Lessons learned from hyper-parameter tuning for microservice candidate identification

Rahul Yedida, Tim Menzies
Computer Science, NC State, USA
ryedida@ncsu.edu, timm@ieee.org

Rahul Krishna, Anup Kalia, Jin Xiao, Maja Vukovic
IBM Research, USA
{rkrsn,anup.kalia,jinoaix,maja}@ibm.com

***Abstract*—When optimizing software for the cloud, monolithic applications need to be partitioned into many smaller *microservices*. While many tools have been proposed for this task, we warn that the evaluation of those approaches has been incomplete; e.g. minimal prior exploration of hyperparameter optimization. Using a set of open source Java EE applications, we show here that (a) such optimization can significantly improve microservice partitioning; (b) an open issue for future work is find which optimizer works best for different problems.**

***Index Terms*—microservices, hyper-parameter optimization**

I. Introduction

Traditional software is "monolithic"; i.e. one large entity that handles all concerns. Such software needs to be divided into "microservices" to take full advantage of the flexibility and scalability offered by cloud computing environments [13]. That microservice architecture allows for independent scalability, the use of different programming languages for each microservice, and does not have a single point of failure.

In our experience, the process of manually refactoring a monolithic application to a microservice architecture can be a time-consuming, expensive, and an error-prone endeavor. Different architects may propose different candidate sets when manually constructing a set of microservices. Also, different software systems may have different inherent architectures. Lastly, different engineers may have differing opinions on the number of microservices to split the software into. Hence, like many other researchers [10, 7, 11, 14] we explore machine learning algorithms to identify a candidate set of microservices from a monolithic application. But even here, there are still problems. For example, as shown in column one of Table I, there are many ways to find these partitions. Also, as seen in column two of that table, these methods contain many parameters that must be set via engineering judgment. However, different software architects (and different types of applications) may have different views on the appropriate values for each of these configuration options.

We characterize all these issues as *hyperparameter optimization* (HPO); i.e. the automatic selection of (a) which features/algorithms to use; and (b) what settings to apply to those algorithms. In our case, the algorithms are the methods used to find partitions/microservices. Using the partitioning methods of Table I this paper compares off-the-shelf partitioning methods to two tuning (i.e., HPO) methods. The rest of this paper:

TABLE I: Hyper-parameter choices studied in this paper. All the methods take runtime traces (or uses cases) as their input

Algorithm	Hyper-parameter	Range
mono2micro [11]	Number of clusters	$\{2, 3, ...\}$
FoSCI [10]	Number of clusters	$\{2, 3, ...\}$
	Number of iterations to run NSGA-II	$\{1, 2, ...\}$
	Population size for NSGA-II	$\{5, 6, ...\}$
	Parent size to use in NSGA-II	$\{5, 6, ...\}$
	Threshold for clustering to stop	$\mathbb{R}^+$
MEM [14]	Maximum partition size	$\{2, 3, ...\}$
	Number of partitions	$\{2, 3, ...\}$
Bunch [15]	Number of partitions	$\{1, 3, ...\}$
	Initial population size for hill climbing	$\{2, 3, ...\}$
	Number of neighbors to consider in hill-climbing iterations	$[0, 1]$
CO-GCN [7]	Number of clusters	$\{2, 3, ...\}$
	Loss function coefficients $\alpha_1, \alpha_2, \alpha_3$	$\alpha_i \in \mathbb{R}$
	Number of hidden units in each layer, h_1, h_2	$h_i \in \mathbb{Z}^+$
	Dropout rate	$d \in [0, 1)$

- Documents the improvement seen to microservice creation when the methods of Table I are tuned via HPO. Specifically, tuning can significantly improve results, with the highest improvement being 266% (FoSCI on the plants dataset, on the SM metric).
- Reveals a new research challenge *microservice design strategies*. While our results shows that tuning is better than non-tuning, we also show that no single approach is a clear winner across multiple datasets or multiple performance goals.

Hence, going forward, we can strongly recommend that microservice creation be controlled by automatic tuning. That said, it is still an open issue which tuner to use. To help other researchers build upon our work, we make our code fully open source[1].

II. Background

Business case: A recent report shows that only 20% of the enterprise workloads are in the cloud, and they were predominately written for native cloud architectures [5]. This leaves 80% of legacy applications on-premises, waiting to be refactored (with microservices) and modernized for the cloud. Once ported to microservices, business services can

[1]What URL?

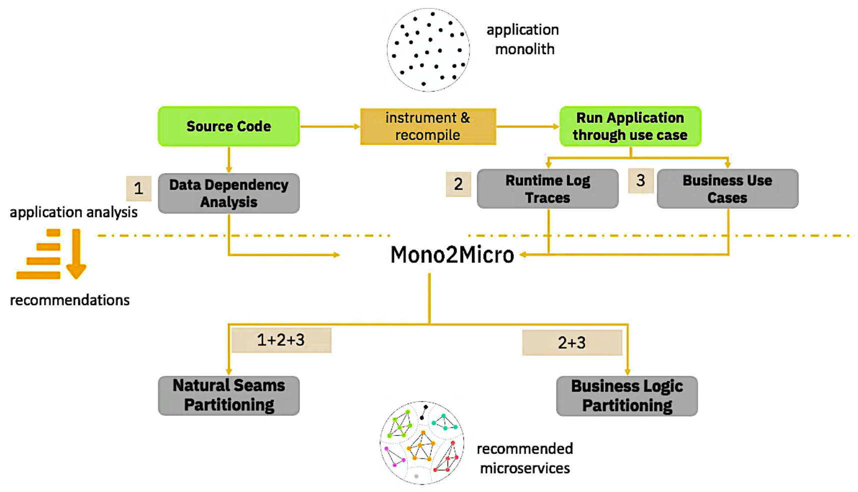

Fig. 1: Conceptual overview of microservice generation.

be independently enhanced and scaled, providing agility and improved speed of delivery.

Application refactoring is the process of restructuring existing code without changing its external behavior and semantics. Currently, refactoring is usually done manually and is expensive, time-consuming, and error-prone. There are many ways to implement application refactoring. For example, Table 1 lists different ways to partition the monolith. In that table, the core idea is to understand what parts of the code call each other other (e.g. by reflecting on test cases or uses cases), then cluster those parts of the code into separate microservices (see Figure 1).

Just for the sake of clarity, we repeat here Daya et al.'s notes on the difference between microservices and *service-oriented architectures* (SOA) [5]. SOA is a set of services that a business wants to provide to a customer. Internally, their architecture usually includes some mediation tool (a.k.a. the traffic cop) that redirects service requests to a set of service providers. This architecture is implemented is some web-oriented architecture (e.g. REST services) and can be viewed as a middleware solution that supports service assembly, orchestration, monitoring and management.

SOAs are usually built manually using a set architectural principles that address issues like as modularity, encapsulation, loose coupling, separation of concerns, reuse and composability. On the other hand, the construction of microservices (as explored here) is guided by some automatically-inferred partitions. These partitions are reversed engineered from test cases or use cases describing taken from some pre-exisiting monolithic application. Microservices can be created relatively cheaply since they are built incrementally just by, one-at-a-time, porting existing monolithic services to the cloud. SOA, on the other hand, requires a large and lengthy transformational initiative.

III. Problem Formulation

Let us denote a set of classes in an application A as $\mathcal{C}^A$ such that $\mathcal{C}^A = \{c_1^A, c_2^A, \ldots, c_k^A\}$, where c_i^A represents an individual class. A partition $\mathcal{P}^A$ on $\mathcal{C}^A$ is a set of subsets $\{P_1^A, P_2^A, \ldots, P_n^A\}$ such that $\bigcup_{i=1}^{n} P_i^A = \mathcal{C}^A$, $P_i^A \neq \phi \forall i = 1, \ldots, n$, and $P_i^A \cap P_j^A = \phi \forall i \neq j$, i.e., the elements of $\mathcal{P}_A$ are disjoint. A partitioning algorithm f is a function that induces a partition $\mathcal{P}_A$ on an application with classes $\mathcal{C}_A$. The goal of a microservice candidate identification algorithm is to identify a function f that maximizes a set of metrics $m_1, m_2, \ldots, m_p$, i.e., given an application characterized by its class set $\mathcal{C}_A$, we aim to find

$$\mathcal{P}_A^* = \underset{\mathcal{P} \in B_{\mathcal{C}_A}}{\arg\max} \sum_{i=1}^{p} \alpha_i m_i(\mathcal{P}) \tag{1}$$

where B_S denotes the set of all partitions of a set S.

Many methods fall into this framework. Kalia et al. [11] discuss mono2micro, which performs hierarchical agglomerative clustering using Jaccard similarity. They use runtime traces as the source of their data. Jin et al. [10] propose FoSCI, which also uses runtime traces as a data source. They remove redundant traces using a trace reduction algorithm, and perform hierarchical clustering into "functional atoms".

When this problem is posed as an optimization problem, this is usually achieved via a linear combination of the metrics. For example, Desai et al. [7] design a loss function incorporating several metrics directly into their graph convolutional network approach, allowing an end-to-end training solution. In their approach (CO-GCN), they aim to minimize the effect of outliers on the resulting partition. To do so, they begin by building upon the graph convolutional network framework of Kipf and Welling [12]. Specifically, while their method of obtaining a latent graph representation is the same, they use a custom loss function with three components: the structural outlier component $\mathcal{L}_{str}$, the attribute outlier component $\mathcal{L}_{attr}$, and the clustering assignment component $\mathcal{L}_{clus}$. Having posed these objectives as *loss functions*, the linear combination of these forms the final loss function that is to be minimized. Therefore, the form of (1) that they obtain is

$$\begin{aligned} &\min_{\theta} \mathcal{L} = \alpha_1 \mathcal{L}_{str} + \alpha_2 \mathcal{L}_{attr} + \alpha_3 \mathcal{L}_{clus} \\ &\text{s.t.constraints} \end{aligned}$$

where we squash all the parameters into θ and do not write out the full list of constraints for brevity.

Bunch [15] characterize the partitioning problem as a search problem, and use computational search procedures to find a partition that maximizes modular quality (MQ) and structural modular quality (SMQ). In particular, we use the hill climbing search algorithm discussed in their paper (we refer the reader to §4.1 of their paper for details).

Mazlami et al. [14] propose MEM, which is based on first constructing a graph based on the change history of files (in detail, they propose three methods of doing this, but we do not discuss it further here). They pose the partitioning problem as a graph cut problem and use minimum spanning trees to find min-cut partitions from the constructed graph.

Jin et al. [10] propose FoSCI, which is based on multi-objective optimization. They use runtime traces as the input data, prune the traces using a *trace reduction* algorithm, and group *functional atoms*, logical units that provide some function. These are grouped using a modified version of NSGA-II [6], a multi-objective optimization algorithm.

We note that across all of these approaches, a common theme is the existence of hyper-parameters, listed in Table I. Because these papers are focused on proposing a novel approach, little attention is paid to tuning these hyper-parameters, or discussing reasonable defaults for them. Therefore, we ask whether tuning can improve the results. The next section provides a background on hyper-parameter tuning.

Hyper-parameter optimization: Hyper-parameter optimizers find good settings for a learner (the meta-options that guide the details of the algorithm's function). These include (e.g.) the number of clusters for a partitioning or a clustering algorithm, or (e.g.) the number of layers in a deep learner.

Bergstra and Bengio [2] advocate for random hyper-parameter tuning, in part, due to the ease of implementation. A more sophisticated optimizer is TPE that uses an "acquisition function" to reflect on the model built so far in order to learn the next most informative thing to evaluated. Prior results are divided into *best* and *rest* and each division is modelled as a Gaussian. Candidate evaluations are then ranked by how well they select for *best* and avoid *rest*. The better candidates are then evalauted which, in turn, udpates *best* and *rest* and their associated Gaussians.

Many papers [8, 17, 1, 19, 18] demonstrate that the tunings learned via such optimizers can lead to models that significantly out-performed the untuned "off-the-shelf" versions. Yet, none of the approaches we study do such tuning. We investigate this research gap by using two of the optimizers recommended by Bergstra et al. [3], random and TPE. Both of these optimizers have been endorsed in major venues (NeurIPS) and have since been widely used by the community–see [4] for a reference implementation).

IV. Experimental Methods

Experimental Setup: We run each algorithm using the default settings and using hyper-parameter optimization. Where defaults are unspecified, we use reasonable default values, either from the papers that proposed them, or through experience with the algorithm. We run the algorithms 30 times each, tuned and untuned, for a statistically valid comparison. We try each approach on open source datasets, described below.

Data: We use four open-source projects to test the algorithms under study. These are *daytrader*[2], a sample online stock trading system, *plants*[3], a sample online store to purchase plants and pets, *acmeair*[4], a fictional airline application, and *jpetstore*[5], a sample pet store. All applications are web apps built using Java Enterprise Edition. Some applications use frameworks to help build them, such as Spring.

Hyper-parameter optimization: We use two hyper-parameter optimization approaches, random sampling [2], and Tree of Parzen Estimators (TPE) [3], using the hyperopt framework [4]. These are two commonly used approaches from the AI literature, and have been compared against in recent SE literature [1]. We run both for 100 iterations.

Because we use several metrics (see §IV), we need to guide the hyper-parameter optimization algorithm to achieve the results that we want. To achieve this, we go back to (1), which is a *loss* function described as a linear combination of metrics. For simplicity, we set each $|\alpha_i| = 1$, with the sign being *positive* if the goal is to *minimize* the metric, and *negative* if the goal is to *maximize* the metric. Having framed our loss function, we use the hyper-parameter optimization algorithms to minimize this goal. Because there is not much disparity in the scale of the metrics, we do not normalize them.

Metrics: For a fair comparison with prior work, we must compare using the same set of metrics. We choose a total of five metrics to evaluate our core hypothesis that hyper-parameter tuning improves microservice extraction algorithms:

Inter-partition call percentage (ICP) is the percentage of runtime calls that are between different partitions. For lower coupling, *lower* ICP is better.

Business context purity (BCP) measures the mean entropy of business use cases per partition. Specifically,

$$BCP = \frac{1}{K}\sum_{i=1}^{K} \frac{BC_i}{\sum_j BC_j} \log_2 \left(\frac{BC_i}{\sum_j BC_j} \right)$$

where K is the number of partitions and BC_i is the number of business use cases in partition i. Because BCP is fundamentally based on entropy, *lower* values are better.

Structural modularity (SM), as defined by Jin et al. [10], combines cohesion and coupling, is given by

[2]https://github.com/WASdev/sample.daytrader7
[3]https://github.com/WASdev/sample.plantsbywebsphere
[4]https://github.com/acmeair/acmeair
[5]https://github.com/mybatis/jpetstore-6

TABLE II: Summary of Results. Each column is a separate clustering method. Each row is a different performance metric.

(a) mono2micro

Metric	Treatment	Wins
BCS [-]	untuned	0
	random	4
	hyperopt	4
ICP [-]	untuned	4
	random	0
	hyperopt	0
SM [+]	untuned	2
	random	3
	hyperopt	3
MQ [+]	untuned	1
	random	3
	hyperopt	3
IFN [-]	untuned	1
	random	4
	hyperopt	4

(b) FoSCI

Metric	Treatment	Wins
BCS [-]	untuned	0
	random	0
	hyperopt	4
ICP [-]	untuned	0
	random	4
	hyperopt	0
SM [+]	untuned	0
	random	4
	hyperopt	0
MQ [+]	untuned	0
	random	0
	hyperopt	4
IFN [-]	untuned	0
	random	4
	hyperopt	0

(c) MEM

Metric	Treatment	Wins
BCS [-]	untuned	3
	random	0
	hyperopt	1
ICP [-]	untuned	2
	random	2
	hyperopt	0
SM [+]	untuned	3
	random	0
	hyperopt	1
MQ [+]	untuned	1
	random	2
	hyperopt	4
IFN [-]	untuned	1
	random	3
	hyperopt	1

(d) Bunch

Metric	Treatment	Wins
BCS [-]	untuned	2
	random	3
	hyperopt	3
ICP [-]	untuned	2
	random	4
	hyperopt	4
SM [+]	untuned	4
	random	2
	hyperopt	2
MQ [+]	untuned	3
	random	3
	hyperopt	3
IFN [-]	untuned	2
	random	4
	hyperopt	4

(e) CO-GCN

Metric	Treatment	Wins
BCS [-]	untuned	2
	random	0
	hyperopt	2
ICP [-]	untuned	0
	random	0
	hyperopt	4
SM [+]	untuned	0
	random	2
	hyperopt	2
MQ [+]	untuned	4
	random	0
	hyperopt	0
IFN [-]	untuned	1
	random	0
	hyperopt	3

$$SM = \frac{1}{K}\sum_{i=1}^{n}\frac{coh_i}{N_i^2} - \frac{1}{N_i(N_i-1)/2}\sum_{i<j<K} coup_{i,j}$$

where N_i is the number of classes in partition[6] i, K is the number of partitions, coh_i is the cohesion of partition i, and $coup_{i,j}$ is the coupling between partitions i and j. Higher values of SM are better. For a subset of classes C, the *cohesion* is defined as the mean pairwise similarity between classes in that subset. Specifically, it is defined as

$$coh_i = \frac{2}{m(m-1)}\sum_{\substack{c_i,c_j\in C\\ i<j}} \sigma(c_i, c_j)$$

where c_i refers to a class in the subset, σ is a general similarity function bounded in $[0, 1]$, and the $i < j$ condition imposes a general ordering in the classes. One can similarly define *coupling* between two *partitions*, C_1, C_2 as the mean pairwise similarity scores of classes, where the classes in the pairs belong to different partitions:

$$coup_{i,j} = \frac{\sum\limits_{\substack{c_1\in C_i\\ c_2\in C_j}} \sigma(c_1, c_2)}{|C_i||C_j|}$$

where C_i, C_j are clusters in the partition.

Modular quality (MQ), coined by Mitchell and Mancoridis [15], is defined on a graph $G = (V, E)$ as

$$\begin{aligned} MQ &= \frac{2\mu_i}{2\mu_i + \epsilon} \\ \mu_i &= \sum_{(e_1,e_2)\in E} \mathbb{1}(e_1 \in C_i \wedge e_2 \in C_i) \\ \epsilon &= \sum_{\substack{(e_1,e_2)\in E\\ i\neq j}} \mathbb{1}(e_1 \in C_i \wedge e_2 \in C_j) \end{aligned}$$

where $\mathbb{1}$ is the indicator function, and C_i, C_j are clusters in the partition. Higher values of MQ are better.

Finally, the **interface number (IFN)** of a partition is the number of interfaces needed in the final microservice architecture. Here, an interface is said to be required if, for an edge between two classes in the runtime call graph, the two classes are in different clusters.

[6]Note that we use the term "partition" to refer to both the set of all class subsets, as well as an individual subset, but it is typically unambiguous.

Statistics: Our comparison method of choice is the Scott-Knott test, which was endorsed at TSE'13 [16] and ICSE'15 [9]. The Scott-Knott test is a recursive bi-clustering algorithm that terminates when the difference between the two split groups is insignificant. Scott-Knott searches for split points that maximize the expected value of the difference between the means of the two resulting groups. Specifically, if a group l is split into groups m and n, Scott-Knott searches for the split point that maximizes

$$\mathbb{E}[\Delta] = \frac{|m|}{|l|}\left(\mathbb{E}[m] - \mathbb{E}[l]\right)^2 + \frac{|n|}{|l|}\left(\mathbb{E}[n] - \mathbb{E}[l]\right)^2$$

where $|m|$ represents the size of the group m.

The result of the Scott-Knott test is *ranks* assigned to each result set (two results have the same rank if their difference is insignificant).

V. Results

Due to space constraints, we show only the summary tables in this paper. Our full results are available online[7].

First we ask **RQ1**, **" What approaches in the literature form the state-of-the-art for the mono-to-micro problem?"**.

Our review of different approaches in §**??** showed a variety of techniques applied to the problem. Due to the low number of papers that we found (the oldest, Bunch, being from 2006), we did not apply further filtering criteria. Rather, we observed that different approaches did better in different metrics, and this was not consistent across datasets. Therefore, rather than declare one approach as *the* state-of-the-art, we use the *set* of approaches as a Pareto frontier forming the state-of-the-art collectively. These approaches along with their hyper-parameters have been listed in Table I. Therefore, we say:

The algorithms shown in the first column of Table I is a representative set of the state-of-the-art.

[7]https://github.com/yrahul3910/tuned-microservices-results

Next we ask **RQ2**, **"Is there an exemplary approach that outperforms the others across multiple datasets?"**

Table II shows our results. In that figure, each row shows results from one assessment metric (BSC, ICP, SM, MQ, IDM) and each column shows results from a different technology from Table 1. Each result is three bar charts showing how often the untuned, random tuned, and TPE (hyperopt) tuner achieved best results first (where "best" was determined by our statistical tests) within our four data sets (daytrader, plants, acmeair, jpetstore). For example, top-left, we see that random and hyperopt both achieved best BCS scores for mono2micro (and the untuned mono2micro method never succeeded).

The general trend across all the results is (a) untuned usually lost to tuning; but (b) there is no stable pattern about which tuner (random or hyperopt) worked best for a particular performance measure. The exception to this general pattern is that (c) untuned MEM which performed best in 3/5 metrics (BCS, ICP, SM). It turns our that MEM ignores its command line options and uses its own internal methods for deciding how many partitions (this was discovered after we had run all the above experiments).

In summary, in answer **RQ2**, we say that

Tuning is better than not tuning, but it is still an open issue is a particular tuner works better than anything else.

VI. Lessons Learned

Our investigation on hyper-parameter tuning for state-of-the-art microservice candidate identification services revealed several important lessons, which can be summarized as:

(i) **Do not use approaches off-the-shelf.** These tend to have subpar results.

(ii) **The choice of tuning algorithm is not as important as the decision to tune.** The random and hyperopt optimizers performed differently on different metrics and datasets, but in a broader view, shared the same number of highest Scott-Knott rankings. Therefore, the decision to tune is important, but the choice of algorithm may not be as important.

(iii) **More research is required on tuning.** This work suggest that a new research challenge is appropriate: *microservice design strategies*. While our results shows that tuning is better than non-tuning, we also show that no single approach is a clear winner across multiple datasets or multiple performance goals

Our long term goal is to can find one optimiser that works best across multiple data sets and evaluation criteria. Realistically, it may take a while to achieve this goal.

References

[1] A. Agrawal, X. Yang, R. Agrawal, R. Yedida, X. Shen, and T. Menzies, "Simpler hyperparameter optimization for software analytics: Why, how, when," *IEEE Transactions on Software Engineering*, 2021.

[2] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization." *Journal of machine learning research*, vol. 13, no. 2, 2012.

[3] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *25th annual conference on neural information processing systems (NIPS 2011)*, vol. 24. Neural Information Processing Systems Foundation, 2011.

[4] J. Bergstra, D. Yamins, D. D. Cox *et al.*, "Hyperopt: A python library for optimizing the hyperparameters of machine learning algorithms," in *Proceedings of the 12th Python in science conference*, vol. 13. Citeseer, 2013.

[5] S. Daya, N. Van Duy, K. Eati, C. Ferreira, D. Glozic, V. Gucer, M. Gupta, S. Joshi, V. Lampkin, M. Martins *et al.*, *Microservices from Theory to Practice: Creating Applications in IBM Bluemix Using the Microservices Approach.* IBM Redbooks, 2016. [Online]. Available: https://books.google.com/books?id=eOZyCgAAQBAJ

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE transactions on evolutionary computation*, vol. 6, no. 2, pp. 182–197, 2002.

[7] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application," *arXiv preprint arXiv:2102.03827*, 2021.

[8] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135–146, 2016.

[9] B. Ghotra, S. McIntosh, and A. E. Hassan, "Revisiting the impact of classification techniques on the performance of defect prediction models," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 789–800.

[10] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng, "Service candidate identification from monolithic systems based on execution traces," *IEEE Transactions on Software Engineering*, 2019.

[11] A. K. Kalia, J. Xiao, C. Lin, S. Sinha, J. Rofrano, M. Vukovic, and D. Banerjee, "Mono2micro: an ai-based toolchain for evolving monolithic enterprise applications to a microservice architecture," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 1606–1610.

[12] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[13] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Software*, vol. 35, no. 3, pp. 96–100, 2018.

[14] G. Mazlami, J. Cito, and P. Leitner, "Extraction of microservices from monolithic software architectures," in *2017 IEEE International Conference on Web Services (ICWS)*. IEEE, 2017, pp. 524–531.

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

[16] N. Mittas and L. Angelis, "Ranking and clustering software cost estimation models through a multiple comparisons algorithm," *IEEE Transactions on software engineering*, vol. 39, no. 4, pp. 537–551, 2012.

[17] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using flash," *IEEE Transactions on Software Engineering*, vol. 46, no. 7, pp. 794–811, 2018.

[18] R. Yedida and T. Menzies, "On the value of oversampling for deep learning in software defect prediction," *IEEE Transactions on Software Engineering*, 2021.

[19] R. Yedida, X. Yang, and T. Menzies, "When simple is better than complex: A case study on deep learning for predicting bugzilla issue close time," *arXiv preprint arXiv:2101.06319*, 2021.