B-MEG: Bottlenecked-Microservices Extraction Using Graph Neural Networks

Gagan Somashekar, Anurag Dutt, Rohith Vaddavalli, Sai Bhargav Varanasi, Anshul Gandhi
PACE Lab, Stony Brook University
Stony Brook, New York, USA
{gsomashekar,adutt,rvaddavalli,sbvaranasi,anshul}@cs.stonybrook.edu

ABSTRACT

The microservices architecture enables independent development and maintenance of application components through its fine-grained and modular design. This has enabled rapid adoption of microservices architecture to build latency-sensitive online applications. In such online applications, it is critical to detect and mitigate sources of performance degradation (bottlenecks). However, the modular design of microservices architecture leads to a large graph of interacting microservices whose influence on each other is non-trivial. In this preliminary work, we explore the effectiveness of Graph Neural Network models in detecting bottlenecks. Preliminary analysis shows that our framework, B-MEG, produces promising results, especially for applications with complex call graphs. B-MEG shows up to 15% and 14% improvements in accuracy and precision, respectively, and close to 10× increase in recall for detecting bottlenecks compared to the technique used in existing work for bottleneck detection in microservices [32].

KEYWORDS

microservices, anomaly detection, bottleneck detection, graph neural networks, dataset

ACM Reference Format:
Gagan Somashekar, Anurag Dutt, Rohith Vaddavalli, Sai Bhargav Varanasi, Anshul Gandhi. 2022. B-MEG: Bottlenecked-Microservices Extraction Using Graph Neural Networks. In *Companion of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22 Companion), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3491204.3527494

1 INTRODUCTION

The microservices architecture is an architectural style that allows applications to be decomposed into fine-grained, modular, and interacting services, called *microservices*. Under this architecture, each microservice can be independently designed, thereby enabling independent development, maintenance, scaling, and fault isolation (at the level of microservices) [14]. These benefits make the microservices architecture well suited for designing online, customer-facing applications where performance and availability are critical [11, 12].

ICPE '22 Companion, April 9–13, 2022, Bejing, China

ACM ISBN 978-1-4503-9159-7/22/04…$15.00
https://doi.org/10.1145/3491204.3527494

Detecting and mitigating performance bottlenecks in online applications is crucial to provide a good customer experience [6, 12]. Long tail latencies that significantly affect the revenues of online applications are often a result of *performance bottlenecks* that do not necessarily lead to errors or faults and instead arise due to resource saturation, resource contention, or microservices application misconfiguration [14, 15, 32, 38]. Regardless of the underlying cause of performance bottlenecks, it is essential to have a technique that quickly adapts to dynamic online workloads and accurately detects bottlenecks with high recall and precision. A low recall is especially problematic as it implies that performance issues go unaddressed.

Microservices architecture has unique characteristics compared to other architectural styles that complicates bottleneck detection:

- While the modular architecture allows isolating performance issues at the level of individual microservices, the complex interaction *between* microservices leads to back-pressure effects and cascading performance degradation, making it difficult to precisely pinpoint the performance bottleneck(s) [30].
- Employing data-driven approaches that can learn such complex interactions is difficult due to scarcity of labeled data for bottlenecked class in production systems [15].
- Frequent software updates, and components like caches, message queues, etc., which are inherent to microservices architecture, lead to time-varying interactions between microservices [26, 32] necessitating a technique that can generalize to such dynamicity.

For applications implemented using monolithic or multi-tier architecture, the problem of bottleneck detection has been studied extensively [3–5, 8, 18, 33, 34, 39, 41, 42]; these studies continue to influence bottleneck detection research for microservices. For the microservices architecture, a popular approach to detect bottlenecks is to employ end-to-end distributed tracing systems like Jaeger [22], that are commonly employed by distributed systems deployed in the industry today [27]. However, such systems cannot capture the complex relationships between different microservices [32]; further, such systems still require manual effort and insight to actually detect performance bottlenecks. In general, the problem of detecting bottlenecks has garnered wide attention from the academic community as well [7, 16, 17, 19, 24, 25, 40, 44–46]. Recently, the availability of vast amount of tracing data has motivated data-driven approaches for performance management of microservices architecture [13, 15, 26, 32]. However, prior works that incorporate data-driven approaches either fail to fully use the structural information of the application deployment [15, 32], or use multiple complex models, thereby complicating the solution [13].

This work explores the use of *Graph Neural Networks* (GNNs) [9, 49] to detect bottlenecks in online microservices applications. GNNs are ideally suited for analyzing microservices applications:

- GNNs and their variants have produced ground-breaking performance on graph data [49] making them a natural choice for analyzing microservices call graphs [26, 28, 30].
- Models like GNNs are ideally suited to capture back-pressure and cascading performance degradation [14, 30] along the call graphs as they learn the dependence of graphs via message passing between the nodes of graphs [49].
- GNNs can generalize to dynamic graphs through transfer learning [20, 21] making them an ideal choice for microservices architecture where the call graphs are dynamic in nature [26, 32], saving retraining costs.
- GNN architectures can be regularized to ensure representation learning equilibrium across multiple classes thereby avoiding the multi-class imbalance problem seen in traditional ML algorithms [35]. The difficulty in collecting traces with bottlenecks in production systems makes GNN an ideal choice as it does not overfit on the majority (non-bottlenecked) class [15].

Motivated by the above observations, this work-in-progress paper explores the use of GNNs for detecting performance bottlenecks in microservices applications by designing B-MEG (Bottlenecked-Microservices Extraction using GNNs), a framework with two stages of GNN models. Preliminary results on a public dataset [31] are encouraging and show that B-MEG performs better than existing work that we compared against [32] for benchmark applications with a large number of microservices and complex call graphs (even when the training dataset is highly imbalanced). Compared to the Support Vector Machine (SVM) model used in existing work, B-MEG provides up to 15% and 14% improvements in accuracy and precision, respectively, and close to 10× improvement in recall of the bottlenecked classes. A detailed empirical comparison of B-MEG against other models and tools, such as those discussed in Section 2.1, is left for future work.

2 BACKGROUND AND RELATED WORK

Call Graphs and Traces: The series of Remote Procedure Calls (RPC) between microservices that service a user request is called a *call graph* [26]. The nodes of the call graph are RPCs of microservices and the edges correspond to an invocation of RPC from an upstream microservice to a downstream microservice. An analysis of microservices deployment in Alibaba clusters showed that at least 10% of the call graphs contain more than 40 microservices, and some call graphs can have thousands of microservices [26].

A single request type can have different call graphs due to different user parameters, components like caches and message queues, and asynchronous executions [26]. Further, agility in microservices architecture can lead to updates in microservices that can change the dependencies between them, thereby changing the call graphs.

Call graphs can be obtained using end-to-end tracing systems like Jaeger [22]. A *trace* is a data/execution path through the system, and can be thought of as a directed acyclic graph of spans, where a span is a logical unit of work. A distributed application can be instrumented at the RPC-level to get call graphs of each request.

Graph Neural Networks (GNN) GNNs are neural network models that are designed to learn representations on graph-structured data via feature propagation and aggregation. The input to a GNN is the graph representation of the problem being solved, where the graph could be explicit like in the case of call graphs, or implicit where an effort is involved to build the graph [49]. GNN outputs a representation for the input graph, called the embedding, using the features of the initial graph representation and the structure of the graph. These learnt representations are used to perform downstream tasks like graph classification, graph clustering, node classification, etc. The key advantage of GNN compared to standard ML frameworks is that GNNs can provide hierarchical convolutions in non-euclidean spaces. This is accomplished by a message passing process aggregating the embeddings of the neighbors of individual nodes, which in turn contain information about their neighbors. This way, the influence of neighboring microservices in a call graph can be learnt and the patterns that lead to propagation of bottlenecks to neighbors can be detected.

2.1 Related Work

Bottleneck detection in microservices applications: There is a large body of literature related to the general problem of bottleneck detection; we refer interested readers to a recent survey [37]. We now discuss more closely related prior works to put our work in context. FIRM [32] uses a Support Vector Machine (SVM) model to detect bottlenecks on the critical path of the call graph. The SVM model is trained using hand-crafted features that capture the per-critical-path and per-microservice performance variability. However, FIRM does not capture structural effects of call graphs as it treats each microservice independently for bottleneck detection.

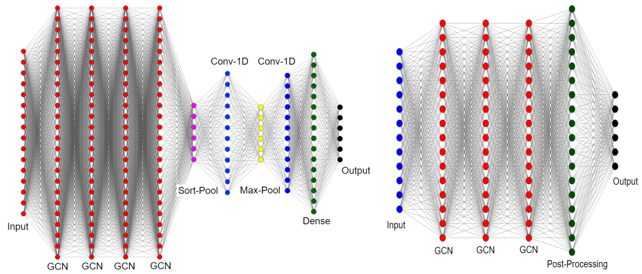
Seer [15] is an online cloud performance debugging system that leverages deep learning to detect and prevent QoS violations. Seer uses a hybrid neural network consisting of CNN and LSTM networks to learn spatial and temporal patterns that lead to QoS violations. However, analysis of Alibaba's production systems suggests that CNN-based approaches fail to characterize complex graph dynamics and are not applicable to real-world applications; instead, the authors suggest the use of GNNs [26], motivating our work.

Sage [13] uses Causal Bayesian Network (CBN) to capture the dependencies between microservices. However, the assumption in Sage that the non-leaf nodes' latency is determined by the wait time of its child nodes might not always hold [26]. Recent works [43, 47] have shown that GNNs can capture such causal relations, making additional models to capture causality redundant.

SuanMing [17] presents a framework for predicting future root causes to prevent the consequent performance loss. However, the assumption in SuanMing that the performance of the application is only dependent on type and amount of requests arriving at each service instance need not hold for data stores of the application which affect performance significantly [14, 15]. Even for stateless microservices, performance can depend on the payload size.

T-Rank [46], using latency as a bottleneck metric, detects bottlenecks based on Spectrum Based Fault Localization (SBFL). However, SBFL cannot capture the complex nature of microservices and incorrectly categorizes hot-spots, microservices that are shared across a significant number of call graphs [26], as bottlenecks.

Brandón et al. [7] present a graph-based framework that employs expert knowledge to detect bottlenecks. Through this framework, the authors also demonstrate the advantages of using graph techniques over ML techniques that do not exploit graph data. Our framework combines these two strategies by using a graph ML technique and alleviates the need of expert knowledge.



(a) Graph classifier (stage 1) **(b) Node classifier (stage 2)**

Figure 1: The two stages in the B-MEG framework.

Application Performance Monitoring (APM) tools: AppDynamics [1] leverages specialized ML models and various metrics collected across the application to detect bottleneck microservices. Dynatrace [2] uses context (topology, traces, and code-level) information to build and analyse a fault-tree to pinpoint bottlenecks.

3 OBJECTIVE AND SYSTEM DESIGN

We divide the problem of detecting bottlenecks into two sub-problems, the detection of potential anomalous traces (i.e., traces affected by bottlenecks), followed by detection of potential bottlenecks in such anomalous traces. We translate the sub-problems of detecting anomalous traces and potential bottlenecks into graph classification and node classification tasks, respectively. This division of problem is motivated by the benefits of hierarchical classifiers [36].

In a flat classification, where a single classifier classifies all the examples, the number of classes for an application with n microservices would be $n + 1$, one for each microservice and one additional class that corresponds to no bottlenecks. Based on the intuition that traces with bottlenecks would be similar to each other irrespective of the specific bottlenecks [29], we categorize them into one meta class—anomalous traces. This allows the use of a binary classifier as the first stage that classifies a trace as anomalous or regular. The traces classified as anomalous are provided as input to the second stage that detects potential bottlenecks in them. The main disadvantage of this design is the error propagation from first stage which can be controlled by varying the classification threshold of the first stage. We empirically compared the performance of a flat classification model versus the hierarchical model (B-MEG) and found that the hierarchical model leads to two simpler models with better performance which further motivated this design.

The B-MEG framework, as shown in Figure 1, consists of 2 stages with the first stage responsible for classifying potential anomalous traces and the second stage responsible for classifying potential bottlenecks. The first stage uses a Deep Graph Convolutional Neural Network (DGCNN) [10] for classifying if a trace is anomalous, and the second stage uses an inductive graph convolution training regime for pinpointing the microservices that are responsible for causing the anomaly. The choice of DGCNN for graph classification is due to its superior performance on inductive learning of graph representations without feature engineering. The node classifier is a vanilla Graph Convolution Network (GCN) architecture where the number of convolution layers were decided based on experiments.

The architecture of the DGCNN model, shown in Figure 1a, consists of four sequential stages: (i) four GCN layers to hierarchically extract the local substructure features of a node and define a node ordering [23]; (ii) one Sort Pooling layer for sorting the ordering under a pre-defined ordering and unifying the input sizes [48]; (iii) a sequence of traditional Convolution 1D layer, a max-pooling layer, and another Convolution 1D layer to read the sorted graph representations; and (iv) one post-processing dense layer followed by a softmax layer to make predictions. For node-classification, we use a semi-supervised graph convolution framework with three GCN layers, followed by a post-processing feed-forward and a softmax layer for predictions. The GCN layers hierarchically extract node features and pass it on to post-processing layer for classification.

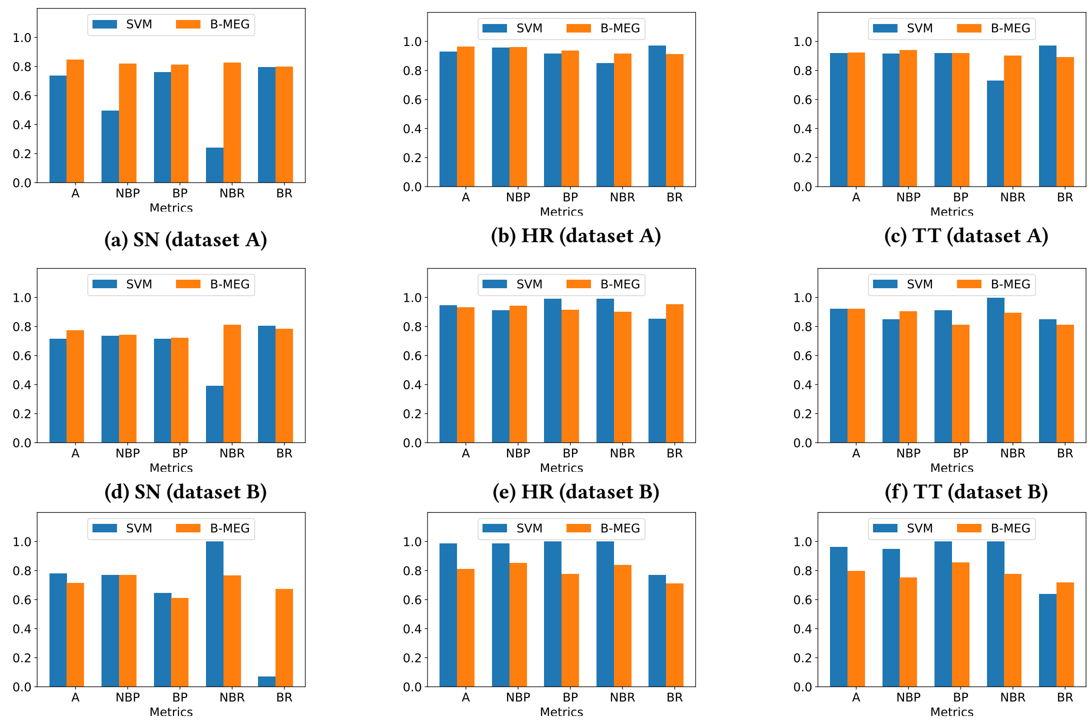
4 EVALUATION

Dataset: The dataset [31] released as part of the FIRM project [32] contains traces of social networking, media microservices, and hotel reservation applications from the DeathStarBench [14] suite and TrainTicket benchmark [50]. Most traces consist of a single bottleneck, the cause of which is an artificially induced resource interference, while the remaining traces have no bottlenecks.

Methodology: In this preliminary work, we focus our methodology on studying the effectiveness of GNN models on *imbalanced* datasets, which are the norm given the scarcity of production systems traces with bottlenecks [15]. To evaluate B-MEG's ability to handle the multi-class imbalance problem, we create three datasets each consisting of 790,000 traces—A, B, C—with the ratio of number of traces in the dataset with a microservice as the bottleneck to the number of traces without bottlenecks being 0.3, 0.1, and 0.01, respectively. The choice of 0.3 is to evaluate the performance of B-MEG for a fairly balanced dataset. The choice of 0.1 and 0.01 is motivated by similar ratios reported in production systems [24]. The datasets are created by random sampling to avoid any unexpected bias in them. We empirically evaluated how the performance of B-MEG varies with the total size of the dataset and chose the size at which the performance plateaued. The training time for the applications varies from 2–3 hours.

We use the bottleneck detection technique from FIRM [32] as the baseline to evaluate B-MEG's performance. FIRM [32] derives two features, the relative importance and congestion intensity, from service time of microservices to train an SVM model to detect bottlenecks. Similar to FIRM [32], we train both the models using service time of microservices as feature as it correlates well with bottleneck occurrence, but without any feature engineering. Using 80% of the traces from each class as the training data, both the models are trained separately and inductively where each trace is treated as a stand-alone instance; the remaining 20% dataset forms the test data. Unlike prior works [15, 32] that focus only on accuracy, we use other metrics like recall and precision which, as discussed in Section 1, are important when the dataset is imbalanced.

Preliminary results: Figure 2 shows the results for datasets A, B, and C (with different degree of class imbalance) and different benchmarking applications for SVM and B-MEG. For the social networking (SN) application, as seen in Figure 2a, B-MEG outperforms SVM with respect to all the metrics for dataset A. This suggests B-MEG's ability to effectively learn patterns that cause bottlenecks with a fairly imbalanced dataset without any feature engineering. For dataset B, B-MEG does better than SVM for all the metrics except for recall of bottlenecked classes, with SVM's value being 0.81 and B-MEG's 0.78. However, this advantage of SVM comes



(a) SN (dataset A) (b) HR (dataset A) (c) TT (dataset A)

(d) SN (dataset B) (e) HR (dataset B) (f) TT (dataset B)

(g) SN (dataset C) (h) HR (dataset C) (i) TT (dataset C)

Figure 2: Performance comparison of SVM and B-MEG on the traces of social networking (SN) [14], hotel reservation (HR) [14], and train ticket (TT) [50] applications. Metrics employed are accuracy (A), precision for non-bottlenecked (NBP) and bottlenecked classes (BP), recall for non-bottlenecked (NBR) and bottlenecked classes (BR). For all metrics, higher values are better.

with a very small recall (0.39) for the non-bottlenecked class, an undesirable trade-off. Moreover, B-MEG is capable of maintaining a good trade-off between overall precision (0.74) and recall (0.8) among all the classes, providing a high recall (0.81) for the non-bottlenecked class even when there is significant class imbalance. For dataset C, where the class imbalance is extreme, SVM has higher accuracy (0.78) than B-MEG (0.71), but suffers from a poor recall for bottlenecked classes (0.07). B-MEG on the other hand, provides a reasonable recall for bottlenecked classes (0.67), proving its ability to balance precision and recall even when the class imbalance is extreme. We see similar trends as dataset C when we further increase the class imbalance ratio from 0.01 to 0.001. We note that the call graph of social networking application in the FIRM dataset [31] has 31 microservices and 18 different paths from the root of the call graph to the leaf nodes, advocating the effectiveness of B-MEG in learning patterns in complex call graphs to detect bottlenecks.

Figures 2b, 2e, and 2h show that SVM either outperforms or performs similarly to B-MEG across all the datasets. Figures 2c, 2f, and 2i show similar trends for the train ticket application. Considering that the call graphs of hotel reservation and train ticket applications consist of 5 microservices with 3 different paths, and 11 microservices with 7 different paths, respectively, the results are not surprising. SVM's inability to exploit the structural information does not penalize its performance for these applications since their simple call graphs aid SVM in learning thresholds that signal bottlenecks. However, B-MEG still maintains a good balance between precision and recall for these two applications.

The above evaluation results show that even when the class imbalance is extreme, B-MEG is effective at detecting bottlenecks for microservices applications with large and complex call graphs. Given that such imbalance is the norm in production system traces [15, 26], we are encouraged by B-MEG's ability to maintain a good trade-off between precision and recall in such cases.

5 CONCLUSION AND FUTURE WORK

This work makes the case for employing GNNs to detect bottlenecks in applications designed using the microservices architecture. We evaluate our framework, B-MEG, using a recently published trace dataset [31] and compare the results against SVM, the model used to detect bottlenecks in FIRM [32]. In our preliminary experiments, B-MEG shows superior performance in detecting bottlenecks on imbalanced datasets for large and complex call graphs compared to SVM. As part of future work, we plan to explore transfer learning to make B-MEG generalizable, thus building on the strengths of GNNs. We also plan on collecting and open-sourcing a dataset with *multiple* bottlenecks. Creating a dataset that contains multiple bottlenecks, where the causes of these bottlenecks are not just resource contention [32], would further aid research in the area of bottleneck detection. Additionally, we will conduct a detailed analysis of the impact of dataset size on performance and on training effort. Finally, we plan empirically compare our improved framework with the tools and models described in Section 2.1.

Acknowledgment: This work was supported by NSF grant CNS-1750109.

REFERENCES

[1] 2022. Anomaly Detection. https://docs.appdynamics.com/4.5.x/en/appdynamics-essentials/alert-and-respond/anomaly-detection.

[2] 2022. Root cause analysis. https://www.dynatrace.com/support/help/how-to-use-dynatrace/problem-detection-and-analysis/problem-analysis/root-cause-analysis.

[3] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. 2003. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*.

[4] Maha Alsayasneh. 2020. *On the identification of performance bottlenecks in multi-tier distributed systems.* Ph.D. Dissertation. Université Grenoble Alpes [2020-....].

[5] Gianfranco Balbo and Giuseppe Serazzi. 1997. Asymptotic analysis of multiclass closed queueing networks: Multiple bottlenecks. *Performance Evaluation* (1997).

[6] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines.*

[7] Alvaro Brandon, Marc Solé-Simó, Alberto Huélamo, David Solans, María Pérez, and Victor Muntés-Mulero. 2020. Graph-based root cause analysis for service-oriented and microservice architectures. *Journal of Systems and Software* (2020).

[8] G. Casale and G. Serazzi. 2004. Bottlenecks identification in multiclass queueing networks using convex polytopes. In *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.*

[9] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. 2021. Machine Learning on Graphs: A Model and Comprehensive Taxonomy. arXiv:arXiv:2005.03675v2 [cs.LG]

[10] Ming Chen, Zhewei Wei, Zengfeng Huang, Bolin Ding, and Yaliang Li. 2020. Simple and Deep Graph Convolutional Networks. In *ICML.*

[11] J. Dean and L. A. Barroso. 2013. The Tail at Scale. *Commun. ACM* (2013).

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-Value Store. *SIGOPS Oper. Syst. Rev.* (2007).

[13] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. 2021. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices *(ASPLOS 2021).*

[14] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, Y. He, and C. Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[15] Yu Gan, Yanqi Zhang, Kelvin Hu, Yuan He, Meghna Pancholi, Dailun Cheng, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS).*

[16] A. U. Gias, G. Casale, and M. Woodside. 2019. ATOM: Model-Driven Autoscaling for Microservices. In *Proceedings of the 39th IEEE International Conference on Distributed Computing Systems.*

[17] Johannes Grohmann, Martin Straesser, Avi Chalbani, Simon Eismann, Yair Arian, Nikolas Herbst, Noam Peretz, and Samuel Kounev. 2021. *SuanMing: Explainable Prediction of Performance Degradations in Microservice Applications.*

[18] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. 2000. NetLogger: a toolkit for distributed system performance analysis. In *Proceedings 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems.*

[19] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. 2020. Graph-Based Trace Analysis for Microservice Architecture Understanding and Problem Diagnosis *(ESEC/FSE 2020).*

[20] Xueting Han, Zhenhuan Huang, Bang An, and Jing Bai. 2021. *Adaptive Transfer Learning on Graph Neural Networks.*

[21] Weihua Hu*, Bowen Liu*, Joseph Gomes, Marinka Zitnik, Percy Liang, Vijay Pande, and Jure Leskovec. 2020. Strategies for Pre-training Graph Neural Networks. In *International Conference on Learning Representations.*

[22] jaeger 2022. Jaeger. https://www.jaegertracing.io/.

[23] Thomas N Kipf and Max Welling. 2016. Semi-Supervised Classification with Graph Convolutional Networks. *arXiv preprint arXiv:1609.02907* (2016).

[24] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. 2021. Practical Root Cause Localization for Microservice Systems via Trace Analysis. In *IEEE/ACM International Symposium on Quality of Service (IWQoS) 2021.*

[25] D. Liu, C. He, X. Peng, F. Lin, C. Zhang, S. Gong, Z. Li, J. Ou, and Z. Wu. 2021. MicroHECL: High-Efficient Root Cause Localization in Large-Scale Microservice Systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice.*

[26] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing.*

[27] Jonathan Mace. 2017. *End-to-End Tracing: Adoption and Use Cases.* Survey. Brown University.

[28] A. Mathai, S. Bandyopadhyay, U. Desai, and S. Tamilselvam. 2021. Monolith to Microservices: Representing Application Software through Heterogeneous GNN. arXiv:2112.01317v2 [cs.SE]

[29] S. Nedelkoski, J. Cardoso, and O. Kao. 2019. Anomaly Detection from System Tracing Data Using Multimodal Deep Learning. In *2019 IEEE 12th International Conference on Cloud Computing.*

[30] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. 2021. GRAF: A Graph Neural Network Based Proactive Resource Allocation Framework for SLO-Oriented Microservices. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies.*

[31] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar Iyer. 2020. Pre-processed Tracing Data for Popular Microservice Benchmarks. https://databank.illinois.edu/datasets/IDB-6738796. Online.

[32] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).*

[33] J.A. Rolia and K.C. Sevcik. 1995. The Method of Layers. *IEEE Transactions on Software Engineering* (1995).

[34] G. Serazzri, G. Casale, and M. Bertoli. 2006. Java Modelling Tools: an Open Source Suite for Queueing Network Modelling andWorkload Analysis. In *Third International Conference on the Quantitative Evaluation of Systems - (QEST'06).*

[35] Min Shi, Yufei Tang, Xingquan Zhu, David Wilson, and Jianxun Liu. 2020. Multi-Class Imbalanced Graph Convolutional Network Learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence.*

[36] C. Silla and A. Freitas. 2011. A survey of hierarchical classification across different application domains. *Data Mining and Knowledge Discovery* (2011).

[37] J. Soldani and A. Brogi. 2022. Anomaly Detection and Failure Root Cause Analysis in (Micro) Service-Based Cloud Applications: A Survey. *ACM Comput. Surv.* (2022).

[38] G. Somashekar and A. Gandhi. 2021. Towards Optimal Configuration of Microservices. In *Proceedings of the 1st Workshop on Machine Learning and Systems.*

[39] Y. Tan, H. Nguyen, Z. Shen, X. Gu, C. Venkatramani, and D. Rajan. 2012. PREPARE: Predictive Performance Anomaly Prevention for Virtualized Cloud Systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems.*

[40] H. Wang, Z. Wu, H. Jiang, Y. Huang, J. Wang, S. Kopru, and T. Xie. 2021. Groot: An Event-graph-based Approach for Root Cause Analysis in Industrial Settings. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering.*

[41] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. 2013. Detecting Transient Bottlenecks in n-Tier Applications through Fine-Grained Analysis. In *2013 IEEE 33rd International Conference on Distributed Computing Systems.*

[42] Q. Wang, Y. Kanemasa, J. Li, D. Jayasinghe, T. Shimizu, M. Matsubara, M. Kawaba, and C. Pu. 2013. An Experimental Study of Rapidly Alternating Bottlenecks in n-Tier Applications. In *2013 IEEE Sixth International Conference on Cloud Computing.*

[43] S. Wein, W. Malloni, A. M. Tomé, S. M. Frank, G. Henze, S. Wüst, M. W. Greenlee, and E. W. Lang. 2020. A Graph Neural Network Framework for Causal Inference in Brain Networks. arXiv:2010.07143v1 [q-bio.NC]

[44] L. Wu, J. Tordsson, J. Bogatinovski, E. Elmroth, and O. Kao. 2021. MicroDiag: Fine-grained Performance Diagnosis for Microservice Systems. In *2021 IEEE/ACM International Workshop on Cloud Intelligence.*

[45] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. 2020. MicroRCA: Root Cause Localization of Performance Issues in Microservices. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium.*

[46] Z. Ye, P. Chen, and G. Yu. 2021. T-Rank:A Lightweight Spectrum based Fault Localization Approach for Microservice Systems. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid).*

[47] M. Zečević, D. S. Dhami, P. Veličković, and K. Kersting. 2021. Relating Graph Neural Networks to Structural Causal Models. arXiv:2109.04173 [cs.LG]

[48] M. Zhang, Z. Cui, M. Neumann, and Y. Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. *Proceedings of the AAAI Conference on Artificial Intelligence* (2018).

[49] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2021. Graph Neural Networks: A Review of Methods and Applications. arXiv:arXiv:1812.08434v6 [cs.LG]

[50] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao. 2018. Poster: Benchmarking Microservice Systems for Software Engineering Research. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion.*