A Complexity Metric for Microservices Architecture Migration

Nuno Santos
Instituto Superior Técnico
Lisbon, Portugal
nuno.v.santos@tecnico.ulisboa.pt

***Abstract*—Monolithic applications tend to typically be difficult to deploy, difficult to upgrade and maintain, and difficult to understand. Microservices, on the other hand, have the advantages of being independently developed, tested, deployed, and scaled and, more importantly, easier to change and maintain. This thesis is focused on the migration from a monolithic to a microservices architecture. In our work we address two new research questions: (1) Can we define the cost of decomposition in terms of the effort to decompose a functionality, implemented in the monolith as an ACID transaction, into several distributed transactions? (2) Will the use of similarity measures that consider information about reads and writes and the sequences of accesses provide a better decomposition? To answer our first research question, we propose a complexity metric for each functionality, in the context of a particular decomposition, in order to provide a better insight into the cost of the decomposition. Regarding our second research question, we propose the experimentation with four similarity measures, each based on a different type of information collected from the monolith. We evaluated our approach with three monolith systems and compared our complexity metric against two industry metrics of cohesion and coupling. We evaluated our similarity measures against the complexity of the decomposition obtained. We also compared, for each system, our generated decompositions with the ones created by an expert of the system. We were able to correctly correlate the complexity metric with other metrics of cohesion and coupling defined in other research. For the second research question, we concluded that no single combination of similarity measures outperforms the other, which is confirmed by the existing research. We found that the developed tool can help on an incremental migration to microservices approach, which is actually defended by the industry experts.**

***Index Terms*—Microservices, Monolithic Application, Transactional Context, Architecture Migration, CAP Theorem**

I. Introduction

Microservices are designed around business capabilities such that they can be assigned to agile small cross-functional teams, which reduces the conceptual gap between business requirements, functionality implementation, service deployment and operation. This is achieved through the definition of independently deployable distributed services, which have well-defined interfaces, very often maintaining several versions of the interface to reduce inter-team communication, and using lightweight communication mechanisms [1]. Contrarily, monolithic applications have limitations in terms of agility, they have a large shared domain model through which all teams have to communicate, do not support different levels of horizontal scalability per service type, which compromises availability and fault-tolerance. Those are the main reasons that triggered the wide adoption of the microservices architecture, such that, well-known companies such as Netflix, Amazon and Ebay have decided to make the transition to microservices [2] and more existing monolith systems are being evolved to a microservices architecture.

However, the migration of monolith systems to microservices architecture raises new problems associated with the need to get microservices boundaries right, the complexity associated with asynchronous distributed computing, and the relaxing of atomic transactional behavior. Additionally, microservices architecture is not a silver bullet of software engineering and, therefore, it is important to evaluate whether it is worth migrating the monolith to microservices.

A. Context

This thesis follows a previous one [3] where a strategy was created to provide a microservice decomposition and a tool was developed as a proof of concept. The strategy is separated in three main stages:

1) Collect the callgraph of the monolith system using static analysis. The callgraph provides the method call chain (callees and callers of a method) which can be used to retrieve the domain entities accessed by each functionality. This was done using the JavaCallgraph[1] tool.
2) Define clusters using a similarity measure. The similarity is calculated by attributing a weight to the relationship between two domain classes. The weight from a class C1 to a class C2 is the quotient between the number of controllers that call both classes and the total number of controllers that call C1. An Hierarchical Clustering algorithm[2] is used to define the clusters, where each cluster corresponds to a microservice.
3) Present in a visualization tool the different partitioning hypothesis in the form of a graph, where the developer can analyse and assess each one.

B. Problem

Transaction management is more complex in a microservices architecture, because each service has its own database,

[1] github.com/gousiosg/java-callgraph
[2] docs.scipy.org/doc/scipy/reference/cluster.html

and the implementation of a functionality is scattered across several distributed microservices. On the other hand, functionalities in monolithic applications are simpler to implement because they have a single database and their transactional manager provides ACID properties which the developers can rely on, such that they can focus on the implementation of the business logic.

Therefore, when introducing microservices, it is necessary to decide on the functionalities trade-off between consistency and availability, as explained by the CAP theorem [4], where the decision to have strict consistency can only be achieved in a distributed transactions context, by the application of a two-phase commit protocol [5] which does not scale to support many replicas.

Consequently, in a microservices architecture it is necessary to decide what is the attainable level of consistency to associate with functionality, given the requirements on availability. Currently, the practice of microservices development uses patterns to handle this problem, like the saga pattern [6]. However, the use of these techniques is not free of cost, both in terms of the system behavior and its implementation:

- From an end-user perspective, there is an impact on the perceived behavior of the system, because intermediate transactional states may be observed due to the relaxed consistency and the occurrence of failures. Therefore it is necessary to do requirement analysis activities with some of the application stakeholders.
- Functionality implementation becomes more complex, because the developer, besides focusing on the business logic, also has to consider all the situations of possible faults in the microservices that implement a functionality, and write code that handles them, like the implementation of compensating transactions. Additionally, the functionality business logic easily become intertwined with the handling of all faults, which make its implementation more cumbersome.

C. Research Questions

In this thesis we address the problem of microservices identification in a monolith while managing the impact that the relaxing of atomic transactional behavior has on the redesign and implementation of the new system from the monolith.

Relaxing the transactional behavior associated with business transactions in a monolith system impacts on the application functionality due to the change in the consistency of information, which may require the redesign of functionalities. On the other hand, the implementation of the distributed business transactions is costly because, to achieve the desired level of consistency, it is necessary to intertwine distributed systems techniques with the functionality business logic.

Interestingly, this has been neglected in the literature where most proposals ignore its impact on the migration process, both in the identification of candidate microservices and in the migration implementation. This situation has been coined in [7] as the *Forgetting about the CAP Theorem* migration smell, which states that there is a trade-off between consistency and availability [8].

In the previous thesis, decompositions were created based on accesses only and validation was done with several systems based on the expert decomposition. This validation didn't allow to measure the quality of the decomposition which was focused in the comparison with the expert. With our approach we intend to improve on the previous work and respond to the following research questions:

1) Is it possible to calculate the cost associated with the migration to a microservices architecture due to the introduction of relaxed consistency into the business behavior?
2) Which similarity measures are more effective in the generation of a candidate microservices decomposition in terms of the cost of migration?

The visualization and modeling tool is used for the evaluation of the research questions. Therefore, the application of the approach is done to three monolith systems and the result decompositions are analysed in the visualization tool. Additionally, decompositions done by domain experts of the three systems were also introduced in the visualization tool to allow the comparison with the candidate decompositions.

D. Contributions

This thesis leverages on a previous work which addresses this problem by defining a similarity measure between two entities in terms of the functionalities that access them. In this thesis we extend this migration approach such that:

- Extracts from the monolith information about its business transactions, which domain entities they access, categorize the accesses in terms of reads and writes, and the sequences of domain entities accesses for each business transaction;
- Defines several similarity measures that are used to identify microservices candidates;
- Define a metric to measure the quality of a microservices systems in terms of the cost associated with having non atomic business behavior;
- Provide a visualization tool to navigate and analyse the candidate decompositions and allow the comparison between several decompositions using the previous metric;
- Enhance the visualization tool with modeling capabilities that supports the manipulation of a candidate decomposition informed by the real-time recalculation of the metrics after each change is done to the model.

The results of this thesis are the following:

- Identification of what are the best similarity measures to generate a set of candidate microservices from a monolith system.
- Definition of a complexity metric for microservices systems in terms of the impact on functionality partitioning and its implementation.

II. Related Work

Most of the articles follow a strategy similar to ours that starts by gathering information about the application entities, mainly by using analysis techniques such as static and dynamic analysis. A clustering algorithm, or some form of grouping methodology, is then used to create clusters of entities which correspond to microservices. Visualization capabilities for the microservice decomposition are occasionally found and often don't provide the ability to change the decomposition, as we do by cutting it and changing the granularity level. Only [9] provides a richer visualization, where it is possible to fine-tune the entities in the proposed microservice. In our approach we are interested in distinguishing clusters based on the transactional context by considering the data accesses. The idea is that the entities involved in a single transaction should be in the same microservice. From the approaches studied we found two that can relate with ours: the data dependency metric in [10] considers the data accesses in BPMN activities, and the consistency constraint coupling criteria in [11] considers that entities in the same database transaction should be grouped together, but they don't mention how that coupling criteria is measured. In [12] they compared their approach against Service Cutter [11] and concluded that Service Cutter requires a detailed and exhaustive specification of the system, together with ad-hoc specification artifacts associated with coupling criteria, and the availability of that documentation is arguable, where they present their approach as only requiring lightweight input that can be automatically generated. Other approaches are based on system requirements and use cases [13], on semantic similarity [12] [14], or on workload information [15] [16].

Metrics for microservices are still relatively new and many descend from existing metrics for service oriented architectures. Assessing software quality through the creation of metrics exists for a long time, and new metrics need to be adapted as new architectures appear. Most articles present metrics of complexity, cohesion and coupling. The IEEE Standard Computer Dictionary defines complexity as the degree to which a system or component has a design or implementation that is difficult to understand and verify [17]. Cohesion measures the degree to which the elements of the system belong together [18]. Coupling is a measure of the extent to which interdependencies exist between software modules [19].

III. Solution Architecture

Considering the impact associated with the relaxing of the transactional behavior in monolith functionalities when they are migrated to a microservices architecture, we intend to estimate what is the cost of the migration, such that the architect can take informed decisions when analyzing candidate decompositions. Additionally, we intend to experiment with similarity measures to study which aspects of microservices, and microservices interactions, may impact on the cost of the implementation of the microservice architecture, like the difference between read and write accesses, and the sequence of invocations between microservices, that can be used to generate decompositions with different costs. Therefore, we expect that when applying these similarity measures to the clustering algorithm, we generate different candidate decompositons, which can be analyzed in terms of their quality.

A decomposition has an impact on the monolith functionalities, due to the relaxing of consistency, and the level of impact varies depending on the type of access. For instance, when a functionality writes an entity and this entity is read in the context of other functionalities, which in this decomposition has to be implemented as distributed transactions, it impacts on the business logic of these functionalities, and the cost of migrating this functionality increases with the number of other functionalities that read the entity because each one of them may need to be redefined to cope with a weaker model of consistency. Note that the functionalities that read the new written data need to consider the possibility that it can be outdated, and have to change their business logic accordingly. On the other hand, when a functionality reads an entity that is written by other functionalities, that have to be implemented as distributed transactions in this decomposition, it may impact on the business logic of the functionalities that write the entity, and, analogously to the previous case, this impact increases with the number of transactions that write the entity. In this case, the functionalities that write need to consider what level of consistency they want to provide.

Note that the cost associated with the impact of reads and writes can have different types of implementation. For instance, one solution would be to replicate the domain entity, where the cost corresponds to the implementation of the replication and the synchronization between replicas, which has an implementation cost associated with the number of replicas and microservices involved, as well as operation costs, which correspond to the frequency of the synchronization and the impact on the behavior perceived by the end-user in terms of data consistency. Another solution would be to implement explicit invocations between the microservices whenever the information is required, which also has implementation and operation costs, for example, due to the handling of failures in the distributed transaction.

Beyond the type of access, we also consider the sequence of 'hops' between the microservices that implement a functionality. At first sight, we can consider that a higher number of 'hops' the more complex the implementation of the microservice is. However, this may be tuned by the type of accesses, if, for example, the implementation of functionality accesses two microservices alternately several times and one of those microservices is always accessed to read the same domain entity.

In this section we describe our strategy to the generation of candidate microservice decompositions and our developed metrics to assess cost of migration to the decomposition. We start by explaining how we gather information from the monolith, and then we present our complexity metric. After that we explain what are the different similarity measures we used to group entities, and in the end we present our visualization tool which, given a decomposition, shows the

above metrics, and serve as a proof of concept.

Our workflow to create microservices leverages on the process presented in I-A.

We start by collecting data from the monolith system using static analysis, where we, instead of collecting only the accesses, gather the reads and writes made to domain entities, and the sequence of those accesses. We also improve the accuracy of the data collected, by capturing method calls made inside Java streams.

In the clustering part we use the hierarchichal algorithm previously used, which requires as input a similarity matrix of entities and return as output a group of clusters, where each cluster is a group of entities. Each entry in the similarity matrix represents a value of similarity between two entities and is calculated using a combination of the new similarity measures that consider the type of access and the sequence of accesses.

In the visualization section we present different views depending on what we are analysing (decomposition, functionality or entities). We also show the complexity metrics and recalculate them as we change the decomposition. It is also possible to compare any two decompositions, where the main interest is comparing our generated decomposition with an expert one.

A. Microservices Architectural Metrics

1) Complexity: To answer our first research question, we propose a complexity metric for each functionality, which in the context of the systems we are going to analyse is represented by a Spring-Boot controller, in the context of a particular decomposition, in order to provide a better insight into the cost of the decomposition. This metric is concerned with the complexity of the functionalities when migrated to the new microservices architecture, due to the change in the level of consistency of the original monolith domain entities. The information used for this metric is the sequence of entities accessed during the transaction and the type of each access. For each microservice accessed (hop) in the execution of a functionality, we attribute a weight to it based on the entities accessed in that microservice. Therefore, we calculate for each entity the cost of reading or writing it, depending on whether the entity is read or written in the context of the functionality. According to our rational for the complexity of reading, the cost of reading an entity is the percentage of other distributed functionalities that write that entity. On the other hand, the cost of writing an entity is the percentage of other distributed functionalities that read that entity.

Complexity of a Functionality For each cluster accessed (sub) during the execution of a functionality f in a decomposition d, we sum the complexity of accessing each cluster, in the context of the sequence of accessed clusters by $sequence(f, d)$. If there is only one cluster (monolith situation), the complexity of the functionality is 0.

$$\sum_{sub \in sequence(f,d)} complexity(f, d, sub) \tag{1}$$

The complexity of accessing a cluster in the context of the sequence sub is represented by the size of the union of the complexities of each entity accessed by the functionality in that cluster.

$$\# \cup_{(e,m) \in sub} complexity(f, d, (e, m)) \tag{2}$$

The complexity of accessing an entity depends on whether the entity is being read or written.

The complexity of reading an entity in the context of the functionality, is the set of other distributed functionalities that write it, where a distributed functionality is a functionality that accesses more than one cluster. If more distributed functionalities write the entity then the complexity of reading it will be high, because there will be more business logic to be redesigned, one for each functionality.

$$\{f' \neq f : distributed(f', d) \wedge (e, w) \in flat(sequence(f', d))\} \tag{3}$$

On the other hand, the complexity of write is given by the set of other distributed functionalities that read it. Similar to the complexity of reads, the rational also applies to the complexity of writes, because it measures the amount of business logic that needs to be redesigned due to the decomposition.

$$\{f' \neq f : distributed(f', d) \wedge (e, r) \in flat(sequence(f', d))\} \tag{4}$$

B. Similarity Measures

A similarity measure represents the distance between two entities. It provides information about how coupled the entities are. In our case, a high value of similarity (value=1) means that two entities are highly correlated and therefore should be in the same microservice, while a value close to 0 means that the entities should be in different microservices. In our approach, this information is given as input to a hierarchichal clustering algorithm in the form of a similarity matrix, where each entry (x, y) represents the distance between two entities $(entities[x], entities[y])$. The similarity values are normalized (between 0 and 1).

Regarding our second research question, we propose the experimentation with four similarity measures, each based on a different type of information collected from the monolith. This measures represent the input parameters that are given to the clustering algorithm to generate candidate monolith decompositions. We intend to combine the four similarity measures and evaluate what are the weights for each element of the combination that generate a decomposition with the lower complexity cost.

Access:

$$\frac{\#\{functionalities(e_1) \cap functionalities(e_2)\}}{\#functionalities(e_1)} \tag{5}$$

In this measure we attribute a similarity value to each pair of entities based on indistinct accesses. The value of the similarity is the number of functionalities they have in common (that access both), divided by the functionalities that access the first one.

Read:

$$\frac{\#\{functionalities(e_1, r) \cap functionalities(e_2, r)\}}{\#functionalities(e_1, r)} \quad (6)$$

In this measure, the value of the similarity is the number of functionalities that read both entities divided by the functionalities that read the first one.

Write:

$$\frac{\#\{functionalities(e_1, w) \cap functionalities(e_2, w)\}}{\#functionalities(e_1, w)} \quad (7)$$

In this measure, the value of the similarity is the number of functionalities that write both entities divided by the functionalities that write the first one.

Sequence:

$$\frac{\sum_{f \in F} countSequence(e_1, e_2, f)}{maxNumberOfConsecutives} \quad (8)$$

In this measure, the value of the similarity is the number of times that e_1 followed by e_2 or the contrary appears in the functionality f sequence of accesses. To normalize all values, we divide each one with the biggest numerator, represented by $maxNumberOfConsecutives$. Unlike the previous measures, this one is symmetrical, meaning that the similarity between e_1 and e_2 is equal to e_2 and e_1.

C. Visualization and Modeling Tool

A tool was developed as a proof of concept, in the form of a web application, implementing the responsibilities for decomposition generation and analysis. The backend was developed with Spring-Boot and the frontend with ReactJS. The tool requires as input a data collection file that represents the codebase of the monolith. This file, which is obtained from the result of the data collection phase, is an intermediate data representation and contains internal information about the invocation tree inside the system. Note that, each Spring-Boot controller corresponds to a functionality. After a codebase is created, we are able to perform three operations to it.

First, we are able to manipulate controller profiles, which means that we can choose which controllers are being considered when calculating the similarity measures and creating the decomposition. For example, we can exclude setup and/or teardown controllers that may have a high impact on the decomposition but are only invoked during setup and shutdown of the system, and in that situation they can be invoked in sequence.

Second, we are able to generate a dendrogram that represents all the possible decompositions given a specific set of similarity measures. From a cut in the dendrogram results a decomposition. There are a number of parameters needed to create a dendrogram, which are the controller profiles that will be used, the linkage type for the hierarchichal algorithm, and the weight percentage for each of the four similarity measures available (access, read, write, sequence) which sum should be 100%.

The third operation is the creation of expert cuts. An expert cut is a decomposition that is created manually by the system owner/developer and represent what the expert intends to be the ideal decomposition of the system. This decomposition is used in the external evaluation to assess the quality of the tool-generated decomposition.

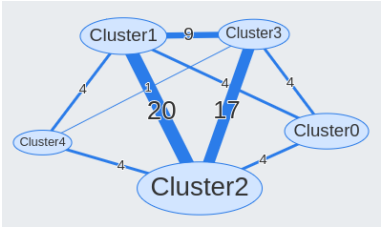


Fig. 1: Cluster view presenting clusters and the relations between them, for a generated decomposition

After performing a cut in the dendrogram, a decomposition is shown, as seen in figure 1, in the form of a graph where each node is a microservice, and the edges show the functionalities that access both adjacent nodes. By clicking or hovering the nodes we are able to see what entities are inside the microservice.

Besides this view with the decomposition (Clusters View), there is also the Transaction View and the Entity View.

In the Transaction View we can choose a functionality and observe the clusters it accesses and its complexity. By clicking or hovering the controller node we are able to see the entities that it accesses, the cluster nodes show the entities inside the cluster, and the edges show the entities the controller accesses inside that specific cluster. There is also the possibility to see the sequence of accesses between the clusters, either in the form of a graph or a table.

In the Entity View we can observe what are the clusters that take part in the functionalities the entity also participates. Clicking the entity shows the functionalities that access it, each cluster node show the entities inside it, and the edges show the functionalities that access both the cluster and the entity. Note that the cluster in which the entity is contained is not presented in the diagram, because we are interested in analysing the distributed functionalities the entity is involved in.

Regarding the metrics, we can see the complexity, cohesion and coupling metrics in the Clusters View for each cluster and the complexity in the Transaction View for each functionality. Additionally, we also have the complexity, cohesion and coupling values for the decompositions.

The tool also allows to compare any two decompositions, which is particularly helpful when we compare our generated decomposition with one created by an expert. The comparison is made using pairwise comparison, meaning that, given two entities, we compare if they are or not in the same cluster, for both decompositions.

This tool is essentially a visualization tool, but it also provides some modeling capabilities. The main ones are the manipulation of a candidate decomposition and the real-time recalculation of the metrics after each change is done to the

model. In relation to the manipulation capabilities, shown in figure 2, there are four main operations to manipulate the graph: rename, merge, split and transfer.

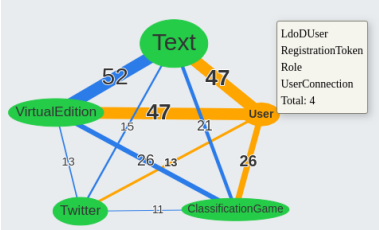


Fig. 2: Manipulated decomposition

We also developed an analyser that iterates through all the possible combination of similarity measures. As a result we obtain all possible decompositions and are able to find the similarity measures that provide better results, either in terms of comparing with the complexity metric or when compared with an expert decompositon, which allows to compare the qualities of the results of applying different similarity measures.

IV. Evaluation

To validate our approach we used three monolith systems, LdoD[3] (122 controllers, 67 domain entities), Blended Workflow[4] (98 controllers, 49 domain entities) and FenixEdu Academic[5] (856 controllers, 451 domain entities).

All applications are layered and client server web applications, where Spring-Boot controllers implement services by accessing a rich object domain model implemented using Fénix Framework.

A. Complexity Metric

Answering our first research question, we evaluate our complexity metric through the analysis of its correlation to a cohesion and a coupling metric. These two metrics are similar to those encountered in other research such as [20] [19] [21], and therefore provide an evaluation of our metric against the research state of the art.

This set of validation metrics were created to answer the question: Does a decomposition that minimizes the transactional complexity also has good results in terms of modularity?

Cohesion: Cohesion refers to the internal grouping of entities, and the reasoning behind it can be understood as the following: everytime an entity is accessed in a functionality, all the other entities from the same microservice should also be acessed during the course of the functionality, showing that those entities implement the same responsibility and should be in the same microservice.

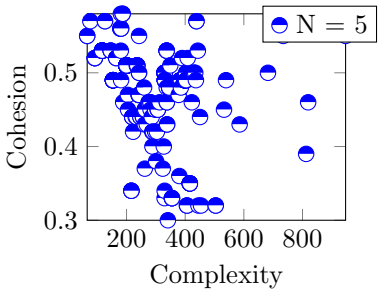
Cohesion of a Cluster: We measure the cohesion of a cluster c as the average percentage of entities accessed for each functionality that accesses the cluster. A cluster has maximum cohesion (cohesion = 1) if every functionality that accesses it does in fact access every entity inside the cluster.

$$\frac{\sum_{f \in functionalities(c)} \frac{\#\{e \in c.entities : e \in f.sequence.entities\}}{\#c.entities}}{\#functionalities(c)} \tag{9}$$

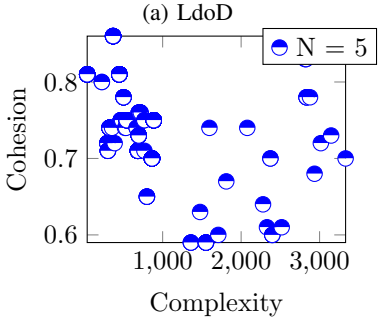
Our cohesion metric is related with the Lack of Cohesion in Methods metric presented in [21], where the methods represent our functionalities, and the instance variables our entities.

Cohesion of a Decomposition: Measures the cohesion of a decomposition d as the average cohesion of its clusters.

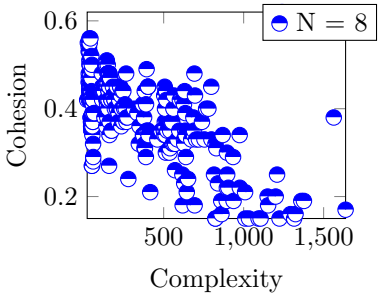
$$\frac{\sum_{c \in d.clusters} cohesion(c)}{\#d.clusters} \tag{10}$$



(a) LdoD



(b) Blended Workflow



(c) FenixEdu Academic

Fig. 3: Correlation between the Complexity and Cohesion metric, for the three monolith systems

Figure 3 shows the correlation between the complexity and cohesion values for the three systems, where each dot corresponds to a decomposition generated for a particular combination of similarity measures in a total of 286 combinations. All the combinations of values have an interval which is a multiple of 10% and the same number of clusters,

[3] github.com/socialsoftware/edition
[4] github.com/socialsoftware/blended-workflow
[5] github.com/FenixEdu/fenixedu-academic

5, for LdoD and Blended Workdflow, and 8 for FenixEdu Academic. From these figures we can denote a slight relation between the values, specially in FenixEdu Academic, where a higher complexity is associated with a lower cohesion. On the other hand, we can conclude, from the analysis of the graphs, that low complexity seems to imply high cohesion but that the inverse is not necessarily true, because there are some decompositions with high complexity and high cohesion, even though they may be outliers. This is not unexpected because the complexity of migrating the functionalities is not directly correlated with the cohesion, a functionality may not need to access all entities of a cluster to have low complexity, for instance if the cluster has a large number of entities. Note that in the case of the monolith, which has complexity 0 and the cohesion is low, provides the insight that the study of the possible correlation between cohesion and complexity should be done in a context where there are several clusters. Overall, we can conclude that for two decompositions with the same complexity the value of cohesion may be used to choose the one that follows the single responsibility principle.

Coupling: Coupling refers to the external distance between groups of entities and it can be used to determine how isolated a microservice is.

Coupling of a Cluster: Measures the level of dependency between clusters. If a cluster c has a high value of coupling, it means that it accesses many entities from other clusters. It is defined as the average level of coupling to the other clusters. Our metric is only applicable if the decomposition has more than one cluster because if there is only one cluster (monolith situation) the coupling value is irrelevant.

$$\frac{\sum_{c' \in d.clusters \ , \ c' \neq c} coupling(c, c', d)}{d.clusters - 1} \tag{11}$$

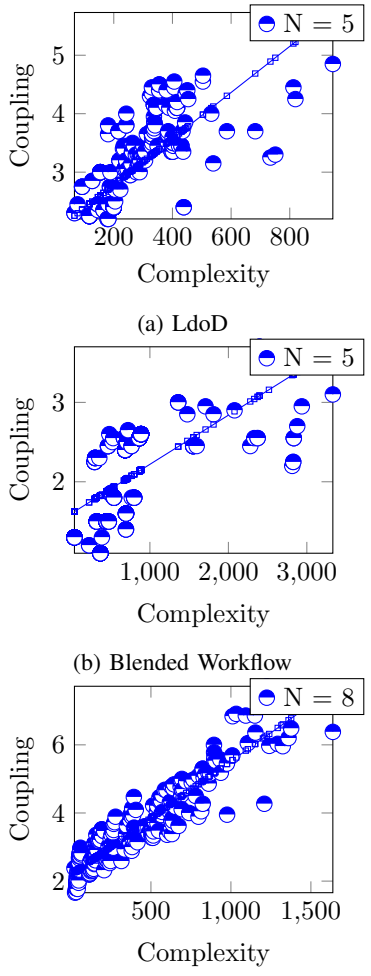
The level of coupling between any two clusters c_1 and c_2 is the number of entities of c_2 accessed by c_1. Note that we define accesses between clusters from the sequence of a functionality, i.e., if a functionality accesses an entity from c_1 and then an entity from c_2, we consider that this is an access from c_1 to c_2 (a hop between the clusters).

$$\# \cup_{f \in F} clusterDependencies(c_1, c_2, d, f) \tag{12}$$

Our coupling metric is very similar to the Weighted Extra-Service Outgoing Coupling of an Element metric showed in [19], defined as the weighted count of the number of system elements not belonging to the same service that are used by this element, where instead of implementation elements we consider clusters. It is also similar to the Coupling Between Object Classes metric in [21], defined as the count of the number of other classes to which it is coupled. This relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. This can be correlated to our metric where a method corresponds to a cluster, and an instance variable to an entity.

Coupling of a Decomposition: Measures the coupling of a decomposition d as the average coupling of its clusters.

$$\frac{\sum_{c \in d.clusters} coupling(c, d)}{\#d.clusters} \tag{13}$$



(a) LdoD

(b) Blended Workflow

(c) FenixEdu Academic

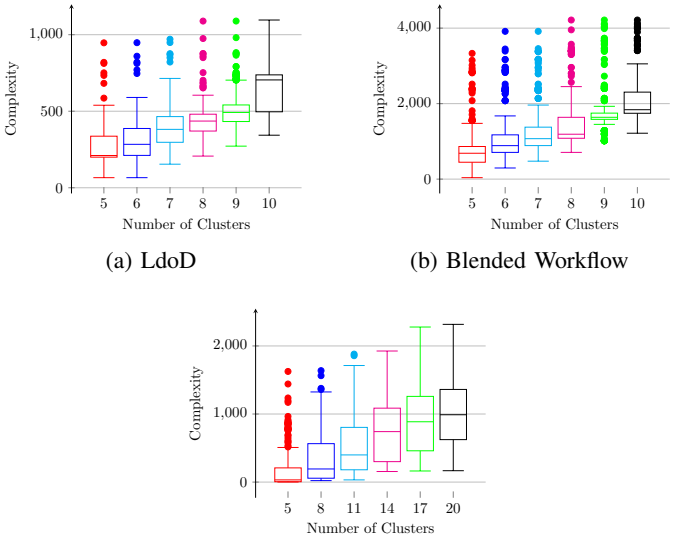
Fig. 4: Correlation between the Complexity and Coupling metric, for the three monolith systems

Figure 4 show the correlation between the complexity and the coupling metrics for the three systems, where each point is a decomposition generated by a combination of similarity measures, composed of 5 clusters for LdoD and Blended Workflow, and 8 clusters in FenixEdu Academic. The figures also contain a linear regression line to better illustrate the correlation. In this figures we can clearly establish a relation between the metrics, which is the higher the complexity the higher the coupling. Decompositions with a high complexity tend to have a high number of distributed transactions for each functionality and this leads to more interactions between clusters, which is what the coupling criteria measures.

From these two comparison (cohesion and coupling) we can conclude that our complexity metric behaves as expected when compared with other metrics.

B. Decomposition Complexity

Answering our second research question, we evaluate our similarity measures against the complexity obtained from each combination.



(a) LdoD

(b) Blended Workflow

(c) FenixEdu Academic

Fig. 5: Complexity of Decompositions, for the three monolith systems

Figure 5 shows the complexities of all decompositions generated for the 286 combinations of similarity measures and different number of clusters for the three monolith systems, from 5 to 10 in LdoD and Blended Workflow, and 5 to 20 with intervals of 3 for FenixEdu Academic. Each point represents a decomposition created by a specific combination of the similarity measures. The box in each graphic contains half of the points (two quartiles) and the line in the middle represents the median. Outside the box there's also the upper and lower bound, and the outlier points. From this figures we can conclude that the complexity metric grows as the number of clusters increases, which is understandable because our metric takes into account the size of the clusters access sequence by functionalities, and when the number of clusters increases the implementation of functionalities gets split into more clusters. We also observe, from the size of the boxes, that many combination of similarity values lead to very similar values of complexity, which indicates that there isn't a unique solution to the values of similarity, that leads to the best decomposition in terms of our complexity metric.

C. Functionality Complexity

For a certain decomposition, we analyse the disparity between the complexity of its functionalities. This information is particularly helpful to understand what are the most complex functionalities in the decomposition.

We calculated the complexities of the functionalities, for some representative combinations of similarity measures, for the three monolith systems. The decompositions were the following:

- Lowest Complexity (LC) is the decomposition with the lowest value for the complexity metric, for any combination of the similarity measures
- Combination of Reads and Writes (RW) is the decomposition with the lowest value for the complexity metric, for any combination of the read and write similarity measures, while the other measures (access and sequence) are fixed at 0%
- Only Writes (W) is the decomposition generated using only the write similarity measure
- Only Reads (R) is the decomposition generated using only the read similarity measure
- Only Accesses (A) is the decomposition generated using only the access similarity measure
- Only Sequence (S) is the decomposition generated using only the sequence similarity measure

We observed that the majority of the functionalities have a similar value of complexity, with only a small amount of outliers increasing the overall complexity.

Complexity	Functionality
1886	AdminController.loadTEIFragmentsStepByStep
1753	AdminController.loadTEIFragmentsAtOnce
986	AdminController.generateCitations
422	SignupController.signup
370	VirtualEditionController.editVirtualEdition

TABLE I: Functionality Complexity Outliers, for the Lowest Complexity Decomposition, N = 5, LdoD

Table I shows the functionalities that have the higher value of complexity, for the LC decomposition, with 5 clusters in LdoD. We can observe that the first two functionalities are detached from the rest and they correspond to functionalities that are executed when booting the system, which can, actually, execute in a non-concurrent context. Therefore, our tool provides a feature that allow the definition of profiles of functionality, such that it is possible to exclude some functionalites which, for instance, have high complexity but are considered not to occur frequently, from input to the hierarchical clustering algorithm.

		Similarity Measures			
	Complexity	Access	Write	Read	Sequence
Lowest Complexity	65.89	30	30	40	0
Reads and Writes	199.95	0	50	50	0
Only Writes	199.95	0	100	0	0
Only Reads	219.34	0	0	100	0
Only Accesses	242.53	100	0	0	0
Only Sequence	946.87	0	0	0	100

TABLE II: Analysis of similarity measures for a set of decompositions, with 5 Clusters, in LdoD

From table II we can conclude that there is no unique combination that leads to the lowest complexity. Using only each measure, we are able to conclude that using only the sequence measure always provided the worst result in terms

of complexity, and there isn't any supremacy of the other measures when each one is used alone. The results also show that using a combination of measures provided better results than using only one measure in terms of complexity, and that combination varies from case to case. Therefore, the complexity metric can drive the identification of the best decompositions, instead of trying to find a winner similarity measure.

D. Expert Decomposition

As part of the external evaluation, we compare a certain decomposition generated by us, with one created by an expert of the monolith system. Considering the expert decomposition as a source of truth help us to assess the relevance of our approach. For this part we were able to obtain an expert decomposition for the LdoD and Blended Workflow systems.

We do a pairwise comparison between the two decompositions, meaning that, for each pair of monolith entities, we count the following:

- True Positive (tp) if both entities belong to the same cluster in both decompositions (expert and ours)
- True Negative (tn) if the two entities belong to different clusters in both decompositions
- False Positive (fp) if both entities belong to the same cluster in our decomposition, and in different clusters in the expert
- False Negative (fn) if the two entities belong to different clusters in our decomposition, and belong to the same in the expert

With this information we calculate the measures of accuracy, precision, recall, specificity and f-score, defined below, which indicate the level of similarity between the two decompositions.

We observed that the expert decomposition is not the one with the lowest complexity. We also analysed the other decompositions that had lower complexity than the expert, and concluded that most of them have one large cluster and other smaller clusters when the number of clusters is relatively small. Although this decompositions are not fit for a complete migration to microservices, they provide a crucial help for an incremental migration, because they have one large cluster, which we can consider as the monolith core, and a number of smaller clusters, that indicate the first services to be splitted from the monolith. When we increase the number of clusters generated, the clustering algorithm starts to break the single large cluster meaning that we can also provide a complete migration to microservices if we increase the number of clusters high enough. Increasing the number of clusters is particularly helpful when executing an incremental migration, because we can propose new services as the number of clusters increases. Interestingly, this is the approach recommended by microservices experts on the migration of a monolith to a microservices architecture [6, Chapter 13].

E. Summary

As a result of our evaluation, we summarize our conclusions as follows:

- Our complexity metric has a correct correlation with other metrics of cohesion and coupling defined in other research;
- Regarding the similarity measures, we are able to conclude that the sequence measure provides the worst results in terms of complexity, but there is no single combination that outperforms the other, which is confirmed by the existing research that, frequently using different similarity measures, report good results in their validation;
- The complexity metric correctly identifies what are the most complex functionalities in a decomposition;
- The expert decomposition is not the one with the lowest complexity, because the lower complex decompositions tend to identify small microservices while preserving a large cluster that continues to behave as a monolith. This leads to the idea that the tool can help on an incremental migration to microservices approach which is actually defended by the industry experts.

Going back to the research questions we defined in section I-C:

1) Is it possible to calculate the cost associated with the migration to a microservices architecture due to the introduction of relaxed consistency into the business behavior?
2) Which similarity measures are more effective in the generation of a candidate microservices decomposition in terms of the cost of migration?

Answering the first research question, we were able to calculate the cost associated with the migration to a microservices architecture, with the introduction of the complexity metric. Answering the second research question, we concluded that there isn't a unique combination of similarity measures that are more effective in the generation of candidate microservices decomposition in terms of the cost of migration.

F. Limitations

Currently, there are some limitations of our approach as described below:

- Static analysis is used to collect data from the monolith. This analysis may not be the most accurate when gathering the entities accessed, as for example, dynamic analysis can be, because only during runtime is possible to determine what entities are actually accessed. Therefore, it is an open research question whether by collecting data through dynamic analysis we can generate decompositions with lower complexity;
- Only a narrow spectrum of systems are considered (Spring-Boot applications that use the Fénix Framework). It would be interesting to verify our results on other systems that don't use the same technology.

V. Future Work

- Explore new similarity measures and microservices architectural metrics. Define metrics for the different qualities associated with a microservices architecture.
- Collect new information on the monolithic system using dynamic analysis. Currently only static analysis is used.
- Analyse monolith applications from code repositories in GitHub. With this we can collect new information, such as commits, etc..
- Generalize the data collection tool to any type of monolith applications. This way we can increase significantly the range of applications and provide a better evaluation. Currently only a narrow spectrum of systems are considered (Spring-Boot applications that use the Fénix Framework) and projects need to be opened in Eclipse to be analysed.
- Extend the tool to provide modeller capabilities. The modeller should guide the architecture through the application of microservices patterns and update information on the impact of the modelling process in the metrics provided. Currently there is only a small part of the tool related to modeling, such as the decomposition graph manipulation and the real-time recalculation of the metrics.

VI. Conclusion

In this work we discussed the problem of microservice architecture migration and presented a complexity metric to assess the cost of the migration. We identified some of the existing research articles related the subject of microservices migration and metrics in service oriented systems, assessed each one and presented a conclusion of the analysed articles. This topic is still relatively recent and there is new research emerging everyday. To tackle this challenge, we leveraged on the strategy created in the previous thesis, where we use static analysis to obtain information on method calls and a clustering algorithm to group the domain entities based on that information. In our work we enriched the previous one with information about the read and write sets, and the sequence of accesses of each functionality. As a proof of concept we developed a visualization tool with different views for the decomposition, functionalities and entities. In this tool we introduced our metric for complexity, cohesion and coupling. The complexity metric correctly identified the most complex decompositions, and the functionalities of the monolith. Regarding the similarity measures, we were able to conclude that the sequence measure provides the worst results in terms of complexity, but there is no single combination that outperforms the other, which is confirmed by the existing research that, frequently using different similarity measures, report good results in their validation. We were also able to conclude that the expert decomposition is not the one with the lowest complexity, because the lower complex decompositions tend to identify small microservices while preserving a large cluster that continues to behave as a monolith. This lead to the idea that the tool can help on an incremental migration to microservices approach which is actually defended by the industry experts. The tool code is publicly available in a GitHub repository[6].

References

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

[2] K. Ismail, "7 tech giants embracing microservices," https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices, 8 2018, accessed: 2018-11-29.

[3] L. Nunes, "From a monolithic to a microservices architecture," Master's thesis, Instituto Superior Tcnico, 9 2018.

[4] S. Gilbert and N. Lynch, "Perspectives on the cap theorem," *Computer*, vol. 45, no. 2, pp. 30–36, Feb. 2012. [Online]. Available: https://doi.org/10.1109/MC.2011.389

[5] Y. Al-houmaily and G. Samaras, *Two-Phase Commit*, 01 2009, pp. 3204–3209.

[6] C. Richardson, *Microservices Patterns: With Examples in Java*. Manning Publications, 2019. [Online]. Available: https://books.google.pt/books?id=UeK1swEACAAJ

[7] A. Carrasco, B. v. Bladel, and S. Demeyer, "Migrating towards microservices: Migration and architecture smells," in *Proceedings of the 2Nd International Workshop on Refactoring*, ser. IWoR 2018. New York, NY, USA: ACM, 2018, pp. 1–6. [Online]. Available: http://doi.acm.org/10.1145/3242163.3242164

[8] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. New York, NY, USA: ACM, 2000, pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[9] 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.

[10] M. J. Amiri, "Object-aware identification of microservices," in *2018 IEEE International Conference on Services Computing (SCC)*, July 2018, pp. 253–256.

[11] M. Gysel, L. Klbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," pp. 185–200, 09 2016.

[12] L. Baresi, M. Garriga, and A. Derenzis, "Microservices identification through interface analysis," pp. 19–33, 09 2017.

[13] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," pp. 68–73, 09 2016.

[14] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," pp. 524–531, 06 2017.

[15] S. Klock, J. M. Van der Werf, J. Pieter Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," pp. 11–20, 04 2017.

[16] O. Mustafa and J. Marx Gomez, "Optimizing economics of microservices by planning for granularity level," 04 2017.

[17] S. Delaune and F. Jacquemard, "A theory of dictionary attacks and its complexity," 07 2004, pp. 2 – 15.

[18] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, Jun. 1974. [Online]. Available: http://dx.doi.org/10.1147/sj.132.0115

[19] M. Perepletchikov, C. Ryan, K. Frampton, and Z. Tari, "Coupling metrics for predicting maintainability in service-oriented designs," in *2007 Australian Software Engineering Conference (ASWEC'07)*, April 2007, pp. 329–340.

[20] D. Athanasopoulos, A. Zarras, G. Miskos, V. Issarny, and P. Vassiliadis, "Cohesion-driven decomposition of service interfaces without access to source code," *IEEE Transactions on Services Computing*, vol. 8, pp. 1–1, 01 2014.

[21] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994. [Online]. Available: http://dx.doi.org/10.1109/32.295895

[6] github.com/socialsoftware/mono2micro