Identifying Performance Issues in Microservice Architectures through Causal Reasoning

Luca Giamattei[1], Antonio Guerriero[1], Ivano Malavolta[2], Cristian Mascia[1], Roberto Pietrantuono[1], Stefano Russo[1]
[1]Università degli Studi di Napoli Federico II, Via Claudio 21, 80125, Napoli, Italy
[2]Vrije Universiteit Amsterdam, 1081 HV Amsterdam, The Netherlands
{luca.giamattei,antonio.guerriero,roberto.pietrantuono,sterusso}@unina.it,i.malavolta@vu.nl,cr.mascia@studenti.unina.it

ABSTRACT

Evaluating the performance of Microservices Architectures (MSA) is essential to ensure their proper functioning and meet end-user satisfaction. For MSA performance analysts, one of the most challenging tasks is to determine the cause of any deviation of relevant metrics from the specified range.

We introduce CAR-PT (CAusal-Reasoning-driven Performance Testing), a model-based technique for workload generation designed for the performance testing of MSA. CAR-PT leverages causal reasoning to effectively explore the space of operational conditions, with the goal of identifying those that lead to performance issues. Preliminary results show that CAR-PT is effective in generating configurations for discovering performance issues of an MSA.

KEYWORDS

Causal Reasoning; Microservices; Testing

ACM Reference Format:
Luca Giamattei[1], Antonio Guerriero[1], Ivano Malavolta[2], Cristian Mascia[1], Roberto Pietrantuono[1], Stefano Russo[1]. 2024. Identifying Performance Issues in Microservice Architectures through Causal Reasoning. In *5th ACM/IEEE International Conference on Automation of Software Test (AST 2024) (AST '24), April 15–16, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3644032.3644460

1 INTRODUCTION

Performance testing is a very challenging task when testing Microservices Architectures (MSA) - today a very popular software architectural style [1]. Indeed, many factors can impact MSA performance, including the deployment environment [2] and the users' behavior in operation [3, 4].

This paper proposes a testing technique for finding the conditions that cause performance issues in MSA. The technique, called CAR-PT (CAusal-Reasoning-driven Performance Testing), is based on causal discovery followed by causal inference [5]. These are used to drive the model-based generation of a workload for MSA performance testing. The aim is to provide automated support to performance engineers in a relevant activity, namely to "determine the cause of any deviation in the counter values from the specified or expected range (e.g., response time exceeds the maximum response time permitted by the service level agreements or memory usage exceeds the average historical memory usage)" [6]. This is achieved by simulating workloads that are both effective in unveiling performance issues and representative of actual workloads.

Starting from historical data (e.g., provided by monitoring tools), CAR-PT first automatically extracts a causal model capturing the cause-effect relationships between performance-related variables of the system (e.g., resources usage, response time, etc.); then, the model is queried to find the most critical configurations from the performance point of view.

AST '24, April 15–16, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0588-5/24/04.
https://doi.org/10.1145/3644032.3644460

2 RELATED WORK

2.1 Performance assessment

Performance testing. Testing is one of the most common solutions for performance assessment for service-based systems [7]. A test is typically a workload configuration aiming to unveil performance issues of the system under test. Barna *et al.* [8] propose a framework to generate workloads by exploring the settings causing the wrong behavior for software and hardware components of transactional systems. Several studies propose techniques for automatic test generation for performance testing. Some of them exploit symbolic execution [9–11]. Han *et al.* [12] focus on automatic dictionary-based input generation for performance bug tracking. Camilli *et al.* [3] propose MIPaRT for integrated performance and reliability testing, where workloads are automatically generated by setting the *workload intensity* and the *behavior mix*, describing the expected and unexpected behavior of the users.

Performance prediction. Many techniques have been proposed to predict the performance of cloud applications. Bertot *et al.* proposed a framework for the execution time and cost prediction of workloads [13]. Wang *et al.* propose CAPT[14] for performance testing of cloud applications which relies on a smart test oracle aiming to predict the performance of CPU and memory-intensive applications. Gambi *et al.* [15] propose a controller using continuous learning to predict service performance under different workloads.

This work exploits a performance prediction approach based on causal models for the generation of performance-critical workload configurations. Causality, more specifically causal structure discovery, has been used in microservices for root cause analysis [16], in which the authors showed how such algorithms can capture the causal relations between performance-related variables. Our aim is to exploit the causal relations to generate critical configuration.

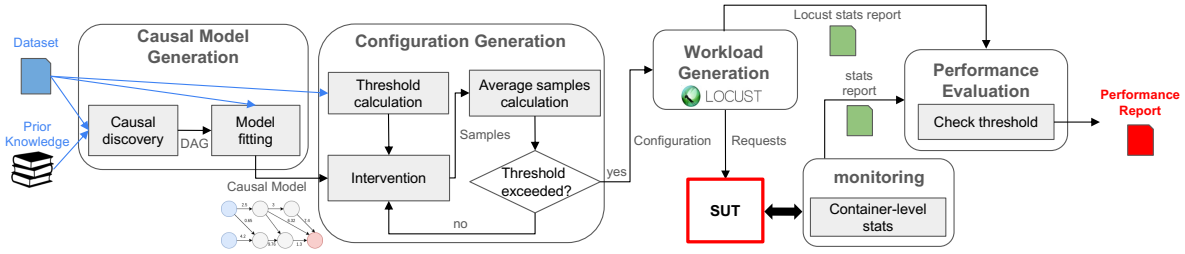

Figure 1: The CAR-PT workflow

2.2 Test generation with causal reasoning

A few causality-based testing techniques have been developed for test generation. Oh *et al.* [17] perform mutation testing by leveraging causal program dependency analysis to discover the causal structure of the SUT and ultimately to sample effective mutants. Similarly, Clark *et al.* [18] use causal inference to automatically generate metamorphic relations, which usually require domain expertise and human input. They observed that metamorphic testing is fundamentally a causal task: apply transformations to input and evaluate the effect on the output. Giamattei *et al.* [19] also recognize software testing as a causal reasoning problem and reformulate the task of test case generation based on causality. The authors describe RBST (Reasoning-based Software Testing), a framework to perform *interventions* on a causal model to generate a set of hypothetical tests, from which only the "best" ones with respect to the testing objective are executed.

In this work, we propose the use of causal reasoning for test cases generation in the context of performance testing.

3 THE CAR-PT TECHNIQUE

Figure 1 shows the workflow of the proposed technique. The steps include the discovery of the causal structure (from which a causal model is built) (Sec. 3.1), used to generate failure-prone configurations (Sec. 3.2); the configuration is the input of the actual workload generation (Sec. 3.3).

3.1 Causal model generation

A causal model is a representation of cause-effect relationships between a set of variables. One of the most commonly used type of models is the Structural Causal Model (SCM), which is a Directed Acyclic Graph (DAG) $\mathcal{G} = (\boldsymbol{X}, \mathcal{E})$, where nodes $\in \boldsymbol{X}$ are random variables and edges $\in \mathcal{E}$ capture the causal relationships between them. The relationships are described as a collection of structural assignments (i.e., *structural equations*) $X_i := f_i(Pa(X_i), U_i)$ that define the (endogenous) random variables X_i as a function of their parents $Pa(X_i)$ and of (exogenous) independent random noise variables U_i. They can be manually built by domain experts or automatically derived from observational data by causal discovery algorithms. These algorithms still allow experts' knowledge to guide the generation, by feeding them a "prior knowledge", namely a set of constraints specifying mandatory/prohibited relationships between variables. For our experiments, we prohibit every incoming edge to input variable. This is because these variables are manipulated by the proposed technique and, therefore, are not causally affected by any other variable.

Table 1: Extract of the dataset used

user size	*load*	*spawn rate*	REQ/s_s0	RT_s0	CPU_s0	MEM_s0
1	*uniform*	1	2.8	6.5	6.6	62.3
1	*unbalanced*	1	10.9	5.7	18.6	90.3
1	*randomly balanced*	1	2.5	6.4	5.7	108.0

RT: response time; MEM: Memory consumption

We use various causal discovery algorithms to build the DAG and an automatic tool, `DoWhy.gcm [20]`, to assign and fit the causal mechanisms (the above structural function f_i of an SCM) for each edge in the causal graph. The assignment compares a linear, polynomial, and gradient boost model and selects the best-fitting one. These steps (i.e., *causal discovery* and *model fitting* in Figure 1) constitute the causal model generation phase. They require a dataset as input, that we build from the execution of various nominal workloads (an excerpt is in Table 1).

3.2 Configuration generation

Similarly to Camilli *et al.* [3], we define a *configuration* as a triple <*users size, load, spawn rate*>, where:

- The *users size* is the number of users simultaneously interacting with the system (sending requests).
- The *load* defines how the users interact with services. This behavior is represented by an invocation matrix where each cell represents the probability the users will send a request to the service on the column after a request to the service on the row. We consider three types of load:
  - *uniform*: users send requests to all the services sequentially;
  - *unbalanced*: a service has a higher probability of receiving a request, namely the load is unbalanced toward one service;
  - *randomly balanced*: the invocation matrix is randomly generated, the load is balanced; each service $s \in S$ has the same probability $\frac{1}{|S|}$ of being called.
- The *spawn rate* is the rate to spawn users (number of users per second).

An example of an invocation matrix is shown in Table 2.

Table 2: Example of invocation matrix: probabilities of invocation (of the service on the row to service on the column)

	s0	*s1*	*s2*
s0	0	1	0
s1	0.50	0	0.50
s2	0.50	0.25	0.25

An example of configuration is <*8, randomly balanced, 2*>, meaning that at most 8 concurrent users, with 2 users spawning every second (*users* × *spawn rate*), send requests to each microservice according to the invocation matrix in Table 2.

The configurations are generated by querying the causal model through *interventions*. An *intervention* on the causal model consists in setting the values of a set of variables (i.e., in our case the variables describing the *configuration*) in order to see the effect on other variables of interest (i.e., performance indicators: response time, memory consumption, CPU). We run interventions iteratively through `DoWhy.gcm` [20] by gradually changing the users size (setting spawn rate and load) until the model prediction does not exceed a threshold in one of the performance indicators for the specified service(s). For instance, a tester may ask for *how many users are needed to trigger a response time issue for the microservice* s_0. The model is queried to predict the effect of increasing the number of users on s_0 response time until the threshold is exceeded.

3.3 Workload generation and monitoring

The generated configuration is given as input to `locust` in order to generate the actual workload. `locust` is configured with a locustfile where the tester defines the users' behavior and the operational profile. Users are modeled as a Markov chain [3] where each state corresponds to a request. The arcs connect the states with a weight representing the probability of executing a request given the previous one. Requests are defined by the tester based on the MSA APIs, and the weights can be derived from historical data or the tester's expectation of the user's behavior in operation.

During the workload execution, <the *requests rate*, *response time*, *CPU consumption*, *memory consumption*> are collected for each service. The first two metrics are collected by `locust` and the second ones by `Dockerstats`.

3.4 Performance evaluation

The detection of performance issues implies the computation of thresholds, which define the nominal/correct behavior of the system under test. The *ground truth* is built by running an ideal workload with a single user, and by monitoring the variables of interest (e.g. response time, CPU, memory usage). Thresholds are defined according to ref. [2] (*scalability thresholds*) as $\tau_X = \mu_X + 3 \cdot \sigma_X$, where X is the variable of interest, and μ_X and σ_X are its mean and standard deviation over past executions. A performance issue is an execution where thresholds are exceeded for at least one variable of interest.

4 EXPERIMENTATION

The preliminary experiments consist of two phases. First, we analyze the performance of various causal discovery algorithms in generating a configuration. Then, we show how CAR-PT works on a simulated MSA. For simulating an MSA we consider `µBench` [21], a tool for benchmarking cloud/edge computing platforms that run microservice applications. It allows to configure custom microservice meshes, where each microservice is configured with a specific stressing function (CPU, MEM, disk, no stress/idle). For our experiments, we generate a simulated system with a random service mesh composed of 10 services, each one with a random stressing function.

Each experiment is repeated five times and lasts three minutes. There is a two-minute pause between two experiments to avoid carryover effects (i.e., consecutive runs influencing each other, e.g., in terms of CPU and RAM). During the execution, a dataset is built with data gathered by `locust` and `Dockerstats`, as described in section 3.3. On this dataset, we ran causal discovery algorithms.

For selecting the causal discovery (CD) algorithm and configuring it, we address the following Questions (Q):

- **Q0: Which is the best-performing CD algorithm in predicting anomalies?**
- **Q1: Does prior knowledge improve performance?**
- **Q2: How does performance vary if we consider one model per performance metric or one model for all metrics?**
- **Q3: How does performance vary by changing the minimum confidence required for a causal relation identification?**

Q0. We compare the models generated by four CD algorithms, by evaluating the ability of predicting the *number of users required to trigger a performance issue* on these models. The algorithms are `dLiNGAM`[22], `DAGMA-LINEAR`[23], `DAGMA-MLP`[23] and `DAG-GNN`[24]. The configurations are generated as a capacity test, i.e., by constantly increasing the number of users until the threshold is exceeded. This is done for each service-metric pair (10 services, monitored on three metrics: *response time*, CPU, *memory consumption*) for all combinations of load (3 levels) and spawn-rate (3 levels) – i.e., on 90 configuration per metric. We compute *precision* (as ratio of correct anomaly predictions to the number of anomalies predicted by the model) and *recall* (as ratio of correct anomaly predictions to the number of real anomalies).

For **Q1**, we consider the best causal discovery algorithm from the previous question, and tried to enrich it with the following prior knowledge:

- $T \rightarrow R_s$, where T is a *treatment*, i.e., a configuration <*users size, load, spawn rate*> used to query the model and R is the request rate for the service s. In this case, the configuration is causally related to the request rate.
- $R_s \rightarrow metric_s$ for all services s and for all metrics
- $response\ time_s \rightarrow response\ time_{s'}$ if $s' \rightarrow s$ in the service mesh (i.e., s is called by s').
- $CPU_s \rightarrow CPU_{s'}$ if $s \rightarrow s'$ in the service mesh
- $memory_s \rightarrow memory_{s'}$ if $s \rightarrow s'$ in the service mesh

The goal is to figure out if this knowledge improves the model.

For **Q2**, we split the dataset into three subsets, one for each metric (CPU, response time, memory). For example, the CPU dataset consists of: the treatments (the configuration triple), the request rate for each service, and the CPU consumption for each container.

For **Q3**, we consider the *causal arrow strength* [25], i.e., an indirect measure, based on explained variance, of the confidence in the existence of a causal relation. We study how performance varies if we set the minimum strength cut-off value σ for an edge to be kept in the model $\sigma = .1$ and $\sigma = .3$, hence obtaining two simplified models. An edge is cut if its strength is below σ. The best model is then executed into CAR-PT by running the entire workflow described in Section 3.

Table 3: Q0: Comparison of causal discovery algorithms

Algorithm	Metric	Precision	Recall
dLiNGAM	Response Time	0.524	0.880
	CPU	0.989	1
	Memory	0.714	0.385
DAGMA-LINEAR	Response Time	0.038	0.200
	CPU	0.967	1
	Memory	0.743	0.321
DAGMA-MLP	Response Time	0.053	0.211
	CPU	0.967	1
	Memory	0.444	0.160
DAG-GNN	Response Time	0.095	0.304
	CPU	0.967	1
	Memory	0.818	0.439

5 RESULTS

Q0. Table 3 shows the results for the compared causal discovery algorithms. `dLiNGAM` emerged as the best algorithm for two out of three metrics: *response time* and *CPU*. In the case of response time, it gives significantly superior precision and recall compared to others. In the *CPU* metric, where all algorithms exhibited strong performance, it achieved marginally higher values. On the other hand, concerning *Memory*, `DAG-GNN` demonstrated the best performance, with `dLiNGAM` ranking third in precision and second in recall. For the next questions, we opted for `dLiNGAM`.

Q1. Table 4a shows the results of `dLiNGAM` with prior knowledge. Leveraging prior knowledge significantly enhanced the predictive performance of the causal model for *response time* and *CPU*. The performance remains unchanged for *memory* compared to the scenario without prior knowledge. We choose the causal model with prior knowledge.

Q2. Table 4b shows the results of `dLiNGAM` applied on a single-metric dataset. The performance for *response time* on a single metric is worse than on the whole dataset because the relationship between different types of metrics is not negligible. However, the precision of the memory metric is greater than in the previous case.

We chose `dLiNGAM` applied on the whole dataset for the better results on the response time.

Q3. Table 5 shows the results of simplified models with two minimum strength cut-off values ($\sigma = .3$ and $\sigma = .1$). The models with $\sigma = .3$ increased the performance of memory prediction precision keeping the same performance for response time and CPU metrics. We opt for the simplified causal model with $\sigma = .3$.

Table 4: Performance of `dLiNGAM` with prior knowledge

	(a) Q1: whole dataset		(b) Q2: single metric	
Metric	Precision	Recall	Precision	Recall
Response Time	0.679	0.905	0.415	0.810
CPU	1	1	1	1
Memory	0.717	0.507	0.848	0.470

Table 5: Q3: Comparison of two thresholded causal models

Threshold	Metric	Precision	Recall
0.1	Response Time	0.378	0.795
	CPU	1	1
	Memory	0.830	0.476
0.3	Response Time	0.679	0.905
	CPU	1	1
	Memory	0.792	0.475

Final model in practice. In the last experiment, we generate three configurations with CAR-PT and the random baseline, comparing the number of performance issues discovered. Each configuration is generated by querying the causal model to give a configuration with a response time, CPU, and memory issue, respectively. CAR-PT and the baseline are compared on the generation of *user size*[1]: CAR-PT generates the user size as in Section 3.2, and the baseline generates a random number between 2 and 30 (30 is the largest user size in the dataset). Each experiment is repeated five times.

Table 6 shows the percentage of performance issues (PI%) discovered by generators per metric and in total. CAR-PT outperforms the random baseline in terms of response time, while there is no difference in CPU and memory.

Table 6: Comparison of CAR-PT and a random generator

Generator	Metric	PI%	PI% total
CAR-PT	Response Time	100%	66.667%
	CPU	100%	
	Memory	0%	
RANDOM	Response Time	40%	46.667%
	CPU	100%	
	Memory	0%	

6 CONCLUSION

We presented CAR-PT, a model-based technique for workload generation in the context of MSA performance testing. The responses to the questions offer guidance on instantiating CAR-PT, particularly in the selection of a CD algorithm and best practices for model construction. Preliminary results show that CAR-PT can be effective in generating configurations for discovering performance issues by exploiting causal relationships, outperforming the baseline.

Our next goal is to experiment with real-world representative MSA like `TrainTicket` [26] and `SockShop`[2], and further investigate the alternatives to instantiate CAR-PT.

ACKNOWLEDGMENT

This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 871342 "uDEVOPS".

[1]We fix *load=uniform* to avoid bias in the comparison (the workload is balanced in all executions) and *spawn rate=5* to reduce the rump up time.
[2]https://github.com/microservices-demo/microservices-demo.

REFERENCES

[1] J. Soldani, S. A. Tamburri, and W.J. Van Den Heuvel. The pains and gains of microservices: A systematic grey literature review. *Journal of Systems and Software*, 146:215–232, 2018.

[2] A. Avritzer, V. Ferme, A. Janes, B. Russo, A. van Hoorn, H. Schulz, D. Menasché, and V. Rufino. Scalability assessment of microservice architecture deployment configurations: A domain-based approach leveraging operational profiles and load tests. *Journal of Systems and Software*, 165:110564, 2020.

[3] M. Camilli, A. Guerriero, A. Janes, B. Russo, and S. Russo. Microservices Integrated Performance and Reliability Testing. In *3rd International Conference on Automation of Software Test*, page 29–39. ACM, 2022.

[4] M. Camilli, C. Colarusso, B. Russo, and E. Zimeo. Actor-driven decomposition of microservices through multi-level scalability assessment. *ACM Transactions on Software Engineering Methodologies*, 32(5), 2023.

[5] A. R. Nogueira, A. Pugnana, S. Ruggieri, D. Pedreschi, and J. Gama. Methods and tools for causal discovery and causal inference. *WIREs Data Mining and Knowledge Discovery*, 12(2):e1449, 2022.

[6] M. D. Syer, Z. M. Jiang, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora. Continuous validation of load test suites. In *Proceedings of the 5th International Conference on Performance Engineering (ICPE)*, page 259–270. ACM, 2014.

[7] S. He, G. Manns, J. Saunders, W. Wang, L. Pollock, and M. L. Soffa. A statistics-based performance testing methodology for cloud applications. In *Proceedings of the 2019 27th Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, page 188–199. ACM, 2019.

[8] C. Barna, M. Litoiu, and H. Ghanbari. Autonomic load-testing framework. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC)*, page 91–100. ACM, 2011.

[9] J. Burnim, S. Juvekar, and K. Sen. Wise: Automated test generation for worst-case complexity. In *31st International Conference on Software Engineering (ICSE)*, pages 463–473. IEEE, 2009.

[10] P. Zhang, S. Elbaum, and M. B. Dwyer. Automatic generation of load tests. In *26th International Conference on Automated Software Engineering (ASE)*, pages 43–52. IEEE, 2011.

[11] B. Chen, Y. Liu, and W. Le. Generating performance distributions via probabilistic symbolic execution. In *38th International Conference on Software Engineering (ICSE)*, pages 49–60. IEEE, 2016.

[12] X. Han, T. Yu, and D. Lo. Perflearner: Learning from bug reports to understand and generate performance test frames. In *33rd International Conference on Automated Software Engineering (ASE)*, pages 17–28. IEEE, 2018.

[13] L. Bertot, S. Genaud, and J. Gossa. Improving cloud simulation using the monte-carlo method. In M. Aldinucci, L. Padovani, and M. Torquati, editors, *Euro-Par 2018: Parallel Processing*, pages 404–416. Springer, 2018.

[14] W. Wang, N. Tian, S. Huang, S. He, A. Srivastava, M. L. Soffa, and L. Pollock. Testing cloud applications under cloud-uncertainty performance effects. In *11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 81–92. IEEE, 2018.

[15] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezzè. Kriging controllers for cloud applications. *IEEE Internet Computing*, 17(4):40–47, 2013.

[16] L. Wu, J. Tordsson, E. Elmroth, and O. Kao. Causal inference techniques for microservice performance diagnosis: Evaluation and guiding recommendations. In *International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 21–30. IEEE, 2021.

[17] S. Oh, S. Lee, and S. Yoo. Effectively sampling higher order mutants using causal effect. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 19–24. IEEE, 2021.

[18] A. G. Clark, M. Foster, B. Prifling, N. Walkinshaw, R. M. Hierons, V. Schmidt, and R. D. Turner. Testing causality in scientific modelling software. *ACM Transactions on Software Engineering and Methodology*, jul 2023.

[19] L. Giamattei, R. Pietrantuono, and S. Russo. Reasoning-based software testing. In *45th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 66–71. IEEE, 2023.

[20] P. Blöbaum, P. Götz, K. Budhathoki, A. A. Mastakouri, and D. Janzing. Dowhy-gcm: An extension of dowhy for causal inference in graphical causal models. arXiv:2206.06821, 2022.

[21] A. Detti, L. Funari, and L. Petrucci. uBench: An Open-Source Factory of Benchmark Microservice Applications. *IEEE Transactions on Parallel and Distributed Systems*, 34(3):968–980, 2023.

[22] S. Shimizu, T. Inazumi, Y. Sogawa, A. Hyvärinen, Y. Kawahara, T. Washio, P. O. Hoyer, and K. Bollen. DirectLiNGAM: A Direct Method for Learning a Linear Non-Gaussian Structural Equation Model. *Journal of Machine Learning Research*, 12:1225–1248, jul 2011.

[23] K. Bello, B. Aragam, and P. Ravikumar. DAGMA: Learning DAGs via M-matrices and a Log-Determinant Acyclicity Characterization. In *Advances in Neural Information Processing Systems*, 2022.

[24] Y. Yu, J. Chen, T. Gao, and M. Yu. DAG-GNN: DAG structure learning with graph neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *36th International Conference on Machine Learning*, volume 97, pages 7154–7163. PMLR, 09–15 Jun 2019.

[25] X. Zheng, B. Aragam, P. K. Ravikumar, and E. P. Xing. DAGs with NO TEARS: Continuous Optimization for Structure Learning. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[26] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2021.