Performance Evaluation 164 (2024) 102411

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Stepwise migration of a monolith to a microservice architecture: Performance and migration effort evaluation

Diogo Faustino [a], Nuno Gonçalves [a], Manuel Portela [b], António Rito Silva [a,*]

[a] INESC-ID, Instituto Superior Técnico, University of Lisbon, Rua Alves Redol 9, Lisboa, 1000-029, Portugal
[b] CLP - Centre for Portuguese Literature, University of Coimbra, Largo da Porta Férrea, Coimbra, 3004-530, Portugal

ARTICLE INFO

Keywords:
Domain-driven design
Modular architecture
Migration effort
Performance evaluation
Microservices

ABSTRACT

Due to scalability requirements and the split of large software development projects into small agile teams, there is a current trend toward the migration of monolith systems to the microservice architecture. However, the split of the monolith into microservices, its encapsulation through well-defined interfaces, and the introduction of inter-microservice communication add a cost in terms of performance. In this paper, we describe a case study of the migration of a monolith to a microservice architecture, where a modular monolith architecture is used as an intermediate step. The impact on migration effort and performance is measured for both steps. Current state-of-the-art analyses the migration of monolith systems to a microservice architecture, but we observed that migration effort and performance issues are already significant in the migration to a modular monolith. Therefore, a clear distinction is established for each of the steps, which may inform software architects on the planning of the migration of monolith systems. In particular, we consider the trade-offs of doing all the migration process or just migrating to a modular monolith.

1. Introduction

Microservices have become increasingly adopted [1] as the architecture of business systems, because they promote the split of the domain model into consistent pieces that are owned by small agile development teams and facilitate scalability [2,3]. Therefore, monoliths are being migrated to the microservice architecture.

Research has been done on the comparison of the performance quality between a monolith system and its correspondent implementation using a microservice architecture, but these results are sometimes contradictory, e.g. [4,5], address different characteristics of a microservices system, e.g. [6,7], or are evaluated using simple systems, e.g. [8]. Furthermore, in these studies, it was not necessary to redesign the monolith functionalities to migrate to the microservice architecture.

Due to the conflicting results, which may be related to the complexity and variability (e.g., technological aspects) of microservice systems, we use a stepwise migration process for a monolith system that separates the concerns of modularization and distribution. Therefore, an intermediate modular monolith architecture is used to support the modularity design in the absence of distribution. By applying this approach to a large monolith system, we can analyze the impact of each aspect, modularization and distribution, on performance and refactoring effort.

We observed that the modularization phase requires most of the refactoring effort, while the impact on latency is already significant in the absence of remote invocations between modules. However, the performance optimization tactics used in the

[*] Corresponding author.
E-mail address: rito.silva@tecnico.ulisboa.pt (A. Rito Silva).

https://doi.org/10.1016/j.peva.2024.102411
Received 3 March 2023; Received in revised form 3 December 2023; Accepted 7 March 2024
Available online 12 March 2024

modular context are less disruptive to the overall design of the system, such as the handling of inconsistent data and the need to introduce asynchronous behavior. These results may guide developers in their migration decisions.

Overall, this paper addresses the following research questions:

- What is the effort associated with the migration of a monolith to a microservice architecture?
- What is the impact on performance of the migration of a monolith to a microservice architecture?
- How is the answer to the previous research questions contextualized by a two-step migration process? How does each of the steps impact the migration effort and performance qualities?

The next section presents related work, and Section 3 describes the stepwise migration process of a monolith to a microservice architecture, using a modular monolith as an intermediate step. Then, in Section 4 the migration process is evaluated in terms of migration cost and performance, and the results are discussed in Section 5, which highlights the migration trade-offs and the lessons learned. Section 6 contains the conclusions.

2. Related work

There are several challenges when migrating from monolith systems to a microservice architecture [9,10], such as the development effort to migrate the monolith and the performance impacts, and we can find in the literature the description of the migration of some large monoliths [11–13], but they do not address the migration effort and performance impact. Therefore, more case studies are necessary that describe the migration efforts and the architectural trade-offs, besides the advantages and drawbacks, of the final product. Due to technological complexity, it is beneficial that these studies follow a separation of concerns approach, so that the analysis can be done according to levels of complexity.

On the other hand, there is work on the impact that migration of a monolith to a microservice architecture has on performance, which is done in toy systems.

Ueda et al. [5] compare the performance of microservices with the monolith architecture to conclude that the performance gap increases with the granularity of microservices, where the monolith performs better. Villamizar et al. [4] show different results, concluding that in some situations the performance is better in the microservices context and that it reduces the infrastructure costs, but the request time increases in microservices due to the gateway overhead. Al-Debagy and Martinek [7] conclude that they have similar performance values for average loads, and the monolith performs better for small loads. In a second scenario, they found that the monolith has better throughput but similar latency when the system is stressed in terms of simultaneous requests. Bjørndal et al. [14] benchmark a library system that has four use cases and considers synchronous and asynchronous relations between microservices. They observe that the 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 to systems that are closer to the kind of systems used by companies. Guamán et al. [15] designed and implemented a multi-stage architectural migration into a microservice architecture and compared the performance of the monolith and the intermediary stages with the microservice architecture. They concluded that the microservice architecture has worse latency. Flygare et al. [16] found that in their case study, the monolith performed better in terms of latency and throughput while consuming less resources than the microservice architecture. Additionally, they also observed some throughput benefits of running the microservice in a cluster instead of running on 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 [17], the characteristics of the running environment, whether it uses containers or virtual machines [6], the particular technology used, such as different microservice discovery technologies [7], or other aspects of microservices deployment [8]. A major aspect of performance is the type of inter-microservice communication and the technology used. Hong et al. [18] compared the performance of synchronous and asynchronous communication and concluded that the asynchronous approach offered a more stable overall performance but a lower response request performance. Fernandes et al. [19] presented similar results, with asynchronous communication outperforming REST communication in performance and data loss prevention in a large data context. Shafabakhsh et al. [20] leverage on Fernandes et al. [19] research and conclude that there is a benefit of synchronous communication under small loads.

These studies compare the performance of monolith and microservice systems in a wide range of technological variations. Interestingly, in most cases, a monolith's functionality is implemented by a single microservice, without any inter-microservice interactions. Therefore, due to the large number of variations in these studies, which make the comparison of results difficult, the research would benefit from a separation of concerns approach that would manage the complexity.

Haywood [21] introduces the modular monolith and compares it with the microservice architecture. The modular monolith decomposes the monolith into a set of modules that do not share any data and interact through well-defined interfaces. In reality, the modular monolith can be seen as an intermediate step for the migration of a monolith to a microservice architecture, where modularity is achieved but independent scalability of modules is not possible.

In this work, we describe the stepwise migration of a large monolith system to a microservice, using the modular monolith as an intermediate step, to do systematic analysis of migration impact on development effort and performance.

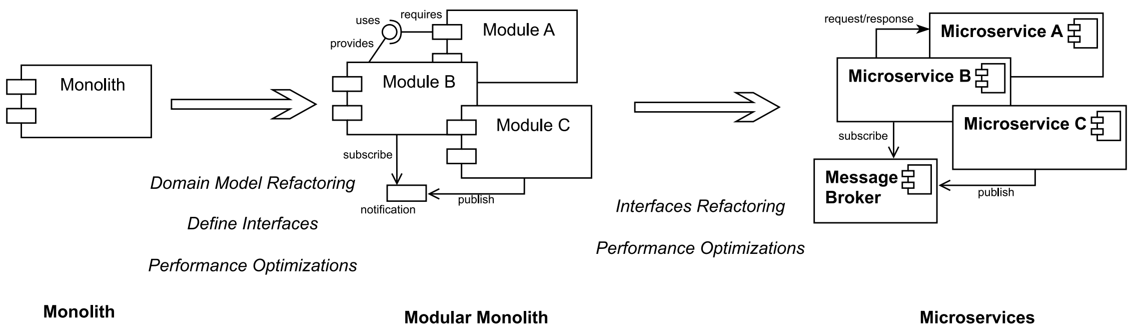

Fig. 1. Two-step migration.

3. Migration

The stepwise migration process of a monolith to a microservice architecture leverages on the previous work on modular monolith [21] and on the patterns of enterprise application [22]. It supports a separation of concerns strategy to the analysis of the problem at hand. Therefore, the description of the process emphasizes the aspects necessary to evaluate the impact of migration on performance and the development effort.

Fig. 1 depicts the two-step migration process, in which a monolith is migrated to a set of modules, with well-defined interfaces (uses and notification), and these modules are migrated to microservices using distribution communication mechanisms, such as request/response and message broker. Next, we describe the activities that are performed in each step.

3.1. Modular monolith

Modularization of a monolith requires the decomposition of its domain model into different modules, so that each of the modules does not interdepend on a shared data repository [21], and the interactions are done through the module interfaces. The domain model represents the persistent domain entities of the application, usually defined using an Object-Relational Mapper (ORM) [22].

3.1.1. Domain model refactoring and module interfaces

To partition a domain model, four aspects have to be considered:

- *Remove inter-module relationships*. The relationships between entities belonging to different modules are refactored:
  - *Multiplicity relationships*. Relationships are replaced by unique identifiers of the entities involved. To restrict dependencies between modules, the identifier is defined in only one of the modules, resulting in uses and notification interactions between modules (see below).
  - *Inheritance relationships*. Inheritance relationships are flattened so that each inheritance hierarchy belongs to a single module.
- *Split domain entities*. To achieve cohesion, it may be necessary to divide domain entities between modules.
- *Define interfaces*: After the domain model is split, the interactions between these smaller domain models should be carried out through the module interfaces. The modular monolith defines two types of interaction between modules:
  - *Uses interaction*. In the uses interaction, a module requires (uses) services from another (used). In this type of interaction, the module that is used has a *provides interface* that matches a *requires interface* defined in the uses module. The uses module depends on the used module. To guarantee the encapsulation of the domain entities of the modules, the provided interface returns Data Transfer Objects (*DTO*) [22].
  - *Notification interaction*. A module sends notifications (events) of its modifications to any module that has subscribed to them. The module that notifies has a *publish interface* while the module that is notified has a *subscribe interface*. The former does not depend on the latter.
- *Redefine methods*. Methods need to be redefined when they interact through the module interfaces.

Fig. 2 exemplifies the partition of the domain model on the left into two modules on the right. Entities `E4` and `E5` have a one-to-many relationship that is replaced in entity `E4` of `Module B` by a unique identifier, `idE5`. This reflects the decision that `Module B` depends on `Module A`, but the latter does not know about the former and therefore does not have any reference to `E4`. These dependencies are illustrated in the module interfaces. `Module A` provides an interface that is required by `Module B`, while `Module A` publishes events that are subscribed by `Module B`. For instance, `Module B` uses `Module A` to obtain a *DTO* of an `E5` entity by sending its unique identifier, `idE5`. On the other hand, `Module B` subscribes to the event that notifies about the deletion of `E5` entities. This occurs because `Module A` does not depend on `Module B` and `Module B` subscribes to know whether

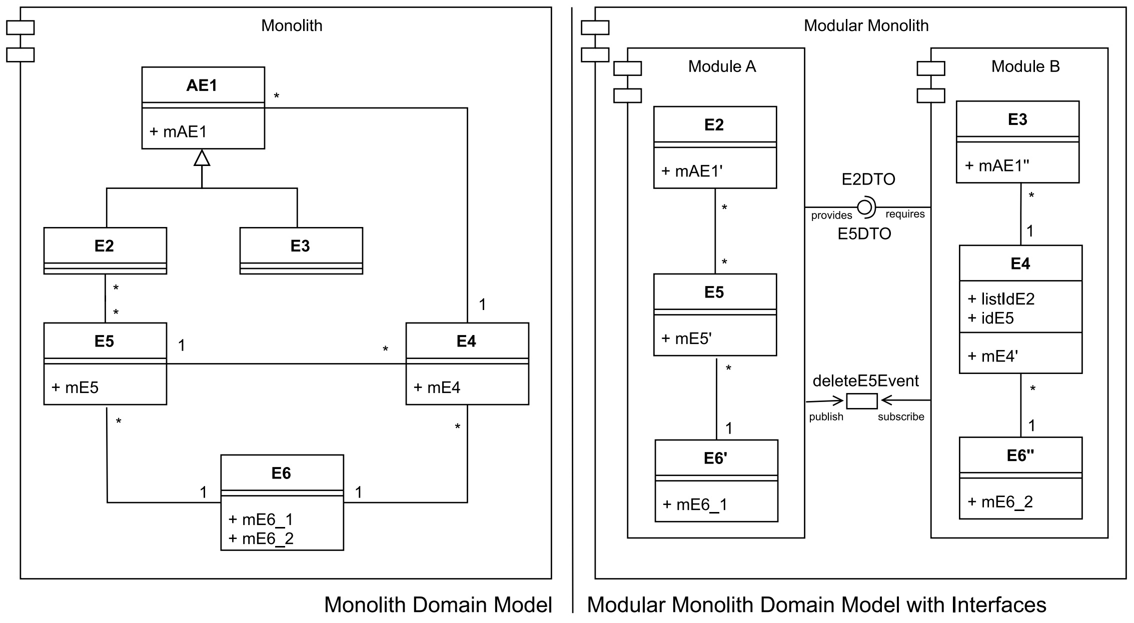

Fig. 2. Domain model refactoring.

its information about `Module A` is up-to-date. Methods `mE4` and `mE5` are refactored because they cannot directly interact with the entities in the other module. The interactions should be done through the module interfaces. It may also be necessary to split classes between modules, as illustrated in Fig. 2, where `E6` is split into classes, one for each module. In this case, the methods do not need to be refactored because they only access entities in its own module, `mE6_1` interacts with `E5` and `mE6_2` interacts with `E4`. The particular case of inheritance relationships that are cut by modularization is dealt with by flattening the inheritance. In the example, `AE1` in Fig. 2 is replaced by the refactoring of its method in `E2` and `E3`, which belong to different modules. The multiplicity relationship is also refactored: in `Module A` it is also removed and replaced by the `listIdE2` field, which contains a list of `E2` identifiers; in `Module B` the multiplicity is refactored to an association with `E3`. Method `mAE1` is refactored in both entities.

3.1.2. Performance optimizations

The definition of several modules in the modular monolith raises performance problems associated with the inter-module communication and the need for data transformations to isolate each module's domain model. During domain model refactoring, it is necessary to evaluate the performance of the migrated functionalities. If they show poor performance, the number of invocations between modules has to be reduced. This is achieved by redesigning the inter-module interactions into coarse-grained invocations. Note that monoliths are implemented using fine-grained object-oriented interactions.

The following performance optimizations are applied:

- Associate a database access index to the unique identifiers of domain entities exported to other modules, due to the frequent invocations to obtain a *DTO* object of the domain entity, given its unique identifier.
- Whenever a *DTO* is generated, by default, it is loaded with the values for all its attributes, which work as a cache, to reduce further inter-module invocations to obtain each one of its values. This introduces memory overhead because some of the attribute values may not be necessary in the context of that particular interaction. However, in general, after systematically applying this tactic, the performance of the system improves.
- In some cases, after performance evaluation, it is necessary to have larger *DTO* objects, which, in addition to the attribute values, also contain the information associated with several interrelated domain entities. For instance, a *DTO* containing a list of *DTO*s may reduce the number of inter-module invocations when a module is interacting with a collection of domain entities in another module.

Actually, these optimizations are common in software architecture. For instance, the use of an identity field for entities that are known among address spaces, and the decision between lazy vs. eager load when copying information between address spaces [22]. Java Persistence API[1] is an example of a technology that applies this type of patterns.

[1] https://docs.oracle.com/javaee/6/tutorial/doc/bnbpz.html.

3.2. Microservices

Migrating a modular monolith to a microservice architecture requires refactoring of inter-module interfaces to enable distributed communication technology. Afterwards, performance needs to be revisited due to the added impact on communication latency.

3.2.1. Interfaces refactoring

It is only necessary to redesign inter-module interfaces, Fig. 2:

- *Uses interaction*. Redesign the provides/requires interface to implement synchronous distribution communication, like REST.
- *Notification interaction*. Redesign the publisher/subscriber interfaces to use a message broker communication mechanism, like ActiveMQ.

3.2.2. Performance optimization

Similarly to the modular monolith, the definition of several microservices and the introduction of synchronous communication in the architecture raise performance problems associated with the number of remote invocations per functionality and the amount of information transferred in *DTOs*. This is due to the extra latency resulting from remote communication, which is significantly higher than in the modular monolith. The following optimizations are applied:

- Implementation of caches in microservices for faster information retrieval speed and to reduce the number of remote invocations for faster performance. Some of these caches are created upon microservice creation and the data is preserved until the microservices stop their execution, this is very effective when the chosen data remains immutable throughout the execution;
- When necessary, *DTOs* containing the information of several interrelated domain entities have to be defined to reduce the number of remote invocations, though it increases the amount of information sent in a single request.

4. Evaluation

A large monolith was migrated to a microservice architecture to evaluate the migration effort, the impact on the performance of the stepwise migration process, and answer research questions.

The *LdoD Archive* monolith[2] is a digital archive for the digital humanities. It offers features such as searching, browsing, and viewing original text fragments, different variations of text fragments, as well as different editions of a book. It also provides a way for users to create their own (virtual) editions of the book, in which they can add, remove and order fragments as they wish [23,24]. Other features include a recommendation feature, which defines a proximity measure between different fragments according to a set of criteria [25], and a game feature, which implements a serious game in which users tag text fragments [26].

The monolith application is implemented in Java using the Spring-Boot framework and an Object-Relational Mapper (ORM) to manage the domain model's persistence. The domain model has 71 domain entities and 81 bidirectional relations between domain entities, which resulted in a strongly coupled domain model. In total, the monolith has 25.862 lines of Java code, 20.039 lines of JSP code, and 7.721 lines of JavaScript code.

The modular monolith implements the same features as the monolith. These features are implemented as a set of modules that apply the uses and notification interactions to preserve the dependencies between features: `Text`, which represents the fragments and their expert editions; `User`, which provides users registration, authentication, and access authorization; `Virtual`, which allows the definitions of virtual editions by end users and requires the `Text` and `User` modules; `Recommendation`, which calculates similarity distances between the fragments, given a set of criteria, such as date and tf–idf (Term Frequency Inverse Document Frequency) [27], between text fragments and uses the `Virtual` and `Text` modules; and `Game`, which implements a tagging serious game to classify the fragments in a virtual edition and uses the `Virtual` module.

The code repository where the evaluation was done is publicly available on GitHub.[3] The *master* branch contains the monolith code (in folder *editon-ldod*), and branches *modular-monolith* (folders with the module name in the top directory) and *microservices* (folders with the microservice name in the *backend* directory) contain the result of the respective migrations.

To reproduce the results of the migration effort, it is necessary to analyze the folders *domain* and *dto* of each branch.

To reproduce the performance results, it is necessary to run each one of the artifacts and execute the JMeter[4] tests that are in the *master* branch, inside *editon-ldod/jmeter* folder. The tests require that the database be loaded with the XML encodings for the 720 fragments, which will be provided upon request to the authors.

4.1. Migration effort

The migration effort from monolith to modular monolith depends on the number of relationships between the new modules' domain entities and on the number of interfaces between modules.

[2] https://ldod.uc.pt/?lang=en.
[3] https://github.com/socialsoftware/edition.
[4] https://jmeter.apache.org/.

Table 1
Impact of the refactoring in the domain model.

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

Table 2
Impact of the refactoring in the provides interfaces (Pro) and requires interfaces (Req) and *DTOs* from each of the microservices in terms of refactored methods and *DTOs*. *Text* and *User* modules do not have required interfaces because they do not use other modules.

	Text	User	Virtual		Recommendation		Game	
	Pro	Pro	Pro	Req	Pro	Req	Pro	Req
Modified/Total methods	91/109 (83%)	43/47 (91%)	152/155 (98%)	6/6 (100%)	6/6 (100%)	7/7 (100%)	14/14 (100%)	9/9 (100%)
New methods	16	2	2	18	1	0	0	2
Modified/Total DTOs	6/15 (40%)	2/3 (67%)	9/14 (64%)		3/11 (27%)		2/6 (33%)	

Table 1 presents the impact, in the different modules, of the refactorings of the domain model in terms of the domain entities and their relationships. It can be seen that there is a large impact on the domain model, where more than 50% of the domain entities had to be changed, and in the case of the `Virtual` module, this value reaches 82%. This corresponds to a considerable effort associated with the migration from the monolith to the modular monolith. A similar situation can be observed in the percentage of *DTOs* that had to be defined, given the total number of domain entities. This has an impact on the changes in the modules that use these *DTOs*, and in particular in the domain entities that interacted, before refactorization, with the original domain entities.

It can also be observed in Table 1 that there is a smaller impact of the refactoring on the relations, because only the relations between domain entities belonging to different modules are affected, while more domain entities can be affected due to indirect relationships.

Migrating from a modular monolith to a microservice architecture focuses on two different major aspects: the provides/requires interface and the publish/subscribe interface. Table 2 presents the impact of refactoring the provides/requires interface into synchronous communication using a REST API. It can be observed that there are significantly high percentages of modified methods in which at least 83% of the methods had to be mapped. This may seem like a significant migration effort, but these changes were simple to implement and consisted of repetitive mapping procedures. Similar results were obtained for the publish/subscribe interface migration.

4.2. Performance

To evaluate the impact that the new architecture has on the performance of the system, we performed performance tests on the application features described above using JMeter. Testing was carried out with the application running inside Docker containers on a dedicated machine with an Intel I7 6 cores, 16 GB of RAM, and 1TB of SSD.

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

The *Source Listing* functionality[5] presents the list of archive sources, where a source corresponds to a physical source document. When using this functionality, the end user obtains all the information about each of the 754 sources, such as the date, dimensions, and type of ink used in the document. Implementing this functionality is done through an interaction between the front-end and *Text* module/microservice.

Fragment Listing functionality[6] is implemented in the modular monolith and microservice as an interaction between the front-end and *Text* to present the list of all fragments. There are 720 text fragments in the archive. For each fragment, information is presented about several interpretations, and each fragment can have 2 to 7 interpretations. The front-end performs a total of 5 invocations of the *Text* module.

The *Interpretation View* functionality[7] presents an interpretation of a text fragment. Its implementation in modular monolith and microservices requires several interactions between the front-end and the *Text*, *User* and *Virtual* modules/microservices. To perform the performance test, an interpretation of a virtual edition that includes several different types of domain entities was chosen. It was not possible to optimize the implementation of functionality using a coarse-grained invocation. On the other hand, the amount of information retrieved in each inter-module interaction is small.

[5] https://ldod.uc.pt/source/list/?lang=pt_EN.
[6] https://ldod.uc.pt/fragments/?lang=pt_EN.
[7] https://ldod.uc.pt/fragments/fragment/Fr494/inter/Fr494_WIT_ED_VIRT_LdoD-Arquivo_1/?lang=pt_EN.

Table 3
Coverage of the domain entities by each of the functionalities.

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

Table 4
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 monolith, we observed an average latency of respectively 17, and 61, milliseconds, where there are respectively 100, and 720, fragments in the database (17/61). We ran the experiment 50 times to obtain an average, and we used 100 and 720 fragments to observe the effect of the amount of data being transferred.

		Monolith	Modular	Microservices	Variation Monolith Modular	Variation Modular Microservices	Variation Monolith Microservices
Source Listing	Latency (ms)	17/61	22/629	438/4317	29%/931%	1891%/586%	2476%/6977%
	Throughput	55.1/16.2	43.7/1.6	2.3/0.23	−21%/−90%	−95%/−86%	−96%/−99%
Fragment Listing	Latency (ms)	103/350	61/1095	725/5715	−41%/213%	1089%/422%	604%/1533%
	Throughput	9.6/2.9	16.1/0.91	1.4/0.18	68%/−69%	−91%/−80%	−85%/−94%
Interpretation View	Latency (ms)	29/25	24/37	155/140	−17%/48%	546%/278%	434%/460%
	Throughput	33.3/38.3	40.9/26.4	6.4/7.1	23%/−31%	−84%/−73%	−81%/−81%
Assisted Ordering	Latency (ms)	137/3801	165/12 295	886/11 444	20%/223%	437%/−7%	547%/201%
	Throughput	2.3/0.24	2.2/0.08	0.85/0.085	−4%/−67%	−61%/6%	−63%/−65%

The *Assisted Ordering* functionality[8] orders fragments according to a set of criteria, such as date and text similarity (tf–idf). This ordering requires more than 250,000 fragment comparisons (considering, for instance, a virtual edition with 720 fragments, we get $720 * 719/2 = 258\,840$ comparisons between fragments), and each comparison requires information about the fragment for up to 4 criteria. For its implementation, the front-end accesses 4 back-end modules/microservices: *Text*, *User*, *Virtual* and *Recommendation*. Because this functionality repeatedly interacts with the same set of data, the information about the fragments is cached in the monolith implementation to improve performance. The modular monolith system utilizes the same cache, while the microservice architecture has implemented two extra caches to address the performance issues that arise from remote calls for the same data set.

Table 3 presents the number of domain entity types for each module that a functionality accesses.

For each functionality, a test case was designed to compare the performance of all three architectures. Each test case run simulates a user that sequentially submits 50 requests after a first request to warm up the caches. The test cases are run for two different loads of the database, for 100 and 720 fragments, respectively. The results can be comparable in a single machine because the requests are sequential. The results are shown in Table 4.

In terms of performance comparison between the monolith and the modular monolith, it can be observed that the impact on the performance of migration to the modular monolith increases with the number of fragments. In 3 of the 4 functionalities, the difference in performance between the monolith and the modular monolith is much higher for 720 fragments than for 100 fragments. This is due to the number of *DTOs* that are generated. Note that for the *Interpretation View* functionality, there is no difference because it is a functionality that does not depend on the number of fragments in the database. In the *Assisted Ordering* functionality, the difference is not as significant as expected, considering the number of operations. This is due to the cache used in both implementations, which reduces the need to transfer *DTOs* between modules, although this is the most computationally demanding functionality.

Additionally, and a little bit surprising, for 100 fragments, some functionalities have slightly better performance in the modular monolith. This is due to the fact that one of the modular monolith optimizations is the association of an index with the unique identifiers that are sent to the other modules. The monolith is implemented following an object-oriented approach, where objects are retrieved by searching in an object graph. Optimization in the modular monolith allows faster retrieval of the domain entities, given their unique identifiers, because there are fewer round trips to the database. This was verified through some test cases in which this optimization was removed, and the results showed a huge decrease in the performance of the modular monolith. Note that, since the *Interpretation View* functionality does not depend on the number of fragments, it performed better than the monolith even for tests with 720 fragments.

In terms of performance comparison between the modular monolith and the microservice architecture, it can be observed that the microservice architecture has a severe negative impact on performance, both in terms of latency and throughput, independently of the functionality and information in the database. The *Fragment Listing* functionality experienced an extreme negative user experience, reaching a latency average of 5715 ms with a variation of 422% for 720 fragments in the database, while also experiencing latency variation values of 1089% for 100 fragments in the database. Similar latency variation values were also

[8] The functionality requires authentication.

observed in the *Source Listing* functionality. These performance values are a consequence of the number of remote invocations necessary to implement the functionality. In each request/response of a remote invocation, the latency values increase due to network overhead, such as serialization/deserialization times, which severely affect the performance as the amount of information increases, showing a severe drawback of remote invocations.

The *Interpretation View* functionality also had performance drawbacks due to its communication between microservices, which had a negative impact on performance. However, the variation was significantly lower than the functionalities previously described and still capable of providing a good experience to the end-user. This is due not only to a significant difference in the number of remote invocations but also to the fact that the amount of information sent in the invocation is relatively small, which reduces the network overhead.

Surprisingly, the microservice architecture did not impact the performance of the *Assisted Ordering* functionality (Table 4, −7% for the variation for modular microservices when 720 fragments). This is mainly due to two reasons. First, the functionality is computationally demanding, which reduces the impact of distributed communication on overall performance. Second, the use of caches proved to be effective in optimizing the performance of the functionality. In the modular monolith, a cache that stored the computational results used in each request was already implemented. Additionally, two more caches were implemented during the development of the *Recommendation* microservice that store domain information related to fragments. Because functionality interacts repeatedly with the same set of data, these caches reduce the number of remote invocations necessary to perform functionality.

During the migration of the modular monolith to the microservice architecture, performance degradation was initially identified due to the high number of fine-grained invocations between the microservices. Therefore, to have coarse-grained interactions, it was necessary to increase the amount of domain information preemptively sent in certain *DTOs*, which were frequently used together. This replaced remote invocations with local invocations, reducing network overhead and improving performance. By adding additional information to *DTOs*, we were able to cache the information and support the composition of several *DTOs* in a single object. With this simple change, the information is now accessible through a single invocation, where before it would require several inter-microservice invocations. However, it is necessary to consider some trade-offs when choosing the information to send to avoid sending information that is not necessary, which will introduce additional overhead. It is necessary to analyze whether the data is frequently used together by the functionalities.

Overall, when comparing the performance of the monolith with the microservices system, and although all optimizations, the differences were significant for all functionalities, varying the latency between 201% and 6977%. This is the consequence of the introduction of additional network overhead caused by remote invocations. Note that a major performance bottleneck of remote invocations came from the need to serialize and deserialize *DTOs* on each invocation because it introduces additional latency that becomes more noticeable as the amount of information increases. For instance, we measured and observed that even in coarse-grained communication, the serialization/deserialization time of the *Source Listing* functionality corresponded to 82% of the average latency of the functionality, reaching a serialization time of over 3000 ms and a deserialization time of 560 ms. This has a considerable impact on the performance of the functionality, which further increases the latency of the functionality in the modular monolith.

4.3. Scalability

So far, we have analyzed the impact of migration on performance but not on the scalability aspects associated with the possibility of independent scalability in microservice architecture.

Therefore, another testing setup was necessary. This testing was carried out with the microservices being deployed in a Google Kubernetes Engine cluster with 8 nodes, 16 vCPU and 32 GB of memory. To allow a comparison with a deployment without independent scalability, two run-time architectures were considered: a single instance multi-instance deployments. The former is composed of a single instance of each microservice, and the latter is composed of five instances of each of the `Text` and `Virtual` microservices, and a single instance of the remaining microservices. There are also five instances of the front-end that send requests. This allows us to evaluate how increasing the resources of specific microservices affects performance. Additionally, to allow comparison with the previous experiment, two different workloads were considered: a sequential workload and a concurrent workload. The former is similar to the local experiment and the latter simulates 50 different users concurrently invoking the functionalities.

Table 5 presents the results of the test cases. It can be observed that there are some benefits and drawbacks to the different run-time versions of the architecture under different usages of the application. In terms of concurrent workload, there is a significant increase in the throughput of running multiple instances of specific microservices 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 an increase in throughput between 150% and 200%, which is a significant improvement in scalability. This is because the deployment of more instances increases the use of resources and supports the parallel processing of requests.

Similar performance benefits could also be observed in the *Interpretation View* functionality of parallel processing, with an increase in 89% throughput. 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, such as *Source and Fragment Listing*, and with fine-grained invocations, such as *Interpretation View*.

Table 5
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 the Google Kubernetes Engine cluster. The 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 21 806, milliseconds, where there are respectively 100, and 720, fragments in the database (1768/21 806).

Functionality	Source Listing					
Samples	1 × 50			50 × 1		
	Single	Multi	Variation	Single	Multi	Variation
Avg time (ms)	1768/21 806	2075/20 329	17.4%/−7%	71 791/1 021 199	24 232/297 820	−66.2%/−70.8%
Min time (ms)	1531/18 004	1944/17 839	27%/−1%	67 074/1 007 652	17 745/219 864	−73.5%/−78.2%
Max time (ms)	2452/28 400	3256/33 605	32.8%/18%	73 926/1 028 616	27 893/400 253	−62.3%/−61.1%
Std. Dev.	124.0/3061.41	179/2946.77	–	1909.2/6687	2975/64 587	–
Throughput (/s)	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	1 × 50			50 × 1		
	Single	Multi	Variation	Single	Multi	Variation
Avg time (ms)	2848/24 830	3573/26 210	26%/6%	122 648/1 654 885	40 912/441 837	−66.6%/−73.3%
Min time (ms)	2660/22 586	3202/23 616	20%/5%	99 085/1 627 447	33 263/346 403	−66.4%/−78.7%
Max time (ms)	3182/26 388	3893/40 782	22%/55%	127 078/1 661 867	46 205/574 864	−63.6%/−65.4%
Std. Dev.	97.18/843.5	175.6/2390.2	–	5044/9183	3815/83 013	–
Throughput (/s)	0.35/0.04	0.28/0.038	−20%/−5%	0.4/0.03	1.1/0.09	175%/200%
Functionality	Interpretation View					
Samples	1 × 50			50 × 1		
	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 (/s)	3.2/3.2	2.7/2.5	−15.6%/21.9%	6.8/6.2	12.9/11.5	89.7%/85.5%

Focusing on the sequential workload, the measured results were very similar to those obtained in the previous performance evaluation, but with much more latency due to its deployment in 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 microservices 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 such as *Fragment Listing* 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 are not suitable for large payloads of information or fine-grained invocations.

In the *LdoD Archive* microservice architecture, two improvements can still be implemented that will benefit performance in a cloud deployment. First, a redesign of functionalities such as *Fragment Listing* and *Source Listing* functionalities to implement a pagination pattern,[9] which divides large response data sets into manageable and easy-to-transmit pages. This allows us to reduce network overhead and improve 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. 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 performance. The modular monolith and microservice architecture share the modularization requirements to address the decomposition of the domain into modules/microservices, while also addressing the granularity of the interactions between the domain entities for performance reasons. Therefore, the modular monolith offers a beneficial basis for achieving a microservice architecture. Manages migration through the stepwise definition of modularization through well-encapsulated modules, which serve as the foundation of microservices and break the migration effort.

Through the evaluation, we addressed some concerns that are often neglected in the literature, which focuses more on technical aspects such as communication technology, running environments, and performance benchmarks. In what concerns the migration effort, a significant effort was required for the decomposition of the domain model, but it can be done in the modular monolith

[9] https://microservice-api-patterns.org/patterns/quality/dataTransferParsimony/Pagination.

migration, free of the complexities of distributed programming. In what concerns the migration from the modular monolith to the microservice architecture, the inter-microservice communication required changes in the interfaces of each microservice, and the cost is directly related to the size and quality of the REST API. Even though this refactoring is made up of small changes, a poor quality interface can increase the cost of refactoring and propagate the changes to the microservice features. If these constraints are considered when designing modular interfaces, it will help to introduce remote invocations.

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 fosters separation of concerns, highlights complexities that might have to be addressed before implementing a microservice architecture, and helps in the decision on how to migrate. Note that, with the modular monolith, developers can predict the coupling of the microservices, the expected performance degradation of the communication, and decide the type of communication between the microservices to detect functionalities that can be affected by the lack of ACID transactional behavior. Additionally, it is possible to foresee the need to redefine a functionality, for instance, due to performance, and proceed to its refactoring in the context of the modular monolith, which is a more refactoring-friendly environment than the microservices system.

On the other hand, we could also observe the consequence of migrating a modular monolith with a significant number of uses relationships like the *LdoD Archive* in terms of coupling between microservices. As previously stated, the behavior of the uses interaction corresponds to a synchronous request/response style of communication between the microservices in order to maintain the dependencies. However, synchronous communication results in the coupling of the microservices being too tight due to the fine-grained behavior, which becomes problematic due to the weight of remote invocations and results in serious performance degradation. This analysis may trigger the need to redefine the functionalities to have asynchronous behavior.

In terms of performance, its impact was thoroughly evaluated. Focusing on the modular monolith, the impact on performance was related to the amount of information sent between the modules through the *DTOs*, while the number of inter-module invocations did not have a relevant effect. When the amount of data being transferred between modules is low, the modular monolith can often achieve the same, or even better, performance than before due to the introduction of unique identifiers that allow for faster access to the database. In contrast to the modular monolith, the microservice architecture's performance was impacted by the number of inter-microservice calls. Together, these two factors should be avoided when designing the microservice architecture, where performance degradation occurs as the amount of information transferred and the number of invocations increase.

The impact of performance degradation must be considered when deciding whether to migrate to a microservice architecture. The latency it adds to functionalities that are implemented through synchronous invocations of remote transactions can impair the application's usability. Therefore, if the decision is to migrate, it may be necessary to redefine the functionalities that have worse performance, changing the perception the end user has of the application. For instance, the *Fragment Listing* functionality could be redefined to present the results using pagination. This need to change the functionality behavior is often ignored by the literature.

To improve performance, different caches were implemented in the modular and microservice architectures. However, throughout their use in the application, these caches might become inconsistent as the information changes and thus affect the consistency of the application. Therefore, it is also necessary to perform an evaluation of the consistency of cache data and its impact on application behavior. In particular, it is necessary to analyze the immutability level of the data in cache and its frequency of change.

5.1. Research questions

Considering the first research question, we can conclude that most of the effort is associated with the number of multiplicity relationships between modules. There is also some effort in refactoring for distributed interfaces, but this effort is mostly technology intensive, can even be automated, and more complex situations can be prevented by designing the module interfaces in the first step.

Considering the second research question, we observed that there is a huge impact on performance, and this impact increases negatively with the number of information sent. Therefore, it is necessary to apply some performance tactics, such as the use of caches. This has a side effect associated with consistency that needs to be analyzed on a case-by-case basis.

In what regards to the third research question, in relation to the migration effort, we can conclude that using a two-step approach, most of the effort can be in the first step, which is done in a more refactoring-friendly environment. On the other hand, with respect to performance, there is already a significant impact on the performance of the modular monolith. This is due to the encapsulation of the module domain models. Therefore, it is already necessary to apply some architectural tactics to improve performance in this step. Finally, the introduction of distribution has an even greater impact on performance.

5.2. Validity threats

This study was subject to the following potential risks: (1) it is a single example of a migration; (2) the technology and programming techniques used in the monolith could affect the results; and (3) the decomposition was based on the criteria of a small team.

Although it is a single case study, it has some level of complexity, the monolith has 71 domain entities and 81 bidirectional relations, and the literature lacks descriptions of the problems and solutions associated with the migration from monolith to microservice architecture. This study addressed migration to modular monolith and microservice architectures and provided feedback on the challenges faced in migration that benefit 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 [22] based implementation of functionalities 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.

Furthermore, the case study presented specific types of functionalities that requested significant amounts of information or were implemented through fine-grained invocations with strong synchronous behavior. Different types of functionalities can present different performance results, but, in general, most of the functionalities in the *LdoD Archive* were similar to the selected functionalities. Therefore, a good coverage of the functionalities is provided.

The decomposition was driven by the association of modules with teams, so that it minimizes the average number of developers assigned to a microservice, which actually reflects the incremental development of the monolith rather than other decomposition criteria such as scalability or performance. However, to evaluate the migration effort and the impact on performance, the chosen decomposition does not interfere with the evaluation. In particular, considering that we have identified significant migration effort and performance degradation.

5.3. Lessons learned

The key lessons of the stepwise migration of the case study are:

- Modular monolith architecture
  - Modularization has a large impact on the refactoring effort.
  - Modularization introduces a significant performance overhead that already requires the use of performance optimization tactics commonly used in a distributed context, such as coarse-grained invocations and caches.
- Microservice architecture
  - The refactoring effort is reduced; it is mainly technology-related and may be automated.
  - Performance decreases as a result of the cost of remote communication, but the modular monolith allows us to foresee where these issues occur and identify the extent of the impact.
  - Performance optimization tactics in a distributed context introduce data consistency issues that may require a rethinking of functionality.

Therefore, the most costly refactorings can be done in the refactoring-friendly modular monolith, which includes the rethinking of functionality due to the estimated impact that may occur in data consistency when introducing distribution. In addition, it can be used as an architectural decision delay, where the migration to microservices is only done if there are demanding requirements for scalability. Overall, we could observe a scaling benefit of migrating the LdoD monolith to a microservice architecture.

6. Conclusion

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

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

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

Overall, the migration to a microservice architecture presented several challenges with different levels of impact on migration effort, performance, and data consistency, which heavily depended on the structure and semantics of the application. In the *LdoD Archive*, the migration had a serious impact on performance due to network overhead that proved to be far too high compared to previous architectures but offered a more scalable and manageable architecture.

Although we describe a single case study of migration, it is of a system that is in production and implemented using the best practices for the development of web applications. Additionally, our stepwise approach seems suitable for further case studies because it fosters a separation of concerns that helps with the controlled introduction of complexity and the analysis of the results.

CRediT authorship contribution statement

Diogo Faustino: Conceptualization, Investigation, Software, Validation, Writing – original draft. **Nuno Gonçalves:** Conceptualization, Investigation, Software, Validation, Writing – original draft. **Manuel Portela:** Validation, Writing – review & editing. **António Rito Silva:** Conceptualization, Investigation, Methodology, Supervision, Writing – original draft, Writing – review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The code resulting from the migration and the test scripts used to measure the performance are available in a public repository.

Acknowledgment

This work was partially supported by Fundação para a Ciência e Tecnologia (FCT), Portugal through projects UIDB/50021/2020 (INESC-ID) (DOI:10.54499/UIDB/50021/2020) and PTDC/CCI-COM/2156/2021 (DACOMICO) (DOI:10.54499/PTDC/CCI-COM/2156/2021).

References

[1] C. O'Hanlon, A conversation with Werner vogels, Queue 4 (4) (2006) 14–22, http://dx.doi.org/10.1145/1142055.1142065.
[2] J. Thönes, Microservices, IEEE Softw. 32 (1) (2015) 116–116.
[3] M. Fowler, Microservices, Web page: http://martinfowler.com/articles/microservices.html.
[4] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, 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, http://dx.doi.org/10.1109/ColumbianCC.2015.7333476.
[5] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: 2016 IEEE International Symposium on Workload Characterization, IISWC, 2016, pp. 1–10, http://dx.doi.org/10.1109/IISWC.2016.7581269.
[6] 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, http://dx.doi.org/10.1109/ICACEA.2015.7164727.
[7] O. Al-Debagy, 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, http://dx.doi.org/10.1109/CINTI.2018.8928192.
[8] F. Tapia, M. Mora, W. Fuertes, H. Aules, E. Flores, T. Toulkeridis, From monolithic systems to microservices: A comparative study of performance, Appl. Sci. 10 (17) (2020) http://dx.doi.org/10.3390/app10175797.
[9] P. Di Francesco, P. Lago, I. Malavolta, Migrating towards microservice architectures: An industrial survey, in: 2018 IEEE International Conference on Software Architecture, ICSA, 2018, pp. 29–2909, http://dx.doi.org/10.1109/ICSA.2018.00012.
[10] M. Kalske, N. Mäkitalo, T. Mikkonen, Challenges when moving from monolith to microservice architecture, in: I. Garrigós, M. Wimmer (Eds.), Current Trends in Web Engineering, Springer International Publishing, Cham, 2018, pp. 32–47.
[11] J. Gouigoux, 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, http://dx.doi.org/10.1109/ICSAW.2017.35.
[12] A. Bucchiarone, N. Dragoni, S. Dustdar, S.T. Larsen, M. Mazzara, From monolithic to microservices: An experience report from the banking domain, IEEE Softw. 35 (3) (2018) 50–55, http://dx.doi.org/10.1109/MS.2018.2141026.
[13] M.H. Gomes Barbosa, 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, http://dx.doi.org/10.1109/ICSA-C50368.2020.00015.
[14] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, S. Dustdar, F.B. Kessler, T. Wien, Migration from monolith to microservices: Benchmarking a case study, 2020, http://dx.doi.org/10.13140/RG.2.2.27715.14883, unpublished.
[15] D. Guaman, L. Yaguachi, C.C. Samanta, J.H. Danilo, 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.
[16] R. Flygare, A. Holmqvist, Performance Characteristics Between Monolithic and Microservice-Based Systems (Bachelor's thesis), Faculty of Computing at Blekinge Institute of Technology, 2017.
[17] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, M. Steinder, Performance evaluation of microservices architectures using containers, in: 2015 IEEE 14th International Symposium on Network Computing and Applications, 2015, pp. 27–34, http://dx.doi.org/10.1109/NCA.2015.49.
[18] X.J. Hong, H.S. Yang, 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.
[19] J.L. Fernandes, I.C. Lopes, J.J.P.C. Rodrigues, 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, http://dx.doi.org/10.1109/ICUFN.2013.6614932.
[20] B. Shafabakhsh, R. Lagerström, S. Hacks, Evaluating the impact of inter process communication in microservice architectures, in: QuASoQ@ APSEC, 2020, pp. 55–63.
[21] D. Haywood, In defense of the monolith, in: Microservices Vs. Monoliths - The Reality Beyond the Hype, Vol. 52, InfoQ, 2017, pp. 18–37, URL https://www.infoQ.com/minibooks/emag-microservices-monoliths.
[22] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley Longman Publishing Co., Inc., 2002.
[23] M. Portela, A. Rito Silva, A model for a virtual LdoD, Digit. Scholarsh. Humanit. 30 (3) (2014) 354–370, http://dx.doi.org/10.1093/llc/fqu004.
[24] A. Rito Silva, M. Portela, TEI4LdoD: Textual encoding and social editing in web 2.0 environments, J. Text Encoding Initiative 8 (2014) http://dx.doi.org/10.4000/jtei.1171.
[25] J. Raposo, A. Rito Silva, M. Portela, LdoD visual - a visual reader for Fernando Pessoa's book of disquiet: An in-out-in metaphor, Digit. Humanit. Q. 15 (3) (2021) 354–370, URL http://www.digitalhumanities.org/dhq/vol/15/3/000569/000569.html.
[26] G. Montalvão Marques, A. Rito Silva, M. Portela, Classification in the LdoD archive: A crowdsourcing and gamification approach, in: E. Ishita, N.L.S. Pang, L. Zhou (Eds.), Digital Libraries at Times of Massive Societal Transition, Springer International Publishing, Cham, 2020, pp. 313–319, http://dx.doi.org/10.1007/978-3-030-64452-9_29.
[27] G. Salton, C. Buckley, Term-weighting approaches in automatic text retrieval, Inf. Process. Manage. 24 (5) (1988) 513–523, http://dx.doi.org/10.1016/0306-4573(88)90021-0.

Diogo Faustino is a Software Developer based in Portugal. He is also a former student from Instituto Superior Técnico where he pursued his M.Sc. in Information Systems and Computer Engineering.

Nuno Gonçalves is a Software Developer based in Portugal. He is also a former student from Instituto Superior Técnico where he pursued his M.Sc. in Information Systems and Computer Engineering.

Manuel Portela is Professor of English and Director of the Ph.D. Program in Materialities of Literature at the University of Coimbra. His research addresses writing and reading media and how they impact on literary forms and practices. The most significant results of his work can be seen in Scripting Reading Motions: The Codex and the Computer as Self-Reflexive Machines (MIT Press, 2013), LdoD Archive: Collaborative Digital Archive of the Book of Disquiet (2017-2021), co-edited by António Rito Silva, and Literary Simulation and the Digital Humanities: Reading, Editing, Writing (Bloomsbury, 2022).

António Rito Silva is an Associate Professor at Instituto Superior Técnico of the University of Lisbon and researcher of Distributed Parallel and Secure Systems of INESC-ID. His research interests include software architectures for microservices and digital humanities. António has published more than 90 peer-reviewed articles in journals, conferences and workshops. He is leading research on the migration of monolith applications to a microservices architecture (https://github.com/social{software}/mono2micro).