Don't go MAD with Anomalies! Design-time Microservice Anomaly Detection in Migration to Microservices

Valentim Romão[0009−0007−8236−1617], Rafael Soares[0000−0002−0806−8849], Luís Rodrigues[0000−0002−0313−6590], and Vasco Manquinho[0000−0002−4205−2189]

INESC-ID, Instituto Superior Técnico - Universidade de Lisboa, Portugal
{valentim.romao, joao.rafael.pinto.soares, ler, vasco.manquinho}@tecnico.ulisboa.pt

Abstract. The advent of microservices has led multiple companies to migrate their monolithic systems to this new architecture. When decomposing a monolith, a functionality previously implemented as a transaction may need to be implemented as a set of independent sub-transactions, possibly executed by multiple microservices, paving the way for anomalies to emerge. The ability to assess, at design time, the anomalies that different decompositions may generate is key to guide the programmers in finding the most appropriate decomposition that matches their goals. This paper introduces *MAD*, the first framework for automatically detecting anomalies that are introduced by a given decomposition of a monolith into microservices. *MAD* operates by encoding the executions of the original functionalities as an SMT formula and then using a solver to find satisfiable assignments that capture the anomalous interleavings made possible by that specific decomposition. We have applied *MAD* to different benchmarks and show that it can identify precisely the causes of potential anomalous behavior.

Keywords: Anomaly Detection · Microservices · Satisfiability Modulo Theories.

1 Introduction

Microservices have emerged as a promising architecture for implementing large-scale applications. When using microservices, applications are designed as a set of loosely coupled components that may be easily developed and maintained by independent teams [31,21]. The adoption of this architectural style has led many companies, including large companies such as Amazon, Netflix, and Uber, to migrate applications that have been previously implemented as monoliths to microservices [19,5,27,20,10,37,15].

Unfortunately, migrating an application to the microservice architecture is not a trivial task [25,16]. Functionalities that have been designed in the monolith to execute as a single ACID transaction may be required to execute as a sequence of independent transactions after the migration, each implemented by

a different microservice. This breaks the isolation among functionalities that are executed concurrently, leading to anomalous application behavior. Handling anomalous behavior is costly as it often requires the implementation of additional code, such as compensating actions [17], to correct the undesired effects of the loss of isolation and atomicity. Dealing with anomalies typically dominates the migration cost [32] and, in some cases, these costs outweigh the advantages of microservices, forcing developers to revert the application to a monolith [34].

The importance of estimating the complexity associated with a monolith decomposition has been recognized in the literature [4], motivating several works on how to aggregate the domain entities when migrating from a monolith to microservices [29,9,24,22]. However, none captures concrete anomalous behaviors that may result from concurrent executions of functionalities. They also fail to give hints regarding the type of anomalies that can occur, a key information to estimate the cost of compensating actions [26,36].

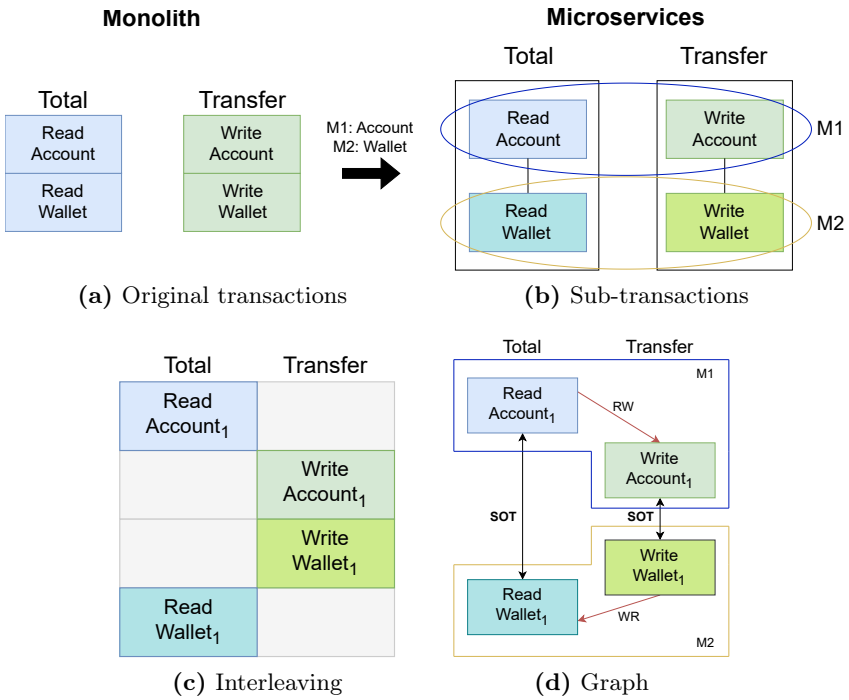
Testing tools, such as *MonkeyDB* [8] and *Cobra* [35], can detect anomalies by using a black box approach. However, there is no guarantee that all possible interleavings are tested and some anomalies may pass unnoticed. Moreover, they require a target decomposition to be implemented before being tested. In opposition, we aim to detect problematic decompositions at design time, avoiding implementing decompositions that generate many anomalies.

Early validation work for transactional systems, such as [13,23], assume a single database and a set of transactions that can be executed in any order. More recent works such as *ANODE* [28] and *CLOTHO* [30] consider distributed storage but assume that all storage nodes replicate all entities, which is not the case in microservice decompositions.

This paper proposes the *Microservices Anomaly Detector* (*MAD*), a new framework to identify anomalies that result from the decomposition of a monolith into microservices considering bounded executions of the system. *MAD* takes a monolithic application, a target decomposition, and the application's SQL schema to automatically generate the set of independent transactions that execute the same functionality in the microservices. Then, *MAD* encodes the interleavings of these transactions that correspond to non-serializable executions of the original functionalities as a *Satisfiability Module Theories* (SMT) formula. The SMT formula's solution provides the interleavings that may generate anomalous behaviors. Hence, *MAD* is the first system that precisely detects the anomalies that will result from the decomposition of monoliths into microservices.

Because *MAD* exhaustively searches the space of all transaction interleavings, the time it takes to analyze a decomposition may be large. To circumvent this limitation, *MAD* implements a novel divide-and-conquer approach to parallelize the search. This makes *MAD* suitable to be applied to non-trivial code bases.

In summary, the contributions of this paper are as follows: (1) we formulate the problem of finding anomalies in a microservice decomposition of a monolith as an SMT problem, (2) we propose a strategy to parallelize the task of finding the satisfiable assignments that capture anomalies, (3) we provide the developer



(a) Original transactions **(b)** Sub-transactions

(c) Interleaving **(d)** Graph

Fig. 1. Example of how two functionalities (a) can be divided when migrating from monolith to microservices (b) and of an interleaving (c) that leads to an anomaly (d).

with the classification of the type of anomaly and the entities and functionalities involved and, (4) we present an experimental evaluation of the resulting system with 7 different benchmarks. Also, we are making the source code of MAD publicly available (upon acceptance for anonymity purposes).

2 Motivation Example

Figure 1 illustrates an example of how a given microservices decomposition originates a previously non-existing interleaving. In this scenario, there are two entities (`Account` and `Wallet`), two transactions (Total and Transfer), and a decomposition where `Account` is managed by microservice M_1 and `Wallet` is managed by microservice M_2. Total gets the total amount of a client's money (its account balance plus its wallet balance). Transfer withdraws an amount of funds from the client's account balance and deposits it in their wallet. Considering that a client's balance can only be transferred from their account to their wallet, it is expected that the total amount of funds of a client always remains the same.

Since entities `Account` and `Wallet` are in different microservices, Total and Transfer would be split into sub-transactions, each of which executes in a different

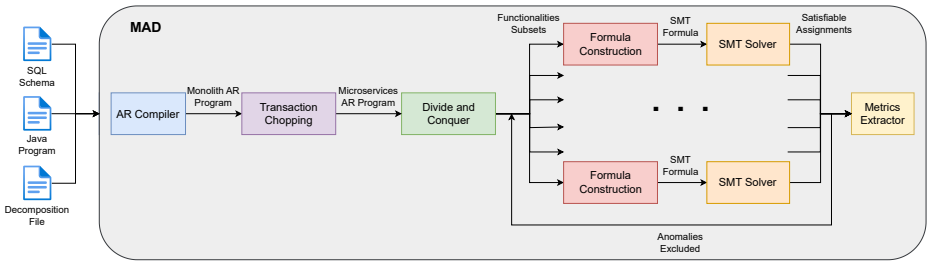


Fig. 2. ***MAD***'s pipeline.

```
CREATE TABLE Account (
  clientId INT,
  balance INT,
  PRIMARY KEY (clientId)
);

CREATE TABLE Wallet (
  clientId INT,
  balance INT,
  PRIMARY KEY (clientId)
);
```

Listing 1.1. Example of *MAD*'s input SQL schema file.

microservice. This can lead to anomalies that were not possible in the monolithic version, as illustrated in Figure 1(c). In this example, the execution of Transfer interleaves with the execution of Total. Total sees an older version of a client's account ($\texttt{Account}_1$), implying that Total is executed before Transfer. However, Total sees the new version of the client's wallet ($\texttt{Wallet}_1$), whose balance was already updated by Transfer, implying that Total is executed after Transfer. There is no serial order of the two functionalities that may lead to this execution. From this execution, one could incorrectly observe that the total funds of the client have changed, something impossible in this scenario.

3 Microservices Anomaly Detector

This section describes the *Microservices Anomaly Detector* (*MAD*), a framework to automatically detect anomalies that would result from implementing a given microservices decomposition of a monolithic application.

3.1 Overview

MAD is modular and its execution performs multiple steps in sequence. Figure 2 presents *MAD*'s pipeline. Next, we provide a description of each step.

```
public class exampleScenario {
  private Connection connect = null;
  private int _ISOLATION = Connection.TRANSACTION_READ_COMMITTED;
  private int id;
  Properties p;

  public exampleScenario(int id) {
    this.id = id;
    p = new Properties();
    p.setProperty("id", String.valueOf(this.id));
    Object o;
    try {
      o = Class.forName("MyDriver").newInstance();
      DriverManager.registerDriver((Driver) o);
      Driver driver = DriverManager.getDriver("jdbc:mydriver://");
      connect = driver.connect("", p);
    } catch (InstantiationException | IllegalAccessException |
        ClassNotFoundException | SQLException e) {
      e.printStackTrace();
    }
  }

  public void Total(int clientId) throws SQLException {
    PreparedStatement stmt1 = connect.prepareStatement("SELECT␣balance␣FROM␣Account"+"␣WHERE␣clientId␣=␣?");
    stmt1.setInt(1, clientId);
    ResultSet rs = stmt1.executeQuery();
    rs.next();
    int account_balance = rs.getInt("balance");

    PreparedStatement stmt2 = connect.prepareStatement("SELECT␣balance␣FROM␣Wallet"+"␣WHERE␣clientId␣=␣?");
    stmt2.setInt(1, clientId);
    ResultSet rs2 = stmt2.executeQuery();
    rs2.next();
    int wallet_balance = rs2.getInt("balance");

    int total_money = account_balance + wallet_balance;
  }

  public void Transfer(int clientId, int accountBalance, int walletBalance, int amount) throws SQLException {
    PreparedStatement stmt1 = connect.prepareStatement("UPDATE␣Account␣SET␣balance␣=␣?"+"␣WHERE␣clientId␣=␣?");
    stmt1.setInt(1, accountBalance - amount);
    stmt1.setInt(2, clientId);
    stmt1.executeUpdate();

    PreparedStatement stmt2 = connect.prepareStatement("UPDATE␣Wallet␣SET␣balance␣=␣?"+"␣WHERE␣clientId␣=␣?");
    stmt2.setInt(1, walletBalance + amount);
    stmt2.setInt(2, clientId);
    stmt2.executeUpdate();
  }
}
```

Listing 1.2. Example of *MAD*'s input Java file.

Input: MAD takes as input a monolithic implementation of an application (including its SQL schema and source code) and a high-level description of how the monolith is decomposed into multiple microservices. Currently, the source code must be a *Java program* written using the JDBC syntax (i.e., that uses SQL queries to access the entities, which are maintained by the application in a database). As an illustration, the input files for the scenario from Figure 1 are depicted in Listings 1.1, 1.2 and 1.3. Although the current prototype only supports Java, the framework has been designed such that it can be extended to support additional programming languages. The decomposition of the monolith is expressed as the clustering of the domain entities into aggregates [29] (entities grouped in the same cluster are assumed to be managed by the same microservice) and is represented by a JSON file (*Decomposition File*). The JSON file used in the example scenario is depicted in Listing 1.3. Using this input, *MAD*

```
{
  "M1": ["Account"],
  "M2": ["Wallet"]
}
```

Listing 1.3. JSON file with a microservices decomposition.

executes the pipeline presented in Figure 2, which is composed of the following sequence of steps:

Step 1 (AR Compiler): The source code is compiled to an *abstract representation* (AR) that captures how the functionalities access the domain entities. In the AR, each functionality is represented as a sequence of read and write operations on domain entities (*Monolith AR Program*). The AR compiler is the only component that needs to be changed to support other programming languages.

Step 2 (Transaction Chopping): From the AR of the functionalities, an AR of the microservice decomposition is automatically generated (*Microservices AR Program*) by chopping the functionalities code into multiple sub-sequences. Each sub-sequence accesses domain entities from a single aggregate, capturing a sub-transaction to be executed at a single microservice. In the microservice decomposition, sub-sequences that are part of the same parent functionality are executed as a sequence of independent transactions.

Step 3 (Divide and Conquer): Often, the AR program is too complex to be represented in a single encoding that can be effectively analyzed. Employing a divide and conquer strategy, *MAD* generates subsets of the functionalities (*Functionalities Subsets*). Each subset represents a possible combination of concurrent functionality executions. These can be analyzed independently and in parallel, allowing the whole problem to terminate in a reasonable time.

Step 4 (Formula Construction): Based on the Microservices AR, for each subset of functionalities, *MAD* generates an SMT formula encoding the possible functionalities' interleavings. The satisfiable assignments of these formulas are the possible interleavings that can lead to anomalies in the decomposition.

Step 5 (SMT Solver): MAD uses Z3 [11] to solve the SMT formulas. Satisfiable assignments correspond to cyclic graphs, with the vertices being operations and the edges the relations between operations, representing unserializable executions of functionalities [2]. These cycles have their length bounded by a system parameter denoted the *Maximum Cycle Length* (MCL), defined as the maximum number of edges to be considered when looking for satisfiable assignments.

Step 6 (Metrics Extractor): After all satisfiable assignments are found, *MAD* processes the anomalies found and extracts metrics to report to the user. These metrics include the total number of anomalies, dividing them in *core anomalies* (cyclic graphs with the minimum cycle length required to represent an anomaly) and their *extensions* (cyclic graphs that are supersets of the *core*

anomalies graphs), the anomalies types, and the entities, functionalities, and sub-transactions involved.

3.2 Supported Syntax

In the current version of *MAD*, the source code needs to be a program written in Java using the JDBC syntax. If needed, it is possible to extend the *AR Compiler* to support additional programming languages. Furthermore, the current version is not able to parse joins or implicit updates. This is not a fundamental limitation, because these operations can be represented through combinations of reads and writes, which are supported by the *AR Compiler* and are enough to cover all benchmarks used in the evaluation. To define a join, one can divide the query with the join operation into two or more queries, each accessing only one table. Similarly, to represent an implicit update, one can divide it into a read, to retrieve the current value, followed by a write with the update expression. These transformations could be implemented by leveraging the current *AR Compiler* or by adding a pre-processing step to automatically decompose the operations. However, this exercise falls outside the scope of this work.

3.3 Abstract Representation

MAD builds a *Monolith AR Program* from the Java source code. The AR facilitates the extraction of information, such as transactions, types of parameters, and execution order. *MAD* receives as input the Java source code of the monolith and compiles it to an abstract representation (AR), originating the *Monolith AR Program*. The AR facilitates the extraction of information, including the transactions, types of parameters, and execution order. Figure 3 presents the structure of the monolith AR.

To represent the SQL schema, *MAD* uses two elements, *Table* and *Column*. *Table* is represented by a *name* and a list of *Columns*. *Column* is represented by a *name*, and a *type* (int, real, string, Boolean).

The application implementation is represented by a set of *Original Transactions*. Each *Original Transaction* captures the code of a functionality in the monolith and has a *name*, a list of *Statements*, a list of *Expressions*, and a list of *Parameters*. Each *Statement* is represented by a *name*, an SQL *query* (select, update, insert or delete), and a *path condition*, which is an expression that associates the execution of the statement with a condition in the program if it occurs inside a conditional block. The *Expressions* can be of four types: *Unary Operation*; *Binary Operation*; *Value*; and *Variable*. The *Unary* and *Binary Operations* have one *operation* and the *Expression(s)* to which the *operation* is applied, respectively. The *Value* expressions represent the static values of the Java program using a *type* and a *value*. At last, the *variable* expressions represent the variables used in the Java program using the variable *name*. These variables include the ones used for the instructions, to store the read rows, and to hold the values of columns read from the database. To conclude, the *Parameters* are a specific type of *Value* expression, which are represented additionally by their *name*.

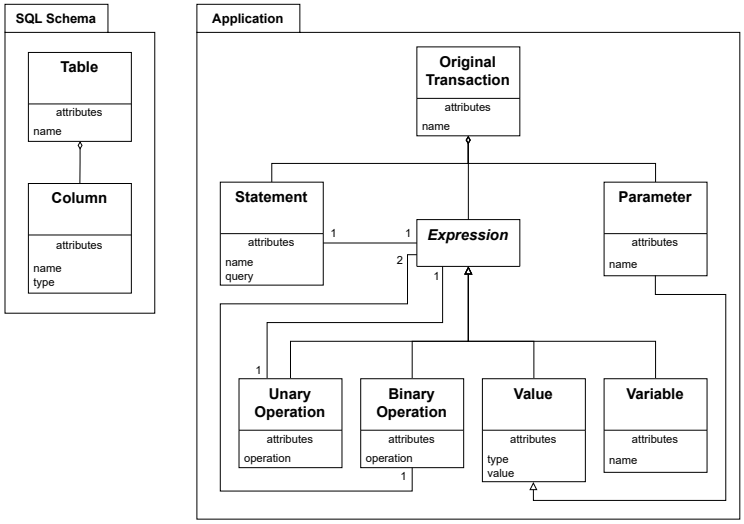


Fig. 3. Monolith AR structure.

3.4 Microservices Decomposition AR

From the Monolith AR Program and the JSON Decomposition File, *MAD* proceeds to create the *Microservices AR Program*. This step consists of applying a transaction chopping algorithm to the original functionalities of the monolith to transform each of them in a sequence of sub-transactions. The *sub-transactions* are represented in the AR by the following attributes: *name*; list of SQL *operations*; their *original transaction name*; and their *microservice name*.

The transaction chopping algorithm works as follows. For each original transaction, *MAD* iterates over the original sequence of operations and splits them into sub-transactions, according to their accessed entities. As an example, consider the scenario from Figure 1. By applying the chopping algorithm, from the original transactions depicted in Figure 1(a), we obtain a representation of the microservices version depicted in Figure 1(b). Each sub-transaction only accesses one microservice and is executed as an independent transaction.

3.5 Divide and Conquer

The *Divide and Conquer* strategy generates all the combinations of functionalities (original transactions) of size smaller than the system parameter *Maximum Cycle Length* (MCL). Note that, since an anomalous interleaving requires that at least two operations belong to the same instance of a functionality, at most $MCL - 1$ functionalities can be involved in a cycle without exceeding the MCL.

Let n be the number of functionalities. The number of generated combinations is given by $\sum_{x=1}^{MCL-1} {}_nC_x$. *MAD* starts by creating one thread for each combination of size 1 (i.e., involving a single functionality). Each thread explores

the possible interleavings between the operations of each functionality. When all these threads finish their analysis, the process is repeated for size 2 combinations, already excluding the anomalies involving only size 1 combinations. This process continues until all combinations of size $MCL - 1$ are analyzed.

Our approach has resemblances to the well-known Cube-and-Conquer algorithm [18], except that *MAD* splits the search space explicitly by taking advantage of domain knowledge. Hence, our approach is closer to a classic divide-and-conquer algorithm. Note that each thread analyzes a different combination of functionalities. Moreover, constraints are added such that solutions with $k - 1$ functionalities are excluded when solving a formula involving k functionalities.

3.6 SMT Encoding

MAD encodes the Microservices AR program into an SMT formula such that any satisfiable assignment corresponds to an anomaly. We first provide a high-level view of the SMT formulas and the information required to be represented in it, and later present the detailed representation.

Observe that *MAD* needs to consider all possible executions of functionalities, transactions and interleavings where each execution is represented as a graph of operations. Hence, *MAD* implicitly searches over all possible graphs that can be built using the available system components. *MAD* does not explicitly build these graphs, but rather leverages SMT to encode the search space in order to check if it is possible to build a graph corresponding to an anomaly. Therefore, each satisfiable assignment to the SMT formula will correspond to a particular execution with an anomaly. Otherwise, if the SMT formula is unsatisfiable, then no anomaly can be produced for all possible executions of the system's functionalities.

We base our anomaly detection on Adya et al. definition of anomaly [2]. An execution is represented as a graph, where vertices correspond to operations belonging to transactions and edges represent data dependencies between operations. An anomaly is present if the graph is cyclic and contains at least two dependency edges and at least two operations belonging to the same transaction, also represented as an edge. As such, we must encode the transactions' operations and the data dependencies that different operations may present.

Two operations might belong to different transactions, executing in different microservices, but belong to the same functionality. Operations in the same functionality must also execute in isolation to provide equivalent execution to the monolith, and interleavings may generate anomalies. Hence, we encode information regarding the functionality and microservice associated with each operation.

First, we represent the basic elements of the system (operations, sub-transactions, original transactions, and microservices). Second, the associations between the basic elements (e.g., to which sub-transaction does a given operation belongs). Third, the consistency guarantees offered by the environment where the microservices execute. Fourth, the possible types of relations between operations. Lastly, the format of the cycles *MAD* wants the SMT solver to find.

Table 1. Key functions used in formulas.

Function	Description
otime(O)	returns the instant of time when operation O executes
oname(O)	returns the operation name ($ONname$) of operation O
tname(T)	returns the sub-transaction name ($TName$) of sub-transaction T
fname(F)	returns the original transaction name ($FName$) of original transaction F
mname(O)	returns the microservice's name ($MName$) where operation O is executed
parent(O)	returns the instance of sub-transaction (T) where operation O belongs
origtx(O)	returns the instance of original transaction (F) where operation O belongs

Table 2. Key predicates used in formulas.

Predicate	Description
is_update(O)	true if operation O is an update operation (update, insert, delete)
ST(O,O)	true if both operation's instances (O,O) belong to the same instance of sub-transaction (ST)
SOT(O,O)	true if both operation's instances (O,O) belong to the same instance of original transaction (SOT)
WR(O,O)	true if there is a write followed by a read dependency (WR) between operation's instances (O,O)
RW(O,O)	true if there is a read followed by a write dependency (RW) between operation's instances (O,O)
WW(O,O)	true if there is a write followed by a write dependency (WW) between operation's instances (O,O)
vis(O,O)	true if the visibility effects of the first instance are visible to the second instance of operation's instances (O,O)
ar(O,O)	true if the first instance is executed before the second instance of the operation's instances (O,O)
D(O,O)	true if there is any dependency relation between operation's instances (O,O) (WR, RW, or WW)
X(O,O)	true if there is any relation between operation's instances (O,O) (ST, SOT, WR, RW, or WW)

Representation of the Basic Elements To represent instances of the operations, sub-transactions, and original transactions, *MAD* uses three sorts, O, T, and F, respectively. Based on the AR, *MAD* defines a unique name for each operation, sub-transaction, original transaction, and microservice. These unique identifiers are declared using the following sorts: *ONames* for operations; *TNames* for sub-transactions; *FNames* for original transactions; and *MNames* for microservices. Associated with the basic elements, there are functions and predicates used in the formulas, listed in Table 1 and Table 2.

Formula Components Let $FNames = \{Txn_1, \ldots, Txn_f\}$ denote the name set of f original transactions and let $TNames = \{Txn_{1,1}, \ldots, Txn_{f,t}\}$ denote the name set of sub-transactions, where $Txn_{i,j}$ refers to the j^{th} sub-transaction in original transaction Txn_i. Moreover, let $ONames = \{Op_1, \ldots, Op_k\}$ and $MNames = \{M_1, \ldots, M_m\}$ denote the name set of k update operations and m microservices, respectively. For any two operations Op_i and Op_j from the same transaction where Op_i occurs before Op_j, then we have $j > i$. Finally, let $F(Op_i), T(Op_i)$ and $M(Op_i)$ denote the names of the original transaction, sub-transaction and microservice of execution for operation Op_i. Note that all these name sets and name associations are defined through a simple analysis of the program. Afterwards, these are used to encode the relations between the system's components as follows (see Figure 4):

System Component's Constraints:

$$\forall o_1, o_2 \in O : WR(o_1, o_2) \Rightarrow vis(o_1, o_2) \quad (1)$$
$$\forall o_1, o_2 \in O : WW(o_1, o_2) \Rightarrow ar(o_1, o_2) \quad (2)$$
$$\forall o_1, o_2 \in O : RW(o_1, o_2) \Rightarrow \neg vis(o_2, o_1) \quad (3)$$
$$\forall o_1 \in O : is\_update(o_1) \Leftrightarrow (oname(o_1) = Op_1 \vee \ldots \vee oname(o_1) = Op_k) \quad (4)$$
$$\forall o_1 \in O,\ Op_j \in ONames : (oname(o_1) = Op_j) \Rightarrow ((tname(parent(o_1)) = T(Op_j)) \quad (5)$$
$$\forall o_1 \in O,\ Op_j \in ONames : (oname(o_1) = Op_j) \Rightarrow ((fname(origtx(o_1)) = F(Op_j)) \quad (6)$$
$$\forall o_1 \in O,\ Op_j \in ONames : (oname(o_1) = Op_j) \Rightarrow (mname(o_1) = M(Op_j)) \quad (7)$$
$$\forall o_1, o_2 \in O : ar(o_1, o_2) \Rightarrow (otime(o_1) < otime(o_2)) \quad (8)$$
$$\forall o_1, o_2 \in O,\ Op_i, Op_j \in ONames,\ j > i,\ F(Op_i) = F(Op_j) : ((parent(o_1) = parent(o_2)) \vee (origtx(o_1) = origtx(o_2)) \wedge (oname(o_1) = Op_i) \wedge (oname(o_2) = Op_j)) \Rightarrow (otime(o_1) < otime(o_2)) \quad (9)$$
$$\forall o_1, o_2 \in O : D(o_1, o_2) \Rightarrow (\neg(ST(o_1, o_2) \vee SOT(o_1, o_2)) \wedge (WW(o_1, o_2) \vee WR(o_1, o_2) \vee RW(o_1, o_2))) \quad (10)$$
$$\forall o_1, o_2 \in O : X(o_1, o_2) \Rightarrow (ST(o_1, o_2) \vee SOT(o_1, o_2) \vee D(o_1, o_2)) \quad (11)$$

Consistency Models Constraints:

$$\forall o_1, o_2, o_3 \in O : (ST(o_1, o_2) \wedge vis(o_1, o_3) \wedge (mname(o_1) = mname(o_3))) \Rightarrow vis(o_2, o_3) \quad (12)$$
$$\forall o_1, o_2, o_3 \in O : (ST(o_1, o_2) \wedge vis(o_3, o_1) \wedge (mname(o_1) = mname(o_3))) \Rightarrow vis(o_3, o_2) \quad (13)$$
$$\forall o_1, o_2 \in O : (ar(o_1, o_2) \wedge (mname(o_1) = mname(o_2))) \Rightarrow vis(o_1, o_2) \quad (14)$$

Edge Type Constraints:

$$\forall o_1, o_2 \in O : (parent(o_1) = parent(o_2)) \Leftrightarrow ST(o_1, o_2) \quad (15)$$
$$\forall o_1, o_2 \in O : ((origtx(o_1) = origtx(o_2)) \wedge (parent(o_1) \neq parent(o_2))) \Leftrightarrow SOT(o_1, o_2) \quad (16)$$

Cycle Length Constraints:

$$\exists o_1, o_2, \ldots o_k \in O : Distinct(o_1, o_2, \ldots, o_k) \wedge (ST(o_1, o_2) \vee SOT(o_1, o_2)) \wedge D(o_2, o_3) \wedge \ldots \wedge X(o_i, o_{i+1}) \wedge \ldots \wedge D(o_k, o_1) \quad (17)$$

Fig. 4. ***MAD***'s models constraints

- **C1:** Every two instances of operations related by an *WR* edge have the effects of the first instance visible to the second instance (Equation 1).
- **C2:** Every two instances of operations related by an *WW* edge have the first instance happening before the second instance (Equation 2).
- **C3:** Every two instances of operations related by an *RW* edge have the effects of the second instance not visible to the first instance (Equation 3).
- **C4:** Every instance of an update operation must have an *OName* from the set of update operations names, and vice-versa (Equation 4).
- **C5-C7:** Every instance operation with a given *OName* needs to belong to an instance of sub-transaction (specific *TName*), original transaction (specific *FName*) and microservice (specific *MName*) as specified in Equations 5, 6 and 7.
- **C8:** Every two instances of operations with an *ar* relation between them follow an execution order where the first instance occurs before the second (Equation 8).
- **C9:** Every two instances of operations that belong to the same original transaction instance need to follow a sequential execution order according to their order in the Java program (Equation 9).

- **C10:** Every two instances of operations that have a dependency relation between them (D) do not have an ST or SOT relation and do have an WW, WR or RW relation (Equation 10).
- **C11:** Every two instances of operations that have any relation between them (X) do have an ST, SOT, WW, WR or RW relation (Equation 11).

Consistency Models To make an analysis faithful to the environment where the microservice systems will execute, operations of the same sub-transaction respect Serializability. Also, all sub-transactions that execute at a given microservice respect Serializability (even if they belong to different functionalities). Serializability is encoded into the SMT formula by the combination of three constraints: *Read Committed*, *Repeatable Read*, and *Linearizability* [6,1], which are presented in Figure 4.

- **C12 (Read Committed):** for every three instances of operations (o_1, o_2, o_3), if o_1 and o_2 belong to the same instance of sub-transaction, the effects of o_1 are visible to o_3, and o_1 and o_3 belong to the same microservice, then the effects of o_2 are also visible to o_3 (Equation 12).
- **C13: (Repeatable Read):** for every three instances of operations (o_1, o_2, o_3), if o_1 and o_2 belong to the same instance of sub-transaction, the effects of o_3 are visible to o_1, and o_1 and o_3 belong to the same microservice, then the effects of o_3 are also visible to o_2 (Equation 13).
- **C14 (Linearizability):** for every two instances of operations (o_1, o_2), if o_1 happens before o_2, and o_1 and o_2 belong to the same microservice, then the effects of o_1 are visible to o_2 (Equation 14).

Types of Edges To encode the ST and SOT edges, MAD expresses the following two properties, respectively:

- **C15:** Every two instances of operations that belong to the same instance of sub-transaction are related via an ST edge, and vice-versa (Equation 15).
- **C16:** Every two instances of operations that belong to the same instance of original transaction and do not belong to the same instance of sub-transaction are related via an SOT edge, and vice-versa (Equation 16).

For the dependency edges (RW, WR, WW), constraints are defined for each pair of sub-transactions to establish whether or not it is possible to have a dependency between instances of operations of those sub-transactions. If two operations o_1 and o_2 access different tables, no dependency can ever occur. Otherwise, if either o_1 or o_2 executes an update operation, then a constraint is added to verify if it is possible to have a row where the operations conflict.

Cycles Assertions A cyclic anomalous graph is a cycle with at least one ST or SOT edge and at least two dependency edges (RW, WR, WW). This is captured in Equation 17 where k denotes the size of the cycle.

Recalling the example anomaly presented in Figure 1(c). *MAD* detects that anomaly by finding the cyclic graph that can be seen in Figure 1(d). The cycle contains two SOT edges and two dependency edges (RW and WR), and represents the interleaving of the original transactions Total and Transfer.

3.7 Metrics Extractor

MAD includes a Metrics Extractor component, which gathers information regarding the possible anomalies that the microservices application may face when the given decomposition is used. The information collected includes the total number of anomalies, the number of anomalies per type and, finally, the entities, functionalities, and sub-transactions involved in each anomaly. These indicators allow the programmer to have a better understanding of the potential anomalies and the amount of effort required to prevent each of them. Furthermore, these metrics capture the types of anomalous behaviors to be expected and the combinations that originate these anomalies.

All metrics are generated based on the analysis of the satisfiable assignments of the SMT formula. Each satisfiable assignment corresponds to an anomaly, and its type is classified as defined in Adya et al [2]. Moreover, *MAD* also groups the anomalies by sets of entities, functionalities, and sub-transactions. Furthermore, *MAD* categorizes the anomalies as either *core anomalies* or *extensions*. A *core anomaly* is one whose execution cycle has the minimum size required to express its categorized anomaly. *Extensions* are anomalies whose execution cycle includes a *core anomaly* with additional operations that do not affect the categorized anomaly. This distinction is important since if a developer fixes the source of the *core anomalies*, it will also remove its related *extensions*.

4 Evaluation

MAD [1] aims at identifying the anomalies that can emerge when migrating a monolith to microservices following a given decomposition. Note that *MAD* is the first tool that automatically identifies anomalies when migrating an application from a monolithic to a microservices architecture. Hence, the experimental evaluation is mainly targeted at analyzing *MAD*'s performance, since we are unable to perform a direct comparison with other tools. Our experimental evaluation focuses on the following key research questions:

- **RQ1:** Can *MAD* find differences between decompositions?
- **RQ2:** How long does it take to execute *MAD*?

[1] *MAD*'s code, benchmarks, and results presented in this paper are publicly available at https://doi.org/10.5281/zenodo.18302601.

- **RQ3:** Can *MAD* classify the anomalies to help identify access patterns that cause the errors?
- **RQ4:** How effective is the *Divide and Conquer* strategy?

4.1 Experimental Setup

We use seven applications found on GitHub as benchmarks for the experimental evaluation. Our process to set the benchmarks consists of three steps: 1) gathering monolithic applications from GitHub; 2) adapting them to the syntax processed by *MAD* ; 3) generating two microservices decompositions of each application with the help of a migration tool [29] that supports programmers on the task of grouping the entities by the microservices.

The applications we have used in the evaluation are the following:

- **TPC-C**[2] is defined in the OLTP-Bench [12] project and simulates the behavior of a delivery and warehouse management system;
- **FindSportMates**[3] (findmates) is an application used as a benchmark in other microservices works [38,33] and simulates a platform where users can manage and find events to connect with other users;
- **jpabook**[4] simulates a shop where members can order items and track the delivery process;
- **JPetStore**[5] (jpetstore) is an application highly used by previous microservices works [38,14,9,22,39] and simulates an online pet store where each user has an account and can browse through a catalog of pets to choose which pets they want to order;
- **spring-petclinic**[6] (petclinic) is an application used as a benchmark by a previous microservices work [14] and simulates the operation of a pet clinic;
- **myweb**[7] is an application that simulates the behavior of the web allowing users to have roles and manage resources;
- **spring-mvc-react**[8] (react) 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 influences the users' popularity.

We have selected these benchmarks because they cover a wide range of domain areas and have implementations that address real-world scenarios and, yet, they are simple enough to be processed by the *MAD* prototype in a reasonable time. For each benchmark, we analyze three decompositions. First, the *mono*

[2] https://github.com/oltpbenchmark/oltpbench/tree/master/src/com/oltpbenchmark/benchmarks/tpcc
[3] https://github.com/chihweil5/FindSportMates
[4] https://github.com/holyeye/jpabook/tree/master/ch12-springdata-shop
[5] https://github.com/mybatis/jpetstore-6
[6] https://github.com/spring-projects/spring-petclinic
[7] https://github.com/Jdoing/myweb
[8] https://github.com/noveogroup-amorgunov/spring-mvc-react

Table 3. Application Profiles.

			microservices			transactions		
Benchmark	**#E**	**#F**	**mono**	**"best"**	**full**	**mono**	**"best"**	**full**
TPC-C	9	5	1	9	9	5	6	22
findmates	2	9	1	2	2	9	13	13
jpabook	5	10	1	2	5	10	15	21
jpetstore	13	11	1	3	13	11	17	48
petclinic	6	12	1	2	6	12	14	27
myweb	5	20	1	2	5	20	22	32
react	5	23	1	2	5	23	28	39

(E=entities; F=functionalities)

decomposition, which represents the monolithic version of the benchmark. For this decomposition, the number of sub-transactions equals the number of functionalities. Second, the *"best"* decomposition, which is the one with the highest *Silhouette Score* (a metric used to assess how well the clustering of the entities is done) calculated by the migration tool [29]. At last, in the *full* decomposition, each entity is managed by a different microservice. Table 3 provides an overview of application complexity, such as the number of entities and functionalities, and the resulting decomposition sizes.

In this context, we use *MAD* considering four as the maximum cycle length. The process to choose this value consisted of starting with value three since it is the minimum number of edges required to detect a cycle with an anomaly, and incrementing it to ensure that all core anomaly types could be detected (requires length four [7,30]), while still assuring that *MAD* was able to analyze the benchmarks within a timeout limit of 4 hours (14400 seconds).

The evaluation was performed in a virtual machine with 32 virtual CPU cores running on two Intel(R) Xeon(R) Gold 5320 CPUs at 2.2GHz and 128GB of DDR4 RAM with Intel Optane Memory configured in App Mode. The virtual machine uses Ubuntu 18.04.4 LTS, Java 8, and version 4.12.3 of Z3 with the default configuration.

4.2 RQ1: Can *MAD* find differences between decompositions?

Table 4 presents the results obtained when applying *MAD* to our benchmarks. Results show that *MAD* allows the 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*, *jpetstore*, and *react*, even their *"best"* decompositions have anomalies. This occurs because the *"best"* decompositions for these cases require the functionalities to be chopped into multiple sub-transactions, opening the door for more interleaving between the functionalities. Note also that all microservices decompositions, except *jpetstore best* and *myweb full*, have a fairly higher number of anomalies when compared with the number of *core anomalies*. Thus, a fair portion of the anomalies found

Table 4. Anomalies detected.

	MAD								
	core anomalies			total anomalies			execution time (s)		
Benchmark	**mono**	**"best"**	**full**	**mono**	**"best"**	**full**	**mono**	**"best"**	**full**
TPC-C	0	0	28	0	0	98	10	9	306
findmates	0	0	0	0	0	0	23	31	31
jpabook	0	22	25	0	70	79	48	136	222
jpetstore	0	40	85	0	52	188	454	7,239	10,249
petclinic	0	0	0	0	0	0	93	89	128
myweb	0	0	7	0	0	7	309	358	514
react	0	13	18	0	45	58	545	996	1,517

(E=entities; F=functionalities)

(between 54 and 72 percent) are simply *extensions* of smaller number of core anomalies and, therefore, can be eliminated if the core anomalies are eliminated. The *findmates* and *petclinic* benchmarks have no anomalies in any of their decompositions because their operations are mostly reads and the functionalities tend to be short. This, respectively, leads to fewer conflicts between accesses and fewer interleavings between functionalities.

4.3 RQ2: What is *MAD*'s execution time?

The last column of Table 4 shows the time required to execute MAD on each decomposition. As can be observed, although *MAD* can analyze these cases, *MAD* often requires non-negligible time to perform the analysis. *MAD* does not require long executions for simple applications, nor when the decompositions do not use many sub-transactions. However, for complex applications with a high number of functionalities and/or sub-transactions, *MAD* needs to analyze a large number of combinations of transactions, therefore taking more time to finish its execution. Nevertheless, since this is done at design-time, programmers can use *MAD* to try several decompositions to microservices before implementation.

4.4 RQ3: Can *MAD* Classify Anomalies?

Table 5 shows the number of anomalies found for each decomposition by type of anomaly. We are only considering the analysis of the sub-transactions metrics for one decomposition, but this can be done for all decompositions. Note that we omit the results for the *findmates* and *petclinic* benchmarks since they have no anomalies in any decomposition. Also, the *Write Skews* are counted together with the *Lost Updates* because the write skew pattern also corresponds to one of the lost update patterns. The only way to distinguish between them would be to also consider the rows accessed. If the same row was being accessed, then it would be a *Lost Update*. Otherwise, it would be a *Write Skew*. However, the current patterns used in our approach only consider the graph cycle edges and

Table 5. *MAD* anomalies found per type.

Benchmark	#DR	#DW	#LU/WS	#LU/	#NRR	#PR	#RS	#Ext	#Total
	"best" / full	"best" / full	"best" / full	"best" / full	"best" / full	"best" / full	"best" / full	"best" / full	"best" / full
TPC-C	/	/ 13	/	/ 3	/	/	/ 12	/ 70	/ 98
jpabook	/	/	3 / 3	10 / 11	/	/	9 / 11	48 / 54	70 / 79
jpetstore	/	4 / 7	/	12 / 14	/ 2	/	24 / 62	12 / 103	52 / 188
myweb	/	/	/	/ 3	/	/	/ 4	/	/ 7
react	/	/	/	/	1 / 2	1 / 4	11 / 11	32 / 40	45 / 58

(DR=dirty read, DW=dirty write; LU=lost update; WS=write skew,
NRR=non-repeatable read; PR=phantom read; RS=read skew, Ext=extensions)

Table 6. *TPC-C full core anomalies* entities.

Entities	#Anomalies	Anomalies Types
[oorder, order_line]	5	[DW, RS]
[customer, district]	4	[DW]
[customer, warehouse]	4	[DW]
[new_order, order_line]	4	[LU/WS, RS]
[new_order, oorder]	3	[LU/WS, RS]
[customer, new_order]	2	[LU/WS, RS]
[customer, oorder]	1	[DW]
[customer, order_line]	1	[DW]
[district, order_line]	1	[RS]
[district, stock]	1	[DW]
[district, warehouse]	1	[DW]
[order_line, stock]	1	[RS]

(DW=dirty write; LU=lost update; WS=write skew, RS=read skew

operation types, and do not include information regarding the accessed rows which are required to make this distinction.

Finally, *MAD* also analyses the satisfiable assignments provided by the SMT solver and provides the programmer access to which entities and functionalities are associated with the anomalies found. For example, Table 6 presents the *core anomalies* found in the *TPC-C full* decomposition. Developers may leverage this information to guide their decomposition design, identifying the most costly entity decouplings. For instance, notice that the combination of entities *[oorder, order_line]* are heavily coupled, as decomposing these entities into different microservices generates five new anomalies. Merging these entities into the same microservice could significantly reduce anomaly mitigation costs when migrating the application. Furthermore, Table 7 presents the combinations of sub-transactions that originate the *core anomalies* in the decomposition. This information highlights the key sections in the application that require more care when migrating the application, allowing developers to be better informed during the decomposition process on what anomalies they will face and what kind of techniques will be required to mitigate the effects of said anomalies.

4.5 RQ4: How effective is the Divide and Conquer strategy?

The results in Table 8 show that the divide and conquer strategy can significantly shorten the analysis time. Gains are more significant for complex cases since each SMT formula is much smaller, and it mitigates the time and space complexities by not considering all the original transactions simultaneously. As a result, all

Table 7. *TPC-C full core anomalies* sub-transactions.

Functionalities	Sub-transactions	#Anomalies	Anomalies Types
[payment]	[payment_0, payment_2]	4	[DW]
[payment]	[payment_1, payment_2]	4	[DW]
[delivery]	[delivery_0, delivery_1]	2	[LU/WS, RS]
[delivery]	[delivery_0, delivery_2]	2	[LU/WS, RS]
[delivery]	[delivery_0, delivery_3]	2	[LU/WS, RS]
[orderStatus, newOrder]	[orderStatus_1, orderStatus_2, newOrder_3, newOrder_7]	2	[RS]
[delivery, newOrder]	[delivery_0, delivery_2, newOrder_4, newOrder_7]	2	[RS]
[delivery, newOrder]	[delivery_1, delivery_2, newOrder_3, newOrder_7]	2	[RS]
[payment]	[payment_0, payment_1]	1	[DW]
[newOrder]	[newOrder_2, newOrder_6]	1	[DW]
[delivery]	[delivery_1, delivery_2]	1	[DW]
[delivery]	[delivery_1, delivery_3]	1	[DW]
[delivery]	[delivery_2, delivery_3]	1	[DW]
[newOrder, stockLevel]	[stockLevel_0, stockLevel_1, newOrder_2, newOrder_7]	1	[RS]
[newOrder, stockLevel]	[stockLevel_1, stockLevel_2, newOrder_6, newOrder_7]	1	[RS]
[delivery, newOrder]	[delivery_0, delivery_1, newOrder_3, newOrder_4]	1	[RS]

(DW=dirty write; LU=lost update; WS=write skew, RS=read skew

Table 8. Divide and conquer performance (s).

Benchmark	Without			Sequential			Parallel		
	mono	"best"	full	mono	"best"	full	mono	"best"	full
TPC-C	7	7	1,715	18	20	339	10	9	306
findmates	4	5	5	20	24	24	23	31	31
jpabook	7	4,263	5,342	37	220	286	48	136	222
jpetstore	38	(timeout)	(timeout)	850	7,670	10,435	454	7,239	10,249
petclinic	10	11	13	67	68	92	93	89	128
myweb	11	11	213	218	227	353	309	358	514
react	153	(timeout)	(timeout)	406	982	1,209	545	996	1,517

analysis can be performed within the time limit (4 hours = 14,400 seconds). Without this strategy, *MAD* exceeds the timeout when analyzing decompositions *"best"* and *full* of the *jpetstore* and *react* benchmarks. However, for simple cases, *MAD*'s performance tends to be worse with the divide and conquer strategy. This occurs because the strategy originates an overhead to *MAD*'s analysis by requiring unnecessary iterations over combinations with no anomalies. We also note that the strategy is not fully parallelizable, since the threads of bigger size combinations need to wait for all the threads of smaller size combinations to finish in order to start. Therefore, the analysis time for a given combination size is bounded by the analysis time of the slowest thread that is analyzing a combination of that size.

5 Conclusions and Future Work

This paper has presented *MAD*, the first framework that automatically detects anomalies that result from the migration of a monolith application to a microservices architecture. *MAD* avoids the under/overestimation limitations of previous works and can classify the anomalies according to the access patterns that cause them, thus helping the programmer by identifying scenarios that

result in anomalies. Experimental results from applying *MAD* to different decompositions of benchmarks inspired by applications on GitHub show that *MAD* can offer insights on the complexity of a decomposition that are more precise than the metrics extracted by related work. We plan to extend the framework in several ways. For instance, we will add support for associations between entities in different microservices, such as JPA relationships, foreign keys, and semantic invariants that the system must respect.

Acknowledgments. This work was supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: doi.org/10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: doi.org/10.54499/-UID/PRR/50021/2025), PTDC/CCI-COM/2156/2021 (DOI: doi.org/10.54499/PTDC/-CCI-COM/2156/2021), 2023.14280.PEX (DOI: doi.org/10.54499/2023.14280.PEX), and LISBOA2030-FEDER-00771200 (DOI: doi.org/10.54499/2023.18452.ICDT).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Jepsen consistency models. https://jepsen.io/consistency. Accessed: 24/12/2022.
2. Atul Adya, Barbara Liskov, and Patrick O'Neil. Generalized isolation level definitions. In *ICDE*, San Diego (CA), USA, February 2000.
3. Deepthi Akkoorath, Alejandro Tomsic, Manuel Bravo, Zhongmiao Li, Tyler Crain, Annette Bieniusa, Nuno Preguiça, and Marc Shapiro. Cure: Strong semantics meets high availability and low latency. In *ICDCS*, pages 405–414, Nara, Japan, June 2016.
4. João Almeida and António Silva. Monolith migration complexity tuning through the application of microservices patterns. In *ECSA*, L'Aquila, Italy, September 2020.
5. Nuha Alshuqayran, Nour Ali, and Roger Evans. A systematic mapping study in microservice architecture. In *SOCA*, Macau, China, 2016.
6. Peter Bailis, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph Hellerstein, and Ion Stoica. Highly available transactions: Virtues and limitations. In *VLDB*, volume 7, page 181–192, Trento, Italy, November 2013.
7. Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, page 1–10, San Jose (CA), USA, May 1995.
8. Ranadeep Biswas, Diptanshu Kakwani, Jyothi Vedurada, Constantin Enea, and Akash Lal. MonkeyDB: Effectively testing correctness under weak isolation levels. In *OOPSLA*, Chicago (IL), USA, October 2021.
9. Miguel Brito, Jácome Cunha, and João Saraiva. Identification of microservices from monolithic applications through topic modelling. In *SAC*, page 1409–1418, Virtual Event, Republic of Korea, March 2021.
10. Phil Calçado. Building products at SoundCloud — Part I: Dealing with the monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith. Accessed: 14/12/2023.
11. Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, TACAS'08, Budapest, Hungary, April 2008.

12. Djellel Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. OLTP-Bench: An extensible testbed for benchmarking relational databases. In *VLDB*, page 277–288, Trento, Italy, December 2013.
13. Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, jun 2005.
14. Gianluca Filippone, Nadeem Qaisar Mehmood, Marco Autili, Fabrizio Rossi, and Massimo Tivoli. From monolithic to microservice architecture: an automated approach based on graph clustering and combinatorial optimization. In *ICSA*, pages 47–57, L'Aquila, Italy, 2023.
15. Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. Microservices migration in industry: Intentions, strategies, and challenges. In *ICSME*, pages 481–490, Cleveland (OH), USA, 2019.
16. Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. *IEEE Computer Architecture Letters*, PP:1–1, 05 2018.
17. Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD*, page 249–259, San Francisco (CA), USA, December 1987.
18. Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *HVC*, Haifa, Israel, 2011. Springer.
19. Jeremy Hillpot. 4 microservices examples: Amazon, Netflix, Uber, and Etsy. https://blog.dreamfactory.com/microservices-examples/. Accessed: 12/12/2023.
20. Steven Ihde and Karan Parikh. From a monolith to microservices + REST: the evolution of LinkedIn's service architecture. https://www.infoq.com/presentations/linkedin-microservices-urn/. Accessed: 14/12/2023.
21. Pooyan Jamshidi, Claus Pahl, Nabor Mendonça, James Lewis, and Stefan Tilkov. Microservices: The journey so far and challenges ahead. *IEEE Software*, 35:24–35, 05 2018.
22. Wuxia Jin, Ting Liu, Qinghua Zheng, Di Cui, and Yuanfang Cai. Functionality-oriented microservice extraction based on execution trace clustering. In *ICWS*, pages 211–218, San Francisco (CA), USA, 2018.
23. Sudhir Jorwekar, Alan Fekete, Krithi Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, page 1263–1274, Vienna, Austria, 2007. VLDB Endowment.
24. Anup Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. In *ESEC/FSE*, page 1214–1224, Athens, Greece, August 2021.
25. Miika Kalske, Niko Mäkitalo, and Tommi Mikkonen. Challenges when moving from monolith to microservice architecture. In *ICWE*, pages 32–47, Rome, Italy, 02 2017.
26. Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. Consistency rationing in the cloud: Pay only when it matters. *Proc. VLDB Endow.*, 2(1):253–264, August 2009.
27. Tony Mauro. Adopting microservices at Netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/. Accessed: 14/12/2023.
28. Kartik Nagar and Suresh Jagannathan. Automated detection of serializability violations under weak consistency. In *CONCUR*, Beijing, China, September 2018.

29. Luís Nunes, Nuno Santos, and António Silva. From a monolith to a microservices architecture: An approach based on transactional contexts. In *ECSA*, page 37–52, Paris, France, September 2019. Springer-Verlag.
30. Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. Clotho: Directed test generation for weakly consistent database systems. In *OOPSLA*, Athens, Greece, October 2019.
31. Chris Richardson. *Microservices Patterns: With examples in Java*. November 2018.
32. Nuno Santos and António Silva. A complexity metric for microservices architecture migration. In *ICSA*, Salvador, Brazil, March 2020.
33. Anfel Selmadji, Abdelhak-Djamel Seriai, Hinde Lilia Bouziane, Rahina Oumarou Mahamane, Pascal Zaragoza, and Christophe Dony. From monolithic architecture style to microservice one based on a semi-automatic approach. In *ICSA*, pages 157–168, Salvador, Brazil, 2020.
34. Ruoyu Su, Xiaozhou Li, and Davide Taibi. Back to the future: From microservice to monolith. In *Microservices*, 2023.
35. Cheng Tan, Changgeng Zhao, Shuai Mu, and Michael Walfish. Cobra: Making transactional key-value stores verifiably serializable. In *OSDI*, Virtual Event, November 2020.
36. Douglas Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP '13*, November 2013.
37. Johannes Thones. Microservices. *IEEE Software*, 32(1):116, January 2015.
38. Pascal Zaragoza, Abdelhak-Djamel Seriai, Abderrahmane Seriai, Anas Shatnawi, and Mustapha Derras. Leveraging the layered architecture for microservice recovery. In *ICSA*, pages 135–145, Honolulu (HI), USA, 2022.
39. Yukun Zhang, Bo Liu, Liyun Dai, Kang Chen, and Xuelian Cao. Automated microservice identification in legacy systems with functional and non-functional metrics. In *ICSA*, pages 135–145, Salvador, Brazil, 2020.