# Graph Neural Network to Dilute Outliers for Refactoring Monolith Application

**Utkarsh Desai, Sambaran Bandyopadhyay, Srikanth Tamilselvam**
IBM Research
udesai26@in.ibm.com, samb.bandyo@gmail.com, srikanth.tamilselvam@in.ibm.com

**Abstract**

Microservices are becoming the defacto design choice for software architecture. It involves partitioning the software components into finer modules such that the development can happen independently. It also provides natural benefits when deployed on the cloud since resources can be allocated dynamically to necessary components based on demand. Therefore, enterprises as part of their journey to cloud, are increasingly looking to refactor their monolith application into one or more candidate microservices; wherein each service contains a group of software entities (e.g., classes) that are responsible for a common functionality. Graphs are a natural choice to represent a software system. Each software entity can be represented as nodes and its dependencies with other entities as links. Therefore, this problem of refactoring can be viewed as a graph based clustering task. In this work, we propose a novel method to adapt the recent advancements in graph neural networks in the context of code to better understand the software and apply them in the clustering task. In that process, we also identify the outliers in the graph which can be directly mapped to top refactor candidates in the software. Our solution is able to improve state-of-the-art performance compared to works from both software engineering and existing graph representation based techniques.

## 1 Introduction

Microservices is an architectural style that structures an application as a set of smaller services [1]. These services are built around business functionalities and follow "Single Responsibility Principle"[2]. This allows the team to develop business functionalities independently. Also, they naturally benefit from cloud deployment due to the support for differential and dynamic addition of resources like CPU, memory, disk space to specific services based on the demand. However, there are lot of existing monolith applications currently in use that cannot fully tap these benefits due to their architecture style. Monoliths package all the business functionalities into a single deployable unit making them unsuitable to fully leverage cloud benefits. Therefore, there is a surge in enterprises wanting to refactor their monolith applications into microservices. This is done by mapping business functions onto the code structure and identifying the functional boundaries such that there are less dependencies across the services (Jin et al. 2019). In typical monoliths, there are classes (or programs) loaded with overlapping functionalities. We refer to such classes as outliers or refactorable candidates. They typically require top attention from the developers for modification during refactoring to make the microservices independent and deployable. But identifying functional boundaries on the existing code is a hard task (Gouigoux and Tamzalit 2017) and the effort gets multiplied when done without the help of original developers, which is typically the case with legacy applications.

In the software engineering community, the problem is often referred as software decomposition and several approaches (Fritzsch et al. 2018) have been proposed. The approaches range from process mining, genetic algorithms to graph based clustering. Graphs are a natural way to represent application implementation structure. The classes in the application can be considered as nodes and its interaction with the other classes can be considered as edges. Further, the nodes can carry multiple features based on their type and their invocation pattern. Figure 1 demonstrates the translation of an application into a graph. Therefore, the application refactoring problem can be viewed as a graph based clustering task. In the past, many clustering techniques have been applied on code (Shtern and Tzerpos 2012), but they often consider only the structural features of the application i.e the dependency of classes. Also, none of these approaches have looked into attributed graph networks or attempted to minimize the effect of outlier nodes during clustering.

Graph based mining tasks have received significant attention in recent years due to development of graph representation learning that maps the nodes of a graph to a vector space (Perozzi, Al-Rfou, and Skiena 2014; Hamilton, Ying, and Leskovec 2017). They have also been applied to a diverse set of applications such as social networks (Kipf and Welling 2017), drug discovery (Gilmer et al. 2017), traffic networks (Guo et al. 2019), etc. In this work, we propose a novel graph neural network based solution to refactor monolith applications into a desired number of microservices. The main contributions of our paper are listed below.

1. We propose a novel way to translate the application implementation structure to an attributed graph structure

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

[1] https://www.martinfowler.com/articles/microservices.html

[2] https://www.infoq.com/articles/microservices-intro

through static program analysis.

2. We introduce two types of outliers that reflect the top refactoring program candidates.
3. We propose a novel graph neural network (GNN), referred as CO-GCN[3] (Clustering and Outlier aware Graph Convolution Network), that unifies node representation, outlier node detection & dilution and node clustering in a sing framework for refactoring monolith applications.
4. We improve the state-of-the-art performance with respect to both software engineering and graph representation based techniques to refactor four publicly available monolith applications.

## 2 Related Work

Fritzsc et al. (Fritzsch et al. 2018) presented a survey on ten different approaches towards refactoring a monolith application into microservices. Of these, only four works were applied directly on application code and the rest used other application artefacts such as logs, commit histories, UML diagrams etc. However, all of these works have drawbacks since they either (1) focus on only structural features; or (2) propose partitions focusing more on technical layers which is not desirable (Taibi and Lenarduzzi 2018); or (3) partition only a subset of program files like EJBs in java. (Mazlami, Cito, and Leitner 2017) proposed a graph based clustering approach with a focus on version history. (Jin et al. 2019) proposed hierarchical clustering of program files, but requires access to the runtime behavior of application which is practically difficult. Moreover, these approaches do not exploit the power of representation learning and graph neural networks. Also, they do not recommend refactorable classes.

Graph representation learning (Hamilton, Ying, and Leskovec 2017) shows promising results on multiple downstream graph mining tasks. Graph neural networks (Wu et al. 2020) apply neural network directly on graphs. In Graph convolution networks introduced by (Kipf and Welling 2017), a localized first-order approximation of spectral graph convolutions is proposed and experimented for semi-supervised node classification. An unsupervised variant, GCN autoencoder is proposed in (Kipf and Welling 2016). GNNs are also proposed for supervised (Chen, Li, and Bruna 2019) and unsupervised community detection (Zhang et al. 2019) in graphs. Recently, a self-supervised learning based GNN, Deep Graph Infomax (DGI) (Veličković et al. 2019) is proposed for obtaining node representation using the principle of information maximization. Outlier nodes are present in any real-world graph and are shown to have adverse effect on the embeddings of regular nodes in a graph (Liang et al. 2018). Unsupervised algorithms to minimize the effect of outliers in the framework of graph representation learning are proposed recently (Bandyopadhyay, Lokesh, and Murty 2019; Bandyopadhyay et al. 2020; Bandyopadhyay, Vivek, and Murty 2020). However, minimizing the effect of outliers in the GNN framework has not been studied in the literature.

[3]Code available at: https://github.com/utkd/cogcn

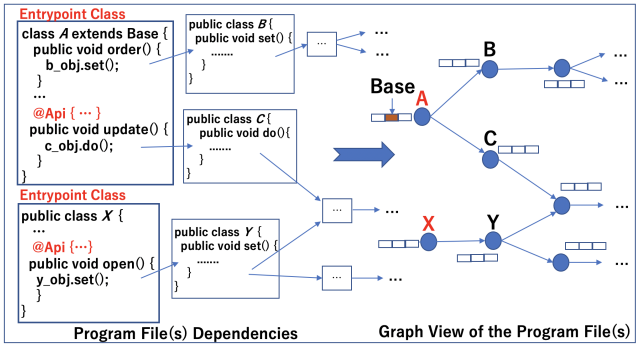


Figure 1: Representation of a sample Java application as graph: The method *order()* from class *A* invokes the method *set()* from class *B*, establishing a direct relation between the two classes. If we represent classes *A* and *B* as nodes in a graph, we can define a directed edge, $e(A, B)$ from *A* to *B*.

## 3 Methodology

Given a monolith application, we want to partition the monolith into $K$ clusters of classes, with $K$ provided by a subject matter expert (SME), where each cluster is a group of classes that perform a well-defined functionality. The clusters should exhibit high cohesion, i.e., have strong interaction within the cluster and low coupling i.e., less interaction between clusters. We also want to identify the following outlier classes from a monolith application (Bandyopadhyay, Lokesh, and Murty 2019) to be handled by an SME.

- *Structural Outlier*: A class which has high interaction with classes from different clusters.
- *Attribute Outlier*: A class which has attributes, such as usage patterns, similar to attributes from other clusters.

### 3.1 Converting Applications to Graph

We now describe our approach to represent an application as a graph, given it's source code. Consider a simple Java application comprising of multiple classes as shown in Figure 1. Each class in the application can be represented as a node in a graph. We denote the set of such nodes as $V$. We establish a *directed* edge from node A to node B if there is method in the class A that calls a method from class B. We perform static analysis[4] of the application code to identify all such method calls between classes and obtain a set of edges, $E$ between the corresponding nodes. The edges are unweighted and multiple method calls from class A to class B are still represented by a single edge from A to B.

We now describe the process to generate the attribute matrix, $X$, coressponding to the nodes $V$ of the graph. Most modern web applications expose multiple APIs that perform various functions. These APIs (UI elements in the case of a non web-based application) are referred to as EntryPoint Specifications (Dietrich, Gauthier, and Krishnan 2018), or simply, *Entrypoints* (EPs). The methods invoked through these APIs are specially annotated as such and are called *entrypoint methods* in this work. Figure 1 shows an example of

[4]https://github.com/soot-oss/soot

such methods annotated with *@API*. We refer to the classes containing such entrypoint methods as *entrypoint classes*. Each entrypoint class can thus be associated with multiple Entrypoints. Starting from an entrypoint method, we can follow the call sequence of methods through the application, keeping track of all classes invoked during the execution trace of that Entrypoint. If $P$ is the set of Entrypoints in an application, we can define a matrix $EP^{|V|\times|P|}$, such that $EP(i,p) = 1$ if class $i$ is present in the execution trace of entrypoint $p$, else 0. Additionally, we define $C^{|V|\times|V|}$ such that $C(i,j)$ is the number of Entrypoint execution traces that contain both classes $i$ and $j$. If a class is not invoked in an execution trace for any Entrypoint, we remove the corresponding node from the graph. Finally, classes may also inherit from other classes or Interfaces. In Figure 1, class *A* inherits from class *Base*. Although this establishes a dependency between the classes, it does not involve direct method invocation. Hence, we do not include this dependency as an edge in the graph, but as a node attribute. Therefore, we set $In(i,j) = In(j,i) = 1$ if classes $i$ and $j$ are related via an inheritance relationship and 0 otherwise. The attribute matrix $X$ is the concatenation of $EP$, $C$ and $In$ matrices. Thus, $X \in \mathbb{R}^{|V|\times F}$ where $F = |P| + 2|V|$. Each constituent of $X$ is row-normalized individually. The application can thus be represented as a graph $G = (V, E, X)$.

### 3.2 Proposed Graph Neural Network

Given the graph $G$, we want to develop a graph neural network which can jointly (i) derive vector representations (embeddings) of the nodes, (ii) minimize the effect of outlier nodes in the embeddings of other regular nodes, (iii) obtain communities in the graph. Let us use $A \in \mathbb{R}^{|V|\times|V|}$ to denote the adjacency matrix of $G$, where $A_{ij}$ is the weight of the edge $e_{ij}$ if it exists, otherwise $A_{ij} = 0$. We use a 2-layered graph convolution encoder (Kipf and Welling 2017) to obtain representation of each node as shown below:

$$Z = f(X, A) = \text{ReLU}(\hat{A}\ \text{ReLU}(\hat{A}XW^{(0)})\ W^{(1)}) \quad (1)$$

where each row of $Z \in \mathbb{R}^{|V|\times F'}$ contains the corresponding node representation. We compute $\tilde{A} = A + I$, where $I \in \mathbb{R}^{|V|\times|V|}$ is the identity matrix and the degree diagonal matrix $\tilde{D_{ii}}$ with $\tilde{D_{ii}} = \sum_{j\in V} \tilde{A}_{ij}$, $\forall i \in V$. We set $\hat{A} = \tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$. $W^{(0)}$ and $W^{(1)}$ are the trainable parameter matrices of GCN encoder. Traditionally, these parameters are trained on a node classification or link prediction loss (Kipf and Welling 2016) in a graph.

However, our objective in the work is to consider and minimize the effect of outlier nodes in the framework of graph convolution. We also want to do this in an unsupervised way as obtaining ground truth class labels and outlier information are extremely difficult for monolith applications. Towards this, we use the following GCN based decoder to map the $F'$ dimensional node embeddings to the input feature space.

$$\hat{X} = f(Z, A) = \text{ReLU}(\hat{A}\ \text{ReLU}(\hat{A}ZW^{(2)})\ W^{(3)}) \quad (2)$$

Here, $\hat{X} \in \mathbb{R}^F$, $W^{(2)}$ and $W^{(3)}$ are the trainable parameters of the decoder. Let us use $\mathcal{W} = \{W^{(0)}, \cdots, W^{(3)}\}$ to denote the parameters of the encoder and decoder combined. In the ideal world scenario when there is no outlier node present in a graph, one can train the parameters of the GCN autoencoder by directly minimizing some reconstruction loss. But as mentioned in Section 1, the presence of outliers in monolith applications is prevalent and if not handled properly, they can adversely affect the embeddings of regular regular nodes in a graph (Bandyopadhyay et al. 2020). To address them, we use the framework of multi-task learning where we design two loss components to detect structural and attribute outliers respectively. We denote structural and attribute outlierness (positive scalars) by $O_{si}$ and $O_{ai}$ respectively, for each node $i \in V$.

First, we ensure that presence of an edge should be preserved by the similarity of the two corresponding node embeddings in the vector space for the regular nodes. However, structural outliers being inconsistent in their link structure, do not necessarily follow this assumption. Hence, we design the following loss component which needs to be minimized with respect to the parameters of GCN and structural outlierness of the nodes:

$$\mathcal{L}_{str} = \sum_{i\in V} \log\Big(\frac{1}{O_{si}}\Big)||A_{i:} - ZZ_{i:}^T||_2^2 \quad (3)$$

Here, $A_{i:}$ is the $i$th row of the adjacency matrix and $Z_{i:}$ is the $i$th row (embedding of node $i$) of the node representation matrix. Clearly, higher the value of $O_{si}$, i.e., higher the outlierness of node $i$, less will be the value of $\log\Big(\frac{1}{O_{si}}\Big)$. Consequently, contribution of the structural outlier nodes in this loss component will be less. We also assume that total structural outlierness in a graph is bounded. So we set $\sum_{i\in V} O_{si} = 1$. Without such a bound, the optimization in Equation 3 would reach to a degenerate solution with each $O_{si}$ assigned to $+\infty$ at the infimum. We also tried replacing 1 with a hyperparameter $\mu$ as the bound, but that does not have much impact on the quality of the final solution.

Next, to preserve the impact of node attributes in the node representations, we want the reconstructed attributes in Equation 2 from the GCN decoder to match the initial node attributes for most of the regular nodes in the graph. However for attribute outliers, as their node attributes are significantly different from the attributes of their respective neighboring nodes, we reduce their contribution in the attribute reconstruction loss as follows:

$$\mathcal{L}_{att} = \sum_{i\in V} \log\Big(\frac{1}{O_{ai}}\Big)||X_{i:} - \hat{X}_{i:}||_2^2 \quad (4)$$

Here, $X$ and $\hat{X}$ are the given and reconstructed node feature matrices. Similar to the case of structural outlierness, nodes with more attribute outlierness score $O_{ai}$ would have less impact in Equation 4 and consequently the optimizer will be able to focus more on the regular nodes of the graph. Again, we assume that $O_{ai} > 0, \forall i \in V$ and $\sum_{i\in V} O_{ai} = 1$.

Minimizing the loss components in Equations 3 and 4 with respect to the parameters of GCN and outlier scores would be able to provide unsupervised node embeddings.

This will also detect the outlier nodes while minimize their negative impact on the other nodes of the graph. However as discussed in Section 1, our main goal in this work is to separate microservices within a monolith application. This needs discovering clusters of nodes (or communities) in the graph. One can potentially obtain the node embeddings first and then use a clustering algorithm (for example, k-means++ (Arthur and Vassilvitskii 2006)) as a post-processing step. But such a decoupled approach often leads to a suboptimal solution as shown in (Yang et al. 2017). Hence, we integrate node embedding, outlier detection and node clustering in a joint framework of graph neural network. To achieve this, we use the following loss to cluster the nodes in the graph, assuming their embeddings are already given.

$$\mathcal{L}_{clus} = \sum_{i=1}^{N}\sum_{k=1}^{K} M_{ik}||Z_{i:} - C_k||_2^2 \tag{5}$$

where $M \in \{0,1\}^{|V|\times K}$ is the binary cluster assignment matrix. We assume to know the number of clusters $K$. $M_{ik} = 1$ if node $i$ belongs to $k$th cluster and $M_{ik} = 0$ otherwise. $C_k \in \mathbb{R}^{F'}$ is the center of each cluster in the embedding space. Equation 5 needs to be minimized with respect to $M$ and $C = [C_1 \cdots C_K]^T$ to obtain the clustering. We call this method CO-GCN (Clustering and Outlier aware Graph Convolution Network) and the joint loss function is:

$$\min_{\mathcal{W},\mathcal{O},M,C} \mathcal{L}_{total} = \alpha_1\mathcal{L}_{str} + \alpha_2\mathcal{L}_{att} + \alpha_3\mathcal{L}_{clus} \tag{6}$$

$$\text{such that,} \quad \sum_{i\in V} O_{si} = \sum_{i\in V} O_{ai} = 1 \tag{7}$$

$$M \in \{0,1\}^{|V|\times K}, \;\; O_{si}, O_{ai} > 0 \;\; \forall i \in V \tag{8}$$

### 3.3 Optimization Procedure

The nature of the optimization problem in Eq. 6 is different with respect to different variables. We use alternate minimization technique, where we minimize the objective only with respect to one set of variables, keeping others fixed.

**Parameters of GCN** The set $\mathcal{W}$ contains all the parameters of the GCN encoder and decoder as described in Section 3. We use standard ADAM optimization technique (Kingma and Ba 2014) to minimize the total loss w.r.t. $\mathcal{W}$, keeping other variables fixed. We use an initial learning rate of $0.01$ and exponential decay rate of $0.95$ every $100$ iterations.

**Outliers** One can show that optimization in Equation 6 is convex with respect to each outlier variable when all other variables are fixed. This is because $0 < O_{si}, O_{ai} \leq 1, \forall i$ and $\log(\cdot)$ is a concave function and thus, $-\log(\cdot)$ is convex. Finally, L2 norms in both Equations 3 and 4 are non-negative. We aim to find the closed form update rules for the outlier terms to speed up the optimization process.

Taking the Lagrangian of Eq. 6 with respect to the constraint $\sum_{i\in V} O_{si} = 1$, we get (after ignoring terms that do not include $O_{si}$),

$$\frac{\partial}{\partial O_{si}} \sum_{j\in V} \log\Big(\frac{1}{O_{sj}}\Big)||A_{j:} - ZZ_{j:}^T||_2^2 + \lambda(\sum_{j\in V} O_{sj} - 1)$$

$\lambda \in \mathbb{R}$ is the Lagrangian constant. Equating the partial derivative w.r.t. $O_{si}$ to 0:

$$-\frac{||A_{i:} - ZZ_{i:}^T||_2^2}{O_{si}} + \lambda = 0, \Rightarrow O_{si} = \frac{||A_{i:} - ZZ_{i:}^T||_2^2}{\lambda}$$

But, $\sum_{j=1}^{N} O_{ji} = 1$ implies $\sum_{j\in V} \frac{||A_{j:} - ZZ_{j:}^T||_2^2}{\lambda} = 1$. Hence,

$$O_{si} = \frac{||A_{i:} - ZZ_{i:}^T||_2^2}{\sum_{j\in V} ||A_{j:} - ZZ_{j:}^T||_2^2} \tag{9}$$

The final update rule for structural outliers turns out to be quite intuitive. Our goal while deriving the loss in Equation 3 was to approximate adjacency structure of the graph by the similarity in the embedding space with outliers being discounted. The structural outlierness of a node in Equation 10 is proportional to the difference between the two after every iteration. In other words, if some node is not able to preserve its adjacency structure in the embedding space, it is more prone to be a structural outlier.

Similar to above, update rule for attribute outlier at each iteration can be derived to the following.

$$O_{ai} = \frac{||X_{i:} - \hat{X}_{i:}||_2^2}{\sum_{j\in V} ||X_{j:} - \hat{X}_{j:}||_2^2} \tag{10}$$

Because of the convexity of total loss in Equation 6 w.r.t. individual outlier scores, derivations of the update rules for outlier scores ensure the following lemma.

**Lemma 1** *Keeping all other variables fixed, the total loss in Equation 6 decreases after every update of the outlier scores by Equations 10 and 9 until it reaches to a stationary point.*

**Clustering Parameters** The total loss of CO-GCN also involves clustering parameters $M$ and $C$. While all other variables to be fixed, cluster assignment matrix $M$ can be obtained as:

$$M(i,k) = \begin{cases} 1, \text{ if } k = \underset{k'\in\{1,\cdots,K\}}{\text{argmin}} ||Z_i - C_{k'}||_2^2 \\ 0, \text{ Otherwise} \end{cases} \tag{11}$$

In the next step, $k$th row of cluster center matrix $C$ can be obtained as (Arthur and Vassilvitskii 2006):

$$C_k = \frac{1}{N_k}\sum_{i\in\mathcal{C}_k} Z_{i:} \tag{12}$$

where $\mathcal{C}_k = \{i \in V \mid M_{ik} = 1\}$ is the $k$-th cluster and $N_k = |\mathcal{C}_k|$ is the size of $k$-th cluster.

### 3.4 Pre-training, Algorithm and Analysis

To run CO-GCN, we first pre-train the GCN encoder and decoder by minimizing $\mathcal{L}_{str}$ and $\mathcal{L}_{att}$ in Equations 3 and 4 respectively, initializing $O_{si}, O_{ai}\ \forall i \in V$ to uniform values. We also use k-means++ (Arthur and Vassilvitskii 2006) to initialize the cluster assignment and cluster center matrices. Then over iterations, we sequentially solve $\mathcal{L}_{total}$ by alternating minimization technique described in Section 3.3 with respect to different variables. Overall procedure of CO-GCN is presented in Algorithm 1.

**Algorithm 1 CO-GCN**

**Input**: Class dependencies and Entrypoint definitions
1: Convert the application to a graph representation as defined in Section 3.1 and obtain the $V$, $E$ and $X$
2: Initialize outlier scores $O_{si}$ and $O_{ai}$ uniformly $\forall i \in V$.
3: Pre-train the GCN encoder and decoder
4: Use k-means++ to initialize the cluster assignments and cluster centers
5: **for** T iterations **do**
6: Update outlier scores $\mathcal{O}$ by Eq. 10 and 9.
7: Update cluster assignment and centers by Eq. 11 and 12
8: Update the parameters by GCN encoder and decoder by minimizing Eq. 6 using ADAM.
9: **end for**
**Output**: Cluster assignment matrix $M$, Cluster center matrix $C$ and the outlier scores $\mathcal{O}$

| Dataset | Description | Lang | # Class | C.Size |
|---|---|---|---|---|
| DayTrader | Trading App | Java | 111 | 8 |
| PBW | Plant Store | Java | 36 | 6 |
| Acme-Air | Airline App | Java | 38 | 4 |
| DietApp | DietTracker | C# | 32 | 5 |

Table 1: Details about the monolith applications studied

**Time Complexity** Time taken by GCN encoder and decoder is $O(|E|FF')$. Updating each value of outlier score takes $O(NF')$ and the total time to update all outlier scores is $O(N^2F')$. Updating the parameters of cluster assignment and cluster center matrices takes $O(NF'K)$ time. Thus, each iteration of CO-GCN takes $O(|E|FF' + N^2F' + NF'K)$. The outlier update rules although expensive, converge quickly because of the closed-form solution and theoretical guarantee (Lemma 1). Also, for most real-world monolith applications, number of classes is not very large (in 1000s). So the quadratic dependency of the runtime on the number of classes is not a bottleneck. However, one can try negative sampling approaches (Goldberg and Levy 2014) to approximate the similarity between the embeddings in the outlier update rules for other applications if needed.

# 4 Experimental Evaluation

## 4.1 Datasets (Monolith Applications) Used

To study the effectiveness of our approach, we chose four publicly-available web-based monolith applications namely Daytrader [5], Plantsbywebsphere [6], Acme-Air[7], Diet App[8]. They vary in programming languages, technologies, objectives and complexity in terms of lines of code, function sizes etc. Details of the monoliths are provided in Table 1.

## 4.2 Metrics

To evaluate the quality of the clusters identified as microservice candidates, we define four metrics. The first two aim to capture the structural quality of the clusters recommended as microservices and are the primary metrics in the evaluation. The other two metrics define additional properties of the clusters that are desirable.

1. **Modularity**: Modularity is a commonly used metric to evaluate the quality of clusters in a graph (Newman and Girvan 2004)(Newman 2006). It measures the fraction of edges of the graph between members of the same cluster relative to that of the same partition members but randomly generated graph edges. Higher values of Modularity indicate a stronger community structure.
2. **Structural Modularity**: An alternate measure of structural soundness of a cluster that is more suited to software applications is defined in (Jin et al. 2019). Structural Modularity, (SM) is defined as

$$SM = \frac{1}{K}\sum_{k=1}^{K}\frac{u_k}{N_k^2} - \frac{1}{K(K-1)/2}\sum_{k_1\neq k_2}^{K}\frac{\sigma_{k_1,k_2}}{2(N_{k_1}N_{k_2})}$$

and $u_k$ is the number of edges that lie completely within a cluster $k$, $\sigma_{k_1,k_2}$ is the number of edges between cluster $k_1$ and cluster $k_2$. $N_{k_1}$ and $N_{k_2}$ are the number of members in clusters $k_1$ and $k_2$ respectively.
3. **Non-Extreme Distribution(NED)**: It is desired that a microservice may not have too many or too few classes. We therefore measure how evenly distributed the sizes of the recommended clusters are as

$$NED = \frac{\sum_{k=1,k\ not\ extreme}^{K} n_k}{|V|}$$

$n_k$ is the number of classes in cluster $k$ and $V$ is the set of classes. $k$ is not extreme if it's size is within bounds of $\{5, 20\}$. $NED$ captures the architectural soundness of the clusters (Wu, Hassan, and Holt 2005)(Bittencourt and Guerrero 2009). For better interpretability, we measure $1 - NED$ and lower values are favorable.
4. **Interface Number(IFN)**: As defined in (Jin et al. 2019), this is the average number of published interfaces of a microservices partitioning.

$$IFN = \frac{1}{K}\sum_{k=1}^{K} ifn_k, \;\; ifn_k = |I_k|$$

where $I_k$ is the number of published interfaces in the microservice $k$ and $K$ is the number of such micorservices. We define a published interface as any class in the microservice that is referenced by another class from a different microservice. Lower values of $IFN$ are preferred.

## 4.3 Experimental Setup and Baselines

For each application in Table 1, we generate the adjacency matrix, $A$ and the feature matrix, $X$. The CO-GCN encoder comprises of two layers with dimensionality 64 and 32. The decoder consists of one layer of size 64 followed by another of the appropriate feature dimension. We pretrain for 250 iterations and set $T = 500$ in Algorithm 1. The final values of $M(i, k)$ are used as the cluster assignments from our algorithm. We set $\{\alpha_1, \alpha_2, \alpha_3\} = \{0.1, 0.1, 0.8\}$ in Eq. 6.

[5]https://github.com/WASdev/sample.daytrader7
[6]https://github.com/WASdev/sample.plantsbywebsphere
[7]https://github.com/acmeair/acmeair
[8]https://github.com/SebastianBienert/DietApp/

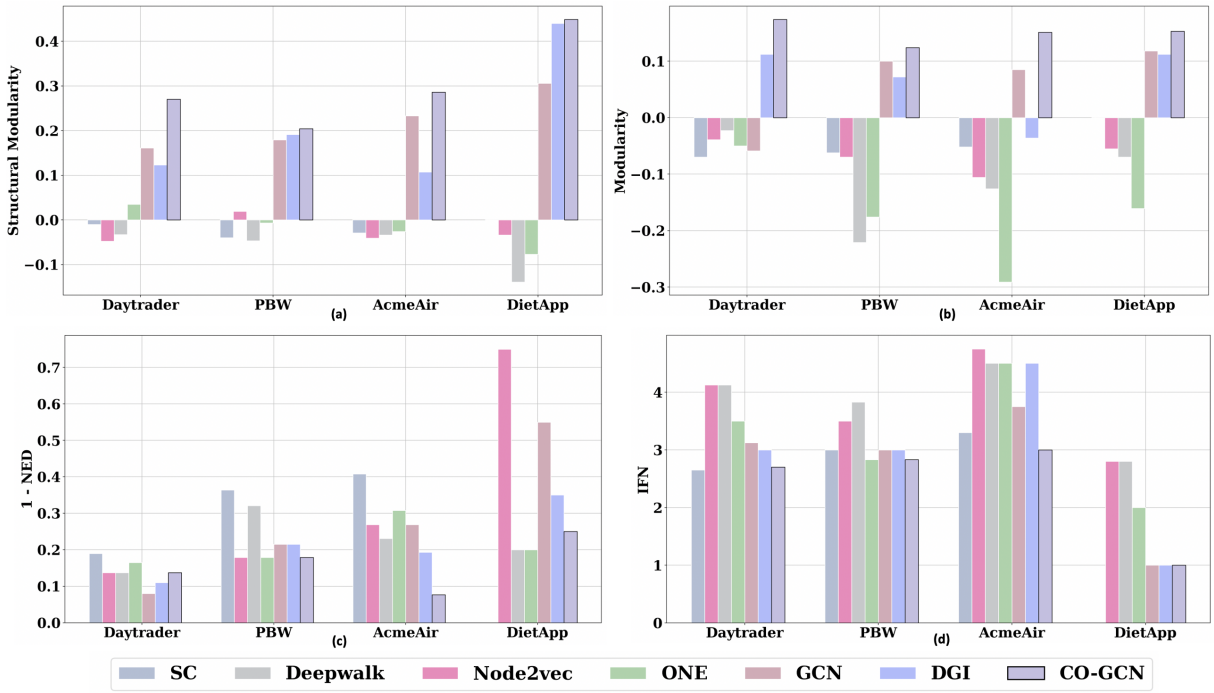


Figure 2: Comparison of the CO-GCN method with the baselines across the four applications on the (a) Structural Modularity (b) Modularity (c) 1-NED and (d) IFN metrics. The CO-GCN method clearly outperforms the baselines considered.

We evaluate our approach against multiple unsupervised baselines for learning node representations: Deepwalk (Perozzi, Al-Rfou, and Skiena 2014), Node2vec (Grover and Leskovec 2016), ONE (Bandyopadhyay, Lokesh, and Murty 2019) GCN (Kipf and Welling 2016) and DGI (Veličković et al. 2019). Among these, ONE accounts for the effects of outliers in learning node embeddings. For all our experiments, we set the size of the node embeddings to be 32. We use k-means++ algorithm on the embeddings generated by these baselines to obtain clusters. K is carefully chosen based on online sources and SME inputs.In contrast to these representation learning based baselines, the method of (Mazlami, Cito, and Leitner 2017) is a state-of-the-art approach for extracting microservices from a monolith application. This leverages Semantic Coupling (SC) information with graph partitioning to identify clusters. We also use it as a baseline. Since the implementation for the SC method does not support .Net applications, we do not use it for DietApp.

## 4.4 Results on Separating Micro Services

Figure 2 shows the metrics values on all four application for the evaluated methods. The three attributed graph neural network based methods (GCN, DGI and CO-GCN) outperform the rest of the methods by a significant margin. The CO-GCN method consistently achieves better modularity and structural modularity scores which clearly validates the inclusion of outlier and clustering objectives in the training. The CO-GCN method also achieves better $NED$ and $IFN$ scores in most cases. Another interesting observation is the negative scores for many of the baseline methods. This implies that are many inter-cluster edges for the clusters recommended by these methods, hinting at the fact that monolith applications may have several high-traffic nodes and assigning them to appropriate clusters is difficult, but critical. Figure 3 shows the identified clusters for the PBW application and our manual annotations to highlight the functionalities

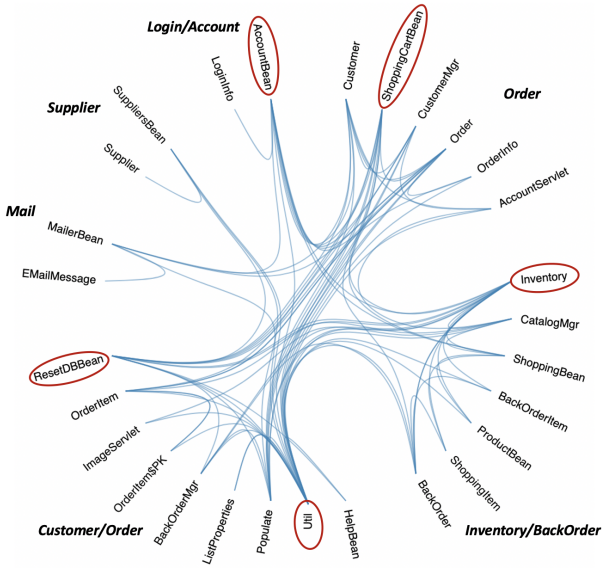


Figure 3: Clusters and top 5 outliers identified for the PBW application, with manual labels about their functionality.

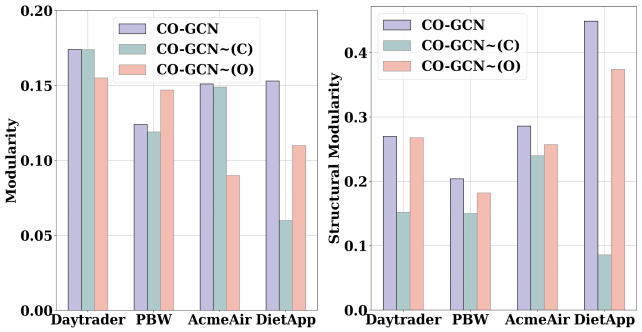


Figure 4: Results from the ablation study on the structural modularity and modularity metrics across the applications

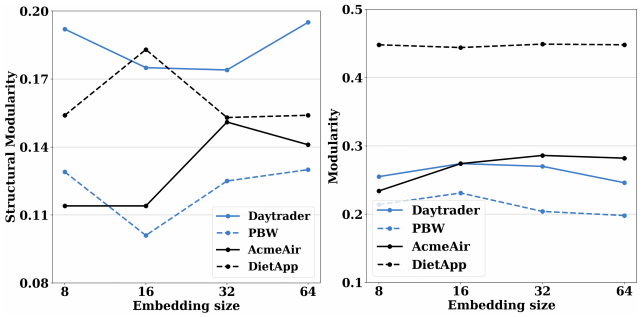


Figure 5: Sensitivity analysis on embedding size

offered. We can notice the clear distinction of functionalities

### 4.5 Detecting Outliers

The values of $O_{si}$ and $O_{ai}$ at the end of training represent the final outlier scores of each node. The ranked list of outlier nodes represents the top candidates for refactoring as part of microservices decomposition. Figure 3 highlights the combined top 5 outliers detected (across structural and attribute outlier scores) for PBW application by our approach. Among the baselines, we report outlier detection results only for GCN and DGI as they performed good for obtaining microservices. As GCN and DGI do not output outliers directly, we use Isolation forest (Liu, Ting, and Zhou 2008) on the embeddings generated by them to detect outliers.

To study the goodness of the outliers, we performed a qualitative study with five software engineers who have minimum seven years industrial experience. We randomly presented them with two out of the four monoliths and shared their code repositories. We asked them to rank the top five refactor candidate classes and compared them with the outliers identified by GCN, DGI and CO-GCN. On an average, the top five outliers provided by the annotators overlapped with our approach by 60%, GCN by 45% and DGI by 55%. We can conclude that the outliers identified by our approach are more relevant. The low overlap numbers indicate the highly difficult and subjective nature of this task.

### 4.6 Ablation and Sensitivity Analysis

We perform another set of experiments to measure the usefulness of individual components of CO-GCN.

1. We remove the clustering objective from $\mathcal{L}_{total}$., i.e., set $\alpha_3 = 0$ in Equation 6. Comparing the performance of this variant with CO-GCN shows marginal contribution of integrating the clustering loss. We denote this variant as CO-GCN$_\sim$(C). We use k-means++ on the node embeddings generated by this approach to obtain the clusters.
2. We remove the effect of the $O_{si}$ and $O_{ai}$ on $\mathcal{L}_{str}$ and $\mathcal{L}_{att}$ respectively, by removing the $\log(\cdot)$ terms. This is equivalent to traditional link and attribute reconstruction, with the clustering loss $\mathcal{L}_{clus}$. The goal is to evaluate the usefulness of minimizing the effect of outliers for identifying good clusters. We denote this variant as CO-GCN$_\sim$(O).

The results of the ablation study are shown in Figure 4. In general, incorporating outlier scores and the clustering objective does result in higher modularity and structural modularity scores. However, the degree to which these components contribute to the overall clustering quality vary for each application and the metric used. For instance, in the Daytrader application, removing the clustering objective reduces structural modularity significantly, but has no effect on modularity. Conversely, removing the outlier information reduces the modularity score, but has negligible effect on structural modularity. This effect is also visible in the other applications. Interestingly, removing the outlier information leads to improved modularity for PBW, but this is balanced by a reduced structural modularity score. We can still conclude that including the outlier scores and clustering loss in the training objective improves cluster quality in general.

Finally, we also evaluate the effect of the node embeddings size on the modularity and structural modularity values for each application. We experiment with embedding sizes in $\{8, 16, 32, 64\}$. The results are presented in Figure 5. We notice the modularity scores do not have any significant variation with a change in node embedding size. There is relatively more variation in the structural modularity scores with change in embedding sizes and once again, this variation is application dependent. There is not enough evidence to make any substantial claims, but in general, the performance seems to be better at higher embedding sizes.

## 5 Conclusion

We introduced the traditional software engineering problem of monolith to microservices decomposition as a clustering task leveraging graph representation learning. We showed how the application implementation structure can be translated into an attributed graph network. We then proposed a multi-objective Graph Convolution Network (GCN) based novel framework to not just generate candidate microservices, but also identified the outliers in the graph which can be considered as the important refactor classes for the architect to focus on. Our approach improved state of the art on multiple metrics from both graph and software engineering literature and performed well in human evaluation for the outlier detection. In future, we want to extend this work to automatically identify the number of microservices and expand to procedural programming languages like COBOL.

## Acknowledgements

We would like to thank Giriprasad Sridhara, Amith Singhee, Shivali Agarwal, Raunak Sinha from IBM India Research Labs and Yasu Kastuno, Ai Ishida, Aki Tozawa, Fumiko Satoh from IBM Tokyo Research Labs for their insightful suggestions during this work and valuable feedback towards improving the paper.

## Ethical Impact

We believe this work doesn't have any direct societal or ethical impact.

## References

Arthur, D.; and Vassilvitskii, S. 2006. k-means++: The advantages of careful seeding. Technical report, Stanford.

Bandyopadhyay, S.; Lokesh, N.; and Murty, M. N. 2019. Outlier aware network embedding for attributed networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 12–19.

Bandyopadhyay, S.; Lokesh, N.; Vivek, S. V.; and Murty, M. 2020. Outlier Resistant Unsupervised Deep Architectures for Attributed Network Embedding. In *Proceedings of the 13th International Conference on Web Search and Data Mining*, 25–33.

Bandyopadhyay, S.; Vivek, S. V.; and Murty, M. N. 2020. Integrating Network Embedding and Community Outlier Detection via Multiclass Graph Description. In *ECAI 2020 - 24th European Conference on Artificial Intelligence, 29 August-8 September 2020, Santiago de Compostela, Spain*, 976–983. IOS Press. doi:10.3233/FAIA200191.

Bittencourt, R. A.; and Guerrero, D. D. S. 2009. Comparison of graph clustering algorithms for recovering software architecture module views. In *2009 13th European Conference on Software Maintenance and Reengineering*, 251–254. IEEE.

Chen, Z.; Li, L.; and Bruna, J. 2019. Supervised Community Detection with Line Graph Neural Networks. In *International Conference on Learning Representations*. URL https://openreview.net/forum?id=H1g0Z3A9Fm.

Dietrich, J.; Gauthier, F.; and Krishnan, P. 2018. Driver Generation for Java EE Web Applications. In *2018 25th Australasian Software Engineering Conference (ASWEC)*, 121–125. IEEE.

Fritzsch, J.; Bogner, J.; Zimmermann, A.; and Wagner, S. 2018. From Monolith to Microservices: A Classification of Refactoring Approaches. *CoRR* abs/1807.10059. URL http://arxiv.org/abs/1807.10059.

Gilmer, J.; Schoenholz, S. S.; Riley, P. F.; Vinyals, O.; and Dahl, G. E. 2017. Neural message passing for Quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, 1263–1272.

Goldberg, Y.; and Levy, O. 2014. word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method. *arXiv preprint arXiv:1402.3722* .

Gouigoux, J.-P.; and Tamzalit, D. 2017. From monolith to microservices: Lessons learned on an industrial migration to a web oriented architecture. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 62–65. IEEE.

Grover, A.; and Leskovec, J. 2016. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, 855–864.

Guo, S.; Lin, Y.; Feng, N.; Song, C.; and Wan, H. 2019. Attention based spatial-temporal graph convolutional networks for traffic flow forecasting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 922–929.

Hamilton, W. L.; Ying, R.; and Leskovec, J. 2017. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584* .

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

Kingma, D. P.; and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* .

Kipf, T. N.; and Welling, M. 2016. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308* .

Kipf, T. N.; and Welling, M. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. URL https://openreview.net/forum?id=SJU4ayYgl.

Liang, J.; Jacobs, P.; Sun, J.; and Parthasarathy, S. 2018. Semi-supervised embedding in attributed networks with outliers. In *Proceedings of the 2018 SIAM International Conference on Data Mining*, 153–161. SIAM.

Liu, F. T.; Ting, K. M.; and Zhou, Z.-H. 2008. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, 413–422. IEEE.

Mazlami, G.; Cito, J.; and Leitner, P. 2017. Extraction of Microservices from Monolithic Software Architectures. In *2017 IEEE International Conference on Web Services (ICWS)*, 524–531.

Newman, M. E. 2006. Modularity and community structure in networks. *Proceedings of the national academy of sciences* 103(23): 8577–8582.

Newman, M. E.; and Girvan, M. 2004. Finding and evaluating community structure in networks. *Physical review E* 69(2): 026113.

Perozzi, B.; Al-Rfou, R.; and Skiena, S. 2014. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 701–710.

Shtern, M.; and Tzerpos, V. 2012. Clustering Methodologies for Software Engineering. *Adv. Soft. Eng.* 2012. ISSN 1687-8655. doi:10.1155/2012/792024. URL https://doi.org/10.1155/2012/792024.

Taibi, D.; and Lenarduzzi, V. 2018. On the Definition of Microservice Bad Smells. *IEEE Software* 35(3): 56–62.

Veličković, P.; Fedus, W.; Hamilton, W. L.; Liò, P.; Bengio, Y.; and Hjelm, R. D. 2019. Deep graph infomax. In *International Conference on Learning Representations*. URL https://openreview.net/forum?id=rklz9iAcKQ.

Wu, J.; Hassan, A. E.; and Holt, R. C. 2005. Comparison of clustering algorithms in the context of software evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 525–535. IEEE.

Wu, Z.; Pan, S.; Chen, F.; Long, G.; Zhang, C.; and Philip, S. Y. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* .

Yang, B.; Fu, X.; Sidiropoulos, N. D.; and Hong, M. 2017. Towards k-means-friendly spaces: Simultaneous deep learning and clustering. In *international conference on machine learning*, 3861–3870. PMLR.

Zhang, X.; Liu, H.; Li, Q.; and Wu, X.-M. 2019. Attributed graph clustering via adaptive graph convolution. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, 4327–4333. AAAI Press.