Microservice Remodularisation of Monolithic Enterprise Systems for Embedding in Industrial IoT Networks

Adambarage Anuruddha Chathuranga De Alwis[1][0000-0002-4954-6595], Alistair Barros[1][0000-0001-8980-6841], Colin Fidge[1][0000-0002-9410-7217], and Artem Polyvyanyy[2][0000-0002-7672-1643]

[1] Queensland University of Technology, Brisbane, Australia
{adambarage.dealwis,alistair.barros,c.fidge}@qut.edu.au
[2] The University of Melbourne, Parkville, Victoria, Australia
artem.polyvyanyy@unimelb.edu.au

Abstract. This paper addresses the challenge of decoupling "back-office" enterprise system functions in order to integrate them with the Industrial Internet-of-Things (IIoT). IIoT is a widely anticipated strategy, combining IoT technologies managing physical object movements, interactions and contexts, with business contexts. However, enterprise systems, supporting these contexts, are notoriously large and monolithic, and coordinate centralised business processes through software components dedicated to managing business objects (BOs). Such objects and their associated operations are difficult to manually decouple because of the asynchronous and user-driven nature of the business processes and complex BO dependencies, such as many-to-many and aggregation relationships. Here we present a software remodularisation technique for enterprise systems, to support the discovery of fine-grained microservices, which can be extracted and embedded to run on IIoT network nodes. It combines the semantic knowledge of enterprise systems, i.e., the BO structure, with syntactic knowledge of the code, i.e., various dependencies at the level of classes and methods. Using extracted feature sets based on both semantic and syntactic dependencies, K-Means clustering and optimisation is then used to recommend microservices, i.e., redistributions of BO operations through microservices from BO-centric components of enterprise systems. The approach is validated using the Dolibarr open source ERP system, in which we identify processes comprising both "edge" operations and request-response calls to the Cloud-based enterprise system. Through experimentation using Amazon GreenGrass deployments, simulating IIoT nodes, we show that the recommended microservices demonstrate key non-functional characteristics, of high execution efficiency, scalability and availability.

Keywords: microservice discovery, system remodularisation, cloud migration.

1 Introduction

The Industrial Internet of Things (IIoT) is widely expected to transform automation processes of construction, manufacturing, utilities and other asset-intense sectors through the real-time integration of physical environments and enterprise systems. Under the IoT, physical object movements, interactions and contexts are tracked and controlled through sensors and actuators, and data is transceived, via gateways, with Cloud systems providing intelligent analytics. The IIoT extends the scope of coordination to

business contexts, where the processes, rules and data of enterprise systems are opened up through IoT devices and contexts. Examples from construction [28] include: real-time tracking of physical construction/assembly work against production schedules and constraints (e.g., time allocation, stock use, wastage and budget impact); automatic re-ordering of products, in-situ, subject to stock threshold levels and supplier contract conditions; and automatic "wayfinding" of new stock to demand points on large sites. Such examples require that software components of enterprise systems be integrated with, and partially embedded to run on, IIoT nodes, to support low-latency, real-time processing. In addition, IoT (and thus IIoT) networks have recently been endowed with distributed computing tiers, through developments in Fog computing. As such, IIoT nodes support processors, designated as the master, worker and edge nodes, each of which can host and run parts of systems. This means, an enterprise system could have its parts simultaneously deployed to run across the nexus of Cloud and IIoT nodes while being connected to other distributed processes [21].

However, major uncertainty exists as to how microservices, compatible with IIoT, can be created by decoupling and reusing parts of existing enterprise systems. This is essential to preserve continuity with, and exploit the large investment in, enterprise systems that have been developed over many years. Such systems [5] manage thousands of inter-dependent BOs, across a multitude of software packages and support asynchronous and unstructured business processes [6–8]. For example, an order-to-cash process in SAP ERP has multiple sales orders, with deliveries shared across many customers, shared containers in transportation carriers, and multiple invoices and payments, processed before or after delivery [9]. This poses challenges for identifying fine-grained, modular tasks, to implement as IIoT-based microservices.

Microservices must exhibit high cohesion, low coupling, object encapsulation and composability, as per basic modularisation principles [10–12]. Applied to enterprise systems, they provide subsets of BO create, read, update and delete operations (corresponding to decomposed business tasks). Microservices should also improve the scalability, availability (resilience) and execution efficiency of the overall system [3]. Therefore, an efficient re-distribution of BO operations is required from existing enterprise systems components, reflecting these properties. Specifically, highly dependent operations of a BO need to be combined into highly scalable, available and efficient microservices, while the business processes, across existing enterprise systems and newly introduced microservices, must still execute correctly.

Software remodularisation techniques have been proposed to scan different aspects of systems, extract relevant structural and behavioural feature sets, and recommend new modules using multi-objective optimisation. They have focussed on a system's code implementation, or syntactic properties, through two areas of coupling and cohesion evaluation. The first is structural coupling and cohesion [10], involving structural relationships between the software classes in the same or in different components. These include structural inheritance relationships between classes and structural interaction relationships resulting when one class creates another class and uses an object reference to invoke its methods. The structural relationships are automatically profiled through Module Dependency Graphs (MDG), capturing classes as nodes and structural relationships as edges [11], and are used to cluster classes using K-means, Hill-climbing, and

other clustering algorithms. The second form is structural class similarity [12] based on information retrieval (IR) techniques, for source code comparison of classes. Relevant terms are extracted from the classes and used for latent semantic indexing and cosine comparison to calculate similarity values between them.

Nonetheless, despite many proposals for automated analysis of systems, studies show that the success rate of software remodularisation remains low given the limited insights available from purely syntactic structures [13].

More recently, semantic knowledge available through BOs of enterprise systems has been exploited to improve the feasibility of applications' architectural analysis [16]. Our previous research on MS discovery from enterprise systems for cloud deployments, involving analysis of source code and systems logs, similarly exploits knowledge of BO relationships [17, 18]. This was based on class-level feature set extractions for software remodularisation analysis: structural inheritance relationships (class supertypes and subtypes), structural interaction relationships (class level creations and invocations), structural class similarity (intra-class level), and class semantic properties (class and BO dependencies for BOs managed through classes). However, for the highly distributed context of the IIoT, more fine-grained dependency analysis is critical and must be at the level of individual methods (i.e., operations of classes).

Here we present a novel combination of syntactic and semantic remodularisation analysis techniques. It applies both static (source code) and dynamic (event log) analysis to extract crucial dependencies between classes of components, between classes and BOs, and between BOs, and uses these insights to reason more reliably about fine-grained, remodularisation and effective distribution at the level of class methods for IIoT applications. It uses the following feature set extractions: method interactions (intra- and inter-class), method similarity (intra-method level), and method semantic properties (method and BO dependencies for BOs manipulated through SQL statements in methods). Recommended clusters of operations for creating microservices are based on subsets of BO operations.

We validated the technique using an open-source Enterprise Resource Planning system, Dolibarr. Amazon Greengrass was used for testing the recommended microservices for the required non-functional properties of high scalability, availability and execution efficiency. Greengrass nodes were used as IIoT nodes to host and run the test microservices. The microservices then ran business processes involving BO operations and made request-response calls to corresponding BO components in Dolibarr.

The remainder of the paper is structured as follows. Section 2 describes the related works and background on system remodularisation techniques. Section 3 provides a detailed description of our microservice discovery approach while Section 4 describes its implementation and evaluation. Section 5 discusses the outcomes and possible future work. The paper concludes with Section 6.

2 Background and Motivation

This section provides details of existing software remodularisation and reengineering techniques while comparing their relative strengths and weaknesses. We then give an overview of the architectural context of enterprise systems and their alignments with

microservices for the IIoT. This context is assumed in the presentation of our software remodularisation techniques in Section 3.

2.1 Related Work and Techniques Used for Software Remodularisation

Software remodularisation techniques have been introduced to analyse different facets of systems, including software structure, behaviour, functional requirements, and non-functional requirements. They focus on the behavioural and structural aspects of software systems. The static analysis applies to code structure and the database schemas of software systems while dynamic analysis involves the mining of systems logs for method invocations occurring at run time. Both of these techniques can be used to provide complementary details in the system remodularisation process.

Traditional static analysis techniques are used to remodularise software systems in order to improve the coupling and cohesion of system modules. These are based on structural interaction relationships between classes and object reference relationships between classes resulting when one class creates another class and uses an object reference to invoke its methods [10]. These relationships are profiled through Module Dependency Graphs (MDG) while capturing classes as nodes and structural relationships as edges [10,11]. They are used to cluster methods using K-means, Hill-climbing, NSGA II and other clustering algorithms. Some other techniques were developed to evaluate class-level relationships by considering their conceptual similarity using information retrieval (IR) techniques [12].

However, given the code's complexity and the semantic complexity of the structural interaction relationships, such analyses are not enough. As such structural method similarity (i.e., *conceptual similarity*) [12] was introduced to capture semantic similarities between methods using information retrieval (IR) techniques. This technique compares methods under the assumption that similarly named variables, object references, etc., infer conceptual similarity of methods. The extracted terms from methods are used for latent semantic indexing and cosine comparison to calculate the similarity values between them.

Despite many proposals for automated analysis of systems, studies show that the success rate of software remodularisation remains low [13]. One of the major reasons for this is the limited insights available from purely structural system analysis which only focuses on the systems' source code. Recent research shows that the semantic insights available through BO relationships provide information regarding the systems' behavioural aspects and these can be exploited to improve the feasibility of applications' architectural analysis. Enterprise systems manage domain-specific information using BOs, through their databases and business processes [7]. Evaluating such BO relationships and deriving useful insights from them to remodularise software systems falls under the category of *semantic structural relationships* analysis. Such semantic relationships are highlighted by Pẽrez-Castillo *et al.*'s experiments [15], 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 [16], in which SAP ERP logs were used to demonstrate process discovery based on BOs. Also, our own previous research on microservice discovery based on BO relationship evaluation [17,18] showed the impact of considering semantic structural relationships in software remodularisation.

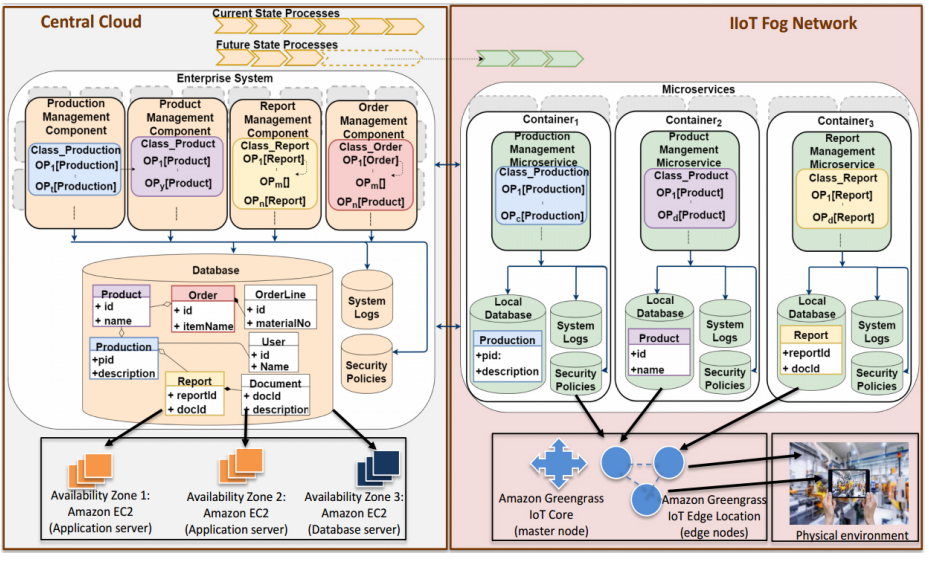


Fig. 1. Overview of an enterprise system extended with extracted microservices.

However, to date, techniques related to semantic structural relationships have not been integrated with static syntactic techniques at the method level. As a result, currently proposed design recommendation tools provide insufficient insights for software remodularisation targeting IIoT applications.

2.2 Architecture for Enterprise System to Microservice Remodularisation

In this section, we define the importance of considering the different factors detailed in Section 2.1 with respect to the architectural configuration of an enterprise system and related microservices in an IIoT network, underpinned by "fog" nodes in which much of the computation is done on "edge" devices. In order to provide a clear understanding of the structural complexity and behavioural implications of combining an enterprise system with an IIoT network, consider Figure 1, in which the current and future process states are depicted. Current-state processes, typically triggered by user actions, involve interactions through the methods of the enterprise system only. Future-state processes cover both a central enterprise system and its MSs deployed in the IIoT Network.

Figure 1 shows a central administration process for a construction/manufacturing scenario involving Production Management lists for Users (workers) which refer to Products being assembled and Reports for auditing and risk detection. It also includes Orders for faulty parts listed in Order Lines. In the future-state processes, some operations of the Production Management, Product Management and Report Management components are decoupled as microservices and embedded in a physical environment so that real-time and low-latency scheduling, checking, reporting and risk detection is enabled. This use case is inspired by Oswald *et al.*'s business analysis [19].

The internal structure of the enterprise system consists of a set of self-contained modules related to advance manufacturing drawn from different subsystems and is

deployed on a "backend". Each module is a combination of software classes that contain methods that manage one or more BOs through create, read, update and delete (CRUD) operations. These methods guide the system's execution through method calls between different classes in the same module or in different modules. For example, operation '$OP_1[Production]$' references an external method '$OP_1[Product]$' of the 'Product Management Module' through an object reference call and '$OP_1[Report]$' references an internal method in the 'Report Management Module' which calls '$OP_m[]$' in the same class. Execution of these methods generates system logs and the security of the modules and the system is governed through the security policies defined.

The microservices each support a subset of methods through classes that are specifically related to individual BOs, as depicted in Figure 1. This results in high cohesion within microservices and low coupling between the microservices. The microservices communicate with each other and with the enterprise system through API calls. Execution of methods across the enterprise system and microservices is coordinated through business processes, which means that invocations of methods in the enterprise system will trigger methods on microservices by passing parameters required by the microservices' APIs. Data consistency of different microservice databases and the enterprise system's database is achieved via regular synchronisation.

Based on this understanding of the structure of the enterprise system and its microservices, it is apparent why we must consider both semantic and syntactic information for our microservice discovery process. To capture the method call relationships in the enterprise system, we need *structural interaction relationship* analysis methods. This analysis helps to group methods that are highly coupled into one group, such as the grouping of '$OP_1[Report]$' and '$OP_m[]$' operations in the 'Report Management Module'. However, those relationships alone would not help to capture method similarities at the code level. To capture such similarities, we have to use the structural method similarity analysis techniques based on information retrieval (IR) techniques.

With *structural method relationships* and *structural method similarity* we can cluster methods into different modules. However, such modules might not align with the domain relationships until we consider the BO relationships of different methods. It is important to consider *semantic structural relationships* in the microservice derivation process, since each microservice should contain methods that are related to each other and should perform method invocations on the same BO. Previous research has extensively used *structural relationships* in system remodularisation [10–12]. However, when it comes to microservice derivation, combining *semantic structural relationships* with *syntactic structural relationships* will allow deriving better method clusters for IIoT deployed microservices.

3 Clustering Recommendation for Microservice Discovery

In order to derive IIoT-based microservices while considering the factors defined in Section 2, we developed a five-step approach, as illustrated in Figure 2. In Step 1, we derive the BOs by evaluating the SQL queries in the source code structure and also the database schemas and data as described by Nooijen *et al.* [20]. In Step 2, we identify the semantic structural relationships by deriving the method and BO relationships. Steps 3

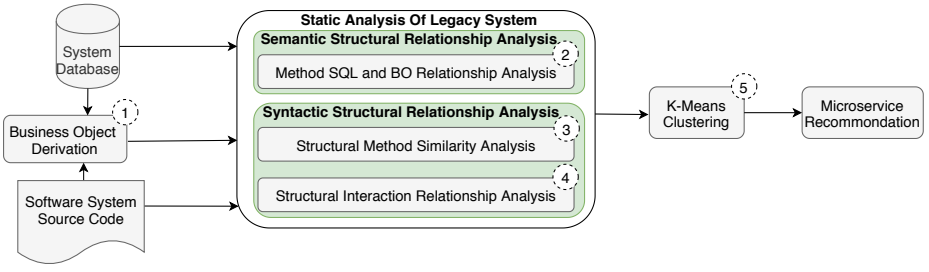


Fig. 2. Overview of our microservice discovery approach.

and 4 are used to discover the syntactic details related to the enterprise system. In Step 3, we measure the structural method similarities between methods in the same class and in different classes, and in Step 4 we capture the structural interaction relationships between different methods. The details obtained through Steps 2 to 4 are used in Step 5 where a K-means clustering algorithm is used to evaluate and recommend effective combinations of methods for IIoT-based microservice deployment. These steps are described further in Section 3.1.

3.1 Clustering Discovery Algorithms

As depicted in Figure 2, we supply a K-means algorithm with three main feature sets to derive a satisfactory clustering of system methods and suggest microservice designs. To derive these sets, we use Algorithm 1, which is composed of eight steps. We use the following formalisation here onwards to describe the algorithm.

Let $\mathbb{I}$, $\mathbb{O}$, $\mathbb{OP}$, $\mathbb{B}$, $\mathbb{T}$ and $\mathbb{A}$ be a universe of *input* types, *output* types, *operations*, *BOs*, *database tables* and *attributes* respectively. We characterise 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., $b \subseteq \mathbb{T}$. An *operation/method op*, either of an enterprise system or 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.[3] Each *class* $cls \in CLS$ is defined as a collection of operations/methods, i.e., $cls \subseteq \mathbb{OP}$.

The *BOS* function in Algorithm 1 is used to derive BOs B from enterprise systems as detailed by Nooijen *et al.* [20] (line 1). In the second step of the algorithm, function *CLSEXT* is used to extract code related to each class $cls \in CLS$ from the system code by searching through its folder and package structure (line 2). The extracted classes *CLS* are provided to the next step of the algorithm which uses the *MTDEXT* function to extracts the methods related to these classes (line 3). This step extracts the methods and the comments related to each method into separate text files and saves them for further processing.

In the fourth step, we rely on the information required for structural method similarity analysis using information retrieval (IR) techniques. As such, in the third step,

[3] A^* denotes application of the Kleene star operation to set A.

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*

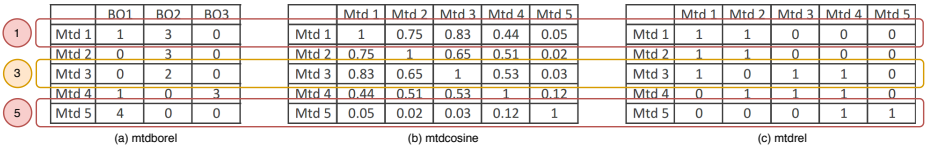
Output: Feature set data *borel*, *cosine*, *subtyperel*, *referencerel* and BOs *B*

1 $B = \{b_1, \ldots, b_n\} := BOS(SC, DB)$
2 $CLS = \{cls_1, \ldots, cls_m\} := CLSEXT(SC)$
3 $MTD = \{mtd_1, \ldots, mtd_m\} := MTDEXT(CLS)$
4 $UW = \langle uw_1, \ldots, uw_z \rangle := UWORDEXT(MTD, STW)$
5 **for each** $mtd_i \in MTD$ **do**
6 **for each** $b_k \in B$ **do**
7 $mtdborel[i][k] := BCOUNT(mtd_i, b_k)$;
8 **end**
9 **for each** $uw_s \in UW$ **do**
10 $mtduwcount[i][s] := WCOUNT(uw_s, mtd_i)$;
11 **end**
12 **end**
13 **for each** $mtd_i, mtd_k \in MTD$ **do**
14 $mtdcosine[i][k] := MTDCOSINECAL(mtduwcount[i], mtduwcount[k])$;
15 **end**
16 $mtdrel := MTDRELCAL(MTD)$;
17 **return** *CLS*, *MTD*, *mtdborel*, *mtdcosine*, *mtdrel*, *B*

	Product	Order	Inventory	Purchase	Shipment	Customer	Budget	Tax
Method 1	0	3	0	5	0	0	1	0
Method 2	3	0	0	7	0	0	0	0
Method 3	0	0	0	0	6	0	0	0
Method 4	5	7	0	0	0	0	0	0
Method 5	0	0	6	0	0	7	0	0
Method 6	0	0	0	0	7	0	0	0
Method 7	1	0	0	0	0	0	0	0
Method 8	0	0	0	0	0	0	0	0

Fig. 3. Example word matrix extracted from program code (*mtduwcount* in Algorithm 1).

the algorithm identifies unique words *UW* related to all the methods using function *UWORDEXT* (line 3), which requires all the source codes of the methods *MTD*, and stop words *STW*, which should be filtered out from the methods. In general, IR techniques analyse documents and filter out the contents that do not provide any valuable information for document analysis, referred to as 'stop words'. In our case, the stop words (*STW*) contain syntax related to the methods, standard technical terms used in coding in that particular programming language (in our case PHP) and common English words that would not provide any specific insight into a method's purpose. These are specified by the user based on the system's programming language. Function *UWORDEXT* first filters out the stop words *STW* from the methods *MTD* and then identifies the collection of unique words *UW* in methods *MTD* as a 'bag of words' [22]. This produces a collection of non-repeating words as depicted by the column names in the example in Figure 3.



	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

(a) mtdborel

	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

(b) mtdcosine

	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

(c) mtdrel

Fig. 4. Examples of matrices derived from code using Algorithm 1 (*mtdborel*, *mtdcosine* and *mtdrel*).

In the fifth step, the algorithm evaluates each method *mtd* ∈ *MTD* extracted in the third step and identifies the BOs which are related to each method. For this purpose, the algorithm uses function *BCOUNT* which processes the SQL statements, comments and method names and counts the number of times tables relating to BOs appear in the methods. This information is stored in matrix *mtdborel* (lines 5–8). In this matrix, each row represents a method, and each column represents the number of relationships that method has with the corresponding BO, as depicted in Figure 4(a). This helps capture the semantic structural relationships (i.e., BO relationships), which provides an idea about the "boundedness" of methods to BOs. For example, Method 1 ('Mtd 1') is related to 'BO1' and 'BO2' in Figure 4(a).

In the sixth step, the algorithm derives another matrix *mtduwcount*, which keeps a count of unique words related to each method using function *WCOUNT* (lines 9–11). Figure 3 provides an overview idea of a possible matrix that can be generated for *mtduwcount*. Again, in this matrix, rows correspond to methods, and columns correspond to unique words identified in step four of the algorithm that appear in the corresponding methods. The values in *mtduwcount* are then used in the seventh step to calculate the cosine similarity between the methods using function *COSINECAL* (lines 12–14).

Next, the algorithm's eighth step extracts the structural interaction relationships (i.e., method call relationships)using function *MTDRELCAL* (line 16). In this function, the code is first evaluated using the Mondrian code analysis tool[4], which generates graphs based on method call relationships as depicted in Figure 5. In Figure 5 the red circle shows the class, the grey squares show the methods in different classes and the arrow between them shows the method call relationships. Then the graphs are analysed to create matrix *mtdrel* which summarises the method call relationships for further processing (see the example in Figure 4(c)).

The feature set data in variables *mtdborel*, *mtdcosine*, *mtdrel* and the BOs *B* obtained from Algorithm 1 are provided as input to the K-Means algorithm to cluster the methods related to BOs based on their syntactic and semantic relationships. We followed a similar approach in our previous work [14], in which we adapted class-level relationships for microservice cluster discovery. However, here we have moved to the next level of system analysis by evaluating method level relationships. As such, in Algorithm 1, each dataset captures different aspects of relationships between the methods in the given system (Figure 4). Finally, as per our earlier work [14], we configured the K-Means algorithm to produce a set of clusters that group the methods of the analysed enterprise system as recommendations for constructing microservices.

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

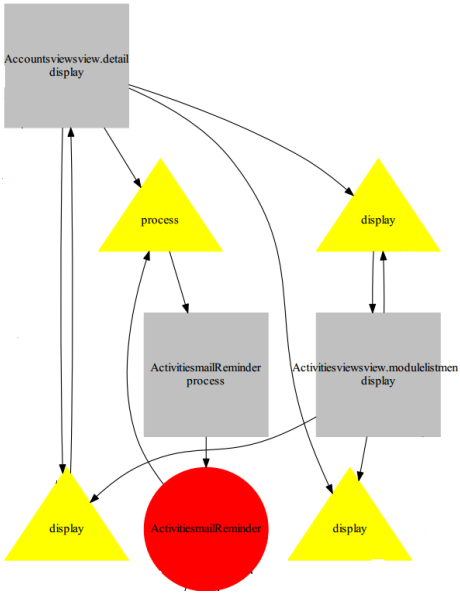


Fig. 5. Mondrian method call graphs

4 Implementation and Validation

To demonstrate our approach's applicability we developed a prototype microservice recommendation system[5] capable of discovering coherent method clusters related to different BOs, which lead to different microservice configurations. The system was tested using the Dolibarr open-source enterprise management system. Dolibarr consists of about 11,000 files and out of them around 1850 classes are related to its core functionality. Dolibarr's database uses MySQL and consists of 250 tables containing around 660 attributes.

Using our implementation, we performed the static analysis of Dolibarr's source code to identify the BOs it manages. As a result, 39 BOs were identified, e.g., Product, Order, Shipment, etc. Then, we performed the static analysis of the system to derive matrices, similar to those depicted in Figure 4, summarising the BO relationships, method similarity relationships and, method call relationships. All the results obtained were processed by our prototype software to identify method clusters and recommend microservices. The prototype identified 39 method clusters related to the BOs in Dolibarr, such that each cluster groups methods for developing a microservice that relates to a single BO.

4.1 Experimental Setup

In order to evaluate the effectiveness of the microservices suggested by our prototype for potential IIoT deployment, we compared the performance of the enterprise system with

[5] https://github.com/AnuruddhaDeAlwis/KMeans.git

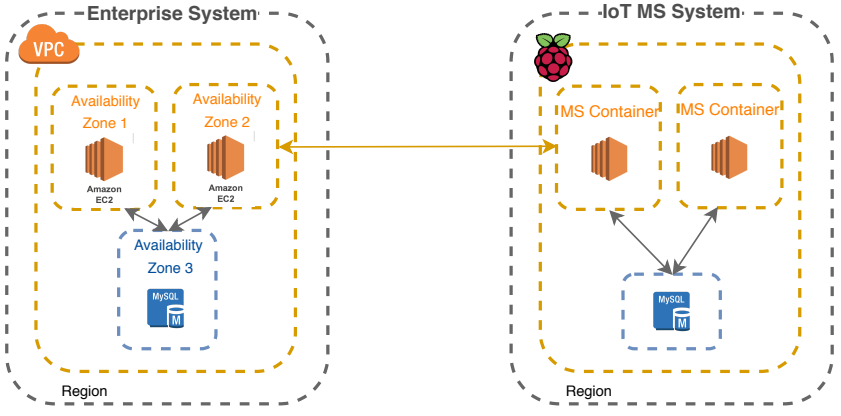


Fig. 6. System implementation using Amazon Web Services and Raspberry Pis.

and without microservice extensions. Each enterprise system was hosted in an Amazon Web Services cloud by creating two EC2 instances having two virtual CPUs and a total memory of 2GB, as depicted on the left side of Figure 6. Amazon Greengrass nodes were then used to simulate IIoT nodes running on Raspberry Pis as shown on the right. The systems' data were stored in a MySQL relational database instance which has one virtual CPU and total storage of 20GB.

These systems were tested against 200 and 400 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 (see the first two rows in Table 1). During the executions we tested the functionality related to operation 'order product'. The simulations were conducted using Selenuim[6] scripts which ran the system in a way similar to a real user.

Next, we introduced the 'purchase order' microservice from the Dolibarr system. As depicted on the right side of Figure 6, we hosted each microservice on an Amazon Greengrass node run on a Raspberry Pi 4, each containing its own local MySQL database. The tests were also performed on the original enterprise system, again simulating ordering a product. 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 results. Again, we recorded the total execution time, average CPU consumption, and average network bandwidth consumption for the entire system, i.e., enterprise system and microservice as a whole (see rows 3 and 4 in Table 1).

The scalability, availability and execution efficiency of the systems were calculated based on the measured results. The obtained results are summarised in Table 2 as *ES with MSs (1)* (second row in Table 2). Scalability was calculated according to the resource usage over time, as described by Tsai *et al.* [23]. 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

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

Table 1. Legacy vs microservice system results for Dolibarr.

System Type	No of Requests	Ex. Time (ms)	Avg CPU EC2	Avg CPU DB
ES only	200	822000	9.54	2.37
ES only	400	1740000	8.81	2.13
ES with MSs (as recommended)	200	816000	5.05	1.67
ES with MSs (as recommended)	400	1728000	5.23	1.55
ES with MSs (when 'disrupted')	200	819000	8.88	1.88
ES with MSs (when 'disrupted')	400	1734000	8.05	2.00

Table 2. Legacy vs microservice system EC2 characteristics comparison for Dolibarr.

System Type	Scalability [CPU]	Scalability [DB CPU]	Availability [200]	Availability [400]	Efficiency [200]	Efficiency [400]
ES only	3.458	3.365	99.27	99.31	1.000	1.000
ES with MSs (1)	3.398	3.045	99.27	99.31	1.007	1.007
ES with MSs (2)	3.427	4.031	99.27	99.31	1.003	1.003

described by Bauer *et al.* [24]. 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 led to the calculation of efficiency gain.

Next, we tested the quality of our system's microservice recommendations by disrupting its suggestions and developed a 'purchase order' microservice, while introducing operations related to the 'user' microservice, also running on an Amazon Greengrass deployment. Again, with this change, we set up the experiment as described earlier and measured the results (rows 5 and 6 in Table 1). Then we calculated the scalability, availability and execution efficiencies of the systems, summarised in Table 2 as *ES with MSs (2)* (third row in Table 2).

Based on these obtained experimental results we evaluated the effectiveness of our approach in two aspects. Firstly, we evaluated the performance differences between the microservice system and the original enterprise system. Secondly, we evaluated the performance differences between the microservices suggested by our prototype and other microservice designs. These comparisons are detailed below.

Recommended microservices vs original enterprise system. As per Tsai *et al.*'s metric [23], the lower the measured number, the better the scalability. Thus, it is evident that the microservice systems derived based on our clustering algorithm managed to achieve 0.7% improved system execution efficiency and 1.74% scalability improvement (considering CPU scalability), see Table 1. As such, our recommendation system discovers microservices that can achieve improved cloud capabilities such as high scalability, high availability and high execution efficiency. Notably, the integrated ES with MSs system achieved 59% (5.23/8.81) and 72% (1.55/2.13) CPU utilisation at EC2 instances and DB as compared to the original ES.

Recommended microservices vs other microservices. Microservices developed based on the suggestions provided by our recommendation system for Dolibarr managed to achieve: (i) 1.74% calability improvement in EC2 instance CPU utilisation; (ii) 9.51% scalability improvement in database instance CPU utilisation; and (iii) a 0.7% improve-

ment in execution efficiency. However, the "disrupted" microservices that violated the recommendations reduced (i) EC2 instance CPU utilisation to 0.89%; (ii) database instance CPU utilisation to (-)19.79%; and (iii) execution efficiency to 0.3%. As such, it is evident that the microservices developed by following the recommendations of our system provided better cloud characteristics than the microservices developed against these recommendations.

4.2 Limitations

Although this paper presents an algorithm that resolves some of the challenges in discovering IIoT microserviceable components from enterprise systems, there remain several limitations that should be addressed in future research.

Limitation of BO derivation: To derive the BOs related to the given enterprise systems, we used Nooijen *et al.*'s method [20]. However, as Lu *et al.* explain [16], such methods cannot always derive BOs accurately without some domain knowledge from from the system's developers. We tried to avoid errors by manually evaluating the results obtained for the BOs by referring to the system's manuals and documentation. Still, such an approach remains complex and error prone.

Limitation of structural method similarity analysis: The structural method similarity analysis obtained a 'bag of words' term frequency and, finally, calculated the cosine similarity between the documents. The first limitation of this method is the potential filtering out of valuable information in the data preprocessing stage. We mitigated this by manually evaluating the stop words used in the text preprocessing step. In addition, the cosine values might not provide an accurate idea about structural method similarity since it may also depend on the terms used in the definitions of the method names and descriptions given in the comments. We mitigated this to a certain extent by evaluating the code structure of the software systems and verifying that the method names and comments provide valuable insights into the logic behind the methods that implement the system, but again it is easy to make mistakes during such a manual process.

5 Discussion

This paper we showed how to identify the components in enterprise systems that can be developed as IIoT deployable microservices. However, through the introduction of microservices, new behaviours can arise in relation to current state enterprise systems, given increased flexibility of execution, resulting from asynchronous and branching actions and new extension points introduced by microservice architectures. In order to evaluate the behavioural changes caused by the introduction of IIoT components to enterprise systems, testing should be conducted using methods such as conformance checking.

Similarly, distributing enterprise systems in "fog" networks, where significant parts of the computation occur on edge devices, opens up significant security vulnerabilities. Under a central system, the users' and systems' interactions are subject to local access control, constraining data access via permissions and security modes. However, the distributed architecture of IIoT and fog computing poses new threats to authentication

and trust, secure communications, and end-user privacy [25]. In particular, a fog network makes it difficult to authenticate the identity of nodes as they enter and leave the network [26], is vulnerable to data breaches caused by malicious or malfunctioning nodes, risks end-user privacy due to the large amount of user-specific data generated by nodes, and inhibits anomaly detection due to the difficulty of monitoring large numbers of nodes [27]. Developing new security technologies and verification methods for IIoT applications would be another interesting future research area.

6 Conclusion

Here we presented a novel technique for automated analysis and remodularisation of enterprise systems as IIoT deployable microservices by combining techniques that consider semantic knowledge and syntactic knowledge about the system's code. A prototype recommendation system was developed and validated by implementing the microservices recommended by the prototype for Dolibarr which is an open-source ERP system. The experiment showed that our approach could derive method clusters that produced IIoT deployable microservices with desired Cloud characteristics, such as high scalability, high availability, and processing efficiency.

References

1. Dahlqvist, F., Patel, M., Rajko, A. and Shulman, J., 2019. Growing Opportunities in the Internet of Things. https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things
2. Columbus, L., 2018. 2018 Roundup Of Internet Of Things Forecasts And Market Estimates. https://www.forbes.com/sites/louiscolumbus/2018/12/13/2018-roundup-of-internet-of-things-forecasts-and-market-estimates/#1980d3f57d83
3. Newman, S., 2015. Building Microservices. O'Reilly Media, Inc.
4. Mauro, T., 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
5. Barros, A., Duddy, K., Lawley, M., Milosevic, Z., Raymond, K. and Wood, A., 2000, October. Processes, roles, and events: UML concepts for enterprise architecture. In International Conference on the Unified Modeling Language (pp. 62–77). Springer, Berlin, Heidelberg.
6. Schneider, T., 2012. SAP Business ByDesign Studio: Application Development (pp. 24–28). Boston: Galileo Press.
7. Decker, G., Barros, A., Kraft, F.M. and Lohmann, N., 2008, December. Non-desynchronizable service choreographies. In International Conference on Service-Oriented Computing (pp. 331–346). Springer, Berlin, Heidelberg.
8. Barros, A., Decker, G. and Dumas, M., 2007, May. Multi-staged and multi-viewpoint service choreography modelling. In Proceedings of the Workshop on Software Engineering Methods for Service Oriented Architecture (SEMSOA), Hannover, Germany. CEUR Workshop Proceedings (Vol. 244).
9. Barros, A., Decker, G., Dumas, M. and Weber, F., 2007, March. Correlation patterns in service-oriented architectures. In International Conference on Fundamental Approaches to Software Engineering (pp. 245–259). Springer, Berlin, Heidelberg.
10. Praditwong, K., Harman, M. and Yao, X., 2010. Software module clustering as a multi-objective search problem. IEEE Transactions on Software Engineering, 37(2), pp. 264–282.

11. Mitchell, B. S. and Mancoridis, S., 2006. On the automatic modularization of software systems using the bunch tool. IEEE Transactions on Software Engineering, 32(3), pp. 193–208.
12. Poshyvanyk, D. and Marcus, A., 2006, September. The conceptual coupling metrics for object-oriented systems. In 22nd IEEE International Conference on Software Maintenance (pp. 469–478). IEEE.
13. Candela, I., Bavota, G., Russo, B. and Oliveto, R., 2016. Using cohesion and coupling for software remodularization: Is it enough?. ACM Transactions on Software Engineering and Methodology (TOSEM), 25(3), p. 24.
14. De Alwis, A. A. C., Barros, A., Fidge, C. and Polyvyanyy, A., 2020. Remodularization Analysis for Microservice Discovery Using Syntactic and Semantic Clustering. In Proceedings of the 32nd International Conference on Advanced Information Systems Engineering (CAiSE 2020). Vol. 12127 of Lecture Notes in Computer Science. Springer-Verlag.
15. Pérez-Castillo, R., García-Rodríguez de Guzmán, I., Caballero, I. and Piattini, M., 2013. Software modernization by recovering web services from legacy databases. Journal of Software: Evolution and Process, 25(5), pp. 507–533.
16. Lu, X., Nagelkerke, M., van de Wiel, D. and Fahland, D., 2015. Discovering interacting artifacts from ERP systems. IEEE Transactions on Services Computing, 8(6), pp. 861–873.
17. De Alwis, A. A. C., Barros, A., Fidge, C. and Polyvyanyy, A., 2019. Business Object Centric Microservices Patterns. In Proceedings of the 27th International Conference on Cooperative Information Systems (CoopIS 2019). Vol. 11877 of Lecture Notes in Computer Science, pp. 476–495. Springer-Verlag.
18. De Alwis, A. A. C., Barros, A., Polyvyanyy, A. and Fidge, C., 2018. Function-splitting heuristics for discovery of microservices in enterprise systems. In Proceedings of the 16th International Conference on Service Oriented Computing (ICSOC 2018). Vol. 11236 of Lecture Notes in Computer Science, pp. 37-–53. Springer.
19. Oswald, D., Zhang, R.P., Lingard, H., Pirzadeh, P. and Le, T., 2018. The use and abuse of safety indicators in construction. Engineering, construction and architectural management.
20. Nooijen, E. H., van Dongen, B. F. and Fahland, D., 2012, September. Automatic discovery of data-centric and artifact-centric processes. In International Conference on Business Process Management (pp. 316–327). Springer, Berlin, Heidelberg.
21. Zhang, H., Xiao, Y., Bu, S., Niyato, D., Yu, F.R. and Han, Z., 2017. Computing resource allocation in three-tier IoT fog networks: A joint optimization approach combining Stackelberg game and matching. IEEE Internet of Things Journal, 4(5), pp.1204-1215.
22. Lebanon, G., Mao, Y. and Dillon, J., 2007. The locally weighted bag of words framework for document representation. Journal of Machine Learning Research, 8(Oct), pp. 2405–2441.
23. Tsai, W.T., Huang, Y. and Shao, Q., 2011, December. Testing the scalability of SaaS applications. In 2011 IEEE International Conference on Service-Oriented Computing and Applications (SOCA) (pp. 1–4). IEEE.
24. Bauer, E. and Adams, R., 2012. Reliability and availability of cloud computing. John Wiley & Sons.
25. Mukherjee, M., et al., 2017. Security and Privacy in Fog Computing: Challenges. IEEE Access, Vol. 5, pp. 19293–19304.
26. Zhang, Z.-K., et al., 2014. IoT Security: Ongoing Challenges and Research Opportunities. Proceedings of the 7th International Conference on Service-Oriented Computing and Applications, pp. 230-234, IEEE 2014.
27. Khan, S., Parkinson, S., and Qin, Y., 2017. Fog Computing Security: A Review of Current Applications and Security Solutions. Journal of Cloud Computing: Advances, Systems and Applications, 6(19).
28. Woodhead, R., Stephenson, P., and Morrey, D., 2018. Digital construction: From point solutions to an IoT ecosystem. Journal of Automation in Construction, 93(35–46).