Automatic Detection of Anomalies in the Migration to Microservices Architectures

(extended abstract of the MSc dissertation)

VALENTIM ROMÃO, Instituto Superior Técnico, Portugal
SUPERVISORS: PROFESSOR LUÍS RODRIGUES and PROFESSOR VASCO MANQUINHO, Instituto Superior Técnico, Portugal

The microservices architecture allows structuring an application as a set of loosely coupled services. This architecture has several advantages, such as modularity and scalability, which motivate the migration of monoliths to microservices, despite the challenges posed by the lack of isolation between functionalities that require invoking multiple microservices. In a monolithic application, each functionality is typically executed as a single transaction that accesses a single database with ACID properties. In a microservices architecture, a functionality may be divided into multiple independent sub-transactions and each may be executed by a different microservice. The interleaving between these sub-transactions, when the functionalities execute concurrently, may lead to unexpected results, also called anomalies. In this work, we present a tool capable of automatically detecting these anomalies, as well as an experimental evaluation of the tool, using microbenchmarks and several real-world applications.

1 INTRODUCTION

The microservices architecture structures an application as a set of loosely coupled components (services), contrasting with the traditional monolithic architecture composed of a single centralized component. This architecture has several advantages when compared with the monolithic architecture. Firstly, each microservice only implements the logic related to a small subset of the entities managed by the application, making the code of each service more cohesive and easier to develop and maintain. Secondly, each microservice can be developed independently, allowing the development of the application as a whole to be more agile. Thirdly, the architecture offers a more flexible management of the system, since the microservices can be managed independently. Due to these advantages, several companies are currently adopting the microservices architecture when developing their applications. In many cases, companies also migrate their pre-existing monolithic applications to the microservices architecture [14].

Although the microservices architecture has many advantages, it also introduces challenges. A monolithic application is traditionally composed of several functionalities, each commonly defined as a transaction that executes in a single database. These transactions offer the *ACID* (Atomicity, Consistency, Isolation, Durability) properties, therefore guaranteeing the isolation between concurrent executions of these functionalities. When migrating from a monolithic application to microservices, a functionality may need to be chopped into several sub-transactions, with each sub-transaction possibly executing in a different microservice, breaking the isolation of the functionality as a whole.

Typically, the microservices use the *database per service* [10] design pattern. Although each sub-transaction is isolated from the remaining sub-transactions that execute in the same microservice, the concurrent execution of functionalities may lead to interleavings between sub-transactions that execute in different microservices, something that did not occur in the monolith. This may lead to unexpected results, also called anomalies, which derive from non-serializable executions of transactions.

The number of anomalies that can arise during the execution of functionalities in microservices depends on how the monolith is decomposed, in particular the number of microservices that interact with each other and which entities are managed by each microservice. Finding these anomalies using an accurate analysis technique can be useful, considering that concurrency anomalies are notoriously difficult to identify via testing [8]. One way of handling this problem is by having a tool capable of generating all the possible interleavings and comparing the executions that can occur in the monolith with the executions that can occur in a given decomposition since these new anomalous executions result from the migration. To the best of our knowledge, existing tools aimed at doing this sort of analysis do not account for all the monolith to microservices migration aspects, such as the chopping of a transaction into a sequence of independent sub-transactions, as well as the effects of the sub-transactions reading mutually inconsistent versions of remote objects by accessing their microservice's local storage. In this thesis, we describe the design and implementation of a novel tool, named *Microservices Anomaly Detector* (*MAD*), which can generate all the possible interleavings that originated from the decomposition of a monolith into microservices. The tool works by encoding the problem in a *Satisfiability Modulo Theories* (SMT) formula and using Z3 [4] to find the satisfiable assignments. We also performed an experimental evaluation to assess the accuracy and applicability of *MAD* using *microbenchmarks* and test cases inspired by real-world codebases.

2 RELATED WORK

In this section, we will present several tools with implementations and objectives similar to the ones of our tool. We divide the related work into three distinct classes: (1) tools that do not detect anomalies but provide an estimate of the number of anomalies that may occur (typically using heuristics); (2) tools that detect anomalies without having access to the program's code; (3) tools that detect anomalies having access to the source code.

2.1 Heuristics for Anomaly Awareness

There are several proposals of algorithms to estimate the number of anomalies that can occur when applying a given decomposition of a monolith into microservices. These algorithms are based on heuristics and intend to help the programmer choose the best decomposition for the migration, considering that the programming effort associated with a decomposition will be proportional to the number

of anomalies estimated. In this category, we highlight the works presented in "A Complexity Metric for Microservices Architecture Migration" (*CMMAM*) [12], and "Mono2Micro - From a Monolith to Microservices: MetricsRefinement" [2]. These works use heuristics that analyse the source code to identify patterns in the accesses to the entities, which may expose intermediate states and generate anomalies. For example, cases where the execution of a functionality resorts to several different microservices. One advantage of these heuristics is that they can be executed efficiently and applied to monoliths with a high number of lines of code. One disadvantage is that they only provide approximate values, which may be an underestimate or an overestimate depending on the heuristic. In particular, they can present false positives since they do not have a fine enough granularity to distinguish that some access may be performed to different instances of the same entity.

2.2 Anomaly Detection using Black Box Approaches

The black box approaches for anomaly detection do not need to have access to the program's code that they are analysing. The procedure of these tools consists of giving input values and observing the output values, and based on this information determining if any anomaly occurred by comparing the differences between the observed behaviour and the expected behaviour. *Cobra* [13] and *MonkeyDB* [3] are examples of tools that use this approach. These tools are more generic than the heuristics presented before because they do not need to have access to the system's code that they are analysing. However, their analysis may be incomplete, if, when generating the input values and the scheduling of the operations, never occurs a specific combination that would cause an anomalous behaviour from the system.

2.3 Anomaly Detection using White Box Approaches

At last, we analyse works that use a white box approach (have access to the source code) to detect anomalies. These tools first extract information from the program, such as the transactions, types of parameters, and execution order, among others, and, based on this information, consider all the program's executions that have anomalies. Examples of tools that use this approach are *ANODE* [6], *CLOTHO* [9], and *CLOTHO+* [11]. These tools have two advantages with regard to the previous tools, namely, they do not generate false positives and can perform a complete analysis, which detects all the anomalies that may occur. The main disadvantages of these tools consist of the need to have access to the source code of the system being analysed, and their temporal and spatial complexities, due to the time and memory space required to generate all the possible anomalous executions of a system.

Since *MAD* uses *CLOTHO* as its basis, we present this tool with more detail. *CLOTHO* is a tool designed to analyse the transactions of a storage system that provides weak consistency semantics, to detect if there are any serializability anomalies. *CLOTHO* has additionally the capacity to simulate the anomalies found in concrete executions. The way this tool's analysis works consists of creating *First Order Logic* (FOL) formulas where the satisfiable assignments correspond to cyclic graphs, with the vertices being the operations and the edges the relations between operations. The types of edges considered are: *ST* (same transaction); *RW* (read followed by a write); *WR* (write followed by a read); and *WW* (write followed by a write). The cyclic anomalous graphs are defined as having at least one *ST* edge and at least two dependency edges (*RW*, *WR*, *WW*). These graphs represent non-serializable executions of the transactions (anomalies) [1].

First, *CLOTHO* receives a Java program and converts it to an *Abstract Representation* (AR), which resembles an SQL-like language and is described in *CLOTHO*'s paper. This step eases the extraction of information from the input program, such as the edges between operations that can access the same object. Second, using the program in the AR, the goal is to construct a FOL formula. This formula is the conjunction of the following five sets of constraints: $\varphi_{context}$ to represent the values that are plausible to be in the database; φ_{db} to represent the consistency and isolation guarantees provided by the database; $\varphi_{dep\rightarrow}$ to represent the dependency arrows between operations that belong to the dependency cycle; $\varphi_{\rightarrow dep}$ to represent the dependency arrows between operations that do not belong to the dependency cycle, and; $\varphi_{anomaly}$ to bound the possible anomaly structures to a maximum number of transactions in a serial execution, a maximum number of transactions in a concurrent execution, and a maximum length for a dependency cycle. Third, after having the FOL formula built, *CLOTHO* uses an SMT solver (in *CLOTHO*'s case Z3 [4]), to compute the assignments that satisfy the formula, each of them representing an abstract execution that contains an anomaly. In the cases where the formula is unsatisfiable, this means that no anomalies were detected within the bounds defined for the formula construction. Finally, *CLOTHO* allows the user to simulate the concrete execution of the anomalies found, however, this aspect falls outside the scope of our work.

CLOTHO+ [11] presents three extensions to *CLOTHO* to model partially the behaviour of the microservices. Namely, the implementation of the consistency models formalized in *CLOTHO*, annotations to indicate in which microservice each transaction executes, and the introduction of a relation to indicate that two operations belong to the same class/type of transaction.

2.4 Comparison

Table 1 presents a comparison of the tools. We split the table into two sections. The first section (Properties) refers to the characteristics of the tools. We divided this section into three columns which capture, respectively, the approach used by the tool to analyse a system (Analysis), the consistency guarantees that the tool is expecting the system under test to respect, considering the ones the tool can be extended to support (Consistency Model), and if the tool is oriented to analyse microservices architectures (Microservices Oriented). The second section (Techniques) refers to the mechanisms used by the tools to fulfil their purpose. We divided this section also in three columns which capture, respectively, how the tool identifies the existence of anomalies (Method), if the tool uses an SMT solver (SMT Solver), and which type of executions the tool analyses (Executions Analysed) ("-" means that the tool does not consider different executions of the system). Note that our tool, *MAD*, offers a white box analysis applicable to any consistency model and that takes into account all the aspects of the microservices architecture, which is something none of the other tools do.

Table 1. Characterization of existing tools.

	Properties			Techniques		
	Analysis	**Consistency Model**	**Microservices Oriented**	**Method**	**SMT Solver**	**Executions Analysed**
CMMAM	Heuristic	Eventual Consistency	Yes	Static Analysis	No	-
Metrics Refinement	Heuristic	Eventual Consistency	Yes	Static Analysis	No	Abstract
Cobra	Black Box	Serializability	No	Testing	Yes	Abstract
MonkeyDB	Black Box	Any	No	Testing	No	Concrete
ANODE	White Box	Any	No	Static Analysis	Yes	Abstract
CLOTHO	White Box	Any	No	Static Analysis	Yes	Abstract/ Concrete
CLOTHO+	White Box	Any	Partial	Static Analysis	Yes	Abstract
MAD	White Box	Any	Yes	Static Analysis	Yes	Abstract

3 MAD

In this section, we describe the design and implementation of the *Microservices Anomaly Detector* (*MAD*) tool, which automatically detects anomalies that may occur when applying a given decomposition from monolith to microservices. We focused our efforts on developing a precise tool that avoided false negatives and false positives while considering all the aspects related to the migration to microservices. In Section 3.1, we present an overview of *MAD*'s design, and, in Section 3.2, we explain the implementation of *MAD*.

3.1 Overview

MAD takes as input the source code of a monolithic implementation of an application and a high-level description of how the monolith is decomposed into multiple microservices. In the current version, the source code must be a *Java program* written using the JDBC syntax (uses SQL queries to access the entities). The decomposition of the monolith is expressed as the clustering of the domain entities into *aggregates* [7] (entities grouped in the same cluster are assumed to be managed by the same microservice) and is represented by a JSON file (*Decomposition File*). Using this input, *MAD* executes the pipeline presented in Figure 1, which we will now explain.

During the compilation to the AR, *MAD* performs two steps. First, the *AR Compiler*, which extracts the monolith's transactions (designated by us as original transactions), parameters, and expressions, originating the *Monolith AR Program*. Second, the *Transaction Chopping* where the representations of the sub-transactions are generated based on the original transactions and the decomposition considered, originating the *Microservices AR Program*.

Often, the AR program is too complex to be represented in a single encoding that can be easily analysed. Employing a divide and conquer strategy, *MAD* generates subsets of the original transactions (*Functionalities Subsets*). Analysing these subsets independently allows for the analysis of the whole problem to be simpler and done in a reasonable time. Therefore, instead of creating a single SMT formula with all the assertions, *MAD* creates several formulas, one for each subset, with each formula only having the assertions related to the original transactions of their given subset.

When constructing the formulas, *MAD* uses already existing assertions to encode the basic behaviour of a distributed system and the logic to detect cyclic graphs (executions with anomalies). However, since those were not enough to model the context of our problem, we had to add and adapt assertions of the formula. To illustrate that in a graph there is a relation between two operations of the same original transaction that are in different sub-transactions, we consider a new edge type, *SOT*. To model where each operation executes and the operations' visibility effects, we add the notion of microservices together with the usage of consistency model assertions.

After the SMT solver (Z3 [4]) finishes its analysis, we obtain the *Satisfiable Assignments*, which represent the anomalies found. To present more metrics, *MAD* has a *Metrics Extractor*, which applies a pattern-matching technique to enumerate the number of anomalies of each type, groups the anomalies by sets of sub-transactions considering the sub-transactions involved in each anomaly, and counts the number of occurrences of each sub-transaction in the anomalies.

3.2 Architecture

We will now address how each of the different aspects of *MAD*'s design are implemented in the tool. More specifically, the input files, the notions of sub-transactions and microservices, the consistency models, the search algorithm, and the metrics extractor.

3.2.1 Input Files. *MAD* receives as input a Java file with all the original transactions and their respective operations using JDBC and SQL queries to access the entities, and a JSON file that represents the decomposition and indicates the existent microservices and the entities assigned to each microservice.

3.2.2 Sub-transactions and Microservices Notions. To represent the migration, two essential aspects need to be considered. These aspects are: (1) sub-transactions, transactions that originated from the division of a transaction, and (2) microservices, the nodes where the sub-transactions will execute. We divided the task of modeling these aspects into three steps: 1) representing the sub-transactions; 2) introducing a feature to indicate that two transactions are related since they will be executed under the same functionality (original transaction); 3) introducing the concept of microservice and associating it with the operations.

Upon compiling the *Java program* to the *Monolith AR Program*, *MAD* proceeds to generate the representation of the sub-transactions considering the assignments between entities and microservices from the *Decomposition File*. For each original transaction, *MAD* applies the procedure described in Algorithm 1.

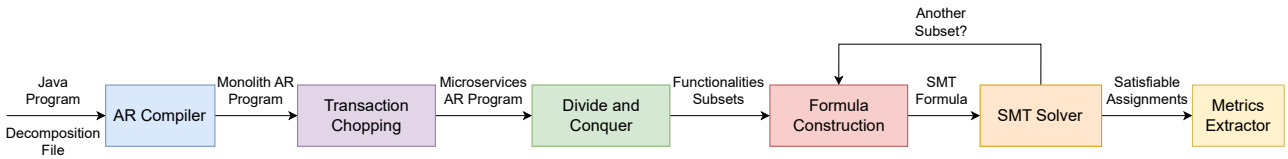

Fig. 1. *MAD*'s pipeline.

Algorithm 1: *MAD*'s Transaction Chopping

Input: body: decompiled Java body of the original transaction function, entitiesMicrosMap: mapping between entities and microservices, tables: list of input program's tables
Output: origTxn: representation of the original transaction

```
1  Function MADTransactionChopping(body, entitiesMicrosMap, tables)
2      name ← body.getMethod().getName();
3      origTxn ← OriginalTransaction(name);
4      unitHandler ← UnitHandler(body, tables);
5      unitHandler.extractParams();
6      currentSubTransactionIdx ← -1;
7      currentMicroservice ← "";
8      subTxns ← ∅;
       // Iterate over all the statements (operations)
9      foreach s in unitHandler.data.getStmts() do
10         entityName ← ((InvokeStmt)
             s).getQuery().getTable().getName();
           /* Create a new sub-transaction when the
              microservice of the operation's entity is
              different from the current sub-transaction's
              microservice                                  */
11         if currentMicroservice ≠ entitiesMicrosMap.get(entityName)
             then
12             currentMicroservice ←
                 entitiesMicrosMap.get(entityName);
13             currentSubTransactionIdx ← currentSubTransactionIdx
                 + 1;
14             newSubTxn ← Transaction(name + "_" +
                 currentSubTransactionIdx);
15             newSubTxn.setOriginalTransaction(name);
16             newSubTxn.setMicroservice(currentMicroservice);
17             newSubTxn.addStmt(s);
18             subTxns ← subTxns ∪ newSubTxn;
           /* Add the operation to the current
              sub-transaction when the microservice of the
              operation's entity is the same                */
19         else
20             subTxns.get(currentSubTransactionIdx).addStmt(s);
21     origTxn.addStmt(s);
```

The rationale behind Algorithm 1 is to iterate over the operations of an original transaction (*for* loop in Line 9) and generate sub-transactions based on the entities that are accessed by the operations. In each iteration, a verification is performed (*if else* in Lines 11 and 19) to assess if the entity accessed by the iteration's operation (*entityName*) belongs to a different microservice from the previous operation's entity (*currentMicroservice*), or not. If the entity belongs to a different microservice (Line 11), then the current microservice is updated and a new sub-transaction is created with the seen operation since it would execute in a different microservice. Else (Line 19), the operation is added to the most recent sub-transaction created since it would execute in the same microservice.

In the migration to microservices, some transactions may need to be chopped, which originates sub-transactions. Although these sub-transactions can be seen as independent transactions, they are still related to each other in the sense that they are part of the same functionality (original transaction) and they need to be executed sequentially, to provide the same behaviour as the original transaction. Therefore, to represent the relation between operations from different sub-transactions that belong to the same original transaction, we introduce an edge type, *SOT* (same original transaction).

To accomplish this, we add to the SMT formula a sort (*OT*), a data type (*OTType*), two formula functions (*ottype*, which receives an instance of an original transaction and returns its type, *OTType*; and *original_transaction*, which receives an instance of an operation and returns the instance of original transaction where it belongs, *OT*), and one predicate (*step_sibling*, which receives two instances of operations and returns true if they are related as operations of different sub-transactions but the same instance of an original transaction). The *ottype* function values are initialized at the beginning of the formula by expressing that for all the operations of a given operation type (*OType*), the instance of original transaction they belong to has to be of a specific *OTType*. The assertion can be found in Equation 1 with *otype* being a function that receives an instance of an operation and returns its type, *OType*. The *original_transaction* function is involved in the establishment of dependency edges and has its values assigned during the analysis when the SMT solver is looking for satisfiable assignments.

$$\forall o1((otype(o1) = op1) \Rightarrow (ottype(original\_transaction(o1)) = orig\_txn1)) \quad (1)$$

When the *SOT* edge is established between two operations, it indicates that both operations are in different transactions but belong to the same original transaction, which allows *MAD* to understand that the operations will always follow a specific order and that an interleaving between the two operations may lead to an anomaly.

Considering this new edge type, our definition of a cyclic anomalous graph is a cycle with at least one *ST* or *SOT* edge and at least two dependency edges (*RW*, *WR*, *WW*). In Equations 2 and 3, we present the anomalous cycles definition for lengths three and four, respectively. For these definitions, we use two predicates besides

the edge types: *D*, which receives two instances of operations and returns true if there is any dependency relation between them (*RW*, *WR* or *WW*); and *Any*, which also receives two instances of operations and returns true if there is any relation between them (*ST*, *SOT*, *RW*, *WR* or *WW*).

$$\forall o1, o2, o3((ST(o1, o2) \vee SOT(o1, o2)) \wedge D(o2, o3) \wedge D(o3, o1)) \tag{2}$$

$$\forall o1, o2, o3, o4((ST(o1, o2) \vee SOT(o1, o2)) \wedge D(o2, o3) \wedge Any(o3, o4) \wedge D(o4, o1)) \tag{3}$$

Regarding the microservices notion, we defined a new data type (*MType*) and a new formula function (*mtype*, which receives an instance of an operation and returns the microservice it executes on, *MType*) in the SMT formula. These are used to associate the operations with the microservice where they will be executed. The *mtype* function values, similar to the *ottype* function, are initialized at the beginning of the formula by expressing that for all the operations of a given operation type (*OType*), their microservice has to be of a specific *MType*. The assertion can be found in Equation 4.

$$\forall o1((otype(o1) = op1) \Rightarrow (mtype(o1) = M1)) \tag{4}$$

This is used to restrict the consistency models and to control the visibility between operations, as we will later discuss in Section 3.2.3.

In Figure 2, we present an example scenario of how the transactions would be divided in a migration from monolith to microservices. In this example, we consider two entities with only one attribute each, *Member* and *Item*, two transactions, *Txn1* and *Txn2*, and a decomposition where *Member* goes to microservice *M1* and *Item* goes to microservice *M2*. Both transactions, *Txn1* and *Txn2*, are composed of three operations. *Txn1*'s operations are reading an instance of *Member*, reading an instance of *Item* and updating the *Member*'s instance that was read. *Txn2*'s operations are reading an instance of *Member*, updating the *Member*'s instance that was read and updating an instance of *Item*. Considering that *Member* and *Item* will be in different microservices, the transactions *Txn1* and *Txn2* need to be divided into two sub-transactions, each of which executes in the microservice managing the entity they are accessing.

This division of the transactions allows for multiple executions where the sub-transactions interleave with each other, something that was not possible in the monolithic version and can lead to anomalies. One example of a new possible interleaving that leads to an anomaly can be seen in Figure 3. In this case, the execution of *Txn2* interleaves with the execution of *Txn1*, which leads to a non-serializable execution of the system, since *Txn1* sees an older value for $Member_1$, assuming that it executed before *Txn2*, but at the same time it sees the new value for $Item_1$ that was written by *Txn2*, assuming that it executed after *Txn2*, which is contradictory.

MAD detects this anomaly by finding the cyclic graph that can be seen in Figure 4. The cycle contains two *SOT* edges and two dependency edges (one *RW* edge and one *WR* edge) and represents the interleaving of the original transactions *Txn1* and *Txn2*.

3.2.3 Consistency Models. To make an analysis faithful to the environment where the microservice systems will execute, we assume two consistency models: Serializability and Eventual Consistency. Between operations of the same sub-transaction and other operations of the same microservice, we assume that they will respect Serializability. Between operations of different sub-transactions or that execute on different microservices, we assume Eventual Consistency [1]. By default, *MAD* enforces Eventual Consistency between the visibility effects of the operations. However, to enforce Serializability, we use the consistency models' assertions defined in *CLOTHO*'s paper [9] and implemented in *CLOTHO+* [11] with some adjustments so that Serializability is only applicable between operations that execute in the same microservice.

The consistency models' assertions rely on two relevant predicates, *vis* and *ar*. *vis* receives two instances of operations and returns true if the effects of the left operation are visible to the right operation. *ar* also receives two instances of operations and returns true if the left operation is executed before the right operation. The assertions are implemented as follows (our adjustments are in blue):

$$Read\_Committed = \forall o1, o2, o3(ST(o1, o2) \wedge vis(o1, o3) \wedge (mtype(o1) = mtype(o3)) \Rightarrow vis(o2, o3)) \tag{5}$$

$$Repeatable\_Read = \forall o1, o2, o3(ST(o1, o2) \wedge vis(o3, o1) \wedge (mtype(o1) = mtype(o3)) \Rightarrow vis(o3, o2)) \tag{6}$$

$$Linearizability = \forall o1, o2(ar(o1, o2) \wedge (mtype(o1) = mtype(o2)) \Rightarrow vis(o1, o2)) \tag{7}$$

$$Serializability = Read\_Committed \wedge Repeatable\_Read \wedge Linearizability \tag{8}$$

3.2.4 Search Algorithm. *MAD*'s search algorithm follows a *Divide and Conquer* principle. Instead of taking into account all the original transactions of the system at the same time, *MAD* restricts the search for cycles to only consider a subset of the original transactions per iteration. By using this method, we can divide a large problem into several smaller problems, which can be solved in a much shorter period of time. This search algorithm can also be applied to other cases, such as *CLOTHO*'s analysis, by adapting the algorithm to use the transactions (in *MAD* designated as sub-transactions), instead of the original transactions.

The first step of the search algorithm is to generate all the combinations of size smaller than the maximum cycle length (in our case four). The number of combinations generated when there are at least three original transactions can be obtained by replacing the value of n in $\sum_{x=1}^{3} {}^{n}C_{x}$ by the number of original transactions.

After generating the combinations, we proceed to restrict the original transactions of the operations that we are considering for the cycle in the SMT formula to be from the combination that is being assumed at that iteration. The anomalies can have at most the current cycle length minus 1 ($current\_cycle\_length - 1$) original

[1] https://martinfowler.com/articles/microservice-trade-offs.html

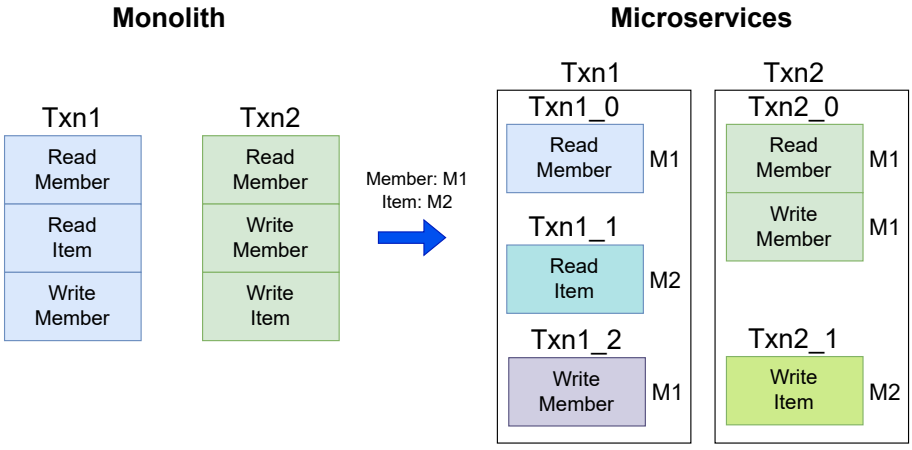

Fig. 2. Example scenario of how the transactions are divided in a migration from monolith to microservices.

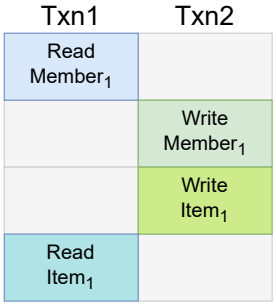

Fig. 3. Example of a possible interleaving that leads to an anomaly.

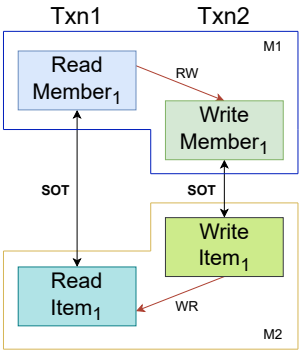

Fig. 4. *MAD*'s cyclic graph for the example anomaly.

transactions involved because they must have at least one ST or SOT edge, which implies that at least two operations share the same original transaction. Therefore, *MAD* only iterates over the combinations of size $current\_cycle\_length - 1$ in each cycle length analysis. As an example, if we have five original transactions, *order*, *sell*, *buy*, *look* and *update*, assume that the current cycle length is 3 and the combination [*order*, *sell*], then, complementing the cycle assertions from Equation 2, we would have the assertions defined in Equation 9 (for simplicity, we only present the assertions related with o1, since for o2 and o3 they would be the same only changing the operation instance used as argument).

$$\begin{aligned}
&\forall o1, o2, o3(((ottype(original_transaction(o1)) = order) \\
&\vee (ottype(original_transaction(o1)) = sell)) \\
&\wedge (\neg(ottype(original_transaction(o1)) = buy)) \\
&\wedge (\neg(ottype(original_transaction(o1)) = look)) \\
&\wedge (\neg(ottype(original_transaction(o1)) = update)) \wedge ...) \qquad (9)
\end{aligned}$$

Besides restricting each of the cycle's operations to belong to one of the original transactions of the combination considered, *MAD* also restricts the assertions that are added to each formula, so that each only includes the assertions related to the original transactions of their combination. This procedure minimizes the overall complexity of each SMT formula since it will only have the essential assertions for the analysis. By doing this, the search space is reduced and the analysis of each combination is done faster.

3.2.5 Metrics Extractor. Besides presenting the total number of anomalies found, *MAD* also displays two additional metrics, the number of anomalies by type of anomaly and the number of anomalies considering the sub-transactions involved in each anomaly.

The first additional metric is the number of anomalies by type of anomaly (dirty read, dirty write, lost update, write skew, and read skew). The technique we use consists of having a set of patterns only considering the types of edges between operations and, when an anomaly is detected, checking if the anomaly cycle matches any of the patterns. In the cases where the anomaly execution is more complex than our anomaly types' patterns and the cycle found does not match any of the patterns, then the anomaly is "Unclassified".

Regarding the second additional metric, it focuses on presenting two aspects. First, the number of anomalies that occur when the exact same set of sub-transactions is involved. This is used to capture the impact caused by the interaction between certain sub-transactions towards the number of anomalies. Second, the number of occurrences of each sub-transaction in the anomalies found. This

aspect helps to understand how relevant each sub-transaction is regarding the existence of anomalies in the system.

4 EVALUATION

In this chapter, we present the evaluation that we performed to assess *MAD* capabilities. In Section 4.1, we describe all the benchmarks that we used and the conditions under which the evaluation was performed. Section 4.2 presents a comparison between *MAD* and a heuristic approach tool oriented to microservices, "A Complexity Metric for Microservices Architectures Migration" (*CMMAM*) [12], also displayed in Table 1. At last, in Section 4.3, we present the results obtained by *MAD* when applied to real-world codebases.

4.1 Experimental Setup

For this experimental evaluation, we use two types of benchmarks: (1) handcrafted benchmarks that we name *microbenchmarks*; (2) real-world codebases that can be found on GitHub.

Regarding the *microbenchmarks*, we created three instances, each with a different set of transactions, however, all assuming the same context. The context consists of an application that has two entities, *Member* and *Item*, with *Member* having three attributes, *id*, *status* and *money*, and *Item* also having three attributes, *id*, *price* and *stock*. All the *microbenchmarks* assume the same decomposition, *Member* executing on microservice *M1* and *Item* executing on microservice *M2*. The purpose of these *microbenchmarks* is to illustrate the differences between *MAD* and *CMMAM*. *Microbenchmark1* targets dependencies between writes (WW) and is composed of two functionalities, *UpdateMI* that updates a member's status and an item's price, and *ResetMI* that resets a member's status and an item's price (writes 0 on both). *Microbenchmark2* targets accessing different instances of the same entity and is composed of two functionalities, *UItem1* that reads the status of a specific instance of *Member* (Member_1) and updates the price of a specific instance of *Item* (Item_1), and *UMember2* that reads the price of a specific instance of *Item* (Item_2) and updates the status of a specific instance of *Member* (Member_2). At last, *Microbenchmark3* targets accessing different attributes of the same entity and is composed of two functionalities, *UItem* that reads a member's status and updates an item's stock, and *UMember* that reads an item's price and updates a member's money.

To evaluate *MAD*'s applicability to real-world scenarios, we gathered several software applications that can be found on GitHub. The list of applications and their respective description is the following:

- **TPC-C** [2] is defined in the OLTP-Bench [5] project and simulates the behaviour of a delivery and warehouse management system;
- **jpabook** [3] simulates a shop where members can order items and track the delivery process;
- **spring-framework-petclinic** (petclinic) [4] simulates how a pet clinic operates regarding the interactions between owners, pets, and veterinarians, as well as the visits to the pet clinic;

[2]https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/tpcc
[3]https://github.com/holyeye/jpabook/tree/master/ch12-springdata-shop
[4]https://github.com/spring-petclinic/spring-framework-petclinic

Table 2. *MAD* and *CMMAM* microbenchmarks results.

	MAD #Anomalies	**CMMAM Complexity**
Microbenchmark1	3	0
Microbenchmark2	0	4
Microbenchmark3	0	4

- **myweb** [5] is an application that simulates the behaviour of the web allowing users to have roles and perform operations to interact with resources. The operations are create, read, update, and delete (CRUD);
- **spring-mvc-react** (react) [6] is a platform where users can post questions and answers with tags associated with them. Besides that, the system also allows users to upvote or downvote publications, which has an impact on the users' popularity.

For both types of benchmarks, we use *MAD* considering four as the maximum cycle length. The evaluation was performed in a virtual machine with 32 CPUs and 128GB of RAM using Ubuntu 18.04.4 LTS, Java 8, and version 4.12.3 of Z3.

4.2 Comparison between *MAD* and a Heuristic Approach Tool

The comparison between *MAD* and *CMMAM* consists of the analysis of the results that can be obtained when applying each of the tools to the three *microbenchmarks*. The results can be found in Table 2.

First, in *Microbenchmark1*, the migration leads to anomalies when instances of the *UpdateMI* and *ResetMI* functionalities execute concurrently, since their interleaving results in an inconsistent database state at the end of the execution. *MAD* detects three anomalies, whereas *CMMAM* returns that the given decomposition leads to no problems. The anomalies *MAD* detects are the interleaving between different instances of *UpdateMI*, and instances of *UpdateMI* with instances of *ResetMI*. *CMMAM* does not alert this type of situations, since it only accounts for observable states of the application.

Second, in *Microbenchmark2*, there are no anomalies, because functionalities *UItem1* and *UMember2* always access different instances of the entities, $\textit{Member}_1$ and $\textit{Item}_1$, and $\textit{Member}_2$ and $\textit{Item}_2$, respectively. Since *MAD* considers the instance being accessed in each operation, it returns that there are zero anomalies. On the other hand, *CMMAM* returns that the migration has complexity four, due to the fact that it only considers the entity that is being accessed.

Third, in *Microbenchmark3*, similar to *Microbenchmark2*, there are no anomalies. However, in this case, both functionalities, *UItem* and *UMember*, can access the same instances of the entities. The difference is that the operations are accessing different attributes. Therefore, there are no conflicts between the values that are read and the values that are written. *MAD* considers the attributes accessed and returns zero anomalies, whereas *CMMAM*, by not considering those aspects, returns complexity four for this migration.

[5]https://github.com/Jdoing/myweb
[6]https://github.com/noveogroup-amorgunov/spring-mvc-react

4.3 *MAD* Results for Real-World Applications

To evaluate *MAD*'s applicability to real-world scenarios, our process can be divided into four steps: 1) gathering five monolithic applications that can be found on GitHub; 2) adapting them to the syntax processed by *MAD* (adjusted some queries and used JDBC instead of JPA considering each controller method as a transaction); 3) generating two decompositions of each application with the help of a migration tool [7] that supports programmers on the task of grouping the entities by the microservices; 4) using *MAD* on each decomposition. We assume that the monolithic version of each application is correct and contains no anomalies. Therefore, any anomaly that arises in the microservices versions must have resulted from the migration. Besides the overall results, we also showcase *MAD*'s ability to display the number of anomalies found in each decomposition by anomaly type and set of sub-transactions, together with the number of occurrences of each sub-transaction in the anomalies. We are only considering the analysis of the sub-transactions metrics for one decomposition, but it could be done for the other decompositions as well. At last, we discuss the improvement provided by the search algorithm by comparing *MAD*'s performance without and with it when applied to the same decompositions.

4.3.1 Overall Results. As we previously stated, for this part of the evaluation, we use five monolithic applications (*mono*) with two decompositions to microservices for each application. The microservices decompositions are: *"best"*, which is the decomposition with the highest *Silhouette Score* (a metric used to assess how well the clustering of the entities is done) calculated by the migration tool [7]; and *full*, which is a decomposition where each entity is managed by a different microservice, resulting in the largest functionalities' division possible in a microservices migration, illustrating the worst case scenario in terms of anomalies. In Table 3, we present the results from applying *MAD* to each of the decompositions.

By analysing Table 3, we can observe several relevant aspects associated with *MAD*'s analysis of the decompositions. As expected, the number of anomalies found in the *full* decompositions is bigger than in the *"best"* decompositions since they originate more sub-transactions. By looking at the number of anomalies found in each application's decompositions, *MAD* allows programmers to assess how problematic each decomposition will be, therefore enabling them to make a more informed decision when migrating to microservices. For instance, in *jpabook* and *react*, even though their *"best"* decompositions have the highest *Silhouette Scores* out of their possible decompositions, they still have anomalies. Note that in the *petclinic* application no decomposition led to anomalies. After analysing the application, we noticed that this occurred because the application's operations are mostly reads and the functionalities tend to be short and access few entities.

Although *MAD* can provide these results with precision, in some cases, it takes a relatively long period of time to perform its analysis, as can be seen from the results in column "Execution Time [s]". This problem is less common in simple applications or when the microservices version of the application does not originate many sub-transactions. However, in complex applications with a high number of functionalities and/or sub-transactions, *MAD* will tend to take a larger amount of time.

Another aspect one can notice is that all the *"best"* decompositions have two microservices. From our understanding, this phenomenon may be related to Martin Fowler's Strangler Fig pattern [7], where the migration to microservices is done incrementally by extracting a few services at a time, considering the coupling between entities. The *Silhouette Score* might indirectly take this into account, resulting in its value suggesting that from the monolithic implementation, the most appropriate decomposition to microservices is to migrate to two microservices. Moreover, in most cases, the *"best"* decompositions lead to a migration where no anomalies emerge. However, in two applications (*jpabook* and *react*), this does not occur since the *"best"* decompositions for these cases required the functionalities to be more divided, therefore generating more sub-transactions and allowing more interleaving between functionalities.

4.3.2 Anomalies Found per Type. In Table 4, we present the number of anomalies found for each decomposition by type of anomaly. These results may be helpful to the programmers since they allow them to anticipate unexpected behaviours that the application might have after the migration to microservices following a given decomposition. Another aspect to point out is the relatively high number of anomalies in the column "#Unclassified". This happens due to the fact that the types' counters are not incremented when there is an execution with two or more anomalies and our patterns only consider the minimum number of operations to describe each type of anomaly. For example, in a dirty read, it is only required two writes in one functionality but in different sub-transactions and one read in a different functionality, so that it is possible to have an execution where an intermediate value is read. However, if there is any extra operation in the execution considered, then that execution will not be identical to the one of our dirty read pattern, and, therefore, will be considered "Unclassified".

4.3.3 TPC-C Anomalies Found per Sub-Transactions. In Listing 1, we present the number of anomalies found by sets of sub-transactions and the number of occurrences of each sub-transaction in the anomalies of the *TPC-C full* decomposition. By analysing these results, one can notice that two functionalities alone cause most of the anomalies. These functionalities are "payment" and "delivery", which are responsible for 44 (4+20+20) and 24 (4+4+4+4+4+4) anomalies, respectively, out of the 98 anomalies found. As expected, the sub-transactions of these functionalities ("payment_0", "payment_1", "payment_2", and "delivery_0", "delivery_1", "delivery_2", "delivery_3") occur in a large number of anomalies. Based on these results, one possible approach to mitigate this issue is to analyse the entities involved in those functionalities, and either reorder the operations, so that the operations that access the same entity are sequential, or design a decomposition where the entities managed by each of these functionalities are in the same microservice.

4.3.4 Performance of the Search Algorithm. To conclude *MAD*'s evaluation, we address the impact that our divide and conquer search algorithm has regarding *MAD*'s performance, as well as its capability to mitigate the existence of applications/decompositions that are too complex for *MAD* to analyse in a reasonable period of time. Table 5 presents *MAD*'s execution time to analyse each of the

[7] https://martinfowler.com/bliki/StranglerFigApplication.html

Table 3. *MAD* overall results.

Application	#Entities	#Functionalities	Decomposition	#Microservices	#Sub-Transactions	#Anomalies	Execution Time [s]
TPC-C	9	5	mono	1	5	0	13
			"best"	2	6	0	13
			full	9	22	98	994
jpabook	5	10	mono	1	10	0	33
			"best"	2	15	70	356
			full	5	21	79	465
petclinic	6	14	mono	1	14	0	77
			"best"	2	14	0	79
			full	6	27	0	119
myweb	5	20	mono	1	20	0	259
			"best"	2	22	0	265
			full	5	32	7	400
react	5	23	mono	1	23	0	571
			"best"	2	28	45	1677
			full	5	39	58	1951

Table 4. *MAD* anomalies found per type.

Application	Decomposition	#Dirty Reads	#Dirty Writes	#Lost Updates	#Lost Updates/ Write Skews	#Read Skews	#Unclassified	#Total
TPC-C	mono							0
	"best"							0
	full		13			27	58	98
jpabook	mono							0
	"best"			3	6	10	51	70
	full			3	7	15	54	79
petclinic	mono							0
	"best"							0
	full							0
myweb	mono							0
	"best"							0
	full				2	4	1	7
react	mono							0
	"best"					12	33	45
	full					12	46	58

decompositions without and with the usage of the search algorithm that we described in Section 3.2.4.

From the analysis of Table 5, we can observe that the divide and conquer search algorithm does not always provide a performance improvement. However, for long analyses, it can significantly shorten their analysis time. The reason behind this is that the search algorithm was developed to mitigate the time and space complexities of *MAD* for large applications/decompositions, unintentionally neglecting the performance for simpler cases. This occurs because the search algorithm originates an overhead to *MAD*'s analysis by requiring that *MAD* unnecessarily iterates over combinations with no anomalies. Without the search algorithm, *MAD* would not need to do that since if after a search iteration it did not find any anomalies, it would increment the cycle length considered or end the analysis if it reached the maximum cycle length assumed. For complex cases, the search algorithm presents a significant performance improvement since it manages to simplify the SMT formula and mitigate the time and space complexities by not having to find anomalies while considering all the original transactions at the same time. Another positive aspect of the search algorithm is that it enables the analysis of complex cases to be performed under a timeout period (4 hours = 14400 seconds) that we defined as the reasonable amount of time a programmer would wait for the analysis to be complete. Without the search algorithm, *MAD* exceeds the timeout limit when analysing decompositions *"best"* and *full* of the *react* application, only finding 6 anomalies out of 45 and 6 anomalies out of 58, respectively.

5 CONCLUSIONS AND FUTURE WORK

In this thesis, we proposed an approach to automatically detect the anomalies that may appear when one migrates from a monolith to microservices following a given decomposition. We developed *MAD*, a tool that implements the proposed approach. The thesis described *MAD*'s design and implementation, which includes several strategies to improve its performance and scalability. We performed an experimental evaluation of *MAD*, aiming to compare the results with alternative tools and assess its performance when used with

Listing 1. Number of TPC-C *full* anomalies by sub-transactions.

```
[newOrder_3, newOrder_7, orderStatus_1, orderStatus_2]: 2/98
[delivery_0, delivery_2, newOrder_4, newOrder_7]: 8/98
[delivery_1, delivery_2, newOrder_3, newOrder_7]: 8/98
[delivery_0, delivery_1, newOrder_3, newOrder_4]: 4/98
[newOrder_6, newOrder_7, stockLevel_1, stockLevel_2]: 1/98
[newOrder_2, newOrder_7, stockLevel_0, stockLevel_1]: 1/98
[newOrder_1, newOrder_2, payment_0, payment_1]: 2/98
[payment_0, payment_1]: 4/98
[payment_1, payment_2]: 20/98
[payment_0, payment_2]: 20/98
[delivery_0, delivery_3]: 4/98
[delivery_1, delivery_3]: 4/98
[delivery_0, delivery_2]: 4/98
[delivery_0, delivery_1]: 4/98
[delivery_1, delivery_2]: 4/98
[delivery_2, delivery_3]: 4/98
[newOrder_2, newOrder_6]: 4/98
delivery (delivery_0): 24/98
delivery (delivery_1): 24/98
delivery (delivery_2): 28/98
delivery (delivery_3): 12/98
newOrder (newOrder_1): 2/98
newOrder (newOrder_2): 7/98
newOrder (newOrder_3): 14/98
newOrder (newOrder_4): 12/98
newOrder (newOrder_6): 5/98
newOrder (newOrder_7): 20/98
orderStatus (orderStatus_1): 2/98
orderStatus (orderStatus_2): 2/98
payment (payment_0): 26/98
payment (payment_1): 26/98
payment (payment_2): 40/98
stockLevel (stockLevel_0): 1/98
stockLevel (stockLevel_1): 2/98
stockLevel (stockLevel_2): 1/98
```

Table 5. MAD performance comparison without (w/o) and with (w/) the search algorithm (SA).

Application	Decomposition	w/o SA [s]	w/ SA [s]
TPC-C	mono	7	13
	"best"	6	13
	full	2767	994
jpabook	mono	6	33
	"best"	5996	356
	full	7133	465
petclinic	mono	9	77
	"best"	7	79
	full	13	119
myweb	mono	11	259
	"best"	11	265
	full	248	400
react	mono	114	571
	"best"	(timeout)	1677
	full	(timeout)	1951

real-world applications. The results highlighted not only how *MAD*'s analysis can be more accurate than a heuristic approach, but also that it can be applied to existing applications, providing insights regarding how one can migrate their monolithic application to a microservices architecture.

Although we have incorporated in *MAD* techniques to process realistic use cases with reasonable performance, *MAD* still has some limitations. First, even though our search algorithm reduced the time and space complexities of the analysis, it can still be a problem for more complex cases. Second, our pattern-matching technique to determine the anomaly type is simple, lacking the notions that would allow it to assign the anomalies to each type better. Third, *MAD* only considers one replica of each node, therefore not being able to reproduce scenarios where there is replication of the nodes, and where the impact of the consistency model enforced between replicas would also have to be taken into account. At last, *MAD* lacks the notion of association between entities when they are in different microservices, such as JPA relationships, foreign keys, and semantic invariants that the system must respect. For future work, we consider that addressing these limitations may represent a significant improvement to *MAD* since it would allow the tool to be faster, more accurate, and able to target a wider range of scenarios.

6 ACKNOWLEDGMENTS

This work was supported by FCT - Fundação para a Ciência e a Tecnologia, as part of the projects with references UIDB/50021/2020 and DACOMICO (financed by the OE with ref. PTDC/CCI-COM-/2156/2021).

REFERENCES

[1] Atul Adya, Barbara Liskov, and Patrick O'Neil. 2000. Generalized Isolation Level Definitions. In *Proceedings of the 16th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, San Diego (CA), USA.

[2] João Almeida and António Silva. 2020. Monolith Migration Complexity Tuning Through the Application of Microservices Patterns. In *Proceedings of the 14th European Conference on Software Architecture (ECSA)*. L'Aquila, Italy. https://doi.org/10.1007/978-3-030-58923-3_3

[3] Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. 2021. MonkeyDB: Effectively Testing Correctness under Weak Isolation Levels. *Proceedings of the ACM on Programming Languages* 5, OOPSLA, Article 132 (October 2021), 27 pages. https://doi.org/10.1145/3485546

[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems (TACAS) (TACAS'08)*. Budapest, Hungary, 4 pages.

[5] Djellel Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proceedings of the VLDB Endowment* 7, 4 (December 2013), 277–288. https://doi.org/10.14778/2732240.2732246

[6] Kartik Nagar and Suresh Jagannathan. 2018. Automated Detection of Serializability Violations Under Weak Consistency. In *Proceedings of the 29th International Conference on Concurrency Theory (CONCUR)*, Sven Schewe and Lijun Zhang (Eds.). Beijing, China. https://doi.org/10.4230/LIPIcs.CONCUR.2018.41

[7] Luís Nunes, Nuno Santos, and António Silva. 2019. From a Monolith to a Microservices Architecture: An Approach Based on Transactional Contexts. In *Proceedings of the 13th European Conference on Software Architecture (ECSA)*. Springer-Verlag, Paris, France, 37–52. https://doi.org/10.1007/978-3-030-29983-5_3

[8] Christos Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (October 1979), 631–653. https://doi.org/10.1145/322154.322158

[9] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2019. CLOTHO: Directed Test Generation for Weakly Consistent Database Systems. *Proceedings of the ACM on Programming Languages* 3, OOPSLA, Article 117 (October 2019), 28 pages. https://doi.org/10.1145/3360543

[10] Chris Richardson. 2018. *Microservices Patterns: With examples in Java*.

[11] Madalena Santos. 2022. *Microservice Decomposition for Transactional Causal Consistent Platforms*. Master's thesis. Instituto Superior Técnico, Universidade de Lisboa.

[12] Nuno Santos and António Silva. 2020. A Complexity Metric for Microservices Architecture Migration. In *Proceedings of the 17th IEEE International Conference on Software Architecture (ICSA)*. Salvador, Brazil. https://doi.org/10.1109/ICSA47634.2020.00024

[13] Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. 2020. COBRA: Making Transactional Key-Value Stores Verifiably Serializable. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Virtual Event.

[14] Johannes Thones. 2015. Microservices. *IEEE Software* 32, 1 (January 2015), 116. https://doi.org/10.1109/MS.2015.11