Automated Identification of Monolith Functionality Refactorings for Microservices Migrations

José Correia
INESC-ID
Instituto Superior Técnico, University of Lisbon
Email: j.meneses.correia@tecnico.ulisboa.pt
Supervisor: António Rito Silva

***Abstract*—The process of migrating a monolith to a microservices architecture has a cost due to the migration of its functionalities in an eventual consistent transactional context. On the other hand, the object-oriented approach commonly followed in the development of monolith systems promotes fine-grained interactions in the functionalities implementation, which further increases the migration cost due to the large number of remote invocations between microservices. We propose a heuristic tool to help the software architect identify possible functionality refactorings and reduce the cost of their migration to microservices when applying the *SAGA* pattern in the functionality microservices implementation. The heuristic accuracy and efficiency of the tool are evaluated using a dataset extracted from 78 codebases and comparing with expert refactorizations.**

I. Introduction

The microservices architecture allows the split of a software development project into several small agile cross-functional teams and facilitates independent scalability of the services that constitute the product [1], [2]. On the other hand, it is common practice to start developing a complex system as a monolith [3], due to the shorter time to market, and the fact that it is difficult to find the correct modularization of a system without doing several refactorings. Therefore, many monolith systems have to be migrated to a microservices architecture due to the aforementioned advantages.

To address this problem, there is significant recent research on the migration of monolith to a microservices architecture [4], [5], [6], [7], [8], [9], [10], [11]. Although they propose different techniques, they concur on the definition of three steps of the migration process: collection, analysis, and visualization/migration. During collection, techniques like static and dynamic analysis are applied to extract data about the monolith behavior, either based on models or the monolith's codebase. This information is fed to the second step, which identifies candidate decompositions of the monolith into a set of microservices, based on a set of metrics to define the expected quality of the decomposition. Finally, some of the approaches also support visualization activities, where the architect can interact with a visual representation of the candidate decomposition and even change the decomposition while being informed by the recalculation of the quality metrics. Most of the approaches, but [6], [12], do not consider the interactive capability of experimenting with different decompositions.

The values returned by the quality metrics are strongly dependent on the monolith functionalities structure, but this aspect has not been addressed by the literature, except in [13], where we analyze how the refactoring of a functionality can significantly impact on the value of complexity metrics for the monolith decomposition.

In this article, we leverage on this previous work to support the architect with a recommendation mechanism for the refactoring of the monolith functionalities in the context of a candidate decomposition. In [13] two metrics are proposed to measure the complexity of migrating a monolith functionality to a microservices architecture. These metrics measure the migration effort due to the relaxing of the functionalities ACID transactional behavior into eventual consistency, when they are re-implemented using the *SAGA* pattern [14] in its orchestration style. Additionally, we observed a code smell for migrations with high complexity. We verified that the high complexity associated with the migration of a functionality is due to the number of inter-candidate microservices invocations, which increases the number of the *SAGAs* intermediate states. The main reason for this situation is that monoliths are implemented using fine-grained object-oriented invocations.

Therefore, although our previous work allowed the architect to be informed about the cost of migrating the functionality, given a candidate decomposition, she is not aware of the possible reductions that can exist if fine-grained inter-microservice invocations are refactored into a smaller number of coarse-grained interactions. This situation is worsened because a monolith can have hundreds, if not thousands, of functionalities, making it impractical for a manual inspection of each functionality structure, which indicates that some form of automation would be desirable.

We propose a recommendation system that, given a candidate decomposition of a monolith, and the monolith set of functionalities, informs the architect about the impact that transforming fine-grained invocations between microservices into coarse-grained ones has on the overall cost of migration.

Additionally, the recommendation system indicates, for each refactored functionality, what should be the *SAGAs* microservice orchestration coordinator. This will also help the developer in the refactoring activities because, given a coordinator, it is easier to identify how the refactoring can be done.

Therefore, given a candidate decomposition of a monolith, its set of functionalities, their sequences of accesses to domain entities, the data dependencies between those accesses, and metrics for the effort of migrating the functionality as a *SAGA* pattern in the context of the candidate decomposition, we address the following research question:

- Is it possible to recommend the refactoring of a functionality, by merging fine-grained inter-microservice interactions into coarse-grained ones, that minimizes its migration effort as an orchestrated *Saga* in the context of the decomposition?
- Can we characterize the *Saga* orchestrators that allow for a higher reduction of the migration effort?

–TODO : Change

In the next chapter, we discuss the work on monolith migrations and the identification of code-smells and design patterns. Chapter III-B presents defines the conceptual framework that is used in Chapter III-C to design the recommendation system. Chapter V evaluates the recommendation system on 78 codebases and compares some automatically generated refactorings with the refactorings produced by an expert. Finally, Chapter VI presents the conclusions.

II. Related work

The so-called gray literature proposes the use of the *SAGA* pattern to implement business functionalities in the microservices architecture [14]. This pattern is based on the seminal work by Hector-Molina and Kenneth [15] and addresses the lack of isolation due to the creation of intermediate states that are visible outside the scope of the functionalities execution. These intermediate states result in the introduction of eventual consistency, which adds to the complexity of the functionality because the business logic becomes intertwined with the management of the intermediate states, for instance, the need to have compensating transactions. Based on the concept of *SAGA*, in [13] are proposed two metrics to measure the complexity associated with a functionality implementation in a microservices architecture and the complexity that its implementation adds to the implementation of other functionalities. These metrics are built on the number of intermediate states and the number of inter-microservices invocations associated with the functionality implementation.

In [6] is proposed a set of operations for changing a candidate decomposition of a monolith system, like move entity between clusters or merge clusters, while recalculating the amount of inter-cluster communication. In [13] has been proposed three operations to refactor a functionality, given a candidate microservices decomposition, where the refactoring of fine-grained invocations into coarse-grained ones shown to have a significant impact on the reduction of complexity of migrating a functionality in a distributed context.

The current research on identifying code-smell and design patterns follow two main trends: heuristic and machine learning classification. In the heuristic classification approach, a set of code metrics is computed and combined to create detection rules [16], [17], [18], [19]. However, some drawbacks have been identified in this approach due to the low agreement between different detectors and difficulties in finding suitable thresholds to be used for detection [20]. Fontana et al. [21], [22], [23], propose the use of machine learning techniques. These techniques tend to be more flexible and independent compared to heuristic classification, since learning by example allows for better handling of distinct scenarios. However, they require extensive manual classification work to train the machine learning algorithm.

The research on the migration of monolith systems to a microservices architecture that uses automatic and semi-automatic methods already use heuristic, e.g. [4], [7], and machine learning, e.g. [24], [25], techniques to the identification of candidate decompositions, but there is no work on the automatic identification of code-smells and recommend refactorings to ease monolith functionality migration in the context of a candidate decomposition.

III. Solution

A. Strategy

In order to make a decision about the approach that better fits this use case, we start by analyzing how the human developer decides which changes must be done to the functionalities callgraphs to refactor them as Saga orchestrations.

In previous research [13], the authors defined a set of operations that can be made to a functionality callgraph, in the context of a candidate decomposition, in order to refactor it as a Saga orchestration without breaking the data-state layer of the codebase:

- *Sequence Change*: the flow of execution of the functionality is changed, which happens by swapping the order of the original sequence of local transactions, Fig. 1.
- *Local Transaction Merge*: used when two local transactions in the same cluster become adjacent in the sequential callgraph. Since it does not make sense to have a remote invocation between the same cluster, we can merge both the local transactions as is seen in Fig. 2.
- *Define Coarse-Grained Interactions*: this operation happens when both Sequence Change and Local Transaction Merge operations are applied sequentially, in that order.

Fig. 1: Sequence Change operation applied to a decomposition callgraph.

The authors proceeded to refactor a set of functionalities in the LdoD codebase into Saga orchestrations, and evaluate the results when it comes to the reduction of the FMC and SAC metrics.

Fig. 2: Local Transaction Merge operation applied to a decomposition callgraph.

Picking up on this work, we started by reverse-engineering the author's refactorings to understand his decision-making and conclude if it follows a standard pattern. We extended this analysis to 8 functionalities of the codebase, for which we extracted the initial sequential callgraph and the final callgraph resulting from the refactorings of the authors, and analyzed the source code to make conclusions on the design decisions made. With this, we reached the following conclusions:

- When it comes to the data dependencies, where two local transactions must not be swapped if the second depends on data read by the first, we saw that not always did the developer respect them. This conclusion does not suggest that the developer broke the data state changes in the refactored design, but instead, it shows sometimes when having a write operation in cluster B after a read operation in cluster A, the write in B does not actually use the data read in A.
- The rules defined in [13] were not enough to reach the refactorings made by the authors, since some data dependencies where not taken into consideration while refactoring the functionalities.
- Regarding the last question to be answered, we could not find a pattern regarding the decision, by the authors, of the clusters that should orchestrate each one of the functionalities.

By answering the questions above, we concluded that heuristic tools based on defined rules are too strict and do not account for the large number of levels of freedom that codebases have. These observations lead us to believe that a machine-learning classifier approach may be the end-game at developing a refactorization system that can analyze codebases implementing all kinds of design patterns, since theoretically, it is possible to train the classifier with a large dataset of functionalities to achieve a much better behavior when the model views data that is new and was not included in the dataset.

However, this characteristic of the machine-learning classifier, which requires an extensive dataset with human-made refactorizations, is challenging to solve. It is hardly possible to have a dataset of at least 100 functionality refactors analyzed purely by human developers without using existing automated tools simply because it takes an immense amount of time to complete. The only dataset we have available has the refactorizations made by the author in [13], and they are not enough for training a deep learning model.

Considering this, and although their limitations, we decided to approach the problem from a purely heuristic standpoint.

B. Functionality Migration

The migration of a monolith functionality is based on the information collected on the functionalities accesses to the monolith domain entities, the sequences of accesses, and their data dependencies. Therefore, a monolith is defined as a triple (F, E, T), where F denotes its set of functionalities, E the set of domain entities, T a set of traces of the monolith functionalities accesses.

The traces are defined as a triple (A, S, D), where $A = E \times M$ is a set of read and write accesses to domain entities ($M = \{r, w\}$), $S = A \times A$ a execution sequence relation between elements of A, which indicates that the first element of the pair was invoked, in the context of a monolith functionality, immediately before the second element, and $D = A \times A$ the data dependencies between accesses in the context of a sequence, where the first element is a read access, the producer, and the second element a write access, the consumer. Given a sequence, $s \in S$, its transitive closure, s_t, is a total order, in particular, any element is comparable and there is no circularities, $\forall_{(a_i,a_j)\in s_t}(a_j, a_i) \notin s_t$ For each functionality $f \in F$, $f.s \in S$ denotes its set of sequence accesses. Additionally, the data dependencies occur in the context of a functionality sequence of access and conform to the sequence order, $\forall_{(a_i,a_j)\in D}\exists_{s\in f.s}(a_i, a_j) \in s_t$.

Functionality migration occurs in the context of a candidate decomposition, in which each candidate microservice is represented by a cluster of domain entities. Therefore, given the set of sequence of accesses $f.s$ of a functionality f and a decomposition into a set of clusters of the domain entities, $C \subseteq 2^E$, where the clusters are non-empty and a domain entity is in exactly one cluster, the partition of a sequence s of a functionality f, $s \in f.s$, $P(s, C) = (LT, RI)$ is defined by a set of local transactions LT and a set of remote invocations RI, where each local transaction:

- is a tuple (A, S)
- is a subsequence of the functionality sequence of accesses, $\forall_{lt\in LT} : lt.s \subseteq s$;
- contains only accesses to the domain entities of a single cluster, $\forall_{lt\in LT}\exists_{c\in C} : lt.a.e \subseteq c$;
- contains all consecutive accesses in the same cluster, $\forall_{a_i\in lt.a, a_j\in s.a} : ((a_i.e.c = a_j.e.c \wedge (a_i, a_j) \in s) \Rightarrow (a_i, a_j) \in lt.s) \vee ((a_i.e.c = a_j.e.c \wedge (a_j, a_i) \in s) \Rightarrow (a_j, a_i) \in lt.s)$;

From the definition of local transaction, results the definition of remote invocations, which are the elements in the sequence that belong to different clusters, $RI = \{(a_i, a_j) \in s : a_i.e.c \neq a_j.e.c\}$.

Note that, in these definitions, we use the dot notation to refer to elements of a composite or one of its properties, e.g., in $a_j.e.c$, $.e$ denotes the domain entity in the access, and $.c$ the cluster the domain entity belongs to, and $lt.a$ denotes the set of accesses in the local transaction.

A partition of a functionality f, given a decomposition C, is the union of the partition of each one of its sequences, $P(f, C) = \cup_{s\in f.s} P(s, C)$, where local transactions and re-

mote invocations that are in the common prefixes of sequences are not repeated. Additionally, sequence and data dependence relations between local transactions are inferred from the functionality original sequence and data dependence relations and are denoted by $<_S$ and $<_D$, respectively.

The partition of a functionality f, given a decomposition C, and obtained using the above rules, is called the initial partition of f and it is denoted by $P_i(s, C)$ and $P_i(f, C)$, for, respectively, a sequence s and all sequences of f.

The refactor of a functionality, given a decomposition, is done from its initial partition to a *SAGA* implementing an orchestration, where the orchestrator is a pivot of the interactions between local transactions. Therefore, a partition of a functionality sequence of accesses is an orchestration if it is involved in all remote invocations, c is an orchestrator if $\forall_{(a_i,a_j)\in RI} : a_i.e.c = c \vee a_j.e.c = c$. Note that, by definition, a remote invocation occurs between different clusters.

The refactor of a functionality sequence of accesses is the change of its initial partition into a partition that is an orchestration. This refactorization is done to minimize its migration complexity in the context of a distributed transaction, because in the new implementation the functionlity business logic has to be migrated. This complexity is measured by the number of local transactions and the impact that a local transaction has on other functionalities migration. This is due to the lack of isolation that each local transaction brings to the functionality migration in a decomposition, which requires the introduction of compensating transactions and the need to handle intermediate states of the domain entities accessed by different functionalities [12], [13].

Functionality Migration Complexity (FMC). The complexity of migrating a functionality f, given its partition (LT, RI) in a decomposition, is the sum of the complexity of each one of its local transaction:

$$complexity(f, (LT, RI)) = \sum_{lt \in LT} complexity(lt) \quad (1)$$

Local Transaction Complexity. The complexity of a local transaction depends on the domain entities it writes because it is necessary to implement compensating transactions for when the functionality execution has to rollback. It also depends on the domain entities it reads because it is necessary to consider the intermediate states that other functionalities introduce when they write them in their local transactions.

$$complexity(lt) = \#writes(\{lt.s\}) + \sum_{e \in reads(\{lt.s\})} \#\{f_i \in F \backslash \{f\} : \exists_{lt_j \in P(f_i,C)} e \in writes(lt_j.s)\} \quad (2)$$

where

$$writes(s) = \{e : \exists_{s_i \in s}(e, w) \in prune(s_i)\} \quad (3)$$

$$reads(s) = \{e : \exists_{s_i \in s}(e, r) \in prune(s_i)\} \quad (4)$$

and the prune function, when applied to a sequence of accesses, removes all read accesses to a domain entity after the first read access to that entity, and removes all accesses to an entity after the first write access to that entity. This function identifies which accesses are relevant for other functionalities, because the local transaction executes with strict consistency properties and so, after a write local reads do not need to concern about external writes done by other functionalities, and only the first read has to consider intermediate generated by other functionalities.

System Added Complexity (SAC). The migration of a functionality impacts other functionalities migration complexity. For instance, if a write is done on an entity e due to the execution of a functionality f_i then every other functionality f_j (where $i \neq j$) that read the same entity e must have to be changed to handle the possible intermediate states. Hence, the cost of migrating f_j depends on the number of writes done by f_i in entities that f_j reads.

$$addedComplexity(f_i, (LT, RI)) = \sum_{lt \in P(f_i,C)} \sum_{f_j \in F \backslash \{f_i\}} \#(\{e : \exists_{lt_j \in P(f_j,C)} e \in reads(lt_j.s)\} \cap writes(lt.s)) \quad (5)$$

C. Saga Refactorization Algorithm

The refactoring is implemented by a brute-force heuristic algorithm to calculate, for each functionality, the orchestrations that have the lowest migration complexity. In Fig. 3 we present a flowchart with the high-level execution flow of the algorithm. It performs one refactorization for each one of the clusters in a functionality, picking in each iteration a different cluster as orchestrator.

In order to create a Saga design, the algorithm copies the initial callgraph structure and inserts one invocation of the orchestrator cluster between each invocation of two other clusters in the functionality. These invocations are added without containing any accesses to domain entities, to create the intermediate state where each service reports back to the orchestrator to inform if its transaction was successful or not. After this, the tool proceeds to execute a recursive method that iterates through each invocation in the functionality callgraph and checks if it can be merged with the previous invocation of the same cluster.

As we will explore later, two invocations can be merged if there are no read accesses in the latest δ invocations of other clusters before the second one. This scope allows us to have some configuration of the rigidity of the data dependence classification, and theoretically, lower values could result in more merge-operations.

If the invocations are mergeable, they collapse into the first one, and their domain entity accesses are pruned. The recursive method stops once we reach the end of a cycle through the callgraph where the algorithm did not find any invocation that could be merge, which indicates that the functionality is at the most simplified state that does not break data dependencies.

After all the invocation merges are complete, the Saga refactorization is saved in a data structure. The algorithm proceeds to execute the logic again, now with another cluster as orchestrator. When all of the clusters in the functionality have been considered, the work is done, and the execution reaches its end.

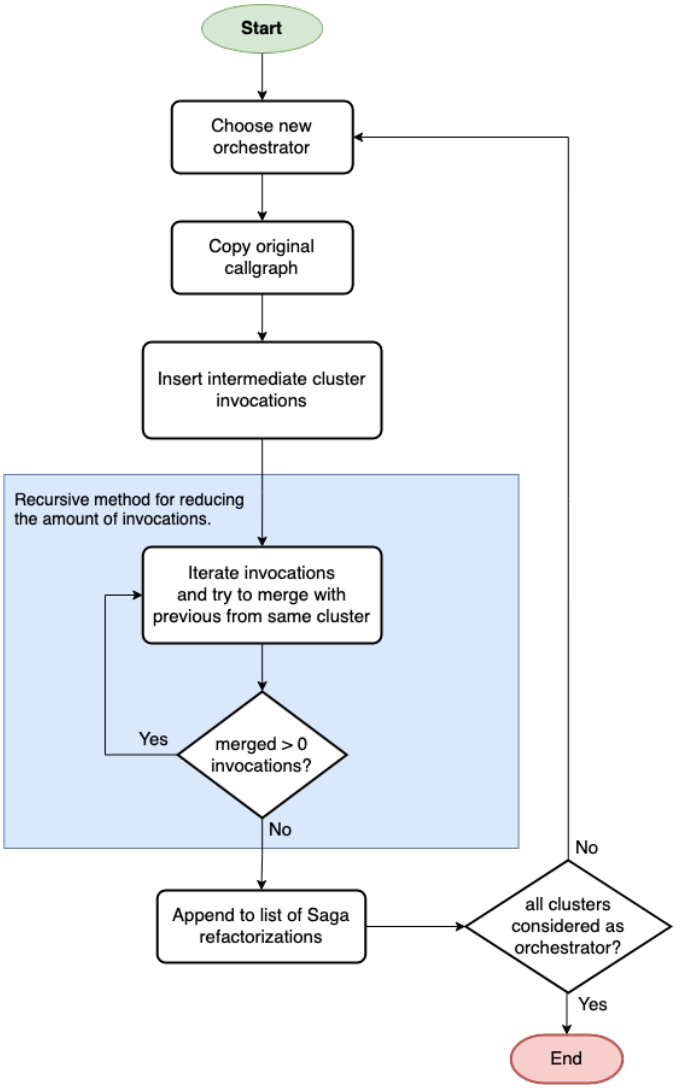

Fig. 3: Flowchart for the algorithm that computes the list of possible Saga orchestrations for a functionality.

The main algorithm $estimateSagas(F, C)$, Listing 1, uses the functionalities initial partitions P_i, given a candidate decomposition, and calculates, for each functionality, what is the complexity migration values when considering each of the clusters that are involved in the functionality implementation as orchestrators. The result is an ordered array of migration complexities where the cluster that has the lower value is the recommended orchestrator for the functionality refactoring. The algorithm has three steps: (1) sets a cluster orchestrator of the functionality, $setOrquestrator(P_i(s, C), c)$; (2) merges the invocations between the same pairs of clusters, to obtain coarse-grained invocations, $mergeInvocations(p, c)$; (3) calculates the complexity, $complexity(f, p_c)$.

Listing 1: *SAGA* estimator

```
estimateSagas(F,C) {
   complexities := array[F.size, C.size]
   for f := range F {
      for c := range C {
         p_c := {}
         for s := range f.s {
            p := setOrquestrator(P_i(s,C), c)
            p := mergeInvocations(p, c)
            p_c := p_c ∪ p
         }
         complexities[f,c] := complexity(f, p_c)
      }
   }
   complexities[f].sorted
}
```

To set a cluster as the orchestrator of an initial partition of a functionality it is necessary to add empty local transactions to the orchestrator cluster and remote invocations to the other clusters, Listing 2. The sequence of execution of the local transactions is preserved because the only change is the introduction of the empty local transactions, which work as pivots between the invocations. Therefore, the data dependencies between the local transactions are not changed.

Listing 2: Set Orchestrator for Partition

```
setOrchestrator((LT,RI),c) {
   resultLT := LT
   resultRI := {}
   skip := false
   for lt = range LT.sortedBy(<_S) {
      if (!skip) {
         if (lt.c == c) {
            resultRI := resultRI ∪ {(lt,
                lt.next)}
            resultRI := resultRI ∪ {(lt.next,
                lt.next.next)}
            skip := true
         } else {
            emptyLT := new emptyLT(c)
            resultLT := resultLT ∪ {emptyLT}
            resultRI := resultRI ∪ {(emptyLT,
                lt)}
            skip := false
         }
      }
   }
   return (resultLT, resultRI)
}
```

Fig.4 exemplifies a transformation made according to *setOrchestrator* in Listing 2, where we set the empty pivot orchestrator invocations between each one of the other cluster invocations

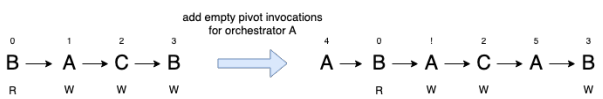

Fig. 4: Operation of adding pivot orchestrator invocations

The merge of invocations, Listing 3, merges the invocations that occur between the orchestrator and another cluster, if there is no data dependence with a local transaction that occurs in between. It repeats while two remote invocations can be merged into a coarse-grained remote invocation, and the `canMerge` condition, which applies to pairs of invocations between the same clusters, is defined by the data dependence relation $<_D$, such that, it does not exist a local transaction between the two remote invocations that the local transactions in the second remote invocation have a data dependence on. As a result, the merged local transaction sequences are concatenated and the data dependencies between them are preserved.

Listing 3: Refactor through merge of fine-grained invocations into coarse-grained.

```
mergeInvocation((LT,RI),c) {
    resultLT := LT
    resultRI := RI
    while (ri1 in range RI, ri2 in range
        RI.after(ri1), canMerge(ri1,ri2)) {
       ri1.caller.prune(ri2.caller)
       ri1.callee.prune(ri2.callee)
       resultLT := resultLT \ {r21.caller, ri2
           .callee}
       resultRI := resultRI ∪
           {(ri2.previous.caller, ri2
           .next.callee)} \ {ri2.previous, ri2,
           ri2.next}
    }
    return (resultLT, resultRI)
}
```

The $canMerge(ri_1, ri_2)$ function can be parameterized to allow variations on the scope (δ) of data dependencies to be considered. When $\delta = 1$ a data dependence is only considered if it occurred in the local transaction immediately before the local transaction being analyzed, however, if $\delta = \infty$ are considered all the local transactions in between the ones to be merged. Allowing different scopes permits different evaluations because a previous read does not necessarily imply that it is used in a subsequent write.

Figure 5 illustrates a transformation performed by *mergeInvocation* in Listing 3, where all invocation that do not have data dependencies, according to the value of δ, are merged.

Fig. 5: Operation of merging and pruning invocations without data dependence

IV. Implementation

The main contribution of this work is a Functionality Refactorization Service[1] , built on Golang in order to take part of the excelent handling of concurrency that this language implements. Given a codebase analyzed by the Mono2Micro static analyzer and a cluster decomposition, the service applies the algorithm presented in Chapter III-C which can estimate the Saga refactorings that minimize the migration cost of a codebase to a microservices architecture. In order to communicate with external services, the transport layer of the tool exposes a REST API with support for HTTP requests.

The tool starts by extracting the valid controllers of the codebase decomposition that can be implemented using a Saga Pattern. A JSON file is written in the file system containing data about the codebase being refactored, the configuration parameters of the request, and the controllers classified as valid to be implemented as an orchestration. From this moment forward, if the user performs a *View Status* request to the service, he will receive the data in this JSON file, which will be updated as refactorizations finish. With this, the service starts iterating through each one of the controllers and instantiates a *goroutine* to handle the computations. This way, we can achieve concurrency and parallelism, assuring that Golang's scheduler manages the routines memory allocation and uses multiple processors to reach a lower execution time. After spawning a *goroutine* for each controller, the service performs the HTTP response to the client immediately, without waiting for the refactorization results, which allows for reduced latency.

The Refactorization Service stands as a separate Docker container that runs in port 5001 and shares a file system volume with Mono2Micro's backend application, which is used to read the codebase decomposition files generated by the other functionalities of the Mono2Micro system.

In order to integrate seamlessly with Mono2Micro, we developed a graphical user interface that an architect can use to interact with the Refactorization Service directly after decomposing the monolithic codebase, by using the systems already existing features. This interface allows to request the refactorization of a specific codebase, using a configurable data dependence threshold. The results are them progressively shown, as the tool finishes refactoring each functionality.

V. Evaluation

To answer the first research question and validate that refactorings can minimize the cost associated with the migration, we executed the tool for a dataset of 78 codebases and analyzed the results to evaluate the reduction of complexity. To validate the accuracy of refactorizations we compared with the refactorizations done by an expert [13] on the LdoD codebase in terms of complexity reduction and feasibility.

A. Complexity Reduction

The 78 codebases are monolith systems implemented using Spring-Boot and an Object-Relational Mapper (ORM), where Java Persitence API (JPA) is used in 75 of the codebases and Fénix Framework in the other 3. The Spring-Boot controllers are used to implement the monolith transactional functionalities that access the persistent domain entities implemented using the ORM. The data set was generated through static

[1]https://github.com/socialsoftware/mono2micro/tree/master/tools/functionality_refactor/src

analysis of the source code, where sequences of read/write accesses to the domain entities are obtained for the functionalities.

Each codebase was decomposed into clusters, and, after applying the initial partition to the functionalities sequences, the functionalities were selected based on two conditions: (1) the sequence of accesses has least one write access; (2) the sequence of accesses includes more than two local clusters. These are pre-conditions for the implementation of the functionalities as *SAGA*s, by excluding queries and functionalities that can not be implemented as *SAGA*s because only have one or two clusters. After these selections, the dataset was formed by 652 functionalities.

Table I presents the results, on average, of applying the refactoring heuristic to the 652 functionalities. It compares the initial complexity with the complexity of the refactored functionalities. The heuristic was applied for three variations of the data dependencies between local transactions, where $\delta = 1$ means that there is no data dependence with the local transaction that executed immediately before, $\delta = 2$ means that there no data dependence with the two local transactions that executed immediately before, and $\delta = \infty$ that there is no data dependence with any local transaction that executes in between the data transactions to be merged. It is necessary to analyze these cases because the data dependencies are obtained from an analysis of the sequences and this may not necessarily mean that a local transaction effectively uses the data read on previous local transactions. For instance, if there is a write access in a local transaction we consider that data dependencies exist with all the local transactions that occur before and that have at least one read access, because the value read may be used to calculate the value that is going to be written. This is a limitation of the static analysis procedure that captures the monolith behavior, which does not capture the data-flow, and so we considered the worse situation in terms of data dependencies.

The table also presents the percentage of reduction in the number of local transactions and domain entities access. Additionally, it includes the average number of merges, and the heuristic performance, which was calculated running 10 samples on an Intel i5 2.90GHz, 2 Cores with 4 Threads each.

From the observation of the table can be concluded that there is a significant positive impact of the refactorings on the complexity reduction and that the scope of data dependencies also impacts on final complexity, as expected a larger scope will reduce the number of possible merges, but it is not significant, which means that in most of the functionalities there is a data dependence between a local transaction and the local transaction that executes immediately before. It is also worth mentioning the reduction in the number of domain entities access, which results from the fact the access are localized inside the same local transactions and, so, the visibility of their effects to other local transactions is reduced.

The analysis of the codebases allows us to partially answer the first research question, due to the reduction in the complexity, but it is necessary to analyze the accuracy of the results: are the recommended refactorings feasible?

TABLE I: Average values for running the recommendation heuristic for 652 functionalities, considering $\delta = 1, 2, \infty$.

Metric	**Initial**	**Final**			**% Reduction**		
		$\delta = 1$	$\delta = 2$	$\delta = \infty$	$\delta = 1$	$\delta = 2$	$\delta = \infty$
FMC	329.0	78.4	86.7	95.6	76.1%	73.6%	70.9%
SAC	404.9	150.7	170.6	201.3	62.7%	57.9%	50.3%
Local Transactions	35.5	4.3	4.7	5.1	87.9%	86.8%	85.6%
Entity Accesses	38.7	10.3	11.3	12.4	73.4%	70.8%	67.9%
Number of Merges		22.5	21.7	21.4			
Execution time (sec)		3.76	4.13	4.40			

B. Heuristic Accuracy

The tool was applied to the LdoD archive, which consists of 133 controllers and 67 domain entities. The decomposition used was done by an which cuts the system into 5 clusters. The analysis involved comparing the results with the refactorings of 9 controllers of LdoD [13], where the expert refactored the code and calculated the initial and final FMC and SAC. We do not compare the SAC complexity because it depends on the type of local transaction, which the algorithm cannot predict. For instance, whether a local transaction is compensatable or not. The heuristic always considers the worst-case scenario.

Table II presents the results when the heuristic was applied with a data dependence scope of 1. It can be observed a clear relation between the complexity reduction of the functionalities when refactored by the heuristic algorithm and by the expert, since the average reduction from both differs only 7.4%, with the tool being able to achieve bigger complexity reductions on average. In the last column, we present the relative distance between the complexity of the best refactoring calculated by the tool and the complexity of the refactoring with the same orchestrator as chosen by the expert. As it can be seen in cells with a 0% distance value, 5 out of the 9 functionalities, had both the expert and the tool selecting the same orchestrator. Note that the value is 0% because orchestration distance comparison is not done with the complexity values reported by the expert, but with the values calculated by our tool.

After having the best refactoring estimated by the algorithm, the next step of the evaluation involved manually verification in the source code, to check if the recommended orchestrations are feasible, for instance, that they do not break any data dependence between entity accesses.

The selection process of the functionalities was based on the values from Table II and the functionalities selected fall in one of the following categories: (1) the tool recommendation has better complexity reduction; (2) the expert refactoring has a better reduction; (3) the orchestrators selected by the tool

TABLE II: Functionality Migration Complexity reduction resulting from a refactoring using the tool with $\delta = 1$ and by an expert.

Functionality	**Initial FMC**		**Final FMC**		**Reduction %**		**Orch. dist.**
	Tool	Expert	Tool	Expert	Tool	Expert	
removeTweets	151	134	109	82	27.8%	38.8%	0%
getTaxonomy	317	317	159	192	49.8%	39.4%	1.24%
createLinear-VE	2978	1790	263	383	91.1%	78.6%	0%
signup	1550	1490	314	376	79.7%	74.8%	0%
approve-Participant	193	190	113	147	41.5%	22.6%	0%
associate-Category	1813	1803	642	662	64.6%	63.3%	14.97%
delete-Taxonomy	261	253	187	164	28.4%	35.2%	11.79%
dissociate	806	772	358	489	55.6%	36.7%	0%
merge-Categories	485	453	187	253	61.4%	44.2%	37.87%
Averages	950.4	800.2	259.1	305.3	55.5%	48.2%	

and the expert are different; (4) the initial complexity value calculated by the tool is much different than the one calculated by the expert. This allows us to identify 6 cases where to do this analysis.

The refactoring of the *mergeCategories* functionality has a complexity reduction of 62.7% when compared to the initial architecture and converted 32 independent cluster invocations to 5. The algorithm chose a different orchestrator than the expert and achieved a complexity 18.5% lower. After manually checking the source code, we verified that this refactor is a valid option, all the operations are valid and the data dependencies that exist in the functionality are all taken into consideration. We can see that the tool computed fewer cluster invocations, fewer entity accesses, and fewer write operations, which reduces the complexity compared to the expert refactor. By looking at the code, we can see that the expert choice was based on selecting the cluster that contains more entity accesses as the orchestrator, which resulted in more repetitive invocations of other clusters. By choosing an orchestrator that was not so obvious at first glance, the algorithm was able to obtain a better refactor. The same happened for the functionality *getTaxonomy*, where the tool recommend a different orchestration than the expert, but it was able to obtain an higher reduction of the complexity by having all the operations condensed in 1 invocation per cluster, while the expert refactorization created more intermediate states. The source code also revealed that no data dependencies were broken by the tool's operations, and the sequence is applicable.

In the functionality *approveParticipant*, both, the tool and the expert, selected the same orchestrator cluster and did a very similar refactorization, where only 1 additional merge operation by the tool was enough to reduce the complexity in 41.5% when compared to the expert's 22.5%. The functionality *dissociate* falls in this same category since it shows that the tool was able to achieve a lower complexity with the same orchestrator, however by checking the source code we see that some invocations were merged while having a data dependence with previous invocations, which reveals invalid merges. This occurred because the data dependence distance had a scope > 1 and due to the test being executed with $\delta = 1$ the tool did not consider it when verifying if an invocation can be merged.

The refactor of *removeTweets* saw a complexity reduction of 26.8% when compared to the initial architecture and converted 14 independent cluster invocations to 7. Both the algorithm and the expert chose the same orchestrator and performed a very similar refactor but the expert refactor achieves a higher complexity reduction. After manually checking the source code, we verified that this refactor is a valid option, except for one invocation that could have been merged with a previous one. Since in the original trace there was a data dependence between that operation and the one immediately before, the tool did not do the merge. This behavior reveals that the heuristic rules can be too restrictive and since the static analysis is not capturing the data-flow, we cannot accurately conclude that a write following a read does not use data from the first, which is something that an expert can directly observe by reading the code.

In functionality *createLinearVE* there is a significant difference of 1188 between the tool's initial complexity and the one calculated manually by the expert. This functionality is quite complex, with 108 local transactions and 210 entity accesses, and from looking at the source code, it involves many if/else conditions that define which accesses are performed during run-time. Ultimately, this difference relates to how the static analysis builds the functionality trace since it does not account for conditions and so all branches are collapsed in a single sequence. When the expert manually calculates the complexities, she only considers one of the if/else conditions, which results in fewer entity accesses and lower initial complexity.

To answer the first research question fully, we can say that most of the refactorizations applied to LdoD were feasible. However, there was at least 1 case where data dependencies were broken due to the configuration of the data dependence scope δ as 1. This difficulty when defining thresholds is still one of the major drawbacks of most heuristic techniques for code-smell and design-pattern detection [20].

C. Orchestrators characterization

The validation of the refactorization of the LdoD codebase, when compared to the expert, has shown that the recommended refactorings highly reduce the complexity of migrating the functionalities and they, in most of the cases, even when the scope is 1, do not break the data dependencies between the accesses. Another interesting observation is that the heuristic can suggest better refactors than the ones envisioned by the expert. To try to explain why the experts intuition may be misleading, take us to the second research question: what are the characteristics of a cluster that make it a better fit for being a *SAGA* orchestrator.

We defined 4 metrics that measure, for each cluster and in the context of the functionality being refactored, several aspects, like the number of read and write accesses to the cluster domain entities, the number of times the cluster is invoked in the context of the functionality, and the number of the invocations that cannot be merged due to a data dependence.

TABLE III: Correlation between the metrics for the orchestrator cluster and the reduction of the functionality migration complexity and system added complexity

Metric	**Correlation** (r)		
	FMC	SAC	FMC+SAC
# read accesses to domain entities	0.145	-0.083	0.014
# write accesses to domain entities	-0.145	0.083	-0.014
# times the cluster is invoked in functionality	-0.103	-0.090	-0.112
# initial data dependencies, when $\delta = \infty$	-0.110	0.210	0.087

These metrics were visualized for the orchestrators of each one of the recommended refactorizations for 78 codebases and correlations were calculated between the value of the metrics and the reduction of the complexity. Table III shows the resulting correlations with values between $[-0.145, +0.210]$, which demonstrates that the data has a big variation and does not necessarily follow a trend line, with points that are too dispersed in space. The metric with higher correlation was related to the metric for reads accesses of orchestrators, but, even though, not significative.

The low correlation values show that the metrics cannot characterize the orchestrator, due to the great diversification of code patterns in the functionalities. A good orchestrator for a given codebase might have a lot of invocations, for example, while in another codebase we verified that the opposite may happen, which makes it hard to set a characterization for an orchestrator that applies to all the cases.

To answer the second research question we can conclude that it was not possible to extract enough metrics from the functionality sequence to characterize what is a good orchestrator. This increases the need of the recommendation heuristic tool which applies refactorization operations and validates if a given candidate orchestrator has the lowest complexity.

D. Threats to Validity

In terms of internal validity, it is verified that the refactorings provide a significant reduction of the complexity, even though the data dependencies are inferred from the analysis of the sequence of accesses and not directly obtained from the code. This may have impact on the accuracy of the results, but the experiment with variations of the scope of data dependencies has shown consistent results. Therefore, even if the architect uses the wider scope, the recommended refactorings do not have a significant variation on the complexity. The sequences of accesses used for the validation, obtained from static analysis linearize all the accesses of a functionality in a single sequence. However, we obtained results similar to the expert refactorizations, sometimes even better. This also depends on the used dataset, a future version of that static analyser that captures the data-flow will allow to work with a dataset that will provide more precise results.

In terms of external validity, the dataset is for a small set of technologies, Spring-Boot and JPA, but the functionalities logic is technology independent.

VI. Conclusions

This paper proposes an heuristic to help a software architect identify the possible refactorings of monolith functionalities to reduce their migration cost to a microservices architecture applying the *SAGA* pattern. This cost is due to the functionality migration, because of the introduction of eventual consistency on the functionality behavior, which adds an extra complexity to its implementation.

The results show that the tool was able to recommend refactorings that allow a significant reduction of the functionalities migration. Interestingly, some of the recommended refactorings provide higher reductions than the ones identified by an expert, which suggests that it would be interesting to characterize what are the properties of a good candidate for a *SAGA* orchestrator. However, a study with 4 metrics that characterize the clusters accessed by the functionality, shows that there is no statistical significance in the correlation between the values of the metrics and the reductions on complexity. This problem is left for future work, and also indicates that a heuristic approach, which calculates all combinations, fits these cases where there is significant variation in the dataset.

References

[1] C. O'Hanlon, "A conversation with werner vogels," *Queue*, vol. 4, no. 4, p. 14–22, May 2006. [Online]. Available: https://doi.org/10.1145/1142055.1142065

[2] M. Fowler, "Microservices." [Online]. Available: http://martinfowler.com/articles/microservices.html

[3] ——, "Bliki: Monolithfirst." [Online]. Available: https://martinfowler.com/bliki/MonolithFirst.html

[4] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, M. Aiello, E. B. Johnsen, S. Dustdar, and I. Georgievski, Eds. Cham: Springer International Publishing, 2016, pp. 185–200.

[5] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE, 2017, pp. 524–531.

[6] R. Nakazawa, T. Ueda, M. Enoki, and H. Horii, "Visualization tool for designing microservices with the monolith-first approach," in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, Sep. 2018, pp. 32–42.

[7] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[8] L. F. A. Nunes, N. A. V. Santos, and A. R. Silva, "From a monolith to a microservices architecture: An approach based on transactional contexts," in *European Conference on Software Architecture (ECSA)*, ser. LNCS, vol. 11681. Springer International Publishing, Sep. 2019, pp. 37–52.

[9] 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.

[10] M. Daoud, A. E. Mezouari, N. Faci, D. Benslimane, Z. Maamar, and A. E. Fazziki, "Automatic microservices identification from a set of business processes," in *Smart Applications and Data Analysis*, M. Hamlich, L. Bellatreche, A. Mondal, and C. Ordonez, Eds. Cham: Springer International Publishing, 2020, pp. 299–315.

[11] A. Selmadji, A. Seriai, H. L. Bouziane, R. Oumarou Mahamane, P. Zaragoza, and C. Dony, "From monolithic architecture style to microservice one based on a semi-automatic approach," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 157–168.

[12] N. Santos and A. R. Silva, "A complexity metric for microservices architecture migration," 2020.

[13] J. F. Almeida and A. R. Silva, "Monolith migration complexity tuning through the application of microservices patterns," *Lecture Notes in Computer Science*, vol. 12292, pp. 39–54, 2020.

[14] C. Richardson, *Microservices Patterns*. Manning Publications Co., 2019.

[15] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '87. New York, NY, USA: Association for Computing Machinery, 1987, p. 249–259. [Online]. Available: https://doi.org/10.1145/38713.38742

[16] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui, "A bayesian approach for the detection of code and design smells," in *2009 Ninth International Conference on Quality Software*, 2009, pp. 305–314.

[17] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. D. Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 36, pp. 20–36, 2010.

[18] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. L. Meur, "A method for the specification and detection of code and design smells," *IEEE Transactions on Software Engineering*, vol. 41, pp. 462–489, 2015.

[19] F. Palomba, A. Panichella, A. Lucia, R. Oliveto, and A. Zaidman, "A textual-based technique for smell detection," 05 2016.

[20] F. Pecorelli, F. Palomba, D. D. Nucci, and A. D. Lucia, "Comparing heuristic and machine learning approaches for metric-based code smell detection," *ICPC '19: Proceedings of the 27th International Conference on Program Comprehension*, vol. 11681, pp. 93–104, 2019.

[21] M. Zanoni, F. A. Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *The Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[22] ——, "Comparing and experimenting machine learning techniques for code smell detection," *Empirical Software Engineering*, vol. 21, p. 1143–1191, 2016.

[23] M. Zanoni and F. A. Fontana, "Code smell severity classification using machine learning techniques," *Knowledge-Based Systems*, vol. 128, pp. 43–58, 2017.

[24] S. Ma, Y. Chuang, C. Lan, H. Chen, C. Huang, and C. Li, "Scenario-based microservice retrieval using word2vec," in *2018 IEEE 15th International Conference on e-Business Engineering (ICEBE)*, 2018, pp. 239–244.

[25] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," *Journal of Systems and Software*, vol. 151, pp. 243–257, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121219300408