Mono2Micro - From a Monolith to Microservices: MetricsRefinement

João Francisco Almeida
Instituto Superior Técnico
Lisbon, Portugal
joao.santos.almeida@tecnico.ulisboa.pt

***Abstract*—The microservices architecture has become mainstream for the development of business applications because it supports the adaptation of scalability to the type of demand, but, most importantly, because it fosters an agile development process based on small teams focused on the product. Therefore, there is the need to migrate the existing monolith systems to microservices. Current approaches to the identification of candidate microservices in a monolith neglect the cost of redesigning the monolith functionality due to the impact of the CAP theorem. In this paper we propose a redesign process, guided by a set of complexity metrics, that allows the software architect to analyse and redesign the monolith functionality given a candidate decomposition. Both, the redesign process and the metrics are evaluated in the context of candidate decompositions of two monolith systems.**

***Index Terms*—Microservices architecture, Monolith migration, Complexity metrics, Microservices patterns**

I. Introduction

Microservices architecture emerged due to the need to have highly available and scalable systems that can be developed by multiple teams in an agile environment. This is achieved through the definition of independently deployable distributed systems, implemented around business capabilities. As the monolith application size increases, it imposes several drawbacks as the lack of agility, modifiability and deployablity. As a consequence there is the need to migrate monolith systems to a microservices architecture.

However, this transition imposes a cost because the application cannot preserve the behavior that existed in the monolith. This is due to the introduction of distributed transactions, as the monolith functionalities will be implemented through multiple independent microservices (transactions). Therefore, transaction management is more complex in a microservice architecture because transactions cannot be executed according to the ACID (Atomicity, Consistency, Isolation, Durability) properties, which introduces extra complexity for developers to handle. This extra complexity is explained by the CAP theorem [6], where the decision to maintain the same level of consistency as in a monolith can only be achieved through the application of a two-phase commit protocol, which does not scale with many local transactions. To solve this problem, the use of sagas [5] was suggested, in the context of the microservices architecture [11], [13], as the main alternative to the two-phase commit protocol to handle distributed transactions. On the other hand, the *API Gateway* pattern has been proposed [11], [13] to implement queries in a distributed system.

The *SAGA* pattern can be applied to functionalities that create or update data, and consist in dividing a transaction in multiple local transactions, where each local transaction is executed inside a single service following the ACID properties. A saga can have two different structures: (1) choreography where the decision and sequencing is distributed through the saga participants, or (2) orchestration, where the decision and sequencing is decided in one orchestractor class, inside a cluster. Independently of the structure, the usage of sagas can guarantee the properties atomicity, consistency, and durability but cannot ensure the isolation property [13].

The lack of isolation can generate anomalies such as: (1) lost updates - when a saga overwrites data without reading changes performed by others sagas, (2) dirty reads - when a saga reads data changed by others sagas that have not been committed, (3) nonrepeatable reads - a saga reads the same data twice and gets different results. To correct this anomalies there is a set of countermeasures that can be applied. One of them is the semantic lock that corrects these errors by creating intermediate states as application-level locks that indicate if an entity was written by one saga, alerting others concurrent sagas to these events. It can be integrated with the *SAGA* pattern typically indicating the current saga state. Because now all the functionalities must be aware of the semantic lock, this imposes extra complexity to the functionalities design and implementation. It also adds complexity to queries, that have to handle the possible combination of semantic locks of the data that they are integrating.

Due to the lack of isolation, the saga local transactions can be of three types: (1) pivot - a transaction that if succeeds then the saga is going to succeed; (2) retriable - transactions that occur after the pivot transactions, do not rollback; and (3) compensatable - the transactions that may have to rollback. In a saga there is at most one pivot transaction, and all transactions that are not retriable nor the pivot transaction, are compensatable.

In previous work [12], [14], a tool was developed that collects information from monolith systems and, based on similarity measures, suggests a microservice candidate decomposition. Its level of complexity can be assessed through a complexity metric that determines the decomposition complexity based on the mean of its functionalities complexity.

This metric calculates the impact that the relaxing of atomic transactional behaviour has on the redesign and implementation of the functionalities. In this paper we leverage on the previous work by, given a decomposition and a complexity value, support the redesign of the functionalities and queries, applying the *SAGA* and the *API Gateway* patterns, while the complexity value is tuned, because there is more precise information on the decomposition. With our approach we intend to answer the following questions:

- **RQ1.** What set of operations can be provided to the architect such that the functionalities can be redesigned by applying microservices pattern?
- **RQ2.** Is it possible to refine the complexity value associated with the monolith migration when there is additional information about the functionalities redesign?

This paper makes two contributions. First, we define a set of operations that the architect can apply to the initial execution flow of a monolith functionality, such that it can be transformed in a microservices execution flow based on the *SAGA* pattern. Second, we define new metrics that provide a more precise value on the cost of the migration, due to the inclusion of the information about the application of the *SAGA* pattern.

In the next section we present the set of operations used in the redesign. In section III we present the new complexity metrics. In section IV we evaluate our work in the context of two monolith systems. Section V addresses the related work and section VI the conclusions.

II. Functionality Redesign

Definition: Monolith. A monolith is a pair (F, E), where F represents its set of functionalities, the functionalities are represented with lower case f, and E represents its set of domain entities, which are accessed by the functionalities, the domain entities are represented with lower case e.

The entities are accessed by the functionalities in two modes, read and write. Therefore, $M = \{r, w\}$ represents the access modes in a monolith, and an access is a pair domain entity access mode, represented by (e, m).

The accesses of a functionality f are represented as a sequence of accesses s, where S represents all the sequences of accesses done in the monolith by its functionalities to the domain entities, $f.sequence$ denotes the sequence of access of functionality f, $s.entities$ denotes the entities accessed in sequence s.

It is also defined the auxiliary function $entities(s : S, m : M) : 2^E$, as $entities(s, m) = \{e \in E : (e, m) \in s\}$, which returns the entities accessed in s in mode m.

When a monolith is decomposed into a set of candidate microservices, each candidate microservice is a cluster of domain entities.

Definition: Monolith Decomposition. A monolith decomposition into a set of candidate microservices is defined by a set of clusters C of the monolith domain entities, where each cluster c represents a candidate microservice and $c.entities$ denote the domain entities in cluster c, such that all domain entities are in a cluster, $\bigcup_{c \in C} c.entities = E$, and in a single one, $\forall_{c_i \neq c_j \in C} c_i.entities \cap c_j.entities = \emptyset$.

Given $e \in E$ and a decomposition C, $e.cluster$ denotes the entity's cluster, and given a set of entities $E' \subseteq E$, $E'.cluster = \{c \in C : \exists_{e \in E'} e \in c.entities\}$ denotes its set of entities clusters.

Given a monolith candidate decomposition, the monolith functionalities are decomposed into a set of local transactions, where each local transaction corresponds to the ACID execution of part of the functionality domain entity accesses in the context of a candidate microservice.

Definition: Functionality Decomposition. A monolith functionality f is decomposed, in the context of a candidate decomposition C, by a sequence of sequences of access to domain entities, denoted by $f.subsequences$, where all domain entities in a subsequence are in the same cluster, $\forall_{s \in f.subsequences} \exists_{c \in C} : s.entities \subseteq c.entities$, two consecutive subsequences occur in different clusters, $\forall_{0 \leq i < f.subsequences.size - 1} f.subsequences[i].entities$ $.cluster \neq f.subsequences[i+1].entities.cluster$. In order to have a consistent subsequence associated with a functionality f in a decomposition, the following condition must hold:

$$concat_{i=0..f.subsequences.size-1}(f.subsequences[i]) = prune(f.sequence)$$

Where the $prune$ function removes, for each sequence of accesses inside each cluster $c \in C$, the accesses according to the following rules: (1) if a domain entity is read, all subsequent reads of that entity are removed, (2) if an domain entity is written, all subsequent accesses of that entity are removed. A sequence of domain entity accesses where these two rules hold is pruned, it only contains the read and write accesses that are visible outside the cluster, the ones that are relevant for the semantic lock countermeasure.

In the redesign of a functionality in the context of a decomposition we define the set of local transactions participating in the saga that implements the functionality.

Definition: Local Transaction. A local transaction lt of a functionality f, is a pair (s, t), where s is a pruned sequence of access domain entities, all its accesses are to domain entities of the same cluster, $\exists_{c \in C} : s.sequence.entities \subseteq c.entities$, and t is the transaction type, which can be $compensatable$, $pivot$, and $retriable$.

T denotes the set of transaction types, LT denotes the set of local transactions in a decomposition, lt denotes a local transaction, $lt.cluster$ denotes the cluster where the lt occurs, $lt.sequence$ denotes the sequence of accesses, and $lt.type$ denotes the type of the local transaction.

A local transaction sequence should be pruned, for each domain entity in the sequence there is $0..1$ read accesses and $0..1$ write accesses, and when there is a read and a write access to the same domain entity, the read access has to occur first. These are the accesses that have impact outside the local transaction atomic execution.

The redesign of a functionality in the context of a decomposition corresponds to the application of a set of operations to a graph which represents the functionality execution, where the nodes represent the functionalities' local transactions and the edges the remote invocations between transactions.

Definition: Functionality Execution Graph. A functionality f redesign in the context of a monolith decomposition is represented by a graph g, where the nodes are local transactions, denoted by $g.lt$, and the edges remote invocations between local transactions, denoted by $g.ri$:

- $g.lt$ is the set of local transactions, such that:
  1) $\bigcup_{lt \in g.lt} lt.sequence.entities = f.sequence.entities$
  2) $\#\{lt \in g.lt : lt.type = pivot)\} \leq 1$
- $g.ri$ is the set of local transactions pairs that represent the remote invocations:
  1) $\forall_{(lt_i, lt_j) \in g.ri} \{lt_i, lt_j\} \subseteq g.lt$
  2) $\forall_{(lt_i, lt_j) \in g.ri} \neg \exists_{lt_{k \neq i} \in g.lt} (lt_k, lt_j) \in g.ri$
  3) The remote invocations define a partial order between the local transactions, denoted by $<_g$, and build using the transitive closure of the following initial elements, $\forall_{(lt_i, lt_j) \in g.lt} lt_i <_g lt_j$. Therefore, given $lt_i, lt_j \in g.lt$ if $lt_i <_g lt_j$ then lt_i executes before lt_j.

The redesign of a functionality in the context of a decomposition starts with its initial graph, which is generated from the functionality decomposition.

Definition: Initial Graph. The initial graph g_I of a functionality f has as vertices the local transactions lt associated to each one of the subsequences of f, $g_I.lt = \{lt \in LT : lt.sequence \in f.subsequences \wedge lt.type$ is not defined$\}$, and has as edges the pairs of local transactions associated with consecutive subsequences, $(lt_j, lt_k) \in g_I.ri$ iff $\exists_{0 \leq i < f.subsequences.size-1} : lt_j.sequence = f.subsequences[i] \wedge lt_k.sequence = f.subsequences[i+1]$. It is trivial to observe that the initial graph g_I is a well-formed graph of f.

A semantic lock is an intermediate state set by a compensatable local transaction, a write access, that is visible by the other functionalities, and that may eventually be undone.

Definition: Local Transaction Semantic Lock. Given an execution graph g of a functionality f, and one of its local transactions lt, $lt.sl$ denotes the domain entities with a semantic lock in lt, such that $lt.sl = \bigcup_{(e,m) \in lt.sequence} (lt.type = compensatable \wedge m = w)$.

Definition: Functionality Semantic Lock. Given an execution graph g of a functionality f, $g.sl$ denotes the domain entities with a semantic lock in g, such that $g.sl = \bigcup_{lt \in g.lt} lt.sl$.

Definition: Final Graph. A final graph g_F of a functionality f is a graph of f where all transactions have a type, $\forall_{lt \in g_F.lt} lt.type$ *is defined*, and all the transactions that follow the pivot transaction are retriable, given $lt_i \in g_F.lt : lt_i.type = pivot \implies \forall_{jt_j : lt_i <_{g_F} lt_j} lt_j.type = retriable$. Additionally, it is not possible to have a remote invocation between local transactions belonging to the same cluster, $\forall_{(lt_i, lt_j) \in g_F.ri} \{lt_i.cluster \neq lt_j.cluster\}$. Given that a graph has at most one pivot transaction, and in a final graph all transactions have a defined type, it is trivial to observe that all the transactions that do not occur after the pivot transaction should be compensatable.

Definition: Redesign Process. The redesign of a functionality f is a process that starts with its initial graph g_I and through the application of graph operations produces a final graph g_F, where, in a first step, the software architect will perform operations over the execution graph to redesign the execution flow of f, and, finally the architect will characterize the type of local transactions, such that the *SAGA* pattern is applied to the functionality f in the context of the monolith decomposition.

We propose three basic operations and a composed operation to support the redesign of a functionality. The basic operations are: *Sequence Change*, where the order by which the local transactions are invoked is changed; *Local Transaction Merge*, where two local transactions belonging to the same cluster are merged; and, *Add Compensating*, where a new local transaction is added when it is necessary to undo the changes done by local transactions. Additionally, we propose a composed operation, *Define Coarse-Grained Interactions*, where repetitive fine-grained interactions between two candidate microservices are synthesized into a single coarse-grained interaction.

By applying these operations, the software architect transforms the sequence of local transactions in the initial graph to a saga like interaction, either an orchestration or a choreography, where in the former case there is a cluster that coordinates the execution flow between the local transactions.

Definition: Sequence Change. Given a graph g of functionality f, three distinct local transactions, $lt_1, lt_2, lt_3 \in g.lt$ where $lt_1 \neq lt_2 \neq lt_3 \neq lt_1$, $lt_3 <_g lt_2$, and a remote invocation $ri = (lt_1, lt_2) \in g.ri$, it is possible to replace ri by $ri' = (lt_3, lt_2)$, such that g is transformed to $g' = (g.lt, g.ri \setminus \{ri\} \cup ri')$, a graph of f. It is trivial to observe that the transformed graph is a well-formed graph of f in the context of the decomposition, because lt_3 executes before lt_2 we can conclude that the resulting order continues to be a partial order and all local transactions are remotely invoked by at most one local transaction.

The *change sequence* operation is used to change the flow of execution of the functionality in the context of the decomposition and it is possible to apply when no local transaction in the invocation chain between lt_3 and lt_2 requires data produced by lt_2. For instance, to change the local transaction (hence the cluster) that is responsible to trigger the execution of another particular local transaction, which may be useful to centralize the control of execution in a microservice that coordinates the execution of other local transactions, and so reduce the transactional complexity behavior.

Definition: Local Transaction Merge. Given a graph g of functionality f and two local transaction $lt_1, lt_2 \in g.lt$, such that they belong to the same cluster, $lt_1.cluster = lt_2.cluster$, and they have adjacent executions, either (1) $(lt_1, lt_2) \in g.ri$

or (2) $\exists_{lt_i \in g.lt} : (lt_i, lt_1) \in g.ri \wedge (lt_i, lt_2) \in g.ri$, a new graph g' of f is produced, where, considering the two cases: (1) $g'.lt = g.lt \setminus \{lt_1, lt_2\} \cup lt_m$, where $lt_m.sequence = prune(concat(lt_1.sequence, lt_2.sequence))$ and $g'.ri = g.ri \setminus \{(lt_1, lt_2)\} \setminus \{(lt_o, lt_1) : (lt_o, lt_1) \in g.ri\} \setminus \{(lt_k, lt_l) \in g.ri : lt_k = lt_1 \vee lt_k = lt_2\} \cup \{(lt_o, lt_m) : (lt_o, lt_1) \in g.ri\} \cup \{(lt_m, lt_i) : (lt_1, lt_i) \in g.ri \vee (lt_2, lt_i) \in g.ri\}$; (2) $g'.lt = g.lt \setminus \{lt_1, lt_2\} \cup lt_m$, where $lt_m.sequence = prune(concat(lt_1.sequence, lt_2.sequence))$ or $lt_m.sequence = prune(concat(lt_2 .sequence, lt_1.sequence))$, and $g'.ri = g.ri \setminus \{(lt_i, lt_1), (lt_i, lt_2)\} \setminus \{(lt_k, lt_l) \in g.ri : lt_k = lt_1 \vee lt_k = lt_2\} \cup \{(lt_i, lt_m)\} \cup \{(lt_m, lt_l) : (lt_1, tl_l) \in g.ri \vee (lt_2, tl_l) \in g.ri\}$.

The *local transaction merge* operation is used when, in the redesign process, two local transactions become adjacent in the execution graph, and can be included into a single local transaction. From the transactional perspective, it is necessary to integrate their execution sequences, what is achieved with the prune function, and in the second case, is the software architect that decide the order by which the sequences are integrated. As result of applying this operation, the number of intermediate states in result of the distributed execution of the functionality is reduced.

Definition: Add Compensating. Given a graph g of functionality f, a new graph $g' = (g.lt \cup \{lt_c\}, g.ri \cup \{ri_c\})$ of f is produced, where $lt_c \notin g.lt$, $lt_c.sequence.entities = \bigcup_{lt_i \in g.lt} \{e \in entities(lt_i.sequence, w) : lt_i.cluster = lt_c.cluster \wedge lt_i.type = compensatable \wedge lt_i <_g ri_c[1]\}$, $\forall_{(e,m) \in lt_c.sequence} m = w$, $ri_c \notin g.ri$ and $ri_c = (lt_j, lt_c)$, where $lt_j.cluster \neq lt_c.cluster \wedge lt_j \in g.lt$.

This operation is used to create new local transactions that access some of the domain entities changed by other local transactions. It can be used to create the compensating transactions that are necessary for each compensatable transaction.

Definition: Define Coarse-Grained Interactions: Given a graph g of functionality f, two candidate microservices, represented by the clusters $c_1 \neq c_2$, two remote invocations $\{(lt_{11}, lt_{21}), (lt_{12}, lt_{22})\} \in g.ri$, where the remote invocations are between the given microservices, $c_1 = lt_{11}.cluster = lt_{12}.cluster \wedge c_2 = lt_{21}.cluster = lt_{22}.cluster$, and lt_{11} executes before lt_{12}, $lt_{11} <_g lt_{12}$, a new graph g' of f is produced by applying the basic operations *change sequence* and *local transaction merge*. First, *change sequence* operation is applied for lt_{11} and (lt_i, lt_{12}), to produce a new graph with remote invocation (lt_{11}, lt_{12}). Note that is possible to apply the operation, because $lt_{11} <_g lt_{12}$ and so there exists the remote invocation (lt_i, lt_{12}). Then, *local transaction merge* operation is applied to lt_{11}, lt_{12} to produce a new local transaction lt_{1m} which has remote invocations to lt_{21} and lt_{22}. Finally, *local transaction merge* operation is applied to lt_{21} and lt_{22} which results in lt_{2m} local transaction and a coarser-grained remote invocation (lt_{1m}, lt_{2m}). Note that this operation can be applied to any number of remote invocations between two cluster, in the given conditions.

This operation is used to create two coarse-grained local transactions, one in c_1 and another in c_2, by joining local transactions that are executed in those clusters, in order to reduce the number of remote invocations. It must be used when, after the automatically generated decomposition, the software architect realizes that there are several recurring fine-grained interactions between two candidate microservices, due to an object-oriented programming style in the monolith, which promotes the use of fine-grained invocations between the domain entities.

After the operations have been applied to the initial graph g_I of functionality f, the last step of the redesign is to produce a final graph g_F through the characterization of each one of the local transactions. Therefore, the software architect must select one transaction in the graph to be the $pivot$ transaction. Transactions that follow the pivot transaction are guaranteed to succeed are classified as $retriable$, and all other local transactions are classified as $compensatable$. The compensatable transactions that have semantic locks need to have at least one compensating transaction because some of the transactions that execute after it in the saga might fail.

III. Complexity Metrics

Given a monolith decomposition, the base metric [14] was defined in previous works, which measures the complexity associated with the migration of a monolith system to a microservices architecture. It considers the complexity of each functionality redesign for the overall complexity of redesigning the monolith system, due to relaxing the functionality execution isolation, because the redesign of a functionality has to consider the intermediate states introduced by the execution of other functionalities.

Definition: Functionality Complexity in a Decomposition. Given a candidate decomposition C of a monolith, the complexity associated with the migration of a monolith functionality f is given by

$$\sum_{s_i \in f.subsequences} \# \bigcup_{(e,m) \in s_i} \{f_i \in F \setminus \{f\} : (e, m^{-1}) \in prune(f_i.sequence)\}$$

Where f_i is a distributed transaction, it executes in more than one cluster, and m^{-1} represents the inverse access mode, $r^{-1} = w$ and $w^{-1} = r$.

The overall idea behind the metric is to count, for each subsequence of a functionality, executing inside a cluster, the impact domain entities accesses have. The impact of a write depends on other functionalities that read it, and, therefore, they may have to consider this new intermediate state, while the impact of a read depends on how many other functionalities write it, and, therefore, introduce new intermediate states to be considered by the functionality. This metric reflects how many other functionalities need to be considered in the redesign of a functionality, thus, how the functionality redesign is intertwined with others functionalities business logic redesign.

However, during the redesign process, while the functionalities are redesigned, the concepts of local transaction and remote invocation are introduced, which allows a refinement of the previous metrics, such that, during the redesign process, the software architect can have more precise values about the complexity.

Therefore, and because the metric will be used to inform the functionality redesign activity, we distinguish between the complexity of redesigning the functionality from the complexity that the functionality redesign adds to the redesign of other functionalities.

Definition: Functionality Redesign Complexity. The complexity of redesigning a functionality f, executed as a graph g, is the sum of the complexity of each one of its local transaction:

$$complexity(f) = \sum_{lt \in g.lt} complexity(lt)$$

The complexity of one local transaction depends on the number of semantic locks that are introduced, because each semantic lock corresponds to an intermediate state for which may be necessary to write a compensating transaction, and it also depends on the intermediate states set by other functionalities that the local transaction may have to consider in its reads. Note that, during the redesign of a functionality, some of the functionalities that f interacts with may not have been redesigned yet, and so, the metric should take into account both situations.

Definition: Local Transaction Redesign Complexity. The complexity of lt is given by the number of semantic locks implemented in entities of $lt.sequence$, plus the number of other functionalities that write in entities read in lt, or which have semantic locks in those entities:

$$complexity(lt) = \#lt.sl + \sum_{(e,r) \in lt.sequence} \#\{f_i \in F \setminus \{f\} : (e,w) \in writes(f_i)\}$$

where

$$writes(f_i) = \begin{cases} \{(e,w) : (e,w) \in prune(f_i.sequence)\} \\ \qquad\qquad\qquad \text{if } f_i \text{ not redesigned} \\ \{(e,w) : (e,w) \in g_i.sl\} \\ \qquad\qquad\qquad \text{if } f_i \text{ redesigned as } g_i \end{cases}$$

Note that when a functionality is redesigned some writes may not be considered, because if they belong to pivot or retriable local transactions, they will not introduce intermediate states, and so, the metric will provide a more precise value.

The redesign of a functionality impacts on other functionalities redesign complexity. For instance, if a semantic lock is created in one 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 accommodate the existence of the semantic lock. Hence, the cost of redesigning f_j depends on the amount of semantic locks created by f_i in entities that f_j access.

Definition: System Added Complexity. Given the redesign of a functionality f executed as a graph g, the system added complexity introduced by redesign g, is given by:

$$addedComplexity(f,g) = \sum_{lt \in g.lt} \sum_{f_i \in F \setminus \{f\}} \#(reads(f_i).entities \cap lt.sl.entities)$$

where

$$reads(f_i) = \begin{cases} \{(e,r) : (e,r) \in prune(f_i.sequence)\} \\ \qquad\qquad\qquad \text{if } f_i \text{ not redesigned} \\ \{(e,r) : \exists_{lt \in g_i.lt} (e,r) \in lt.sequence\} \\ \qquad\qquad\qquad \text{if } f_i \text{ redesigned as } g_i \end{cases}$$

The redesign of functionality f may introduce inconsistent states in the application when it has two or more semantic locks. However, this situation only occurs when the entities belong to different clusters, because inside one cluster the entities are updated simultaneously by ACID transactions. Hence, we consider that a functionality changes a cluster when it introduces a semantic lock in one of its entities. If we consider that a functionality f writes in more than one cluster, this behaviour may introduce inconsistency views for any other functionality f_i that reads two or more of the changed clusters. Therefore, any functionality f_i that reads domain entities in different clusters, previously changed by f, might encounter inconsistent states.

From a redesign point of view, the inconsistency state complexity is particular relevant for functionalities that only read and have a single local transaction for each cluster they access. We call queries to this type of functionalities.

Definition: Query. A query q is functionality which graph g has the following properties: (1) its local transactions are read only, $\forall_{lt \in g.lt} lt.sequence.mode = \{r\}$; and (2) they only access a cluster at most once, $\forall_{lt_i \neq lt_j \in g.lt} lt_i.cluster \neq lt_j.cluster$.

Note that, if there is a functionality that only has read accesses, it is possible, by applying the redesign operations, to generate an execution graph that is a query. We define the cost of implementing a query as the inconsistency state it has to handle.

Definition: Query Inconsistency Complexity. Given a query q, its inconsistency complexity is the sum of all the other functionalities that write in at least two clusters that q also reads:

$$queryInconsistencyComplexity(q) =$$

$$\#\{f_i \in F \setminus \{q\} : \#clusters(entities(prune(q.sequence)) \cap writes(f_i).entities) > 1\}$$

where $writes(f_i)$ is defined as in the local transaction complexity metric.

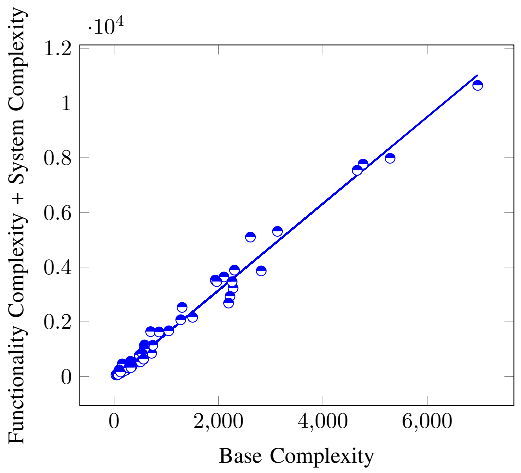

Fig. 1: Correlation between the base metric and the sum of Functionality complexity and System complexity in the LdoD system.

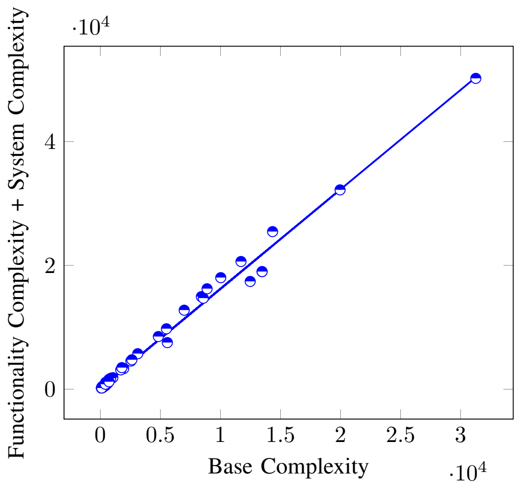

Fig. 2: Correlation between the base metric and the sum of Functionality complexity and System complexity in the Blended Workflow system.

IV. Evaluation

To evaluate the operations and metrics presented we analyzed two systems, LdoD[1] (122 controllers, 67 domain entities) and Blended Workflow[2] (98 controllers, 49 domain entities)[3]. Since the operations and metrics are applied in the context of a candidate decomposition, we used the expert decompositions of these systems. As the main goal of this work is to refine the existing complexity metric we start by showing that the base metric and the new metrics are correlated, when applied for the initial graph where every local transaction is typed as compensatable. In figures 1 and 2 we can observe the correlation graphs, for the monolith functionalities, where each point represent for one functionality its values according to the base metric and to the sum of the refined metrics, complexity and added complexity. It can be observed that the metrics are correlated.

A. Operations Evaluation

Firstly, the set of redesign operations were defined, and formalized, after an extensive experimentation by the expert that identified which changes have to be applied to the decomposition of a functionality to create a suitable SAGA implementation, while preserving its semantics.

To validate the proposed operations we started by filtering the functionalities in each system. The goal is to have two sets of functionalities, one with the functionalities that perform some create, update or delete operation (CUD operations), i.e, functionalities that write domain entities and that will be implemented using the *SAGA* pattern, and another with the functionalities that only read entities, i.e, functionalities that are queries and which implementation is done using other type of patterns, e.g. *API Gateway* pattern. Then, for each system, we performed a quartile analysis over the complexities in the CUD set where we got 4 distinct groups of functionalities, grouped by their complexity. We randomly picked a functionality from each group and after careful analysis of the source code we applied the operations to redesign the functionality for the given decomposition. The redesign goal was done to achieve a saga orchestration style as recommended in [13], to minimise the remote invocations between services and reduce the network latency effect.

In table I are, for each of the selected functionalities, and for the final execution graph, the number of transactions of each type, the total number of local transactions and the total number of accessed clusters. Additionally, it presents the sum of the two complexity metrics, for the initial and final graph. We can also observe that, to preserve the data dependencies in the functionality, it is not possible to apply the operations until the number of local transactions is equal to the number of accessed clusters.

In what concerns the local transactions types, one clear and obvious conclusion, since the complexity depends on the number of local transactions, is that the sum of the refined metrics increases with the number of transactions. We can also observe that the number of compensatable transactions impacts on the complexity. This is due to the fact that the existence of compensatable transactions involves the creation of semantic locks (if the access mode is write) and also the creation of more transactions to implement the compensating transactions logic needed in case of a transaction abort.

B. Complexity Metrics Evaluation

Table II shows the complexities for each functionality analyzed in the systems LdoD e Blended Workflow. We can observe that for both systems the reduction in the functionality complexity surpasses, in average, 50%. This shows the advantage of the proposed redesign operations and the refined

[1]data/ecsa2020 in https://github.com/socialsoftware/mono2micro
[2]https://github.com/socialsoftware/blended-workflow
[3]https://github.com/socialsoftware/edition

	Local Transactions					Metrics	
Functionality	C	P	R	Total	# Access Clusters	Sum for the g_I	Sum for the g_F
LdoD							
removeTweets	0	1	3	4	4	442	82
getTaxonomy	0	1	4	5	3	529	192
associateCategory	7	1	4	12	4	3470	1067
signUp	0	1	4	5	4	3861	376
Blended Workflow							
updateView	2	1	0	3	3	415	257
removeSequence ConditionToActivity	2	1	3	6	2	1110	455
addActivity	6	1	2	9	3	3323	1493
extractActivity	25	1	3	29	4	20628	5636

TABLE I: Operations performed and local transactions types in the functionalities of LdoD and Blended Workflow. C - Compensatable; P - Pivot; R - Retriable; # Clusters - number of accessed clusters.

metrics. On the other hand, we also observe the relation between the functionality *associateCategory* and *signUp*. Before redesign the *associateCategory* has a lower complexity value than *signUp*. However after the redesigning, and the application of the *SAGA* pattern, that relation is reversed and the complexity value for the *associateCategory* is higher than for *signUp*. This shows that the impact of the redesign operations is not the same and depends on the functionality business logic. Additionally, it shows an advantage of allowing the software architect to redesign the model that results from the automatic decomposition of the monolith, in particular the verification of whether the most complex functionalities, according to the base metric, can or not be significantly reduced.

By analysing the system added complexity values, in both system we got a significant reduction in the complexity values after the redesign, which allows us to provide the architect with more precise values on the impact the functionalities redesign has in the system.

Since the refined metrics separate the base metric into two different concerns, functionality complexity and added complexity to the system, we can do a more rich and precise analysis. For instance, only the functionality *associateCategory* has a non zero value in the LdoD system, which indicates that only this functionality, of the four functionalities analyzed, introduces complexity in to others functionalities redesign despite that in g_I all functionalities had a non zero value. A strong example is the functionality *signUp*, which at the beginning was considered to have the most impact on the system, ended up having no impact on the system since it does not create any semantic locks.

When analyzing both tables, it is visible the relation between the existence of compensatable transaction and a positive value for the system complexity, where the only exception is the functionality *removeSequenceConditionToActivity* in the Blended Workflow system that contains 2 compensatable transactions and 0 system complexity. After analysing the final redesing graph we noted that the 2 compensatable transactions were read only transactions. They are considered compensatable because, by definition, all the local transaction that do not occur after the pivot transaction are compensatable, but in this case they do not need a compensating transaction in case of a transaction abort. However, as previously noted in the relation between the number of compensating transactions and complexity, we can conclude that most of the compensating transactions require semantic locks.

Due to space restrictions, the Query Inconsistency Complexity (QIC) analysis is omitted. However its evalution on queries of both systems (LdoD and Blended Workflow) showed that, despite this new metric does not derive from the base metric, the complexity value from QIC increases as the base metric increases, they are correlated.

Going back to the research questions raised:

1) What set of operations can be provided to the architect such that the functionalities can be redesigned by applying microservices pattern?
2) Is it possible to refine the complexity value associated with the monolith migration when there is additional information about the functionalities redesign?

To answer them, first we have defined a suitable set of operations that the architect can use in the design stage in order to design functionalities in a microservices architecture. Secondly, by separating the base metric in two distinct metrics we can target different affected areas during the functionalities design and implementation, and we obtained more precise values for the functionality migration cost.

C. Threats to Validity

In terms of internal validity, the use of the expert decomposition has no impact on the validation conclusions, actually, to evaluate the metrics refinement and the operations, any candidate decomposition could be used. The validation of the operations was done to a small subset of functionalities, but a systematic method to select them was chosen and

Functionality	Initial FRC	Final FRC	% Reduction	Initial SAC	Final SAC	% Reduction
LdoD						
removeTweets	134	82	39 %	308	0	100 %
getTaxonomy	317	192	39 %	212	0	100 %
associateCategory	1803	662	63 %	1667	405	76 %
signUp	1490	376	75 %	2371	0	100 %
		Average:	54 %			94 %
Blended Workflow						
updateView	204	134	34 %	211	123	42 %
removeSequence ConditionToActivity	861	376	56 %	249	79	68 %
addActivity	1775	721	59 %	1548	772	50 %
extractActivity	13849	2930	79 %	6779	2706	60 %
		Average:	57 %			55 %

TABLE II: Functionality complexity and System complexity for the functionalities in the systems LdoD and Blended Workflow. FRC - Functionality Redesign Complexity; SAC - System Added Complexity.

functionalities with different levels of complexity were also chosen. Another threat to internal validity is that the redesign was done to follow an orchestration style for the functionalities sagas. However, considering that: (1) we are evaluating the applicability of the redesign operation; (2) evaluating whether the new metrics can provide a more precise value, this is not biased by following a orchestration style, though the complexity values reduction could be smaller, but precise anyway.

In terms of external validity, we believe that our conclusion can be generalized to the monolith systems that were implemented using a rich domain model, which is the case of the two analyzed systems, that were implemented using fine-grained object-oriented interactions.

V. Related Work

Previous work on the migration monolith systems to a microservices architecture [1], [7]–[9] and on the quality of microservices architectures [2]–[4], evaluate the candidate decompositions and the microservices architectures through metrics that focus on aspects like modifiability, cohesion, coupling, performance or even the size of clusters. However, they do not analyse the complexity associated with the effort of migrating the monolith, according to the candidate decomposition. Therefore, their focus is not in the complexity added to the functionalities business logic, as explained by the CAP theorem [6]. In this paper we address the effort required in the monolith functionalities redesign and the application of microservices design patterns.

Some research has been done on metrics and microservices patterns. In [15] a set of metrics is proposed to assess the architecture conformance to microservice patterns. Their metrics evaluate characteristics like independent deployability and shared dependencies between components. In [4] it is developed a system that evaluates microservices architectures conformance to a set of microservices design principles. For each principle a metric is defined. None of these research addresses the monolith functionality redesign cost using microservices patterns.

Although there are many proposals on how to decompose a monolith into a microservices architecture, as far as we know, only in [10] is proposed a tool that, besides providing visualisation of the decomposition, allows the creation of new microservices, move classes between microservices, and clone a class in several microservices, while recalculating a set of metrics on the decomposition quality. However, their modeling focus is not on functionality redesign.

The proposed redesign process, and metrics refinement, leverages on our previous work on the decomposition of monolith systems based on the identification of transactional contexts, to reduce the impact of transactional context changes on the functionalities behavior [12], and on a complexity metric for migration decomposition [14].

VI. Conclusions

This paper proposes a set of operations for the redesign of monolith functionalities given a decomposition on a set of candidate microservices. The redesign is guided by a set of metrics which calculate the complexity associated with functionalities business logic rewriting, due to the lack of isolation. The *SAGA* pattern is applied to the functionalities and the number of semantic locks is used to calculate the complexity. On the other hand, by dividing the complexity into two distinct metrics, it becomes possible to distinguish between the complexity inherent to the each functionality redesign, and the complexity added in the redesign of other functionalities. As an extension of these two metrics, we also propose a query inconsistency metric that measures the cost of applying the *API Gateway* pattern. As result of the evaluation, we observed that through the application of the operations a suitable execution flow of the functionality, following the *SAGA* patterns, is obtained. As expected, more operations are required when the complexity of the functionality, before the redesign, is higher, but we also observed that the percentage of complexity reduction depends on the business logic functionality.

References

[1] D. Athanasopoulos, A. V. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis. Cohesion-driven decomposition of service interfaces without access to source code. *IEEE Transactions on Services Computing*, 8(4):550–562, 2015.

[2] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Automatically measuring the maintainability of service-and microservice-based systems: a literature review. In *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement*, pages 107–115. ACM, 2017.

[3] Mario Cardarelli, Ludovico Iovino, Paolo Di Francesco, Amleto Di Salle, Ivano Malavolta, and Patricia Lago. An extensible data-driven approach for evaluating the quality of microservice architectures. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1225–1234, New York, NY, USA, 2019. Association for Computing Machinery.

[4] Thomas Engel, Melanie Langermeier, Bernhard Bauer, and Alexander Hofmann. Evaluation of microservice architectures: A metric and tool-based approach. In Jan Mendling and Haralambos Mouratidis, editors, *Information Systems in the Big Data Era*, pages 74–89, Cham, 2018. Springer International Publishing.

[5] Hector Garcia-Molina and Kenneth Salem. Sagas. *ACM Sigmod Record*, 16(3):249–259, 1987.

[6] Seth Gilbert and Nancy Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, February 2012.

[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*, pages 1–1, 2019.

[8] S. Klock, J. M. E. M. van der Werf, J. P. Guelen, and S. Jansen. Workload-based clustering of coherent feature sets in microservice architectures. In *2017 IEEE International Conference on Software Architecture (ICSA)*, pages 11–20, 2017.

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

[10] 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)*, pages 32–42, Sep. 2018.

[11] Evangelos Ntentos, Uwe Zdun, Konstantinos Plakidas, Daniel Schall, Fei Li, and Sebastian Meixner. Supporting architectural decision making on data management in microservice architectures. In *European Conference on Software Architecture*, pages 20–36. Springer, 2019.

[12] Luís Nunes, Nuno Santos, and António Rito Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In Tomas Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture*, pages 37–52, Cham, 2019. Springer International Publishing.

[13] Chris Richardson. *Microservices Patterns*. Manning Publications Co., 2019.

[14] Nuno Santos and António Rito Silva. A complexity metric for microservices architecture migration. In *Proceedings of the IEEE 17th International Conference on Software Architecture (ICSA 2020)*, pages 169–178. IEEE, 20202.

[15] Uwe Zdun, Elena Navarro, and Frank Leymann. Ensuring and assessing architecture conformance to microservice decomposition patterns. In *International Conference on Service-Oriented Computing. ICSOC. Lecture Notes in Computer Scienceg*, pages 411–429. Springer, 2017.