Method-Level Syntactic and Semantic Clustering for Microservice Discovery in Legacy Enterprise Systems

ADAMBARAGE ANURUDDHA CHATHURANGA DE ALWIS[1], ALISTAIR BARROS[1], COLIN FIDGE[1], and ARTEM POLYVYANYY[2]
[1]Queensland University of Technology (QUT), Australia (e-mails: anuruddhadealwis@gmail.com, alistair.barros@qut.edu.au, c.fidge@qut.edu.au)
[2]The University of Melbourne, Australia (e-mail: artem.polyvyanyy@unimelb.edu.au)

Corresponding author: Colin Fidge (e-mail: c.fidge@qut.edu.au).

The research was supported in part through Australian Research Council Discovery Project DP220101516, *Embedding Enterprise Systems in IoT Fog Networks through Microservices*

ABSTRACT Enterprise systems, such as enterprise resource planning, customer relationship management, and supply chain management systems, are widely used in corporate sectors and are notorious for being large, inflexible and monolithic. Their many application-specific methods are challenging to decouple manually because they manage asynchronous, user-driven business processes and business objects having complex structural relationships. We present an automated technique for identifying parts of enterprise systems that can run separately as fine-grained microservices in flexible and scalable Cloud systems. Our remodularization technique uses both semantic properties of enterprise systems, i.e., domain-level business object and method relationships, together with syntactic features of the methods' code, e.g., their call patterns and structural similarity. Semantically, business objects derived from databases form the basis for prospective clustering of those methods that act on them as modules, while on a syntactic level, structural and interaction details between the methods themselves provide further insights into module dependencies for grouping, based on K-means clustering and optimization. Our technique was prototyped and validated using two open-source enterprise customer relationship management systems, SugarCRM and ChurchCRM. The empirical results demonstrate improved feasibility of remodularizing enterprise systems, inclusive of coded business objects and methods, compared to microservices constructed using class-level decoupling of business objects only. Furthermore, the microservices recommended, integrated with "backend" enterprise systems, demonstrate improvements in execution efficiency, scalability, and availability.

INDEX TERMS Microservice discovery, enterprise systems, system remodularization.

I. INTRODUCTION

Microservice (MS) architectures are the latest systems architectural style, evolving service-oriented architectures (SOAs) to enable high-performance, Internet-scale, continuously deployed, and composable applications [1], [2]. They support fine-grained and autonomous software components, which communicate via lightweight protocols and manage local and synchronized databases. Many research and practitioner articles highlight the common ground between an SOA and a microservice architecture, starting with both systems comprising component based services. The general properties of services stemming from SOAs, and extended to microservices, include: high cohesion (intra-module dependencies); low coupling (inter-module dependencies); and encapsulation functionality through operations focussed on the business domain of objects, which for enterprise systems (ESs) entail business objects (BOs) [1], [2]. The earliest microservice conceptions were framed on these properties, together with non-functional properties such as high scalability, availability and efficiency through execution-time decoupling [2], [3].

Despite the precedent of SOAs for microservice architectural concepts, prominent differences have emerged [4]. An SOA applies to a centralized system, where data is stored and accessed by components centrally, and service interactions between components are coordinated centrally. Instead, microservices operate in an autonomous and peer-to-peer fashion, where each manages its own local database, synchronized with those of other, distributed microservices. This allows microservices to be developed and deployed independently, while achieving overall performance improvements.

With the increasing demand for highly scalable and highly reliable services, service providers are now moving from

SOAs to microservices. The majority of reported experiences on microservice development to date concern "greenfield" developments [2], where microservices are produced from "scratch." Considerable uncertainty remains as to how microservices can be created by decoupling and reusing parts of an existing system through reengineering. This is of critical importance for corporate sectors relying on large-scale enterprise systems, e.g., enterprise resource planning (ERP) and customer-relationship management (CRM), to manage their operations. These systems often reflect decades of development and investment, and their uninterrupted, long-term operation is essential.

Analyzing enterprise systems to identify parts suitable for decoupling as microservices is technically cumbersome, given the millions of lines of code, thousands of database tables, and extensive functional dependencies typical of their implementations. The BOs embodied by the system have complex relationships to enable asynchronous, user-driven processes [5]–[8]. For example, an order-to-cash process in SAP ERP has multiple sales orders, with deliveries shared across different customers, shared containers in transportation carriers, and multiple invoices and payments, which may be processed before or after delivery [9]. This poses challenges for extracting effective and efficient microservices from enterprise system code using classical software reengineering principles.

In early research, software remodularization techniques [10]–[12] were proposed for static analysis of software systems to identify key characteristics and dependencies of modules, typically to abstract suitable classes using graph formalisms [13], [14]. New modules can be recommended using clustering algorithms and coupling and cohesion metrics. The focus of such static analysis techniques includes inter-module structure (class inheritance hierarchies), i.e., structural *inheritance* relationships, and inter-module interactions (object references), i.e., structural *interaction* relationships. Given that degradation of the logical design reflected in a software implementation can result in classes with low cohesion, other techniques have been proposed to compare structural properties of classes and methods using information retrieval techniques [12], i.e., structural class and method *similarity*.

At the same time, various microservice design patterns have emerged in the literature to help develop microservices by considering the semantics provided through BO relationships [15]–[18]. Nonetheless, the success rate of software remodularization projects remains low [19] and little work has been done on combining both semantic and syntactic aspects of software systems to derive microservices.

In our initial investigations into this problem, we previously presented a software remodularization technique for automatically discovering microservices in enterprise systems via features of software classes [20], [21]. This allowed us to identify multi-function software modules that combine classes and their related business objects as potential microservices. Here we refine that previous work to the level of individual *methods* to provide a finer-grained remodularization technique. As well as traditional create, read, update, and delete operations in enterprise systems, such an approach allows for modern Internet-based services that provide high-throughput, single-functionality operations such as video streaming [22].

Therefore, in this article we (a) consolidate our previous class-level microservice discovery techniques [20], [21], (b) refine them to the level of individual methods, and (c) directly compare the effectiveness of class-level and method-level microservice extraction from monolithic enterprise systems. "Syntactic" properties are derived from method interaction relationships and structural method similarity in the program code, while "semantic" properties are derived from analyzing the BO-to-BO relationships in the system's database and BO-to-method relationships.

To confirm the value of the approach, here we consider microservice recommendations produced by both our earlier class-level analysis [20] and our new method-level analysis, using the same enterprise system case studies and clustering techniques, to answer the following three research questions.

- **RQ1:** How do cohesion and coupling of microservices recommended by method-level and class-level analyses compare?
- **RQ2:** How do scalability, availability, and execution efficiency in microservice-based cloud environments recommended by method-level and class-level analyses compare?
- **RQ3:** How do scalability, availability, and execution efficiency in microservice-based cloud environments recommended by method-level analyses compare with microservices that do not follow the recommendations?

The remainder of this article: describes the background on system remodularization techniques and an architecture for remodularization of enterprise systems to microservices (Section II); presents our microservice discovery approach (Section III); describes the implementation and evaluation of our approach in order to answer the above research questions (Section IV); discusses related work (Section V); and reflects on conclusions drawn (Section VI).

II. BACKGROUND AND MOTIVATION

This section provides a brief review of existing software remodularization and microservice discovery techniques, including their relative strengths and weaknesses (Section II-A). Further detail can be found in the discussion of related research below (Section V). It then presents the architecture of enterprise systems and their links to microservices (Section II-B) as assumed by our microservice discovery and software remodularization technique (Section III).

A. SOFTWARE REMODULARIZATION TECHNIQUES

Software remodularization techniques [10], [12] involve analysis of different facets of systems, including software structure and behavior, functional and non-functional requirements, and provide recommendations for system reengineering (i.e., restructuring or redeveloping software components to obtain better code structure and performance). Static analysis techniques have focused on analyzing code structure and database schemas of software systems [23] while dynamic analysis studies interactions of systems [24]. Both approaches provide complementary information for assessing properties of system modules based on high cohesion and low coupling, and make recommendations for improved modularity. However, static analysis is preferable for broader units of analysis (i.e., systems or subsystems level) as all possible cases of a system's execution are covered compared to dynamic analysis which considers observed event sequences only [11].

Traditionally, research into software remodularization based on static analysis has focused on a system's implementation through two aspects of coupling and cohesion evaluation. The first is structural coupling and cohesion, which focuses on relationships between declared classes and methods in the program code. These include structural *inheritance* relationships between classes and structural *interaction* relationships which result when one class creates another class and uses an object reference to invoke its methods [10]. Structural relationships such as these are automatically profiled through Module Dependency Graphs (MDG) [10], [11], and are used to group classes using K-means, Hill-climbing, NSGA II and other clustering algorithms.

The second is structural class and method *similarity* (otherwise known as *conceptual similarity* of the classes and methods) [12]. This draws from information retrieval (IR) techniques, for source code comparison of classes and methods, under the assumption that similarly named variables, methods, object references, tables and attributes in database query statements, etc., infer conceptual similarity of classes and methods. Relevant terms are extracted from the classes and methods and used for latent semantic indexing and cosine comparison to calculate the similarity value between them. Class and method similarity thus provides intra-module measurements for evaluating coupling and cohesion, in contrast to the inter-module measurements provided through structural coupling and cohesion as described above.

Despite many proposals for automated analysis of systems, studies show that software remodularization remains a major challenge with a low success rate [19]. A prevailing problem is the limited insights produced by purely syntactic structures of software code for deriving structural and interactional relationships between modules. More recently, semantic insights available through BO relationships have been exploited to improve the feasibility of architectural analysis of applications. Enterprise systems manage domain-specific information using BOs, through the ES's databases and business processes [7]. Evaluating the BO relationships and deriving valuable insights from them to modularize software systems falls under the category of *semantic structural relationships* analysis.

Such semantic relationships are highlighted by Pẽrez-Castillo *et al.*'s experiments [25] in which the transitive closure of strong BO dependencies derived from databases was used to recommend software function hierarchies, and by Lu *et al.*'s experiments [26] in which SAP ERP logs were used to demonstrate process discovery based on BOs.

Our own previous research on microservice discovery based on BO relationship evaluation showed the value of considering semantic structural relationships in software remodularization [27], [28]. In this previous work, we highlighted the impact of integrating semantic structural relationships with class level structural inheritance relationship analysis and class level structural interaction relationship analysis [20]. However, microservices are fine-grained components typically based on the single responsibility principle, making them focus on single functionality [2] rather than broad units such as classes.

To achieve granularity at the level of single-function microservices both domain-driven design (i.e., object-based scoping) and capability-driven design (i.e., task-based scoping) are required. Capabilities, representing business tasks, are used to refine components that focus on individual BOs, rather than the classical scoping of components under the SOA paradigm. Furthermore, relationships between software components, i.e., intra- and inter-structural relationships such as object inheritance, association and aggregation [29] should be analyzed at the level of individual operations to obtain proper insights into system decomposition for microservices. Much research relies on expert knowledge [30], [31] to identify potential microservices or defines ‘‘classes as the smallest unit for microservice extraction’’ [32]. Therefore, in this paper, we focus on method-level analysis to perform a finer-grained analysis of systems than in previous research.

B. ARCHITECTURE FOR ENTERPRISE SYSTEM TO MICROSERVICE REMODULARIZATION

As detailed above, multiple factors must be considered for microservice derivation. In this section we define the importance of factors relating to the architectural configuration of the enterprise system and its microservices.

As depicted in Figure 1, an enterprise system consists of a set of self-contained modules drawn from different subsystems and deployed on a ‘‘backend’’ platform. A module consists of a set of software classes that are developed using object-oriented programming (OOP) languages such as Java, .Net, C# and PHP, and use object references to access and share inter- and intra-module data [33], [34]. The classes contain internal system operations and operations that manage BOs through create, read, update, and delete (CRUD) operations. For instance, in Figure 1 the classes in the ‘Order Management Module’ are ‘Class_Order’, ‘Class_OrderCal’ and ‘Class_OrderMan’. They all contain operations that manipulate data related to ‘Order’ BOs, whereas ‘Class_ProductVal’ contains operations manipulating data related to ‘Product’ BOs, and ‘Class_ECM’ has operations related to ‘ECM’ BOs.

Although operations reside in classes used to manipulate the data in BOs, some operations transfer or pass data between them for processing. This can lead to a situation where one operation calls (i.e., refers to) another operation, either requesting information or passing processed information as

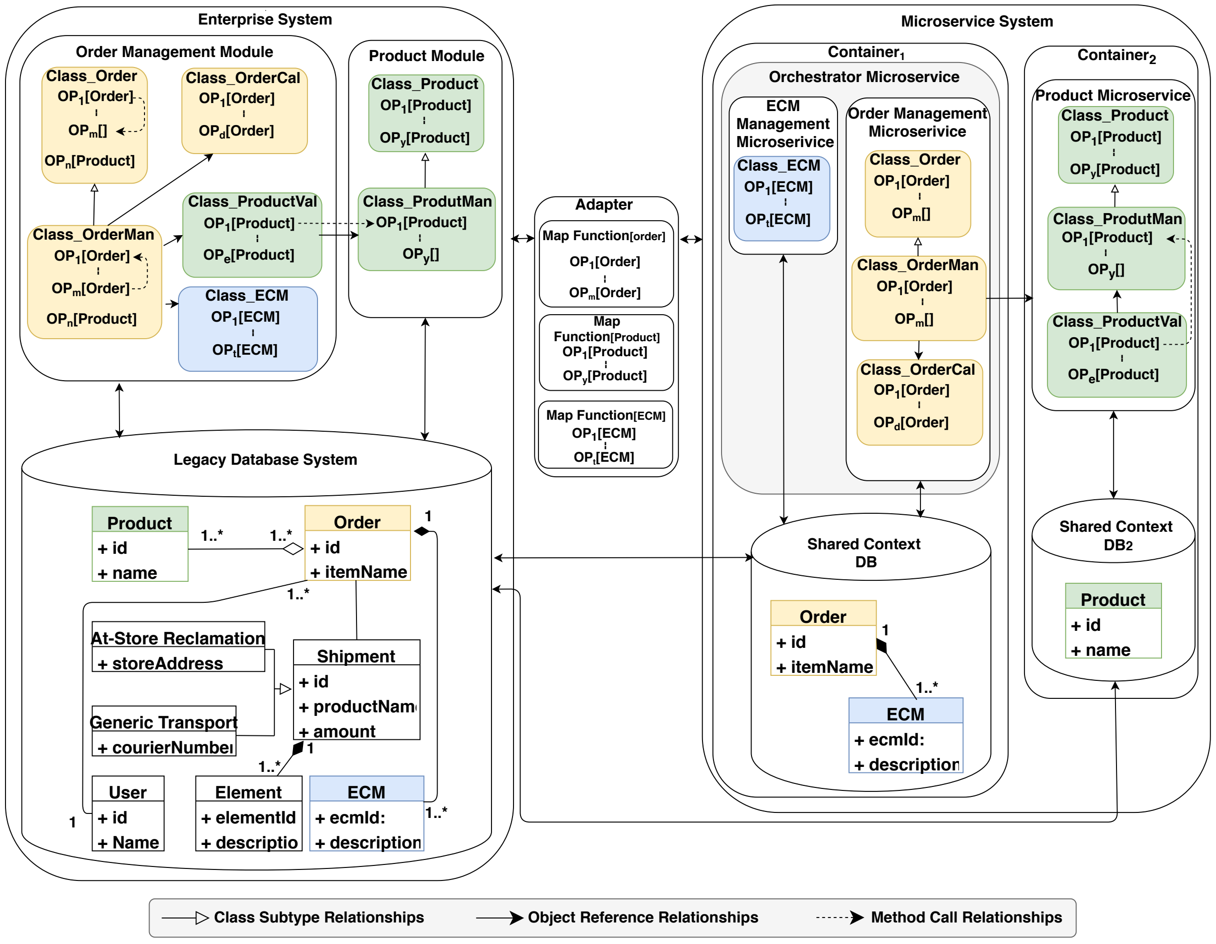


FIGURE 1. **Overview of an enterprise system extended with extracted microservices.**

parameters for additional processing. For example, in Figure 1 the relationship between operations ‘OP_1[Order]’ and ‘OP_m[Order]’ in ‘Class_OrderMan’, can be considered as a situation where ‘OP_1[Order]’ refers to ‘OP_m[Order]’ to extract some information related to the ‘Order’ BO. On the other hand, the relationship between ‘OP_1[Order]’ and ‘OP_m[]’ in ‘Class_Order’, can be considered as a situation where ‘OP_1[Order]’ passes some information to ‘OP_m[]’ for additional processing. Sometimes these operational relationships are extended between classes in different software modules as depicted by the operational relationship existing between ‘OP_1[Product]’ in ‘Class _ProductVal’ and ‘OP_1[Product]’ in ‘Class_ProductMan’ in which ‘Class_ProductVal’ belongs to the ‘Order Management Module’ and ‘Class_ProductMan’ belongs to the ‘Product Module’.

Data management of an enterprise system is performed through BOs residing in databases managed via CRUD operations, as depicted by the legacy database system in Figure 1. Such BOs can have complex lifecycle relationships [27], [28], e.g., *subtyping* is shown by the ‘At-Store Reclamation’ and ‘Generic Transport’ objects, which are specialized from ‘Shipment’. ‘Shipment’ objects are made up of ‘Elements’ and control their creation/deletion resulting in an *exclusive containment* relationship. An ‘Order’ object is made up of ‘Products’ but does not control their creation/deletion, and so embodies an *inclusive containment* relationship. Objects can also have general *association* relationships as depicted by ‘User’ and ‘Order’.

Microservices, on the other hand, support a subset of system operations through classes and methods which are related to individual BOs only. Such implementations lead to high cohesion within microservices and low coupling between the microservices (see the ‘Order Management Microservice’ and ‘Product Microservice’ in Figure 1). More specifically, these microservices contain operations related to the same BO while having generalization and object reference relationships. Developed in this way, microservices are able to maximize operational calls within modules and minimize operational calls between them. This is depicted by the ‘Product Microservice’ in which ‘Class_ProductVal’ is clustered with two other classes, which are highly related to the ‘Product’ BO while ensuring that operation ‘OP_1[Product]’ in ‘Class _ProductVal’ makes an internal object reference and operational calls to ‘OP_1[Product]’ in ‘Class_ProductMan’.

The microservices communicate with each other through API calls when they require information related to BOs residing in other microservices. For example, an ‘Order Management Microservice’ can acquire product data through an API call to the ‘Product Microservice’ as depicted in Figure 1. Execution of operations across the enterprise and microservice systems are coordinated through business processes, which means that invocations of BO operations on the microservices will trigger operations on enterprise system functions involving the same BOs. As is necessary in distributed systems, BO data must be synchronized across databases managed by microservices and the enterprise system periodically.

Based on this understanding of the structure of enterprise systems and microservices, syntactic relationships can be analyzed at two levels, i.e., at the class level and the method level, while including the semantics provided by the object relationships. Evaluation of class level relationships to discover microservices is detailed in our previous work [20], in which we used the structural *inheritance* and *interaction* relationships for class clustering. However, such class-level syntactic and semantic analysis produces coarse-grained components for microservice derivation, since each class contains multiple methods. To obtain finer-grained components, in this article the unit of analysis is shifted from classes to methods.

Interactions between the methods (i.e., calls) can be identified using structural interaction relationship analysis. Such techniques do not capture method similarities at the code level but this information can be captured using structural method similarity measurement. Based on both structural interaction relationships and structural similarity, we can then use clustering techniques to group methods into modules. To do this, we study the relationships between business objects and methods. In Figure 1, different classes and methods in the enterprise system relate to different BOs. The relationships between these BOs can be used to identify (at least) five microservice patterns as follows [27], [28].

- **Subtyping** is where a microservice manages subsets of operations related to a BO subtype. Since there can be multiple BO subtypes of a supertype, a microservice can apply to any given subtype, at any subtype hierarchy level, and a subset of operations related to that subtype.
- **Read-Write Separation** is where microservices manage subsets of CRUD operations related to a BO such that each microservice supports either read operations or create, update and delete operations, i.e., read and write operations are not mixed in the same microservice.
- **Exclusive Containment** occurs when one microservice relates to a parent (or composite) BO and other microservices manage its child BOs, such that the existence of a child BO depends on the continuing existence of the parent BO, via create, update and delete operations.
- **Inclusive Containment** occurs when one microservice relates to a parent (or aggregate) BO and other microservices manage its child BOs, but the existence of the child BOs does not depend on the parent’s existence. The children exist independently but are used in the context of the parent.
- **Association** is where one microservice relates to one BO and the other microservice manages another BO, but the existence of both BOs are independent of each other. Since there is no dedicated tight-coupling, there is no requirement to strictly co-locate them.

This understanding emphasizes the importance of considering semantic structural relationships in the microservice derivation process, since each microservice should aim to contain methods that are related to each other and perform operations on the same BO (like the ‘Order Management Microservice’ and ‘Product Microservice’ in Figure 1).

Previous research has extensively used structural relationships in system remodularization [10]–[12]. However, when it comes to microservice derivation, our research assumes that combining both *semantic structural* and *syntactic structural* relationships will allow derivation of better method clusters suitable for microservice implementation. Previously we did this at the level of software classes [20], but here refine the concept to the method level, using the above system architecture context and our understanding of the features that should be evaluated for microservice systems to develop our algorithms for microservice discovery (Section III).

III. CLUSTERING RECOMMENDATION FOR MICROSERVICE DISCOVERY

In our previous class-level research we used a five-step microservice discovery process [20], [21], and again follow the same strategy for the method-level experiments described herein.

1) We identify the *business objects* manipulated by the enterprise system by evaluating the SQL queries in the source code and also the database schemas and data as explained by Nooijen *et al.* [35].
2) We identify *semantic structural relationships* by deriving the relationships between SQL queries in methods and the corresponding BOs.
3) We analyze the enterprise system’s syntactic structure firstly by measuring the *structural method similarities* between methods, based on the ‘bag of words’ [36] produced by extracting meaningful identifiers from the methods’ source code.
4) We capture the *structural interaction relationships* between different methods by analyzing their code to discover their call patterns.
5) The details obtained through Steps 2 to 4 are used by a K-means clustering algorithm to recommend effective combinations of methods for microservice deployment.

More detail about the specific algorithms used appears below, adapted from our previous class-level process [20], [21]. These algorithms were then implemented as a prototype recommendation system as explained in Section IV-A and used to produce the experimental results discussed in Section IV-B.

We use the following formalization from here onwards to describe the algorithms. Let $\mathbb{I}$, $\mathbb{O}$, $\mathbb{OP}$, $\mathbb{B}$, $\mathbb{T}$, and $\mathbb{A}$ be a universe of *input* types, *output* types, *operations*, *business objects*, *database tables*, and *attributes*, respectively.

We characterize a *database table* $t \in \mathbb{T}$ by a collection of attributes, i.e., $t \subseteq \mathbb{A}$, while a *business object* $b \in \mathbb{B}$ is defined as a collection of database tables, i.e., $b \subseteq \mathbb{T}$ containing information about the same BO. Let A^* denote application of the Kleene star operation to set A. An *operation op*, either of an enterprise system or a microservice system, is given as a triple (I, O, T), where $I \in \mathbb{I}^*$ is a sequence of *input types* the operation expects for input, $O \in \mathbb{O}^*$ is a sequence of *output types* the operation produces as output, and $T \subseteq \mathbb{T}$ is a set of *database tables* the operation accesses, i.e., either reads or augments. Each *class* $cls \in CLS$ is defined as a collection of operations, i.e., $cls \subseteq \mathbb{OP}$.

In order to derive a satisfactory clustering of system methods for microservice recommendations, directly comparable to our previous class-level clustering [20], we supply a K-means clustering algorithm with different feature sets. (We used K-means since it helps us evaluate multiple features and obtain rational clusters, although equivalent results may be possible using different clustering techniques such as hierarchical clustering, B-Squared, error-based clustering, RB Clustering, or Direct clustering [37], [38].)

To derive these feature sets, we use Algorithm 1. This method-level discovery algorithm was also used in our earlier research into microservice discovery for Industrial Internet-of-Things applications [21], and a longer description of it can be found there.

In the first step let BOS (line 2) be a function used to derive BOs B from enterprise systems as detailed by Nooijen *et al.* [35], and $CLSEXT$ (line 3) be a function which searches through different folder and package structures and extracts classes' source code. The extracted classes CLS are then used in function $MTDEXT$ (line 4) which extracts the source code of the methods related to the classes.

The second step then extracts the information needed for structural method similarity analysis using information retrieval (IR) techniques. Let $UWORDEXT$ (line 5) be a function on the source code of the methods MTD, and unhelpful "stop words" STW [39] which provide no insights into the code's purpose. In our scenario, these are standard keywords related to the method definitions in the particular programming language (PHP here) and common English words, as determined by the user. The function filters out the stop words from the given method, identifies words that are not already in unique word sequence UW, and adds them. The result is a 'bag of words' [36] relevant to the enterprise system's purpose. For instance, a typical enterprise system's code will contain meaningful words such as 'Product', 'Order', 'Inventory', 'Purchase', 'Shipment', 'Customer', 'Tax', and so on.

In the third step the algorithm evaluates each method mtd to identify their related business objects. Let $BCOUNT$ (line 8) be a function on a method's code that counts the SQL statements, comments and method names that refer to database tables forming a particular business object. This information is stored in matrix $mtdborel$ (lines 6–8) which contains the number of references that each method i makes to each business object k. These numbers help capture the semantic structural relationships, providing a measure of the "boundedness" of methods to BOs.

Similarly, in the fourth step, let $WCOUNT$ (line 10) be a function which returns the number of times a method i is related to a unique word s. It is used to construct matrix $mtduwcount$.

The values in $mtduwcount$ are then used to calculate the cosine similarity between each pair of methods i and k using function $MTDCOSINECAL$ (lines 11 and 12). This function first normalizes the term frequencies with the respective magnitude L2 norms, then calculates the cosine similarity between two given methods, by calculating the cosine value between the corresponding vectors of matrix $mtduwcount$ (i.e., those rows related to the two methods) and saves the values in matrix $mtdcosine$. This step provides the conceptual method similarity data we require for clustering. The lower the cosine similarity, the lower the possibility of methods being related to the same concept.

In the sixth step, the algorithm extracts the structural interaction relationships (i.e., method call relationships). Let $MTDRELCAL$ (line 13) be a function on the source code which first produces graphs of classes, their methods, and the method call relationships. (In practice we used the Mondrian code analysis tool[1] to do this.) The function then analyzes the graphs to create binary matrix $mtdrel$ which summarizes the method call relationships between any two methods. Value $mtdrel[i][j]$ is 1 if method i calls method j and 0 otherwise.

The feature set data in matrices $mtdborel$, $mtdcosine$, $mtdrel$ (see the example in Figure 2) and the BOs B obtained from Algorithm 1 are then provided as input to Algorithm 2 to cluster the methods related to BOs based on their syntactic and semantic relationships. This algorithm was used in our previous class-level experiments [20], so is re-used here to allow a direct comparison of the class-level and method-level results. (An earlier, shorter description of the algorithm was provided previously [20], but for clarity here we have expanded the explanation of the concrete example in Figure 2.)

Another input to the algorithm is an array of initial centroid values. Each $intcent \in IntCent$ is a row number in the data set that we provide. For example, one can select the first row of the data set as we have in Figure 2, which is highlighted in red as an initial centroid point. In that situation, $IntCent$ will contain the data related to that specific row of the data set.

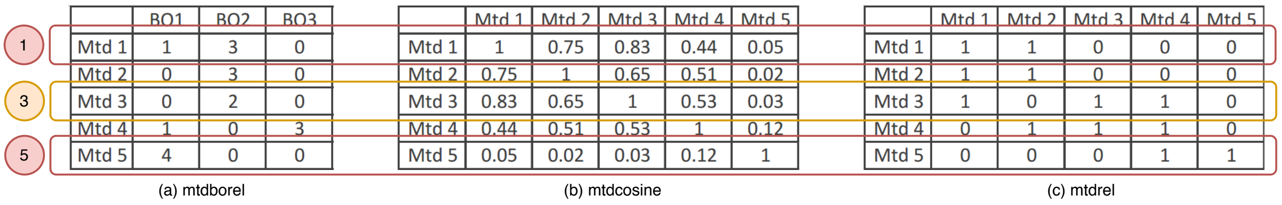
Given these data sets as input to Algorithm 2, we initialize a distance difference value $distDif$ to some constant, e.g., 10 (line 1). Variable $distDif$ is responsible for capturing the distance differences between the initial centroids $IntCent$ and the newly calculated centroids $NewCent$ (line 14). The algorithm terminates when this difference is zero (line 2).

The first step in each iteration is to initialize a set of clusters $CLUS$ (line 3), which we use to store the node groups identified by the algorithm. To start the process, let $INITCLUSTERS$ be a function which allocates random "nodes", i.e., rows in Figure 2, to each of k clusters.

[1]https://github.com/Trismegiste/Mondrian

Algorithm 1: Discovery of BO and method relationships

```
Input: System code $SC$ of an enterprise system $s$, stop words related to methods $STW$ and system database $DB$
Output: Classes $CLS$, methods $MTD$, feature set matrices $mtdborel$, $mtdcosine$, $mtdrel$ and BOs $B$
1  Initialise matrices $mtdborel$, $mtduwcount$ and $mtdcosine$ with zeros
2  $B = \{b_1, \dots, b_n\} \leftarrow BOS(SC, DB)$;
3  $CLS = \{cls_1, \dots, cls_m\} \leftarrow CLSEXT(SC)$;
4  $MTD = \{mtd_1, \dots, mtd_m\} \leftarrow MTDEXT(CLS)$;
5  $UW = \langle uw_1, \dots, uw_z \rangle \leftarrow UWORDEXT(MTD, STW)$
6  for $mtd_i \in MTD$ do
7      for $b_k \in B$ do
8          $mtdborel[i][k] \leftarrow BCOUNT(mtd_i, b_k)$;
9      for $uw_s \in UW$ do
10         $mtduwcount[i][s] \leftarrow WCOUNT(uw_s, mtd_i)$;
11 for $mtd_i$, $mtd_k \in MTD$ do
12     $mtdcosine[i][k] \leftarrow MTDCOSINECAL(mtduwcount[i], mtduwcount[k])$;
13 $mtdrel \leftarrow MTDRELCAL(MTD)$;
14 return $CLS$, $MTD$, $mtdborel$, $mtdcosine$, $mtdrel$, $B$;
```



(a) mtdborel

	BO1	BO2	BO3
Mtd 1	1	3	0
Mtd 2	0	3	0
Mtd 3	0	2	0
Mtd 4	1	0	3
Mtd 5	4	0	0

(b) mtdcosine

	Mtd 1	Mtd 2	Mtd 3	Mtd 4	Mtd 5
Mtd 1	1	0.75	0.83	0.44	0.05
Mtd 2	0.75	1	0.65	0.51	0.02
Mtd 3	0.83	0.65	1	0.53	0.03
Mtd 4	0.44	0.51	0.53	1	0.12
Mtd 5	0.05	0.02	0.03	0.12	1

(c) mtdrel

	Mtd 1	Mtd 2	Mtd 3	Mtd 4	Mtd 5
Mtd 1	1	1	0	0	0
Mtd 2	1	1	0	0	0
Mtd 3	1	0	1	1	0
Mtd 4	0	1	1	1	0
Mtd 5	0	0	0	1	1

FIGURE 2. Example method relationship matrices derived using Algorithm 1 and used as input into Algorithm 2

Algorithm 2: K-Means clustering for microservice discovery

```
Input: Feature set matrices $mtdborel$, $mtdcosine$ and $mtdrel$, $k$ which is the number of BOs $B$, and the initial Centroid
        values $IntCent$
Output: Set $CLUS$ which captures the clustered microservice recommendations
1  $distDif \leftarrow 10$;
2  while $distDif \neq 0$ do
3      $CLUS = \{clus_1, \dots, clus_k\} \leftarrow INITCLUSTERS(k)$;
4      for $0 \leq i <$ size of $mtdborel$ do
5          $minEuclideanDis \leftarrow 10,000$;
6          for $intcent_j \in IntCent$ do
7              $newEuclideanDis \leftarrow EUCAL(intcent_j, mtdborel[i], mtdcosine[i], mtdrel[i])$;
8              if $newEuclideanDis < minEuclideanDis$ then
9                  $minEuclideanDis \leftarrow newEuclideanDis$;
10                 $clusterNumber \leftarrow j$;
11         $clus_{clusterNumber} \leftarrow clus_{clusterNumber} + i$;
12     for $clus_i \in CLUS$ do
13         $NewCent = \{newcent_1, \dots, newcent_n\} \leftarrow NEWCENTCAL(clus_i)$;
14     $distDif \leftarrow DISTANCECAL(IntCent, NewCent)$;
15     $IntCent \leftarrow NewCent$;
16 return $CLUS$
```

Next, we need to identify the cluster that each row of our dataset should belong to by comparing the distance between each node in the dataset and each node in the initial centroids $intcent \in IntCent$. Hence we iterate through each row of the dataset we obtained from Algorithm 1 (line 4 in Algorithm 2), while calculating the Euclidean distance between each row and each initial centroid $intcent \in IntCent$ (lines 5 to 10).

For this calculation, as the initial step, we define the minimum Euclidean distance value $minEuclidianDis$ and initialize it to a very large value, e.g., 10,000 (line 5). We assign such a huge value to distance $minEuclidianDis$ to ensure it is larger than the value it may receive at the end of the iteration (line 9).

Then we calculate the Euclidean distance between one data set, for example, row 1 in Figure 2, and each initial centroid point given (lines 6 to 10). Next, we identify the centroid that has the minimum Euclidian distance to the node we obtained (line 9) and allocate that node number to that particular cluster $clus \in CLUS$ (line 10). This process is carried out until all the nodes are towards the centroid, with a minimal distance based on the three feature sets, $mtdborel$, $mtdcosine$ and $mtdrel$.

This clustering emphasizes that the methods related to a particular cluster are bound to the same BO and are bound to each other syntactically and semantically. When calculating the Euclidean distances, we calculate them separately for each feature set. Let $EUCAL$ (line 7) be a function which, given a centroid value and three rows from the feature sets, returns the sum of their Euclidean distances from the centroid. To illustrate the calculation done by $EUCAL$, consider the feature set in Figure 2, where vectors across all three matrices represent the "nodes" used in the algorithm. Given Node 1 and Node 3 in Figure 2, we calculate the Euclidean distance between the feature sets of $mtdborel$, $mtdcosine$ and $mtdrel$ separately and then get the overall Euclidean distance value by summing them. We do this because we want to give more focus to the $mtdborel$ and $mtdrel$ features, which emphasize actual syntactic and semantic structural relationships of the classes. As such, we multiply the Euclidean distances they produce by two. Applying this calculation to Nodes 1 and 5 in Figure 2, then the $mtdborel$ distance is 18, the $mtdcosine$ distance is 3.0803 and the $mtdrel$ distance is 3, which leads to a total distance of $(18 \times 2) + 3.0803 + (3 \times 2)) = 45.0803$.

The next step is to calculate the new centroids based on the clusters obtained (lines 12 and 13). K-means clustering identifies the nodes related to each cluster by measuring the distance from the center node to the other nodes. Let $NEWCENTCAL$ be a function used to calculate the mean distances of the node data sets belonging to each cluster. This set is assigned to the new centroid $NewCent$. The algorithm's initial iteration identifies the set of nodes related to each cluster. After this first clustering, each iteration takes the average of each node related to each cluster, which becomes the new centroid, i.e., center point, of that cluster.

Finally, let $DISTANCECAL$ (line 14) be a function which returns the distance difference between two centroids. It is used to find the distance between the initial centroids and the new centroids, *distDif* . If this difference is zero, it means that there is no change of the centroid points and the algorithm has come to the optimum clustering point, so stops (line 2). If not, the newly calculated centroids become the initial centroids for the next iteration of the algorithm (line 15). The final set of clusters, each of which contains a set of methods related to the same BO, are the result (line 16), and can be used as the basis for creating new microservices.

IV. IMPLEMENTATION AND VALIDATION

To demonstrate the applicability of the approach described in Section III, we developed a prototype microservice recommendation system[2] capable of discovering method clusters related to different business objects, to produce different microservice configurations. The system was tested against two open-source customer relationship management systems, SugarCRM[3] and ChurchCRM[4]. SugarCRM consists of more than 8,000 files and 600 attributes in 101 tables, while ChurchCRM consists of more than 4,000 files and 350 attributes in 55 tables. However, most of the files are HTML files which relate to third-party components used by the systems. For the clustering, we used only the 1,400 program code classes of SugarCRM and 280 classes of ChurchCRM which capture the systems' core functionalities.

Using our implementation, we performed static analyses of the source code to identify the BOs managed by the systems. As a result, 18 BOs were identified in SugarCRM, e.g., 'account', 'campaign', and 'user', and 11 BOs in ChurchCRM, e.g., 'user', 'family', and 'email'. Then we performed static analyses of both systems to derive matrices, similar to those depicted in Figure 2, summarizing the BO relationships, method similarity relationships and method call relationships. All the obtained results were processed by our prototype to identify method clusters as the basis for recommending microservices. Based on the input, the prototype identified 18 method clusters related to the BOs in SugarCRM and 11 method clusters related to the BOs in ChurchCRM. Each such cluster suggests methods for developing a microservice that relates to a single business object.

A. EXPERIMENTAL SETUP

To answer our three research questions we conducted multiple experiments. Experiments to evaluate class-level clustering alone were conducted in our previous work [20] so we have not repeated the details here, but have included the empirical results below for comparison with our new experiments. Below we detail the new experimental setup we used to evaluate method-level clustering for microservice discovery. The results from both sets of experiments are compared in Section IV-B.

For method-level evaluation, we set up the following experiment consisting of three steps. In the first, we evaluated the

[2]https://github.com/AnuruddhaDeAlwis/KMeans.git
[3]https://www.sugarcrm.com/
[4]http://churchcrm.io/

effectiveness of method clustering based on the feature sets that we derived (i.e., feature 1: *mtdborel*, feature 2: *mtdcosine*, feature 3: *mtdrel*). We evaluated this by measuring the Lack of Cohesion (LOC) and Structural Coupling (StrC) of the clusters, as detailed by Candela *et al.* [19]. The values for the enterprise system were calculated by first grouping the classes into folders while conserving the original package structure, as per the first rows in Tables 1 to 4. Then we clustered the classes created using our new method-level analysis. The method-level values are reported in the last rows of Tables 1 to 4. Note that we have recorded the cohesion and coupling values we obtained in our previous class-level analysis [20] in the second rows in Tables 1 to 4 for comparative purposes.

After evaluating the effectiveness of various features for clustering, we assessed the efficacy of introducing microservices to the enterprise system. To this end, we first hosted each enterprise system in an AWS cloud by creating two EC2 instances having two virtual CPUs and a total memory of 2GB, as depicted on the left side of Figure 3. The systems' data were stored in a MySQL relational database instance which has one virtual CPU and total storage of 20GB. Afterward, these systems were tested against 100 and 200 executions generated by four machines simultaneously, simulating customer requests. We recorded the total execution time, average CPU consumption, and average network bandwidth consumption for these executions, as shown in Tables 5 and 6. For SugarCRM, we tested the functionality related to process "campaign creation", while for ChurchCRM we tested the functionality related to "adding new people" to the system. The simulations were conducted using Selenuim[5] scripts which ran the system in a way similar to a real user.

Next, we introduced 'campaign' and 'user' microservices to the SugarCRM system and 'person' and 'family' microservices to the ChurchCRM system which were developed based on the method clusters suggested by our algorithms. For SugarCRM the CRUD method clusters suggested were related to the 'campaign' and 'user' BOs and for ChurchCRM the CRUD method clusters suggested were related to the 'person' and 'family' BOs.

As depicted on the right side of Figure 3, we hosted each microservice on an AWS elastic container service (ECS), containing two virtual CPUs and a total memory of 1GB. The BO data of each microservice (i.e., the campaign BO and user BO data of SugarCRM and the person BO and family BO data of ChurchCRM) were stored in separate MySQL relational database instances with one virtual CPU and a total storage of 20GB. Afterward, tests were performed on both enterprise systems, again simulating "campaign creation" for SugarCRM and "adding new people" for ChurchCRM.

Since the microservices were refactored parts of the enterprise systems in these tests, the enterprise systems used API calls to pass the data to the microservices and the microservices processed and sent back the data to the enterprise systems. The microservice databases and the enterprise system databases were synchronized using the Amazon database migration service. This helps to preserve consistency between the ES database and the microservices. Furthermore, the databases' ACID properties are preserved through this synchronization.

Again, we recorded the total execution time, average CPU consumption, and average network bandwidth consumption for the entire system, i.e., the enterprise system and microservices as a whole, as per the third and fourth rows in Tables 5 and 6. The scalability, availability and execution efficiency of the systems were calculated based on the attained values. The results obtained are summarized in the third rows of Tables 7 to 9 as "ES & method-level MSs". Scalability was calculated according to the resource usage over time, as described by Tsai *et al.* [40]. To determine availability, first we calculated the packet loss for one minute when the system is down and then obtained the difference between the total up-time and total time, i.e., up-time + down-time, as described by Bauer *et al.* [41]. Dividing the total time taken by the legacy system to process all requests by the total time taken by the corresponding enterprise system which has microservices defined the efficiency gain.

In the third experiment, we created deliberately poorly-designed, "arbitrary" microservices that did not follow the suggestions provided by our recommendation system. We developed 'campaign' and 'user' microservices for SugarCRM, while introducing operations relating the 'campaign' and 'user' microservices and operations relating the 'user' and 'campaign' microservices, respectively. Similarly, for ChurchCRM, we developed 'person' and 'family' microservices such that the 'person' microservice contains operations related to the 'family' microservice and the 'family' microservice contains operations related to the 'person' microservice. With this change, we again set up the experiment as described earlier and obtained the experimental results shown in the final two rows in Tables 5 and 6. Then we calculated the scalability, availability and execution efficiencies of the systems which are summarized in the final rows in Tables 7 to 9 as "ES & 'arbitrary' MSs".

B. EXPERIMENTAL RESULTS

The outcomes of all of our experiments are shown in Tables 1 to 9. Tables 1 and 2 compare "lack of cohesion" and "structural coupling" for each of the clusters related to the 11 business objects identified in ChurchCRM and SugarCRM, respectively, showing values for the original enterprise system, the microservices recommended by our previous class-level analysis [20], and the newly-obtained results using method-level analysis. The last column shows average "lack of cohesion" values across all BOs—the lower the value the higher the cohesion. Similarly, Tables 3 and 4 do the same for the 18 BOs extracted from SugarCRM.

Tables 5 and 6 show various performance characteristics for SugarCRM and ChurchCRM, respectively. Values are shown for each system stressed by 100 and 200 simultaneous

[5]https://www.seleniumhq.org/

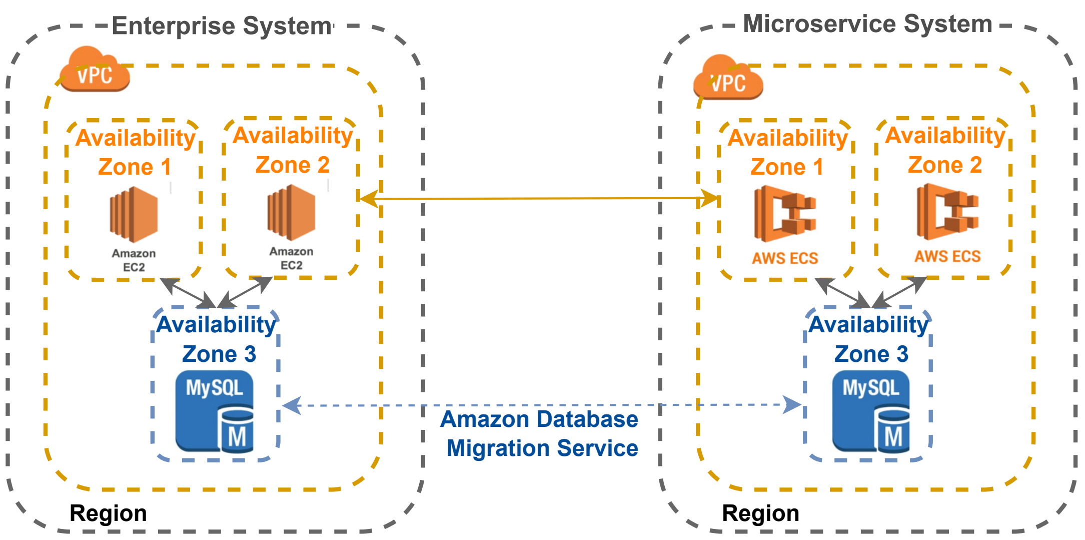


FIGURE 3. System implementation using Amazon Web Services.

TABLE 1. Lack of Cohesion comparison for 11 ChurchCRM business objects: Legacy Enterprise System, and Class-level and Method-level microservices.

BO #	1	2	3	4	5	6	7	8	9	10	11	Avg.
Enterprise System	61	188	853	7	4	1065	31	378	3064	13	17	516.45
Class-level MSs	58	188	820	7	3	1059	31	351	3012	10	15	504.90
Method-level MSs	351	325	28	325	325	253	153	276	210	91	210	234

TABLE 2. Structural Coupling comparison for ChurchCRM.

BO #	1	2	3	4	5	6	7	8	9	10	11	Avg.
Enterprise System	41	26	61	17	16	70	29	31	123	27	19	41.81
Class-level MSs	42	25	34	17	15	63	29	3	112	26	17	34.81
Method-level MSs	48	3	11	3	3	31	25	55	6	8	27	20

TABLE 3. Lack of Cohesion comparison for 18 SugarCRM business objects.

BO #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Avg.
Enterprise System	32	19	1255	698	1482	0	163	693	349	45	171	1803	1058	0	66	47	317	522	484.44
Class-level MSs	19	0	1201	626	1173	0	86	459	170	45	120	1027	587	0	36	5	590	268	356.22
Method-level MSs	15	3	1005	520	973	64	56	415	170	21	120	953	587	0	34	8	313	231	298.22

requests, comparing performance by the legacy enterprise system, the microservices recommended by our new method-level analysis, and those produced by "poorly-designed" microservices that fail to follow our recommendations.

Tables 7 and 8 show other system characteristics for SugarCRM and ChurchCRM, respectively, again with each system stressed by 100 and 200 simultaneous requests. (This was done only for the AWS EC2 instances.) Values are shown for the legacy enterprise system alone, the ES with microservices created using our previous class-level clustering [20], the ES with method-level clusters, and the ES with deliberately poorly-designed microservices. Table 9 shows database performance characteristics under the same circumstances.

Based on these experimental results we evaluated the effectiveness of the algorithms by answering the posed research questions. Note that in order to answer **RQ2** we have summarized the results we obtained using class-level microservice discovery [20] in the second rows in Tables 7 to 9.

RQ1: Cohesion and Coupling Impact. When comparing the cohesion and coupling results achieved by class-level and method-level analysis, it is evident that the method-level analysis provided better cohesion and coupling values for both SugarCRM and ChurchCRM (Tables 1 to 4). Specifically, the method-level analysis improved the cohesion of software modules of ChurchCRM and SugarCRM by 54.69% and 2.009% and the coupling of software modules by 52.16% and 40.23%, respectively, when compared with the original enterprise systems' values. This is a considerable improvement over class-level analysis, i.e., 2.24% improvement in cohesion for ChurchCRM and 18.75% and 16.74% improvement in coupling for SugarCRM and ChurchCRM, respectively.

It is important to note that the class-level clustering was based on four different feature sets as detailed in our previous research [20]. Our previous paper highlighted the improve-

TABLE 4. Structural Coupling comparison for SugarCRM.

BO #	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Avg.
Enterprise System	24	12	121	63	48	4	99	24	29	29	28	101	67	0	16	10	82	85	46.77
Class-level MSs	12	2	116	53	48	4	87	18	22	29	27	79	45	0	15	3	41	83	38.00
Method-level MSs	3	7	16	14	16	33	23	33	96	6	56	45	14	69	21	0	18	32	27.89

TABLE 5. Performance for SugarCRM: Legacy Enterprise System, ES with recommended microservices, and with "poorly-designed" microservices.

System Type	No. of Requests	Exec. Time (ms)	Avg. CPU Percentage EC2	Avg. CPU Percentage DB	Avg. Network (Kbs) DB
ES only	100	672000	3.212	1.61	5.67
ES only	200	1302000	2.963	1.97	6.42
ES & recommended MSs	100	648000	3.413	1.67	5.66
ES & recommended MSs	200	1302000	2.865	1.58	4.98
ES & "arbitrary" MSs	100	660000	3.103	1.67	5.32
ES & "arbitrary" MSs	200	1320000	3.108	1.60	5.36

TABLE 6. Performance for ChurchCRM.

System Type	No. of Requests	Exec. Time (ms)	Avg. CPU Percentage EC2	Avg. CPU Percentage DB	Avg. Network (Kbs) DB
ES only	100	150000	3.215	1.71	8.453
ES only	200	264000	4.905	2.013	12.32
ES & recommended MSs	100	114000	3.225	1.813	7.638
ES & recommended MSs	200	222000	2.080	1.730	9.702
ES & "arbitrary" MSs	100	120000	3.275	1.713	10.015
ES & "arbitrary" MSs	200	228000	3.373	1.636	9.969

TABLE 9. Database characteristics for SugarCRM and ChurchCRM.

System Type	SugarCRM Scalability [CPU]	SugarCRM Scalability [Network]	ChurchCRM Scalability [CPU]	ChurchCRM Scalability [Network]
ES only	2.972	2.759	3.109	3.405
ES & class-level MSs	2.532	2.352	2.751	3.663
ES & method-level MSs	2.529	2.351	2.749	3.653
ES & "arbitrary" MSs	2.544	2.680	2.657	2.769

ment we obtained over traditional syntactic analysis [10]–[12] by introducing BO semantics. The results detailed in the second rows of Tables 1 to 4 were the best values obtained after introducing all the syntactic and semantic aspects in the class-level clustering process. By contrast the method-level analysis detailed herein improved the cohesion and coupling values even further, as detailed in the third rows of Tables 1 to 4. As such, the new approach presented in this paper outperforms both traditional syntactic clustering and even our own previous class-level clustering technique.

RQ2: Scalability, availability and efficiency impact. According to Tsai *et al.* [40], the lower the measured number, the better the scalability (Tables 7 to 9). Thus, it is evident that the microservices developed based on the method-level suggestions provided by our recommendation system for SugarCRM and ChurchCRM managed to achieve: (i) 36.4% and 47.9% scalability improvement in EC2 instance CPU utilization, respectively; (ii) 14.9% and 11.6% scalability improvement in database instance CPU utilization, respectively; while achieving (iii) 3.7% and 31.6% execution efficiency, respectively, as shown in the third rows of Tables 7 to 9. As such, our recommendation system discovers MSs that have better cohesion and coupling values than the original enterprise system's modules and can achieve improved cloud capabilities such as high scalability, high availability and high execution efficiency.

Furthermore, it is evident that the microservices developed for SugarCRM and ChurchCRM based on the method-level analysis were able to achieve 0.58% and 0.16% scalability improvement compared to the microservices developed for SugarCRM and ChurchCRM based on class-level analysis (the second rows of Tables 7 to 9). However, there is little difference in availability between the microservices developed based on both approaches.

RQ3: Recommended microservices vs arbitrary microservices. Compared to the poorly-designed, "arbitrary" microservices, those developed based on the method-level suggestions provided by our recommendation system for SugarCRM and ChurchCRM managed to achieve: (i) 36.4% and 47.9% scalability improvement in EC2 instance CPU utilization, respectively; (ii) 14.9% and 11.6% scalability improvement in database instance CPU utilization, respectively; while achieving (iii) 3.7% and 31.6% execution efficiency, respectively. However, when microservices were developed *in opposition to* the suggestions provided by the recommendation system, the EC2 instance CPU utilization was reduced to 24.24% and 19.32% for SugarCRM and ChurchCRM, respectively, while database instance CPU utilization was reduced to 14.3% for SugarCRM. Furthermore, the execution efficiency of SugarCRM and ChurchCRM was reduced to 1.8% and 2.5%, respectively. As such, it is evident that the microservices developed by following the recommendations of our system produce better system characteristics than those of microservices developed against our recommendations.

C. LIMITATIONS

In this section we discuss two limitations of our approach.

Limitation of structural method similarity analysis. In

TABLE 7. System characteristics for SugarCRM EC2: Legacy Enterprise System, and ES with class-level, method-level and "poorly-designed" microservices.

System Type	Scalability [CPU]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
ES only	3.521	99.115	99.087	1.000	1.000
ES & class-level MSs	2.246	99.082	99.086	1.037	1.000
ES & method-level MSs	2.238	99.081	99.086	1.037	1.000
ES & "arbitrary" MSs	2.662	99.098	99.099	1.018	0.986

TABLE 8. System characteristics for ChurchCRM EC2.

System Type	Scalability [CPU]	Availability [100]	Availability [200]	Efficiency [100]	Efficiency [200]
ES only	3.565	99.385	99.418	1.000	1.000
ES & class-level MSs	1.859	95.000	94.871	1.316	1.189
ES & method-level MSs	1.856	95.000	94.871	1.316	1.189
ES & "arbitrary" MSs	2.874	95.238	95.000	1.250	1.158

this research we used cosine similarity, partly because it's used elsewhere in the literature for similar clustering applications and, more importantly, so that we could directly compare our new method-level results with those we previously produced at the class-level using cosine similarity [20]. However, the cosine values calculated might not provide an accurate insight into structural method similarity since the structural similarity may also depend on the terms used in the definitions of the method names and descriptions given in comments. This was mitigated to a certain extent by evaluating the code structure of the software systems before evaluating and verifying that the method names and comments provide useful insights into the logic behind the methods that implement the system. Nonetheless, there remain a number of ways in which the discovery and matching process could be improved, such as recognizing situations in which different terms represent specifications or generalizations of the same concept.

Limitation of microservice data orchestration. Given that microservices are implemented independently with their own databases, they must be able to synchronize their data with the central database system and the other microservices if required. Even though we achieved this to a certain extent using the AWS data migration service that synchronizes the central database with the microservices, data synchronization between different microservices was not tested in the research described herein.

V. RELATED WORK

Software remodularization [11] is considered as an NP-hard graph partitioning problem. Traditionally, static analysis-based software remodularization research has focused on a system's implementation through two areas of coupling and cohesion evaluation. The first is *structural coupling and cohesion*, which focuses on structural relationships between classes and methods in the same module or in different modules [10], [11], [42]. The second is *structural class/method similarity* (otherwise known as *conceptual similarity* of the classes/methods) [12], [43]. This draws from information retrieval (IR) techniques, for source code comparison of classes/methods, under the assumption that similarly named variables, methods, object references, tables and attributes in database query statements, etc., infer conceptual similarity of classes/methods.

Microservice derivation is the newest form of software remodulization and, as in our own research, various approaches have typically relied on both primitive database operations to help identify business objects and class hierarchies to help identify business-level operations [44], but there has been limited work to date on automating the process.

However, research into service-oriented architectures has identified patterns that can help to develop microservices. Early examples provided the main groups for such patterns, namely decomposition, integration, database, observability and cross-cutting concern patterns [45], [46]. Richardson proposed thirteen different architecture segments based on the functions they perform, namely decomposition, data management, transactional messaging, testing, deployment, cross-cutting concern, communication style, external API, service discovery, reliability, security, observability and UI patterns [47], as well as 44 patterns for implementing microservices [48]. Messina *et al.* focussed on the database tier; the service is strictly coupled to the data, hence this pattern is even stronger than the "database-server per service" pattern, because the database itself acts as a business service [49]. Microsoft has provided generalized patterns for cloud implementations, which include most of the patterns described by previous sources such as the "circuit breaker," "command query responsibility segregation," and "gateway aggregation" patterns [47]. Zimmerman *et al.* [18] provide a consolidated development of microservice patterns and categorizes them into: processing responsibilities of systems (e.g., API, event, activity and other forms of processing and information); data structures (e.g., parameter, object or linked elements); quality management considerations (e.g., service level agreements); and software evolution principles (e.g., version management).

With increasing interest in microservice architectures and the development of microservice patterns, recent studies have focussed on frameworks of comparison. A major influence on the architectural concepts and patterns developments is systems quality considerations [50]. In a recent survey [51], numerous patterns were identified from the literature, classified in broad systems categories (DevOps, Frontend, Backend, Orchestration, Migration and IoT patterns), and linked to quality attributes (flexibility, testability, elasticity, performance, scalability, etc).

When we consider the essence of most of these patterns it is evident that they stem from early capability-driven and domain-driven design principles [52], which basically say that microservices are developed around business objects. As such, each microservice should be a component which contains operations related to a single BO. Thus, business objects provide important semantic aspects in microservice derivation and development [2]. For instance, function hierarchies can be derived from BO dependencies in databases [25] and ERP logs can support process discovery based on BOs [26]. Considering such BO relationships, our own research captures both "syntactic" and "semantic" relationships in the MS derivation process by considering the information inherent in the program code and captured in system execution logs and database relationships [20], [21], [27], [28].

However, system execution logs can provide only limited syntactic insights about the system. Klock *et al.* optimized static relationships to improve already developed microservice systems by clustering different microservices to groups which perform highly related tasks [31] and, more recently, Ding *et al.* prioritised runtime performance as a heuristic for extracting microservices from legacy systems [32]. However, to the best of our knowledge, previous research still lacks a technique which combines the analysis of both the syntactic and semantic relationships during system remodularization for microservices. In our previous work we presented class-level analysis of enterprise systems while considering syntactic and semantic aspects of the system to help develop microservices [20]. In the current article we have further extended this approach to method-level analysis.

VI. CONCLUSION

We have presented a method-level technique for automated analysis and remodularization of enterprise systems as microservices by combining techniques which consider semantic knowledge about business object relationships with syntactic knowledge about operation call structures. Unlike previous class-level approaches, which typically produce multi-function services, our technique has the potential to identify smaller-scale, single-functionality microservices as are needed in contemporary high-throughput, data-oriented networks [22].

This article answered three research questions designed to compare the microservice recommendations made by previous class-level analyses with method-level analyses. To do so a prototype recommendation system was developed based on the algorithms presented herein and validation was conducted by implementing the microservices recommended by the prototype for two open source enterprise systems, SugarCRM and ChurchCRM. The experiment showed that the method-level approach derived clusters which led to microservices with better cohesion and coupling characteristics than those based on class-level analysis.

Specifically, the method-level analysis improved the cohesion of software modules of ChurchCRM and SugarCRM by 54.69% and 2.009% and coupling of software modules by 52.16% and 40.23% respectively, when compared with the enterprise systems' values. This is a considerable improvement compared to the results obtained by class-level clustering (i.e., 2.24% improvement in cohesion for ChurchCRM and 18.75% and 16.74% improvement in coupling for SugarCRM and ChurchCRM, respectively). This answered the first research question (**RQ1**).

Furthermore, the microservices developed based on the suggestions provided by the prototype achieved better scalability, availability and processing efficiency than microservices which do not align with the recommendations. Specifically, the microservices developed based on the suggestions provided by the recommendation system for SugarCRM and ChurchCRM managed to achieve: (i) 36.4% and 47.9% scalability improvement in EC2 instance CPU utilization, respectively; (ii) 14.9% and 11.6% scalability improvement in database instance CPU utilization, respectively; while achieving (iii) 3.7% and 31.6% execution efficiency, respectively. Furthermore, it was evident that the microservices developed for SugarCRM and ChurchCRM based on the method-level analysis were able to achieve 0.58% and 0.16% scalability (refer to the second rows of Tables 7 to 9). This was a substantial improvement compared to the microservices developed for SugarCRM and ChurchCRM based on class-level analysis. This answered the second and third research questions (**RQ2, RQ3**).

As future work we will extend the method-level analysis to derive components which are compatible for Internet of Things (IoT) integration. Given that IoT applications mainly focus on processing at network edge locations, this will require derivation of components which need only low processing power. We believe the single operation separation achievable by method-level analysis will provide insights capable of identifying such components.

REFERENCES

[1] T. Erl, *SOA Design Patterns*. Pearson Education, 2008.

[2] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, Inc., 2021, second edition.

[3] M. Fowler, "Microservices Guide," August 2019, https://martinfowler.com/microservices/.

[4] T. Cerny, M. J. Donahoo, and J. Pechanec, "Disambiguation and comparison of SOA, microservices and self-contained systems," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems (RACS)*, 2017, pp. 228–235.

[5] A. Barros, K. Duddy, M. Lawley, Z. Milosevic, K. Raymond, and A. Wood, "Processes, roles, and events: UML concepts for enterprise architecture," in *International Conference on the Unified Modeling Language*. Springer, 2000, pp. 62–77.

[6] T. Schneider, *SAP Business ByDesign Studio: Application Development*. Galileo Press Boston, 2012.

[7] G. Decker, A. Barros, F. M. Kraft, and N. Lohmann, "Non-desynchronizable service choreographies," in *Proceedings of the International Conference on Service-Oriented Computing (ICSOC 2008)*. Springer, 2008, pp. 331–346.

[8] A. Barros, G. Decker, and M. Dumas, "Multi-staged and multi-viewpoint service choreography modelling," in *2007 Proceedings of the Workshop on Software Engineering Methods for Service Oriented Architecture (SEMSOA), Hannover, Germany. CEUR Workshop Proceedings*, vol. 244. Citeseer, 2007, pp. 1–15.

[9] A. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation patterns in service-oriented architectures," in *Proceedings of the 2007 International Conference on Fundamental Approaches to Software Engineering*. Springer, 2007, pp. 245–259.

[10] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Transactions on Software Engineering*, vol. 37, no. 2, pp. 264–282, 2010.

[11] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the Bunch tool," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[12] D. Poshyvanyk and A. Marcus, "The conceptual coupling metrics for object-oriented systems," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance*. IEEE, 2006, pp. 469–478.

[13] T. Matias, F. Correia, J. Fritzsch, J. Bogner, H. Ferreira, and A. Restivo, "Determining microservice boundaries: A case study using static and dynamic software analysis," in *Proceedings of the 2020 European Conference on Software Architecture (ECSA)*, ser. Lecture Notes in Computer Science, A. Jansen, I. Malavolta, H. Muccini, I. Ozkaya, and O. Zimmermann, Eds., vol. 12292. Springer Verlag, 2020.

[14] 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, Oct. 2022.

[15] L. Krause, "Microservices: Patterns and applications: Designing fine-grained services by applying patterns," 2014, self published.

[16] E. Gamma, *Design patterns: Elements of reusable object-oriented software*. Pearson Education India, 1995.

[17] N. Malhotra, "Microservices design patterns," ValueLabs, Tech. Rep., 2023, white paper. [Online]. Available: https://www.valuelabs.com/wp-content/uploads/2023/05/Microservices-Design-Patterns.pdf

[18] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges*. Addison-Wesley, 2022.

[19] I. Candela, G. Bavota, B. Russo, and R. Oliveto, "Using cohesion and coupling for software remodularization: Is it enough?" *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 3, p. 24, 2016.

[20] A. A. C. De Alwis, A. Barros, C. Fidge, and A. Polyvyanyy, "Remodularization analysis for microservice discovery using syntactic and semantic clustering," in *Proceedings of the 32nd International Conference on Advanced Information Systems Engineering (CAiSE 2020)*, ser. Lecture Notes in Computer Science, vol. 12127. Springer-Verlag, 2020, pp. 3–19.

[21] ——, "Microservice remodularisation of monolithic enterprise systems for embedding in industrial iot networks," in *Proceedings of the 33rd International Conference on Advanced Information Systems Engineering (CAiSE 2021)*, ser. Lecture Notes in Computer Science, vol. 12751. Springer, 2021, pp. 432–448.

[22] T. Mauro, "Adopting microservices at Netflix: Lessons for architectural design." Feb. 2015, https://www.f5.com/company/blog/nginx/microservices-at-netflix-architectural-best-practices.

[23] M. Joselyne, G. Bajpai, and F. Nzanywayingoma, "A systematic framework of application modernization to microservice based architecture," in *Proceedings of the 2021 International Conference on Engineering and Emerging Technologies (ICEET), Turkey*. IEEE, 2021, pp. 1–6.

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

[25] R. Pérez-Castillo, I. García-Rodríguez de Guzmán, I. Caballero, and M. Piattini, "Software modernization by recovering web services from legacy databases," *Journal of Software: Evolution and Process*, vol. 25, no. 5, pp. 507–533, 2013.

[26] X. Lu, M. Nagelkerke, D. van de Wiel, and D. Fahland, "Discovering interacting artifacts from ERP systems," *IEEE Transactions on Services Computing*, vol. 8, no. 6, pp. 861–873, 2015.

[27] A. A. C. De Alwis, A. Barros, C. Fidge, and A. Polyvyanyy, "Discovering microservices in enterprise systems using a business object containment heuristic," in *Proceedings of the 26th Interenational Conference on Cooperative Information Systems (CoopIS 2018)*, ser. Lecture Notes in Computer Science, vol. 11230. Springer-Verlag, 2018, pp. 60–79.

[28] A. A. C. De Alwis, A. Barros, A. Polyvyanyy, and C. Fidge, "Function-splitting heuristics for discovery of microservices in enterprise systems," in *Proceedings of the 16th International Conference on Service Oriented Computing (ICSOC 2018)*, ser. Lecture Notes in Computer Science, vol. 11236. Springer-Verlag, 2018, pp. 37–53.

[29] T. Halpin and T. Morgan, *Information modeling and relational databases*. Morgan Kaufmann, 2010, second edition.

[30] W. Assunção, T. Colanzi, L. Carvalho, and J. Pereira, "A multi-criteria strategy for redesigning legacy features as microservices: An industrial case study," in *Proceedings of the 2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021.

[31] S. Klock, J. M. E. Van Der Werf, J. P. Guelen, and S. Jansen, "Workload-based clustering of coherent feature sets in microservice architectures," in *Proceedings of the IEEE International Conference on Software Architecture (ICSA 2017)*. IEEE, 2017, pp. 11–20.

[32] Z. Ding, Y. Xu, B. Feng, and C. Jiang, "Microservice extraction based on a comprehensive evaluation of logical independence and performance," *IEEE Transactions on Software Engineering*, vol. 50, no. 5, May 2024.

[33] SAP, "Tools and programming languages," Aug. 2015, https://developers.sap.com/topics/tools-programming-languages.html.

[34] IFS World Operations, "IFS applications architecture," Jun. 2025, https://docs.ifs.com/techdocs/Foundation1/010_overview/100_architecture.

[35] E. H. Nooijen, B. F. van Dongen, and D. Fahland, "Automatic discovery of data-centric and artifact-centric processes," in *Proceedings of the 2012 International Conference on Business Process Management*. Springer, 2012, pp. 316–327.

[36] G. Lebanon, Y. Mao, and J. Dillon, "The locally weighted bag of words framework for document representation," *Journal of Machine Learning Research*, vol. 8, pp. 2405–2441, Oct. 2007.

[37] G. Shahmohammadi, S. Jalili, and S. M. H. Hasheminejad, "Identification of system software components using clustering approach." *Journal of Object Technology*, vol. 9, no. 6, pp. 77–98, 2010.

[38] K. Mahdavi, M. Harman, and R. M. Hierons, "A multiple hill climbing approach to software module clustering," in *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*. IEEE, 2003, pp. 315–324.

[39] A. Zaman, P. Matsakis, and C. Brown, "Evaluation of stop word lists in text retrieval using Latent Semantic Indexing," in *Proceedings of the Sixth International Conference on Digital Information Management*. IEEE, 2011, pp. 133–136.

[40] W.-T. Tsai, Y. Huang, and Q. Shao, "Testing the scalability of SaaS applications," in *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2011, pp. 1–4.

[41] E. Bauer and R. Adams, *Reliability and Availability of Cloud Computing*. John Wiley & Sons, 2012.

[42] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse, "Towards automatically improving package structure while respecting original design decisions," in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 212–221.

[43] A. Marcus, D. Poshyvanyk, and R. Ferenc, "Using the conceptual cohesion of classes for fault prediction in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 34, no. 2, pp. 287–300, 2008.

[44] F. Freitas, A. Ferreira, and J. Cunha, "Refactoring Java monoliths into executable microservice-based applications," in *Proceedings of the 25th Brazilian Symposium on Programming Languages (SBLP '21)*. Association for Computing Machinery, 2021, pp. 100–107.

[45] R. Bhojwani, "Popular design patterns for microservices architectures," Aug. 2022, https://dzone.com/articles/popular-design-patterns-for-microservices-architec.

[46] M. Udantha, "Microservice architecture and design patterns for microservices," Jul. 2019, https://dzone.com/articles/microservice-architecture-and-design-patterns-for.

[47] C. Richardson, *Microservices from Design to Deployment*. Springer, 2016.

[48] ——, *Microservices Patterns with Examples in Java*. Manning, 2018.

[49] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso, "The database-is-the-service pattern for microservice architectures," in *Proceedings of the International Conference on Information Technology in Bio- and Medical Informatics*. Springer, 2016, pp. 223–233.

[50] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, and M. A. Babar, "Understanding and addressing quality attributes of microservices architecture: A systematic literature review," *Information and Software Technology*, p. 106449, 2020.

[51] G. Márquez, H. Astudillo, and R. Kazman, "Architectural tactics in software architecture: A systematic mapping study," *Journal of Systems and Software*, vol. 197, Mar. 2023.

[52] E. Evans and E. J. Evans, *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

DR. ADAMBARAGE ANURUDDHA CHATHURANGA DE ALWIS is a software engineer with certifications in Google Cloud (Professional Cloud Architect and Associate Cloud Engineer) and Amazon Web Services (Associate Cloud Architect). He completed his Ph.D. at the School of Information Systems, Queensland University of Technology, Australia. His research interests include service development and management, system reengineering, cloud computing and optimisation. His current research is focused on reengineering enterprise systems into microservices.

PROF. ALISTAIR BARROS is Head of the School of Information Systems, Queensland University of Technology, Australia. He has ICT experience across industrial, academic, industrial R&D and industry roles, including as Global Research Leader and Chief Development Architect at SAP AG, the third largest software company worldwide. His research interests include the design, engineering and evolution of enterprise systems platforms in contemporary cyber-physical settings, supported by Cloud, Internet-of-Things and Blockchain infrastructure. He has led large research projects across Europe and Australia including for the Smart Services CRC, Internet of Services projects of EU Framework Program 7 and German BMBF, and Australian Research Council projects.

DR. COLIN FIDGE is an Adjunct Professor affiliated with the School of Computer Science, Queensland University of Technology. His research interests include safety-critical, mission-critical and security-critical systems engineering, especially in the defence and energy sectors. He has extensive experience in research project management, including leading 23 externally-funded projects, 14 through Australian Research Council grants, one from the Australian Learning and Teaching Council and a variety of others through industry and government funding in the areas of industrial asset management and cyber security.

A/PROF. ARTEM POLYVYANYY is an Associate Professor at the School of Computing and Information Systems, Faculty of Engineering and Information Technology, at the University of Melbourne (Australia), where he leads the Process Science and Technology research group. He is a Vice-Chair of the Steering Committee of the IEEE Task Force on Process Mining. His research and teaching interests include Computing Systems, Distributed Systems, Process Mining, Process Querying, Artificial Intelligence, and Algorithms. Artem is the editor of and a contributor to the book entitled *Process Querying Methods*.