Localizing Failure Root Causes in a Microservice through Causality Inference

Yuan Meng[1,4], Shenglin Zhang[2]*, Yongqian Sun[2]
Ruru Zhang[2], Zhilong Hu[2], Yiyin Zhang[3], Chenyang Jia[3], Zhaogang Wang[3], Dan Pei[1,4]
[1]Tsinghua University, [2]Nankai University, [3]Alibaba Group
[4]Beijing National Research Center for Information Science and Technology (BNRist)

***Abstract*—An increasing number of Internet applications are applying microservice architecture due to its flexibility and clear logic. The stability of microservice is thus vitally important for these applications' quality of service. Accurate failure root cause localization can help operators quickly recover microservice failures and mitigate loss. Although cross-microservice failure root cause localization has been well studied, how to localize failure root causes in a microservice so as to quickly mitigate this microservice has not yet been studied. In this work, we propose a framework, MicroCause, to accurately localize the root cause monitoring indicators in a microservice. MicroCause combines a simple yet effective path condition time series (PCTS) algorithm which accurately captures the sequential relationship of time series data, and a novel temporal cause oriented random walk (TCORW) method integrating the causal relationship, temporal order, and priority information of monitoring data. We evaluate MicroCause based on 86 real-world failure tickets collected from a top tier global online shopping service. Our experiments show that the top 5 accuracy ($AC@5$) of MicroCause for intra-microservice failure root cause localization is 98.7%, which is greatly higher (by 33.4%) than the best baseline method.**

I. Introduction

Microservice has gained an increasing popularity in recent years, especially for the applications that need to support a broad range of platforms, *e.g.,* IoT, mobile, and Web [1]. The performance quality of microservice is of vital importance to the Internet company, because a microservice failure can degrade the user experience and bring economic loss [2], [3]. Therefore, an efficient root cause localization of online failures, which enables rapid service recovery and loss mitigation, becomes increasingly more important for microservices.

In the microservice architecture, an application is decoupled into multiple microservices. Recently, several works have been proposed to understand how a failure is propagated across microservices and try to localize the root cause *microservice* that leads to this failure [4]–[6]. However, localizing failure root causes within a microservice has not yet been investigated. It is important for operators to further understand why a microservice fails, otherwise they cannot take measures to mitigate the failure.

To find the root cause of a microservice failure, we need to know how a microservice works. Here we take "discount coupon" microservice in the online shopping platform as an example, as shown in Fig. 1. For a microservice, there are usually upstream components (*e.g.,* a microservice, a middleware) calling it, and it can call some downstream components (*e.g.,* a microservice, a middleware). For example, the order creation microservice will call the discount coupon service when a customer places an order. To implement its function, the discount coupon microservice will call the inventory microservice and some middlewares, *e.g.,* the notice publishing platform, database. Moreover, a microservice is typically deployed on one or more containers or virtual machines. *Typically, there are mainly three types of components that may lead to a microservice failure: upstream component, downstream component, deployment environment.*

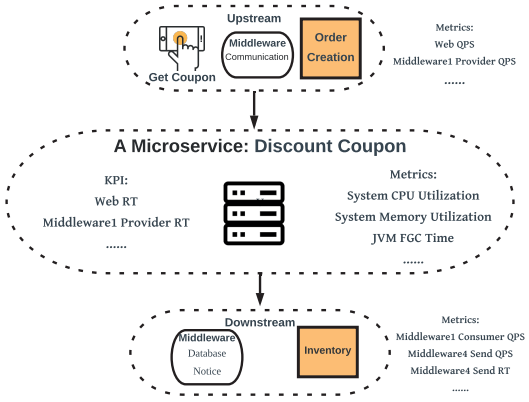


Fig. 1. An example of microservices architecture. FGC means full garbage collection in java virtual machine.

Operators have configured a collection of indicators to continuously monitor the performance of each microservice. As shown in Fig. 1, these indicators include *KPIs* which are user perceived indicators such as response time (RT) of Web users, and *metrics* include upstream component related indicators (*e.g.,* queries per second (QPS) of Web users), downstream component related indicators (*e.g.,* middleware RT), deployment environment related indicators (*e.g.,* CPU utilization of the underlying container). When a KPI becomes abnormal, it usually indicates a microservice failure, caused by the microservice's upstream component, downstream component, or deployment environment, which usually becomes anomalous. Typically, the anomalous component or deploy-

*Shenglin Zhang is the corresponding author.

978-1-7281-6887-6/20/$31.00 © 2020 IEEE

ment environment can be reflected by an anomaly in one or more metrics. Besides, an anomalous metric can further tell why this root cause component becomes anomalous. *Consequently, the root cause of a microservice failure can be denoted by an anomalous metric.* For example, as shown in Fig. 2, a microservice failure indicated by the anomalous RT (KPI) is caused by the anomalous Web QPS (metric).

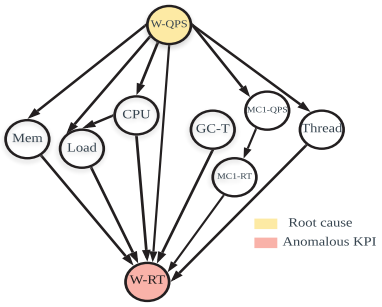


Fig. 2. An example of the causal graph of a microservice failure. *W-RT* is Web RT and *W-QPS* is Web QPS. *Mem*, *Load*, *CPU* are the memory utilization, load, and CPU utilization of the underlying container, respectively. *GC-T* and *Thread* are respectively the number of garbage collections and that of threads in the underlying java virtual machine. *MC1-RT* and *MC1-QPS* are the average consumers' RT and QPS of middleware1, respectively.

Currently, operators usually manually localize the root cause metric when a microservice failure occurs. However, due to the large number and complicated dependency relationship of KPIs and metrics, this manual way is time-consuming, labor-intensive, and error-prone. Therefore, *in this work we aim to design an automotive framework to robustly localize the root cause metric in a microservice.* Usually, localizing failure root causes for computer systems has two main parts: learning the relationship of components to construct dependency graphs, and inferring root causes through random walk [7]–[9].

However, this idea faces three major challenges in our scenario as follows.

- Traditional causal graph construction method, which is designed for independent and identically distributed (iid) data, cannot fully utilize propagation delays.
- Current random walk algorithm, which is based on the assumption that, an abnormal metric more correlated with an anomalous KPI is more likely to be the root cause, is not always true in our scenario.

We make the following contributions in this paper when addressing the above challenges:

- To tackle the first challenge, we design a novel PCTS (path condition time series) algorithm to learn the dependency graph of monitoring indicators with propagation delays being fully utilized. In PCTS, we first adopt the improved PC [10] to learn the causal graph of each point of time series. Then we generate the edges between two time series and generate a failure causal graph.
- To tackle the second challenge, we propose a novel temporal cause oriented random walk (TCORW) approach. In TCORW, we successfully integrate three types information: (1) causal relationship of monitoring indicators, (2) anomaly information of metrics including occurrence time and anomaly degree, and (3) metric priority which is obtained based on domain knowledge.
- Combining PCTS and TCORW, we propose a novel framework, MicroCause, to infer the top N root causes of the failure in a microservice. To the best of our knowledge, this is the first work to localize failure root causes in a microservice.
- We collected 86 online failure tickets from a large online shopping platform in three months. We evaluate the MicroCause and the baseline methods based on these cases. The $AC@5$ of MicroCause is 98.7%, which is higher 33.4% than the best performance of baseline method.

II. Preliminaries

In this section, we first introduce the architecture of microservice in Section II-A, followed by the statement of the studied problem in Section II-B. The basic concepts of the PC algorithm and random walk are depicted in Section II-C and Section II-D, respectively.

A. Microservice Architecture

Microservice becomes increasingly more popular recently due to its flexibility, scalability, and ease of deployment. In the microservice architecture, an application is decoupled into multiple microservices, each of which is responsible for a specific function, *e.g.,* querying data, creating orders. In this way, a microservice, due to its reusability, can serve several applications at the same time. Therefore, there are direct or indirect calling relationships among microservices in diverse application scenarios.

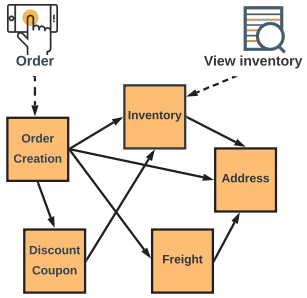


Fig. 3. The call graph of microservices in the process of placing an order

Here we take the process of placing an order, which is a frequently used application on online shopping platforms, as an example. As Fig. 3 shows, this application involves several microservices. Firstly, the order creation microservice receives a user's request and calls the inventory microservice to check whether there are enough commodities. After that, the inventory microservice calls the address microservice to calculate the closest warehouse and counts its inventory of the requested commodity. Besides, the order creation microservice

calls the address microservice to generate the delivery address, calls the discount coupon microservice to check whether this order can use any coupon, and calls the freight microservice to deliver the commodity. Consequently, an application is completed through a series of callings among microservices.

B. Problem Statement

Failure and anomaly: A microservice failure is an event that the microservice is not functional and seriously impacts the user's quality of experience. For example, when the order creation microservice create orders very slowly or even cannot create any order, this service is failed. It must be observable, from a person, an application, or another microservice [11]. Besides, a microservice is anomalous, *e.g.,* its underlying containers suffer from extremely high memory utilization, is when its behavior deviates from the normal one. However, for a microservice an anomaly status does not necessarily lead to a failure, thanks to its load balancing mechanism, self-recovery strategy, *etc.*

KPI and metric: Usually, operators carefully configure a collection of microservices' indicators to continuously monitor applications' status, quickly locate the root causes of failures, and timely mitigate failures, to guarantee the quality of users' experience. These indicators, which are essentially time series data, can be classified into two categories: key performance indicator (KPI) and metric. A KPI, *e.g.,* the average response time of a microservice, is a user-perceived indicator that directly reflects the quality of service. When a KPI of a microservice becomes anomalous, the microservice will be deemed to be failed. A metric, on the other hand, indicates the status of a microservice's underlying component. It can be the CPU utilization of a microservice's underlying physical machine/virtual machine/container, the queries per second (QPS) of its upstream microservice, or the response time of its downstream microservice. *An anomalous metric can be the potential root cause of an abnormal KPI indicating a microservice failure.* However, an anomaly of metric does not necessarily lead a KPI to be anomalous. For example, when a microservice's underlying database fails and the database related metrics become anomalous, the microservice may be normal due to the load balancing strategy.

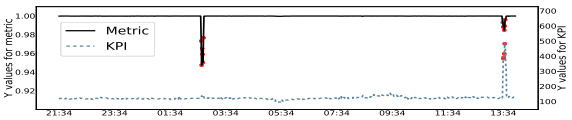


Fig. 4. Failure ticket in microservice A

To get readers better understand the difference between KPI and metric, and that between anomaly and failure, we give an example in Fig. 4. Microservice A becomes anomalous at 02:34 and 13:34, because one of its metrics behaves abnormal then. However, the microservice fails only at 13:34 because its KPI becomes abnormal at that time.

Failure ticket: In our scenario, a failure ticket of microservice usually consists of three key elements: a microservice ID indicating where the failure occurs, a KPI representing which KPI becomes anomalous, and a timestamp showing when this failure happens. Therefore, it can be denoted as a tuple, *i.e.,* $\{$*microservice ID*, *KPI*, *timestamp*$\}$. For example, $\{$*Microservice A, Web RT, 17:18*$\}$ is a microservice failure.

Problem definition: With the above definitions, our objective can be formulated as follows. Given one failure ticket, MicroCause will try to localize the top N metrics that are most likely to be the root cause of the failure.

C. PC Algorithm

Causality learning aims to discover the causal relationship from data, which can help to better understand physical mechanisms. The data needed can come from well-designed comparative experiments. However, due to cost, ethics, and other reasons, the use of experimental data for causal analysis is nearly infeasible in practice. In recent years, using observational data for causal analysis has been demonstrated to be effective in many fields [12] [13]. Causality usually has two representations, namely, causal graph and structural equation. Compared to the structural equation, causal graph is more commonly used in practical applications because it is more intuitive. The most popular method to construct a causal graph from observational data is PC algorithm, which was proposed by Peter Spirtes and Clark Glymour [14].

PC algorithm aims to learn the causal relationship among random variables. Suppose we prepare to learn a causal graph among M random variables. The input is N independent identically distributed samples. Each sample contains M values, which represent the observed values of the M random variables, respectively. PC algorithm will output a directed acyclic graph (DAG) G with M nodes, where each node represents one random variable. [9] treated each time series as one random variable, and the data at each time point as a sample. PC algorithm is based on the assumption that there is no edge between variable A and variable B. Formally, given a variable set S, A is independent of B, denoted as $A \perp B|S$. There are four steps in PC algorithm:

1) Construct a fully connected graph of the M random variables (all nodes are connected).
2) Perform a conditional independence test on each adjacent variables under the significance level α. If a conditional independence exists, the edge between the two variables is removed. In this step, the size of the conditional variable set S increases step by step until there are no more variables that can be added into S.
3) Determine the direction of some edges based on v-structure [15].
4) Determine the direction of the rest of the edges.

D. Random Walk

Random walk is a statistical model that consists of a series of trajectories, each step of which is random. Random walk can be performed in various types of spaces, *e.g.,* a graph, a vector. Typically, the random walk algorithm is as follows.

Step 1. Generate a relationship graph G, where V is the node set and E is the edge set. $e_{ij} \in E$ is set to 1 when node v_i is one of the causes of v_j.

Step 2. Calculate a matrix Q:

1) *Forward step (walk from the result node to the cause node).* Specifically, we assume that the node which is more related to the abnormal node is more likely to be the root cause. Thus $Q_{ij} = R(v_{abnormal}, v_j)$, where $R(v_{abnormal}, v_j)$ is the correlation coefficient between $v_{abnormal}$ and v_j, and $e_{ji} = 1$.
2) *Backward step (walk from cause node to result node).* To avoid the algorithm getting trapped into the node which has low correlation with the abnormal node, random walk enables a step from the cause node to the result node. Formally, if $e_{ji} \in E$ and $e_{ij} \notin E$, $Q_{ji} = \rho R(v_{abnormal}, v_i)$, where ρ is a parameter controlling the impact of the backward step and $\rho \in [0, 1]$.
3) *Self step (stay in the present node).* If the algorithm walks to a node where its neighbor nodes all have lower correlation with the abnormal node, this node likely denotes a root cause. Then the walker should stay in the present node, and $Q_{ii} = \max[0, R(v_{abnormal}, v_i) - \max_{k:e_{ki} \in E} R(v_{abnormal}, v_k)]$.

Step 3. Normalize every row of Q, and get the transition probability matrix $\bar{Q}$ as follows:

$$\bar{Q}_{ij} = \frac{Q_{ij}}{\sum_j Q_{ij}} \tag{1}$$

Step 4. Do random walk over G, and the probability of random walk from v_i to v_j is $\bar{Q}_{ij}$.

After the above four steps, the node which is visited the most frequently is the most likely to denote the root cause.

III. Challenges and Design Overview

A. Challenges

The objective of MicroCause is to robustly localize the root cause metric in a microservice. Motivated by the popular design of failure root cause localization models for computer systems [7]–[9], MicroCause has two main parts, *i.e.,* causal graph construction aiming to learn the relationship of components, and random walk trying to infer root causes. The design of MicroCause faces the three following challenges.

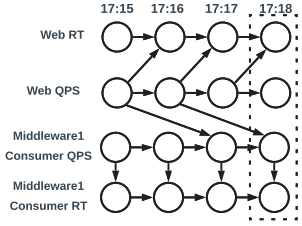


Fig. 5. Causal relationship among a KPI and three metrics. A circle denotes a time point of a KPI/metric, and an arrow represents a causal relationship

iid based causal graph cannot capture propagation delays. Usually, a causal graph learning algorithm assumes that the data is independent and identically distributed (denoted as iid), and thus it cannot capture the propagation delays of different metrics and KPIs. For example, path condition (PC) algorithm, which is a popular causal graph learning algorithm and has been used to learn the causal relationship of APIs [9], [14], treats each time point of metric/KPI as one iid data sample. However, the causal graph ignoring sequential patterns can fail to accurately learn the causal relationships. For instance, as shown in Fig. 5, PC algorithm treats the data points at 17:18 (in the dotted box) as a iid data sample, and assumes that the data sample is independent with the data sample at 17:16 and that at 17:17. Therefore, PC algorithm can only learn that an anomaly of middleware consumer QOS can cause middleware consumer RT to be anomalous. However, when a metric becomes anomalous, it can also lead a KPI or another metric to be anomalous after some period of time due to propagation delays. For example, when Web QPS becomes anomalous at 17:16, it causes Web RT to be anomalous at 17:17 and Middleware consumer QPS to to be anomalous at 17:18. Because PC algorithm ignores the propagation delays among metrics and KPIs, it cannot accurately deduce that the root cause of a Web RT anomaly is an anomalous Web QPS.

Correlation based random walk may not accurately localize root cause. Random walk has been widely used for root cause localization [7]–[9]. It is based on the assumption that, an anomalous metric, which is more correlated with the anomalous KPI, is more likely to be the root cause. However, in our scenario, the monitoring indicators (KPIs and metrics) are heterogeneous. Usually, the monitoring indicators of the same category are more correlated than those of different categories. For example, the anomalous KPI in Fig. 6, Web RT, is more correlated with the middleware consumer RT. However, it is not correlated with the root cause metric, *i.e.,* Web QPS. That is because both Web RT and middleware consumer RT experience large spikes when they become anomalous. On the contrary, a Web QPS more likely suffers from a level shift when an anomaly occurs.

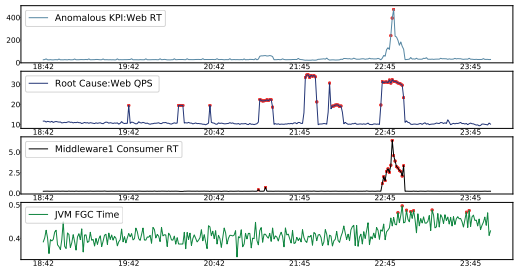


Fig. 6. Monitoring indicators of failure case {Microservice A, Web RT, 22:45-22:55}

B. Design Overview

To solve the above problem, we propose MicroCause, which is a failure root cause localization framework in the microservice through causality inference. In Fig. 7, we demonstrate how the MicroCause works via a failure ticket X {*Microservice A, Web RT, 17:18*}. When the online anomaly is

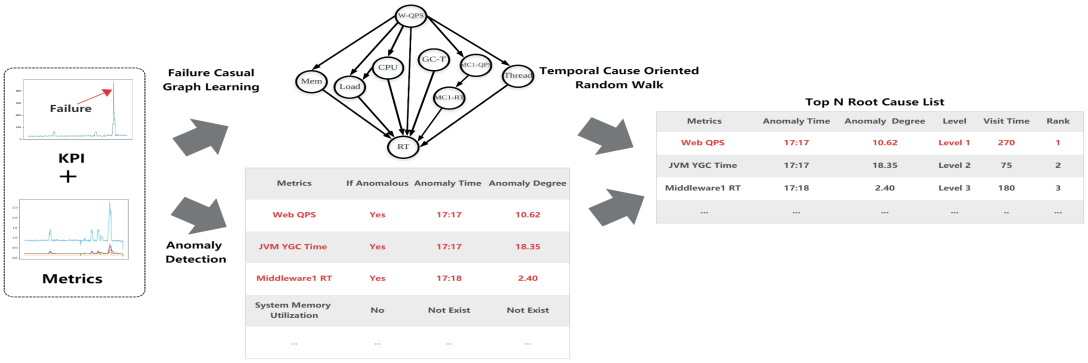


Fig. 7. The overall architecture of MicroCause

detected in KPI (*e.g.* Web RT), MicroCause will be activated. Based on the empirical failure propagation time, the monitoring indicators of the failure microservice, which are from 4 hours before failure time to this moment will be used as the MicroCause 's input. For the failure ticket X, the monitoring indicators of the microservice A, which are from 13:18 to 17:18, will be used.

The input dataset will be used to generate the failure causal graph based on the PCTS algorithm we propose for the time series causal graph learning. In the meantime, the metrics in the input dataset will be checked whether there are anomalies in anomaly detection module. The failure causal graph learning module and the anomaly detection module can be processed in parallel. Then the temporal cause oriented random walk (denoted as TCORW) will give the top N potential root cause of the failure based on the failure causal graph and the anomaly information of the metrics.

IV. Methodology

A. Failure Causal Graph Learning

In the previous works [7]–[9], the dependency graph (*e.g.,* the call graph of APIs) is essential for the failure root cause localization. However, in our scenario, we cannot build the graph via system tools [8] [7], because we investigate the relationship among the monitoring indicators, not the API or virtual machine. Thanks to the research of causal inference [14] [15] [16], we can use the observation data to obtain the relationship between indicators. In [9], they use the PC algorithm [14] to learn causal graph of the APIs. They treated the monitoring data (time series) of APIs as iid data. It means they assumed the independent relationship between $\mathbf{I}_{t\ \tau}$ and $\mathbf{I}_t$, where $\mathbf{I}_t = (I_t^1, \ldots, I_t^N)$ for the time series dataset with N variables. Only the instantaneous causal relationship (*e.g.* from I_t^i to I_t^j) can be learnt based on this assumption. However, the causal relationship between time series often has time lag, *e.g.* the relationship in Fig. 5. *The PC algorithm will fail to report lots of sequential causal relationship between the time series.*

Here we propose PCTS, which is built on the top of the improved PC algorithm [10], to learn the failure causal graph based on the monitoring indicators. The improved PC algorithm [10] has been used in climate science [10], sociology [17]. To the best of our knowledge, it is the first time that this algorithm is adopted in the root cause analysis in the network system.

In improved PC algorithm, given a failure case, such as failure case X, the monitoring indicators of microservice A dataset $\mathbf{I}_t^i, t = 0, \ldots, T, i = 1, \ldots, N$ with N time series , including metrics and KPIs, will be used as input. Here we treat each time point in one time series as one variable I_t^j, which also represents one node in the causal graph. Because there is one sample for the variable $I_t^j, t = 0, \ldots, T$ in the observed time series. We define the max lag as τ_{max}, which means the maximum time lag of the causal impact. Therefore, if we want to find the cause of I_t^i, only the past variables from $\mathbf{I}_t$ to $\mathbf{I}_{t\ \tau_{max}}$ should be taken into consideration. Then we can use the sliding window to construct the independence test samples. We will initialize the preliminary parents $\widehat{\mathcal{P}}(I_t^i) = (\mathbf{I}_{t\ 1}, \ldots, \mathbf{I}_{t\ \tau_{max}})$ for each variable I_t^i. If we assume the instantaneous causality, we can add $\mathbf{I}_t \setminus I_t^i$. Then we conduct conditional independence tests like the PC algorithm, and remove $I_{t\ \tau}^j$ from $\widehat{\mathcal{P}}(I_t^i)$ if the null hypothesis $I_{t\ \tau}^j \perp I_t^i | S$ cannot be rejected at the significance level α_{IPC}, given the conditional set S. $S \in \mathbf{I}_t \setminus I_{t\ \tau}^j$, where $\mathbf{I}_t = (\mathbf{I}_{t\ 1}, \mathbf{I}_{t\ 2}, \ldots, \mathbf{I}_{t\ \tau_{max}})$. This stage converges if no more conditions can be tested and all independent parents with I_t^i are removed from $\widehat{\mathcal{P}}(I_t^i)$. However, the result of the improved PC is a causal graph (denoted as G_C), which each node represents one time point of the monitoring indicators, as shown in the Fig. 5. It can not be used to localize the root cause. Because we need a graph, which each node represents an indicator. Then we can localize the root cause by taking the propagation path.

In PCTS, we improve G_C to the graph where each node represents one monitoring indicator. We assume if there is one edge from $I_{t\ \tau}^j$ to I_t^i in G_C, the final causal graph should have the edge from I^j to I^i. We denote the final failure causal graph as G_{FCG} and the edge set of G_{FCG} is E_{FCG}. It can be formed that if $I_{t\ \tau}^i \in \widehat{\mathcal{P}}(I_t^j), \tau = 0, \cdots, \tau_{max}$, then $e_{ij} \in E_{FCG}$. The Fig. 2 shows G_{FCG} of failure ticket X. There are several metrics impacting the anomalous KPI (Web RT) , such as system CPU utilization, system memory utilization and middleware1 Consumer RT.

B. Anomaly Detection

In MicroCause, we assume that the root cause metrics should become anomalous in some time before failure time. We adopt the SPOT [18], to detect anomaly of the metrics. Because SPOT detects the sudden change in time series via the extreme value theory. It accords with the characteristics of the anomaly metrics in microservice. For each time point in the time series, SPOT will generate a threshold based on the extreme value distributions of the past data. The time point, whose value is higher than the high threshold or lower than the low threshold, will be treated as an anomaly. Based on the physical meaning of the metrics, we list the detection type of the metrics in Table I.

TABLE I
SPOT DETECTION TYPE OF METRICS

Metric type	Detection type
All queries per second, memory-related	High threshold and low threshold
All success rate	Low threshold
Others	High threshold

In addition to detecting the anomalies of indicators, we also evaluate the anomaly degree of metrics based on the SPOT result in this module. Here we define the anomaly degree of the metric i as η^i_{max}. Given a time series of metric $\mathbf{M}^i = M^i_0, M^i_1, \ldots, M^i_T$, the index set of the anomaly point of $\mathbf{M}^i$ is O. We denote the threshold in SPOT for M^i_t is $\phi_{M^i_t}$.

Then the η^i_{max} is calculated as follows:

$$\eta^i_{max} = \max_{k \in O} \frac{|M^i_k - \phi_{M^i_k}|}{\phi_{M^i_k}} \tag{2}$$

For the metrics, which are detected with both high threshold and low threshold, the max value of η^i_{max} of this two conditions will be used. After the anomaly detection module, the anomaly time and anomaly degree of each metric will be output. For example, QPS of Web in failure ticket X becomes anomalous at 17:17 with the anomaly degree 10.62. YGC time of JVM becomes anomalous at 17:17 with the anomaly degree 18.35. The memory utilization of system does not become anomalous.

C. Temporal Cause Oriented Random Walk

In the last modules, we get the anomalous metrics in the anomaly detection module and the anomaly causal graph G_{FCG} in the failure causal graph learning module. Here we denote the node set as V and the edge set as E. Each node v_i represents one monitoring indicator, including KPIs and metrics. And each edge $e_{ij} \in E$ is set to 1 when the indicator (node) v_i is one cause of indicator (node) v_j. Because each of node in G_{FCG} is one time series and different monitoring indicators have different characteristics, we design the temporal cause oriented random walk (TCORW) to rank the potential root causes (metrics) and give the top N root causes.

There are four steps in TCORW:

Step 1: Cause oriented random walk: In this step, we propose cause oriented random walk to find the possible root cause via the causal relationship between the metrics and KPIs. In the traditional random walk (introduced in Section II-D), they use correlation to quantify the relationship between metrics and anomalous KPI. The applied researches have proved "correlation is not equal to causality" [19] [20]. Because the correlation cannot remove the third variable's impact (named as confounder in the causality research). In cause oriented random walk, we calculate Q_{ij} via partial correlation [21], which can remove the effect of confounders We calculate the matrix Q in random walk as follows:

1) Forward step (walk from result indicator to cause indicator):

$$Q_{ij} = R_{pc}(v_{ak}, v_j | Pa(v_{ak}) \setminus v_j, Pa(v_j)) \tag{3}$$

Here the R_{pc} represents the partial correlation, and we take the Pearson correlation as the correlation algorithm of partial correlation. $Pa(v_{ak})$ is the parent node set of v_{ak}. $Pa(v_{ak}) \setminus v_j$ means that v_j is removed from the parent node set of v_{ak}. We take the $Pa(v_{ak}) \setminus v_j$ and $Pa(v_j)$ as the confounders in partial correlation.

2) Backward step (walk from cause indicator to result indicator): We also allow the walker to walk backward to avoid getting trapped in the node which has low causal relationship with v_{ak}. If $e_{ji} \in E$ and $e_{ij} \notin E$, Q_{ji} is set as :

$$Q_{ji} = \rho R_{pc}(v_{ak}, v_i | Pa(v_{ak}) \setminus v_i, Pa(v_i)) \tag{4}$$

where ρ is a parameter controlling the impact of the backwark step and $\rho \in [0, 1]$.

3) Self step (stay in the present node): We encourage the walker stay in the present node if no neighbors have high causal correlation with v_{ak}. We set the Q_{ii} as:

$$\begin{aligned} Q_{ii} &= \max[0, R_{pc}(v_{ak}, v_i \mid Pa(v_{ak}) \setminus v_i, Pa(v_i)) \quad P^{max}_{pc}] \\ P^{max}_{pc} &= \max_{k:e_{ki} \in E} R_{pc}(v_{ak}, v_k \mid Pa(v_{ak}) \setminus v_k, Pa(v_k)) \end{aligned} \tag{5}$$

Then we get the transition probability matrix $\bar{Q}$ as follows by normalizing every row of Q,

$$\bar{Q}_{ij} = \frac{Q_{ij}}{\sum_j Q_{ij}} \tag{6}$$

With $\bar{Q}_{ij}$, we start the random walk from v_{ak}. The walker walks into next node from v_i following the probability vector Q_i. After N_{rw} steps, the walker stops and each node is visited c_i times. For example, for failure ticket X, after 1000 steps, Web QPS are visited the most times, which is 270. Middleware1 RT ranks as the second with 182 times and the JVM YGC time is visited 75 times.

Step 2: Potential root cause score: Beside the casual relationship with the anomalous KPI, we also take the anomaly degree η^i_{max} of the metrics into the consideration to localize the root cause. Because some metrics which impact the anomalous KPI may not be anomalous, such as the memory

utilization of system in Fig. 2. Here we define the potential root cause score of metric i as γ_i. It is calculated as:

$$\gamma_i = \lambda \bar{c}_i + (1-\lambda)\bar{\eta}^i_{max} \tag{7}$$

$\bar{c}_i$ is the normalized visit time c_i. $\bar{\eta}^i_{max}$ is the normalized anomaly degree η^i_{max}. λ controls the contribution of metric's causal relationship with the anomalous KPI and the anomaly degree of the metric.

Step 3: Rank the root causes: For the time series monitoring data, the anomaly time is a non-negligible information to infer the root cause. However some metrics may look like that they become abnormal at the same time, owing to the data aggregation and the instantaneous propagation of the anomaly. Different metrics reflect the performance of different parts in microservice. They have different priorities in the propagation path. For example, when the QPS of Web (upstream) rises in microservice A, the consumer QPS of middleware1 (downstream) will grow up because tasks of the microservice A increase. The CPU of system of microservice A increases likewise. But the consumer QPS of middleware1 cannot impact QPS of Web. Here we classify the metrics into three levels, which is summarized in Table II. Normally the high level metrics (*e.g.* Level 1) impact the low level metrics(*e.g.* Level 2). Therefore, the high level metrics have a higher priority to be considered as the root cause.

In this step, we design an algorithm, which combines the potential root cause score of the metric, the priority of the metric and the anomaly time of the metric to rank all the potential root cause of the failure ticket, as shown in Algorithm 1. We will give the top N results of the RankResultSet returned by Algorithm 1.

Algorithm 1: Rank the root cause

Input: 1 Levels of metrics, 2 γ_i of metric i,
3 anomaly time t_i of metric i
Output: RankResultSet
ResultSet $\leftarrow$ [ ]
for *j=1,2,3* **do**
  $R_j \leftarrow$ rank metrics in Level j by γ_i in descending order.
  ResultSet $\leftarrow$ append the top 2 result in R_j
end
RankResultSet $\leftarrow$ rank ResultSet by t_i in ascending order

For failure ticket X, the top 1 metric in RankResultSet is QPS of Web. Because QPS of Web is at Level 1 in the priority of the metrics. QPS of Web gets abnormal and the young garbage collection count of java virtual machine at the same time at 17:17. The labelled root cause of failure ticket X is QPS of Web.

V. Experiment

In this section, we evaluate MicroCause based on the 86 online failure cases in a microservice based online shopping platform, which has hundreds of millions of users.

A. Experimental Setup

1) Dataset: In the large online shopping platform with thousands of applications, there are hundreds of microservices working online for the operation of the whole system. From Sep. 2019 to Jan. 2020, we collected 86 online failure tickets by monitoring more than 400 microservices' status. Each of these cases is checked by the professional operator to get the root cause. The dataset of each microservice contains 64 metrics and 4 KPIs. The four KPIs is Web RT, Middleware1 Provider RT, Middleware2 Receive RT and Middleware3 Receive RT. All the metrics are listed in Table II, which can be divided into three types according to the its relationship with the microservice.

2) Evaluation Metric: To evaluate MicroCause and the baseline algorithms, we introduce two performance metrics: $AC@k$ and $Avg@k$. These two metrics are the most commonly used metrics to evaluate the rank result of the root cause localization task in recent works [7]–[9]. $AC@k$ represents the probability that top k results given by each algorithm includes the real root causes for all given failure cases. When the k is small, the higher $AC@k$ indicates the algorithm identifies the actual root cause more accurate. It prompts the efficiency of operators' further investigation because of the smaller search space. Given the failure cases set A, $AC@k$ is calculated as follows:

$$AC@k = \frac{1}{|A|}\sum_{a\in A}\frac{\sum_{i<k} R^a[i] \in V^a_{rc}}{min(k, |V^a_{rc}|)} \tag{8}$$

where $R^a[i]$ is the result of rank of all metrics for failure case a. V^a_{rc} is the root cause set for failure case a. $Avg@k$ evaluates the overall performance of each algorithm by computing the average $AC@k$, which defines as:

$$Avg@k = \frac{1}{k}\sum_{1\le j\le k} AC@j \tag{9}$$

3) Baseline Algorithms:

1) Anomaly Time Order: We will rank the metrics based on the start of the anomaly time of each metric.
2) TON18 [8], MonitorRank [7]: [8] used a tracer tool named PreciseTracer and nova APIs to construct the dependency graph between the virtual machines. The dependency graph is used to localize the root cause virtual machine based on random walk (denoted as RW-1). MonitorRank used the Hadoop tools to generate the call graph between APIs, the call graph is used to trace the root cause API based on RW. In our scenario, we can not obtain the call graph or dependency graph based on system tools, therefore we use the PCTS to generate the graph needed in their algorithms.
3) CloudRanger [9]: CloudRanger used the PC algorithm to generate the dependency graph between the APIs in the cloud native system. Then a second-order random

TABLE II
METRICS AND ITS PRIORITY IN FAILURE PROPAGATION. YGC MEANS YOUNG GARBAGE COLLECTION. FGC MEANS FULL GARBAGE COLLECTION. QPS MEANS THE QUERIES PER SECOND.

Metrics Type	Metrics	Priority
Upstream	Web QPS; Middleware1 Provider QPS; Middleware2 Receive QPS; Middleware3 Receive QPS	Level 1
Self	**Java virtual machine (JVM) related**: JVM YGC Count; JVM YGC Time; JVM FGC Count; JVM FGC Time; JVM Max Heap Memory; JVM Used Heap Memory; JVM Usage Heap Memory; JVM Max Nonheap Memory; JVM Used Nonheap Memory; JVM Memory Usage Metaspace Pools; JVM Memory Usage Code Cache Pools; JVM Max Mapped Bufferpool; JVM Used Mapped Bufferpool; JVM Max Direct Bufferpool; JVM Used Direct Bufferpool; JVM Thread Count; JVM Deamon Thread Count; JVM Deadlock Thread Count; JVM Runnable Thread Count; JVM File Descriptor Utilization;	Level 2
	System related: System CPU Utilization; System CPU Steal; System Load1 Utilization; System Load5 Utilization; System Load15 Utilization; System Load1; System Load5; System Load15; System Memory Utilization; System Swap Utilization; System Net In; System Net Out; System Net Retran Utilization; System Net Established; System Disk Utilization; System Disk Read; System Disk Write; System Dish Inode;	Level 2
Downstream	**Queries per second (QPS):** Middleware1 Consumer QPS; Middleware4 Read QPS; Middleware1 Write QPS; Middleware5 Read QPS; Middleware5 Write QPS; Middleware2 Send QPS; Middleware3 Send QPS;	Level 2
	Response time (RT): Middleware1 Consumer RT; Middleware4 Read RT; Middleware4 Write RT; Middleware5 Read RT; Middleware5 Write RT; Middleware2 Send RT; Middleware3 Send RT;	Level 3
	Success rate: Middleware1 Consumer success rate; Middleware4 Read success rate; Middleware4 Write success rate; Middleware5 Read success rate; Middleware5 Write success rate; Middleware2 Send success rate; Middleware3 Send success rate;	Level 3

walk (denoted as RW-2) is used to trace the root cause API.

4) Microscope [22]: Microscope used the PC algorithm to construct the non-communicating service dependencies and used the network connection information to construct the communicating service instance dependencies. Then they used the Pearson correlation coefficient of service level objective between the anomalous front service and the root cause candidates, which are the anomalous parent services of the anomalous service in the dependency graph.

B. Evaluation of the overall performance

The accuracy of MicroCause and the baseline method are listed in Table III, which shows the best performance of all method obtained by MicroCause. Specifically, we outperform TON18 and MonitorRank, the best performance baseline, by 12.0% and 14.7% on $AC@1$ and $AC@2$, respectively. As for $AC@5$, the proposed algorithm achieves at least 29.1% improvement over other methods. In general, MicroCause shows the best performance in this task, and it's $Avg@5$ achieves 69.7%, which is equivalent to more than 40% relative improvement compared with the second place.

TABLE III
RESULT OF MICROCAUSE AND THE BASELINE ALGORITHMS

Method	$AC@1$	$AC@2$	$AC@5$	$Avg@5$
MicroCause	**46.7%**	**62.7%**	**98.7%**	**69.7%**
TON18 [8], MonitorRank [7]	34.7%	48.0%	65.3%	48.2%
CloudRanger [9]	19.0%	32.9%	69.6%	46.8%
Microscope [22]	12.2%	21.9%	29.3%	23.9%
Anomaly Time Order	11.4%	21.5%	43.0%	28.4%

C. Evaluation of Failure Causal Graph Learning

In this part, we will analyze the performance of failure causal graph learning part. PCTS will be compared with the PC algorithm, which is used in the previous root cause localization works [9] [12]. Here we take the significance level of independent test in PC as 0.05, which is a common setting in previous works. For all algorithms, the TCORW will be used to localize the root cause. The results summarized in Table IV show the importance of the proposed PCTS algorithm. Specifically, PCTS outperforms PC by at least 1.8% and 3.7% in $AC@1$ and $AC@3$ this task,respectively. In most commonly used $AC@5$ in practical application, PCTS achieves 5.1% improvement than PC.

TABLE IV
COMPARISON OF THE CAUSAL GRAPH LEARNING ALGORITHMS

Method	AC@1	AC@2	AC@5	Avg@5
MicroCause	**46.7%**	**62.7%**	**98.7%**	**69.7%**
MicroCause w/PC	44.9%	59.0%	93.6%	67.4%

Case study: Fig. 8 shows the failure causal graph of failure ticket X via PC algorithm. Unlike the graph via PCTS (Fig. 2), the graph is divided into two parts. Because there is no path from the Web RT, the random walker can not walk into the real root cause Web QPS. This phenomenon is not rare. In [12], the author declared the isolated subgraphs generated by the PC algorithm based on time series data. It is because we treat the time series as iid data when we use PC algorithm. In fact, we treat the $\mathbf{I}_t = (I_t^1, \ldots, I_t^N)$ and $\mathbf{I}_{t-\tau} = (I_{t-\tau}^1, \ldots, I_{t-\tau}^N)$ as independent samples, which means there is no edge between $I_{t-\tau}^i$ and I_t^j. But it is acknowledged the causal relationship between time series is often along with the time lag. Therefore the time lag causal relationships in practice can not be recognized via PC algorithm, such as the impact from QPS of Web to memory utilization of the system, which is not instantaneous because of the time of the program initialization.

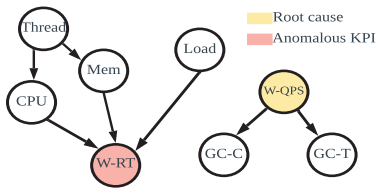


Fig. 8. Failure causal graph via PC algorithm of failure ticket X. *Mem* is the System Memory Utilization. *Cpu* is the System CPU Utilization. *Load* is the System Load1. *W-QPS* is the Web QPS. *GC-C* and *GC-T* are the young garbage collection count of java virtual machine and the young garbage collection time of virtual machine respectively. *W-RT* is the anomalous KPI: Web RT.

D. Evaluation of TCORW

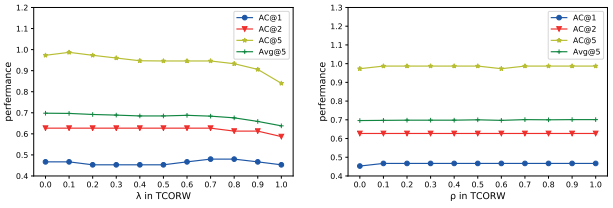
In this part, we analyze the performance of root cause analysis part: TCORW. Here the RW-1 [8] [7] and RW-2 [9] will be used as the baseline algorithms. The results are shown in Table V. All the results are based on PCTS for the root cause analysis. In all methods, the MicroCause achieves the best performance for the task of failure root cause localization and MicroCause w/RW-1 works better than MicroCause w/RW-2 based on all the evaluation metrics. Overall, the performance will be reduced remarkably if we use PC to replace PCTS (e.g., more than 30% performance drop in $AC@5$), which proves the effectiveness of the proposed TCORW algorithm.

TABLE V
COMPARISON OF TCORW WITH THE BASELINE METHOD

Method	AC@1	AC@2	AC@5	Avg@5
MicroCause	**46.7%**	**62.7%**	**98.7%**	**69.7%**
MicroCause w/RW-1	34.7%	48.0%	65.3%	48.2%
MicroCause w/RW-2	29.3%	46.7%	62.7%	46.3%

E. Evaluation of The Parameters in MicroCause

In this part, we will analyze the impact of parameters λ and ρ in TCORW on the root cause localization performance of MicroCause. Here we change the λ from 0 to 1 and the results can be found in Fig. 9(a). The best performance of MicroCause is achieved when the λ is equal to 0.1. When the λ changes from 0.1 to 0.7, $AC@1$, $AC@2$, $Avg@5$ almost do not change, and $AC@5$ remains above 90.0%. From this point, MicroCause is robust to the λ in TCORW, when λ is less than 0.8.



(a) (b)

Fig. 9. The performance of MicroCause based on different λ and ρ in TCORW

Here we change the ρ from 0 to 1 and the results are shown in Fig. 9(b). The results show the performance of the MicroCause is not influenced by ρ, when the ρ is greater than 0. MicroCause is not sensitive to ρ in TCORW, based on the experiments with the online shopping platform dataset.

VI. RELATED WORK

Because it is of vital importance to quickly localize the root causes of computer system failures for assuring the quality of users' experience, a large number of methods have been proposed for this purpose. However, these works all aim to localize the root cause cross-microservices. Normally there are clear calling relationships between the microservices. The system tools or network connection information can be used to learn the communicating [8] [7] [22]. For the non-communicating relationship, [9] [22] use the one service level objective metric to represent each service and learn the non-communicating relationship between the microservices via the causality method, such as PC. But there is no clear communicating relationship in a microservice. We have to learn the dependence relationship between different metrics to localize the root cause in a microservice.

As listed in Table VI, root cause localization for system failures typically include two parts: learning the relationship of components to construct dependency graphs, and inferring root causes through random walk.

TABLE VI
THE RELATIONSHIP LEARNING METHODS AND ROOT CAUSE INFERENCE METHODS USED IN PREVIOUS WORKS FOR FAILURE ROOT CAUSE LOCALIZATION

Type	Model	Relationship learning	Root cause inference
Cross-microservice	TON18 [8]	OpenStack APIs	Random walk
	MonitorRank [7]	Hadoop tools	Random walk
	CloudRanger [9]	PC	Second order random walk
	Microscope [22]	PC	Pearson correlation
Inter-microservice	MicroCause	PCTS	TCORW

A. Relationship Learning

There are mainly two types of relationship learning methods used in previous failure root cause localization models: system tool based methods and data analysis based methods as follows:

- **System tool based methods**: Kim *et al.* proposed MonitorRank [7] to localize the root cause API of failures in service-oriented architectures. MonitorRank generated a call graph of the studied APIs, which was easily generated by those batch processing systems (*e.g.,* Hadoop), to learn APIs' relationship. Besides, Weng *et al.* [8] inferred the dependencies of virtual machines in a cloud platform to localize failure root causes at the virtual machine level. The APIs of OpenStack were applied to obtain physical relationships, and a popular trace analysis tool, PreciseTracer, was used to capture call relationships.
- **Data analysis based methods**: In [23], Chen et al. adopted a Bayesian network as a diagnostic tool to infer the relationship between alerting signals and outages in cloud service systems. Moreover, in [24], the AutoRegressive eXogenous (ARX) model was used to learn the invariant relationship of monitoring indicators. Based on historical alarms, [9] applied the PC algorithm to learn the causal relationships of service APIs.

In our scenario, however, we cannot obtain the relationship of KPIs and metrics within a microservice through system tools. Besides, the previously proposed methods, *e.g.,* the Bayesian network, ARX, and the PC algorithm, which cannot capture the sequential relationship of KPIs and metrics, do not achieve satisfactory performance in our scenario as demonstrated in Section V-B.

B. Root Cause Inference

Motivated by the successful applications of random walk in biological research (*e.g.,*, inferring the source of scent), [7] introduced random walk on the generated calling graph to trace the root cause APIs of failures. Random walk was also used to analyze the root cause virtual machines of failures in cloud platforms [8]. To improve the performance of random walk, [9] proposed a second-order random walk, which calculates the transition probability between two edges rather than that of two nodes. However, neither the native random walk method nor the second-order one achieves good performance in our scenario, namely, they are more likely to generate false root causes. This is because they neither fully utilize the temporal information of KPI anomaly and metric anomalies, nor leverage the priority information of metrics, which has been demonstrated in Section V-B.

VII. Conclusion

Microservice-based architectures are becoming more common in current large-scale Internet systems. Quick failure root cause localization can improve quality of service and reduce loss in efficiency and revenue. In this paper, we first propose to investigate the failure root cause in a microservice, which is an important step in the failure localization of the whole system. We design a framework, MicroCause, to localize the failure root cause in a microservice, which achieves high performance in the experiments based on 86 the online failure tickets. In MicroCause, we design PCTS, which can learn the causal graph of monitoring indicators. We believe this method can be used in other time series related root cause localization problems.

VIII. Acknowledgement

We thank the anonymous reviewers for their valuable feedbacks. We thank Juexing Liao and Ping Liu for their helpful suggestions and proofreading. This work has been supported by the National Key R&D Program of China under grant No. 2019YFB1802504, the National Natural Science Foundation of China under Grant No. 61902200, the Fundamental Research Funds for the Central Universities under grant No. 63191424, the China Postdoctoral Science Foundation under Grant No. 2019M651015, and the Beijing National Research Center for Information Science and Technology (BNRist).

References

[1] https://www.itproportal.com/features/microservices-architecture-is-helping-organisations-in-achieving-new-heights, accessed January 16, 2020.

[2] https://tech.sina.com.cn/it/2019-03-03/doc-ihsxncvf9295268.shtml, accessed February 1, 2020.

[3] https://www.epochtimes.com/gb/19/2/12/n11038862.html, accessed February 1, 2020.

[4] C. Heinze-Deml, M. H. Maathuis, and N. Meinshausen, "Causal structure learning," *Annual Review of Statistics and Its Application*, vol. 5, pp. 371–391, 2018.

[5] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, D. Liu, Q. Xiang, and C. He, "Latent error prediction and fault localization for microservice applications by learning from system trace logs," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 683–694.

[6] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Algorithms and Architectures for Parallel Processing*, J. Vaidya and J. Li, Eds. Cham: Springer International Publishing, 2018, pp. 560–572.

[7] M. Kim, R. Sumbaly, and S. Shah, "Root cause detection in a service-oriented architecture," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 93–104, 2013.

[8] J. Weng, J. H. Wang, J. Yang, and Y. Yang, "Root cause analysis of anomalies of multitier services in public clouds," *IEEE/ACM Transactions on Networking*, vol. 26, no. 4, pp. 1646–1659, 2018.

[9] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: root cause identification for cloud native systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2018, pp. 492–502.

[10] J. Runge, P. Nowack, M. Kretschmer, S. Flaxman, and D. Sejdinovic, "Detecting and quantifying causal associations in large nonlinear time series datasets," *Science Advances*, vol. 5, no. 11, p. eaau4996, 2019.

[11] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang *et al.*, "Prefix: Switch failure prediction in datacenter networks," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 2, no. 1, pp. 1–29, 2018.

[12] P. Chen, Y. Qi, P. Zheng, and D. Hou, "Causeinfer: automatic and distributed performance diagnosis with hierarchical causality graph in large distributed systems," in *IEEE INFOCOM 2014-IEEE Conference on Computer Communications*. IEEE, 2014, pp. 1887–1895.

[13] E. K. Kao, "Causal inference under network interference: A framework for experiments on social networks," *arXiv preprint arXiv:1708.08522*, 2017.

[14] P. Spirtes, C. N. Glymour, R. Scheines, and D. Heckerman, *Causation, prediction, and search*. MIT press, 2000.

[15] L. G. Neuberg, "Causality: models, reasoning, and inference, by judea pearl, cambridge university press, 2000," *Econometric Theory*, vol. 19, no. 4, pp. 675–685, 2003.

[16] C. Heinze-Deml, M. H. Maathuis, and N. Meinshausen, "Causal Structure Learning," *arXiv:1706.09141 [stat]*, Jun. 2017, arXiv: 1706.09141. [Online]. Available: http://arxiv.org/abs/1706.09141

[17] A. Bovet and H. A. Makse, "Influence of fake news in twitter during the 2016 us presidential election," *Nature communications*, vol. 10, no. 1, pp. 1–14, 2019.

[18] A. Siffer, P.-A. Fouque, A. Termier, and C. Largouet, "Anomaly detection in streams with extreme value theory," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2017, pp. 1067–1075.

[19] Y. Meng, S. Zhang, Z. Ye, B. Wang, Z. Wang, Y. Sun, Q. Liu, S. Yang, and D. Pei, "Causal analysis of the unsatisfying experience in realtime mobile multiplayer games in the wild," in *2019 IEEE International Conference on Multimedia and Expo (ICME)*, July 2019, pp. 1870–1875.

[20] R. Kold-Christensen and M. Johannsen, "Methylglyoxal metabolism and aging-related disease: Moving from correlation toward causation," *Trends in Endocrinology & Metabolism*, 2019.

[21] K. Baba, R. Shibata, and M. Sibuya, "Partial correlation and conditional correlation as measures of conditional independence," *Australian & New Zealand Journal of Statistics*, vol. 46, no. 4, pp. 657–664, 2004.

[22] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing*, C. Pahl, M. Vukovic, J. Yin, and Q. Yu, Eds. Cham: Springer International Publishing, 2018, pp. 3–20.

[23] Y. Chen, X. Yang, Q. Lin, H. Zhang, F. Gao, Z. Xu, Y. Dang, D. Zhang, H. Dong, Y. Xu *et al.*, "Outage prediction and diagnosis for cloud service systems," pp. 2659–2665, 2019.

[24] W. Cheng, K. Zhang, H. Chen, G. Jiang, Z. Chen, and W. Wang, "Ranking causal anomalies via temporal and dynamical analysis on vanishing correlations," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016, pp. 805–814.