2022 IEEE International Conference on Service-Oriented System Engineering (SOSE) | 978-1-6654-7534-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/SOSE55356.2022.00007

Monolith to Microservices: VAE-Based GNN Approach with Duplication Consideration

Korn Sooksatra
Department of Computer Science
Baylor University
Waco, Texas, United States
Korn_Sooksatra1Baylor.edu

Rokin Maharjan
Department of Computer Science
Baylor University
Waco, Texas, United States
Rokin_Maharjan1@Baylor.edu

Tomas Cerny
Department of Computer Science
Baylor University
Waco, Texas, United States
Tomas_Cerny@Baylor.edu

***Abstract*—With the rise of cloud computing, many applications have been implemented into microservices to fully utilize cloud computing for scalability and maintainability purposes. However, there are some traditional monolith applications that developers would like to partition into microservices. Unfortunately, it is difficult to find a solution when considering multiple factors (i.e., the strong dependency in each cluster and how often different microservices communicate with each other). Further, because we allow duplications of classes in multiple microservices to reduce the communications between them, the number of duplicated classes is also another important factor for maintainability. Therefore, we need to use machine learning algorithms to approximate a good solution due to the infeasibility of finding the optimal solution. We apply the variational autoencoder to extract features of classes and use the fuzzy c means to group the classes into microservices according to their extracted features. As a result, our approach outperforms the other baselines in some significant metrics. Also, when we allow duplication, we find that it is helpful in terms of reducing the overhead of communications between microservices.**

***Index Terms*—Microservices, Machine learning, Clustering, Graph neural networks, Variational autoencoder**

I. Introduction

Despite advancements in cloud-native systems, many available web applications still follow the traditional architecture style, called a monolith. Monolith applications are composed of all the business functionalities in one place. Thus, it has essential drawbacks: scalability, maintainability and cost. When a monolith application is overloaded, we usually scale it up (e.g., increasing the space of its main memory and improving its CPU). These actions are costly and involve humans migrating the application from one server to another. We also can scale it out to mitigate the problem. However, we need to clone the whole application to other machines although the overloaded part is tiny in the application. Further, it is difficult to determine who is responsible for a specific functionality because all the functionalities are in the same place. It is not clear to specify who is responsible for which part. Therefore, we start using the idea of microservices for implementing web applications. The microservices architecture style designs an application as a group of services, each of which one team may take care of. Hence, it is trivial to maintain this kind of application.

Additionally, if any service is overloaded, we can scale out only that service. The microservices architecture is more scalable and easier to maintain than the monolith architecture. Also, Auer *et al.* [1] showed that maintainability and scalability are ones of the major reasons that many web application practitioners would like to choose the microservices architecture rather than the monolith architecture.

Even though microservices can provide several benefits, some web application practitioners have already implemented their monolith applications and would like to modify them to microservice-based applications. This problem is non-trivial since we do not have the best solution for this modification. Several attempts try to apply software engineering techniques and machine learning techniques for this problem. Nonetheless, we do not have the best technique so far. Although there are some approaches [2]–[7] trying to improve the microservices and detect some issues in them, it is better to generate good microservices in the first place after partitioning a monolith application.

Our work focuses on applying machine learning techniques because we need to solve a multi-objective problem for partitioning a monolith application into microservices. We consider some factors in this work: dependency, entrypoints and entrypoints co-existence. We obtain this information from static analysis, and we do not use inheritance relationships since the applications used in the experiments do not have them. Furthermore, we apply a graph neural network for this problem because we can transform the application into a graph. Moreover, our approach allows functionality in multiple microservices to address the dependency between microservices. Therefore, instead of k-means used in various works [8]–[10], we use the fuzzy c-means [11] for clustering to allow duplicated functionalities in multiple microservices. At last, we extensively construct experiments to compare the results from our approach to the state-of-the-art works. Then, we can list our contributions as follows:

- We propose the first machine-learning-based work that applies the variational autoencoder and fuzzy c means to partition a monolith application into microservices.
- This work allows duplication in multiple microservices where the previous machine-learning-based approaches did not pay attention to this.

2642-6587/22/$31.00 ©2022 IEEE
DOI 10.1109/SOSE55356.2022.00007

- We build experiments and show that our approach outperforms the state-of-the-art machine-learning-based approaches in multiple metrics.

This paper is organized as follows; Section II briefly describes some works that are close to our approach and shows the distinction of our work; Section III explains the problem of transferring a monolith application into microservices and formulates it in an optimization problem; Section IV summarizes our approach and demonstrates each component of our approach in detail; Section V describes how we construct the experiments and discusses their results; Section VI discusses the limitation of our approach, points out the future works and concludes everything.

II. Related Works

On these days, many works have paid attention to decomposing monolith applications into microservices to utilize cloud computing. Nonetheless, it is not trivial due to several factors (e.g., low cohesion between microservices). Then, software engineering techniques have been applied to help software architects decompose applications. However, when considering various factors, the problem became too complicated to be solved. Therefore, several works focused on utilizing machine learning algorithms to approximate the solution to this problem. Further, we describe the literature related to our software-engineering-based and machine-learning-based work.

A. Software-Engineering-Based Approach

There have been many different approaches to converting a monolith application to a microservice application. In 2017, Li, Chen, and Li [12] have implemented a dataflow-driven approach to decompose monolith applications into microservices where they first manually constructed a Data Flow Diagram (DFD). Then, they condensed the DFD into a decomposable DFD by combining the same operations with the same type of output data. Finally, microservice candidates were identified from the decomposable DFD. In 2019, Taibi and Systa [13] proposed a decomposition framework that utilized the static analysis (e.g., dependency graph) and dynamic analysis (i.e., process mining). Their framework included six steps: execution path analysis, frequency analysis of the execution paths, removal of circular dependencies, identification of decomposition options, metric-based ranking of the decomposition options and selection of decomposition solution. However, their framework was not automatic and still required some experts in some steps during the process. In 2020, Krause-Glau *et. al* [14] presented an approach to decompose monolith applications to microservices based on static and dynamic analysis of the monolith application. In this approach, they combined bounded-context pattern of domain-driven design with static code analysis and dynamic software visualization in order to identify the microservice boundaries. Firstly, they performed domain analysis and divided the application into smaller bounded contexts. Then, they performed static code analysis to partition the source code. Lastly, dynamic analysis was used for trace visualization. Auer $et.al$ [15], in 2021, have come up with an assessment framework for migrating a monolithic application to microservices. They surveyed industry professionals to identify metrics that someone would need to consider before and after making the transition. The assessment framework considered functional stability, performance efficiency, reliability, maintainability, and cost.

B. Machine-Learning-Based Approach

Since obtaining the optimal solution for partitioning a monolith application into microservices is not feasible, some existing works applied machine learning algorithms to this problem. Eski and Buzluca [16] proposed an automatic extraction approach. In particular, they applied the static analysis to an application and the dynamic analysis to log files. Then, they utilized the agglomerative hierarchical algorithm [17] for partitioning the application into microservices. They evaluated their approach by computing a similarity between their approach and references implemented by experts. In 2019, Abdullah *et al.* [18] partitioned a monolith application based on URIs and applied the k-means algorithm to clustering them with respect to their document sizes and response times. Then, they provided a virtual machine (VM) for each microservice by considering its document size and response time. After that, they proposed a simple auto-scaling algorithm for the VMs for dynamically scaling out overloaded VMs. In 2020, Kalia *et al.* [8] invented the Mono2Micro framework. This framework used the dynamic analysis to create a tree and then applied the static analysis to obtain information from the code. Then, they combined these results and, at last, applied the hierarchical clustering algorithm to partition the application into microservices.

In 2021, some researchers started utilizing a graph neural network (GNN) on this problem. Desai *et al.* [9] converted a monolith application to a graph by treating each class as a node and each dependency as an edge. They applied the graph convolution network (GCN) [19] on the generated graph. Then, they used an autoencoder to extract an embedding vector for every node and utilized the k-means algorithm to cluster the nodes with respect to their embedding vectors. Later, Mathai *et al.* [10] proposed a similar approach to [9]. Nonetheless, they used the heterogeneous graph neural network [20] for handling many types of nodes and edges. Then, Yedida *et al.* [21] attempted to optimize the hyperparameters of existing machine-learning-based monolith to microservices methods and used the work in [9] as an example.

III. Problem Formulation

We are given a monolith application consisting of n classes and would like to partition it into a group of classes in the same microservice. However, when the result is too much coarse-grain, we will face the same issue as in a monolith application as described in Section I. On the other hand, when the result is too fine-grain, the application will suffer from communication between microservices since many microservices may strongly depend on each other. Therefore, the first objective is that

all classes in a microservice strongly depend on each other. The second objective is that two microservices should not frequently communicate with each other. Further, since we allow duplicated classes in multiple microservices to easily achieve the second objective, the third objective is that the number of the duplicated classes should be minimized due to maintainability. Then, we can create a multi-objective optimization problem as

$$\min_{M} \quad \sum_{m\in M}\sum_{c_1\in m}\sum_{c_2\in m} d_1(c_1,c_2) + \lambda_1 \sum_{m_1\in M}\sum_{m_2\in M} d_2(m_1,m_2) + \lambda_2 \sum_{c\in C} d_3(c) \tag{1}$$

where M is a set of microservices, C is a set of classes, $d_1(c_1, c_2)$ is a distance function between class c_1 and c_2, $d_2(m_1, m_2)$ indicates the number of communication between microservice m_1 and m_2, $d_3(c)$ denotes the number of class c in the microservices and λ_1 and λ_2 are the balancers of the optimization problem. Explicitly, the optimal solution of this problem is not feasible. Thus, we desire to apply a machine-learning-based approach to determine a local minimum of this problem.

IV. Our Approach

As mentioned in the previous section that problem 1 is infeasible to solve, our approach utilizes machine learning and greedy algorithms to determine the solution. Our approach is based on the work in [9], and we describe it in this section. Fig. 1 illustrates the flow of our approach including three main steps. In the data preparation step, we use tools to extract useful information and create a dependency graph from the application. Then, we produce more information from the graph and preprocess this information to obtain a feature matrix. After that, we apply the graph convolution network to the information to extract embedding vectors for nodes in the graph in the embedding-vector creation step. At last, we utilize the fuzzy c-means algorithm to determine the microservices in the clustering step.

A. Data Preparation

To feed the machine learning model, we collected three kinds of data from a monolithic application: dependency graph, entrypoint existence matrix, and entrypoint co-existence matrix. Note that we denote P as the set of entrypoints. The dependency graph (A) is a matrix that lists all the classes a particular class is dependent upon. The entrypoint existence matrix (E) indicates which classes are in at least one path initiated by which entrypoints. Thus, E_{ij} is one when class i is in at least one path initiated by entrypoint j. The entrypoint co-existence matrix (Co) indicates how often two classes co-exist in the same paths initiated by the same entrypoints. That is, Co_{ij} is the number of paths where class i and class j co-exist in the same entrypoints.

We only considered web-application projects built on Spring Framework for this approach. We used JavaParser, a Java library that provides an Abstract Syntax Tree (AST) of our code. The AST can then be used to extract information from our code. To generate the dependency graph for a particular class, we first fetched a list of all the classes it had imported. Then, we filtered the list only to include the classes inside the project's root package and exclude the classes imported from external libraries.

To obtain the entrypoint existence matrix (E), for class i, we perform depth first search (DFS) in the dependency graph by having entrypoint j as the root. If we find class i at any point of DFS, we set E_{ij} to be one. Furthermore, to obtain the entrypoint co-existence matrix (Co), we also perform DFS in the dependency graph and list all the paths initiated by the entrypoints. Given class i and class j, we set Co_{ij} to be the number of paths where class i and class j exist. After that, we create a feature matrix ($\hat{X}$) as

$$\tilde{X} = E \odot Co$$

where $\cdot \odot \cdot$ is the concatenate operation. Thus, the size of $\hat{X}$ is $|C|$ by $|P| + |C|$. Then, with the graph convolutional network (GCN) technique, we normalize $\hat{X}$ according to the adjacency classes in the graph by

$$X = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\tilde{X}$$

where $\tilde{A} = A + I$, I is the identity matrix, $\tilde{D}$ is the degree diagonal matrix, and $\tilde{D}_{ii} = \sum_{j\in C}\tilde{A}_{ij}$.

B. Embedding-Vector Creation

Given the dependency graph A and feature matrix X from the previous step, we would like to find an embedding matrix extracted from X to use a similarity metric (i.e., L_2 norm) on two embedding vectors of two classes. Note that the higher the similarity, the more dependent the two classes become and the more similar workloads they have. We use the variational autoencoder (VAE) [22] for creating the embedding matrix because it has been proved that it could organize the latent vectors such that two similar vectors indicated two similar inputs. Specifically, X is an input of VAE, and $\hat{X}$ is the reconstruction of X. The latent space of VAE is the feature matrix Z where $Z \in \mathbb{R}^{n\times l}$ and l is the length of each embedding vector in Z. Hence, in this step, we train VAE such that it can output $\hat{X}$ that is as close to X as possible. We use the mean square error (MSE) to compute the reconstruction error between X and $\hat{X}$ because we do not want to pay much attention to tiny errors on some features. Furthermore, I would like to inject the graph architecture into the embedding matrix (Z). Hence, the embedding vector of a node needs to be similar to its neighbors. Also, according to VAE, we need to make sure that the embedding matrix is a normal distribution by using the Kullback-Leibler (KL) divergence. Therefore, we can formulate the loss function of VAE as

$$\sum_{i=1}^{n} ||X_i - \hat{X}_i||_2^2 + \sum_{i=1}^{n} ||A_i - Z_i Z^T||_2^2 + \mathrm{KL}(Z, p(Z)) \tag{2}$$

where $||\cdot||_2$ denotes the L_2 norm, n is the number of classes (i.e., nodes), X_i is row i of matrix X, $p(Z)$ is the prior

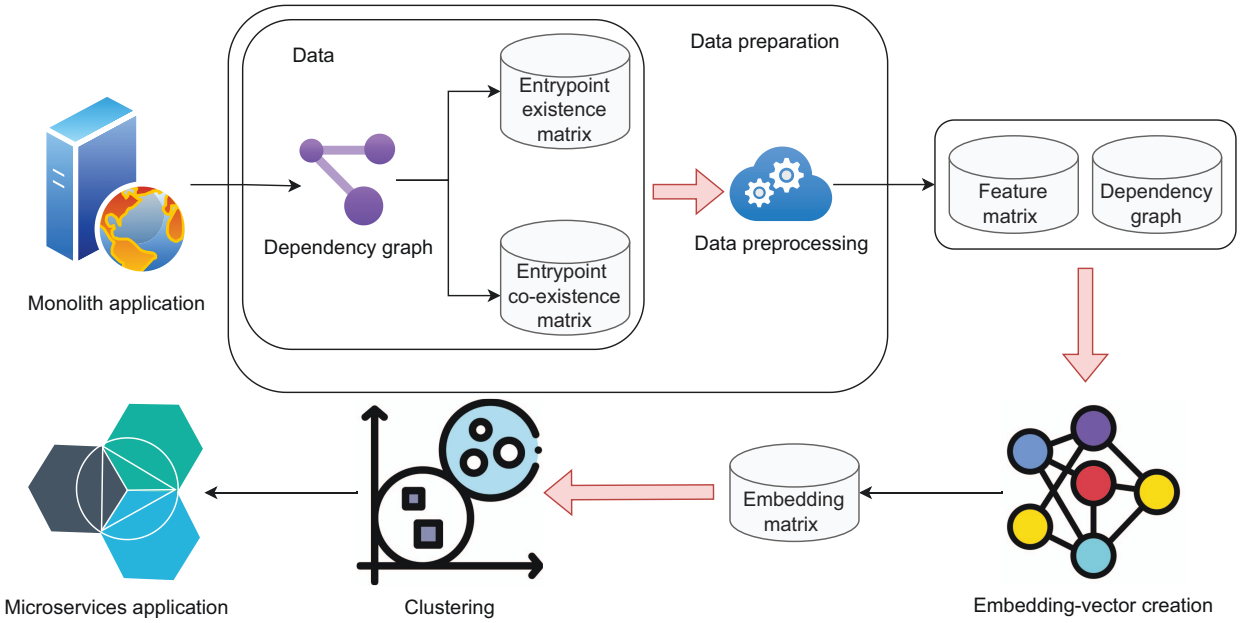


Fig. 1: The flow of our approach

distribution of Z where we assume that it is the normal distribution and $\mathrm{KL}(Z, p(Z))$ is the KL divergence function between Z and $p(Z)$. The first term is the reconstruction loss, and the second term is the architecture loss. The last term is the latent distribution loss. After obtaining Z from training VAE with respect to the loss, we use it in the next step for clustering.

C. Clustering

This step applies the fuzzy c-means algorithm on Z, and we then obtain the membership matrix (denoted by W) where $W \in [0,1]^{n \times k}$, k is the number of microservices (i.e., clusters) and $\sum_{j=1}^{k} W_{ij} = 1$ for every i. Note that the fuzzy c-means algorithm determines W and a centroid matrix (denoted by Γ where $\Gamma \in \mathbb{R}^{k \times p}$) by finding the local minimum of

$$\sum_{i=1}^{k} \sum_{j=1}^{n} W_{ji} ||Z_j - \Gamma_i||_2^2 \tag{3}$$

where W_{ji} is a membership of class j on microservice i. Then, we assign class i to the microservice on which the class has the most membership. Moreover, since we allow duplicated classes in multiple microservice, we also assign class i to other microservices, on which the class has memberships greater than the maintainability threshold. When the threshold is high, the number of redundancies is less. Also, when the threshold is 0.5, it is not different from the approach without duplication because for any class j, $\sum_{i=1}^{k} W_{ji} = 1$. Then, there is at most one microservice where a class has a membership greater than 0.5.

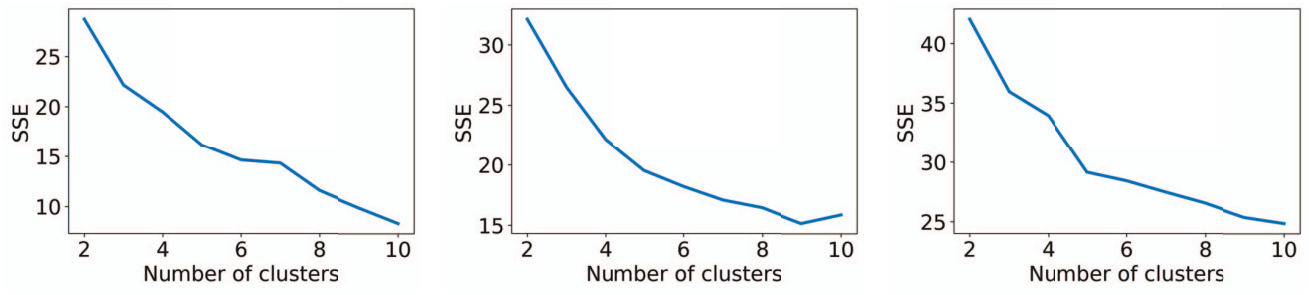
V. Experiments and Results

To evaluate our approach, we first pick datasets and baselines for comparison. The baselines are based on a state-of-the-art approach in the literature. Then, we construct the experiments to check if our approach can outperform the baselines in terms of some metrics described later. Also, when we allow duplication in the microservices, our approach can reduce the communication cost between microservices. After that, we show that the microservices generated by our approach are better than the ones generated by the baselines in Section V-E.

A. Datasets

We choose three monolith web applications that have been implemented by Spring framework: *Bearboard*[1], *Autocare*[2] and *Pharmacy*[3]. To justify the selection of our dataset applications, these applications have been developed by one of the coauthors. Hence, we are familiar with them and can provide reliable evaluation of the decomposition. Table I shows some information with respect to the datasets. We will partition the applications into five clusters because we plotted graphs according to sum square errors (SSE) for one to ten clusters. Then, we found that five clusters are the best number according to the Elbow method [23]. Fig. 2 shows SSE achieved by the fuzzy c means algorithm on those datasets and demonstrates that after the cluster size of five, SSE does not significantly decrease.

[1]https://github.com/rokinmaharjan/bear-board
[2]https://github.com/rokinmaharjan/autocare-nepal
[3]https://github.com/rokinmaharjan/pharmacy



(a) *Bearboard* (b) *Autocare* (c) *Pharmacy*

Fig. 2: Sum square errors (SSE) over the number of clusters generated by fuzzy c means

TABLE I: Information of our datasets

Dataset	# Classes	# Entrypoints	# Clusters
Bearboard	37	5	5
Autocare	47	4	5
Pharmacy	59	7	5

B. Models

This section discusses what models we will use for experiments and comparison. There are three models:

1) Autoencoder with kmeans (AE-K): This approach is the state-of-the-art work [9]. We use the autoencoder as the feature extractor and the kmeans algorithm for partitioning the monolith application.
2) Autoencoder with fuzzy c means (AE-C): We use the autoencoder as the feature extractor and the fuzzy c means algorithm for partitioning the monolith application.
3) Variational autoencoder with fuzzy c means (VAE-C): This is our approach. We use the variational autoencoder as the feature extractor and the fuzzy c means algorithm for partitioning the monolith application.

Table II shows the architecture of the embedding-vector creation part for each dataset. They are all composed of dense layers.

TABLE II: Architecture of the neural network part in each dataset. Note that each number is the number of neurons in each layer.

Dataset	Encoder	Decoder
Bearboard	42, 21, 10, 5, 2	2, 4, 8, 16, 42
Autocare	51, 25, 12, 6, 2	2, 4, 8 ,16, 51
Pharmacy	66, 33, 16, 8, 4, 2	2, 4, 8, 16, 32, 66

C. Metrics

To evaluate the quality of a cluster, human-related evaluation may be costly. Thus, we use two metrics used in the previous works instead. The first metric focuses on the structure of the cluster, and the second one pays attention to the size of each cluster.

1) Structural modularity (SM): This metric indicates how well the members in each cluster are related to each other, compared to the relationship outside the cluster. We define it as

$$\text{SM} = \frac{1}{k}\sum_{i=1}^{k}\frac{e_i}{n_i^2} - \frac{2}{k(k-1)}\sum_{i=1}^{k-1}\sum_{j=i+1}^{k}\frac{e_{i,j}}{2n_i n_j}$$

where k is the number of clusters, e_i is the number of edges inside cluster i, n_i is the number of classes inside cluster i and $e_{i,j}$ is the number of edges between cluster i and cluster j. We desire high SM for resulted clusters.
2) Non-Extreme Distribution (NED): This metric indicates how well the number of classes is in each cluster. We desire that each cluster does not have too many classes or too few classes. We can define this metric as

$$\text{NED} = \frac{1}{|C|}\sum_{i=1}^{k} a(5 \le n_i \le 20) n_i$$

where $a(x)$ is 1 when x is true and 0 otherwise, and C is a set of classes. Explicitly, we count only clusters that contain 5 to 20 classes. Therefore, we prefer to have high NED.
3) Interface number (IFN): This metric indicates the average number of interfaces in a microservice. Note that an interface means a class that other classes from other microservices depend on. Hence, the lower IFN becomes, the better microservices are produced.

D. Results

We construct two kinds of experiments. The first experiment involves partitioning monolith applications into microservices without duplication. Hence, the models used in this experiment are AE-K, AE-C and VAE-C. The second experiment partitions the applications into microservices by considering duplication. Then, the models evaluated in this experiment are AE-C and VAE-C because AE-K does not allow duplication. Note that the results from VAE-C are computed by the average of five attempts since VAE's latent space is based on the normal distribution.

TABLE III: SM, NED and IFN for each model in each dataset. Our approach (i.e., VAE-C) outperforms the other models.

Model	*Bearboard*			*Autocare*			*Pharmacy*		
	SM	NED	IFN	SM	NED	IFN	SM	NED	IFN
AE-K	-2.71	0.89	3.6	-6.14	0.49	4.8	-4.93	0.51	7.2
AE-C	-2.98	0.89	3.4	-3.51	0.51	4.8	-4.73	0.54	7.2
VAE-C	**-1.07**	**0.94**	**2.84**	**-1.81**	**1**	**4.3**	**-3.79**	**0.91**	**7.12**

1) No Duplication: Table III shows all the metrics in the experiments, and VAE-C (i.e., our approach) outperforms the other models in every datasets and every metric. Although VAE-C achieves almost the same NED as AE-C in *Bearboard*, it can have significantly higher NED in the other datasets than AE-C. This phenomenon indicates that VAE can organize the latent spaces of the classes better than AE.

2) Duplication: This experiment considers only SM and IFN because NED can exceed one when we allow duplication of classes in multiple microservices. Fig. 3 demonstrates that VAE-C outperforms AE-C in every maintainability threshold although it does not show the significant advantage in *Bearboard*. Noticeably, the lower maintainability can increase SM in VAE-C since the latent spaces from VAE can represent the features of the classes very well. Also, the trend shows that when we allow duplication, the approach tries to include the classes that are also relevant to the corresponding clusters due to the increase of SM. Therefore, allowing the duplication can significantly reduce the overhead of communications between clusters (i.e. microservices), and we can set the maintainability threshold according to how easy we would like to maintain the application. On the other hand, we cannot find any pattern of SM achieved by AE-C in the figure. Furthermore, Fig. 4 demonstrates that VAE-C could produce microservices that have lower interfaces than AE-C. Nevertheless, for *Autocare* dataset, with the maintainability threshold of 0.1, IFN of VAE-C is higher than AE-C because AE-C puts several duplicated irrelevant classes to the microservices.

E. Case Study: Bearboard

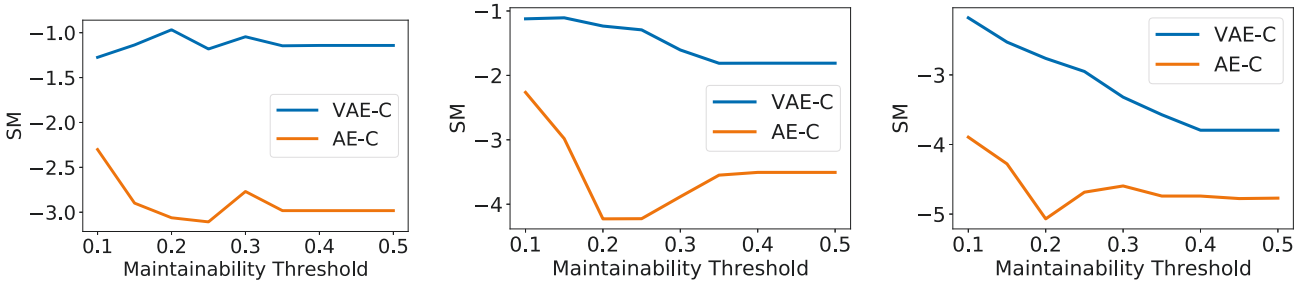
For this case study, co-author Rokin Maharjan did the analysis. He is a software engineer with more than five years of industry experience with monolithic applications and about 1.5 years of experience with microservices. We used the project *Bearboard* to perform the case study. *Bearboard* is a monolithic REST API application we built from scratch using Spring Framework. Since we know the ins and outs, it made sense to choose this project for this study. We used two metrics to judge the results of the different machine learning models: placement of all the necessary classes in every microservice including duplicated classes, and whether the microservices can run independently.

This case study will compare the results of three approaches: AE-K, VAE-C without duplication, and VAE-C with duplication with a maintainability threshold of 0.15 and 0.2. The AE-K approach decomposed the monolithic *Bearboard* application into five microservices fairly well as seen in Fig. 5, but there were a few misclassifications. For instance, the *RoleRepository* is a repository class that has methods to perform operations on the *Role* domain, which in turn is associated with the *User* domain. However, the *RoleRepository* has been placed in microservice 2 while the classes associated with *Role* and *User* have been placed in microservice 1. Similarly, microservice 4 has a *UserService* class while the other classes in the microservice are related to *Event*. Also, the utility class *CustomBeanUtils* is shared across multiple classes in the project. Nonetheless, it is only placed in a single microservice since this approach does not allow duplication of classes. Although the result is not perfect, we understand how the classes can be structured into different microservices.

In our first approach (i.e., the VAE-C without duplication), the result in Fig. 6 was comparable to the AE-K approach. Some of the microservices (e.g., microservice 1 and microservice 2) were self-sufficient and could be run independently because microservice 1 has all the *Whitelist* components, and microservice 2 has all the *Award* components. However, there were some misclassification issues and not replicating common classes. For instance, the classes associated with *User* and *Role* should be in the same microservice, but the classes *RoleRepository*, *UserController*, and *User* are spread across three microservices. Furthermore, the utility class *CustomBeanUtils* is used by both *AwardService* and *EventService*, but it is only placed in microservice 2 with the *Award* components. To solve these issues, we allowed duplication in the VAE-C with a maintainability threshold of 0.15. The result of this in Fig. 7 was far better than the previous two. Although some of the microservices had unnecessary classes, they had every class required for it to be self-sufficient and independently runnable. The common class *CustomBeanUtils* was also duplicated in all the microservices that required it. We also tested the VAE-C with a maintainability threshold of 0.2. The result of this is shown in Fig. 8. This resulted in a fewer number of unnecessary classes in each microservice. However, it failed to include some of the necessary classes for the microservice to be self-sustained and independently runnable. For example, microservice 4 lacked the *RoleRepository*, *Roles* enums, and *UserRepository* for the *User* module to be complete.

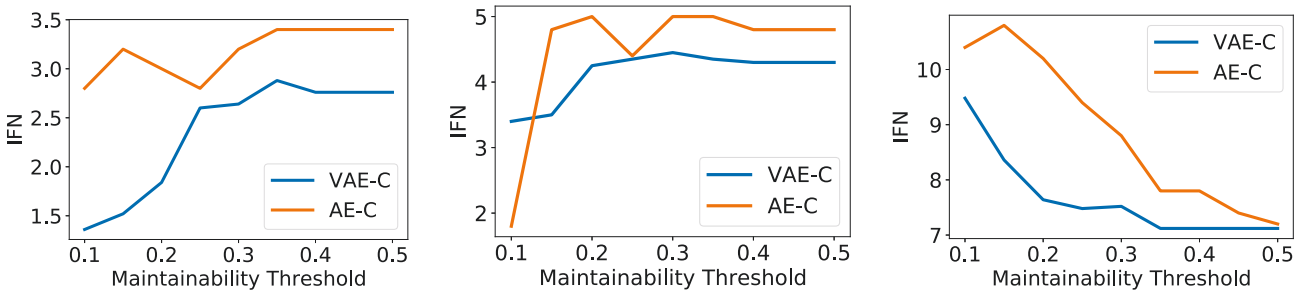
VI. Conclusion and Future Works

Throughout this work, we mathematically formulate the problem of partitioning a monolith application to microservices into a multi-objective optimization problem. This problem implies that it is infeasible to optimize it directly. Then, we design a machine-learning-based approach to solve the problem by applying the variational autoencoder and the fuzzy c means algorithm. As a result, our approach can outperform all the baselines including a state-of-the-art approach based on the autoencoder and the k means algorithm in terms of



(a) *Bearboard* (b) *Autocare* (c) *Pharmacy*

Fig. 3: Structural modularity (SM) over the maintainability thresholds by AE-C and VAE-C



(a) *Bearboard* (b) *Autocare* (c) *Pharmacy*

Fig. 4: Interface number (IFN) over the maintainability thresholds by AE-C and VAE-C

Microservice 1
"User", "UserRegistrationTest", "UserRepository", "UserController", "EventService", "Roles", "Role"

Microservice 2
"WebConfig", "PasswordUtils", "AwardController", "AwardService", "BaylorBoardApplicationTests", "AwardRepository", "RoleRepository", "TwitterConfig", "WebSocketConfig", "WebSecurityConfig", "CustomBeanUtils", "Award", "BaylorBoardApplication", "BadRequestException", "KafkaConfig"

Microservice 3
"Tweet", "TweetServiceTest", "TweetController", "TweetRepository", "TweetStatus"

Microservice 4
"EventController", "UserService", "Event", "EventRepository"

Microservice 5
"WhitelistUserRepository", "WhitelistUserService", "WhitelistUser", "WhitelistUserController", "TweetService"

Fig. 5: Microservices of *Bearboard* dataset generated by AE-K

Microservice 1
"WhitelistUserRepository", "WhitelistUserService", "WhitelistUser", "WhitelistUserController", "TweetService"

Microservice 2
"AwardController", "AwardService", "AwardRepository", "CustomBeanUtils", "Award"

Microservice 3
"WebConfig", "PasswordUtils", "UserRepository", "BaylorBoardApplicationTests", "RoleRepository", "TwitterConfig", "WebSocketConfig", "WebSecurityConfig", "BaylorBoardApplication", "BadRequestException", "KafkaConfig"

Microservice 4
"UserController", "EventController", "UserService", "EventService", "Event", "EventRepository"

Microservice 5
"User", "Tweet", "UserRegistrationTest", "TweetServiceTest", "TweetController", "TweetRepository", "TweetStatus", "Roles", "Role"

Fig. 6: Microservices of *Bearboard* dataset generated by VAE-C (i.e., our approach) without duplication

Microservice 1
"WhitelistUserRepository", "WhitelistUserService", "WhitelistUser", "WhitelistUserController", "TweetService", "BaylorBoardApplicationTests"

Microservice 2
"AwardController", "AwardService", "AwardRepository", "CustomBeanUtils", "Award", "WebConfig", "BaylorBoardApplicationTests"

Microservice 3
"WebConfig", "PasswordUtils", "UserRepository", "BaylorBoardApplicationTests", "RoleRepository", "TwitterConfig", "WebSocketConfig", "WebSecurityConfig", "BaylorBoardApplication", "BadRequestException", "KafkaConfig", "User", "UserRegistrationTest", "UserService", "Roles", "Role", "TweetService", "CustomBeanUtils"

Microservice 4
"UserController", "EventController", "UserService", "EventService", "Event", "EventRepository", "User", "UserRegistrationTest", "UserRepository", "RoleRepository", "Role", "CustomBeanUtils"

Microservice 5
"User", "Tweet", "UserRegistrationTest", "TweetServiceTest", "TweetController", "TweetRepository", "TweetStatus", "Roles", "Role", "TweetService"

Fig. 7: Microservices of *Bearboard* dataset generated by VAE-C (i.e., our approach) with duplication and the maintainability threshold of 0.15. Note that the classes in red are the ones added to the result of VAE-C without duplication.

Fig. 8: Microservices of *Bearboard* dataset generated by VAE-C (i.e., our approach) with duplication and the maintainability threshold of 0.2. Note that the classes in red are the ones added to the result of VAE-C without duplication.

structural modularity, non-extreme distribution and interface number in every dataset. Also, when we consider duplication, the structural modularity and the interface number are better when we reduce the maintainability threshold.

On the other hand, the approach with the autoencoder does not show this pattern due to its unorganized latent space. In addition, we show the microservices formed by the previous approach, our approach without duplication and our approach with duplication. We found that our approach can create more reasonable microservices than the previous one. Also, when we consider duplication, the microservices become more independent and self-contained than the one without considering duplication.

Despite the success of our approach, it still does not consider the dynamic analysis that can provide valuable data (e.g., document size and response time). With this data, we can partition an application better if we would like to perform auto-scaling when some microservices need to scale themselves because we can group classes that have similar loads together. Moreover, this approach still stays in theory. Therefore, we also plan to create an application that automatically makes actual microservices from a particular monolith application. We leave these ideas for future works.

VII. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research https://research.redhat.com.

References

[1] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106600, 2021.

[2] A. Bakhtin, A. Al Maruf, T. Cerny, and D. Taibi, "Survey on tools and techniques detecting microservice api patterns," in *IEEE International Conference on Services Computing (SCC)*, 2022.

[3] D. Taibi, B. Kehoe, and D. Poccia, "Serverless: From bad practices to good solutions," in *IEEE International Conference on Services Computing (SCC)*, 2022.

[4] T. Cerny and D. Taibi, "Static analysis tools in the era of cloud-native systems," in *4th International Conference on Microservices*, 2022.

[5] A. Al Maruf, A. Bakhtin, T. Cerny, and D. Taibi, "Using microservice telemetry data for system dynamic analysis," in *2022 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2022.

[6] T. Cerny, A. Abdelfattah, V. Bushong, A. A. Maruf, and D. Taibi, "Microvision: Static analysis-based approach to visualizing microservices in augmented reality," in *2022 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2022.

[7] ——, "Microservice architecture reconstruction and visualization techniques: A review," in *2022 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, 2022.

[8] 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.

[9] U. Desai, S. Bandyopadhyay, and S. Tamilselvam, "Graph neural network to dilute outliers for refactoring monolith application," in *Proceedings of 35th AAAI Conference on Artificial Intelligence (AAAI'21)*, 2021.

[10] A. Mathai, S. Bandyopadhyay, U. Desai, and S. Tamilselvam, "Monolith to microservices: Representing application software through heterogeneous gnn," *arXiv preprint arXiv:2112.01317*, 2021.
[11] J. C. Bezdek, R. Ehrlich, and W. Full, "Fcm: The fuzzy c-means clustering algorithm," *Computers & geosciences*, vol. 10, no. 2-3, pp. 191–203, 1984.
[12] R. Chen, S. Li, and Z. E. Li, "From monolith to microservices: A dataflow-driven approach," 12 2017, pp. 466–475.
[13] D. Taibi and K. Systä, "From monolithic systems to microservices: A decomposition framework based on process mining," 2019.
[14] A. Krause-Glau, C. Zirkelbach, W. Hasselbring, S. Lenga, and D. Kröger, "Microservice decomposition via static and dynamic analysis of the monolith," 03 2020.
[15] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106600, 04 2021.
[16] S. Eski and F. Buzluca, "An automatic extraction approach: Transition to microservices architecture from monolithic application," in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, 2018, pp. 1–6.
[17] W. H. Day and H. Edelsbrunner, "Efficient algorithms for agglomerative hierarchical clustering methods," *Journal of classification*, vol. 1, no. 1, pp. 7–24, 1984.
[18] M. Abdullah, W. Iqbal, and A. Erradi, "Unsupervised learning approach for web application auto-decomposition into microservices," *Journal of Systems and Software*, vol. 151, pp. 243–257, 2019.
[19] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.
[20] C. Zhang, D. Song, C. Huang, A. Swami, and N. V. Chawla, "Heterogeneous graph neural network," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019, pp. 793–803.
[21] R. Yedida, R. Krishna, A. Kalia, T. Menzies, J. Xiao, and M. Vukovic, "Partitioning cloud-based microservices (via deep learning)," *arXiv preprint arXiv:2109.14569*, 2021.
[22] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," *arXiv preprint arXiv:1312.6114*, 2013.
[23] P. Bholowalia and A. Kumar, "Ebk-means: A clustering technique based on elbow method and k-means in wsn," *International Journal of Computer Applications*, vol. 105, no. 9, 2014.