the information and its impact on the functionalities.

5.4.1 Eventual Consistency

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 services. Most of the events are related to the removal of a domain entity. For instance, upon removing a text 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. Therefore, it is important to understand the types of events sent between the services, how they affect the data consistency and how eventual states affect the different functionalities.

Event Type	Publisher	Subscriber	Impacted Domain Entities	Frequency	Event Trigger	Impact
Fragment-Remove	Text	Virtual	VirtualEditionInter	Low	VEInter-Remove, Tag-Remove	High
ScholarInter-Remove	Text	Virtual	VirtualEditionInter	Low	VEInter-Remove, Tag-Remove	High
SimpleText-Remove	Text	Virtual	Annotation	Low	Tag-Remove	High
User-Remove	User	Virtual	Member, SelectedBy, Tag and Annotation	Low	Tag-Remove	High
		Game	ClassificationGame	Low	-	High
		Recommendation	RecommendationWeights	Low	-	High
VirtualEdition-Remove	Virtual	Game	ClassificationGame	Medium	-	Low
		Front-End	-	Medium	-	Low
VirtualEdition-Update	Virtual	Front-End	-	Medium	-	Low
VirtualEditionInter-Remove	Virtual	Game	ClassificationGame	Medium	-	Low
Tag-Remove	Virtual	Game	ClassificationGame	Medium	-	Low
Virtual-Export	Virtual	Game	-	Medium	-	Low

Table 5.5: Different event types with the corresponding publisher/subscriber services, impacted information, frequency of occurrence and the impact on the overall information

Table 5.5 presents an overview of the different type of events of the LdoD application with the respective publisher/subscriber services and impact on data consistency. There are nine types of events that affect different services and domain entities. Each event has different degrees of frequency and impact

on the consistency of the impacted domain entities. Most events relate to the removal of domain entities that require write transactions that span across the publisher and subscriber services. Therefore, having an impact on the consistency. Additionally, the events can trigger a chain of events that further spans the write transactions throughout the services.

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 of the information, affecting three different databases and multiple domain entities from the *Virtual*, *Game* and *Recommendation* services. 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.

Therefore, the events have a higher impact if they trigger a large number of changes, like the removal events of the *Text* and *User* services 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 they are almost like immutable entities. On the other hand, it is very uncommon to delete the archive users. So, the *Text* and *User* service information remains static throughout the execution of the application, only removed under exceptional contexts with administrator privileges. Thus, making the occurrence of these events rare, which reduces the complexity of managing the consistency.

Under a regular 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 an end-user can interact with its virtual edition, removing some of their entities. On a positive note, the impact of the inconsistency focuses on the *Game* service, having a low effect on the overall application. Note that the *Front-End* is also affected by the `VirtualEdition-Remove` but, the impact focuses on the session information, which may only fail in the execution of a functionality. Since it may happen that the target of the functionality does not exist anymore.

5.4.2 Functionalities

In another perspective, it is also important to analyse how eventual consistency affects the application from the perspective of an end-user. By evaluating the behaviour of the functionalities, we can determine how they performed under inconsistent states. Three functionalities were chosen for this evaluation, considering the domain entities the functionalities interact with and the dependencies on different events and databases:

- Interpretation View: It requires the interaction between the *Text*, *User* and *Virtual* services to retrieve and display the information of an interpretation. This functionality presents the categories used by a user, which can be inconsistent due to an occurrence of the `User_Remove` event.

- Virtual Edition Listing: presents the interpretations of a virtual edition with their categories and provides a descriptive information of the edition. The information accessed by the functionality may be inconsistent due to the occurrence of events related to the domain entities, `Fragment`, `ScholarInter` and `User`.

- Game Listing: presents the active games of a virtual edition and the virtual interpretation used in the game, the game participants and the creator. The information accessed by this functionality may be inconsistent due to the occurrence of events related to *Virtual*, *Text* and *User* services.

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 is a behaviour that results from 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 serious consequences 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 eventual consistency from the described asynchronous communication was an efficient alternative to distributed transactions in the microservice architecture. Since, at the end of the migration, most functionalities were not affected by inconsistent states and preserved the consistency of the information. Therefore, the eventual consistency had an overall low impact on the architecture. On the other hand, it is mainly due to the simple transactional behaviour of the architecture that does not span across different services and can apply ACID properties through local transactions in most situations. In addition, the immutability of the information had a significant role in reducing the impact of inconsistencies. So, it is relevant to note that the results can vary accordingly to the architecture and the mutability.

5.4.3 Caches

To improve the performance of the application, several caches were implemented in the monolith and microservice architecture. However, these caches might become inconsistent throughout the use of the application and affect the consistency of the functionalities. Therefore, it is important to evaluate the consequences of inconsistencies in the caches and their impact on the application.

Table 5.6 presents the different implemented caches in the LdoD microservice application with the

Cache	Service	Cached Information	Impact	Purpose
ScholarInter	Text	Database Access Index	Low	Faster access speed
Fragment	Text	Database Access Index	Low	Faster access speed
VirtualEdition	Virtual	Database Access Index	Low	Faster access speed
VirtualEditionInter	Virtual	Database Access Index	Low	Faster access speed
ScholarInterDto	Virtual	Domain Information	Low	Reduce network overhead
FragmentDto	Recommendation	Domain Information	Low	Reduce network overhead
Category	Recommendation	Domain Information	Medium	Reduce network overhead
TF-IDF	Recommendation	TF-IDF values	Low	Reduce network overhead
CommonTF-IDF	Recommendation	TF-IDF Common Terms	Low	Reduce network overhead
FragmentVectors	Recommendation	Vectors	Medium	Reduce computation
StoredVector	Recommendation	Vectors	Medium	Reduce computation

Table 5.6: Different event types with the corresponding subscriber and publisher services and the domain entities that are modified from it

respective service and stored information. It can be observed three types of caches with different impacts on the consistency and purposes of the application: caches that implement database access indexes for faster access retrieval of information from the databases, caches that save frequently requested information from a service to reduce the communication between services and, caches that save the computed vectors utilized in the functionalities provided by the *Recommendation* service, like the *Assisted Ordering*, that allow to reduce the computationally demanding operations of each execution.

The impact of the caches inconsistency depends on the type of cached information and its mutability. We observed that the database access index caches and caches that store domain information of the *Text* service, `FragmentDto` and `ScholarInterDto`, were not affected by inconsistent information, thus resulting in a low impact. This was due to the immutability of those caches since the database access indexes and domain entities cached, `Fragment` and `ScholarInter`, are mainly immutable, as already discussed. So there are no inconsistencies in them.

On the other hand, inconsistent information in some of the caches from the *Recommendation* service presented a larger impact on the functionalities. There were two types of caches in this service with different purposes. In the first type, the caches stored domain information to reduce the network overheads of the computationally demanding functionalities, such as the *Assisted Ordering*. The impact of this type of cache also correlates to the mutability of the stored information. We observed that the `Category` cache was the only cache that could become inconsistent due to its mutability and affect the functionalities. As for the second type, these caches store computed information to reduce the computations required for the functionality. However, these caches can also become inconsistent as the information changes, affecting the final results of the functionalities.

Under a regular context, the overall impact of inconsistency in the caches is low, focusing on the `Category` and `StoredVectors` caches since they cache mutable information that affects the functionalities. Therefore, we can conclude that the impact of the caches also correlates to the mutability of the information. However, it can also be stated that as the mutability of the information increases, the worse

the effects on the application. Therefore, it is important to consider which information can be cached and how the inconsistent information can affect the application.

5.5 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 migra-

tion into a modular and microservice architecture. In the modular monolith, the performance degradation was related to the amount of information sent between the modules through the *dtos* but, 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 access to the database through the unique identifiers. On the other hand, the number of remote invocations also severely affected the performance of the microservice architecture. Therefore, these two factors should be avoided when designing the microservice architecture.

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. In what concerns the data consistency, the LdoD microservice architecture presented a simple case of eventual consistency, which did not require complex solutions like SAGAs or two-phase commit like protocols. 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.

5.5.1 Threats to Validity

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.

In the microservice architecture, the technology used for the inter-service communication can also affect the migration results since different technologies may face different incompatibilities to migrate the interfaces of the modules. But, in general, we made some observations from the migration which apply to most microservice architectures.

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.

6

Conclusion

Contents

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.

6.1 Conclusions

In this case study, the modular monolith can be used as an intermediate artifact that facilitated the migration process of the LdoD Archive monolith into a microservice application by addressing aspects like the functional decomposition and encapsulation of the application, while providing a more agile software development environment before tackling the challenges of a microservice architecture. In addition, the migration results also made visible the necessary refactoring and redesign of the functionalities, in the first step of the migration, to help handle the concerns associated with the overall migration to a microservice architecture, such as the type of inter-service communication, eventual consistency and the performance impact and optimizations.

The migration of a modular monolith into a microservice architecture revealed the impact on the migration effort and on the performance. Therefore, the migration effort to further decompose the modules and implement the inter-service communication depends on the quality and compatibility of the interfaces with the type of distributed communication and technology utilized for the inter-service communication. A serious consequence of the migration was the large impact on performance associated with the latency of the implementation of the uses interactions as synchronous remote invocations, which augments the latency associated with the execution of each one of the end-user requests.

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 depends 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. Note that, as observed in the literature review, the benefits and drawbacks of the migration vary accordingly to the architecture and the requirements of the application, therefore through this work we provided an additional case study that evaluated a mainly synchronous microservice architecture while addressing the efforts and the optimizations applied.

6.2 Future Work

While the main objectives from this thesis were accomplished, the LdoD microservice architecture can still be improved and expanded upon, to either improve the quality of the implementation and reduce the

limitations of the architecture.

The performance of the microservice architecture is a crucial limitation of the current migration, since, despite the optimizations and understanding of the faced performance bottlenecks, the difference in performance between the architectures is still considerably high. One way to address this concern is to fine-tune the functionalities with the correct amount of information by either limiting the information sent in that functionality or by caching the most utilized information in the front-end services to diminish the impact of remote invocations. Another possible approach would be to experiment with a different serialization protocol like the protocol buffers described in [28] that are compatible with the REST API and offer better performance for the serialization which may improve the latency of the functionalities.

Regarding the architecture, a fully asynchronous approach of the architecture might be interesting to be explored in future work to increase the service's availability and evaluate the effects on the scalability of the application. To explore this approach, a significant redesign of the functionalities has to be done, which may also include changes to its behaviour, for instance, by redesigning the user interface to not present all information at once.

Bibliography

[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, accessed on 2021-10-13. [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] V. Velepucha and P. Flores, “Monoliths to microservices-migration problems and challenges: A sms,” in *2021 Second International Conference on Information Systems and Software Technologies (ICI2ST)*. IEEE, 2021, pp. 135–142.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[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] 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

[22] 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.

[23] 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.

[24] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, “Serverless computing: An investigation of factors influencing microservice performance,” in *2018 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, 2018, pp. 159–169.

[25] 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.

[26] 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.

[27] B. Shafabakhsh, R. Lagerström, and S. Hacks, “Evaluating the impact of inter process communication in microservice architectures.” in *QuASoQ@ APSEC*, 2020, pp. 55–63.

[28] P. Johansson, “Efficient communication with microservices,” Master’s thesis, Umeå University, June 2017.

[29] M. Jayasinghe, J. Chathurangani, G. Kuruppu, P. Tennage, and S. Perera, “An analysis of throughput and latency behaviours under microservice decomposition,” in *International Conference on Web Engineering*. Springer, 2020, pp. 53–69.

[30] A. Furda, C. Fidge, O. Zimmermann, W. Kelly, and A. Barros, “Migrating enterprise legacy source code to microservices: on multitenancy, statefulness, and data consistency,” *IEEE Software*, vol. 35, no. 3, pp. 63–72, 2017.

[31] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, and M. Kalinowski, “Data management in microservices: State of the practice, challenges, and research directions,” *arXiv preprint arXiv:2103.00170*, 2021.

[32] C. K. Rudrabhatla, “Comparison of event choreography and orchestration techniques in microservice architecture,” *International Journal of Advanced Computer Science and Applications*, vol. 9, no. 8, pp. 18–22, 2018.

[33] R. Laigner, M. Kalinowski, P. Diniz, L. Barros, C. Cassino, M. Lemos, D. Arruda, S. Lifschitz, and Y. Zhou, “From a monolithic big data system to a microservices event-driven architecture,” in *2020*

46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE, 2020, pp. 213–220.

[34] P. Kookarinrat and Y. Temtanapat, "Design and implementation of a decentralized message bus for microservices," in *2016 13th International Joint Conference on Computer Science and Software Engineering (JCSSE)*. IEEE, 2016, pp. 1–6.

[35] A. Lesniak, R. Laigner, and Y. Zhou, "Enforcing consistency in microservice architectures through event-based constraints," in *Proceedings of the 15th ACM International Conference on Distributed and Event-based Systems*, 2021, pp. 180–183.

[36] N. Gonçalves, "A product family for digital humanities repositories," Master's thesis, Instituto Superior Técnico, October 2019.

[37] K. J. Lienberherr, "Formulations and benefits of the law of demeter," *SIGPLAN Not.*, vol. 24, no. 3, pp. 67–78, Mar. 1989. [Online]. Available: http://doi.acm.org/10.1145/66083.66089

A

Code and Schemas

Listing A.1: ScholarInterDto after the optimization

```
1   public class ScholarInterDto {
2
3       private final TextProvidesInterface textProvidesInterface =
4           new TextProvidesInterface();
5       .
6       .
7       .
8       public ScholarInterDto(ScholarInter scholarInter) {
9           setXmlId(scholarInter.getXmlId());
10          .
11          .
12          .
13      }
```

```
14          .
15          .
16          .
17          public LdoDDateDto getLdoDDate() {
18              return this.textProvidesInterface
19                  .getScholarInterDate(this.xmlId);
20          }
21
22          public HeteronymDto getHeteronym() {
23              return this.textProvidesInterface
24                  .getScholarInterHeteronym(this.xmlId);
25          }
26
27          public ExpertEditionDto getExpertEdition() {
28              return this.textProvidesInterface
29                  .getScholarInterExpertEdition(this.xmlId);
30          }
31
32          public SourceDto getSourceDto() {
33              if (isSourceInter) {
34                  return new TextProvidesInterface()
35                      .getSourceOfSourceInter(this.xmlId);
36              }
37              return null;
38          }
39
40          public List<AnnexNoteDto> getSortedAnnexNote() {
41                 return this.textProvidesInterface
42                      .getScholarInterSortedAnnexNotes(this.xmlId);
43          }
44      }
```

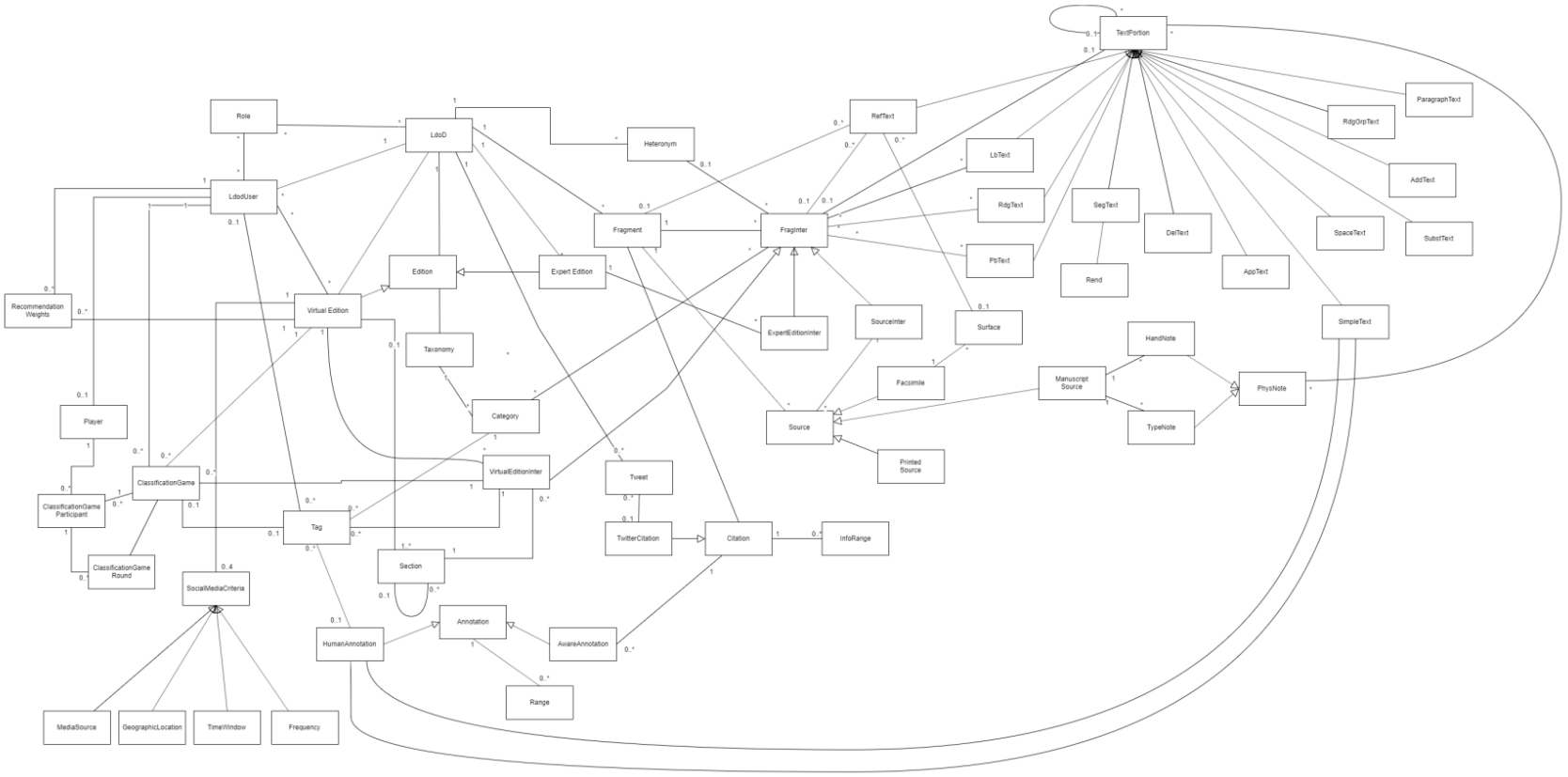

Figure A.1: LdoD monolith domain model