Identification of Microservices from Monolithic Applications through Topic Modelling

Miguel Brito
pg38419@alunos.uminho.pt
University of Minho
Portugal

Jácome Cunha
jacome@di.uminho.pt
University of Minho &
HASLab/INESC Tec
Portugal

João Saraiva
saraiva@di.uminho.pt
University of Minho &
HASLab/INESC Tec
Portugal

ABSTRACT

Microservices emerged as one of the most popular architectural patterns in the recent years given the increased need to scale, grow and flexibilize software projects accompanied by the growth in cloud computing and DevOps. Many software applications are being submitted to a process of migration from its monolithic architecture to a more modular, scalable and flexible architecture of microservices. This process is slow and, depending on the project's complexity, it may take months or even years to complete.

This paper proposes a new approach on microservice identification by resorting to topic modelling in order to identify services according to domain terms. This approach in combination with clustering techniques produces a set of services based on the original software. The proposed methodology is implemented as an open-source tool for exploration of monolithic architectures and identification of microservices. A quantitative analysis using the state of the art metrics on independence of functionality and modularity of services was conducted on 200 open-source projects collected from GitHub. Cohesion at message and domain level metrics' showed medians of roughly 0.6. Interfaces per service exhibited a median of 1.5 with a compact interquartile range. Structural and conceptual modularity revealed medians of 0.2 and 0.4 respectively.

Our first results are positive demonstrating beneficial identification of services due to overall metrics' results.

ACM Reference Format:
Miguel Brito, Jácome Cunha, and João Saraiva. 2021. Identification of Microservices from Monolithic Applications through Topic Modelling. In *Proceedings of SAC 2021.* ACM, New York, NY, USA, 11 pages. https://doi.org/DOI

1 INTRODUCTION

The decomposition of systems into modules began to be systematised and debated by Parnas [31] long before the massification of software systems. Parnas intended to demonstrate that the efficiency of the modularisation of a system depends on the criteria used, contrary to the pure fragmentation of systems into small modules.

The relevance of the functional decomposition of systems initially mentioned by Parnas was reinforced by the demand to distribute complex systems through network infrastructures such as web services and remote objects resulting from efforts to deal with systems of greater dimension and complexity [19].

From that demand microservice-based architectures emerged, consisting of small services that focus on one particular functionality [29]. The main idea of such microservice architectures (MA) is that they have the potential to increase the flexibility and agility of software development [29].

Microservices have/are been quickly adopted to developed new software. There are, however, many legacy software systems that were developed before MA was introduced, but that can benefit from the agility MA software development offers. This is particularly relevant when we consider the usual maintenance and evolution processes required in a modern software lifecycle. In order to benefit from MA such legacy software systems - that we call monolithic software systems - need to be refactored into a semantically equivalent microservice-based one. Performing such refactoring manually is both complex/time consuming and prone to errors, and its quality is often strongly linked to the experience and knowledge of the specialist leading said refactoring [19, 30]. The refactoring is typically done following a Strangler pattern, that is, incrementally migrating and replacing modules of the system into a new architecture until the migrated systems overcome the old system. As a consequence, we need an automated process to transform a monolithic system into a MA one.

The identification of microservices in legacy monolithic systems is still an open problem with just a few proposed approaches [8, 15, 17–19, 26]. Most of these proposals, however, use their own quality metrics to assess the quality of the achieved transformation. Moreover, they are not supported by a tool that can be automatically applied to a legacy system, and as consequence the approaches are *validated* in a small (less than ten) number of monolithic systems. The exception is Jin et al. [17], which uses a dynamic analysis approach: it runs the legacy software system, to infer the microservices so to migrate it into a MA one. Although it provides good results, it has the disadvantage of requiring inputs or test cases to properly execute the system. Unfortunately, this is not the case in most legacy systems [32].

In this paper we propose a static analysis technique to identify microservices in a legacy software system based on topic models. Topic models [20] allow to mine a set of topics across a collection of

SAC 2021,
,

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/DOI

documents. Thus, by applying topic modelling to a monolithic software system we identify the systems' topics, which correspond to domain terms, and represent the microservices implemented by that legacy systems. Such topic models are inferred from the collection of lexical information in the source code, namely method declarations, method invocations, variables, method and class names, etc. For instance, considering a software for managing stocks, one would expect to find terms related to products, suppliers, etc. The collection of components with related names are likely part of the same microservice. We explore in detail this idea in this paper. The mined topics can then be combined into the structural information of the source code into a graph. Such graph is then clustered in order to identify microservices.

To assess the quality of the identified microservices we use the MA metrics proposed in the work of Jin et al. [17] that evaluate the independence of functionality and modularity of microservices. Furthermore, the proposed methodology is implemented as an open-source tool[1] for exploration of monolithic applications. This tool was validated by performing a quantitative analysis study on 200 open-source monolithic software systems collected from GitHub. The results obtained concerning MA metrics' are positive. Regarding independence of functionality, CHM and CHD presented a median of roughly 0.6; IFN presented a median of 1.5. Modularity metrics of SMQ and CMQ demonstrated median values of 0.2 and 0.4 respectively. Overall the results are positive, showing relevant proposals of microservices and a promising first step to refactor monolithic applications.

The remainder of the paper is structured as follows: Section 2 presents the methodology we devised for microservice identification; Section 3 discusses a case study and a walkthrough of the methodology applied to an example project; Section 4 describes the steps taken to quantitatively analyse our methodology and concludes with our results; Section 5 describes the current state of the art and Section 6 concludes our paper.

2 PROPOSED METHODOLOGY

In this section, we describe the proposed methodology to identify microservices. Figure 1 illustrates an overview of the steps composing the identification process.

First, we extract lexical and structural information from the source code of the monolithic system being migrated to MA.

Next, we use the extracted information to fit a topic modelling technique allowing to identify topics and their distributions for each component of the software project.

Finally, the topic distribution and the structural information are combined and fed into a clustering algorithm identifying microservice proposals.

The next three subsections describe these steps in detail.

Note the process we describe is generic and not specific for a particular language or paradigm. However, since our tool is instantiated for the Java language, and in particular for the Spring Framework [2], the examples presented are written in this framework.

2.1 Information Extraction

The building blocks of our microservice identification methodology are the lexical and structural dependency information occurring in the source code of the monolithic software system under analysis. Next, we describe how such information is computed.

Lexical Extraction. Lexical extraction is defined as the extraction of all the lexical/textual terms from source code that are relevant to identify what a given component represents in the context of the domain of the project.

We perform this extraction in a structured version of the source code: its underlying Abstract Syntax Tree (AST). The main reason to extract textual terms from the AST instead of its textual representation (and applying filters, stop words and other kinds of preprocessing) is to have better control of the information being extracted. This greatly reduces the amount of analysis needed to identify the most relevant terms. The entire AST is built and types are matched against their declaration. Each type that does not belong to the project is ignored and not considered as relevant information. With a pure Natural Language Processing (NLP) approach applied to the source code, filtering keywords from the language would be simple. However with the addition of external abstraction (in the case of languages supporting it) and external libraries that introduce terms that could be completely unrelated to the domain of the problem, a pure NLP approach would produce worst results.

To handle the presented problem, only terms (such as variable types) referenced to the project are taken into account. Without losing generality, let us consider the following line of code written in Java (Spring Framework):

```
return new ResponseEntity<>(
            user, responseHeaders, HttpStatus.OK);
```

A project parsing approach gives us the possibility to only extract a certain parameter, for instance `user`, much more related to the domain than the whole expression that is mostly composed by Spring Framework terms and abstractions. For instance, given that `ResponseEntity` is so popular in projects based on the Spring Framework, an NLP approach could calculate that the addition of it to a list of stop words would solve the problem, however, that would only work for the common terms or require significant manual work when dozens of libraries are used in a project. Any addition of libraries that represent a strong connection to the domain, contrary to abstraction and helpers libraries, can still be included by a whitelist of types. Overall, our approach works by trying to filter out abstractions related to external libraries and frameworks and focus on the terms more related to the domain of the project. This is done by extracting textual terms from components' names, variable declarations, unctions declarations and their parameters, and method/functions invocations.

Structural Dependency Extraction. Most of the dependencies are straightforward to obtain working on the AST. However, other tasks such as the identification of types of expressions or finding the usage of a symbol are not so simple as they involve significant work over the AST. For those, and considering the particular case of the Java language, we used *JavaParser Symbol Solver* [1], which allows to identify the types of a given expression. Each dependency

[1] https://anonymous.4open.science/r/61cd83e3-47b7-405d-b5ec-8ac90c81d6c7

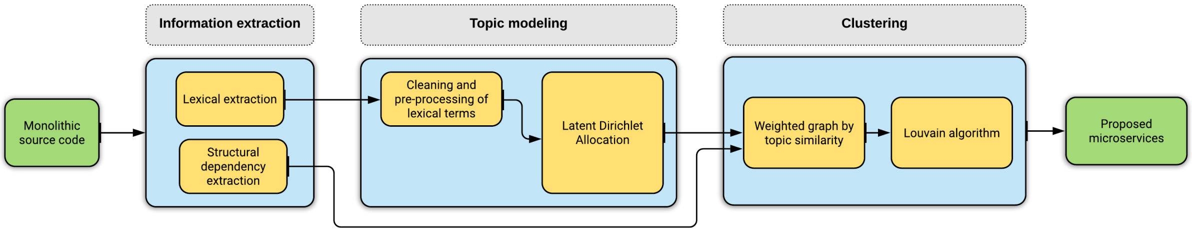
Information extraction
Topic modeling
Clustering
Monolithic source code
Lexical extraction
Structural dependency extraction
Cleaning and pre-processing of lexical terms
Latent Dirichlet Allocation
Weighted graph by topic similarity
Louvain algorithm
Proposed microservices

Figure 1: Overview of the architecture of the proposed method and developed tool

between classes is identified by finding method invocations and resolving the class to which the method belongs, and consequently its type.

The use of structural dependencies enables a better representation of the software architecture at hand, as it fits nicely into a graph representation and presents interaction between modules.

2.2 Topic modelling

Topic modelling techniques allow to identify latent semantic structures from a set of documents, similarly to how a developer would analyse a software project and identify domain terms to grasp how it can be decomposed on a semantic level.

The quality of topics proposed by these techniques is highly dependent on the quality of the inputs fed into them. Accordingly, textual terms t are pre-processed going through tokenization, stop word removal and stemming in order to remove terms without much significance as domain terms and reduce variations of the same root words. Extremes of very common and rare terms present in the corpus are filtered and then collected as a bag of words.

Having computed the lexical and structural information, we can now use a topic modelling classifier to group such information in clusters, which will then form the microservices identified in the legacy system.

In an exhaustive and comprehensive state of the art review on topic modelling done by Kherwa and Bansal [20], four major groups of topic modelling classifiers are identified: Probabilistic Latent Semantic Analysis (PLSA), Latent Dirichlet Allocation (LDA), Latent Semantic Analysis (LSA) and Non-negative Matrix Factorisation (NMF). This work also presents a detailed quantitative analysis comparing LDA *versus* LSA concluding with the superiority of LDA: it yields higher coherence values across topics and less overlap between topics. In an exploratory work done by Stevens et al. [36] comparing NMF, LDA and LSA and analysing its weaknesses and strengths, they conclude in favour of LDA due to its flexibility and coherence advantages over others. Sun et al. [37] propose a technique for clustering classes in packages in order to increase program comprehension and reducing large packages resorting to LDA, PLSA and Latent Semantic Indexing (LSI) as a base of clustering methods. From the case studies conducted, LDA resulted in better clustering results and the topics identified were more useful for comprehension by developers.

Overall, LDA is widely used and the most popular on the topic modelling field, being the core of evolution and extension to other models, such as Dynamic topic model, Author topic model, Multilingual Topic Model [20, 38]. Thus, we also adopt the LDA classifier in our microservice identification approach, describing it next.

2.2.1 *Latent Dirichlet Allocation.*

LDA categorises documents by topics via a generative probabilistic model [4]. It treats each document as a random mixture of latent topics, and each topic as a distribution of words of the corpus. The words with higher probabilities that represent a topic usually give a good overview of what the topic is describing and talking about, hence allowing to discover a set of concepts representing the entire corpus [16]. LDA is an unsupervised model requiring only the corpus of the documents without any extra metadata. LDA does not consider the order of the words in the documents or their semantic importance being only fed with a bag of words (BOW) – simplified representation of a corpus containing count occurrence for each word. These features allow LDA to be scalable to thousands or millions of documents [37].

The components being used as the basis of work in the LDA model are formally described as follows:

(1) A word represents the basic unit extracted from the source code of a software project representing the textual terms denoted as $t = \{w_1, w_2, ..., w_n\}$.
(2) A document, identified as a component (*eg.* class in Java, module in C) in the context of a software project is a collection of words and described as $d = \{w_1, w_2, ..., w_n\}$.
(3) A corpus is a collection of documents identified as $c = \{d_1, d_2, ..., d_n\}$.

Choosing the number of K topics to be identified by the LDA model is a challenge by itself. Ideally, the number of topics should be selected after an analysis of the domain terms of the project at hand complemented with an analysis of inter-topic distance [35], in order to assess how well defined are the topics, its independence from each other and the amount of overlap. The distribution of terms per topic should also be taken into consideration to avoid topics from being composed of a large chunk of concepts across the domain, resulting in lower levels of topic coherence.

Figure 2 and Figure 3 illustrate inter-topic distance and term distribution per topic for a model of 8 and 11 topics respectively. By comparing both figures it can be concluded that the model shown in Figure 3 represents an excessive number of defined topics given the amount of overlap introduced by the increase in number of topics.

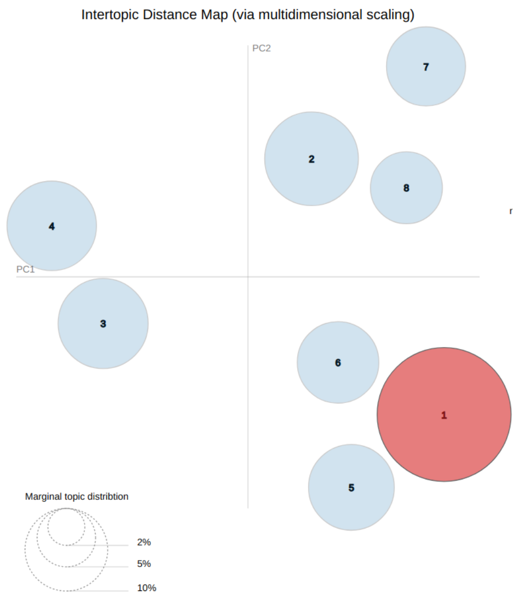

Figure 2: Intertopic distance with number of topics set to 8

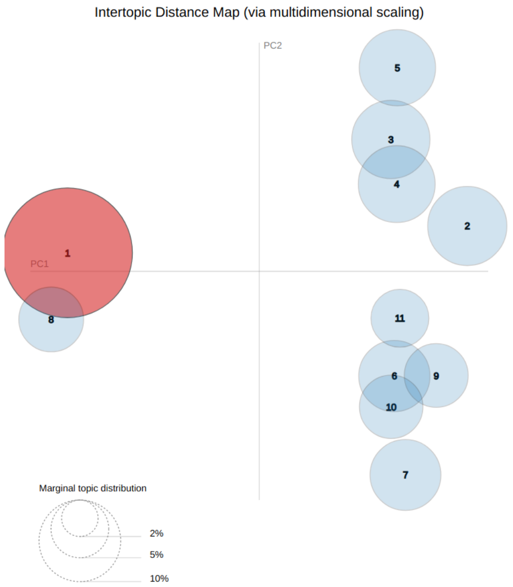

Figure 3: Intertopic distance with number of topics set to 11

An automated approach to identify an adequate amount of topics, yet computationally more expensive, can be simply done by creating models for a wide range of topics, measuring the coherence for each one and deciding upon the best topic. The main goal of measuring coherence of topics is to verify if a *"set of facts support each other"* [34] and refer to a specific domain of knowledge. Among the multiple metrics proposed and the extensive analysis of the state of the art on coherence measurements done by Röder et al. [34], the c_v metric (which results from the combination of previous metrics) is the one having the highest correlation to human ratings on topic coherence.

With multiple values of c_v measured for a range of number of topics, the best topic is selected by identifying the knee point.

2.3 Clustering

The fitting of the LDA model against K topics produces a distribution of topics across documents. That distribution can be used on its own to cluster components into groups of proposed services. However, that would take into account the domain aspect of the software, only, ignoring the structural relationship and dependencies between classes. Thus, we combine the previously extracted structural dependencies with the distribution of topics into an edge-weighted graph G. In the graph $G = (E, V)$, the vertices $v_i \in V$ correspond to a component $c_i \in C$ from the monolithic project. Each edge $e_i \in E$ is weighted by a weight function determining how strong is the relationship according to topic distribution. The higher the value the stronger the relationship between topics identified are. Each component c is then identified by a vector $\vec{v}$ to represent the distribution of probability across topics t, $\vec{v} = \langle t_1, t_2, ..., t_n \rangle$. The cosine similarity is then used to find how similar two vectors $\vec{v_1}, \vec{v_2}$ are, as follows:

$$\cos(\theta) = \frac{\vec{v1} \cdot \vec{v2}}{||\vec{v1}||||\vec{v2}||} = w(e_i) \in G \tag{1}$$

An illustration of a formal definition of graph G is presented in Figure 4.

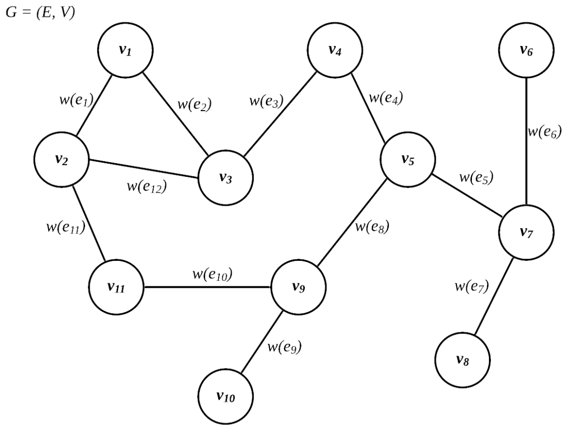

Figure 4: Weighted graph representation

We conducted an analysis of the state of the art of clustering/-community detection algorithms to decide how to cluster the graph into propositions of services. Rahiminejad et al. [33] performed a topological and functional comparison of community detection algorithms in biological networks. Six algorithms are analyzed: Combo, Conclude, Fast Greedy, Leading Eigen, Louvain and Spinglass. The main criteria of evaluation for those algorithms were: appropriate community size (neither too small nor too large), performance in terms of speed and two other features regarding gene and biological functions. From the conducted evaluation on two distinct data sets they conclude favouring Louvain given that the communities found were very similar to the top methods and Louvain was the fastest community detection method.

The Louvain algorithm presented by Blondel et al. [5] is an heuristic algorithm based on modularity maximization. Its main goal is to maximize network modularity. Modularity is a measure of strength of division of a network into clusters/communities. Higher modularity represents dense connections within nodes in a community but sparse connections between different communities [28]. It is an unsupervised algorithm not requiring the number of communities to be identified nor their sizes.

At its core the algorithm is divided into two main steps repeated iteratively [5, 27]:

- Step 1 - Each node in the network is assigned to its own community. The number of communities is equal to the number of nodes N; For each neighbour j of node i, it is tested if the modularity increases by moving it from community i to community j. If there is an increase, the node is moved, otherwise it stays in the original community. This step is repeated for all the nodes in a sequential order and repeated until no improvements in modularity can be achieved.
- Step 2 - The network is rebuilt by merging the nodes in the same community.

Steps 1 and 2 are executed iteratively until the merging of communities does not change and a maximum modularity is reached. The majority of the computational work is done on the first iterations. Step 2 exponentially reduces the amount of work as it gets closer to the final iterations.

A common problem identified as a limitation of the Louvain algorithm and other algorithms that use modularity as its core is that it may fail to identify modules smaller than a given scale [12]. That problem was observed initially on projects more complex and bigger in number of components (classes in our case – Java). In order to avoid that, the resolution parameter is manipulated allowing to discover clusters at different scales [23]. A higher resolution results in more iterations of the merging step of the algorithm, resulting in less but bigger clusters. Similarly a lower resolution results in less merging, meaning more clusters of smaller size. For each resolution the project is clustered and executed against metrics of independence of functionality and modularity presented in Section 4. From that metric execution the best resolution can be selected. Although a resolution is chosen and consequently its proposed services, we provide the user all the proposed services for other resolutions. With different granularities of proposed microservices the user can do a more informed and qualitative identification of what represents the best solution for the current project.

3 CASE STUDY

In this section we present a walkthrough of the proposed methodology on a Java Spring application named JPetStore[2]. JPetStore is a shopping application regarding pet selling. Our goal is to present the most relevant steps, namely the application of the topic modelling as a way to analyse the identified topics and words that compose the given topics. Since it is a relatively small project, steps concerning identification of K topics and Louvain resolution are not included. Skipping the initial information extraction and pre-processing of the textual terms the LDA model is inferred according to the pre-processed textual terms against a number of topics of *K=4*. In Table 1 the 10 top words in its stemmed version are presented for each topic:

- Topic 0 - The first five top words represent strong domain concepts referring to users/accounts.
- Topic 1 - Strong domain relation to cart management.
- Topic 2 - Strong domain relation to catalog: products, category, item
- Topic 3 - Strong domain relation to order execution: line, status, total, price, cart, stock.

Topic	*Top 10 words representing a topic*
0	account username password signon profile clear resolut status version serial
1	item cart line quantiti big total price stock increment inventori
2	product catalog categori item resolut serial clear profil total name
3	order sequenc line statu usernam total price version cart stock

Table 1: Top 10 stemmed words belonging to each topic

The topic distribution for each class of the project is presented in Table 2. The vast majority of the classes has a strong association to one of the topics (highlighted in bold face). The class `web.actions.AbstractActionBean` stands out as having a very similar distribution across topics because it represents an abstraction extended by the main entities exposing the application functionality as beans. Although there is still no way to deal with abstraction classes being split from the classes that directly require them (on the implemented tool), this kind of information may be useful in the future as a way to alert and guide the final user.

Finally, with the calculation of cosine similarity between the structural dependencies between classes resorting to topic distribution the clusters of classes as potential microservices are identified, as illustrated in Figure 5. Four clusters are identified as propositions of microservices (each cluster is coloured differently in the figure). In this example the clusters are very similar and are according to the topics identified and its distribution. For larger and more complex applications the clustering will have a more significant impact, given higher number of topics and its distribution and an increased number of dependencies between components.

In the context of the example, the identified clusters propose four different services related to: accounts/users, cart management, catalogue management and order execution.

4 EVALUATION

In order to quantitatively assess the quality of the microservices proposed by our approach, we collected 200 projects from *GitHub* and computed the MA metrics proposed by the work of Jin et al. [17] regarding independence of functionality and modularity. In this section we describe in detail this evaluation, starting by the the metrics.

4.1 Independence of functionality

Independence of functionality refers to external independence, meaning how independent and well-defined the services are. The

[2] Repository found at https://github.com/mybatis/jpetstore-6

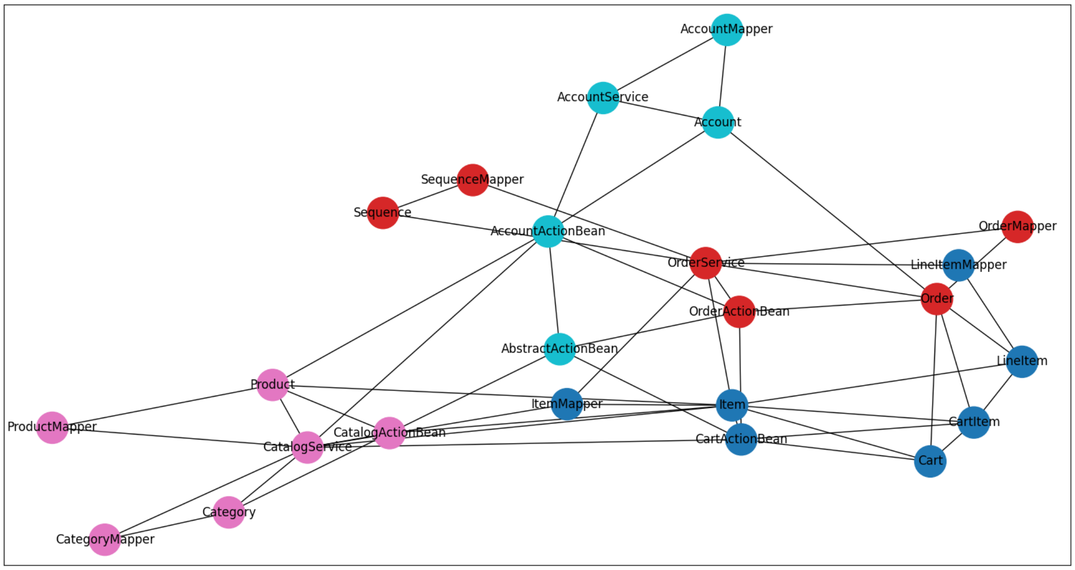

Figure 5: Clustering using topic distribution similarity as weight

following metrics are calculated resorting to the interfaces I of a service. An interface is any class that exposes functionality as an endpoint. For each interface, methods are considered as operations O.

The value **ifn** (*interface number*) quantifies the number of interfaces for a given service. It is based on the Single Responsibility principle. A smaller *ifn* represents a higher likelihood of any given service assuming a single responsibility [17]. *IFN* represents the average of all *ifn*.

$$IFN = \frac{1}{N}\sum_{i=1}^{N} ifn_i \tag{2}$$

$$ifn_i = |I_i| \tag{3}$$

The value **chm** (*cohesion at message level)* quantifies cohesiveness at message level of interfaces of a given service. A higher *chm* represents an higher cohesiveness of the service. *CHM* represents the average of all *chm*. Messages are composed by the terms of method parameters (*par*) and method returns (*ret*).

$$CHM = \frac{1}{N}\sum_{i=1}^{N} chm_i \tag{4}$$

$$chm_i = \begin{cases} \frac{\sum_{(k,m)} f_{msg}(opr_k,opr_m)}{\frac{1}{2}|O_i|\times(|O_i|-1)}, & \text{if } |O_i| \neq 1 \\ 1, & \text{if } |O_i| = 1 \end{cases} \tag{5}$$

Jaccard index is used as function of similarity both for similarity of return terms and parameter terms.

$$f_{msg}(opr_k, opr_m) = \frac{\left(\frac{|ret_k \cap ret_m|}{|ret_k \cup ret_m|}\right) + \left(\frac{|par_k \cap par_m|}{|par_k \cup par_m|}\right)}{2} \tag{6}$$

The value **chd** (*cohesion at domain level*) quantifies the cohesiveness at domain level of the interfaces of a given service. It is quantified very similarly to *chm* varying only on the function of similarity. Instead of using only message terms, all domain terms are considered.

$$CHD = \frac{1}{N}\sum_{i=1}^{N} chd_i \tag{7}$$

$$chd_i = \begin{cases} \frac{\sum_{(k,m)} f_{dom}(opr_k,opr_m)}{\frac{1}{2}|O_i|\times(|O_i|-1)}, & if|O_i| \neq 1 \\ 1, & if|O_i| = 1 \end{cases} \tag{8}$$

$$f_{dom}(opr_k, opr_m) = \frac{|f_{term}(opr_k) \cap f_{term}(opr_m)|}{|f_{term}(opr_k) \cup f_{term}(opr_m)|} \tag{9}$$

The value **IRN** (*interaction number*) quantifies the number of method calls across two different services. The smaller the *IRN* the better [18].

$$IRN = \sum_{(opr_j,opr_k)} w_{j,k} \tag{10}$$

4.2 Modularity

Modularity evaluates how cohesive are the services in its internal interactions and how loosely coupled are the interactions across services.

SMQ (*Structural Modularity Quality*) quantifies modularity from a structural viewpoint. Higher *SMQ* represents better modularized services.

$$SMQ = \frac{1}{N}\sum_{i=1}^{N} scoh_i - \frac{1}{N(N-1)/2}\sum_{i \neq j}^{N} scop_{i,j} \tag{11}$$

SMQ quantification is divided into the quantification of intra-connectivity and inter-connectivity. The value ***scoh*** quantifies cohesiveness of a given service while ***scop*** quantifies coupling between services. High *scoh* and low *scop* represent a cohesive and loosely

Class	*Topic distribution*			
(org.mybatis.jpetstore.)*	0	1	2	3
web.actions.AccountActionBean	**0.79**	0.03	0.14	0.04
domain.Account	**0.82**	0.04	0.09	0.05
service.AccountService	**0.72**	0.09	0.09	0.95
mapper.AccountMapper	**0.81**	0.06	0.06	0.07
web.actions.AbstractActionBean	0.26	0.25	0.25	0.23
mapper.LineItemMapper	0.1	**0.61**	0.1	0.18
domain.Item	0.05	**0.73**	0.18	0.05
domain.LineItem	0.04	**0.82**	0.04	0.11
domain.CartItem	0.06	**0.83**	0.06	0.06
web.actions.CartActionBean	0.12	**0.71**	0.13	0.04
mapper.ItemMapper	0.09	**0.66**	0.16	0.09
web.actions.CatalogActionBean	0.03	0.08	**0.86**	0.03
domain.Category	0.13	0.13	**0.62**	0.12
mapper.ProductMapper	0.11	0.11	**0.68**	0.11
mapper.CategoryMapper	0.14	0.14	**0.57**	0.14
domain.Product	0.1	0.1	**0.7**	0.1
service.CatalogService	0.05	0.17	**0.74**	0.05
mapper.SequenceMapper	0.15	0.15	0.15	**0.55**
mapper.OrderMapper	0.11	0.1	0.1	**0.69**
domain.Sequence	0.19	0.18	0.19	**0.44**
domain.Order	0.08	0.16	0.02	**0.73**
service.OrderService	0.06	0.24	0.06	**0.64**
web.actions.OrderActionBean	0.2	0.08	0.05	**0.66**

Table 2: Topic distribution for each class on the JPetStore project (The distribution is expected to sum to 1, however due to rounding there are cases where that does not happen)

coupled architecture. μ_i represents the total edges for a service. An edge is counted when there is a structural call dependency between N entities. $\sigma_{i,j}$ is similar to μ but acts on a service to service level, meaning that an edge occurs when there is a dependency between service i and service j.

$$scoh_i = \frac{u_i}{N_i^2}, \quad scop_{i,j} = \frac{\sigma_{i,j}}{2(N_i \times N_j)} \tag{12}$$

CMQ (*Conceptual Modularity Quality*) quantifies modularity from a conceptual viewpoint. Higher CMQ represents better modularity.

$$CMQ = \frac{1}{N}\sum_{i=1}^{N} ccoh_i - \frac{1}{N(N-1)/2}\sum_{i \neq j}^{N} ccop_{i,j} \tag{13}$$

$$ccoh_i = \frac{u_i}{N_i^2}, \quad ccop_{i,j} = \frac{\sigma_{i,j}}{2(N_i \times N_j)} \tag{14}$$

CMQ is very similar to SMQ but textual terms are used instead of call dependencies. Therefore, an edge is considered if the intersection between terms is not empty.

4.3 Project collection

With the goal of evaluating the quality of the services proposed by our methodology, we collected 200 Java Spring applications from *GitHub* using its search API (v3) and executed against the state of the art metrics proposed by Jin et al. [17] on independence of functionality and modularity.

We used the GitHub search API for code as a mean to identify repositories using the terms *"RequestMapping"* and *"Controller"*, as those are very common and unique to applications built using the Spring Framework. Since any request to a GitHub endpoint is bound to 1000 results per each search, we used the parameter of file size to be able to find more repositories with Java Spring projects. Thus, we created a set of queries by increasing the size interval (min_size and max_size) by 200 bytes and bound to a starting point of 500 bytes and final point of 200000 (around 200 KB). The query (template) used was as follows:

https://api.github.com/search/code?q=RequestMapping+
Controller+language:java+size:{min_size}..{max_size}

Executing this set of queries we identified 104024 results, however, many of them represent results on the same repository something that the API does not allow to exclude. These results were then filtered by uniqueness, removing duplicate references to the same repository and forks of the same repository. Filters were also applied to some very common repositories, such as Spring-Boot forks of the framework, demo and test projects by using the following stop words *{'release', 'framework', 'learn', 'source', 'spring', 'study', 'demo', 'test', 'practice', 'practise'}*. That reduced the original 104024 results to 29368.

Based on the criteria taken by Borges et al. [6] and Ma et al. [25] on works regarding criteria collection of GitHub repositories we selected the top repositories based on GitHub stars. Thus, for every repository, we parsed the number of stars, open issues, subscribers and forks.

In order to guarantee that the projects are monolithic applications, only projects containing one "src" folder are considered. Any project with less than 30 classes is also discarded as a project with that dimension may represent just a "toy" Java project. From the final projects the top 200 by highest number of stars are selected as evaluation data set.

The histogram of the projects collected by the number of classes is presented in Figure 6. In perspective, the biggest project considered is roughly around 2500 classes.

4.4 Setup

We implemented the presented methodology in a proof of concept and tested the collected projects against state of the art metrics. Our main goal is, from a quantitative point of view, to understand if the microservices being proposed are relevant regarding independence of functionality and modularity.

Regarding the number of topics, we selected a range of arbitrary values according to quantity of classes. The ranges are wide enough in order to allow the identification of a knee point on coherence values, but not too large as that would slow down the process in a meaningful way.

Clustering resolution was also arbitrarily set. From our analysis a range from 0.3 to 1 should be able to deal with most applications in

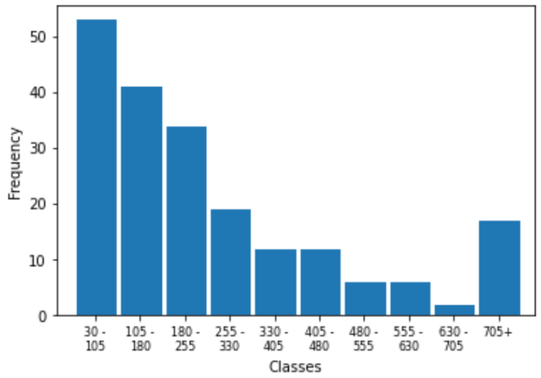

Figure 6: Histogram of collected projects by class count

the hundreds of components: a resolution of 1 can identify clusters of larger sizes in small applications (in the tens) and resolution of 0.3 can identify smaller clusters in large applications (in the hundreds). The set range guarantees us that we are able to identify smaller or larger microservices if they quantitatively represent the best cohesion and loosely coupling.

The selection factor of the best proposition of microservices is done by maximising the combination of *CHM, CHD, SMQ, CMQ, IRN* and *IFN*. Regarding operation identification as a mean to measure CHM and CHD we collected all the methods present in controllers' classes. We also applied a threshold to calculate CHD since only tight terms related to the domain should be considered, requiring extensive cleaning and manual labour, something not feasible given the significant set of applications. In fact this is a similar process to the one present in the work proposing the metrics. The use of stop words for pre-processing was also very generic without any specificity per project. Ideally an analysis and addition of common terms irrelevant to the domain should be added despite the effort we put in information extraction stage to reduce the inclusion of external abstraction.

4.5 Results

The boxplot of the results obtained across all 200 applications can be found in Figure 7.

CHM and CHD which represent independence of functionality at operation level (methods exposed by services as interfaces) both show medians roughly around 0.6 which represent positive proposals regarding independence and well-defined services.

SMQ and CMQ which represent modularity of services and are bound between -1 and 1 presented medians roughly around 0.2 and 0.4 respectively. Some negative outliers regarding SMQ were observed, however CMQ remained positive across all projects.

IFN quantifies the interfaces exposed by a given interface, hence, a smaller IFN represents a higher likelihood of a service having a single responsibility. The median and both Q1 and Q3 on the boxplot presented in Figure 8 are quite compact and close to values representing a single responsibility service (*ie.* IFN of 1).

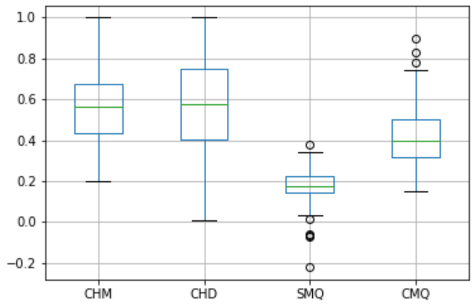

Figure 7: Metrics' boxplot across 200 projects

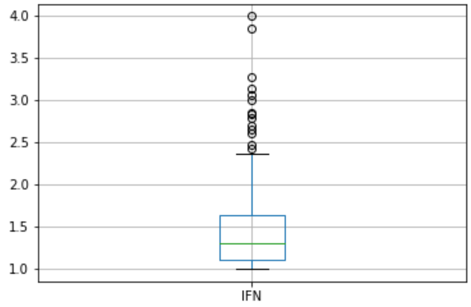

Figure 8: IFN's boxplot across the 200 projects

The identified projects and its results (*i.e.* metrics and proposed services) can be found at our open-source repository[3].

4.6 Discussion

Regarding metrics of independence of functionality, both CHM and CHD presented good median values, however there is a big interval between upper and lower whisker. The upper whisker results are more frequent on smaller applications and applications better designed with strong domain concepts. The lower whisker could be described as projects of larger complexity, with weaker domain concepts or an overall lack of pre-processing and cleaning of domain terms.

The higher values of CMQ compared to SMQ could be justified by the usage of an approach mainly based on the extraction of lexical terms. The underlying structure of the graph for clustering is based on the structural dependencies, however, edge weights when based on semantics will privilege classes semantically closer. Both SMQ and CMQ are bound between -1 and 1, despite some negative outliers on SMQ the results obtained are positive and are relevant regarding the modularity of proposed services. Higher values of CMQ over SMQ are expected given the semantic-focused approach of our methodology. Another reason for lower SMQ is higher levels of abstraction which are discussed later.

[3] https://anonymous.4open.science/r/61cd83e3-47b7-405d-b5ec-8ac90c81d6c7

Regarding IFN, our proposed tool seems to have some tendency to choose smaller services (due to metrics combination) resulting in better IFNs and worse SMQ (*i.e.* with smaller microservices it is expected that more external connections exist).

Considering that we didn't apply customised pre-processing for each project, when there are levels of abstraction composed of multiple components, there is a possibility that those same abstractions are identified as a topic and eventually result in a service. Better pre-processing would definitely help, however there always will be cases where identified topics do not represent the reality of the domain. Allowing the user to discard topics that purely represent abstractions or are composed by very scattered terms could bring benefits to the process of service identification.

Our approach uses classes as the unit of decomposition of a project. Even though we can identify multiple classes from a java file, such granularity might be too high to identify cross-cutting concepts and segregate them into their unique services. Using finer units of decomposition such as methods might help the identification of such cross-cutting concepts and improve the overall identification of domain terms and consequently better cohesion and loosely coupling.

The current value of our proposed tool for developers is mainly to explore an architecture and guide the user to identify microservices according to metrics of independence of functionality and modularity. Although we identify a resolution at which the independence of functionality and modularity are at its highest, choosing the adequate number of services is dependent on the subjective understanding of what represents good microservices to the expert. Considering that each cluster of classes results in a direct proposition of a microservice the trade-off between cohesion and coupling is an important feature that said expert should evaluate. In short, choosing lower resolution, hence smaller microservices, results inherently in higher cohesion with higher coupling, while higher resolutions results in the opposite. The appropriate balance between cohesion and coupling at a class level is not that straightforward to identify, hence the ultimate decision should be made by the expert.

We also conducted an analysis with the goal to understand on how the methodology performs for projects of different sizes regarding number of classes (results found at [4]). We didn't find any evidence that the methodology performs significantly better for any of the groups. We hypothesise that the main cause of success or failure to the methodology is abstraction, considering that it hinders topic identification and in some cases the identification of abstraction as a topic might result in isolating such abstraction into an independent proposal of a service. Although abstraction usually increases with bigger projects, our data set is composed of applications wildly varying in their domain, hence, a smaller but more complex application could have more abstraction than a large application more focused on the domain. Further analysis will have to be conducted to confirm our hypothesis.

Overall, all the metrics demonstrate promising results towards microservice identification regarding independence of functionality and modularity.

4.7 Threats to validity

In this section we discuss the threats to the proposed method organised according to [39].

Conclusion validity. A possible threat is related to the reliability of measures, in this case of the metrics. We have implemented the metrics based on their original publication and when in doubt we have contacted the authors to discuss and clarify them. We have also performed extended tests to guarantee the correctness of the results.

Internal validity. Another threat to validity refers to how the parameters of the study are selected. Ideally we should do an extensive analysis of how the number of topics and resolution affect the results directly, however, that would require a vast amount of work given the high number of applications and possible permutations. To mitigate such threat, we defined arbitrary ranges for each parameter. Regarding the resolution parameter, the selected arbitrary range demonstrated a level of granularity capable of identifying small microservices in large applications as well as larger microservices in smaller applications, in other words, it should have the capability to identify different levels of granularity in different sized applications. Ultimately, resolution is tested against metrics and we can understand how it performs. Unfortunately the same cannot be done with number of topics. To identify the appropriate number of topics we resort to a metric of coherence which evaluates if the terms of each topic make sense together. This is applied on an arbitrary range of number of topics. However, the ultimate contribution of the number of topics to the method can only be evaluated after clustering and no further individual conclusions can be achieved.

Construct validity. A construct validity threat relates to the quality of microservices being proposed. Even though we used the state of the art metrics regarding microservices, it is theoretically possible, however unlikely, to achieve good metrics that do not necessarily represent good propositions of microservices. The amount of projects taken into consideration should decrease such possibility, however, a qualitative analysis of the metrics used would have to be conducted in order to make further conclusions. Bringing experts to conduct an analysis of the decomposition we propose would also help understand the quality of such propositions, and help identifying possible improvements to the overall process.

External validity. An external threat relates to the architectures of projects we used. Our goal was to take monolithic applications and thus it was necessary to filter out projects composed of other architectures such as SOA and MA. Repositories built upon such architectures are often composed of multiple projects, hence multiple `src` folders. To mitigate such occurrences only projects containing one `src` folder were considered given the definition of monolithic applications as being composed of a single program. However, there is no absolute guarantee that all the projects considered follow the definition of a monolith. A second threat relates to the fact that we use only open-source projects. Nevertheless, it is now common to find companies and other entities making their code available. For

[4] https://anonymous.4open.science/r/61cd83e3-47b7-405d-b5ec-8ac90c81d6c7

instance, our list of projects includes a project by the Australian Government (AtlasOfLivingAustralia/biocache-service). However, it is possible the results may vary for proprietary software.

5 RELATED WORK

With the increased popularity of MA, an increase in demand of formalising the process of migration also emerged. The migration itself is a complex process, very dependant on the domain of the software project which might require multiple iterations until completion [10]. In the work done by Balalaie et al. [3] and Fritzsch et al. [13] with the goal of systematising the process by identifying patterns, strategies and challenges, both reported the decomposition of software systems as being one of the main struggles. In Fritzsch et al. [13] research, none of the participants was aware of automated systems that could assist the migration to MA.

The work done to date is especially focused on the identification of microservices. No methodology/tool has been proposed that can identify and refactor a whole system into a working version of a MA. The identification and proposal of microservices research so far can be divided into three main approaches: solutions based on static analysis, solutions based on dynamic analysis and model-oriented solutions.

Static analysis solutions. Static analysis techniques are widely used on software testing and code analysis and also promising in this area given the amount of information that can be extracted from source code. Mazlami et al. [26] propose three different strategies of coupling that can be used as a base to clustering resulting in microservice proposals. The strategies proposed resort in the source code of the project and meta-data collected from a versioning system. The first strategy proposed, logical coupling, relies on the premise that changes made to a monolith are only done to a small set of classes on a given module, thus, by analysing the history changes and taking into account that the classes that change together should also be together in a microservice. Their second approach is built upon semantic similarity, meaning that classes that talk about the same concepts should be grouped together. Lastly, coupling by contribution, based on Conway [9] law expressing that the structure of the software represents the structure of the organization, thus, specific modules that receive changes by a specific group of people/teams should be maintained together. Kamimura et al. [19] propose a solution heavily focused on clustering. The clustering technique used by them is proposed by Kobayashi et al. [21] and has the main goal to undermine the importance of modules that are omnipresent in the project, enabling a better identification of components [22]. The information used to feed the clustering is extracted starting by the endpoints of specific framework annotations (such as @Controller in Java Spring Framework). Other annotations such as @Entity and @Table are also used to identify data persistence and domain knowledge.

Dynamic analysis solutions. Dynamic analysis techniques have emerged as an alternative to static analysis using program execution analysis (for example, *logs*) in order to obtain extra information about the software in question.

According to Candela et al. [7], techniques that process code analysis based on their syntactic relationships, using metrics such as coupling and cohesion or naming conventions may not be sufficient for optimal identification, given that the code-level relationship may not be the same in terms of functionality [18]. In order to deal with this limitation, Jin et al. [18] propose the use of traces collected during the execution of certain test cases created by the user, which according to [11] allow for a better exposure of the true functionality of software. They later extend their work by proposing *Functionality-oriented Service Candidate Identification (FoSCI)* [17] resorting to a search-based functional atom grouping algorithm based on Non-dominated Sorting Genetic Algorithm-II (NSGA-II) using as optimisation objectives both intra and inter structural connectivity and inter and intra conceptual connectivity. Their work also results in the extension and proposition of new metrics for quantitative evaluation of proposed microservices.

Model-oriented solutions. The importance of models in the development of software systems and model-driven-development (MDD) allows for the use of model-based approaches since they also represent a view over the interactions between system's components.

Gysel et al. [15] follow a model-oriented approach resorting to high-level models such as domain models and use cases diagrams to extract representations of the system into a graph. The graph is then clustered resorting to Epidemic Label Propagation [24] and Girvan-Newman algorithms [14]. Chen et al. [8] take a similar path, however, resorting to data-flow diagrams. The main challenge with model oriented approaches is how heavily oriented they are towards user input, usually requiring extensive work to provide the tools with up to date diagrams and appropriate formats.

Summary. Overall, most of the techniques proposed use semantics in one way or another as part of their methodology. After all, the software is ultimately being read and created by developers and the way modules are grouped together to fit nicely with the domain of the project is one of the main ways to have better software comprehension.

All the techniques presented use semantic techniques as a mean to identify domain terms and coherent groups of information. However, they resort to basic techniques such as Jaccard distance or Tf-IDF (term frequency–inverse document frequency). On the other hand, we proposed an enhanced usage of semantics as a mean to identify coherent microservices by applying topic modelling techniques which produce more accurate results.

6 CONCLUSION

We present a methodology to identify microservices from monolithic software architectures. The methodology proposed is agnostic of the programming language and paradigm. We have implemented the methodology in a proof of concept and tested against the state of the art quantitative metrics on independence of functionality and modularity of microservices. The evaluation was conducted against the collection of 200 open-source Java Spring applications from GitHub.

Our proposed methodology performed well regarding independence of functionality with medians roughly close to 0.6 for both CHM and CHD and low values of IFN representing relevant propositions of microservices. The results concerning modularity are also positive, with better performance regarding the CMQ over SMQ given the nature of our semantic based approach.

As future work, we plan to produce executable microservices according to the identifications made. We also intend to investigate and extend LDA to better handle abstractions and its impact on topic identification. Moreover, an empirical evaluation where developers evaluate the results of the methodology is also an important step as it can bring to the methodology improvements unforeseen.

ACKNOWLEDGMENTS

This work is financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia within project UIDB/50014/2020.

REFERENCES

[1] [n.d.]. About the Symbol Solver · javaparser/javaparser Wiki. https://github.com/javaparser/javaparser/wiki/About-the-Symbol-Solver. (Accessed on 07/14/2020).

[2] [n.d.]. Spring | Home. https://spring.io/. (Accessed on 07/30/2020).

[3] Armin Balalaie, Abbas Heydarnoori, Pooyan Jamshidi, Damian Tamburri, and Theodore Lynn. 2018. Microservices migration patterns. *Software: Practice and Experience* 48 (07 2018). https://doi.org/10.1002/spe.2608

[4] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. 2003. Latent Dirichlet Allocation. *J. Mach. Learn. Res.* 3, null (March 2003), 993–1022.

[5] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. 2008. Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment* 2008, 10 (oct 2008), P10008. https://doi.org/10.1088/1742-5468/2008/10/p10008

[6] H. Borges, A. Hora, and M. T. Valente. 2016. Understanding the Factors That Impact the Popularity of GitHub Repositories. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344.

[7] Ivan Candela, Gabriele Bavota, Barbara Russo, and Rocco Oliveto. 2016. Using Cohesion and Coupling for Software Remodularization: Is It Enough? *ACM Transactions on Software Engineering and Methodology* 25 (06 2016), 1–28. https://doi.org/10.1145/2928268

[8] Rui Chen, Shanshan Li, and Zheng (Eddie) Li. 2017. From Monolith to Microservices: A Dataflow-Driven Approach. 466–475. https://doi.org/10.1109/APSEC.2017.53

[9] Melvin Conway. [n.d.]. Conway's Law. http://www.melconway.com/Home/Conways_Law.html (Accessed on 12/27/2019).

[10] Zhamak Dehghani. 2018. How to break a Monolith into Microservices. https://martinfowler.com/articles/break-monolith-into-microservices.html. (Accessed on 12/26/2019).

[11] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: A taxonomy and survey. *Journal of Software Maintenance and Evolution: Research and Practice* 25 (01 2013). https://doi.org/10.1002/smr.567

[12] Santo Fortunato and Marc Barthélemy. 2007. Resolution limit in community detection. *Proceedings of the National Academy of Sciences* 104, 1 (2007), 36–41. https://doi.org/10.1073/pnas.0605965104 arXiv:https://www.pnas.org/content/104/1/36.full.pdf

[13] Jonas Fritzsch, Justus Bogner, Stefan Wagner, and Alfred Zimmermann. 2019. Microservices Migration in Industry: Intentions, Strategies, and Challenges. https://doi.org/10.1109/ICSME.2019.00081

[14] Michelle Girvan and Mark Newman. 2001. Community structure in social and biological networks. *Proceedings of the National Academy of Sciences of the United States of America* 99 (11 2001), 7821–7826.

[15] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. 2016. Service Cutter: A Systematic Approach to Service Decomposition. 185–200. https://doi.org/10.1007/978-3-319-44482-6_12

[16] Hamed Jelodar, Yongli Wang, Chi Yuan, Xia Feng, Xiahui Jiang, Yanchao Li, and Liang Zhao. 2019. Latent Dirichlet Allocation (LDA) and Topic Modeling: Models, Applications, a Survey. *Multimedia Tools Appl.* 78, 11 (June 2019), 15169–15211. https://doi.org/10.1007/s11042-018-6894-4

[17] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. 2019. Service Candidate Identification from Monolithic Systems based on Execution Traces. *IEEE Transactions on Software Engineering* (2019), 1–1. https://doi.org/10.1109/TSE.2019.2910531

[18] W. Jin, T. Liu, Q. Zheng, D. Cui, and Y. Cai. 2018. Functionality-Oriented Microservice Extraction Based on Execution Trace Clustering. In *2018 IEEE International Conference on Web Services (ICWS)*. 211–218. https://doi.org/10.1109/ICWS.2018.00034

[19] M. Kamimura, K. Yano, T. Hatano, and A. Matsuo. 2018. Extracting Candidates of Microservices from Monolithic Application Code. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*. 571–580. https://doi.org/10.1109/APSEC.2018.00072

[20] Pooja Kherwa and Poonam Bansal. 2018. Topic Modeling: A Comprehensive Review. *ICST Transactions on Scalable Information Systems* 7 (07 2018), 159623. https://doi.org/10.4108/eai.13-7-2018.159623

[21] K. Kobayashi, M. Kamimura, K. Kato, K. Yano, and A. Matsuo. 2012. Feature-gathering dependency-based software clustering using Dedication and Modularity. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*. 462–471. https://doi.org/10.1109/ICSM.2012.6405308

[22] Kenichi Kobayashi, Manabu Kamimura, Keisuke Yano, Koki Kato, and Akihiko Matsuo. 2013. SArF Map: Visualizing Software Architecture from Feature and Layer Viewpoints. https://doi.org/10.1109/ICPC.2013.6613832 arXiv:1306.0958 [cs.SE]

[23] Renaud Lambiotte, Jean-Charles Delvenne, and Mauricio Barahona. 2014. Random Walks, Markov Processes and the Multiscale Modular Organization of Complex Networks. *IEEE Transactions on Network Science and Engineering* 1, 2 (Jul 2014), 76–90. https://doi.org/10.1109/tnse.2015.2391998

[24] Ian X. Y. Leung, Pan Hui, Pietro Liò, and Jon Crowcroft. 2009. Towards real-time community detection in large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics* 79 6 Pt 2 (2009), 066107.

[25] W. Ma, L. Chen, Y. Zhou, and B. Xu. 2016. What Are the Dominant Projects in the GitHub Python Ecosystem?. In *2016 Third International Conference on Trustworthy Systems and their Applications (TSA)*. 87–95.

[26] G. Mazlami, J. Cito, and P. Leitner. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*. 524–531. https://doi.org/10.1109/ICWS.2017.61

[27] Abhishek Mishra. [n.d.]. Demystifying Louvain's Algorithm and Its implementation in GPU | by Abhishek Mishra | WalmartLabs | Medium. https://medium.com/walmartlabs/demystifying-louvains-algorithm-and-its-implementation-in-gpu-9a07cdd3b010. (Accessed on 07/16/2020).

[28] M. E. J. Newman. 2006. Modularity and community structure in networks. *Proceedings of the National Academy of Sciences of the United States of America* 103, 23 (06 Jun 2006), 8577–8582. https://doi.org/10.1073/pnas.0601602103 16723398[pmid].

[29] S. Newman. 2015. *Building Microservices: Designing Fine-Grained Systems.* O'Reilly Media. https://books.google.pt/books?id=jjl4BgAAQBAJ

[30] Claus Pahl and Pooyan Jamshidi. 2016. Microservices: A Systematic Mapping Study. 137–146. https://doi.org/10.5220/0005785501370146

[31] D. L. Parnas. 1972. On the Criteria to Be Used in Decomposing Systems into Modules. *Commun. ACM* 15, 12 (Dec. 1972), 1053–1058. https://doi.org/10.1145/361598.361623

[32] Jean Petrić, Tracy Hall, and David Bowes. 2018. How Effectively Is Defective Code Actually Tested? An Analysis of JUnit Tests in Seven Open Source Systems. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering* (Oulu, Finland) *(PROMISE'18)*. Association for Computing Machinery, New York, NY, USA, 42–51. https://doi.org/10.1145/3273934.3273939

[33] Sara Rahiminejad, Mano R. Maurya, and Shankar Subramaniam. 2019. Topological and functional comparison of community detection algorithms in biological networks. *BMC Bioinformatics* 20, 1 (27 Apr 2019), 212. https://doi.org/10.1186/s12859-019-2746-0

[34] Michael Röder, Andreas Both, and Alexander Hinneburg. 2015. Exploring the Space of Topic Coherence Measures. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining* (Shanghai, China) *(WSDM '15)*. Association for Computing Machinery, New York, NY, USA, 399–408. https://doi.org/10.1145/2684822.2685324

[35] Carson Sievert and Kenneth Shirley. 2014. LDAvis: A method for visualizing and interpreting topics. In *Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces*. Association for Computational Linguistics, Baltimore, Maryland, USA, 63–70. https://doi.org/10.3115/v1/W14-3110

[36] Keith Stevens, Philip Kegelmeyer, David Andrzejewski, and David Buttler. 2012. Exploring Topic Coherence over Many Models and Many Topics. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, Jeju Island, Korea, 952–961. https://www.aclweb.org/anthology/D12-1087

[37] Xiaobing Sun, Xiangyue Liu, Li Bin, Bixin Li, David Lo, and Lingzhi Liao. 2017. Clustering Classes in Packages for Program Comprehension. *Scientific Programming* 2017 (01 2017), 1–15. https://doi.org/10.1155/2017/3787053

[38] X. Sun, X. Liu, B. Li, Y. Duan, H. Yang, and J. Hu. 2016. Exploring topic models in software engineering data analysis: A survey. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. 357–362.

[39] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering.* Springer Publishing Company, Incorporated.