Steinmetz: Toward automatic decomposition of monolithic software into microservices

Jakob Löhnertz
Picnic Technologies
Amsterdam, Netherlands
mail@jakob.codes

Ana Maria Oprescu
University of Amsterdam
Amsterdam, Netherlands
A.M.Oprescu@uva.nl

Graduate School of Informatics, University of Amsterdam

CEUR-WS.org/Vol-2754/paper2.pdf

Abstract

Industry is adopting the microservices paradigm for greenfield development as well as for migrating monolithic software. However, the latter approach involves significant manual work, specifically in the early stages of the decomposition process, when determining boundaries between the services.

We devise a methodology to automatically generate microservice candidate recommendations for any given monolithic software. We leverage three coupling dimensions: static, semantic, and evolutionary. For each dimension, we calculate a weighted graph. We aggregate the three dimensions into a single graph that we cluster into microservice candidate recommendations.

We evaluate our methodology using several established metrics as well as our PoC implementation, Steinmetz [1]. Preliminary results are encouraging: the methodology works as expected, there are clear results regarding feasibility of metrics to assess the quality of microservice candidate recommendations, and we are able to identify the best suitable graph clustering algorithm.

Copyright © by the paper's authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In: Proceedings of the Seminar Series on Advanced Techniques & Tools for Software Evolution, Virtual conference (originally in Amsterdam, the Netherlands), 01-02 July 2020, published at http://ceur-ws.org

[1] *Steinmetz* is the German word for *stonemason*: it carves the monolith into many smaller stones, the microservices.

1 Introduction

While many software engineers talk about the benefits and challenges of *microservices*, they are nothing more than an architectural pattern for designing backend software. However, the idea mandates new strategies that developers and operations could apply when leveraging microservices in their business logic.

When deciding to move an application to a microservices-based architecture, the decomposition process is a significant challenge: defining where within the existing application one service should end and the next one should begin, i.e., detecting latent boundaries that partition the monolith into smaller pieces is a tedious manual task [FBZW18]. This work covers an algorithmic methodology that can assist software engineers in such decompositions.

Generally, microservices-based software architecture is implemented either in a greenfield approach, meaning that microservices are developed from scratch, or by migrating a monolithic software solution to independent microservices [Zha]. Our work focuses solely on the latter approach. While there is already some research in this domain [FBZW18], utilizing an existing application is still underrepresented.

Accurate boundary detection for each microservice is difficult, especially when the teams working on the existing software are captive in their mental models [FBZW18]. To address such cases, we devise a semi-automated approach to assist software engineers in decomposing an existing monolith into microservices.

Existing approaches mostly leverage graph clustering algorithms to receive a partitioning of the original monolithic software into microservice candidates [FBZW18]. However, apriori knowledge about the expected number of resulting microservices is required.

We formulate three research questions addressing the current shortcomings in semi-automatic microservice decomposition:

RQ1: How can existing monolithic software be analyzed to formulate microservices recommendations?

RQ2: Which numeric metrics are most suitable for evaluating the output of microservice decomposition?

RQ3: Which graph clustering algorithms are most suitable for generating microservice recommendations?

2 Background

Our methodology takes as input only the monolith. Various types of program analysis techniques are leveraged to collect information about the monolith algorithmically.

2.1 Static coupling calculation

Generally, call graphs can be created via static or dynamic program analysis [GKM82]. In an object-oriented program, the vertices represent methods or classes [GDDC97]. One major usage of the *static analysis* is to construct a graph such that each routine from the input program is a vertex in the output graph while each invocation is an edge connecting two routines.

Compared to other types of program analysis, static coupling does not bear an intuitive notion of strength. To counteract this problem, we use a modification of the *Response for a class* (RFC) metric [BDW99]. Other popular metrics, like CBO, are not as feasible as they cannot be geared toward calculating a coupling percentage between a pair of classes. RFC_α of a class counts all methods that can be invoked with a stack depth of α. The default is $\alpha = 0$, counting the directly accessible methods of a class (i.e., the compiler allows them to be called). We calculate RFC_0 values for every pair of classes that are statically coupled via method invocations. The percentage reflects the strength of coupling between two classes c_1 and c_2:

$$\mathrm{RFC}_0(c_1, c_2) = \frac{|(\mathrm{IM}(c_1, c_2) \cup \mathrm{IM}(c_2, c_1))|}{|(\mathrm{AM}(c_1, c_2) \cup \mathrm{AM}(c_2, c_1))|}$$

$\mathrm{IM}(c_i, c_j)$ is the set of methods c_i invokes on c_j, and $\mathrm{AM}(c_i, c_j)$ is the set of methods c_i can access in c_j.

2.2 Semantic coupling calculation

Calculating the semantic similarity between two documents is a traditional discipline in the domain of natural language processing (NLP). Several NLP steps (*Tokenization*, *Normalization*, *Vectorization*, *Weighting via `tf-idf`*, *Latent semantic indexing* (LSI)) are performed for every source file in the application until the resulting vectors can be run through a similarity function [S$^+$01]. A common approach is the *cosine similarity*, which results in a percentage [S$^+$01].

2.3 Evolutionary coupling calculation

Similarly to the *semantic similarity*, the overlap between software development life cycles (creation, maintenance, and eventual deletion) of two entities (e.g., classes) can be computed as a percentage. For instance, entities that have an interface toward each other are likely *evolutionary similar* as a change of the signature or semantics of the interface also involves changes in the other entity. Compared to other types of coupling, however, the *evolutionary similarity* does not have an established method to compute the percentage. We leverage work by Tornhill on *evolutionary similarity* [Tor15]. Their approach is based on mining information out of version control system (VCS) data.

2.4 Graph clustering

Graph clustering partitions a graph into sub-graphs based on some closeness criteria [LF09]. In our work, sub-graphs do not overlap. A cluster is a group of vertices that are interconnected via their *intra-cluster edges* and connected to vertices outside the cluster via *inter-cluster edges* [LF09, New04].

2.4.1 Metrics

Modularity (Q) [New04] measures the quality of a graph clustering by comparing the interconnectedness of intra-cluster vertices to their inter-cluster edges. The metric ranges from lacking clustering structure, 0.0, to a perfect clustering, 1.0. Any given graph has a theoretical maximum in terms of the achievable *Modularity.* However, it is rarely 1.0, and finding the global maximum is an NP-complete optimization problem [BDG$^+$06]. Values over 0.4 already indicate a strong clustering structure [New04, FB07].

Cluster Factor (CF) [MM06] measures the cohesion within a cluster compared to the coupling from that cluster to the other clusters. This factor ranges on a scale from $[0.0, 1.0]$ with higher values being better. The metric measures the quality of a graph clustering by comparing the interconnectedness of intra-cluster vertices to their inter-cluster edges per cluster.

3 Related work

Gysel and Kölbener [GKGZ16] were the first to work toward a complete methodology to extract microservice recommendations from an existing system by clustering an input graph. Their work was qualified by Fritzsch et al. as the benchmark in this domain of research [FBZW18]. However, this groundwork has major shortcomings in terms of usability, as indicated by the authors themselves.

Types of analysis. *Model analysis*, as utilized by Gysel and Kölbener, requires much manual work

[GKGZ16]. Kruidenberg [Kru18] created a generator for the input required by said model analysis using *static*, *dynamic*, and *evolutionary* analysis input. Mazlami [MCL17] chose *semantic*, and *evolutionary* analysis to create their graph representation. Our approach strives to remove any need for manual preprocessing of the input. We utilize three distinct input dimensions for constructing a more precise weighted graph which can then be clustered: *static analysis*, *semantic analysis*, and *evolutionary analysis*.

Graph clustering. Various clustering algorithms were used in related work, all requiring the number of microservices as an input parameter [GKGZ16, Kru18, MCL17]. We use seven clustering algorithms that do not require the expected number of microservices.

Evaluative metrics. Gysel and Kölbener [GKGZ16] evaluated their requirements on a custom scale together with a binary questionnaire to assess the quality of the recommendations. Mazlami [MCL17] devised a set of six metrics that measure the output of their methodology. However, several do not have clear semantics (e.g., *Average Contributors per Microservice*). Kruidenberg [Kru18] used a combination of metrics from the other two [GKGZ16, MCL17]. Instead, we devise a set of six numeric metrics that offers an objective and quantifiable view on the resulting microservice recommendations, to be able to evaluate our methodology as well as the recommendations.

Another related topic is architectural refactoring [HMZ09]. Multiple approaches out of of this domain were studied, while we ultimately chose the work of Mitchell and Mancoridis to evaluate our approach [MM06], since it is the first work out of this domain that has been corroborated by multiple other authors as well [HMZ09].

4 Methodology

In this section, we cover the methodology for creating recommendations, the metrics to evaluate the quality of the recommendations, and the proof of concept (PoC). The methodology is geared toward object-oriented software input while it is language agnostic. However, the PoC is only working with Java software, although it was designed with extensibility for other platforms in mind.

4.1 Rationale

We map the three measured dimensions of coupling to the output by leveraging several *design principles* of microservices [Sar, Mar, New15]. We select the following two based on the importance assigned in literature and their ties to the three types of coupling:

Principle 1. One service should only handle one task. This is tied to *static coupling* and *semantic coupling.* The term *task* is ambiguous, as it has no notion of size or complexity attached. However, the objective size of a microservice is not very important [Mar]. Instead, the idea of the *bounded context* [Eva04] is widely accepted as a helpful methodology to devise the boundaries of each microservice in a given application [New15, Mar]. One service should encapsulate one *bounded context* while keeping them as small as reasonably possible [New15, Mar].

Principle 2. The services should have independent software development life cycles. This is related to *evolutionary coupling.* As this architectural style suits well having separate teams handle one or multiple microservices [New15], services should have independent life cycles. The more dependent the life cycles are, the more coupled the services are, and the longer it could take to apply feature additions, changes, or bug fixes.

4.2 Extraction

This section covers the extraction of coupling data for each weighted graph of each of the input dimensions.

4.2.1 Static coupling

Starting from the source root of the given application (e.g., `src` in Java), an algorithm recursively walks down the classes and their methods. Both class instantiations, as well as method invocations, are registered and result in an edge between two classes. Finally, the RFC_0 values are calculated for every edge to determine its weight.

4.2.2 Semantic coupling

When processing source code, the major difference to common NLP approaches occurs in the *tokenization* and the *stop word filtering.* This first step is specific to the respective programming language as different languages use different sets of control characters and differing compound identifier patterns (e.g., camel case and snake case). Importantly, human-readable, not yet compiled source code is necessary as the semantics are mostly lost in compiled code. Every line is parsed separately, and language-specific keywords such as `package` or `import` trigger that the line gets discarded. Next, every line of the source code is stripped of non-word characters (i.e., `W` from regular expressions) [IEE93]. Additionally, explicit strings as well as every other text that is still left then and matches a compound identifier pattern (e.g., camel case), is split into terms. Moreover, numeric characters, as well as single characters, are filtered out, as both provide no relevant semantic meaning. Next, a list of *stop words* specific to the programming language is used to filter these out (e.g., `public`, `static`, `void`). Finally, everything that was not filtered out until this point is split at white space characters (i.e., `t` from regular expressions) [IEE93]. The result is a flattened list of terms for each document (i.e., source code file) that is then used for the remaining NLP steps.

4.2.3 Evolutionary coupling

The approach leveraged to mine VCS data is modeled after the methodology of Tornhill [Tor15]. Theoretically, every VCS that has the notion of revisions that include files and that is able to generate a log file which details past revisions together with the files involved, can be used to extract *evolutionary coupling* information. We support the following VCS with our PoC: *Git*, *Subversion*, *Mercurial*, *Perforce*, and *TFS*. A caveat with this approach is that by default, the log files will only include complete files. Owing to the fact that in applications written in Java, one file is also equivalent to one class, we can generate the coupling graph without any changes to the log format. After parsing the input log file, a weighted *evolutionary coupling* graph is constructed, such that the vertices $v_i \in V$ are the classes extracted from the VCS log file while the edges $e_i \in E$ are resembling that two classes have overlapping software development life cycles. The weight of the edges is a percentage that indicates how much the life cycles overlap $W(e_i) \in [0.0, 1.0]$. Finally, we use an adjustable threshold for the *minimum shared revisions* to filter out noise.

4.3 Construction

We begin with the graph constructed out of the static analysis G_{static}. The two remaining graphs are merged into G_{static}. *Semantic coupling* and *evolutionary coupling* can be calculated for any two given classes regardless of their connection in the input application codebase. Hence, this step uses set-theoretic intersections to maintain the results out of the static program analysis as a single source of truth: classes that are not connected in it are not able to instantiate or invoke each other in the application.

$$G_{\text{combined}} = ((G_{\text{static}} \cap G_{\text{semantic}}) \cap G_{\text{evolutionary}})$$

For the vertices, only ones from the static analysis graph are used for the same reason. Merging of the edges is twofold. The edges are drawn from the static analysis graphs to connect the vertices while the other two input graphs do not add new edges but just enrich the information of existing ones:

$$E_{\text{combined}} = ((E_{\text{static}} \cap E_{\text{semantic}}) \cap E_{\text{evolutionary}})$$

We use *edge weighting* to define factors for each input dimension to increase or decrease its contribution to the combined weight of each edge $e_i \in E_{\text{combined}}$ in the combined graph G_{combined}.

4.4 Clustering

Once we construct the combined graph G_{combined}, we cluster it such that groups of vertices that are strongly coupled, as indicated by the combined edge weights, are more likely to end up in the same cluster. These clusters mirror *bounded contexts*, and each cluster represents a microservice candidate recommendation.

Various graph clustering algorithms have various goals, mechanisms, and features [LF09, For10, DDGDA05]. We evaluate several established graph clustering algorithms in the context of our goal. Table 1 shows the feasible ones. Although these algorithms differ in terms of the exact input and output formats, the core idea is passing a list of edges as input. A simple example format is a tuple of three items: two vertices $v_1, v_2 \in V$ and the edge weight of the edge connecting them $e_{(1,2)} \in E,\ G = (V,\ E)$.

Table 1: Selected graph clustering algorithms

Name	Runtime complexity
MCL [VD00]	$\mathcal{O}(nk^2)$
Walktrap [PL05]	$\mathcal{O}(mn^2)$
Clauset et al. [CNM04]	$\mathcal{O}(md \log n)$
Louvain [BGLL08]	$\mathcal{O}(n)$
Label Propagation [RAK07]	$\mathcal{O}(n)$
Infomap [RB08]	$\mathcal{O}(n)$
Chinese Whispers [Bie06]	$\mathcal{O}(n)$

n = number of vertices; m = number of edges; k = threshold for the number of resources allocated per vertex; d = depth of the resulting dendrogram.

4.5 Evaluation Metrics

We select six metrics and organize them into four categories: a) input fidelity, b) general clustering quality, c) *Modularity* (Q), d) *Mean Cluster Factor* (mCF).

Input fidelity. We use *input fidelity* to describe the percentage of the classes of the entire application covered by a given input. The *input fidelity* reflects the eventual quality of the clustering. The higher these values are, the more precisely the clustering represents the application's inherent structure.

General clustering quality. This set of metrics has the property that it can not be strictly defined if low values are better or worse. The four metrics in this set are: *number of clusters*, *number of inter-cluster edges*, *ratio of inter-cluster edge weights.*

Mean Cluster Factor. The *Modularization Quality* (MQ) metric was proposed and validated by Mitchell and Mancoridis [MM06]. It measures the quality of a partition by analyzing its interconnectivity and intraconnectivity. First, the *Cluster Factor* (CF) is calculated for every cluster. Originally, the last step in calculating the *Modularization Quality* metric is summing up the *Cluster Factor* values of every cluster. Instead, we take the arithmetic mean of all *Cluster Factor* values. The ones that yield ∞ due to division-by-zero are still treated as 0.0 [MM06]. The final metric value ranges on the scale of $[0.0, 1.0]$. We name the modified metric *Mean Cluster Factor* (mCF).

Modularity. The *Modularity* metric is included but not preferred since some of our supported graph clustering algorithms use it as their fitness function which causes the metric to favor them. Nevertheless, the *Mean Cluster Factor* as well as the *Modularity* metric are specifically interesting when inputs achieve high scores in both of them at the same time.

4.6 Proof of Concept

We published our implementation as open-source software, licensed under the *Apache License v2*. The entire source code can be found on GitHub [2]. We split our semantic coupling analysis implementation into a separate library, as we found no good existing library for this purpose. Similarly, that library can also be found on GitHub [3].

The high-level architecture of our implementation is a three-tier web-based design, shown in Fig. 1. Steinmetz has two major functionalities that can be used via RESTful endpoints. These functionalities are to analyze and persist an input monolithic application as well as clustering and retrieving it again as a separate step, as seen in Fig. 1.

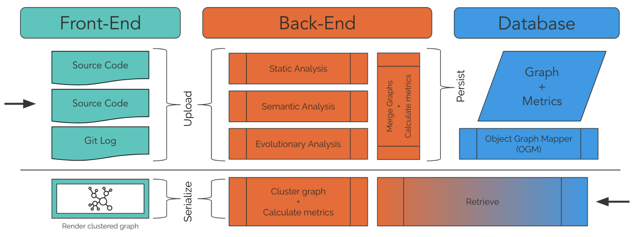

Figure 1: Overview of the *Steinmetz* architecture

5 Evaluation setup

In this section, we cover the setup of the experiments conducted to evaluate our methodology.

5.1 Evaluation applications

A microservices-based architecture targets large-scale, web-based, back-end applications. To be able to analyze all three input dimensions, the applications have to be available as source code and be version-controlled via a supported VCS.

We conducted a survey on GitHub using the query `NOT library NOT framework NOT android NOT distributed NOT client stars:>500 language:Java`. We examine projects with at least 500 stars to filter out educational projects that are most likely not peer-reviewed. The first 50 pages of search results were analyzed for software projects of varying sizes with the following properties: standalone, large-scale, web-based, back-end, and monolithic (i.e., having a single main method). Table 2 lists the selected projects.

Table 2: Selected software projects for the evaluation

Project	SLOC	Commits
Apache OpenMeetings	4,222	2,755
Apache Solr	203,364	33,018
BigBlueButton	36,578	26,774
Google Bazel	340,747	24,442
Halo	23,010	1,788
Heritrix	15,210	2,126
KeyCloak	419,351	12,340
OpenCMS	326,836	23,674
Openfire	77,112	9,368
OpenTripPlanner	67,976	9,560
Red5	8,528	481
Teammates	42,280	17,221
Traccar	52,090	6,202
XWiki	238,253	38,841

We conducted three experiments: the first experiment used all the selected projects measuring all the selected metrics to answer **RQ1**. We analyzed all the 14 experiment subjects via the described methodology and subsequently clustered them with all the seven graph clustering algorithms. The second experiment recorded the major metrics summarized per graph clustering algorithm using the same output used in the first experiment. It helps to answer **RQ2** (feasibility) and **RQ3** (output quality). The third experiment evaluated the runtime performance of the PoC by measuring the runtime during the analysis step of the first experiment. All experiments were executed on a `2.4 GHz 8-Core Intel Core i9-9880H` processor.

6 Preliminary results and discussion

Fig. 2 and 3 are heat maps, depicting the metrics values (color) of each utilized graph clustering algorithm (y-axis) for each of the input applications (x-axis). A large difference between the two is that the range of values is a lot larger with the Q metric.

[2]`https://github.com/loehnertz/Steinmetz/`
[3]`https://github.com/loehnertz/semantic-coupling/`

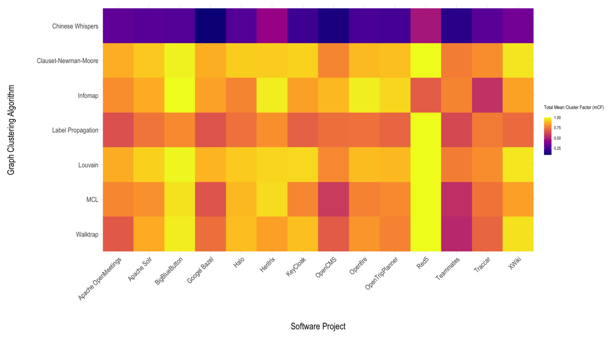

Figure 2: Experiment 1: Overview of analyzed software projects measuring the mCF metric

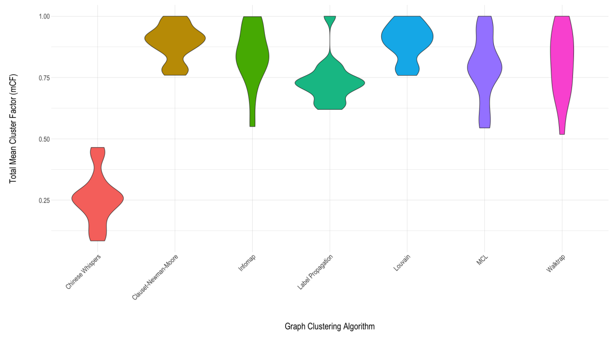

Figure 4: Experiment 2: Comparison of graph clustering algorithms measuring the mCF metric

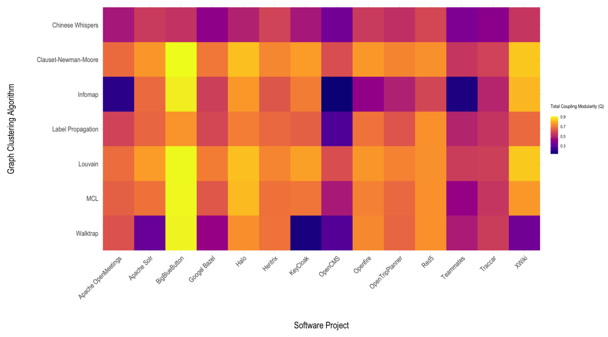

Figure 3: Experiment 1: Overview of analyzed software projects measuring the Q metric

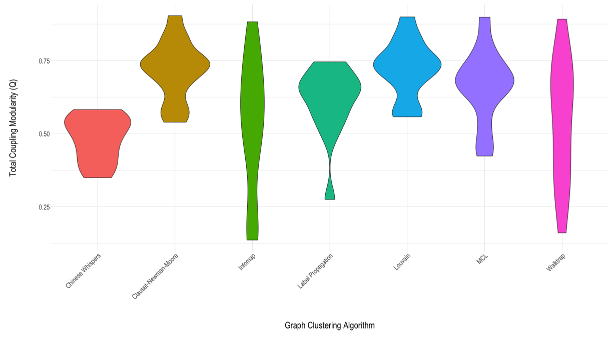

Figure 5: Experiment 2: Comparison of graph clustering algorithms measuring the Q metric

Fig. 4 and 5 are violin plots showing the values of the two metrics (y-axis) across the graph clustering algorithms (x-axis). of the algorithms. The aforementioned major difference is visible in this plot as well. Compared to the heat maps, these plots show more distinctly how the algorithms performed in total.

Fig. 6 shows the bivariate correlation between the source lines of code (SLOC) (x-axis) and the runtime in seconds (y-axis).

Proof of concept. The first experiment shows that *Steinmetz* works as expected. The mean of the highest scores for each input project regarding the clustering metrics are 0.91 for the *Mean Cluster Factor* (mCF) and 0.72 for the *Modularity* (Q). The coefficients of variation of the highest scores are good values of 7.48% and 14.27%, respectively. The fidelity values for the *semantic* and *evolutionary* inputs are 95% and 25% in the mean. The fidelity of the *static* inputs is always 100% as it is the base to calculate the others. All these metrics are promising, indicating our methodology as an answer to **RQ1**.

Metrics. Fig. 2, 3, 4, and 5 show the mCF metric generally scoring high, while the Q values are more spread. The algorithms never achieve 1.0 for the Q metric which aligns with the claim that only fabricated graphs are going to achieve 1.0 [New04]. The mCF metric does score 1.0, even for algorithms that have a large spread. As a result, we deem the Q metric to be more fit than the mCF metric, which answers **RQ2**.

Graph clustering algorithms. Fig. 2, 3, 4, and 5 show the *Chinese Whispers* algorithm performing the worst, while *Louvain* is performing the best overall in both, the mCF and Q metrics, although *Clauset-Newman-Moore* scores almost the same results. Both algorithms are also achieving very high bivariate correlations of $R = 0.89$ and $R = 0.90$ between their results for the mCF and Q metrics. Thus, these two algorithms perform the best, which answers **RQ3**.

Runtime performance. As Fig. 6 depicts, the runtime of the PoC is predictable with a bivariate correlation between input size and runtime being a very high $R = 0.92$. Due to this, the small confidence interval, and the slope of the regression line, the runtime complexity of the PoC seems to be $\mathcal{O}(N)$.

6.1 Threats to validity

Shared code classes (i.e., utility and data classes) are attributed to one microservice, although its coupling might be equally high toward other ones. This could be addressed via a fuzzy clustering approach

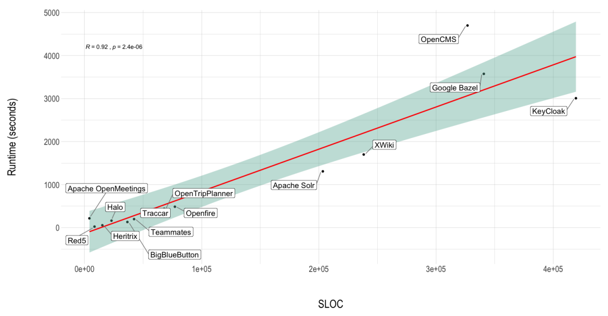

Figure 6: Experiment 3: Runtime performance of the analysis of the software projects

(i.e., one class can belong to multiple services).

Quality assessment of the generated results is the inherent weakness of our and other methodologies. Although we devise objective metrics numeric in nature, a human could still want certain classes not to be put into separate services due to security concerns, compliance reasons, etc.

7 Conclusion

Our area of research is still young compared to others in the domain of software evolution [FBZW18].

We showed how a rationale can tie the input to the output, to be able to assert that the final microservice recommendations were generated via the means of an input that leads to *good* microservice boundaries which were extracted out of the input — the three dimensions of coupling in the case of our methodology. Furthermore, it became apparent that a multitude of graph clustering algorithms is available to be utilized while offering insights into which are feasible for this domain of research by leveraging the discussed metrics. We also saw that some graph clustering algorithms consistently perform better than others.

We devised and implemented a methodology that can assist software engineers by automatically extracting microservice recommendations from a monolithic codebase. We evaluated the quality of the recommendations on several real-world applications.

7.1 Future work

Evaluation study involving human software engineers. We expect that specific patterns might be detected where our methodology falls short.

Automatically generating API of microservices candidates. Offer a more holistic set of recommendations that can be validated from an additional perspective: the quality of the interfaces between the services.

References

[BDG$^+$06] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. Maximizing modularity is hard. *arXiv preprint physics/0608255*, 2006.

[BDW99] Lionel C. Briand, John W. Daly, and Jurgen K Wust. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on software Engineering*, 25(1):91–121, 1999.

[BGLL08] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[Bie06] Chris Biemann. Chinese whispers: an efficient graph clustering algorithm and its application to natural language processing problems. In *Proceedings of the first workshop on graph based methods for natural language processing*, pages 73–80. Association for Computational Linguistics, 2006.

[CNM04] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[DDGDA05] Leon Danon, Albert Diaz-Guilera, Jordi Duch, and Alex Arenas. Comparing community structure identification. *Journal of Statistical Mechanics: Theory and Experiment*, 2005(09):P09008, 2005.

[Eva04] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[FB07] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *Proceedings of the national academy of sciences*, 104(1):36–41, 2007.

[~~FBZW18~~] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. *arXiv preprint arXiv:1807.10059*, 2018.

[For10] Santo Fortunato. Community detection in graphs. *Physics reports*, 486(3-5):75–174, 2010.

[GDDC97] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices*, 32(10):108–124, 1997.

[~~GKGZ16~~] Michael Gysel, Lukas Kölbener, Wolfgang Giersche, and Olaf Zimmermann. Service cutter: a systematic approach to service decomposition. In *European Conference on Service-Oriented and Cloud Computing*, pages 185–200. Springer, 2016.

[GKM82] Susan L Graham, Peter B Kessler, and Marshall K Mckusick. Gprof: A call graph execution profiler. In *ACM Sigplan Notices*, volume 17, pages 120–126. ACM, 1982.

[HMZ09] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Department of Computer Science, King's College London, Tech. Rep. TR-09-03*, page 23, 2009.

[IEE93] IEEE. *ISO-IEC 9945-2: IEEE Std. 1003.2-1992 Information Technology - Portable Operating System Interface: Shell and Utilities.* IEEE Standards Office, 1993.

[Kru18] Dennis Kruidenberg. From monoliths to microservices: The decomposition process, 2018.

[LF09] Andrea Lancichinetti and Santo Fortunato. Community detection algorithms: a comparative analysis. *Physical review E*, 80(5):056117, 2009.

[Mar] Martin Fowler. Microservices. (`https://martinfowler.com/articles/microservices.html`) [last accessed: 2019/01/21].

[MCL17] Genc Mazlami, Jurgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 524–531. IEEE, 2017.

[MM06] Brian S Mitchell and Spiros Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[New04] Mark EJ Newman. Fast algorithm for detecting community structure in networks. *Physical review E*, 69(6):066133, 2004.

[New15] Sam Newman. *Building Microservices.* O'Reilly Media, Inc., 1st edition, 2015.

[PL05] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *International symposium on computer and information sciences*, pages 284–293. Springer, 2005.

[RAK07] Usha Nandini Raghavan, Réka Albert, and Soundar Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical review E*, 76(3):036106, 2007.

[RB08] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences*, 105(4):1118–1123, 2008.

[S$^+$01] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[Sar] Saravanan Subramanian. Microservices Design Principles. (`https://dzone.com/articles/microservices-design-principles`) [last accessed: 2019/07/22].

[Tor15] Adam Tornhill. *Your code as a crime scene: use forensic techniques to arrest defects, bottlenecks, and bad design in your programs.* Pragmatic Bookshelf, 2015.

[VD00] Stijn Marinus Van Dongen. *Graph clustering by flow simulation.* PhD thesis, 2000.

[Zha] Zhamak Dehghani. How to break a Monolith into Microservices. (`https://martinfowler.com/articles/break-monolith-into-microservices.html`) [last accessed: 2019/01/19].