Mono2Micro - From a Monolith to Microservices The dynamic analysis of application in the JVM

Bernardo Andrade
Department of Computer Science and Engineering
Instituto Superior Técnico
Lisbon, Portugal
bernardo.andrade@tecnico.ulisboa.pt

***Abstract*—The migration of monolith applications to a microservice architecture has long been a hot topic around major competitive business companies. By turning their complex systems into independently scalable services and managed by small agile software teams, the ideal of decreasing the time to market and increasing the availability of their services has become much more attainable.**

This work leverages on a previous work [1] which was focused on the problem of the complexity associated with software evolution when migrating a monolith to a microservices architecture by executing a static analysis on the monolith's codebase. Therefore, through a dynamic analysis, this work contributes the following research questions: (1) Does the information collected during run-time, for the same set of similarity measures, provide better results, in terms of the quality of the generated decompositions, when compared with the static information captured on the previous work? (2) Does a similarity measure, based on the dynamic behaviour of a system, generate decompositions which are performance optimized? (3) Are the used maintainability quality metrics correlated with performance?

To address each of these questions, a study was conducted on two monolith systems in which their behaviour was dynamically analysed. As result of the analysis we conclude that (i) neither of the analysis techniques, static and dynamic, outperforms the other, but the dynamic collection of data requires more effort (ii) the performance of a decomposition is correctly correlated with the maintainability quality metrics and no similarity measure provides better results than other.

***Index Terms*—Microservices, Software Evolution, Static Analysis, Dynamic Analysis, Software Architecture**

I. Introduction

Microservices [2] have become main stream in the development of large scale and complex systems when companies, like Amazon and Netflix [3], faced constraints on their systems evolution, due to the coupling resulting from the use of a large domain model maintained in a shared database. However, the adoption of this architectural style is not free of problems [4], where the identification of microservices boundaries is one of the most challenging, because a wrong cut results on the need to refactor between distributed services, which impacts on the services interfaces, and cannot have the support of integrated development environments.

The microservices boundaries identification has been addressed by research, e.g. [5]–[8], in the context of the migration of monolith systems to a microservices architecture. Some approaches take advantage of the monolith's codebase and run-time behavior to collect data, analyse it, and propose a decomposition of the monolith. Although each of the approaches use different techniques, they follow the same basic steps: (1) Collection: collect data from the monolith system; (2) Decomposition: define a decomposition by applying a similarity measure and an aggregation algorithm, like a clustering algorithm, to the data collected in the first step; (3) Analysis: evaluate the quality of the generated decomposition using a set of metrics.

However, the approaches differ on the techniques applied in each one of the steps. In terms of the collection of data, they differ in whether it is collected from the monolith using static analysis of the code [7], or if they observe the monolith execution behavior [8]. Besides, the approaches also propose different similarity measures as input to the decomposition algorithms, where some only consider the accesses to domain entities, others distinguish reads from writes, and others consider the sequence of access.

In this paper we do an extensive analysis of a two monolith systems to study whether these techniques provide significant differences when identifying candidate decompositions and the impact of introducing a performance metric in the scope of our analysis framework. The analysis framework is built on top of what is considered by the gray literature as one of the main difficulties on the identification of microservices boundaries in monolith systems: the transactional contexts [9, Chapter 5]. Transactional contexts generate a coupling between domain entities accessed in the context of the same transaction, due to the complexity of decomposing a transactional behavior into several distributed transactions, problem known as the forgetting of the CAP theorem [10].

Therefore, in this paper we address the following research questions: (1) Does the information obtained using a dynamic code analyser, for the same set of similarity measures, provide better results, in terms of the quality of the generated decompositions, when compared with the static information captured on the previous work? (2) Does a particular similarity measure provide better results in terms of the performance of the generated decompositions? (3) Are the used maintainability quality metrics correlated with performance?

In this section we defined the context of our work. The next section formalizes our analysis framework. Section III describes the use of the dynamic analysis technique. In the evaluation, section IV, the analysis framework is applied to

2 systems. Section V presents related work and section VI discusses the outcomes of this work. Finally, section VII presents the conclusions.

II. Similarity Measures and Metrics

A monolith is defined by its set of functionalities which execute in atomic transactional contexts and, due to the migration to the microservices architecture, have to be decoupled into a set of distributed transactions, each one executing in the context of a microservice.

Therefore, a monolith is defined as a triple (F, E, G), where F defines its set of functionalities, E the set of domain entities, and G a set of call graphs, one for each monolith functionality. A call graph is defined as a tuple (A, P), where $A = E \times M$ is a set of read and write of accesses to domain entities ($M = \{r, w\}$), and $P = A \times A$ a precedence relation between elements of A such that each access has zero or one immediate predecessors, $\forall_{a \in A} \#\{(a_1, a_2) \in P : a_1 = a\} \leq 1$, and there are no circularities, $\forall_{(a_1,a_2) \in P_T}(a_2, a_1) \notin P_T$, where P_T is the transitive closure of P. The precedence relation represents the sequences of accesses associated with a functionality.

A. Similarity Measures

The definition of similarity measures establishes the distance between domain entities. Domain entities that are closer, according to a particular similarity measure, should be in the same microservice. Therefore, since we are interested in reducing the number of distributed transactions a functionality is decomposed in, we intend to define as close the domain entities that are accessed by the same functionalities.

The access similarity measure measures the distance between two domain entities, $e_1, e_2 \in E$, as:

$$sm_{access}(e_1, e_2) = \frac{\#(funct(e_1) \cap funct(e_2))}{\#funct(e_1)}$$

where $funct(e)$ denotes the set of functionalities in the monolith whose call graph has a read or write access to e. This measure takes a value in the interval 0..1. When all the functionalities that access e_1 also access e_2 then it takes the value 1.

Since the cost of reading and writing is different in the context of distributed transactions, because writes introduce new intermediate states in the decomposition of a functionality, the next two similarity measures distinguish read from write accesses in order to reduce the number of write distributed transactions:

$$sm_{read}(e_1, e_2) = \frac{\#(funct(e_1, r) \cap funct(e_2, r))}{\#funct(e_1, r)}$$

$$sm_{write}(e_1, e_2) = \frac{\#(funct(e_1, w) \cap funct(e_2, w))}{\#funct(e_1, w)}$$

where $funct(e, m)$ denotes the set of functionalities in the monolith whose call graph has an access according to mode m, read or write, respectively. These two measures tend to include in the same microservice, domain entities that are read or written together, respectively.

Finally, another similarity measure that is found in the literature groups domain entities that are frequently accessed in sequence, in order to reduce the number of remote invocations between microservices, i.e., the domain entities that are frequently accessed in sequence should be in the same microservice. Therefore, the sequence similarity measure is defined:

$$sm_{sequence}(e_1, e_2) = \frac{sumPairs(e_1, e_2)}{maxPairs}$$

where $sumPairs(e_1, e_2) = \sum_{f \in F} \#\{(a_i, a_j) \in G_f.P : (a_i.e = e_1 \wedge a_j.e = e_2) \vee (a_i.e = e_2 \wedge a_j.e = e_1)\})$, where $G_f.P$ is the precedence relation for functionality f, is the number of consecutive accesses of e_1 and e_2, and $maxPairs = max_{e_i,e_j \in E}(sumPairs(e_i, e_j))$ is the max number of consecutive accesses for two domain entities in the monolith.

B. Complexity Metric

A decomposition of a monolith is a partition of its domain entities set, where each element is included in exactly one subset, a cluster, and a partition of the call graph of each one of its functionalities. Therefore, given the call graph G_f of a functionality f, and a decomposition $D \subseteq 2^E$, the partition call graph of a functionality $partition(G_f, D) = (LT, RI)$ is defined by a set of local transactions LT and a set of remote invocations RI, where each local transaction

(i) is a subgraph of the functionality call graph, $\forall_{lt \in LT} : lt.A \subseteq G_f.A \wedge lt.P \subseteq G_f.P$;

(ii) contains only accesses in a single cluster of the domain entities decomposition, $\forall_{lt \in LT} \exists_{cinD} : lt.A.e \subseteq c$;

(iii) contains all consecutive accesses in the same cluster, $\forall_{a_i \in lt.A, a_j \in G_f.A} : ((a_i.e.c = a_j.e.c \wedge (a_i, a_j) \in G_f.P) \implies (a_i, a_j) \in lt.P) \vee ((a_i.e.c = a_j.e.c \wedge (a_j, a_i) \in G_f.P) \implies (a_j, a_i) \in lt.P)$.

From the definition of local transaction, results the definition of remote invocations, which are the elements in the precedence relation that belong to different clusters, $RI = \{(a_i, a_j) \in G_f.P : a_i.e.c \neq a_j.e.c\}$. Note that, in these definitions, we use the dot notation to refer to elements of a composite or one of its properties, e.g., in $a_j.e.c$, $.e$ denotes the domain entity in the access, and $.c$ the cluster the domain entity belongs to.

The complexity for a functionality migration, in the context of a decomposition, is the effort required in the functionality redesign, because its transactional behavior is split into several distributed transactions, which introduce intermediate states due to the lack of isolation. Therefore, the following aspects have impact on the functionality migration redesign effort:

- The number of local transactions, because each local transaction may introduce an intermediate state;
- The number of other functionalities that read domain entities written by the functionality, because it adds the

need to consider the intermediate states between the execution of the different local transactions;
- The number of other functionalities that write domain entities read by the functionality, because the functionality redesign has to consider the different states these domain entities can be.

This complexity is associated with the cognitive load that the software developer has to address when redesigning a functionality. Therefore, the complexity metric is defined in terms of the functionality redesign.

$$complexity(f, D) = \sum_{lt \in partition(G_f, D)} complexity(lt, D)$$

The complexity of a functionality is the sum of the complexities of its local transactions.

$$complexity(lt, D) = \#\cup_{a_i \in prune(lt)} \{f_i \neq lt.f : dist(f_i, D) \wedge a_i^{-1} \in prune(f_i, D))\}$$

The complexity of a local transaction is the number of other distributed functionalities that read, or write, domain entities, written, or read, respectively by the local transaction. The auxiliary function $dist$ identifies distributed functionalities, given the decomposition; a_i^{-1} denotes the inverse access, e.g. $(e_1, r)^{-1} = (e_i, w)$; and $prune$ denotes the relevant accesses inside a local transaction, by removing repeated accesses of the same mode to a domain entity. If both read and write accesses occur inside the same local transaction, they are both considered if the read occurs before the write. Otherwise, only the write access is considered. These are the only accesses that have impact outside the local transaction.

C. Coupling and Cohesion Metrics

Coupling and cohesion are important qualities of any software system, particularly, in a system implementing a microservices architecture, because one of its goals is to foster independent agile teams. Therefore, we intend to measure the coupling and cohesion of the monolith decomposition.

The cohesion of a cluster of domain entities depends on the percentage of its entities that is accessed by a functionality. If all the cluster's assigned entities are accessed each time a cluster is accessed, it means that the cluster is cohesive:

$$cohesion(c) = \frac{\sum_{f \in funct(c)} \frac{\#\{e \in c.e : e \in G_f.A.e\}}{\#c.e}}{\#funct(c)}$$

where $funct(c)$ denotes the functionalities that access the cluster c, and $G_f.A.e$ the entities that are accessed by functionality f.

The coupling between two clusters is defined by the percentage of entities one cluster exposes to other cluster. It can be defined in terms of the remote invocations between the two clusters.

$$coupling(c_i, c_j) = \frac{\#\{e \in c_j : \exists_{ri \in RI(c_i, c_j)} e = ri[2].e\}}{\#c_j.e}$$

where $RI(c_i, c_j)$ denotes the remote invocations from cluster c_i to cluster c_j

III. Dynamic Data Collection

The different approaches to the migration of monoliths to microservices architectures apply, in the Collection step, either static or dynamic techniques, but there is no evidence in the literature on whether one of them subsumes the other, whether they are equivalent, or even whether they are complementary. Therefore, we collected data using both techniques in order to address this open problem.

Data was collected from two monolith systems used in the previous work [1], LdoD[1] and Blended Workflow[2], that are implemented using the Model-View-Controller architectural style, where the controllers process input events by triggering transactional changes in the model, thus, corresponding to the monolith functionalities. The monolith is designed considering its controllers as transactions that manipulate a persistent model of domain entities. Our collection tool was developed to cope with the widely used Spring-Boot[3] framework and the Fénix Framework[4] Object-Relational Mapper (ORM).

As result of the collection, the functionalities accesses are stored in JSON format. It consists in a mapping between functionality names and functionality objects, where each object has a traces field that consists in a list of trace objects. Each trace is characterized by a unique identifier and a (compressed) list of accesses observed for a specific functionality execution. An $Access$ is composed by the numeric identifier of the domain entity and the access type, either read or write.

During the Decomposition step of the migration process, our tool uses hierarchical clustering (Python SciPy[5]) to process the collected data and, according to the 4 similarity measures, generate a dendrogram of the domain entities. The generated dendrogram can be cut in order to produce different decompositions, given the number of clusters. Our decomposition tool supports different combinations of the similarity measures, for instance, it is possible to generate a decomposition with the following weights (30% access, 30% read, 20% write, 20% sequence).

For the Analysis step our tool allows to generate multiple decompositions, by varying the similarity measures weights and the number of clusters, and compare them according to the maintainability metrics: complexity, coupling and cohesion. Additionally, two different decompositions of the same system can be compared using the MoJoFM [11] distance metric.

MoJoFM is a distance measure between two architectures expressed as a percentage. This measure is based on two key operations used to transform one decomposition into another: moves (Move) of entities between clusters, and merges (Join) of clusters. Given two decompositions, A and B, MoJoFM is defined as:

$$MoJoFM(A, B) = (1 - \frac{mno(A, B)}{max(mno(\forall A, B))}) \times 100\%$$

[1]https://github.com/socialsoftware/edition
[2]https://github.com/socialsoftware/blended-workflow
[3]https://spring.io/projects/spring-boot
[4]https://fenix-framework.github.io/
[5]https://docs.scipy.org/doc/

where $mno(A, B)$ is the minimum number of Move and Join operations needed to transform A into B and $max(mno(\forall A, B))$ is the number of Move and Join operations needed to transform the most distant decomposition into B.

A. Dynamic analysis collection pipeline

The dynamic data collection is done in a running instance of the monolith under analysis and comprises a pipeline of three consecutive steps: Setup, Monitoring and JSON generation. In general, only the first and last steps may require manual adjustments depending on the execution environment and the software in question, whereas the second is fully handled by Kieker[6] and associated tools. In total, three types of deliverables are produced by the end of the pipeline: configuration files, created in the Setup step; raw data files - logs - extracted from the system while being monitored and finally, the desired outcome, the JSON file including semantic data prepared to be analysed.

1) Setup: Establishes the building blocks of the monitoring step and answers the questions of where to collect data from, what data should be collected and how it should be handled after being collected. The first two questions were answered with the help of AspectJ[7]. It was used to instrument the byte code of the Java applications and to develop new technology-specific probes as suggested in Kieker's user guide. Since it leverages the use of AOP, no manual modifications on the source code were required neither the insertion of annotations on intended methods. The implementation of probes was driven primarily by the accuracy and performance of the collection process. Since extra behaviour is added during run-time, care must be taken so that the extra overhead does not jeopardize the user experience. The less precise the collection, the longer the system will be occupied writing meaningless data to the file system. In simple terms, the end goal of this step is to intercept calls to the FenixFramework's data access methods, the ones responsible for manipulating the respective entity's persistent state, formatted as follows $\{get, set, add, remove, delete\} < attributeName >$. For instance, in the *LdoD* codebase exists a class called *Category* that, in order to be persisted, has to extend the already generated class *Category_Base* to access such methods as *getName*, *setName*, *addTag* and *removeTag*. New information also had to be added to be, at least, the same that static analysis gathers to determine the corresponding accesses in a next step. Thus, the implemented instrumentation collects the following information:

1) **Declaring class type name** - The name of the class containing the definition of the method
2) **Method's name** - The name of the called method
3) **Target's type** - The type of the object that calls the method
4) **Method's arguments types** - The types of the objects passed as arguments
5) **Method's return type** - The type of the object returned by the method

As a result, in order to hold this data, the following compact format was adopted: **1:2:3:4:5** . Considering that types are collected during run-time, the instrumentation is accounting for two major contingencies: (i) objects may not have an explicit dynamic type ($null$) (ii) Java generic types employed on iterables[8] are no longer available[9] and therefore, it's impossible to discover the iterable type. Therefore, the following assumptions were taken:

1) Whenever a dynamic type can't be discovered, the static type is used. If neither of them can't be found the type is considered unknown and sent in blank.
2) Whenever in the presence of an iterable object with at least one element, its type is considered to be the type of the first element (for performance reasons). Otherwise, if empty, the type is considered unknown and is sent in blank.

At this time, only the question of how data should be handled after being collected remains open. Straight answer is: it depends on how the DCA tool, Kieker, is configured. The best way to setup Kieker is through the usage of a special configuration file called $kieker.monitoring.properties$. The most impactful properties are the ones related to the queue that holds records[10] such as its maximum capacity, its implementation [11] and action to take when there are incoming records and the queue is full. A compression method was also used to compress each log file as zipped binary file. This property is extremely valuable whenever instrumented systems possess little disk space.

2) Monitoring: Having set up all the requirements, the monitoring process may start. During run-time, when a controller executes, Kieker will capture the accesses to the domain entities in the context of the execution of a controller and will store the records as CSV. Each record not only contains the method's data previously explained but also other Kieker-related contextual information needed for ordering traces after the monitoring has ended such as the execution order index and stack size. Kieker already provides a built-in trace analysis tool that is able to generate a desired representation of an ordered sequence of executions (trace).

As a result, the file $executionTraces.txt$ is generated containing all the valid traces. Listing 1[12] shows an example of a valid execution trace representation. All the desired lines follow the format of $< x\ methodData >$ where x identifies the trace that the executions belongs to and $methodData$ the method's data in the adopted format.

[6]http://kieker-monitoring.net

[7]The Eclipse Foundation (2011). The AspectJ Project. http://www.eclipse.org/aspectj/

[8]https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html

[9]https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[10]A record can be seen as a future line of the log containing

[11]http://javadox.com/org.jctools/jctools-core/1.0/org/jctools/queues/package-summary.html

[12]Irrelevant information omitted to increase readability

Listing 1. Excerpt of a trace from the file *executionTraces.txt*

```
<6 SearchController:getHeteronyms:SearchController::HashMap>
<6 DomainRoot_Base:getLdoD:DomainRoot::LdoD>
```

3) JSON generation: As the name suggests, this step is focused on producing a JSON file from the execution traces obtained in the previous step. After completing three sequential sub-steps - Parsing, Translation and Compression - each trace of (compressed) accesses is prepared to be associated with its corresponding functionality if it proves to be unique.

On the Parsing step, the $executionTraces.txt$ file is parsed line by line to obtain the functionality name, the unique identifier and the methods' data of each trace. Taking the lines of the trace presented in Listing 1 as an example, the functionality name would be $SearchController.getHeteronyms$, the trace identifier 6 and $DomainRoot\_Base : getLdoD : DomainRoot :: LdoD$ the method's data. This data is further parsed to collected each respective semantic information - name and the types of the target, arguments and return objects - and subsequently used as input to the translation step. The goal of the Translation step is two-fold: (1) translate accessed entities names to numeric identifiers and (2) translate the method data, received as an input, to the corresponding accesses to persistent domain entities. The translation to unique numeric identifiers is maintained until all traces are parsed. When a new persistent domain entity is discovered, a counter is increased and its value is assigned to the entity. Hence uniqueness is ensured. The translation algorithm behaves as follows:

- If the method's name starts with get it means that persistent entities were read. The accessed entities can be the target's type and the return type.
- Alternatively, if the method's name starts with set, add, $remove$ or $delete$ then persistent entities may have been written. In this case, the written entities can be the target's type and each argument type.

When translating a type, it's necessary to first check whether or not that it corresponds to a persistent entity. For instance, if there is a case where the target's type is unknown ($null$) and the return/argument is of type $String$, then no persistent entities were actually accessed. If it proves to be a persistent entity, then its numeric identifier is associated with the access type ($Read$ or $Write$) thus constituting an $Access$.

Finally, one of the biggest challenges of this work, that arose from executing a dynamic analysis, was the massive amount of data generated allied to the difficulty of processing large traces due to time and space requirements. The explanation for this amount is quite simple: loops. Since code is executed in this type of analysis, a loop may repeat its body X amount of times and as a consequence, the size of execution traces may increase linearly. Therefore, in the Compression step, after the whole execution trace has been translated into a list of accesses, this list is subject to compression. This compression was achieved by leveraging the use of an enhanced version of the linear sequence compression algorithm called *Sequitur* [12] that can represent a sequence as a hierarchical structure. The authors of [13] were able to improve the compression ratio of *Sequitur*, and still preserving its linear performance, by combining it with Run-length encoding. A compressed sequence is a new sequence composed of a new type of element. For ease of communication, let's call it Reduced Trace Element (RTE). A RTE can be seen as a container that attaches the information of how many times the element that it holds is repeated. It can only hold one of the following elements: a $Rule$ or an element of the original sequence, in our case, an $Access$. A $Rule$ is a special element that specifies how many of the next elements are going to be repeated. In the end, after fully crunching the $executionTraces.txt$, the JSON file is generated. An excerpt is shown in Listing 2 where a functionality called $SearchController : getHeteronyms$, contains two unique traces[13]. Each element of this list, a RTE, is serialized as a list whose content may change as follows:

- if the RTE holds an $Access$, then the numeric identifier of the domain entity is appended to the list after the access type, read ("R") or write ("W");
- if the RTE holds a $Rule$, then the amount of subsequent elements that belong to the same sequence is appended to the list;
- if the element held by the RTE is repeated more than one time, then this number is added at the end of the list;

For instance, the $SearchController : getHeteronyms$ functionality has a $Rule$, serialized as $[2, 2]$, that means the sequence of two elements next to it occurs two times. If the rule was decompressed, these three elements (rule plus next two consecutive elements) would produce 4 access elements. The same rationale is also applied to some of those access elements that possess a third element, for example $["R", 71, 2]$. The decompressed size of the rule and its two elements would be 8, more than double the compressed size (3) just on this short example.

Listing 2. Excerpt of a JSON file

```
{
  "SearchController:getHeteronyms": {
    "t": [
      {
        "id": 0,
        "a": [["R",9,5],[2,2],["R",1,2],["R",7,2],["R",4]]
      },
    ]
  },
}
```

IV. Evaluation

The goal of evaluation is to answer the research questions by (i) assessing which technique, static or dynamic, provides the best results and the effort of applying the latter; (ii) qualifying the impact of each similarity measure on the performance of generated decompositions and (iii) discovering possible correlations between the maintainability quality metrics and performance.

The source of data came from the two already mentioned monolith systems used in the previous work, LdoD (122

[13]Uniqueness is ensured by comparing the compressed list of accesses of a new trace, for the same functionality, against the list of the traces already discovered.

TABLE I
COVERAGE OF DYNAMICALLY COLLECTED DATA

	LdoD			BW
	Prod	**Tests**	**Sim**	**Sim**
Cov Entities (%)	79	82	80	86
Cov Controllers (%)	44	96	84	68

controllers, 71 domain entities) and Blended Workflow (98 controllers, 52 domain entities), henceforth abbreviate as BW, that were subject of dynamic analysis.

Regarding the LdoD system, it was monitored in three different environments: production (Prod), functional testing (Tests) and simulation (Sim). The production monitoring lasted 3 weeks and a total of $490\,\mathrm{GB}$ worth of data was collected. Throughout this period, a tight supervision was necessary to oversee the impact the monitoring had on the performance of the system's functionalities. Since the server hosting the application had a small free disk space (around $20\,\mathrm{GB}$) and a massive drop in performance was observed if it was full, it was mandatory to collect the generated logs from time to time (2-3 days) to not harm the user experience and to gather fresh logs instead of discarding them.

Analyzing the collected data presented in Table I, only 44% of the controllers were exercised in production. Therefore, further processing and evaluation of this data were abdicated due to the substantial effort required to process it and the relatively little coverage. Concerning functional testing, it was achieved by running a suite of 200 integration tests (4.207 lines of code) that exercised 96% of the controllers and 82% of the domain entities, generating a few megabytes ($<200\,\mathrm{MB}$) of data, while the instruction coverage, reported by JaCoCo[14], was 72% for domain entities and 82% for controllers. The reduced size of the collected data is explained by the usage of small subset of the original database's data and so, the traces associated with the execution of functionalities were much shorter. Finally, an expert of the system simulated, during one hour, the use of functionalities, using a database with a minimal set of data, and $200\,\mathrm{MB}$ of data was collected and 84% of the controllers and 80% of the domain entities were exercised.

In what concerns the BW system, it was only simulated by an expert during an hour and 86% of entities and 68% of controllers were exercised. The reduced number of exercised controllers is justified by the deprecation of several controllers that are not reachable through the user interface.

Similarly to how data was evaluated in the previous work regarding the static analysis, for each system, and respective environment, a JSON file was generated, after collection, and then served as input to the analyser feature of Mono2Micro that creates several dendrograms by varying the weights of the four existing similarity measures - Access, Write, Read and Sequence - in intervals of 10 in a scale of 0 to 100. Then several cuts were performed on each one. Each cut resulted in a candidate decomposition of the monolith with a specific number of clusters, varying from 3 to 10. For each generated decomposition, the values for the quality metrics were calculated. The complexity metric value had to be normalized in order to compare them among the monoliths, since they depend on the number of functionalities of each monolith. The uniform complexity of a given decomposition d of a monolith is calculated by dividing the complexity of d by the $maxComplexity$. The $maxComplexity$ value is determined by calculating the complexity of a decomposition of the monolith where each cluster has a single domain entity. Therefore, the uniform complexity of any monolith decomposition is a value in the interval 0 to 1.

Analogously, to assess the correlation between each metric and the weights given to each similarity measure and the number of clusters, a linear regression model was employed using the OLS method. For instance, the linear regression model that correlates the complexity of a decomposition with the weights of each similarity measure is given by:

$$\begin{aligned} uComplexity(d) = \beta_1 \cdot d.weight_A + \beta_2 \cdot d.weight_W \\ + \beta_3 \cdot d.weight_R + \beta_4 \cdot d.weight_S \\ + \beta_5 \cdot \#d.clusters + cons \end{aligned}$$

To test this regression, a hypotheses was defined as follows:

- H_0: $\beta_1 = \beta_2 = \beta_3 = \beta_4 = \beta_5 = 0$; meaning that the complexity of a decomposition does not have a relation with any of the five parameters
- H_1: $\beta_1 \neq 0 \vee \beta_2 \neq 0 \vee \beta_3 \neq 0 \vee \beta_4 \neq 0 \vee \beta_5 \neq 0$; meaning that the complexity of a decomposition does have a relation with at least one of the five parameters

All the obtained regression results in the next sections show enough evidence to reject the null hypothesis at a significance level of 0.05.

Henceforth, to reduce the amount of space occupied by the results depicted in tables, a terminology was also adopted where:

- N means the number of clusters;
- A, W, R and S stand for Access, Write, Read and Sequence similarity measures respectively;
- RC refers to the Regression Coefficient (β_i) obtained for a given dependent variable X_i. It represents the effect of the dependent variable on the independent variable (Y). If β_i is positive, then as X_i increases, the value of Y will also increase. If β_i is negative, then as X_i increases, the value of Y will decrease.
- CI depicts the 95% Confidence Interval. As the name suggests, it ensures that one can assume with 95% of confidence that there is a statistically significant correlation explained by the respective coefficient if zero does not belong to the interval (✓). Otherwise, if zero is within the interval, no correlation can be established (✗).
- R^2 represents the statistical measure that indicates the variability in the dependent variable of the regression model.

[14]https://github.com/jacoco/jacoco

TABLE II
COMPARE COLLECTED DATA - AVERAGE OF COVERED ENTITIES PER CONTROLLER

	LdoD				BW	
	Static	Tests	Static	Sim	Static	Sim
AVG(Cov. E/C)	95%	71%	91%	77%	93%	78%

A. Static and Dynamic Analysis Comparison

A study was also conducted on the impact of similarity measures over the quality metrics with the usage of dynamic analysis. Due to the majority of coefficients having considerably close and low values (< 0.01) and some confidence intervals contain the zero value, no specific combination of similarity measures determines the generation of the best decomposition according to any of the three maintainability quality metrics. This corroborates the conclusion achieved from the previous work that "there isn't a unique solution to the values of similarity, that leads to the best decomposition in terms of the complexity metric."

1) Static and Dynamic Analysis Comparison: Although, it seems that data collected dynamically provides similar insight in terms of the correlation between the similarity measures and the quality metrics, we want to know whether they produce significantly different decompositions. To do this analysis we need to first compare the collected data.

On Table I we can observe, as expected, that all the controllers exercised by dynamic analysis are also captured by static analysis, but the inverse does not occur. For instance, in LdoD, the dynamic collection through tests covered 96% of the system while through the simulation only covered 84% of the controllers. However, for the coverage of the accesses to domain entities in the context of the controllers, in some cases, the dynamic analysis can identify accesses to domain entities, in the context of a controller execution, that the static collection does not, due to late binding. And, of course, the opposite also occurs, because depending on the inputs provided to controllers and data available in the database, some of the domain entities may not be accessed, both in tests and simulation.

To assess the results of the two techniques, we compare the highest quality decompositions, in terms of complexity, from each approach with a decomposition proposed by a domain expert, for both systems. In this analysis we consider the expert decompositions as reference and evaluate, using the MoJoFM metric, which approach provides closer results to it. Since the two techniques may miss some domain entities during the collection phase, we decided that all the unassigned entities would be put in the biggest cluster as this strategy conforms with the incremental decomposition strategy rationale [14, Chapter 13].

The results from the comparisons are represented in Table III, where each cell indicates the MoJoFM percentage value (0 - 100%) between the lowest complexity decomposition with N clusters, using the column's collection technique, and the system's expert decomposition. Overall, the MoJo values obtained for the different collection approaches were very similar, for both systems, which leads us to conclude that there isn't a collection technique that provides better results. However, note that, especially on the simulation technique, the dynamic analysis didn't cover all controllers during the collection phase and also missed more entities than the static approach. Therefore, we decided to assess if the dynamic analysis approach could surpass the static analysis if only the common controllers and entities were considered.

TABLE III
COMPARING GENERATED WITH EXPERT DECOMPOSITIONS

		LdoD			BW	
		Static	Tests	Sim	Static	Sim
N	3	62.12	65.15	68.18	46.67	44.44
	4	60.61	69.7	66.67	44.44	46.67
	5	56.06	68.18	66.67	44.44	60
	6	78.79	66.67	66.67	62.22	57.78
	7	77.27	74.24	68.18	66.67	64.44
	8	83.33	72.73	59.09	66.67	62.22
	9	81.82	74.24	57.58	71.11	62.22
	10	45.45	74.24	56.06	71.11	62.22
avg		68.18	70.64	63.64	59.17	57.5

TABLE IV
COMPARING GENERATED WITH EXPERT DECOMPOSITIONS, CONSIDERING ONLY THE COMMON CONTROLLERS AND ENTITIES

		LdoD				BW	
		Static	Tests	Static	Sim	Static	Sim
N	3	65.15	59.09	63.64	71.21	46.67	44.44
	4	51.52	69.7	62.12	71.21	51.11	46.67
	5	72.73	68.18	63.64	66.67	53.33	60
	6	72.73	54.55	68.18	66.67	51.11	57.78
	7	75.76	74.24	63.64	69.7	68.89	64.44
	8	74.24	72.73	68.18	59.09	66.67	62.22
	9	72.73	74.24	57.58	57.58	68.89	62.22
	10	68.18	72.73	56.06	56.06	77.78	62.22
avg		69.13	68.18	62.88	64.77	60.56	57.5

To evaluate this scenario, we re-ran the static analysis considering only the common controllers and domain entities, for each dynamic technique. The results are represented in Table IV, where we can observe that, on average, both approaches continue to generate decompositions almost equally distant to the expert's, for both systems. The major noticed difference is the average MoJo values obtained for the static approach when *evened* with the dynamic analysis using the expert simulation approach. This suggests that the missed controllers by the dynamic collection using the simulation technique may be crucial to find the couplings between the domain entities that lead to a decomposition closer to the expert's, highlighting the impact of the variability among different dynamic collection approaches.

Based on these results, we conclude that, for both systems, we don't see significant differences between the lowest complexity decompositions obtained using statically and dynamically collected data and that none of the approaches achieve identical decompositions to the expert's, since the average MoJo values obtained vary around 60-70%.

Given the similarities when compared to the expert, we assessed how far apart the static and dynamic decompositions were from each other, considering the common controllers and entities.

TABLE V
COMPARING STATIC WITH DYNAMIC DECOMPOSITIONS, CONSIDERING ONLY THE COMMON CONTROLLERS AND ENTITIES

		LdoD		BW
		Static vs Tests	Static vs Sim	Static vs Sim
N	3	57.41	84.62	83.33
	4	83.02	82.35	63.41
	5	78.85	80.39	50
	6	78.85	78	58.97
	7	78.85	76	57.89
	8	82.69	46.94	50
	9	80.77	48.98	50
	10	60	42.86	37.84
avg		75.06	67.52	56.43

In Table V we can observe, for LdoD, the average MoJo between the evened static and tests approaches was 75%, while between the evened static and simulation approaches was 67%. For BW, the average MoJo between the evened static and simulation approaches was 56%. This leads to the conclusion that, on average, although equally distant to the expert's decomposition, the static and dynamic approaches do not generate similar decompositions, which suggests that there is space for future research on these differences but does nor invalidate our conclusions that neither of the analysis techniques outperforms the other.

B. The impact of Performance

This section is, instead, devoted to (1) understand the impact of each similarity measure in the performance of the generated decompositions and (2) apprehend potential correlations between the maintainability and performance metrics. It consequently provides an answer the last two research questions. The obtained results were exclusively produced from data collected with dynamic analysis.

The introduced performance metric calculates the number of hops between microservices during a distributed transaction, in other terms, the network communication latency of a distributed functionality. Leveraging the formalisation presented in Section II, the latency of a functionality for a given decomposition is expressed as the average of the latency of its traces:

$$latency(f, D) = \frac{\sum_{t_f \in traces(partition(G_f, D))} latency(t_f) - 1}{\#traces(partition(G_f, D))}$$

where $traces(partition(G_f, D)$ denotes the unique sequences of local transactions in the partition call graph of functionality f, and $latency(t_f)$ denotes the number of local transactions in the sequence. Since the goal is to count the number of remote invocations between local transactions, minus one must be subtract. This subtraction assumes that each trace in the partition call graph has a at least one local transaction.

The latency of a decomposition is the average of the latency of its functionalities:

$$latency(D) = \frac{\sum_{f \in F} latency(f, D)}{\#F}$$

TABLE VI
COMPARISON OF THE IMPACT OF SIMILARITY MEASURES ON PERFORMANCE

	LdoD Tests		LdoD Sim		BW Sim	
	RC	CI	RC	CI	RC	CI
N	0.0207	✓	0.0088	✓	0.0223	✓
A	-0.0003	✓	0.0017	✓	-0.0008	✓
W	-0.0004	✓	0.0082	✓	0.0002	✓
R	-0.0001	✗	0.0002	✗	-0.0009	✓
S	-0.0002	✗	0.0012	✓	0.0016	✓
R^2	0.129		0.640		0.460	

TABLE VII
COMPARISON OF THE IMPACT OF NETWORK LATENCY IN COMPLEXITY

	LdoD Tests		LdoD Sim		BW Sim	
	RC	CI	RC	CI	RC	CI
Net. Latency	1.0014	✓	0.8680	✓	1.1896	✓
R^2	0.955		0.962		0.980	

where $latency(f, D)$ corresponds to the latency of a functionality f in the context of a decomposition D.

The values obtained for this metric also had to be normalized given the same rationale applied to the complexity metric as well as the same applied linear regression model to assess the correlation between this metric and the weights given to each similarity measure and the number of clusters.

1) Similarity Measures: Exhibited in Table VI, the results show a comparison across both systems and respective environments to determine a combination of similarity measures that provides low latency decompositions. As expected, there is a consensus indicating that the number of clusters have a statistically significant positive correlation with network latency. A higher number of clusters/microservice candidates is presumed to increase the network communication latency of a functionality. However, in terms of similarity measures, there is no robust conclusion given that (1) some confidence intervals include the zero value (2) results from different environments/systems disagree on the sign of the coefficient and (3) the coefficient values are very close to zero (< 0.01) indicating a very weak correlation.

2) Maintainability quality metrics: The regression results from Table VII and Table VIII show that both complexity and coupling metrics have a statistically significant positive correlation, with considerably low variability ($R^2 > 0.6$), with latency, which allows to conclude that decompositions with higher network latency tend to have worse values regarding the two maintainability metrics. This correlation is expected since decompositions with a high network latency tend to have a high number of distributed transactions for each functionality and inevitably more remote invocations between clusters. Note that, the correlation between latency and complexity is much stronger than with coupling, as can be observed by the magnitude of the coefficients that show almost a direct proportion with complexity.

Concerning cohesion, as shown in Table IX, network latency shows a negative correlation on both simulation environments whereas on the functional testing executed on the LdoD system, a correlation can not be established given the inclusion

TABLE VIII
COMPARISON OF THE IMPACT OF NETWORK LATENCY IN COUPLING

	LdoD Tests		LdoD Sim		BW Sim	
	RC	CI	RC	CI	RC	CI
Net. Latency	0.1751	✓	0.1047	✓	0.5344	✓
R^2	0.722		0.605		0.727	

TABLE IX
COMPARISON OF THE IMPACT OF NETWORK LATENCY IN COHESION

	LdoD Tests		LdoD Sim		BW Sim	
	RC	CI	RC	CI	RC	CI
Net. Latency	0.0201	✗	-0.1766	✓	-0.2576	✓
R^2	0.628		0.672		0.704	

of zero in the confidence interval. This lack of unanimity is, to some extend, expected because the number of remote invocations of a functionality is not directly correlated with cohesion as a functionality does not need to access the entities of all clusters to have low latency.

V. RELATED WORK

In recent years, a myriad of approaches to support the migration of monolith systems to microservices architectures have been proposed [6], [15]–[25], which use the monolith specification, codebase, services interfaces, run-time behavior, and project development data to recommend the best decompositions [26].

In this paper we address the approaches that use the monolith codebase or run-time behavior. Although they follow the same steps, they diverge on what is their main concern and, consequently, on the similarity measures that they use, such as accesses [8], reads [5], [7], writes [5], [7], and sequences [5]. On the other hand, some authors use execution traces to collect the behavior of the monolith, e.g. [8], [25], but there is no empirical evidence on whether it provides better data than the static mechanisms, and what is the required effort to collect the data. As far as our knowledge goes, there is no work on the comparison between the use of static and dynamic analysis in the migration of monolith systems to a microservices architectures.

Some of these approaches also use different metrics to assess the result of their decompositions. Therefore, we studied the literature on microservices quality to identify which metrics to consider. The metrics we used for evaluating the complexity of the decompositions are based on current state of the art metrics for service-oriented systems [27]. We applied the complexity metric for the migration of monolith systems to microservices architecture [1], which was extended to also consider several traces for a functionality, due to the result of the dynamic collection the data. Other complexity metrics use the percentage of services with support for transactions [28], but they lack an integrated perspective that we provide by defining the transactional complexity of a functionality. Another complexity metric considers the number of operations and services that can be executed in response to an incoming request [29], while we consider the complexity of implementing a local transaction in the terms of inter-functionalities interactions, which emphasizes the complexity of cognitive load, i.e., the total number of other functionalities to consider when redesigning a functionality.

In what concerns the coupling and cohesion metrics, the definitions in the context of services consider, for the coupling metric, the number of operations in the service interfaces that are used by other services, where a higher number reflects a higher coupling, and, for the cohesion metric, the percentage of operations of the service interface that are used together [29]. In our implementation, we define coupling as the percentage of domain entities exposed in the interface, and cohesion as the percentage of the cluster domain entities that are used together to accomplish a functionality. Overall, all our metrics are designed on the perspective of the monolith functionality.

There is work that integrates static and dynamic analysis. For instance, in [30], static analysis is used to complement the incompleteness of dynamic analysis, in order to increase programming comprehensibility. Recent work on the migration of microservices also integrates static and dynamic analysis techniques [31], [32], by complementing the data collected through static analysis with dynamic analysis collected data. None of these approaches evaluate or discusses the quality of data obtained with each one of the techniques.

VI. DISCUSSION

A. Lessons Learned

From this research we learned the following lessons:

- It is not possible to conclude that one of the similarity measures determines the generation of the best decomposition according to any of the three metrics.
- The combination of the similarity measure values for the best decomposition according to the quality metrics may be dependent on specific characteristics of the monolith system.
- Currently, the best approach is to run our analyser feature to find the best combinations for each monolith.
- It is not possible to conclude that the decompositions generated using one of the analysis techniques, static or dynamic, outperforms the other.
- The effort to collect data dynamically is significantly superior than the static collection, specially when collecting and evaluating data from production which resulted in a very low coverage. On the other hand, the use of integration tests, that achieved better coverage, has a high development cost, because, contrary to unit tests, which aim to have 100% coverage, integration tests, which are harder to develop and maintain, are designed to verify the modules integration, not the execution of all paths.

B. Threats to Validity

1) Internal Validity: Since dynamic analysis adds an extra layer of computation on top of the monitored systems runtime behaviour, the assumptions made on the instrumentation, to minimize the performance degradation perceived by end-users, are a clear bias on the obtained results given that: (i)

an iterable object type is considered to be the type of the first element and (ii) new records are discarded when Kieker's queue is full.

The approach of placing the entities not found during the collection process into the biggest cluster, when comparing the static and dynamic decompositions with the expert's, may have biased our results, as there is a probability associated with the expert decomposition that may or may not contain those entities in the same cluster. However, we also made the comparisons using other approaches and achieved similar results, thus, we are confident in discarding this as a threat.

2) External Validity: Due to the effort associated with the dynamic collection of data, we only analyzed two systems, but from the comparison with the decompositions generated from statically collected data, we may extrapolate that the quality of one decomposition does not outperforms the other, though the dynamic analysis of more monoliths is necessary. Nevertheless, the conclusions about the incompleteness of data and required effort associated with the dynamic collection of data are evident and shows that a cost/benefit relation may tend for the static analysis approaches.

Due to the diversity of metrics that exist for complexity, coupling and cohesion, can our results be generalized? Despite this diversity, we are confident that the results are relevant because the several metrics analyse the same elements. Our complexity metric focus on the complexity introduced by transactions and the complexity of the interactions, like other metrics do. Concerning cohesion and coupling, our metrics measure how cluster domain entities are visible outside the cluster and how they are accessed together in the context of a functionality, which corresponds to the level of dependencies between clusters, how many domain entities are not encapsulated, and whether all the cluster domain entities contribute to the single responsibility principle.

C. Future Work

As a consequence of the results of this research and the learned lessons we identify the following topics for future work:

- Experiment with machine learning techniques using the metrics as fitness functions to infer which combination of similarity measures is more adequate for a concrete monolith system;
- Further explore the results of the dynamic collection of data, in terms of the frequency of each of the functionalities, and define new similarity measures to verify if it can generate better decompositions;
- Investigate other sequence compression algorithms with the purpose of decreasing the JSON file size and also the time the Mono2Micro framework takes to process it;
- Analyse more monolith systems with different technology stacks using dynamic analysis to reduce the existing bias on the regression models.

VII. Conclusions

The migration of monolith systems to the microservices architecture is a complex problem that software development teams have to address when systems become more complex and larger in scale. Therefore, it is necessary to develop the methods and tools that help and guide them on the migration process. One of the most challenging problems is the identification of microservices. Several approaches have been proposed to automate such identification, which, although following the same steps, use different monolith analysis techniques, similarity measures, and metrics to evaluate the quality of the system.

In this thesis, two monolith systems were analysed to study the impact of applying static and dynamic analysis on the quality of the automatically generated decompositions as well as whether a particular combination of similarity measures provides better decompositions.

From the results of this research, it was concluded that there is no particular similarity measure that generates the best decompositions, considering the quality metrics for complexity, coupling, cohesion, and performance, and we raise the hypothesis that it may depend on particular characteristics of the monolith. Therefore, we suggest the use of our analyzer that generates decompositions using an extensive combination of similarity measures to find the one that has the best quality.

Moreover, as result of the executed analysis, we concluded that different monolith analysis techniques generate decompositions that do not outperform each other, but, it was clear that the effort required by the dynamic analysis is much superior and resulted in less coverage. Although the cost is much higher, both systems were extensively dynamically analyzed, and compared with the static analysis, which is a unique effort with no precedents. Future experiments can be done, but this work already contains, in this aspect, an important and novel contribution.

Lastly, a new metric concerning performance was introduced which proved to be correctly correlated with the maintainability ones. This addition is unequivocally out of the context of state consistency, nonetheless it also plays a key role in a microservices architecture.

As additional contributions, (i) the gathered data from the evaluated monolith systems, using dynamic analysis, is publicly available and can be used by third parties to do further research, (ii) the dynamic data collection was implemented to be as configurable and extensible as possible such that it can handle a wider variety of code bases with different technology stacks that are built using Java in a long-term view.

VIII. Data Availability

The data used and produced in this research is available at https://github.com/socialsoftware/mono2micro/tree/master/data/dynamic

References

[1] N. Santos and A. Rito Silva, "A complexity metric for microservices architecture migration," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 169–178.

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

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

[4] D. Taibi and V. Lenarduzzi, "On the definition of microservice bad smells," *IEEE Software*, vol. 35, no. 3, pp. 56–62, 2018.

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

[6] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, J.-M. Bruel, M. Mazzara, and B. Meyer, Eds. Cham: Springer International Publishing, 2019, pp. 128–141.

[7] S. Tyszberowicz, R. Heinrich, B. Liu, and Z. Liu, "Identifying microservices using functional decomposition," in *Dependable Software Engineering. Theories, Tools, and Applications*, X. Feng, M. Müller-Olm, and Z. Yang, Eds. Cham: Springer International Publishing, 2018, pp. 50–65.

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

[9] N. Ford, R. Parsons, and P. Kua, *Building Evolutionary Architectures*, 1st ed. O'Reilly, 10 2017.

[10] 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: Association for Computing Machinery, 2018, p. 1–6. [Online]. Available: https://doi.org/10.1145/3242163.3242164

[11] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proceedings. 12th IEEE International Workshop on Program Comprehension, 2004.* IEEE, 2004, pp. 194–203.

[12] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *J. Artif. Int. Res.*, vol. 7, no. 1, p. 67–82, Sep. 1997.

[13] D. G. Reichelt, S. Kühne, and W. Hasselbring, "Peass: A tool for identifying performance changes at code level," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2019, pp. 1146–1149.

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

[15] M. Ahmadvand and A. Ibrahim, "Requirements reconciliation for scalable and secure microservice (de)composition," in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, Sep. 2016, pp. 68–73.

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

[17] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *2016 IEEE International Conference on Services Computing (SCC)*, June 2016, pp. 813–818.

[18] L. Baresi, M. Garriga, and A. De Renzis, "Microservices identification through interface analysis," in *Service-Oriented and Cloud Computing*, F. De Paoli, S. Schulte, and E. Broch Johnsen, Eds. Cham: Springer International Publishing, 2017, pp. 19–33.

[19] S. Klock, J. M. E. M. V. D. 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)*, April 2017, pp. 11–20.

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

[21] 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.

[22] L. De Lauretis, "From monolithic architecture to microservices architecture," in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2019, pp. 93–96.

[23] M. H. Gomes Barbosa and P. H. M. Maia, "Towards identifying microservice candidates from business rules implemented in stored procedures," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 41–48.

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

[25] Y. Zhang, B. Liu, L. Dai, K. Chen, and X. Cao, "Automated microservice identification in legacy systems with functional and non-functional metrics," in *2020 IEEE International Conference on Software Architecture (ICSA)*, 2020, pp. 135–145.

[26] F. Ponce, G. Márquez, and H. Astudillo, "Migrating from monolithic architecture to microservices: A rapid review," in *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*, 2019, pp. 1–7.

[27] J. Bogner, S. Wagner, and A. 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*, ser. IWSM Mensura '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 107–115. [Online]. Available: https://doi.org/10.1145/3143434.3143443

[28] M. Hirzalla, J. Cleland-Huang, and A. Arsanjani, "A metrics suite for evaluating flexibility and complexity in service oriented architectures," in *Service-Oriented Computing – ICSOC 2008 Workshops*, G. Feuerlicht and W. Lamersdorf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 41–52.

[29] 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)*, 2007, pp. 329–340.

[30] C. Patel, A. Hamou-Lhadj, and J. Rilling, "Software clustering using dynamic analysis and static dependencies," in *2009 13th European Conference on Software Maintenance and Reengineering*, 2009, pp. 27–36.

[31] A. Krause, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kroger, "Microservice decomposition via static and dynamic analysis of the monolith," in *2020 IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2020, pp. 9–16.

[32] T. Matias, F. F. Correia, J. Fritzsch, J. Bogner, H. S. Ferreira, and A. Restivo, "Determining Microservice Boundaries: A Case Study Using Static and Dynamic Software Analysis," *arXiv e-prints*, p. arXiv:2007.05948, Jul. 2020.