MIST: Microservices Identification from Sequential Traces

Thakshila Imiya Mohottige
The University of Melbourne
Parkville, VIC, 3010
timiyamohott@student.unimelb.edu.au

Artem Polyvyanyy
The University of Melbourne
Parkville, VIC, 3010
artem.polyvyanyy@unimelb.edu.au

Colin Fidge
Queensland University of Technology
2 George St, Brisbane City QLD 4000
c.fidge@qut.edu.au

Rajkumar Buyya
The University of Melbourne
Parkville, VIC, 3010
rbuyya@unimelb.edu.au

Alistair Barros
Queensland University of Technology
2 George St, Brisbane City QLD 4000
alistair.barros@qut.edu.au

***Abstract*—Microservice systems comprise small, loosely coupled, and highly cohesive services that interact to provide system functionality, offering advantages in scalability and flexibility through the use of multiple processing nodes. However, migrating a monolithic software system to a microservice architecture is challenging due to the complexities of decoupling tightly integrated code, managing data used by multiple services, and ensuring efficient inter-service communication. This paper presents *MIST*, an approach for identifying and extracting microservices from monolithic systems using sequential pattern mining. *MIST* analyzes execution traces of a monolithic system to detect repetitive, coherent, low-level method invocation sequences and extracts them as independent, executable microservices. These microservices can be deployed independently while still maintaining communication with the rest of the system to ensure the original system's operational continuity. Furthermore, we investigate the impact of different attributes and sequential patterns on microservices discovery and performance. Based on the case studies conducted using open-source software systems and our publicly available implementation, *MIST* maintains high structural quality. Although extracting microservices based on method execution frequency can negatively affect system performance, the average call stack depth of a method sequence has a positive impact. Attributes such as the consecutiveness of method calls, pattern length, and method execution duration can offer improved performance.**

***Index Terms*—Microservices, Service-oriented architecture, software architecture, process mining, sequential pattern mining.**

I. Introduction

Software system architectures have evolved from monolithic mainframe-based systems via service-oriented systems and, more recently, to microservices architectures. Monolithic systems, implemented as a single component, often face scalability, maintainability, and availability issues [1]. A change in one class of a monolithic system often leads to changes in other classes due to the tight coupling of the system [2].

Service-oriented architecture (SOA) improves the monolithic architecture by developing the system as multiple, logically related, distributed components. Microservice architectures advance SOAs by introducing smaller, technology-agnostic, loosely coupled, and highly cohesive services that can be developed, tested, and deployed independently [1]. The adoption of microservices is further accelerated by cloud-based technologies, which inherently support scalability and high availability. A monolithic system can be migrated to a microservice architecture either by completely redeveloping it or by incrementally extracting microservices from the existing system [2]. Given the significant investments in legacy systems and the high cost of complete redevelopment, incremental extraction is often the cost-effective and pragmatic approach.

Much effort has been devoted to developing methods for extracting microservices from monolithic systems, using static source code analysis, dynamic analysis of run-time logs and system behavior, and model- or domain-driven approaches. Among these, static source code analysis is the most widely used technique in microservice migration [3]. Such tools facilitate the analysis of system structure, component interactions, and circular dependencies. However, they fall short in capturing dynamic behavior and runtime dependencies, such as execution duration and frequency [2]. However, understanding the dynamic aspects of a system is often not feasible solely through source code analysis, particularly in the presence of polymorphism and dynamic binding [4]. The behavior of the system is not explicitly reflected in its source code, and syntactic relationships at the code level do not necessarily correspond to the same semantic functionality [5]. Instead, execution traces, captured as sequences of source code method invocations, reveal program behavior [6].

The performance impact of migration has received limited attention in the existing studies on microservice migration [7]. The primary focus of microservice identification studies is structural qualities like coupling, cohesion, and modularity. However, the migration effort becomes ineffective if well-defined microservices degrade system performance, increase latency, and decrease system throughput. Therefore, it is essential to analyse how the factors used in microservices identification influence the performance of the resulting migrated systems.

Sequential pattern mining plays a vital role in uncovering correlations and associations within large datasets [8]. It has been successfully applied in domains such as financial analysis, biomedical research, DNA sequencing, disaster management, and web usage analysis. In pattern mining, detecting recurring behavioral or operational patterns in sequences of events helps to understand how processes typically unfold over time [9]. This ability is beneficial in Internet-of-Things (IoT) environments, where systems often handle large volumes of event data at the "edge" under tight resource constraints [10]. In such settings, recognizing method call sequences can reveal opportunities to reorganize software into lightweight, reusable microservices that align with actual usage patterns. Identifying fine-grained microservices can improve performance and responsiveness, which are key concerns for edge and IoT systems. Individual methods are the smallest reusable units in source code. Hence, detecting microservices at the method level is well-suited to resource-constrained IoT environments.

In this paper, we present *MIST*[1] (Microservices Identification from Sequential Traces), a framework grounded in dynamic software analysis and sequential pattern mining. MIST performs sequential pattern mining over execution traces of a monolithic system to identify coherent, fine-grained, frequently invoked, and computationally intensive patterns of method calls. These patterns define candidate microservices aimed at optimizing system performance (e.g., in resource-constrained IoT environments). To select which microservice candidates to implement in the reengineered system, *MIST* considers five criteria: support, confidence, pattern length, average call stack depth, and average execution time of the constituent methods.

We applied the *MIST* framework to implement and analyse two case studies on JForum [2] and Apache Roller [3]. The impact of the framework was assessed in terms of throughput and latency. This measure is based on the latest literature review study on microservices [3]. JForum and Apache Roller are the most commonly used benchmarks in dynamic analysis studies, appearing in 22% of such works. Importantly, none of these prior studies on dynamic analysis assessed system performance after implementing the identified candidate microservices, and only 18% of the broader literature on microservices migration reports the runtime performance analytics we used in our experiments [3]. The majority of existing approaches rely on static analysis or require additional artifacts (e.g., source code annotations or architectural models), making our study the first to rely solely on semi-automated dynamic software analysis to identify and implement microservices and to evaluate the impact on system performance. We conducted performance testing using simulated user behavior to generate execution traces. The validity of the dynamic analysis depends on the number of use cases covered in the execution log. As in previous studies [5, 6], system operations were performed using a mix of manually collected and simulated test cases designed to ensure reproducibility. Our test suites cover *all* major system functionalities, incorporating different system users. Microservice identification studies generally rely on structural property validation. Hence, we conducted a validation of structural properties against existing benchmark approaches [5, 6, 11–14] using JForum and Apache Roller.

Specifically, this paper makes these contributions:

- It proposes *MIST*, a framework for identifying candidate microservices using semi-automated dynamic software analysis and sequential pattern mining;
- It presents an empirical study of how five pattern attributes, both individually and together, relate to the runtime performance of the extracted microservices.
- It evaluates the structural quality of the obtained microservices against benchmark approaches on two widely used systems for benchmarking microservices reengineering.

Next, this paper introduces the underlying technical concepts (Section II), presents our new microservice identification framework (Section III), reports on its case studies (Section IV), reviews related work on microservice migration (Section V), highlights the limitations and directions for future research (Section VI), and summarizes our observations (Section VII).

II. Preliminaries

This section introduces the concept of sequential pattern mining and the fundamental concepts required to understand the subsequent discussions.

Sequential pattern mining uncovers meaningful subsequences within datasets [15]. It was initially developed to extract sequential patterns from transactional databases [16], where the relevant patterns do not need to be contiguous. These algorithms are mainly classified as Apriori-based, vertical format–based, and pattern growth algorithms [8]. Apriori uses the principle that all subsets of a frequent itemset must also be frequent, generating candidates iteratively, though at high computational cost. Pattern growth avoids this by using structures like FP-trees, which offer more efficient mining and improved overall performance.

Event e in a software system log is a tuple,

$$e = (i, x, t, a, d, c, \delta), \text{ where}$$

- i is the unique identifier of the event,
- x denotes the execution instance of the system that triggered the event,
- t is the timestamp indicating when the event occurred,
- a is the system method (i.e., activity) whose invocation triggered the event,
- d is the duration of the execution of the method that triggered the event,
- c is the position of the event within the sequence of events generated by the same execution instance x, and
- δ is the call depth of the method invocation that triggered the event within the execution stack.

[1] https://anonymous.4open.science/r/MISt
[2] https://jforum.net/
[3] https://roller.apache.org/

A *trace* τ of a system is a sequence of all events triggered by all method invocations during its single execution, ordered by their time of occurrence from earliest to latest. Hence, all events in a trace are associated with the same execution instance, which is also referred to as the *trace identifier*. An *event log* E_L of a software system is a collection of its traces.

A *pattern* is a set of events in an event log E_L, where their order is not considered. A pattern can appear in multiple traces in the event log, and the event log contains multiple patterns. The *pattern length* is the number of events in the pattern.

The *support* of a pattern X, denoted $\text{Sup}(X)$, is defined as the number or proportion of traces in E_L that contain the pattern X.

The *confidence* of a pattern Y relative to pattern X is the conditional probability that a trace containing X also contains Y. It is given by

$$\text{confidence} = \frac{\text{Sup}(X \cup Y)}{\text{Sup}(X)}.$$

Let E_L contain a frequent pattern with k-events that occur n times. For each occurrence, let δ_k represent the call depths, which refers to the level of nesting for each method/event within the call stack during program execution. The *average depth* is defined as,

$$\text{average depth} = \frac{1}{n}\sum_{i=1}^{n} \min(\delta_k)$$

where it averages the minimum depth of each pattern in the number of occurrences. The *average execution* time is defined as,

$$\text{average execution time} = \frac{1}{n}\sum_{j=1}^{n}\left(\sum_{i=1}^{k} d_i\right)$$

where d_i is the duration of the i-th event in the pattern consisting with k events, and j indexes the n occurrences of that pattern. In other words, for each occurrence j, we sum the duration of all k events in the pattern, and then take the average of these total durations across all n occurrences in the event log.

III. Microservices Discovery

This section describes the design and implementation of the *MIST* framework, which aims to identify microservices from monolithic systems using sequential pattern mining. Figure 1 provides an overview of the *MIST* framework, consisting of an underlying software system integration process and a seven-step pipeline that transforms software execution logs into a collection of frequent sequential patterns.

Our *MIST* framework relies on dynamic software analysis, so availability of a log file that captures possible execution paths is essential. Enterprise software systems typically generate raw log files containing developer-inserted logs, which often lack the necessary information for comprehensive pattern mining. To enable detailed analysis, additional runtime data, such as method signatures, parameter values, method entry and exit timestamps, call stack depth, and session identifiers, must be collected. However, manually updating log statements or instrumenting communication interfaces to incorporate this information is labor-intensive and error-prone, especially in large codebases. A commonly used solution is aspect-oriented programming, particularly the joinpoint-pointcut model, which supports collection of execution data without modifying the original code [4]. This allows unobtrusive monitoring by integrating a third-party dependency into the system to capture detailed execution information. Accordingly, the *MIST* framework begins by instrumenting the target software system using the Kieker monitoring tool[4], which has been widely adopted in dynamic software analysis [2, 5, 6].

The effectiveness of dynamic analysis depends on capturing a comprehensive range of user behaviors. To improve reproducibility, particularly during evaluation, it is important to replay the same user behavior consistently. Therefore, automated user simulation becomes essential. To simulate user behavior, we use JMeter[5], an open-source load-testing tool, to execute predefined test cases that cover the behavior of different users and system functionalities. As shown in Figure 1, the execution of these test cases on the instrumented system produces a Kieker log file, which serves as the input for subsequent processing in the pattern mining and microservice identification workflow. Listing 1 presents an excerpt of a Kieker log file capturing information on three method invocations, one per log line. Using the semicolon symbol as a delimiter, each line captures Kieker log type, unique entry ID, invoked method, execution trace ID, start timestamp of method execution, end timestamp of method execution, computation node ID, order of the method invocation in the system execution, and depth of the method invocation in the call stack. This detailed, method-level execution data is essential for the operation of the *MIST* framework. The generated Kieker log file is then fed into the *MIST* tool, which implements the framework. The tool executes a seven-step pipeline to identify microservices, as depicted in Figure 1.

Listing 1. Excerpt of a Kieker log.

```
$1;1733116555379401700;public static java.lang.
    ↪ String net.jforum.util.preferences.
    ↪ SystemGlobals.getValue(java.lang.String)
    ↪ ;2140194985419472908;1733116555379026700;
    ↪ 1733116555379396200;4180L-212561-W;57;3
$1;1733116555379053100;public java.lang.String net.
    ↪ jforum.util.preferences.SystemGlobals.
    ↪ getVariableValue(java.lang.String)
    ↪ ;2140194985419472908;1733116555379029900;
    ↪ 1733116555379052400;4180L-212561-W;58;4
$1;1733116555379039200;public java.lang.String net.
    ↪ jforum.util.preferences.VariableExpander.
    ↪ expandVariables(java.lang.String)
    ↪ ;2140194985419472908;1733116555379034100;
    ↪ 1733116555379036300;4180L-212561-W;59;5
```

The *MIST* tool starts by converting the integrated Kieker logs to events. Each method invocation in the Kieker log is converted to an event. An event log file is generated as an

[4]https://kieker-monitoring.net/

[5]https://jmeter.apache.org/

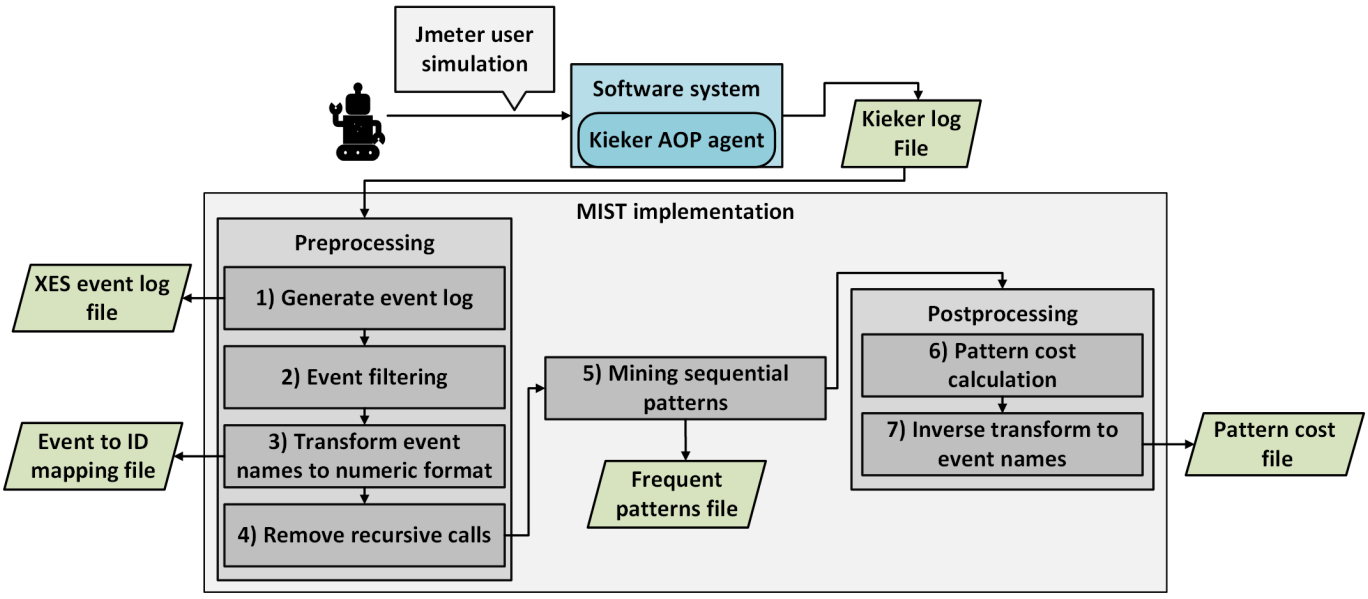

Fig. 1. The MIST Framework.

intermediate output, in the XES[6] format, the de facto standard for event data in process mining. These XES logs typically serve as primary inputs for a wide range of process mining applications. This supports interoperability, allowing seamless integration with established tools such as ProM[7] and Disco[8], both of which natively support the XES format. Although the generated events are used in subsequent steps, the XES file itself is not used in this study; it is produced solely to support future research on applying process mining techniques to microservice discovery. Listing 2 illustrates an example event in XES format, generated by *MIST* from the first line of the Kieker log shown in Listing 1.

Listing 2. Example event (XES format).

```
<trace>
<string key='traceID' value='
    ↪ 2140194985419472908'/>
...
<event>
<string key='concept:name' value='net.
    ↪ jforum.util.preferences.
    ↪ SystemGlobals.getValue(java.lang.
    ↪ String)'/>
<string key='lifecycle:transition' value='
    ↪ complete'/>
<date key='time:timestamp' value='
    ↪ 2024-12-02T17:15:55.26700'/>
<string key='eventId' value='
    ↪ 1733116555379401700'/>
<int key='duration' value='369500.0'/>
<int key='callingOrder' value='57'/>
<int key='depth' value='3'/>
</event>
...
</trace>
```

Event filtering is then performed to exclude specific classes and methods from the pattern mining process. Software systems often include frequently executed methods—such as initialization routines, routing mechanisms, or communication handlers—that are not meaningful for microservice identification. Classes containing such methods and/or specific methods can be excluded from the analysis at this stage using a configurable parameter in the *MIST* tool.

The events are then encoded into a compact numerical representation to boost processing efficiency. In both the Kieker and XES event logs, method calls are stored as strings. For large systems and extensive test suites, this can result in a great variety of distinct entries, and repeated string comparisons quickly become a performance bottleneck. To mitigate this overhead, each unique method signature is mapped to a distinct integer identifier. Subsequent algorithms can then operate on lightweight numeric arrays rather than bulky string data, reducing memory consumption and dramatically accelerating look-ups, joins, and aggregations.

Execution traces often contain repeated method calls due to recursive calls. Although recursion increases the trace length, each method counts only once per trace; thus, it does not affect the method call frequency. Instead, keeping recursive calls adds unnecessary complexity and reduces mining efficiency. Therefore, the fourth step removes recursive calls to enable more effective sequential pattern mining, which focuses on non-recursive behavior.

The next step involves applying sequential pattern mining over the traces in the event log. For this purpose, the open-source SPMF library[9] [17], which provides more than 250 data mining algorithms, is used. In particular, the *MIST* tool employs the PrefixSpan algorithm, a pattern growth-based approach known for its high efficiency. The *MIST* framework adopts a pluggable architecture, enabling easy integration

[6]https://www.xes-standard.org/
[7]https://promtools.org/
[8]https://fluxicon.com/disco/
[9]https://www.philippe-fournier-viger.com/spmf/

and extension with alternative mining algorithms as needed. Since the SPMF library requires a specific input format, this step includes preparing the required input, executing the mining process, and capturing the output, which contains the discovered frequent patterns along with their support values. The sample output after executing frequent pattern mining is illustrated in Listing 3. The numerical values, such as 6, 132, 40, etc., are unique numbers assigned to method calls to reduce processing complexity. The -1 value is the delimiter between pattern and support values.

Listing 3. Pattern mining output.

```
6 -1 #SUP: 500
6 132 -1 #SUP: 150
6 132 216 -1 #SUP: 145
6 40 -1 #SUP: 165
6 40 41 -1 #SUP: 165
6 41 -1 #SUP: 178
6 41 42 -1 #SUP: 178
6 42 -1 #SUP: 196
6 42 43 -1 #SUP: 173
6 42 59 -1 #SUP: 151
6 42 44 -1 #SUP: 173
6 42 79 -1 #SUP: 147
```

The sixth step computes the cost of each mined pattern, which informs the selection of candidate microservices for extraction. Five properties are used to assess pattern cost: support (S), confidence (C), pattern length (P), average depth (D), and average execution time (E), as defined in Section II. Support and confidence originate from sequential pattern mining [15, 16]. *Support* indicates the frequency of a pattern in the event log and is calculated by the pattern mining library. Patterns with higher support values are given greater priority. *Confidence* measures the strength of the association among method calls within a pattern. For sequential patterns, it represents the likelihood that the entire sequence occurs without interruption and is computed using the corresponding support values. For example, the confidence of pattern a, b is calculated as $Sup(a, b)/Sup(a)$. Because a, b is a frequent pattern, its subsequence a also appears in the mining output, as illustrated in the sample output of pattern mining listing 3. Higher confidence increases the resulting pattern cost, as it yields tightly coupled associations. The *pattern length* corresponds to the number of method calls in the sequence, with longer patterns receiving higher priority. *Average depth* captures the mean of the minimum call-stack depth across all occurrences of the pattern; deeper calls tend to represent finer-grained functionality and therefore contribute positively to the cost. Depth information is available in the generated events as shown in listing 2, which is derived from the Kieker log and calculated as described in Section II. *Average execution time* reflects the mean total execution time of a pattern across all its occurrences. The Kieker log contains in and out timestamps, which are used to calculate the duration of the event as illustrated in Listing 2. The duration of all the events in a pattern is aggregated across all occurrences and averaged as specified in Section II. Patterns with higher execution times are prioritized, as they are likely to represent performance-critical behaviour. Since the properties are measured on different scales, they are normalized to a range between zero and one prior to cost computation. Notably, confidence already lies within this range, as it is a conditional probability.

TABLE I
SAMPLE METHOD INVOCATION PATTERNS AND THEIR COSTS.

Pattern	Cost
public static java.lang.String net.jforum.util.pref.SysGlobals.getValue(java.lang.String); public java.lang.String net.jforum.util.pref.SysGlobals.getVariableVal(java.lang.String); public java.lang.String net.jforum.util.preferences.VariableExpander.expandVariables (java.lang.String);	1
public static boolean net.jforum.repository.ForumRepository.isCategoryAccessible(int); public static boolean net.jforum.repository.ForumRepository.isCategoryAccessible(int, int);	1
public java.lang.String net.jforum.util.pref.SystemGlobals.getVariableValue (java.lang.String); public void net.jforum.csrf.CsrfLogger.log (org.owasp.csrfguard.log.LogLevel, java.lang.String);	0.989
public static int net.jforum.util.pref.SysGlobals.getIntValue (java.lang.String); public static int net.jforum.repository.ForumRepository.getTotalMessages(); public static java.util.List net.jforum.repository.ForumRepository.getAllCategories(int);	0.988

Once these five properties are computed, the overall cost of each pattern is calculated using the formula defined in Equation (1). The formula aims to prioritize patterns that are both structurally and operationally relevant for microservice identification, favoring patterns that simultaneously score high values in all five properties.

$$\text{PatternCost} = \text{S} \times \text{C} \times \text{P} \times \text{D} \times \text{E} \tag{1}$$

The final step involves translating the numerical patterns back into their corresponding source code representations. Following the third step, ***MIST*** processes numerical patterns. This step restores the original string values, producing a table of patterns and their associated cost values as the final output. Each pattern is presented as a sequence of method invocations, sorted in descending order by cost. A sample output table is provided in Table I.

IV. CASE STUDIES

This section analyses the impact of the aforementioned attributes and sequential patterns on the performance and the structural qualities of microservices. Details of the applications used for the case studies are listed in Table II. The 'Test cases' column shows the number of functional tests covered during the execution trace generation. These functional test suites constitute an enhanced version of previously established scenarios used in execution-trace-based microservices identification studies [6], extending previous work by incorporating additional functional test cases. The 'Requests' column reports the number of simulated user requests used to generate the execution traces, while the 'Events' column shows the number

TABLE II
SUMMARY OF CASE STUDY APPLICATIONS.

Application	Description	Version	LoC	Classes	Test cases	Requests	Events
Roller	Multi user blog server	5.2.0	53,889	615	92	512	485,379
JForum	Online discussion forum	2.8.3	32,171	353	48	635	24,999

of events produced by the *MIST* framework from these simulated requests.

To assess the microservice performance based on the sequential patterns discovered using different attributes and their combinations, we defined cost functions over all non-empty subsets of five properties: support (S), confidence (C), pattern length (P), average depth (D), and average execution time (E). This yields $2^5 - 1 = 31$ unique combinations. For each subset of attributes, the *MIST* tool generates a corresponding list of frequent patterns, where the cost of each pattern is computed as the product of the normalized values of the selected attributes. For instance, the "S" list ranks patterns solely by normalized support values, while the "C" list uses confidence values. The "S,C" list, in turn, ranks patterns using the product of support and confidence. The final list, labeled "S,C,P,D,E", uses all five properties and computes the cost as defined in Equation (1). As shown in Table I, multiple patterns with respective cost values are provided as the output. The three highest-cost patterns with consecutive methods were selected as microservices for each testing scenario. For instance, methods A, B, C, and E generate two frequent sequences: A, B, C with a cost of 0.75, and A, C, E with a cost of 0.85. Despite A, C, E having the highest cost, the sequence A, B, C was chosen for the experiment due to its consecutive nature. Three microservices were selected for each category to reflect the structure of real-world applications, which often consist of multiple microservices.

The deployment setup consists of a maximum of four servers: one dedicated to the monolithic portion of the system and the remaining three serving the microservices. Testing was conducted in a cloud environment with four Ubuntu servers and one MySQL database instance. Each server contains 4 VCPUs, 16GB RAM, and 30GB of Hard drives. The database instance has 4GB RAM and 4GB of Hard drive. To simulate resource-constrained environments such as IoT devices, the CPU is stressed to 100% using the Ubuntu stress tool. JMeter was employed to simulate the system load during the experiments.

The sample implementations of the monolithic and microservice variants are presented in Listing 4 and Listing 5, respectively, and use RESTful web services. These listings correspond to the first identified pattern shown in table I. Listing 4 demonstrates how the monolithic system handles outgoing requests. Specifically, it intercepts the original execution flow, redirects the relevant functionality to the corresponding microservice via a REST call, and then incorporates the returned response to resume and complete the processing of the original request. In contrast, Listing 5 illustrates the

Listing 4. Send Request (Monolithic).

```
public class SystemGlobals{
    ....
    public static String getValue(String field) {
            MSSenderService msSenderService = new
     ↪ MSSenderService();
            String response = msSenderService.
     ↪ sendSystemGlobalMessage(field);
            return response;
        }
    ....
    }

public class MSSenderService {
  public String sendSystemGlobalMessage(String field) {
    String url=String.join(ip, port, '/service/api/
     ↪ receiveMessage');
    try {
      RestTemplate rest = new RestTemplate();
      Gson gson = new Gson();
      ServiceMessageEntity msg = new ServiceMessageEntity();
      msg.setMessageType(MSMessageTypes.SYSTEM_GLOBAL);
      msg.setField(field);
      URI uri = new URI(url);
      return rest.postForObject(uri, gson.toJson(msg),
     ↪ String.class);
    } catch (URISyntaxException e) {
      System.out.println('Error: ' + e.getMessage());
    }
    return '';
  }
}
```

implementation of the extracted microservice. It receives the request forwarded by the monolithic system, performs the designated processing logic, and returns the computed response to the monolith. This interaction exemplifies the delegation of specific functionality from the monolithic application to an independently deployable microservice while preserving the overall system workflow.

Two performance metrics were evaluated: throughput, the number of requests processed per second, and latency, the time interval between sending the request to the server and receiving the first response byte. Hence, higher throughput and lower latencies indicate better performance. To enhance the reliability of the results, the experiments were repeated three times, and the average value for each metric was calculated.

A. Case study one - Apache Roller

The first case study was conducted using the Apache Roller application. It is a multi-user blog server, with general and admin users. The Apache Roller test suite includes one admin user and one general user, each with 100 threads, for a total of 200 simultaneous threads. These values were chosen based on prior empirical evaluations of the Apache Roller System to ensure the system is overloaded under the given test configuration.

Listing 5. Microservice code.

```
@RestController
@RequestMapping("/api")
public class MSReceiverService {

  @RequestMapping('/receiveMessage')
  public String receiveMessage(@RequestBody String payload)
     ↪ {
    Gson gson = new Gson();
    ServiceMessageEntity serviceMessageEntity = gson.
     ↪ fromJson(postPayload, ServiceMessageEntity.class);
    switch (serviceMessageEntity.getMessageType()) {
        ....
          case SYSTEM_GLOBAL:
          String val = SystemGlobals.getValue(
     ↪ serviceMessageEntity.getField());
          return val;
         ....
    }
  }
}

public class SystemGlobals {
  public static String getValue(String field) {
    {
        return globals.getVariableValue(field);
    }
  public static String getVariableValueServiceCall(String
     ↪ field)
    {
        return globals.getVariableValue(field);
    }
  public String getVariableValue(String field)
    {
        String preExpansion = globals.installation.
     ↪ getProperty(field);
        if (preExpansion == null) {
            preExpansion = this.defaults.getProperty(field);
            if (preExpansion == null) {
              if (LOGGER.isEnabledFor(Level.INFO)) {
                LOGGER.info("Key '" + field + "' is not
     ↪ found in " + globals.defaultConfig + " and " +
     ↪ globals.installationConfig);
              }
              return null;
            }
        }
        return expander.expandVariables(preExpansion);
    }
}
```

Figure 2 illustrates the throughput measures in the 31 microservices and the monolithic Apache Roller application. In the results, the monolith system is represented by 'M', the attributes S, C, P, D, E, and their combinations represent the properties used in the cost formula for each test scenario. Individually, the confidence (C), pattern length (P), and average depth (D) properties yield higher throughput compared to the monolithic application. Furthermore, combinations of confidence and average depth (C, D) and confidence and pattern length (C, P) demonstrate improved throughput, with the highest throughput achieved by the combination of confidence and average depth (C, D). In contrast, support (S) and its combinations result in lower throughput, thus reducing the system performance. '

Figure 3 compares the latency measurements of the Apache roller application. As latency should be minimised, only the average depth (D), confidence (C), and pattern length (P) properties, when applied individually, result in lower latency values. Furthermore, the combination of confidence and pattern length (C, P) yields a lower latency than the monolithic

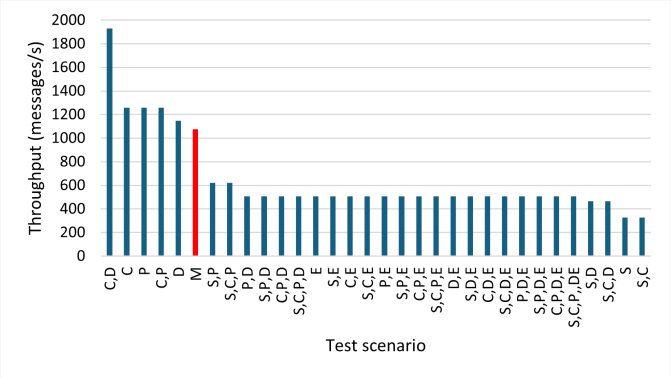

Fig. 2. Apache Roller throughput analysis.

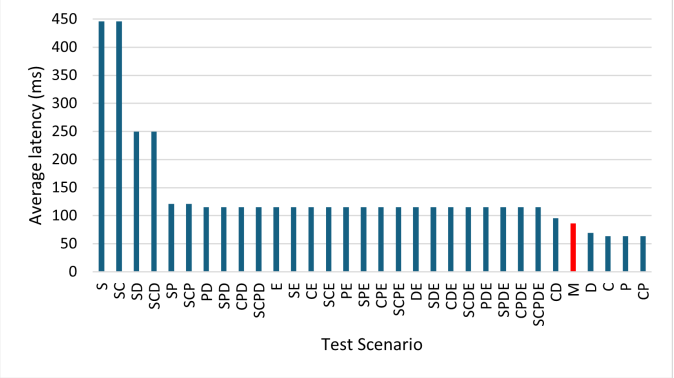

Fig. 3. Apache Roller latency analysis.

application. Moreover, latency measurements confirm that support (S) and its combinations increase system latency, thereby negatively impacting system performance.

This case study concludes that confidence (C), pattern length (P), average depth (D), and the combination of confidence and pattern length (C, P) yield performance-optimized microservices with higher throughput and lower latency compared to the monolithic Apache Roller system. Support (S) and its combinations negatively impact the performance of migrated microservices.

B. Use case two - JForum

The second use case was conducted using the JForum application. The initial JForum test suite comprised six user simulations: two admin users and four application users. By adjusting the thread count and loop properties in JMeter, the load was replicated across 25 threads per user during the evaluation, yielding a total of 150 concurrent threads. These values were selected based on a prior empirical evaluation of the test setup to ensure system overload.

The throughput of the monolithic JForum application, together with the throughput of the 31 corresponding microservice configurations, is presented in Figure 4. The results show that execution time (E) and average depth (D) properties alone yield higher throughput than the monolithic JForum application. Furthermore, combinations of confidence and execution

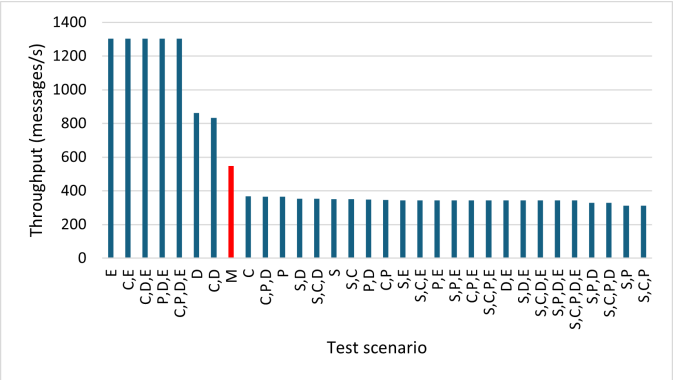

Fig. 4. JForum throughput analysis.

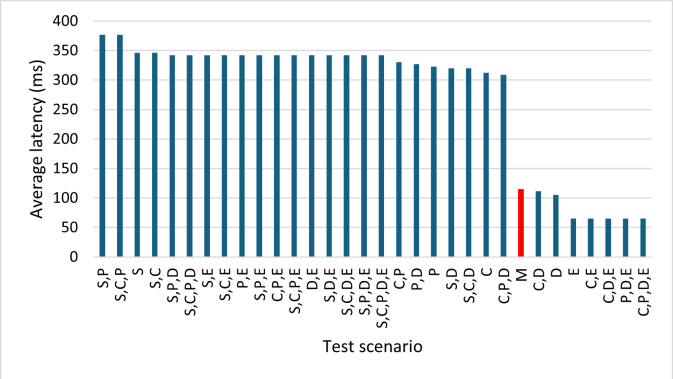

Fig. 5. JForum latency analysis.

time (C, E), and of confidence and average depth (C, D), yield higher throughput. Furthermore, (C, D, E), (P, D, E), and (C, P, D , E) combinations lead to higher throughput. Support (S) and its combinations negatively impact the throughput of the migrated system.

Figure 5 presents the latency of the monolithic JForum application relative to the 31 test scenarios. The results show that the average depth (D) and execution time (E) provide lower latencies. Moreover, the confidence and average depth (C, D), the confidence and execution time (C, E) properties yield lower latencies. Furthermore, higher-order combinations such as (C, D, E), (P, D, E), and (C, P, D, E) also lead to lower latency values. Support (S) and its combinations reduce the performance by increasing latency.

The results of these two case studies conclude that microservices identified using the higher average depth (D) attribute alone achieve higher throughput and lower latency than the monolithic system. In contrast, selecting microservices solely by the Support property consistently reduces throughput and increases latency, indicating degraded system performance. In addition, the results indicate that increasing the number of properties used for microservice identification has the potential to reduce system performance.

C. Structural quality

The majority of studies on microservice identification assess the structural properties, such as modularity, coupling, and cohesion, as quality metrics [3]. Hence, the structural quality was assessed in two case studies. We calculate structural properties using the metrics used in previous studies [5, 6, 11, 13, 18]. Specifically, the quality metrics we assessed are the following:

1) **Structural Modularity** (SM) measures the structural cohesiveness based on intra-connectivity and inter-connectivity of microservices [6, 11, 13, 18]. Higher SM values contribute to increased cohesion within microservices.

$$SM = \frac{1}{N}\sum_{i=1}\frac{u_i}{N_i^2} - \frac{1}{N(N-1)/2}\sum_{i\neq j}^{N}\frac{\sigma_{i,j}}{2(N_i \times N_j)}$$

where N is the number of services, u_i is the number of edges inside service i, $\sigma_{i,j}$ is the number of edges between service i and service j, and N_i and N_j are the numbers of entities inside services i and j.

2) **Interface Number** (IFN) measures the average number of interfaces exposed by microservices [6, 11, 13]. Lower values indicate better microservices due to the reduction of coupling among the microservices.

$$IFN = \frac{1}{N}\sum_{j=1}^{N}|I_j|$$

where I_j is the published interfaces of service j. N is the number of services in the system.

3) **Cohesion at message level** (CHM) defines the cohesiveness of the interfaces published by a service at message level [6, 13]. The higher the value of CHM, the more cohesive the services.

$$CHM = \frac{\sum_{j=1}^{K} n_i\, CHM_j}{\sum_{i=1}^{K} n_i}$$

Where,

$$CHM_j = \begin{cases} \dfrac{\sum_{(k,m)} f_M(Op_k, Op_m)}{|I_i| \times (|I_i|-1)/2}, & \text{if } |I_i| \neq 1, \\ 1, & \text{if } |I_i| = 1 \end{cases}$$

$$f_M(Op_k, Op_m) = \frac{1}{2}(J(res_k + res_m) + J(pas_k + pas_m))$$

where n_i is the number of provided interfaces of Microservice i. K is the number of microservices identified from a monolithic software. CHM_j measures the cohesion of I_i at message level. J is the Jaccard coefficient. Op_k and Op_m are operations provided by the interface

I_i, and $k \neq m$. f_M computes the similarity between two operations at the message level.

4) **Cohesion at domain level** (CHD) defines the cohesiveness of the interfaces provided by the domain level [6, 13]. The higher the value of CHM, the more cohesive the services.

$$CHD = \frac{\sum_{j=1}^{K} n_i \, CHD_j}{\sum_{i=1}^{K} n_i}$$

Where,

$$CHD_i = \begin{cases} \dfrac{\sum_{(k,m)} f_D(Op_k, Op_m)}{|I_i| \times (|I_i| - 1)/2}, & \text{if } |I_i| \neq 1 \\ 1, & \text{if } |I_i| = 1 \end{cases}$$

$$f_D(Op_k, Op_m) = J(T_{Op_k}, T_{Op_m)})$$

where, opr and O are the same symbols as those defined in chm. f_{dom} computes the similarity between operations. f_dom describes the set of domain terms contained in the signature of opr_i. J is the Jaccard coefficient.

For structural quality validation, microservices identified according to the cost formula given in eq. (1) are used. Similar to performance evaluation, the three highest-cost patterns with consecutive methods were selected as microservices.

Table III illustrates the structural properties of candidate microservices. The best-case results reported in existing benchmark approaches were used for this comparison. In general, *MIST* achieves the best or second best metric values, demonstrating that the microservices it identifies exhibit higher cohesion, lower coupling, and stronger structural modularity. Since the selection of patterns for microservice is based on consecutiveness, each microservice exposes only a single interface, resulting in an IFN value of one. Moreover, these independent sequences show no interaction, resulting in CHM and CHD values of one.

All experimental resources—including source code, collected log files, output files, deployment configurations, and test suites—are fully accessible. The experiments are fully replicable, and the results are reproducible. All associated artifacts are available in our publicly accessible repository.

V. Related Work

Migrating legacy software systems to a microservices architecture is a widely experimented area with multiple approaches. These approaches are commonly categorized into static, dynamic, and model- or domain-driven strategies. A hybrid approach integrates elements from these main categories [3]. Model- and domain-driven approaches use business processes, requirements, UML and design diagrams, data flow diagrams, system features, use cases, and other design artifacts to identify bounded contexts for microservices [3]. Service Cutter [19] is a key study that applies a domain-driven approach. Additionally, identification techniques based on data flow diagrams [20] and the analysis of product backlogs and user stories [21] have been developed within this category.

Most previous studies rely on static analysis, using source code to identify potential microservices [3]. In these approaches, classes and methods, along with their dependencies, are typically modeled as graphs, where highly connected components are identified as candidate microservices [22]. Source code can also be represented as abstract syntax trees, enabling semantic analysis through techniques such as topic modeling [23] and semantic similarity measures [24]. These analyses facilitate the construction of structural dependency and evolutionary coupling graphs [12], often incorporating information from semantic and historical change data [25].

Dynamic software analysis is another key technique for microservice identification that uses runtime execution data of software systems. The study on process mining-based decomposition framework [2] employs the Disco tool to analyze event logs and identify frequent system executions. Disco generates a graphical representation of the system's execution behavior, where arrow thickness reflects the frequency of trace executions. Potential microservices can then be inferred through visual inspection of the graphical representation, followed by a metric-based ranking to select the most suitable decomposition. The functionality-oriented Microservice Extraction (FoME) [5], instruments a system and runs test cases to produce logs. Execution traces from these logs are then grouped using hierarchical clustering to identify microservice candidates. Similarly, the Functionality-oriented Service Candidate Identification (FoSCI) framework [6] uses execution traces to discover service boundaries, identifying classes involved in each trace to support microservice identification. It collects the traces by executing the pre-determined functional test suite on monolithic software by instrumenting the software system. Execution trace reduction is then performed to identify the functional atoms that are the minimal coherent units with the same functional logic. Finally, functional atom grouping has been performed using Non-dominated Sorting Genetic Algorithm-II (NSGA II), a multi-objective optimization technique.

The *MIST* framework follows the general methodology used in prior dynamic analysis studies [2, 5, 6]. However, it differs from existing work in that no prior studies have examined how sequential patterns and their associated attributes influence microservice identification or the performance implications.

VI. Limitations and future work

The current implementation of *MIST* is built on the Kieker monitoring framework and the Java programming language. While the underlying principles of *MIST* – particularly the use of execution traces and sequential pattern mining – are independent of specific technologies. Future work will focus on tool-agnostic abstractions and cross-platform validation to demonstrate broader applicability.

This study concentrates on small, highly cohesive, loosely coupled, and modular microservices interacting with a centralized database. Although this controlled setting enables

TABLE III
STRUCTURAL PROPERTY EVALUATION; **BOLD** - BEST VALUE, UNDERLINE - SECOND BEST VALUE.

System	Property	MIST	FoSCI [6]	FOME [5]	MEM [12]	Micro Miner [13]	DRS [14]
Apache Roller	SM(+)	**0.644**	0.319		0.056		
	CHM(+)	**1.000**	0.833	0.700	0.779		0.760
	CHD(+)	**1.000**	0.682	0.900	0.385		0.53
	IFN(-)	**1.000**	1.288	1.750	15.000		
JForum	SM(+)	0.187	**0.435**		0.0359	0.040	
	CHM(+)	**1.000**	0.771	0.800	0.508	0.670	0.730
	CHD(+)	**1.000**	0.486	0.800	0.152	0.666	0.520
	IFN(-)	**1.000**	2.816	2.555	27.000	2.800	

focused experiments on performance improvements, it does not fully reflect the complexity of real-world microservices. In practice, microservices may vary significantly in granularity, often adopt a database-per-service pattern, and prioritize eventual data consistency, maintainability, and evolvability, while distributing business logic across services. Furthermore, challenges such as distributed transaction management, data partitioning strategies, and inter-service coordination are not addressed in the current work. Future research should systematically investigate how *MIST* principles perform under these realistic architectural constraints.

Execution traces are widely used for microservice identification, including the detection of dependent traces, call pattern discovery, and data dependency analysis. However, this study limits its analysis to sequential patterns derived from performance-oriented traces of small services. Broader trace characteristics- such as concurrent interactions, asynchronous communication, and cross-cutting concerns- remain unexplored and need to be focused on as future work. Moreover, systematizing microservice boundary detection by aggregating multiple frequent sequences, analyzing their implications for distributed transactions and data partitioning, and examining the effects of different communication sequences on microservice identification remains as future work.

Another limitation concerns the exclusion of runtime factors. In microservice architectures, performance is influenced not only by network communication overhead but also by security mechanisms, including authentication, authorization, encryption, and API gateway mediation. Future work should assess how security and the middleware layer influence both trace patterns, performance, and the result of microservice decomposition.

The empirical evaluation of this study is based on two case studies, which limits the generalizability of the findings. Although the results demonstrate the feasibility of the approach, future work can focus on validating the framework across a broader range of systems, including event-driven systems, heterogeneous technology stacks, decentralized persistence architectures, transactional systems, multi-domain enterprise applications, and real-life logs.

Although this study provides a valuable foundation by demonstrating that sequential patterns are promising candidates for microservice identification, the current approach to microservice identification relies on a cost model that assigns equal weights to parameters. Future work can refine this model to enable more flexible and systematic identification of microservices.

VII. CONCLUSIONS

This paper introduces *MIST*, a semi-automated framework for identifying microservices and evaluating how sequential patterns and related attributes affect their performance. By examining low-level, repetitive, and consistent method invocation sequences, *MIST* allows for the systematic decomposition of monolithic applications into independent, executable microservices. These services can be deployed separately while maintaining communication with other system components, ensuring operational continuity during the migration.

The results of the two case studies confirm that sequential patterns and related attributes can effectively support the migration of legacy systems to microservices. Among the five analyzed factors (support, confidence, pattern length, average depth, and average execution time), microservice identification based on support (S) is a stronger predictor of performance decline, whereas average depth is a strong predictor for improved performance. Furthermore, the *MIST* demonstrates high structural quality in all conducted evaluations, achieving 100% cohesion and zero coupling of the constructed microservices. This indicates that *MIST* can effectively produce small, highly cohesive, and loosely coupled service boundaries, essential for enhancing system functional independence, scalability, maintainability, and flexibility. Our case studies demonstrate that complex, tightly coupled monolithic systems can successfully migrate to microservices-based architectures, with possible performance improvements by leveraging sequential execution traces to identify meaningful, highly cohesive, and loosely coupled services.

The evaluation uses only two Java systems and one deployment setting, which limits external validity. The assessment focuses on throughput and latency under the chosen workloads, and thus captures only part of the quality of extracted microservices. Moreover, the implementation of candidate microservices requires engineering decisions that

may influence the measured results. Finally, the findings are based on a limited number of repeated runs. Nevertheless, the study provides a comparatively broad evaluation by combining candidate extraction, implementation, structural assessment, and runtime performance analysis, offering useful empirical evidence on a scale rarely reported in the literature.

Reproducibility. The source code of the *MIST* framework, test suites, and experiment results can be accessed at the following URL: https://anonymous.4open.science/r/MISt

REFERENCES

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly, 2015.

[2] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," in *CLOSER*. SciTePress, 2019, pp. 153–164.

[3] T. I. Mohottige, A. Polyvyanyy, C. Fidge, R. Buyya, and A. Barros, "Reengineering software systems into microservices: State-of-the-art and future directions," *Information and Software Technology*, vol. 183, 2025.

[4] M. Leemans and W. M. P. van der Aalst, "Process mining in software systems: Discovering real-life business transactions and process models from distributed systems," in *MODELS*, 2015, pp. 44–53.

[5] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai, "Functionality-oriented microservice extraction based on execution trace clustering," in *(ICWS*, 2018, pp. 211–218.

[6] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, vol. 47, pp. 987–1007, 2021.

[7] R. Capuano and H. Muccini, "A systematic literature review on migration to microservices: a quality attributes perspective," in *ICSA-C*, 2022, pp. 120–123.

[8] J. Han, M. Kamber, and J. Pei, "6 - mining frequent patterns, associations, and correlations: Basic concepts and methods," in *Data Mining (Third Edition)*. Morgan Kaufmann, 2012, pp. 243–278.

[9] N. Tax, N. Sidorova, R. Haakma, and W. M. P. van der Aalst, "Mining local process models," *J. Innov. Digit. Ecosyst.*, vol. 3, no. 2, pp. 183–196, 2016.

[10] B. Shahzaad, A. Barros, and C. Fidge, "Edge-mapping of service function trees for sensor event processing," in *ICWS*, 2024, pp. 1227–1237.

[11] A. K. Kalia, J. Xiao, R. Krishna, S. Sinha, M. Vukovic, and D. Banerjee, "Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices," in *FSE*, 2021, p. 1214–1224.

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

[13] I. Trabelsi, M. Abdellatif, A. Abubaker, N. Moha, S. Mosser, S. Ebrahimi-Kahou, and Y.-G. Guéhéneuc, "From legacy to microservices: A type-based approach for microservices identification using machine learning and semantic analysis," *Journal of Software: Evolution and Process*, vol. 35, no. 10, 2023.

[14] O. Al-Debagy, "A microservice decomposition method through using distributed representation of source code," *Scalable Comput. Pract. Exp.*, pp. 39–52, 2021.

[15] P. Fournier Viger, j. lin, U. Rage, Y. S. Koh, and R. Thomas, "A survey of sequential pattern mining," *Data Science and Pattern Recognition*, pp. 54–77, 2017.

[16] R. Agrawal and R. Srikant, "Mining sequential patterns," in *International Conference on Data Engineering (ICDE)*, 1995, pp. 3–14.

[17] P. Fournier-Viger, C. W. Lin, A. Gomariz, T. Gueniche, A. Soltani, Z. Deng, and H. T. Lam, "The SPMF open-source data mining library Version 2," in *PKDD*, 2016, pp. 36–40.

[18] D. Bajaj, A. Goel, and S. C. Gupta, "Greenmicro: Identifying microservices from use cases in greenfield development," *IEEE Access*, pp. 67 008–67 018, 2022.

[19] M. Gysel, L. Kölbener, W. Giersche, and O. Zimmermann, "Service cutter: A systematic approach to service decomposition," in *Service-Oriented and Cloud Computing*, 2016, pp. 185–200.

[20] S. Li, H. Zhang, Z. Jia, Z. Li, C. Zhang, J. Li, Q. Gao, J. Ge, and Z. Shan, "A dataflow-driven approach to identifying microservices from monolithic applications," *Journal of Systems and Software*, vol. 157, p. 110380, 2019.

[21] F. H. Vera-Rivera, E. G. Puerto-Cuadros, H. Astudillo, and C. M. Gaona-Cuevas, "Microservices backlog - a model of granularity specification and microservice identification," in *SCF*, 2020, p. 85–102.

[22] V. Nitin, S. Asthana, B. Ray, and R. Krishna, "Cargo: AI-guided dependency analysis for migrating monolithic applications to microservices architecture," in *ASE*, 2023.

[23] M. Brito, J. Cunha, and J. a. Saraiva, "Identification of microservices from monolithic applications through topic modelling," in *SAC*, 2021, p. 1409–1418.

[24] K. Sellami, M. A. Saied, and A. Ouni, "A hierarchical DBSCAN method for extracting microservices from monolithic applications," in *EASE*, 2022, p. 201–210.

[25] J. Löhnertz and A. Oprescu, "Steinmetz: Toward automatic decomposition of monolithic software into microservices," in *Seminar on Advanced Techniques and Tools for Software Evolution*, 2020.